# Dependability modelling of a fault tolerant duplex system using AADL and GSPNs

Ana-Elena E. Rugina, Karama Kanoun, Mohamed Kaâniche, Jérémie Guiochet

▶ To cite this version:

Ana-Elena E. Rugina, Karama Kanoun, Mohamed Kaâniche, Jérémie Guiochet. Dependability modelling of a fault tolerant duplex system using AADL and GSPNs. [Research Report] 05315, LAAS-CNRS. 2005. hal-01295346

HAL Id: hal-01295346
https://hal.science/hal-01295346v1

Submitted on 30 Mar 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Dependability modelling of a fault tolerant duplex system using AADL and GSPNs

**(first draft)**

**Prepared by:** Ana Elena Rugina, Karama Kanoun, Mohamed Kaâniche, Jérémie Guiochet (CNRS-LAAS)

**Release type:** ASSERT internal report

LAAS Research Report no. 05315

# Foreword

This research report is intended to explore the possibilities of deriving Generalised Stochastic Petri Nets (GSPNs) dependability models from AADL dependability models in order to estimate dependability measures for computer-based systems.

The AADL dependability models are composed of i) AADL architecture models including the various components of the system and ii) their associated AADL error models, as described in Section 3 of this report. Our reference document for describing error models is the AADL Error Model Annex v0.8.

The main difficulties when building AADL dependability models are due to interactions between the system components. These interactions induce dependencies both at the AADL architectural model level and at the error model level (i.e., between the architectural models of the system components and between the error models associated to components - see Section 2 of this report).

The AADL dependability models are then transformed into GSPN dependability models. One of the advantages of using GSPNs lies in the existence of GSPN-based tools for system dependability measures evaluation (e.g., availability, safety, maintainability).

AADL dependability model construction and transformation into GSPN models are shown on a concrete example in this report. We have selected a fault-tolerant dynamically reconfigurable duplex system as a case study because it is simple enough to allow us to build the whole dependability model in fifty pages and complex enough to allow us to explore several kinds of dependencies.

The work presented in this report is partially supported by the ASSERT project (**A**utomated proof based **S**ystem and **S**oftware **E**ngineering for **R**eal-**T**ime Applications), Project Number: IST 004033.

# Table of contents

1 Introduction ..... 5
2 System description ..... 7
2.1 Hardware – Software dependencies (HW-SW) ..... 7
2.2 Hardware – Hardware dependency (HW-HW) ..... 8
2.3 Software – Software dependency (SW-SW) ..... 8
3 AADL architecture model of the duplex system ..... 8
4 Error model construction and transformation to GSPN ..... 9
4.1 Second phase: Hardware and software components in isolation ..... 13
4.1.1 Hardware component: hypotheses ..... 13
4.1.2 Hardware component: AADL error model ..... 13
4.1.3 Hardware component: GSPN ..... 15
4.1.4 Software component: hypotheses ..... 16
4.1.5 Software component: AADL error model ..... 16
4.1.6 Software component: GSPN ..... 18
4.2 Third phase: Hardware and software components (HW-SW dependencies) ..... 19
4.2.1 Hardware component: AADL error model (HW-SW dependencies) ..... 20
4.2.2 Hardware component: GSPN (HW-SW dependencies) ..... 22
4.2.3 Software component: AADL error model (HW-SW dependencies) ..... 23
4.2.4 Software component: GSPN (HW-SW dependencies) ..... 25
4.2.5 Software component: additional error properties on ports ..... 26
4.2.6 Merging propagations for global HW - SW GSPN ..... 28
4.3 Fourth phase: Hardware components (HW-HW maintenance & repair strategy) ..... 32
4.3.1 Hardware component: AADL error model (HW-HW dependency) ..... 32
4.3.2 Hardware component: GSPN (HW-HW dependency) ..... 34
4.3.3 Repairman component: AADL error model ..... 34
4.3.4 Repairman component: GSPN ..... 35
4.3.5 Global Hardware – Repairman GSPN (HW-HW dependency) ..... 36
4.4 Fifth phase: Software components (fault tolerance dependency) ..... 36
4.4.1 Software component: AADL error model (SW-SW dependency) ..... 39
4.4.2 Software component: GSPN (SW-SW dependency) ..... 41
5 Conclusion ..... 49
Annex 1: Proposals for the AADL Error Model Annex ..... 51

1 Occurrence properties .....51
1.1 Current Occurrence properties.....51
1.2 Our proposal .....51
2 Link between the mode model and the error model .....52
2.1 Current specifications concerning modes .....52
5.1.1 Influences of the error model on the mode model .....52
5.1.2 Influences of the mode model on the error model .....52
2.2 Our proposal .....53
3 Vote_In and Vote_Out properties.....57
3.1 Current Vote_In and Vote_Out properties .....57
3.2 Our proposal .....57
4 Inheritance and refinements .....58
4.1 Current inheritance and refinement mechanisms .....58
4.2 Our proposal .....58
5.1.3 Error model type inheritance .....58
5.1.4 Error model implementation inheritance.....60
Annex 2: Duplex system case study – AADL architecture .....64
Annex 3: Duplex system case study – AADL Error Model .....69

# 1 Introduction

In order to remain competitive with regards to costs and delays, the real-time embedded systems industry must solve crucial problems related to the increasing complexity of new-generation systems. These problems are addressed in the FP6 European Integrated Project ASSERT (***A****utomated proof based* ***S****ystem and* ***S****oftware* ***E****ngineering for* ***R****eal-****T****ime applications*) coordinated by the European Space Agency [Conquet & David 2005]. This project aims mainly at i) identifying reference architectures for different system families, ii) replacing the classical system engineering approach by a proof-based method and iii) demonstrating the validity of the newly introduced concepts on real industrial case studies. In this context, high guarantees on the dependability properties are required at lower costs. Mature dependability-oriented analytical modelling techniques do exist ([Bondavalli et al. 1999], [Kanoun & Borrel 2000], [Betous-Almeida & Kanoun 2004]). A state-of-the-art related to dependability modelling and evaluation is presented in the ASSERT deliverable D-32(345)-1 [Arlat et al. 2005]. These techniques are mainly based on the use of Petri nets and Markov chains. Existing tools support the analysis of such analytical models. However, analytical modelling techniques require substantial amount of training to be used effectively. On the other hand, description languages such as UML (Unified Modelling Language) and AADL (Architecture Analysis and Design Language) [SAE-AS5506 2004] have emerged. They are more and more extensively used by industry. In the context of the ASSERT project, we aim at developing a modelling framework allowing the automatic generation of dependability-oriented analytical models from high-level AADL models. This approach is meant to hide the complexity of analytical models to the end-user and, in this way, to facilitate the evaluation of dependability measures, such as reliability, availability and maintainability.

Similarly to all dependability modelling and analysis approaches, our approach requires various types of information describing the system to be available. The system description must contain i) its structure, ii) its functional behaviour and iii) its behaviour in the presence of faults. Interactions between components of the system must be analysed at this stage, as such interactions induce dependencies between components and consequently between their models.

An overview of our approach, which is composed of four main steps, is illustrated in Figure 1. Its steps are described afterwards.

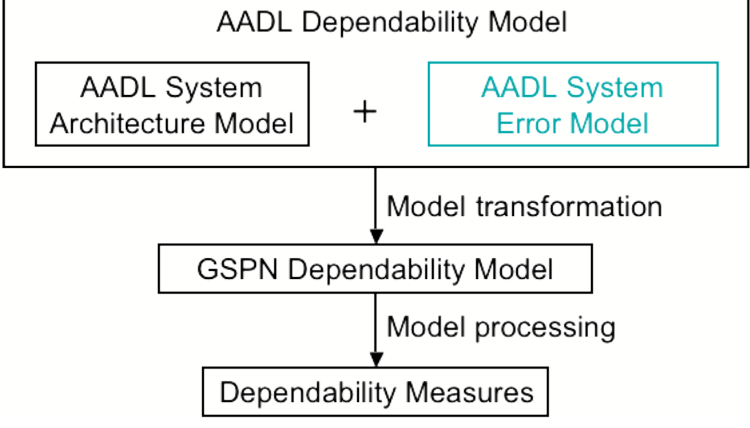



**Figure 1: General approach**

- The ***first step*** is devoted to the modelling of the system architecture in AADL (i.e., its structure in terms of components and operational modes of these components). Sometimes the AADL system architecture is already available, as it may have been already built for other analyses.
- The ***second step*** concerns the description of the system behaviour in the presence of faults through AADL error models associated to components of the AADL architecture model. The set of error models associated to components of the architecture forms the system error model.
- The ***third step*** aims at constructing a global analytical dependability model that can be processed by existing tools. The information that is necessary to the generation of an analytical dependability model is extracted from the AADL dependability model. The global analytical dependability model is generated in the form of a Generalised Stochastic Petri Net (GSPN) by applying model transformation rules. Existing dependability analysis tools can then process the GSPN. It is worth stressing that in the case of an isolated system or in the case of a set of systems considered to be independent, the AADL to GSPN transformation is rather straightforward. However, the transformation becomes complex in the case of realistic systems formed of dependent components as shown further in this report.
- The ***fourth step*** is devoted to the GSPN model processing that aims at obtaining dependability measures. This step is not detailed in this report as it is supposed to be completely automated by using existing analytical model processing tools.

During the second step, the architecture model might need some adjustments to support dependencies related, for example, to the maintenance strategy. This topic will be detailed in section 4 of this report.

When the system to be analysed has many dependencies between its components, we suggest that the second step be incremental in order to master the complexity and the evolution of the system error model. More concretely, in a first phase we propose to introduce the error models to be associated to components, representing their behaviour in the presence of their own faults and repair events only. Error propagations and failure or repair dependencies from or to their environment are not considered. In this case we say that the components are modelled as if they were *isolated* from their environment. Then, we propose to introduce dependencies in an incremental manner. In this way, the final model represents the behaviour of each component not only in the presence of its own faults and repair events, but also in its environment, i.e., faults and repair events in components with which it interacts.

If the user prefers to validate the model progressively, the third step of our approach can also be incremental; as it is possible to enrich the global analytical model each time the second step is iterated.

Our modelling and transformation approach is illustrated in this report on a concrete case study, a fault-tolerant dynamically reconfigurable duplex system. For clarity reasons, the second and the third steps are multi-phased. For this case study, the model transformations were performed manually. Nevertheless, a preliminary set of transformation rules are identified in order to allow automatic tool-based model transformation.

The rest of this research report is organised as follows. Section 2 describes the duplex system. Section 3 describes the first step of our approach applied to the duplex system case study: the AADL architecture model. The construction of the architecture model is based on the specifications described in section 2. Section 4 presents the second and third steps of our

approach, applied to the duplex system case study. It details the successive error modelling phases, based on the description of dependencies, and proposes an incremental AADL to GSPN transformation. Section 5 concludes this report. Annex 1 contains some proposals for the evolution of the AADL Error Model Annex. Annex 2 contains the complete textual AADL architecture model while Annex 3 contains the error models used in the AADL dependability model.

# 2 System description

The duplex system is composed of two (hardware) computers and two software replicas, each software replica on a computer. At each moment of time, one of the replicas has the role Primary, while the other one has the role Backup. Both the Primary and the Backup are active and are capable of delivering the service but only the Primary's outputs are active. The global fault tolerance policy of the duplex system is specified as follows. The two replicas switch roles when the one that is labelled Primary fails or when it stops because of a hardware failure. Then the Backup becomes Primary and its outputs become active. Conversely, the former Primary becomes Backup and is restarted either immediately (if the software failed) or after repair of hardware (if the hardware failed).

The various interactions between the system's components induce dependencies between them. These dependencies have to be taken into account in the dependability modelling. The following sub sections present respectively the structural dependencies between the hardware and the software, the maintenance & repair dependency between hardware components and the fault tolerance dependency between the software components.

## 2.1 Hardware – Software dependencies (HW-SW)

The HW-SW structural dependencies are unidirectional, as we suppose that the hardware faults can propagate and influence the software running on top of it but the software faults cannot propagate to the hardware that supports it. In this case study, we distinguish between permanent and temporary faults because the effects of these types of faults are different.

- Temporary faults in the hardware are more likely to occur than permanent faults. They do not always require hardware maintenance but they can cause error transmissions to the software on top of the hardware. They may also disappear spontaneously. Non-detected permanent faults can also be transmitted to the software.
- Detected permanent faults in the hardware cause hardware failures that require repairing the hardware. Consequently, the hardware cannot support the software running on top.

At this point, we can state that there are two dependencies between the hardware and the software: i) the first one is related to the error transmission from the hardware to the software, in case of temporary faults in the hardware, and ii) the second one is related to the stop / restart of the software in case of hardware failure.

## 2.2 Hardware – Hardware dependency (HW-HW)

We consider that the two hardware components share one repairman. So, the repairman is not simultaneously available for the two components. The dependency between the two hardware components is directly determined by the repair procedure. If the two components fail one after the other, then the second failed component has to wait until the repairman has finished repairing the first component.

The HW-HW maintenance & repair dependency is bidirectional. Each of the two hardware components may fail and be forced to wait for the repairman to be released.

## 2.3 Software – Software dependency (SW-SW)

The software reconfiguration lies in the Primary / Backup role management. The fault tolerance policy is as follows.

- If the Backup fails but the Primary is error free, the Backup is restarted.
- If the Primary fails but the Backup is error free, the two software replicas switch roles. Then, the new Backup is restarted.
- If both replicas fail one after the other, the first one restarted takes the role Primary.

The SW-SW fault tolerance dependency is bidirectional, as any of the two software replicas can have the role Primary or Backup. Consequently, each replica may be sender or receiver of information at different moments of time.

# *3 AADL architecture model of the duplex system*

The chosen representation of the fault tolerant duplex system, shown in Figure 2, is a very high-level one. Therefore, we only use AADL system components in this model. One system component named "simple_duplex" models the whole duplex system. This AADL component contains two AADL system components:

- System1 is an instance of the implementation sub_system.basic_primary. This sub system contains an instance of computer.basic (HW) that models the hardware component, and an instance of software.basic_primary (SW1) that models a software replica whose initial operational mode is Primary.
- System2 is an instance of the implementation sub_system.basic_backup. This sub system contains an instance of computer.basic (HW) that models the hardware component (just like for System1) and an instance of software.basic_backup (SW2) that models a software replica whose initial operational mode is Backup (in opposition with the initial mode of System1).

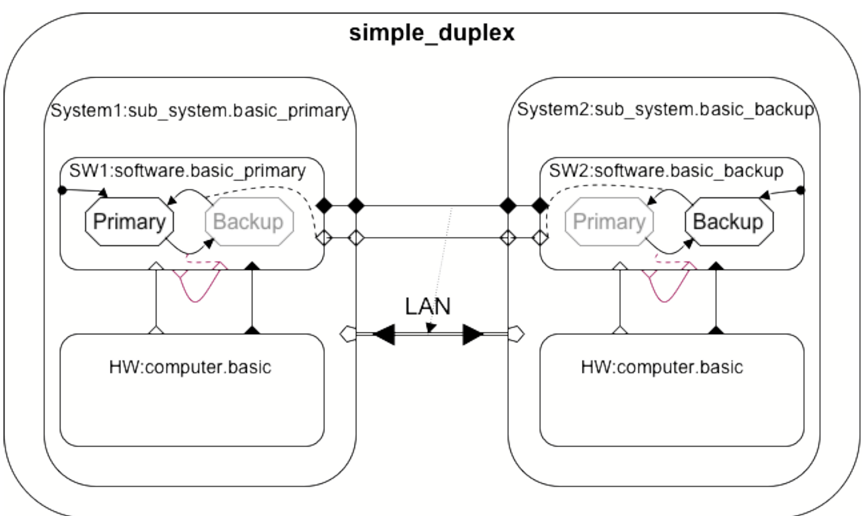



**Figure 2: AADL architecture model for the duplex system**

Auto-connections are highlighted in Figure 2 as their use is a necessary modelling trick: they allow triggering mode transitions by events that are internal to the component. Each software component has two operational modes: Primary and Backup. One of them is initially in Primary mode while the other one is in Backup mode. The switching policy modelled here is as follows. A software component has to move from mode Primary to mode Backup when it fails. Also, it has to move from mode Backup to mode Primary when the other software replica fails.

The internal failure event is sent out by the out event port and received back through the in port that triggers the mode transition. We model here auto-testable software components that are aware of their failures and that are able to switch modes accordingly.

Bidirectional data and event connections are made between software components. They model respectively the checkpoint data sent from the Primary to the Backup and the events that are sent in case of failure of the Primary. Connections are bidirectional as, at different moments of time, each replica may be Primary or Backup, so each replica has to be able to send data and events to the other one. However, at a given moment of time, the event flow is unidirectional only (from the Primary to the Backup). For the sake of completeness, connections between software components are bound to a bus named LAN.

Unidirectional data and event connections are made between the hardware and software components. They model respectively data used by the software from the memory of the computer and hardware exceptions that may be sent to the software.

## 4 Error model construction and transformation to GSPN

This section presents the error model construction phases together with the corresponding phases of the incremental AADL to GSPN transformation. The error model construction phases are enumerated hereafter.

- ***First phase:*** According to the system's specifications, we decide to which architectural components we have to associate error models. In the case of the duplex system, we decided to take into account faults and repair events in the hardware and software components. In this case study we do not take into account faults in the connections and faults in the LAN bus. Consequently, error models are

to be associated to AADL components that model the hardware and the software. The maintenance & repair strategy might require the introduction of artificial components in the architecture. In the case of the duplex system, the two computers share one repairman. The repairman is modelled by an AADL system component. At architecture level, this component can be seen as an empty box as it does not have any particular role in the functional architecture. This component is necessary at the error model level, in order to describe the maintenance & repair dependency between the two computers. In fact, this repairman is a shared resource, as both computers cannot be repaired simultaneously. Bidirectional event connections are made between the hardware components and the repairman, as both the repairman and the hardware that needs to be repaired must be able to send events one to each other.

- ***Second phase:*** We start the construction of error models as if the components to which they are associated were isolated. No AADL propagations are defined at this stage. For the particular case study of the duplex system, we model the behaviour of the *hardware* components in isolation, then the behaviour of the *software* components in isolation. The error model of the *repairman* is left empty at this stage, as it makes no sense without any dependency. This second phase is detailed in section 4.1.
- ***The following phases:*** we progressively add inter-component dependencies. In the case of the duplex system, they are as follows.
  - ***Third phase:*** we introduce in the error model of hardware and software components the *structural dependencies between the hardware and the software (HW-SW dependencies)*. This phase is described in section 4.2.
  - ***Fourth phase:*** we introduce in the error models of hardware and repairman components *the maintenance & repair dependency between the two hardware components (HW-HW dependency)*. This phase is detailed in section 4.3.
  - ***Fifth phase:*** we introduce in the error models of the software components *the fault tolerance dependency between software replicas (SW-SW dependency).* This phase is described in section 4.4.

It is worth mentioning that the phase order may be changed according to the context of the targeted analysis. Generally, maintenance & repair dependencies, as well as fault tolerance dependencies are modelled at the end, as one important aim of the dependability evaluation is to find the best-suited maintenance and fault tolerance policies for a system.

At the end of this incremental process, the error models associated to hardware, software and repairman components contain information (propagation declarations and transitions) describing all dependencies between them. Also, Vote properties will be associated to the AADL architecture model to describe how propagations are filtered or masked. For every component of the global system that has an associated error model, an instance of this error model together with Vote properties, if any, is included in the AADL system error model[1].

---

[1] Note that no Model_Hierarchy expression was defined for any error model. Consequently, all error models are included in the global system error model.

Figure 3 shows our modelling approach applied to the duplex system. The upper part of the figure represents the AADL dependability model while the lower part represents the GSPN dependability model, obtained by model transformation.

The AADL dependability model is formed of the AADL system error model and the AADL architecture model. The five complete error model instances, shown in blue, each modelling the behaviour of one component in the presence of its own faults and of its environment, form the AADL system error model, which is also represented in blue in Figure 1. The various dependencies discussed above are explicitly represented in Figure 3 using different colours. We represented in yellow the error model parts corresponding to the HW-SW dependency, in green the error model parts corresponding to the HW-HW dependency and in pink the error model parts corresponding to the SW-SW dependency. The rest of the information in the AADL dependability model in Figure 3 corresponds to the AADL architecture, part of the model necessary to the AADL to GSPN model transformation. It is worth noting that some architectural details are abstracted away in the AADL dependability model, as they are not used in the model transformation. It is, for example, the case of the LAN bus, which does not have an associated error model, so it is abstracted away from the AADL dependability model. It is worth stressing that information concerning a particular dependency between two components is included in the error models attached to both components.

The GSPN dependability model obtained by model transformation is formed of several GSPN blocks. A block is a sub net describing either the component's behaviour in the presence of its own faults and repair events (component net), or a dependency (dependency net). GSPN dependency blocks are obtained from information concerning a particular dependency existing in two dependent error models. The global GSPN contains one block for the behaviour of each component in the presence of its own faults and repair events, and one block for each dependency between components. In Figure 3, the colours of the GSPN dependency blocks match the colours of the corresponding error model parts, i.e., we represented in yellow the block corresponding to the HW-SW dependency, in green the block corresponding to the HW-HW dependency and in pink the block corresponding to the SW-SW dependency. The GSPN blocks are only connected through arcs. The arrows that link the GSPN blocks in Figure 3 represent the directions of dependencies. For example, the HW-SW dependency is unidirectional. Directions of dependencies in the GSPN are the same as the directions of the connections in the AADL architecture model.

It is worth noting that, in Figure 3, both the error models and the GSPN blocks are named. Error model names have the form *Type.Implementation*. The type name corresponds to the component type, while the name of the implementation corresponds to the integrated dependencies. GSPN block names have the form *M.identifier*. M stands for "model". The identifier is either the name of an error model type and in this case the block is a component net, or the name of a dependency and in this case the block is a dependency net.

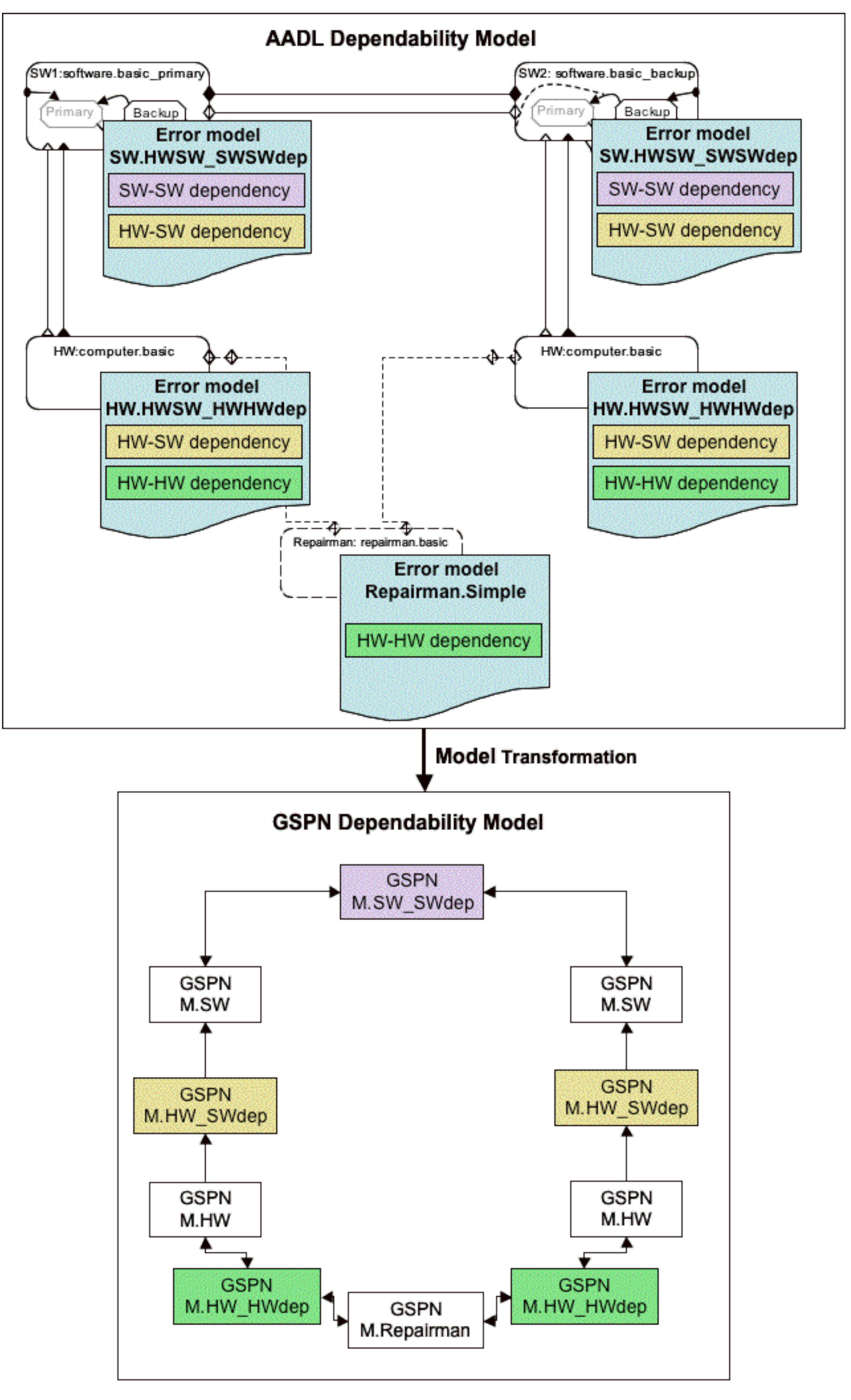



**Figure 3: Global modelling approach applied to the duplex system**

## 4.1 Second phase: Hardware and software components in isolation

### 4.1.1 Hardware component: hypotheses

- Initially the component is in *Error_Free* state
- Faults are activated with a specified rate
- The fault is permanent with a given probability (*ph*) and temporary with the complementary probability (*1-ph*)
- Errors caused by temporary faults disappear after a short period of time
- Errors caused by permanent faults are either detected with a given probability (*dh*), or non-detected (probability *1-dh*). The error detection takes some time. In both cases the component moves to a Failed state. However, if the error is detected, the hardware component is repaired. If not, the failure is perceived after a certain amount of time. Then the component is repaired. Repair takes some time.

### 4.1.2 Hardware component: AADL error model

*Error models* are divided into two parts: the *error model type* - declaring error events, states and propagations - and the *error model implementation* - declaring transitions between states, triggered by events, and properties. The error model to be associated to isolated hardware components is given in Error Model 1. At this stage, error propagations are not considered.

**Error Model 1: AADL Error Model for isolated hardware components**

```
Error Model Type [HW]

error model HW
features
-- events
HW_Fault, HW_Perm_Fault, HW_Temp_Fault,
HW_Error_Detection_Action,HW_Failure_Perceived, HW_Perm_Fault_Detec
ted, HW_Perm_Fault_Non_Detected, HW_Repair_Temp, HW_Repair_Perm:
error event;
-- states
HW_Error_Free: initial error state;
HW_Activation_Fault, HW_Temporary_Erroneous_State,
HW_Permanent_Erroneous_State, HW_End_of_Error_Detection_Action,
HW_Error_Non_Detected, HW_In_Repair: error state;
end HW;
```

**Error Model Implementation [HW.Simple]**

```
error model implementation HW.Simple
transitions
HW_Error_Free-[HW_Fault] -> HW_Activation_Fault;
HW_Activation_Fault-[HW_Temp_Fault] ->
HW_Temporary_Erroneous_State;
HW_Activation_Fault-[HW_Perm_Fault] ->
HW_Permanent_Erroneous_State;
HW_Temporary_Erroneous_State-[HW_Repair_Temporary] ->
HW_Error_Free;
HW_Permanent_Erroneous_State-[HW_Error_Detection_Action] ->
HW_End_of_Error_Detection_Action;
HW_End_of_Error_Detection_Action-[HW_Permanent_Fault_Detected] ->
HW_In_Repair;
HW_End_of_Error_Detection_Action-[HW_Permanent_Fault_Non_Detected]
-> HW_Error_Non_Detected;
HW_Error_Non_Detected-[HW_Failure_Perceived]-> HW_In_Repair;
HW_In_Repair-[HW_Repair_Permanent] -> HW_Error_Free;
properties
-- a fault occurs following a poisson distribution
Occurrence => poisson 10e-2 applies to HW_Fault;
-- a fixed probability is associated with occurrence
-- of Temporary and Permanent Faults
Occurrence => fixed 0.98 applies to HW_Temp_Fault;
Occurrence => fixed 0.02 applies to HW_Perm_Fault;
-- if a Temporary Fault occurs, it disappears after a very short
time
Occurrence => poisson 10e+3 applies to HW_Repair_Temp;
-- The time needed by the error detection mechanisms
Occurrence => poisson 10e+2 applies to HW_Error_Detection_Action;
-- The permanent faults are detected or not with the following
probabilities:
Occurrence => fixed 0.75 applies to HW_Perm_Fault_Detected;
Occurrence => fixed 0.25 applies to HW_Perm_Fault_Non_Detected;
-- if a fault is not detected it will be perceived after the error
latency:
Occurrence => poisson 10e+4 applies to HW_Failure_Perceived;
-- the repair duration is given by:
Occurrence => poisson 10e-1 applies to HW_Repair_Perm;
end HW.Simple;
```

### 4.1.3 Hardware component: GSPN

For this case study, GSPNs for isolated error models (without propagations) are obtained through manual transformation of the AADL dependability model. The rules used for this transformation are enumerated in Table 1. This set of basic rules is to be completed later on, when introducing dependencies between error models.

**Table 1: Basic AADL error model to GSPN transformation rules**

| *AADL error model element* | *GSPN element* | | |
|---|---|---|---|
| Error state | Place | | |
| Initial error state (only one initial state can be declared in an error model) | Token in the place corresponding to this error state | | |
| Error event | (Internal) event, timed or immediate | | |
| Occurrence property | Arrival rate for events (distribution or probability) | | Timed arrival (distribution) |
| | | | Immediate arrival (probability) |
| Transition | State transition triggered by an event | Source state Event Destination state<br>Source state Event Destination state | |

Figure 4 presents the GSPN corresponding to the AADL error model specified for an isolated hardware component (Error Model 1).

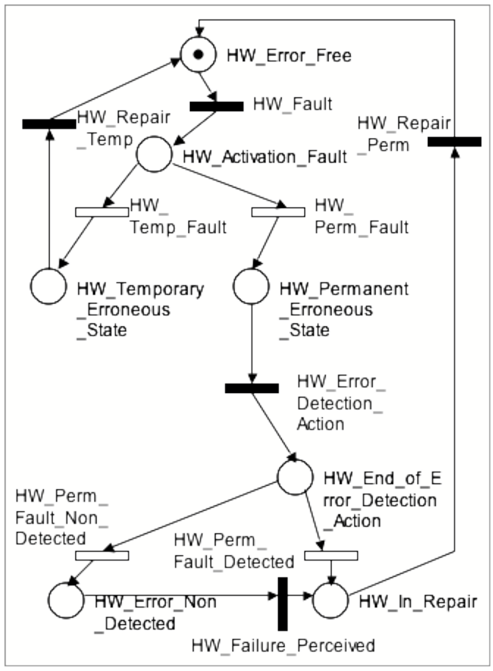



**Figure 4: GSPN modelling the dependability of isolated hardware components (M.HW)**

### 4.1.4 Software component: hypotheses

- Initially the component is in *Error_Free* state
- Faults are activated with a specified rate
- The error detection mechanisms need some time to detect an error. Also, an error is detected with a given probability, *ps* (not detected with the complementary probability *1-ps*).
- A detected error is processed during a certain amount of time. If the error was caused by the activation of a temporary fault (probability *1-ps*), its effects are eliminated by the error detection mechanisms (all temporary faults can be eliminated), so the component moves to the Error_Free state. If the error was caused by a permanent fault, the software needs to be restarted in order to eliminate the effects of the error.
- Effects of a non-detected error may disappear after a certain amount of time or may be perceived after a certain amount of time.

*Note*: The difference between hardware and software behaviours in the presence of faults is that for hardware, temporary and permanent faults are distinguished in this example by their respective consequences, following their activation, whereas for software they are distinguished after specific processing.

### 4.1.5 Software component: AADL error model

As usual, the *error model* is divided into two parts: the *error model type* - declaring error events and states - and the *error model implementation* - declaring transitions between states, triggered by events, and properties. The error model to be associated to isolated software components is given hereafter, in Error Model 2.

**Error Model 2: AADL Error Model for isolated software components**

**Error Model Type [SW]**

```
error model SW
features
-- events
SW_Fault, SW_Detection_Action, SW_Detected,
SW_Non_Detected, SW_Non_Detected_Disappear,
SW_Non_Detected_Perceived, SW_Error_Detected_Handling,
SW_Error_Temp, SW_Error_Perm, SW_Restart: error event;
-- states
SW_Error_Free: initial error state;
SW_Activation_Fault, SW_End_of_Error_Detection_Action,
SW_Error_Non_Detected, SW_Error_Detected,
SW_End_of_Exception_Handling, SW_In_Restart: error state;
end SW;
```

**Error Model Implementation [SW.Simple]**

```
error model implementation SW.Simple
transitions
SW_Error_Free-[SW_Fault]-> SW_Activation_Fault;
SW_Activation_Fault-[SW_Detection_Action] ->
SW_End_of_Error_Detection_Action;
SW_End_of_Error_Detection_Action-[SW_Detected] ->
SW_Error_Detected;
SW_End_of_Error_Detection_Action-[SW_Non_Detected] ->
SW_Error_Non_Detected;
SW_Error_Non_Detected-[SW_Non_Detected_Disappear] ->SW_Error_Free;

SW_Error_Non_Detected-[SW_Non_Detected_Perceived] ->SW_In_Restart;
SW_Error_Detected-[SW_Error_Detected_Handling] ->
SW_End_of_Exception_Handling;
SW_End_of_Exception_Handling-[SW_Error_Temp] ->SW_Error_Free;
SW_End_of_Exception_Handling-[SW_Error_Perm] -> SW_In_Restart;
SW_In_Restart-[SW_Restart] -> SW_Error_Free;
properties
-- a fault occurs following a poisson distribution
Occurrence => poisson 20e-1 applies to SW_Fault;
-- The error detection mechanisms need some time
Occurrence => poisson 10e+2 applies to SW_Detection_Action;
-- a fixed probability is associated with detection and non
detection
```

```
Occurrence => fixed 0.7 applies to SW_Detected;
Occurrence => fixed 0.3 applies to SW_Non_Detected;
-- The effects of a non detected error can dissapear after a while
or be perceived
Occurrence => poisson 10e+10 applies to SW_Non_Detected_Dissapear;
Occurrence => poisson 10e+6 applies to SW_Non_Detected_Perceived;
-- The time needed by the exception handling mechanisms
Occurrence => poisson 10e+2 applies to SW_Error_Detected_Handling;
-- The error is recovered with a given probability or
-- the software must be restarted (with the complementary
probability)
Occurrence => fixed 0.98 applies to SW_Error_Temp;
Occurrence => fixed 0.02 applies to SW_Error_Perm;
-- The restart takes some time
Occurrence => poisson 10e+2 applies to SW_Restart;
end SW.Simple;
```

### 4.1.6 Software component: GSPN

The GSPN shown in Figure 5 corresponds to the AADL error model of the software component. As in the case of the error model for the hardware, it was obtained through manual transformation of the AADL error model. Also, the rules used for the transformation are the same as in the case of the hardware system (see Table 1).

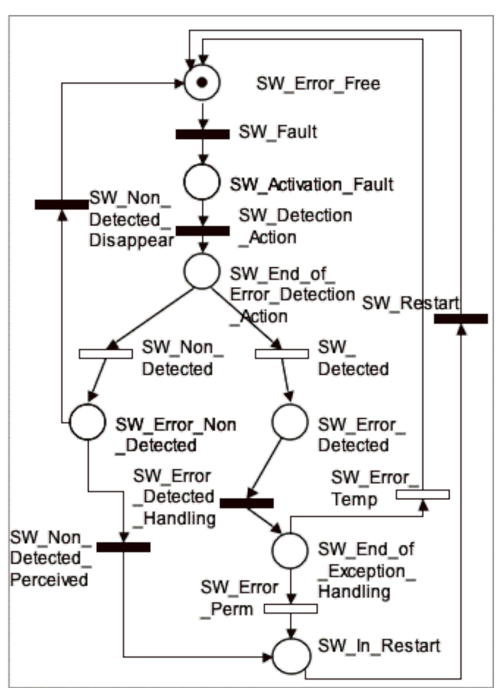



**Figure 5: GSPN modelling the dependability of isolated software components (M.SW)**

## 4.2 Third phase: Hardware and software components (HW-SW dependencies)

Mainly, dependencies between components are expressed through AADL error propagations. Propagations can be seen as events exchanged between error models. This means that when an out propagation (according to a specified Occurrence property, fixed probability or distribution) occurs, it is sent out of the source error model through all ports and bindings of the component to which the error model is attached. Consequently an out propagation arrives to one or more receiver component's error models. If the receiver error model declares an in propagation with the same name as the arriving out propagation, the name-match is done and the in propagation can influence the receiving error model (i.e., it may trigger a transition between error states or a mode transition). Conversely, if the receiver error model does not declare a corresponding in propagation, then it is not affected by the error propagation received.

However, if Vote (Vote_In, Vote_Out) properties are declared for ports of the AADL components that have associated error models, the matching between in and out propagations is performed in a different manner. These Vote properties are appended to the error model and they apply to AADL architectural elements (i.e., ports, data, client and server subprograms). Their role is to filter or mask propagations according to a specified Boolean error expression. The Boolean error expression can i) refer to states and propagations and ii) include Boolean operators (and, or, not, ormore, orless).

These in and out propagation matching principles are materialised on the duplex system case study in the remainder of the section. We remind that the AADL hardware component communicates with the AADL software component through port connections as shown in Figure 2. This interaction at the architectural level is the support of dependencies at the error model level. We remind that there are two dependencies between the hardware and the software (cf. section 2.1):

- The first one related to the error transmission from the hardware to the software, in case of temporary faults or non detected permanent faults in the hardware,
- The second one related to the stop / restart of the software in case of hardware failure.

The remainder of this section presents in a first step error models and corresponding GSPNs for hardware and software components (taking into account the HW-SW dependencies), in a second step Vote properties on ports and in a third step the merged GSPN for hardware and software.

### 4.2.1 Hardware component: AADL error model (HW-SW dependencies)

#### *a) Hardware component: Error transmission dependency*

Error Model 3 only shows what needs to be added to the error model for an isolated hardware component (Error Model 1) in order to describe the HW-SW error transmission dependency. The error model type for hardware components, [HW], needs to be completed in order to include out error propagation declarations corresponding to events that may affect the software running on top of the hardware.

Multiple error model implementations can be declared for the same error model type. We chose to declare one implementation each time a dependency is introduced. In addition to the declarations of the error model implementation [HW.Simple] (see Error Model 1), the error model implementation for hardware components taking into account the HW-SW error transmission dependency, [HW.HW_SW_ErrTransm_dep] (see Error Model 3), declares transitions that are triggered by the newly introduced out propagations. Note that, in this case, the destination error state is systematically the same as the source state when the transition is triggered by an out propagation. This means that when the error model propagates out an external event, it stays in the same state and may continue to propagate external events. One can specify that the destination state is different from the source state. In this latter case, the out propagation occurs only once and the component moves to the specified destination state. The semantics of the out propagations added to model the HW-SW dependency is as follows:

- HW_Temporary: a temporary fault in the hardware can cause error transmissions to the software running on top. The Occurrence property declared for HW_Temp_Fault is defined as a fixed probability, meaning that the propagation will occur with a certain probability if a temporary fault has occurred.
- HW_Permanent_Non_Detect: only a permanent fault that was not detected can cause error transmissions to the software running on top, just like in the case of a temporary fault. As in the case of the out propagation HW_Temporary, the propagation HW_Permanent_Non_Detect has an associated Occurrence property defined as a fixed probability.

**Error Model 3: AADL Error Model for hardware (HW-SW Error transm. dependency)**

```
Error Model Type [HW]

error model HW
features
-- [… ]
HW_Temporary, HW_Permanent_Non_Detect: out error propagation;
end HW;
```

```
Error Model Implementation [HW.HWSW_ErrTransm_dep]

error model implementation HW.HWSW_ErrTransm_dep
transitions
-- [… ]
-- **** specific transitions for the HW-SW error transmission dependency **** --
HW_Temporary_Erroneous_State-[out HW_Temporary] ->
HW_Temporary_Erroneous_State;
HW_Error_Non_Detected-[out HW_Permanent_Non_Detect] -> HW_Error_Non_Detected;
properties
-- [… ]
-- **** specific properties for the HW-SW error transmission dependency **** --
Occurrence => fixed 0.85 applies to  HW_Temporary;
Occurrence => fixed 0.65 applies to HW_Permanent_Non_Detect;
end HW.HWSW_ErrTransm_dep;
```

### *b) Hardware component: HW-SW Stop dependency*

Error Model 4 only shows what needs to be added to the error model for an isolated hardware component (Error Model 1) in order to describe the Hardware-Software Stop dependency. The error model type for hardware components, [HW], needs to be completed in order to include the out error propagation declaration corresponding to the failure of the hardware. In addition to the declarations of the error model implementation [HW.Simple] (see Error Model 1), the error model implementation for hardware components taking into account the HW-SW Stop dependency, [HW.HW_SW_Stop_dep], declares a transition triggered by the newly declared out propagation. The out propagation added to model the HW-SW Stop dependency models the following:

- HW_KO: is supposed to be used by the software component that runs on top of the hardware and by the repairman. When HW_KO is propagated to the software error model, the software cannot continue running anymore, as it needs the hardware to be operational. In addition, the software component can only be restarted when the hardware is repaired. The Occurrence property declared for HW_KO in HW.HW_SW_Stop_dep error model implementation is a fixed probability of 1, meaning that this propagation occurs certainly.

**Error Model 4: AADL Error Model for hardware (HW-SW Stop dependency)**

```
Error Model Type [HW]
error model HW
features
-- [… ]
HW_KO: out error propagation;
end HW;
```

```
Error Model Implementation [HW.HWSW_Stop_dep]
error model implementation HW.HWSW_Stop_dep
transitions
-- [… ]
-- **** specific transitions for the HW-SW stop/repair dependency **** --
HW_In_Repair-[out HW_KO] -> HW_In_Repair;
properties
-- [… ]-- **** specific properties for the HW-SW stop/repair dependency **** --
-- when the hardware fails, it certainly stops interacting with the software
Occurrence => fixed 1 applies to HW_KO;
end HW.HWSW_Stop_dep;
```

### 4.2.2 Hardware component: GSPN (HW-SW dependencies)

Figure 6 shows how the GSPN corresponding to the error model for the isolated hardware components was extended to take into account the out propagations imposed by the HW-SW dependencies. These out propagations are represented as transitions at the right of the figure, anticipating the fusion to in propagations at the connection of the hardware and software error models.

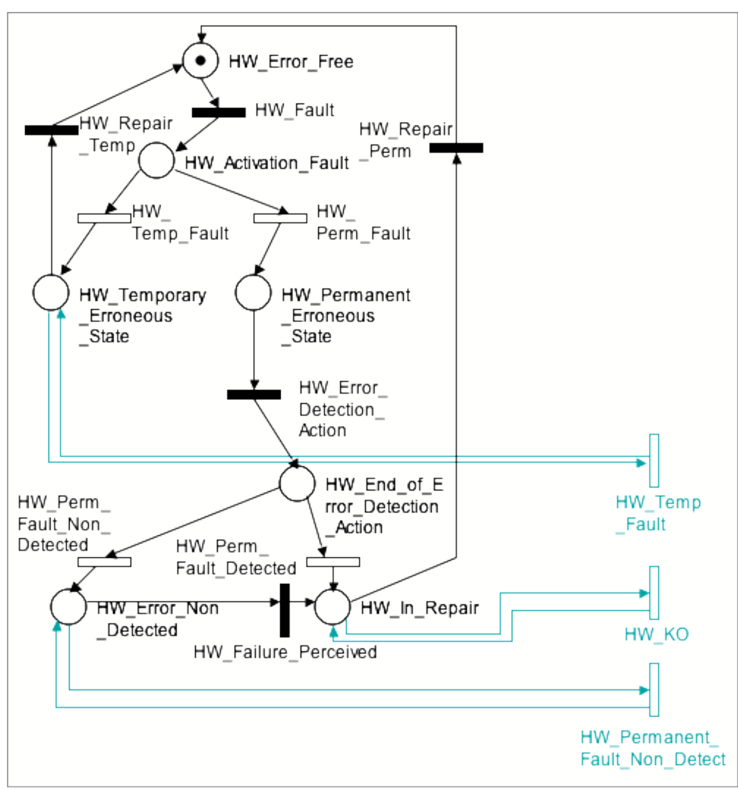



**Figure 6: GSPN modelling the dependability of hardware components - HW-SW dependencies (M.HW_HWSWdep)**

### 4.2.3 Software component: AADL error model (HW-SW dependencies)

The error model associated to the software component must be ready to receive propagations from the error model associated to the hardware component. More precisely, in propagations, corresponding to the out propagations declared in the error model associated to hardware, need to be declared together with transitions triggered by these in propagations.

The error model type specified for the isolated software component is extended with in error propagation declarations. As in and out propagations are matched through name matching across dependent architectural components, the in propagations in Error Model 5 and Error Model 6 have the same names as the out error propagations defined in the corresponding hardware error models (see Error Model 3 and Error Model 4).

#### *a) Software component: HW-SW Error transmission dependency*

The error model type [SW] declares additionally the following in propagations: HW_Temporary and HW_Permanent_Non_Detect. They name-match out propagations added in the error model type associated to the hardware component [HW] (see Error Model 3). The error model implementation [SW.HW_SW_ErrTransm_dep] defines transitions triggered by these in propagations. We consider that internal software errors followed by hardware error transmissions are very unlikely to happen, so these in propagations trigger transitions only if the current error state of the software is SW_Error_Free. Note that properties for in error propagations are not necessary as in propagations only occur as a consequence of an out propagation. Thus, in propagations inherit the properties of the corresponding out propagations.

**Error Model 5: AADL Error Model for Software (HW-SW Error transm. dependency)**

```
Error Model Type [SW]
error model SW
features
-- [… ]
HW_Temporary, HW_Permanent_Non_Detect: in error propagation;
end SW;
```

```
Error Model Implementation [SW.HWSW_ErrTransm_dep]
error model implementation SW.HWSW_ErrTransm_dep
features
transitions
-- [… ]
-- **** specific transitions for the HW-SW Error transmission dependency **** --
-- if the software is error free, the error transmission leads it in the Activation_Fault state.
-- the error is then processed as the internal ones
-- if the software is in another state (internal SW error followed shortly by a propagation)
-- we consider that the HW fault does not affect the state of the software
SW_Error_Free-[in HW_Temporary] -> SW_Activation_Fault;
SW_Error_Free-[in HW_Permanent_Non_Detect] -> SW_Activation_Fault;
properties
-- [… ]
end SW.HWSW_ErrTransm_dep;
```

### *b) Software component: HW-SW Stop dependency*

The error model type [SW] declares additionally the following in propagations: HW_OK and HW_KO. HW_KO name-matches the corresponding out propagation in the error model associated to the hardware component [HW] (see Error Model 4). HW_OK is highlighted in Error Model 6 as it does not directly name-match any out propagation from the error model associated to hardware. In fact, it is complementary to the HW_KO propagation. Section 4.2.5 will explain how this in error propagation is to be integrated into the global dependability model.

Besides declaring transitions triggered by the in propagations, the error model implementation refines the SW_In_Restart state as we need now to synchronise the restart of the software component with the repair of the hardware component. Note that we considered that refined states **are** visible from outside the error model. Refinements are no longer allowed by the AADL Error Model Annex v 0.8. However, we use them in order to trace modifications across model evolution phases.

**Error Model 6: AADL Error Model for Software (HW-SW Stop dependency)**

```
Error Model Type [SW]

error model SW
features
-- [… ]
HW_OK, HW_KO: in error propagation;
end SW;
```

```
Error Model Implementation [SW.HWSW_Stop_dep]

error model implementation SW.HWSW_Stop_dep
features
-- we extend the state SW_In_Restart.
--it is needed for the repair-restart process
SW_Needs_Restart, SW_Restarting: error state refines SW_In_Restart;
transitions
-- [… ]
-- **** specific transitions for the HW-SW Stop dependency **** --
SW_Error_Free-[in HW_KO] -> SW_Needs_Restart;
SW_Activation_Fault-[in HW_KO] -> SW_Needs_Restart;
SW_Error_Detected-[in HW_ KO] -> SW_Needs_Restart;
SW_Error_Non_Detected-[in HW_ KO] -> SW_Needs_Restart;
SW_Needs_Restart-[in HW_OK] -> SW_ Restarting;
properties
-- [… ]
end SW.HWSW_Stop_dep;
```

### 4.2.4 Software component: GSPN (HW-SW dependencies)

The GSPN corresponding to the software component taking into account the HW-SW dependencies is given in Figure 7. Each in propagation becomes a transition in the Petri Net, inherits the stochastic / temporal property of the matching out propagation and occurs only if the out propagation occurs. This cause-effect relationship between the out (cause) and the in (effect) propagation is represented as a dotted in arc on the transition triggered by the in propagation.

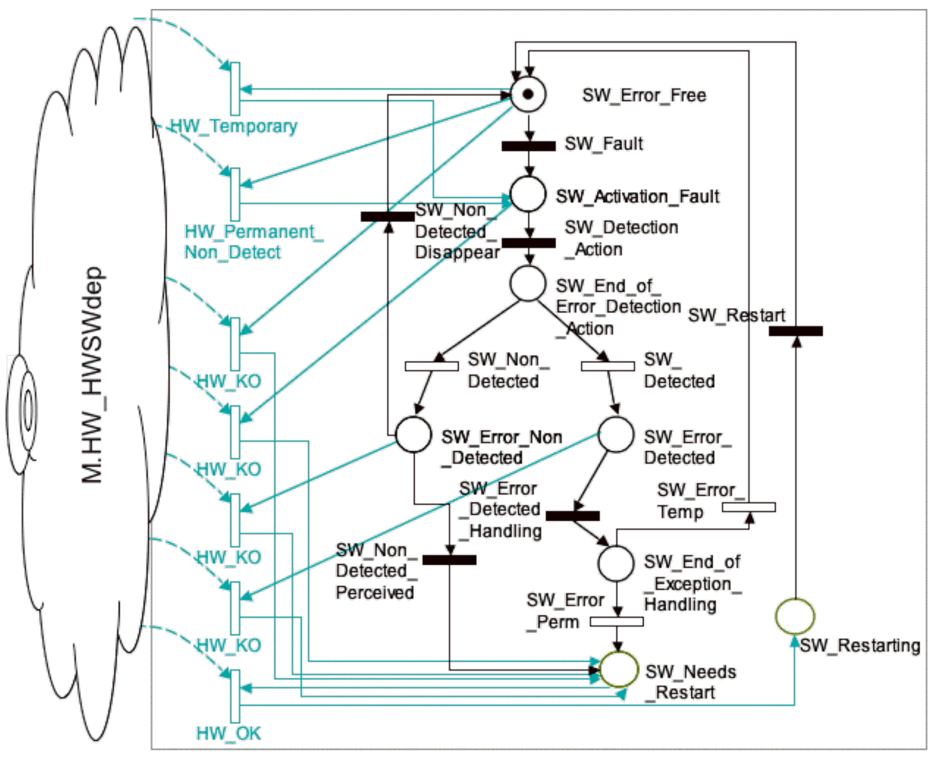



**Figure 7: GSPN modelling the dependability of software components - HW-SW dependencies (M.SW_HWSWdep)**

### 4.2.5 Software component: additional error properties on ports

One can compare names declared respectively for out and in propagations, in Error Model 3 [HW] and Error Model 6 [SW]. The ultimate aim is to match these propagations in order to obtain a model for the HW-SW dependencies. It is clear that the HW_OK in propagation declared in Error Model 6 [SW] does not directly name-match any out propagation declared in Error Model 3 [HW]. In fact the in propagations HW_OK and HW_KO are not independent. One is the complement of each other. If the software receives HW_OK, it means that the sender (i.e. the hardware) is **not** in HW_In_Repair state. Conversely, if the software receives HW_KO, it means that the sender (i.e., the hardware) is in HW_In_Repair state. It seems to us that the best way to express the generation of HW_OK is to use the Vote_In property on the event port used by the AADL software component for the connection to the AADL hardware component. Note that the Vote_In property is directly associated to an architectural feature inside the AADL architecture model of the software component. The way that the Vote_In property is expressed in the AADL architecture model is highlighted in AADL Component 1, which shows the AADL architecture model (type and implementation) for the software component that is initially in Primary mode. The error model discussed in section 4.2.3-b ([SW.SW_HW_Stop_dep]) is associated to the component implementation [software.basic_primary]. In addition, the error annex of this component declares a Vote_In property that applies to port interrupt_from_computer and that generates a HW_OK in propagation when the component that is connected to the interrupt_from_computer port is not in HW_In_Repair state.

**AADL Component 1: AADL architecture model for software component**

```
Component Type [software]
system software
features
interrupt_from_computer: in event port;
data_from_computer: in data port;
communicate: in out data port;
notification: in out event port;
-- we need this to model a mode switch triggered by some
-- internal event (auto-testable component)
inp: in event port;
outp: out event port;
end software;
```

```
Component Implementation [software.basic_primary]
system implementation software.basic_primary
modes
primary: initial mode;
backup: mode;
primary-[inp] -> backup;
backup-[notification] -> primary;
annex Error {** Model => Mymodels::SW.HWSW_Stop_dep;
  Vote_In => HW_OK when not interrupt_from_computer[HW_In_Repair]
            applies to interrupt_from_computer;
end software.basic_primary;
```

*Note:* The AADL Error Model Annex specifies that the Boolean error expression is evaluated only when a propagation arrives to the port to which the Vote property is attached. This means that if no propagation arrives to port interrupt_from_computer, the expression should not be evaluated and therefore, the propagation HW_OK will not be generated. However, we believe that when such Boolean error expressions refer to states, it would be useful to consider a quasi-continuous evaluation of the expression. Moreover, as our approach aims at generating a GSPN from the AADL dependability model, we can abstract away the notion of "evaluation" of the Boolean error expression. Instead, the Boolean expression needs to be transformed into a GSPN block to be included in the global GSPN of the system. The exact moment of evaluation of the Boolean expression may be left open by the standard so that different analyses methods may choose it according to their needs.

### 4.2.6 Merging propagations for global HW - SW GSPN

In order to obtain a global GSPN of a system including hardware and software, in and out propagations from separate models for hardware and software must be either i) directly matched and merged or ii) filtered and translated if Vote properties are declared. The two following sub-sections describe AADL error model to GSPN transformation rules for the two respective cases. Sub-section c) shows the GSPN obtained after application of these rules for the global hardware and software system.

#### *a) AADL error model to GSPN merging rules for direct name-matched propagations*

In case of direct name matching, the Occurrence property of an out propagation determines the value of the Occurrence property for GSPN propagation transitions obtained through direct name matching.

In a simple AADL model, we can suppose that an out propagation declared in **only one** "propagation sender" error model triggers n AADL transitions in this same error model (i.e. a particular propagation can be propagated out from multiple error states). A corresponding in propagation is declared in **only one** "propagation receiver" error model and triggers m transitions in this error model. In this case, the GSPN contains n*(m+1) propagation transitions. Intuitively, each AADL transition triggered by the out propagation in the "propagation sender" error model must be fired whether the "propagation receiver" error model is able to process the propagation (i.e., it is in an error state which is source state for an AADL transition triggered by the corresponding in propagation) or not. The example hereafter illustrates this intuitive rule.

Error Model 7 shows two error models (types and implementations):

- The one at the left corresponds to the propagation sender and declares one out propagation, named I_am_dying. This propagation can be propagated out from states Init and Normal (n=2). The I_am_dead propagation occurs according to a Poisson distribution and leads the component in the state Dead.
- The one at the right corresponds to the propagation receiver and declares one in propagation (named I_am_dying for the name match). When the receiver error model receives the propagation I_am_dying, it moves to the state Helping (m=1).

**Error Model 7: AADL error modelling example – name-matching propagations**

<table>
<tr>
<td>

**Error Model Type [Sender_example]**

```
error model Sender_example
features
Init: initial state;
Normal, Dead: state;
Init_Done: event;
I_am_dying: out error propagation;
end Sender_example;
```

</td>
<td>

**Error Model Type [Receiver_example]**

```
error model Receiver_example
features
Waiting: initial state;
Helping: state;
I_am_dying: in error propagation;
end Receiver_example;
```

</td>
</tr>
<tr>
<td>

**Error Model Implementation [Sender_example.basic]**

```
error model implementation
Sender_example.basic
transitions
Init- [Init_Done] -> Normal;
Init- [out I_am_dying] -> Dead;
Normal- [out I_am_dying] -> Dead;
properties
Occurrence => poisson 1e-1
      applies to Init_Done;
Occurrence => poisson 10e+2 applies to
      I_am_dying;
end Sender_example.basic;
```

</td>
<td>

**Error Model Implementation [Receiver_example.basic]**

```
error model implementation
Receiver_example.basic
transitions
Waiting- [in I_am_dying] -> Helping;
end Receiver_example.basic;
```

</td>
</tr>
</table>

Figure 8 shows the GSPN obtained after transformation of the Error Model 7 (AADL error modelling example concerning name-matching propagations).

The I_am_dying out propagation appears in two AADL transitions of the "propagation sender" model (n=2) while the I_am_dying in propagation appears in one AADL transition of the "propagation receiver" model (m=1). Consequently, these matching propagations correspond to n*(m+1)=2*(1+1)=4 transitions in the merged GSPN. Each of the two I_am_dying out propagations is merged with the corresponding I_am_dying in propagations. Also, the "propagation sender" has to be able to move to the error state Dead each time a I_am_dying out propagation occurs, even if the "propagation receiver" is not ready to process this propagation (i.e., it is not in the error state Waiting).

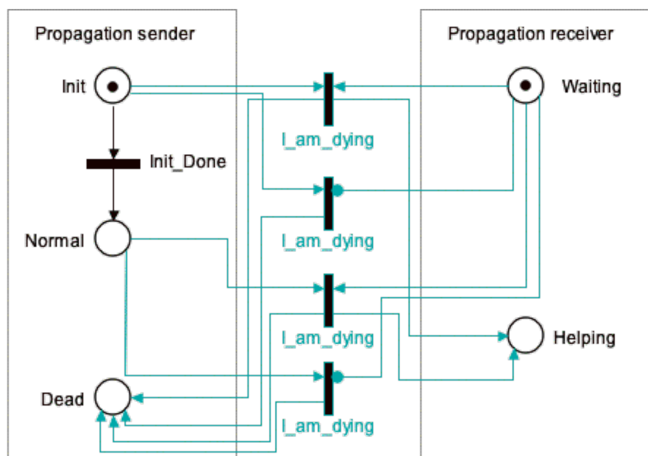



**Figure 8: GSPN modelling the example of the propagation sender and receiver (M.propagations)**

*Note:* In the particular case where the same error state is the source and the destination of transitions triggered by an out propagation in a "propagation sender" error model, the merging rule can be simplified. In fact, the token will not need to be moved to another state in the "propagation sender" error model. So, it is not necessary to fire GSPN transitions in case the "propagation receiver" is not ready to receive the propagation. This means that n*m transitions are sufficient.

The intuitive merging rule presented above needs to be generalised. In the most general case, we can suppose that the modeller declared out propagations named y in p "propagation sender" error models. In each "propagation sender" error model, the out propagation y triggers more than one transition. Also, the modeller declared in propagations named y in q "propagation receiver" error models. The in propagations y trigger more than one transition in each "propagation receiver" error model. The "propagation sender" and "propagation receiver" error models are associated to communicating architectural components (through connections or bindings). In this case, the number of interface GSPN transitions is the following:

$$N_{tr} = \sum_{i=1}^{p}(n_i * \prod_{j=1}^{q}(m_{ij} + 1)) \quad \textbf{(Eq. 1)}$$

where $N_{tr}$ = the number of interface GSPN transitions generated from a named (in/out) propagation;

$p$ = the number of "propagation sender" error models;

$q$ = the number of "propagation receiver" error models;

$n_i$ = the number of AADL transitions triggered by the out propagation in the "propagation sender" error model $i$ ;

$m_{ij}$ = the number of AADL transitions triggered by the in propagation corresponding to an out propagation from the "propagation sender" error model $i$ in the "propagation receiver" error model $j$.

Intuitively, each interface GSPN transition triggered by the out propagation in the "propagation sender" error models, must be fired whether the "propagation receiver" error

models are able to process the propagation or not. Consequently, combinations of source error states in the "propagation receiver" error models enable the interface GSPN transitions in the merged model.

#### *b) AADL error model to GSPN merging rules for propagations filtered and translated with* Vote *properties*

When Vote properties are used, the link between the two communicating error models has to be expressed in the GSPN through a sub-GSPN that translates the corresponding Boolean error expression. As Boolean error expressions may be complex, the corresponding GSPN may also be complex.

The Boolean error expression not interrupt_from_computer[HW_In_Repair], declared in the Vote_In property of the port interrupt_from_computer in AADL Component 1, can be translated to a GSPN inhibitor arc, according to the rule shown in Table 2.

**Table 2: AADL Vote (In/Out) properties to GSPN transformation rules (1)**

| *AADL Boolean error expression (Vote_In)* | *GSPN element* | |
|---|---|---|
| *propag_name* when not *port*[*state_name*] | Inhibitor arc from state *state_name* to *propag_name* transition | Sender model<br>state_name<br>propag_name |

More investigation is required to define an exhaustive set of transformation rules for complex Boolean error expressions.

#### *c) Global Hardware - Software GSPN*

We remind that hardware and software are modelled as AADL system components and they are linked through connections at the architectural level. Consequently, in and out propagations are either i) name-matched or ii) linked according to Vote properties in the error models attached to these components. In this way, a global dependability model is obtained in the form of a GSPN. Figure 9 shows the GSPN representing hardware – software sub systems. The transition HW_OK is obtained through transformation of a Vote_In property, while all the other transitions are obtained after merging direct name-matched in and out propagations. It is worth noting that the Occurrence property of the HW_OK transition is a probability of 1, as the Boolean error expression associated to the Vote_In property does only involve one state and no propagations.

It is worth noting that the source and destination error states for AADL transitions triggered by out propagations in the error model associated to the hardware component are systematically the same. Consequently, we used the simplified merging rule for the direct name-matching propagations (see the *note* in section 4.2.6-a)). This rule allows us to reduce significantly the number of GSPN transitions.

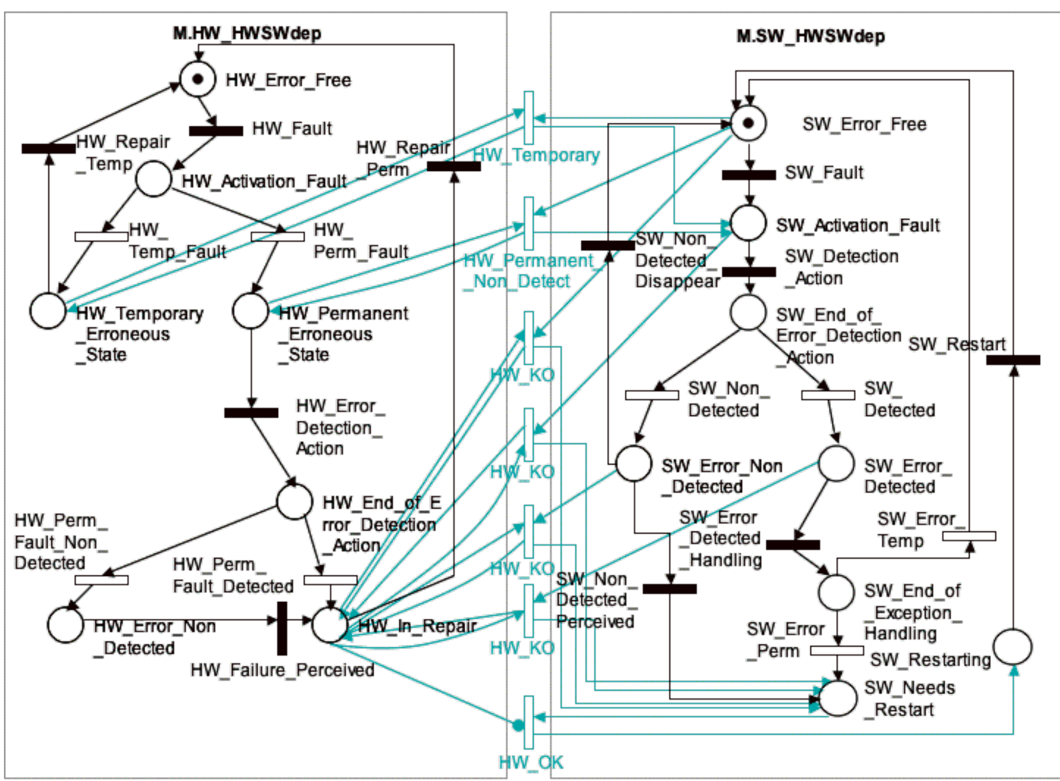



**Figure 9: GSPN modelling the dependability of hardware and software components - HW-SW dependency (M.HW_SWdep)**

## 4.3 Fourth phase: Hardware components (HW-HW maintenance & repair strategy)

The repairman is a shared resource at the level of the global system as we suppose that only one repairman is available for both hardware components. If one hardware component fails, the repairman is busy to repair it for a while. So, if the second hardware component fails during this time, it has to wait the end of the ongoing repair.

At architectural level, the repairman is represented as an AADL system component. It does not have a particular role at this level as it only intervenes in the maintenance & repair strategy that is modelled at error model level. However, it must be connected to the dependent hardware components in order to allow propagations (at error model level) to be exchanged between hardware components and the repairman. The repairman is connected to the hardware components through event connections, as shown previously in Figure 2. The following sub-sections present respectively the error models for the hardware and for the repairman, the respective corresponding GSPNs and the complete GSPN for the hardware – repairman sub system.

### 4.3.1 Hardware component: AADL error model (HW-HW dependency)

Similarly to the previous sections, we show in Error Model 8 only what we need to add to Error Model 3 ([HW] and [HW.HW_SWdep]) in order to take into account the new HW-HW dependency.

In this phase of modelling we need to distinguish between the states where the hardware uses the repairman (shared resource). So, in the hardware error model, the state HW_In_Repair is

refined into three distinct states: HW_Needs_Repair, HW_Repairing, HW_Repaired. In states HW_Needs_Repair and HW_Repairing, the hardware is considered to be Failed. Events and error propagations that used to trigger transitions to and from the state HW_In_Repair will now trigger transitions to and from one of the refined states. The new transitions are declared explicitly in the implementation [HW.HWSW_HWHWdep] (see Error Model 8).

We also need to declare a supplementary out propagation, HW_I_Am_Repaired, which is used to release the repairman when the repair is finished. Transitions are customized according to the refined states and supplementary error propagation.

**Error Model 8: AADL Error Model for Hardware (HW-SW, HW-HW dependency)**

**Error Model Type [HW]**

```
error model HW
features
-- [… ]
HW_I_Am_Repaired: out error propagation;
end HW;
```

**Error Model Implementation [HW.HWSW_HWHWdep]**

```
error model implementation HW.HWSW_HWHWdep
features
-- we refine the state HW_In_Repair to make a difference between Failed states
-- when the repairman is working or not.
HW_Needs_Repair, HW_Repairing, HW_Repaired: error state refines HW_In_Repair;
transitions
-- [… ]
-- **** specific transitions for the HW-HW dependency **** --
HW_Needs_Repair-[out HW_KO] -> HW_Needs_Repair;
HW_Needs_Repair-[in Repairman_I_Repair_You] -> HW_Repairing;
HW_Repairing-[HW_Repair_Perm] -> HW_Repaired;
HW_Repaired-[out HW_I_Am_Repaired] -> HW_Error_Free;
properties
-- [… ]
-- **** specific properties for the HW–HW dependency **** --
--  the hardware is repaired => it certainly frees the repairman and goes to Error_Free state
Occurrence => fixed 1 applies to  HW_Repaired;
end HW.HWSW_HWHWdep;
```

### 4.3.2 Hardware component: GSPN (HW-HW dependency)

The AADL error model for hardware given in Error Model 8 is translated into the GSPN shown in Figure 10. Newly declared propagations are mapped to transitions as previously. Transitions obtained from propagations are represented at the left of the figure, preparing the merging to the corresponding transitions obtained from the repairman's error model. Refined states are highlighted. They correspond to HW_Needs_Repair, HW_Repairing and HW_Repaired.

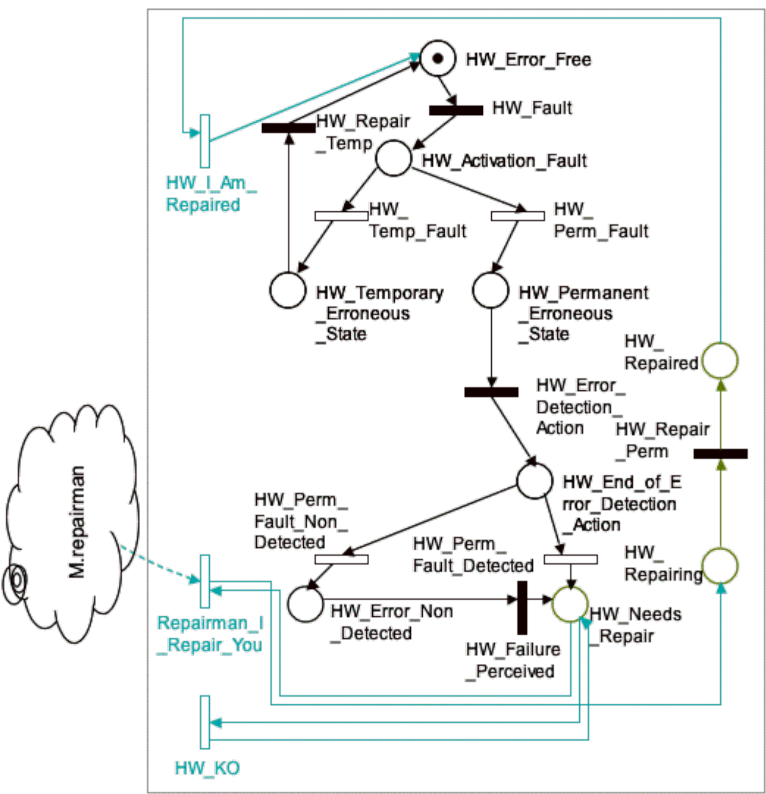



**Figure 10: GSPN modelling the hardware – HW-HW dependency (M.HW_HWHWdep)**

### 4.3.3 Repairman component: AADL error model

The error model of the repairman is given in Error Model 9. It declares two states: Repairman_Free (initial state) and Repairman_Working. The transitions from one to the other are triggered by error propagations coming from hardware. If the hardware fails, it sends an HW_KO out propagation that name-matches with the corresponding in propagation from the error model of the repairman. If the repairman is in state Repairman_Free when it receives the propagation HW_KO, it goes to the state Repairman_Working and sends out the propagation Repairman_I_Repair_You that triggers a state change in the hardware error model. More concretely, the hardware goes from state HW_Needs_Repair to state HW_Repairing. When the hardware is repaired it sends out the propagation HW_Repaired and the repairman goes back to the Repairman_Free state.

**Error Model 9: AADL Error Model for the repairman**

```
Error Model Type [Repairman]

error model Repairman
features
-- propagations
Repairman_I_Repair_You: out error propagation;
HW_KO,HW_I_Am_Repaired: in error propagation;
-- states => Petri Net states
Repairman_Free: initial error state;
Repairman_Working: error state;
end Repairman;
```

```
Error Model Implementation [Repairman.Simple]

error model implementation Repairman.Simple
-- transitions => Petri Net transitions
transitions
Repairman_Free-[in HW_KO] -> Repairman_Working;
Repairman_Working-[out Repairman_I_Repair_You] ->
Repairman_Working;
Repairman_Working-[in HW_I_Am_Repaired] -> Repairman_Free;
properties
Occurrence => fixed 1 applies to Repairman_I_Repair_You;
end Repairman.Simple;
```

#### 4.3.4 Repairman component: GSPN

The GSPN for the repairman is given in Figure 11. It is obtained from the Error Model 9 after applying, as previously, the transformation rules defined in Table 1. Note that all transitions inside this model are obtained from propagations. So, these transitions are to be merged to the corresponding ones from the two GSPN of the hardware components (M.HW_HWHWdep).

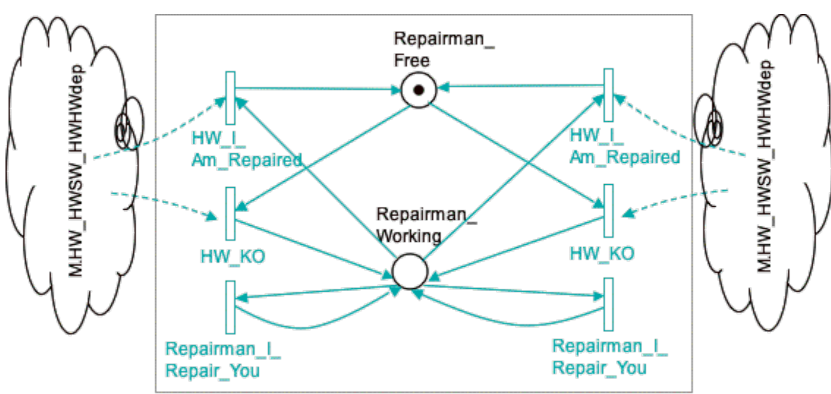



**Figure 11: GSPN modelling the repairman (M.repairman)**

### 4.3.5 Global Hardware – Repairman GSPN (HW-HW dependency)

No Vote properties were declared for the maintenance & repair dependency. Consequently, the GSPN shown in Figure 12 is obtained after direct name matching and merging of in and out propagations from the error models of the hardware and of the repairman. Merging rules are defined in section 4.2.6-a). As in the case of the HW-SW dependencies, the source and destination states of AADL transitions triggered by out propagations are the same. So, we used the simplified merging rule for direct name-matching propagations. Note that the GSPN shown in Figure 12 only represents the two hardware components and the repairman linked by the HW-HW dependency. So, in order to have a global GSPN for the system formed of hardware, software and repairman, this GSPN (Figure 12) must be superposed on the GSPN that represents hardware and software linked by the HW-SW dependency (Figure 9).

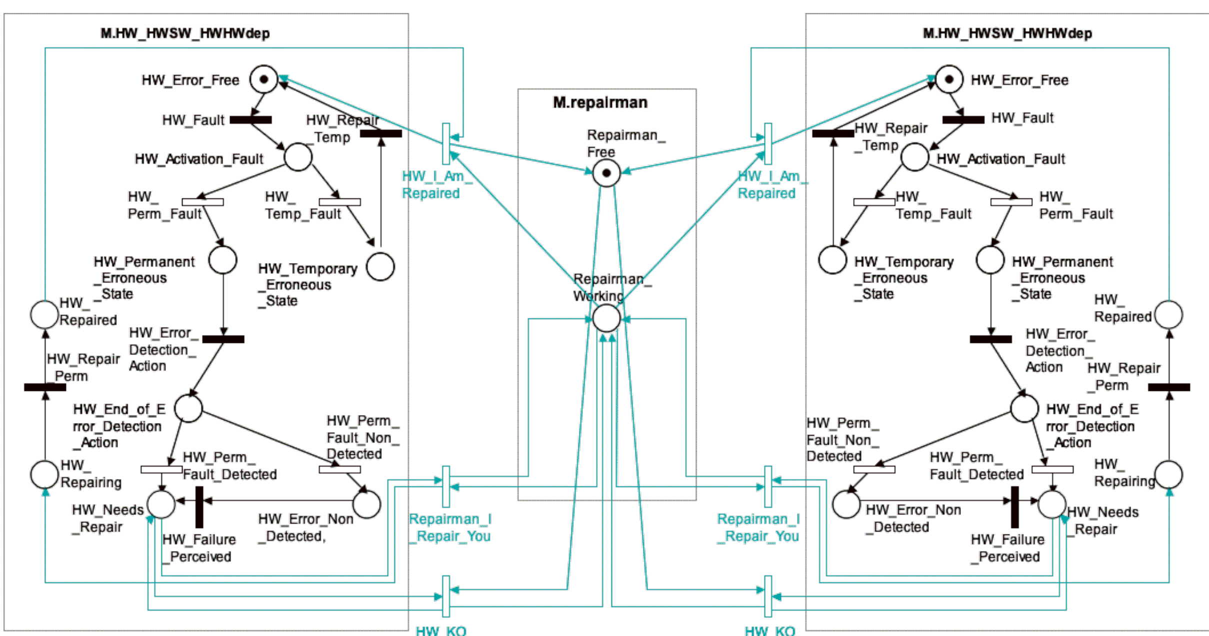



**Figure 12: GSPN modelling the hardware and the repairman, taking into account the maintenance & repair dependency (M.HW_HWdep)**

## 4.4 Fifth phase: Software components (fault tolerance dependency)

We remind that the duplex system we intend to model must tolerate failures of its sub systems. It is composed of two sub systems, each of them formed of a hardware component and a software component. At every moment of time, one of the software replicas has the role Primary and it is active while the other software replica, the Backup, is passive and monitors the Primary. If the Primary fails, the two replicas are supposed to switch roles: the Backup is supposed to become active and to continue delivering service. In the meantime, the failed replica must be restarted.

We consider that Primary and Backup are two operational modes for each software replica. One of the replicas is initially in mode Primary while the other one is in mode Backup. The role switch is performed at architectural level through mode changes.

In AADL, mode changes are triggered by events or error propagations that go through ports. More precisely, one can specify that component Y will go from operational mode A to operational mode B and that the transition is triggered by port P. This means that any signal

(architectural event or error propagation) that goes through port P will trigger the mode change. However, the property Vote_Transition (specified in the Error model annex) can be associated to ports in order to select the signals allowed to trigger the mode change. In this way, some of the signals may be ignored.

AADL Component 2 presents the AADL architecture model for the software replica that is initially in mode Primary. Note that the Vote_In property that generates the propagation HW_OK was not changed here, after the refinement of the state HW_In_Repair. It is supposed that the Boolean error expression applies to all refined states. Also, note that the only difference between the two software models lies in the initial mode: one is initially in mode Backup while the other one is initially in mode Primary. The same error annex (error model + Vote properties) is associated to both software components. AADL Component 2 highlights the Vote_In and Vote_Transition properties on ports inp and notification, as they are the key elements for the mode switches. These properties are used to filter signals arriving through the ports triggering the mode switches.

The highlighted Vote_In property generates an in propagation, SW_Both_Dead, from two in propagations SW_KO arriving through ports inp and notification. SW_Both_Dead is used to change the recovery policy if both replicas fail. In this case, the first one operational becomes Primary.

The two Vote_Transition properties specify that the mode transitions can only be triggered by the error propagations SW_KO or SW_I_Am_Restarted. Concretely, one software component moves from mode Primary to mode Backup either if SW_KO is received through the port inp, meaning that the component itself failed, or if SW_I_Am_Restarted is received through the port notification, meaning that the other component has been repaired after a double software failure. Conversely, one software component moves from mode Backup to mode Primary either if SW_KO is received through the port notification, meaning that the other software component failed, or if SW_I_Am_Restarted is received through the port inp, meaning that the component itself is already repaired after a double software failure.

Note that the last AADL Error Model Annex rule concerning error propagation paths between basic error models ("errors do not propagate from a basic error model to itself") might be violated as we explicitly modelled auto-connections for the software components, to send error propagations that would trigger mode changes in the component itself. We used propagations arriving through these auto-connections in Vote properties.

Note that, in this model, the mode switch is done symmetrically before restarting the failed replica. Concretely, when the Primary fails, it first goes to mode Backup and then it is restarted (according to the constraints imposed by the HW-SW Stop dependency) while the other replica goes to mode Primary. If both software components fail one after the other, the switch policy is different. In this case, the first software component repaired takes the role Primary.

**AADL Component 2: AADL architecture model for software component (Vote_Transition properties)**

```
Component Type [software]
system software
features
interrupt_from_computer: in event port;
data_from_computer: in data port;
communicate: in out data port;
notification: in out event port;
inp: in event port;
outp: out event port;
end software;
```

```
Component Implementation [software.basic_primary]
system implementation software.basic_primary
modes
primary: initial mode;
backup: mode;
primary-[inp] -> backup;
backup-[notification] -> primary;
annex Error {** Model => Mymodels::SW.HWSW_SWSWdep;
Vote_In => HW_OK when not interrupt_from_computer[HW_In_Repair]
  applies to interrupt_from_computer;
-- when 2 transitions SW_I_Am_Dead received => both SW dead
Vote_In =>     SW_Both_Dead when inp[SW_KO] and notification[SW_KO],
        applies to inp, notification;
-- mode transitions allowed only when the SW_KO occurs
Vote_Transition => inp[SW_KO]  or notification[SW_I_Am_Restarted]
  applies to inp;
Vote_Transition => notification[SW_KO]  or inp[SW_I_Am_Restarted]
  applies to notification;
end software.basic_primary;
```

The following sub sections present respectively the error model for the software components, taking into account the SW-SW dependency, and the model transformation to GSPN.

### 4.4.1 Software component: AADL error model (SW-SW dependency)

Error Model 10 shows what we need to add to the previous error model for the software component (Error Model 6), in order to represent the SW-SW dependency. First, we need to declare three additional propagations in the error model type [SW]:

- SW_KO that notifies the failure to the other software component,
- Both_SW_Dead that is generated by a Vote_In property in case of double software failure (i.e., both software replicas failed),
- SW_I_Am_Restarted that notifies the end of the restart procedure after a double software failure (i.e., both software replicas failed) to the other software component.

Second, we need to define transitions triggered by these propagations and Occurrence properties for the out (or in out propagations). When one software component is in the SW_Needs_Restart error state, it propagates out SW_KO. Then, the software component is restarted (if the hardware is not failed). Note that we declared an additional state, SW_Now_Restart in order to give priority to the mode change. We remind that the mode change must be done before restarting the failed software. So, the propagation SW_KO is sent out immediately with a probability of 1. Then, the software component goes to state SW_Now_Restart where it is allowed to restart. Also, if the Both_SW_Dead propagation is received, the component is restarted (according to the constraints of the HW-SW Stop dependency). Note that the model for the HW-SW dependency also needs to be adapted, as the authorisation for software to restart is needed in two states: SW_Now_Restart and SW_Both_Dead. In addition, we need to distinguish the states SW_Restarting and SW_Restarting_Both_Dead as we need to notify the end of restart to the other software replica if both failed. The end of restart is modelled by the state SW_Restarted. From this state, the notification is sent through the propagation SW_I_Am_Restarted. Then, the component goes to state SW_Error_Free.

**Error Model 10: AADL Error Model for the software (SW-SW dependency)**

```
Error Model Type [SW]

error model SW
features
-- [… ]
SW_KO, SW_I_Am_Restarted: in out error propagation;
Tempo: error event;
Both_SW_Dead: in error propagation;
end SW;
```

```
Error Model Implementation [SW.HWSW_SWSWdep]

error model implementation SW.HWSW_SWSWdep
features
-- we also refine the state SW_In_Restart.
SW_Needs_Restart, SW_Now_Restart, SW_Restarting, SW_Restarting_Both_Dead,
SW_Both_Dead, SW_Restarted: error state refines SW_In_Restart;
-- transitions => Petri Net transitions
transitions
-- [… ]
-- **** specific transitions for the SW-SW (and HW-HW) dependency **** --
SW_Needs_Restart-[out SW_KO] -> SW_Needs_Restart;
SW_Needs_Restart-[Tempo] -> SW_Now_Restart;
SW_Needs_Restart-[in Both_SW_Dead] -> SW_Both_Dead;
SW_Now_Restart-[in HW_OK] -> SW_Restarting;
SW_Both_Dead-[in HW_OK] -> SW_Restarting_Both_Dead;
SW_Restarting_Both_Dead-[SW_Restart_Delay] -> SW_Restarted;
SW_Restarting_Both_Dead-[in SW_I_Am_Restarted] -> SW_Error_Free;
SW_Restarted-[out SW_I_Am_Restarted] -> SW_Error_Free;
properties
-- [… ]
Occurrence => fixed 1 applies to  SW_KO;
Occurrence => fixed 1 applies to  SW_I_Am_Restarted;
Occurrence => poisson 10e+10 applies to  Tempo;
end SW.HWSW_SWSWdep;
```

### 4.4.2 Software component: GSPN (SW-SW dependency)

#### a) *Partial GSPN blocks obtained from the AADL architecture model and the AADL error model*

The AADL architecture model contains modes and mode transitions that form a state machine. This behavioural description is the part of the architecture needed to generate the dependability model. The rules used for the transformation from the AADL architecture behaviour model to Petri nets are presented in Table 3. Note that they are very similar to those presented in Table 1 for the AADL error model to GSPN transformation.

**Table 3: AADL architecture model to PN transformation rules**

<table>
<tr><th>AADL architecture element</th><th colspan="3">PN element</th></tr>
<tr><td>Mode</td><td>Place</td><td colspan="2"></td></tr>
<tr><td>Initial mode (only one initial mode can be declared for one component)</td><td>Token in the place corresponding to this mode</td><td colspan="2"></td></tr>
<tr><td>Port</td><td>Event triggering a transition (it has no Occurrence property)</td><td></td><td>(may be immediate or timed if it corresponds to error propagations coming through the port)</td></tr>
<tr><td>Transition</td><td>Transition (triggered by a port)</td><td colspan="2">Source mode Port Destination mode</td></tr>
</table>

Figure 13 shows the Petri Net obtained from the AADL behavioural description of software components (see AADL Component 2). Note that in Figure 13 no Occurrence properties are associated to transitions. These are determined from the Vote_Transition properties associated to ports through the error model. The only difference between the architectural description of the two software replicas lies in the initial operational mode: one is initially in Primary mode while the other one is in Backup mode.

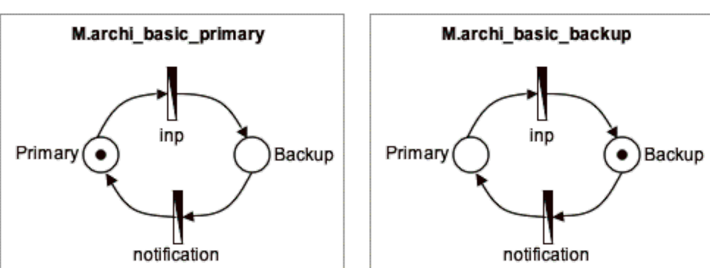



**Figure 13: GSPN modelling the software architectural behaviour (M.archi_basic_primary and M.archi_basic_backup)**

Figure 14 presents the GSPN obtained from Error Model 10. The transformation rules are the same as in the previous sections.

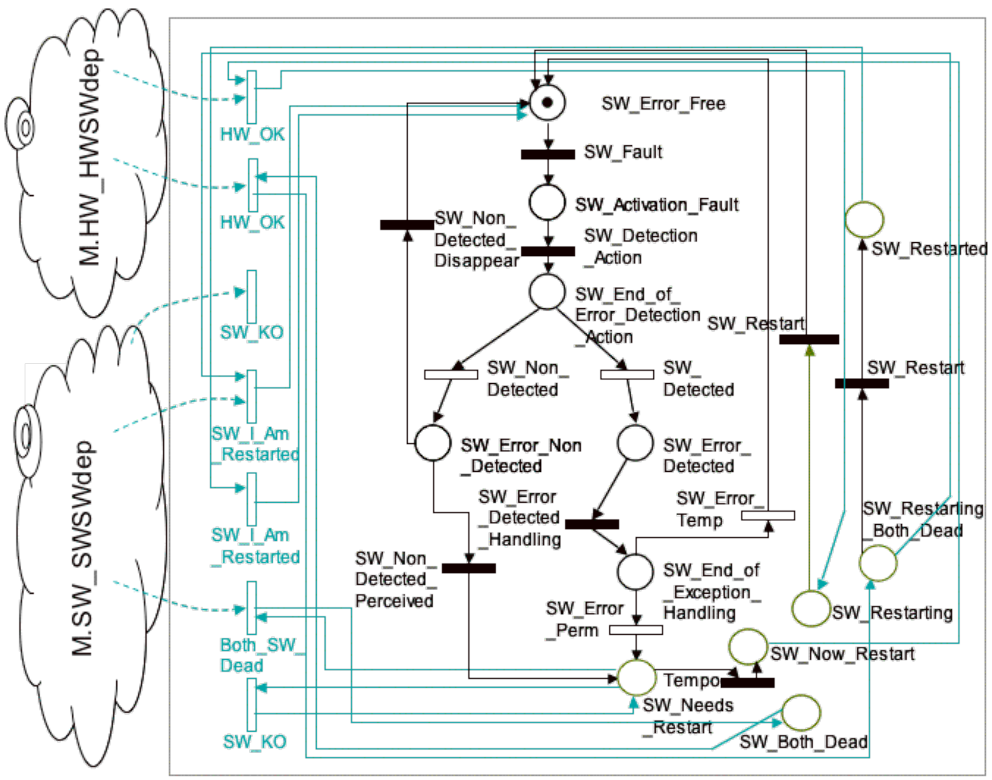



**Figure 14: GSPN modelling the software behaviour in the presence of faults - SW-SW dependency (M.SW_HWSW_SWSWdep)**

***b) Merging partial GSPN blocks describing the SW-SW dependency***

The GSPN shown in Figure 14 needs to be merged to the GSPN shown in Figure 13 in order to obtain a global dependability model for the software components (from the point of view of the SW-SW dependency). For that, some additional rules are necessary. It is mainly necessary to define rules to link the GSPN obtained from the error model to the GSPN obtained from the architecture behaviour model. This link is done through error propagations and Vote_Transition properties. If no Vote_Transition is declared for ports that appear in mode transitions, it means that the mode transition will be triggered by any architectural event or error propagation that goes through that port. It seems to us that a good way to prevent "false" mode transitions and to simplify the model transformation process is to declare Vote_Transition properties for all ports used in mode transitions. In the particular case of our model, Vote_Transition properties were declared for the ports inp and notification that trigger the mode transitions. In this way, mode transitions are allowed only when the specified error propagations are received through these ports. If we allow mode transitions to be triggered by architectural events, it would be difficult or even impossible to attach to them probabilities or time properties that are needed for the dependability analysis. Consequently, for dependability analysis, we inhibit or we ignore any mode transition triggered by architectural events.

We first need to translate into GSPN the Boolean error expressions used in Vote_Transition properties in order to match out propagations to in propagations that may trigger mode changes. This transformation is done in a similar way to that presented for the case of

Vote_In properties (see Table 2). Once the Boolean error expression has been transformed into a GSPN block, the "propagation sender" and the "propagation receiver" models (mode models + error models) need to be merged. The merging rules are similar to those for direct name-matching propagations (see section 4.2.6-a)). Concretely, a particular in error propagation may trigger multiple independent mode changes. Consequently, combinations of source modes enable the interface GSPN transitions in the merged model.

As in the case of Vote_In and Vote_Out properties, the transformation rule used for the Vote_Transition Boolean error expression is given in Table 4.

**Table 4: AADL Vote Transition properties and merging rules for GSPNs obtained from AADL component architectural model and associated error model**

| ***AADL Boolean error expression*** <br> ***(Vote_Transition)*** | ***GSPN element*** |
|---|---|
| Vote_Transition applies to *port* | Identify transitions in architectural model triggered by *port.* |
| *port1*[*propag1*] **or** *port2*[*propag2*] | Merge transitions triggered by *port* in the architectural model respectively with transitions corresponding respectively to *out propagation1* and *out propagation2* in the GSPN obtained from the error model associated to the component connected through an architectural connection to *port*. |

The error annex associated to the software components declares a Vote_In property that applies to ports inp and notification (see AADL Component 2). This Vote_In property transforms SW_KO propagations arriving through both ports (i.e., both software components are in SW_Needs_Restart state) into a different in propagation, SW_Both_Dead. This Vote_In property requires the definition of a new transformation rule. The transformation rule presented in Table 5 applies to a Boolean error expression that translates a combination of several propagations into a new propagation. Of course, more effort is needed to generalise this rule according to the complexity of the expression of the combination of propagations.

**Table 5: AADL Vote (In/Out) properties to GSPN transformation rules (2)**

| ***AADL Boolean error expression (Vote_In)*** | ***GSPN element*** | |
|---|---|---|
| *propag_result* **when** *port1[propag1]* **and** *port2[propag2]* | Each of the two specified propagations (i.e., *propag1* and *propag2*) brings a token in an intermediary place. The resulting propagation (*propag_result*) is represented as an immediate transition enabled by:<br>- the existence of two tokens in the intermediary place and<br>- the existence of a token in a source state of one of the transitions triggered by the resulting propagation.<br>Eventually, a timed transition will evacuate the tokens from the intermediary place (if they were not consumed by the resulting immediate transition). | Receiver model<br>in propag_result<br>in propag1<br>2<br>2<br>in propag2 |

Figure 15 shows the GSPN obtained after merging the architectural models and the error models associated to software components. For the sake of clarity, only the GSPN places used in the SW-SW dependency model are shown here. The GSPN blocks, corresponding to each of the two software components, are highlighted. Also, some of the Petri net places are duplicated.

Note that, for direct name-matching propagations, we generally used the simplified merging rule with one exception. The source and destination states for the AADL transition triggered by the out propagations SW_I_Am_Restarted are different. In this case, we have to use the general merging rule. The semantics of the notations used in section 4.2.6-a) are extended here to include in the propagation receiver model the mode model.

$N_{tr}$ = the number of interface GSPN transitions generated from a named (in/out) propagation;

$p$ = the number of "propagation sender" models[2];

$q$ = the number of "propagation receiver" models;

$n_i$ = the number of AADL transitions triggered by the out propagation in the "propagation sender" error model $i$ ;

[2] Model = component mode model with associated error model

$m_{ij}$ = the number of AADL transitions triggered by the in propagation corresponding to an out propagation from the "propagation sender" error model $i$ in the "propagation receiver" error model *j*.

$l_{ij}$ = the number of mode transitions triggered by the in propagation corresponding to an out propagation from the "propagation sender" error model $i$ in the "propagation receiver" mode model *j*.

In order to take into account the architectural behaviour, the expression of $N_{tr}$ is as follows:

$$N_{tr} = \sum_{i=1}^{p}(n_i * \prod_{j=1}^{q}(m_{ij}+1)*(l_{ij}+1)) \qquad \textbf{(Eq. 2)}$$

The parameter values, in the case of the propagation SW_I_Am_Restarted, are as follows:

- $p$=2 (i.e., SW_I_Am_Restarted is propagated out by both software components), $q$=2 (SW_I_Am_Restarted is a in propagation in both software components),
- $n_1$=1, $n_2$=1 (i.e., only one AADL transition is triggered by the out propagation in the error model associated to both software components),
- $m_{11}$=0 (i.e., no AADL transition is triggered in the error model associated to component SW1 by the in propagation corresponding to the out propagation sent by this same error model) $m_{12}$=1 (i.e., one AADL propagation is triggered in the error model associated to component SW2 by the in propagation corresponding to the out propagation sent by the error model associated to component SW1), $m_{21}$=1 (i.e., one AADL propagation is triggered in the error model associated to component SW1 by the in propagation corresponding to the out propagation sent by the error model associated to component SW2), $m_{22}$=0 (i.e., no AADL propagation is triggered in the error model associated to component SW2 by the in propagation corresponding to the out propagation sent by this same error model),
- $l_{11}$=1 (i.e., one mode change is triggered in the mode model of component SW1 by the in propagation corresponding to the out propagation sent by the error model associated to component SW1), $l_{12}$=1 (i.e., one mode change is triggered in the mode model of component SW2 by the in propagation corresponding to the out propagation sent by the error model associated to component SW1), $l_{21}$=1 (i.e., one mode change is triggered in the mode model of component SW1 by the in propagation corresponding to the out propagation sent by the error model associated to component SW2), $l_{22}$=1 (i.e., one mode change is triggered in the mode model of component SW2 by the in propagation corresponding to the out propagation sent by the error model associated to component SW2).

We can explicit (Eq. 2) as follows.

$$N_{tr} = n_1*[(m_{11}+1)*(m_{12}+1)*(l_{11}+1)*(l_{12}+1)]+n_2*[(m_{22}+1)*(m_{21}+1)*(l_{21}+1)*(l_{22}+1)]$$

$$\Rightarrow N_{tr-SW_I_Am_Restarted} = 16$$

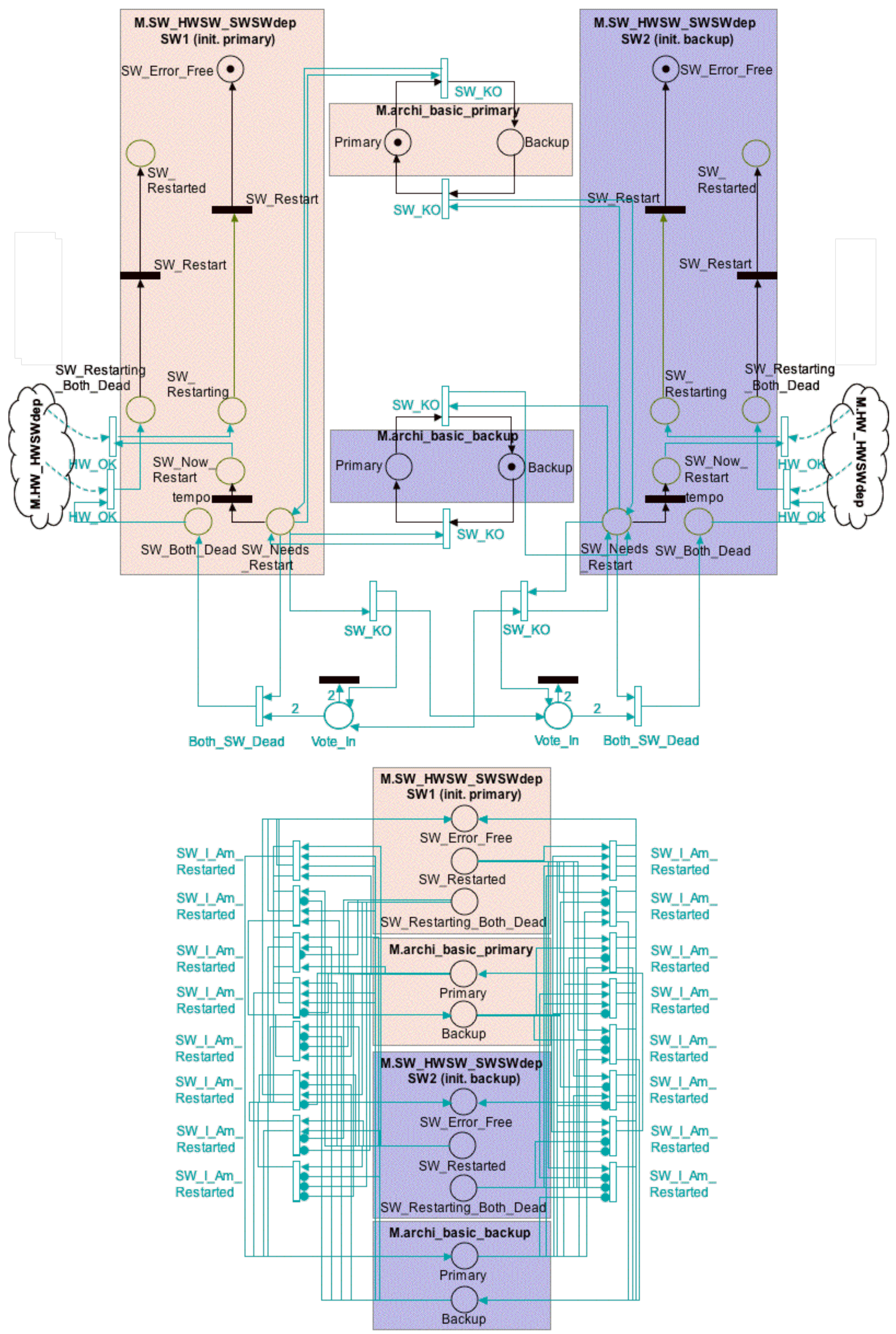
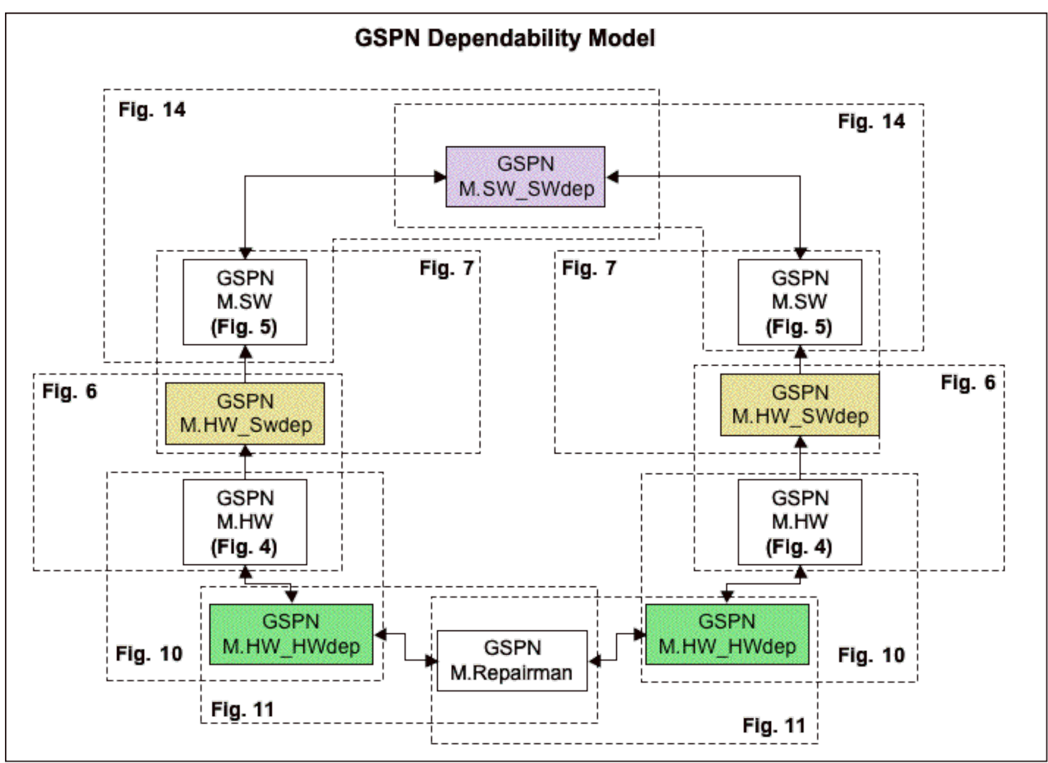



Figure 15: Merged GSPN modelling the software - architecture behaviour and error model (M.SW_SWdep)

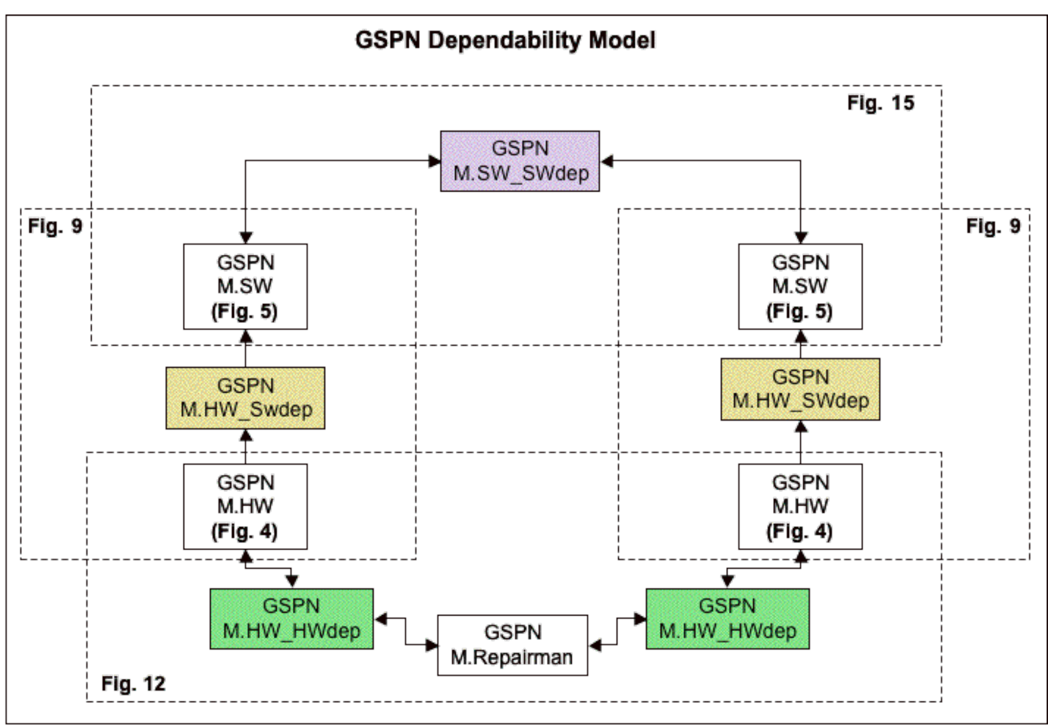
Note that, for the sake of clarity, we duplicated error states involved in transitions triggered by the propagations SW_I_Am_Restarted in Figure 15. In this way, the 16 interface GSPN transitions corresponding to SW_I_Am_Restarted propagations can be represented separately, at the bottom of this figure.

It is worth noting that the number of tokens in the GSPN corresponding to an AADL component is generally constant and equals either one or two (at most one from the architecture behavioural model and one from the error model). Occasionally, the number of tokens increases because of the transformation of Vote properties.

The GSPN of the whole duplex system is obtained by superposition of the partial GSPN models generated during the different modelling phases (M.HW_SWdep, M.HW_HWdep, M.SW_SWdep). Figure 16 is a synthesis of the figures that present the various parts of the global GSPN dependability model along this report. Figure 16-a shows which figures illustrate GSPNs corresponding to each component together with one of its dependencies while Figure 16-b shows which figures illustrate GSPNs corresponding to pairs of components together with a dependency that connects them.




a) Figures showing each component with one of its dependencies




b) Figures showing pairs of components connected through a dependency

Figure 16: Synthesis

# 5 Conclusion

This report presented an approach for system dependability modelling and evaluation using AADL and the AADL Error Model Annex The approach is illustrated on a concrete case study: a fault tolerant dynamically reconfigurable duplex system.

The aim of our approach is to ease the task of evaluating dependability measures, by hiding the complexity of classical analytical models to the end-user. This approach has two main characteristics: i) it is incremental, as it needs to support and trace model evolution and ii) it is based on model transformation, from AADL dependability models (system architecture + system error model) to GSPNs that can be processed by existing tools. Similarly to all dependability modelling and analysis approaches, our approach requires information on the system to be available. The (stepwise) construction of the AADL system architecture model and system error model is based on this information. After having built the AADL dependability model, it is transformed into a GSPN dependability model.

The duplex system case study allowed us to assess the feasibility of our approach. The difficulties encountered during this study are mainly related to the following two aspects: i) the system to be modelled is dynamically reconfigurable and ii) there are interactions between error models attached to its components and the mode model at architecture level. These two aspects are mixed, as the reconfiguration at the mode model level is performed according to what happens at the level of the error model.

Further analysis is still needed to define general transformation rules, particularly for Vote properties. The next step of the work concerns the formalisation of transformation rules in order to automate model transformation.

## References


[Arlat et al. 2005] J. Arlat, M. R. Barone, Y. Crouzet, J.-C. Fabre, M. Kaâniche, K. Kanoun, S. Mazzini, M. R. Nazzarelli, D. Powell, M. Roy, A. E. Rugina and H. Waeselynck, *Dependability Needs and Preliminary Solutions Concerning Evaluation, Testing and Wrapping*, D-32(345)-1 ASSERT deliverable, 2005.

[Betous-Almeida & Kanoun 2004] C. Betous-Almeida and K. Kanoun, "Construction and stepwise refinement of dependability models", *Performance Evaluation*, 56 (1-4), pp.277-306, 2004.

[Bondavalli et al. 1999] A. Bondavalli, I. Mura and K. S. Trivedi, "Dependability Modelling and Sensitivity Analysis of Scheduled Maintenance Systems", in *3rd European Dependable Computing Conference (EDCC-3),* (A. Pataricza, et al., Eds.), (Prague, Czech Republic), pp.7-23, Springer, 1999.

[Conquet & David 2005] E. Conquet and P. David, "Preparing the System and Software engineering of the 21st century for critical systems with the ASSERT project", in *Fifth European Dependable Computing Conference, Supplementary Volume,* (Budapest, Hungary), pp.27-32, 2005.

[Kanoun & Borrel 2000] K. Kanoun and M. Borrel, "Fault-tolerant systems dependability. Explicit modeling of hardware and software component-interactions", *IEEE Transactions on Reliability*, 49 (4), pp.363-376, 2000.

[SAE-AS5506 2004] SAE-AS5506, *Architecture Analysis and Design Language*, Society of Automotive Engineers, 2004.

# Annex 1: Proposals for the AADL Error Model Annex

Our proposals for the evolution of the AADL Error Model Annex are inspired from the dependability modelling of the fault-tolerant dynamically reconfigurable duplex system with AADL and the AADL Error Model Annex, presented in this report. We associate a priority to each of the identified issues, according to its importance. The issues are enumerated below, from the one with highest priority to the one with lowest priority.

- Occurrence properties,
- The link between the mode model and the error model,
- Vote_In and Vote_Out properties,
- Inheritance and refinements.

We consider that the first two of them are necessary while the last two are useful improvements. The four following sections present, in order, current AADL Error Model Annex specifications and some proposals concerning these four issues. Syntax proposed here is based on the syntax presented in the AADL Error Model Annex draft 0.8.

## *1 Occurrence properties*

### 1.1 Current Occurrence properties

Occurrence properties:

1) can be associated to error events and out (or in out) error propagations,
2) define *valued* probabilities or distributions,
3) are not mandatory (if an event or out propagation does not have an associated Occurrence property, the system error modelling and analyst must consider all possible values),
4) are required or not, depending on the analysis performed.

### 1.2 Our proposal

We propose to allow the user to specify unvalued parameters for Occurrence properties. The use of such parametric properties can be of interest for some sensitivity analyses that aim at finding, for example, acceptable limit values for error and repair events in a given context and for a given system. In this context, it seems to us that the possibilities offered by statement 1.1-3) is not sufficient as one may want to specify that an Occurrence property is a fixed probability or a certain kind of distribution without giving a value to the parameter. If no Occurrence property is associated to an error event or out propagation, it is impossible to state that the error event or out propagation is purely probabilistic or follows a certain type of distribution. Also, parametric Occurrence properties are of interest as they allow the construction of generic error models for generic architectures. The generic error models can be made available in a library so as they can be instantiated and reused in specific architectures later on.

We propose Syntax 1 for the error property expression for Occurrence properties. The modified parts are highlighted (with a left vertical bar).

**Syntax 1: Occurrence properties**

```
error_property_expression ::=    distribution_name [ distribution_parameters ]
        distribution_name ::= fixed | poisson | nonstandard identifier
        distribution_parameters ::= real_numeric_literal { , real_numeric_literal }*
                                  | symbolic_expression { , symbolic_expression }*

| symbolic_expression ::= symbolic_literal
|                       | 1 - symbolic_literal | symbolic_literal (+ | - | *) symbolic_literal
```

**Example 1: Occurrence properties**

```
Occurrence => fixed p applies to Fail_Stop;
Occurrence => fixed 1-p applies to Fail_Babbling;
Occurrence => poisson λ applies to Fail;
```

Note: If identical symbols are used for several Occurrence properties, then it is assumed that these properties have the same value.

# *2 Link between the mode model and the error model*

## 2.1 Current specifications concerning modes

### 5.1.1 Influences of the error model on the mode model

Error propagation paths include "an event connection to every mode transition that is labelled with an in event port that is a destination of that connection". This means that error propagations going through an in port specified in a mode transition can trigger the mode transition at the architecture level.

Also, a Vote_Transition property association declares voting and consensus protocols used for in event ports that appear in mode transition declarations. The Boolean error expression associated to a Vote_Transition on an event port is evaluated each time an (architectural) event or error propagation occurs through the event port. If it is evaluated to true, then the mode transition labelled with that event port occurs. A Vote_Event property specifies that an (architectural) event is raised when an error condition is detected by a component as a result of a voting or consensus protocol. This property must be associated to an out event or data port. Vote_Event properties are evaluated when propagations arrive to the component.

### 5.1.2 Influences of the mode model on the error model

Error state transitions labelled activate or deactivate may occur as effects of mode transitions (respectively when the component is activated or deactivated at a mode switch).

## 2.2 Our proposal

It seems to us that the causality link between the mode model and the error model is stronger in one direction (i.e., from the error model to the mode model) than in the other one (i.e., from the mode model to the error model). More concretely, error propagations or (architectural) events raised according to a Vote_Event declaration can trigger mode changes. Consequently, mode changes can occur as a consequence of a certain error state configuration or at the arrival of certain error propagations.

Conversely, the only way to express a causality link from the mode model to the error model is to declare error state transitions triggered at the activation or at the deactivation of the component at a mode switch. This kind of causality link is very poor as the mode model of a component that is always active would never be able to cause an error state transition. We can imagine that one would like to model a mode-dependent error model, meaning that the behaviour of the component in the presence of faults is different in different modes. We consider two ways of modelling this kind of causality link:

1) By allowing users to associate different error models to different modes of a component;
2) By introducing in the AADL Error Model Annex the in modes statement.

The first possibility seems harder to implement, as it needs to specify clear rules about the error model switch synchronised to the mode switch (what is the initial state after a mode switch, etc.). Also, the error model switch would be triggered by (architectural) events or error propagations that were the cause of the mode switch. A certain mode configuration would not be able to trigger the error model switch.

It seems to us that the second possibility is both easier to implement and well adapted to the current form of the AADL standard. Consequently, we propose that the in modes statement be introduced in the AADL Error Model Annex as follows.

- The in modes statement can appear in the **error model type** for declaring mode-specific error states, error events and error propagations. The mode identifiers must be declared in the component to which the error model is associated; otherwise, the declarations followed by in modes statements are ignored when processing the model. If the in modes statement is not present, then all error model type declarations are valid in all modes. The initial (inactive) error state cannot be followed by an in modes statement. If the in modes statement appears in a refinement, the newly specified modes replace the ones declared in the refined feature. The proposed syntax for error model types with in modes statements is given in Syntax 2.

**Syntax 2: Error model type (with modes)**

```
error_model_type ::=
        error model defining_error_model_type_identifier
                features { error_model_feature}+
        end defining_error_model_type_identifier ;
error_model_feature ::= error_event_spec | error_propagation_spec | error_state_spec
error_event_spec ::= defining_error_event_identifier_list : error event
                [ { error_property_association { ; error_property_association }+ } ]

                [in modes list_of_modes];
error_propagation_spec ::=
                defining_error_propagation_identifier_list : ( in | out | in out ) error propagation
                [ { error_property_association { ; error_property_association }+ } ]
                [in modes list_of_modes];
error_state_spec ::=
                defining_error_state_identifier_list : [ initial [ inactive ] ] error state
                [in modes list_of_modes];

error_property_association ::=
        error_property_identifier => error_property_expression ;
```

- The in modes statement can appear in the error model implementation to declare mode-specific error state transitions and properties. As in the case of in modes statements appearing in the error model type, the mode identifiers must be declared in the component to which the error model is associated; otherwise, the declarations followed by in modes statements are ignored when processing the model. If the in modes statement is not present, then all error model implementation declarations are valid in all modes, if the features (error states, error events and error propagations) referred to in these declarations are part of all modes. If this is not the case, the error model implementation declarations are considered to be valid in the modes where the features are valid. The same error model implementation can contain mode-specific declarations of transitions and properties and general transitions and properties. The general ones apply to modes that are not mentioned in mode-specific declarations. The proposed syntax for error model implementations with in modes statements is given in Syntax 3.

**Syntax 3: Error model implementation (with modes)**

```
error_model_implementation ::=
        error model implementation
        error_model_type_identifier . defining_error_model_implementation_identifier
        transitions ( { error_transition }+ | none_statement ) [in modes list_of_modes];
        [ properties ( { implementation_error_property_association }+ | none_statement ) ]
                [in modes list_of_modes];
        end error_model_type_identifier . defining_error_model_implementation_identifier ;
        error_transition ::=
                source_error_state_identifier { , source_error_state_identifier }*
                -[ error_transition_label ]-> destination_error_state_identifier ;
        error_transition_label ::=
                event_or_propagation_identifier { , event_or_propagation_identifier }*
        event_or_propagation_identifier ::=
              error_event_identifier | ( in | out ) error_propagation_identifier | activate | deactivate
        implementation_error_property_association ::=
                error_property_association applies to feature_identifier ;
        none_statement ::= none;
```

Example 2 shows how the in modes statement can be used. Example 2-a shows an example of component type with two in out event ports. We suppose that this component is connected to other components through these ports. Example 2-b shows an example of component implementation corresponding to the component type of Example 2-a. This component implementation declares two modes: *Normal* (the initial mode) and *Restart*. Mode changes are triggered by ports *a* and *b*. An error model is associated to this component implementation. It is declared in Example 2-c (the error model type) and Example 2-d (the error model implementation). The error model type declares one error event, *Fail*, one in out error propagation, *No_Data*, and two error states: *Error_Free* (the initial error state) and *Failed*. The error state *Failed* is only part of the mode *Normal*. The error model implementation specifies that, in mode *Normal*, the component will move from the error state *Error_Free* to the error state *Failed* either if the error event *Fail* occurs or if the in error propagation *No_Data* occurs. The component will remain in the error state *Failed* when propagating out *No_Data*. Note that even if the transitions were not declared explicitly only for the mode *Normal*, they would not be valid in mode *Restart*, as the error state *Failed*, which is part of the transitions, is not valid in mode *Restart*. Occurrence properties are also declared in the error model implementation.

**Example 2: in modes**

```
system Example
features
    a,b: in out event port;
end Example;
```

**2-a) Component type**

```
system implementation Example.basic
modes
    Normal: initial mode;
    Restart: mode;
    Restart-[a]->Normal;
    Normal-[b]->Restart;
annex Error {** Model => Basic.Nominal;}
end Example.basic;
```

**2-b) Component implementation**

```
error model Basic
features
  Fail: error event;
  No_Data: in out error propagation;
  Error_Free: initial error state;
  Failed: error state in modes Normal;
end Basic;
```

**2-c) Error model type**

```
error model implementation Basic.Nominal
transitions in modes Normal
  Error_Free -[Fail, in No_Data]-> Failed;
  Failed -[ out No_Data ]-> Failed;
properties
  Occurrence => poisson 10E-4 applies to Fail;
  Occurrence => fixed 1 applies to No_Data;
end Basic.Nominal;
```

**2-d) Error model implementation**

# 3 *Vote_In and Vote_Out properties*

## 3.1 Current Vote_In and Vote_Out properties

Vote_In and Vote_Out properties specify how error propagations arriving at a component are translated or masked before causing transitions between error states of that component. Boolean propagation mapping expressions are evaluated each time an error is propagated into a component via a feature or shared object with a Vote_In property association, and each time an error is propagated out of a component via a feature or shared object with a Vote_Out property association. Boolean propagation mapping expressions can contain error propagations and / or error states. It is not forbidden to declare Boolean expressions containing only error states.

## 3.2 Our proposal

We propose somehow to extend the semantics of Vote_In and Vote_Out properties by removing from the Error Model Annex the condition on the evaluation of Boolean propagation mapping expressions. In this way, the Boolean expressions could be evaluated when needed by specific analyses and this issue would be left open in the standard. *Boolean propagation mapping expressions* could become *Boolean mapping expressions*, as they could contain only error states and be evaluated when needed, not only when propagations occur.

With this semantic extension, one would be able to model the following behaviour. Supposing that the system is formed of two connected components, A and B, with error models associated, one can specify by using a Vote property that B can restart if and only if A is not in the error state Failed.

# *4 Inheritance and refinements*

## 4.1 Current inheritance and refinement mechanisms

Inheritance and refinement mechanisms are useful when dealing with an incremental modelling approach such as the one proposed in this report. This kind of mechanisms would ease the modeller's job during the model evolution phases by allowing him to enrich models without copying and pasting from the previous modelling phases. Then, the model evolution would be more visible to readers. The AADL Error Model Annex included refinement specifications until its draft version 0.7. Then, refinement specifications were taken out from the annex in the aim of a better integration for them later on.

## 4.2 Our proposal

We propose to integrate the inheritance and refinements in the AADL Error Model Annex in a way similar to the one specified in the AADL core standard, i.e., error model type inheritance and error model implementation inheritance.

### 5.1.3 Error model type inheritance

An error model type can extend (only one) another error model type, inheriting in this way all declarations of initial error state, initial inactive error state, other error states, error events and error propagations. If an error model type extends another error model type, then it can add new error states, error events and error propagations to the inherited ones. However, it cannot suppress any of the features declared in the inherited error model type.

Refinements are possible: a feature (an error state, error event or error propagation) in the inherited error model type can be refined to a set of features in the child error model type. If the initial (or initial inactive) error state is refined, then one and only one of the replacing error states in the child error model type will be declared as initial (or inactive initial) error model type. Error model type extensions form a hierarchy as an error model type can inherit from another error model type that inherits in its turn another error model type.

We propose Syntax 4 for error model type inheritance.

**Syntax 4: Error Model Type (inheritance)**

```
error_model_type ::=
    error model defining_error_model_type_identifier
        [extends unique_parent_error_model_type_identifier]
        features { error_model_feature}+ | {error_model_feature_refinement}+
    end defining_error_model_type_identifier ;
error_model_feature ::= error_event_spec | error_propagation_spec | error_state_spec
error_event_spec ::=
        defining_error_event_identifier_list : error event
        [ { error_property_association { ; error_property_association }+ } ]
        [in modes list_of_modes];
error_propagation_spec ::=
        defining_error_propagation_identifier_list : ( in | out | in out ) error propagation
        [ { error_property_association { ; error_property_association }+ } ]
        [in modes list_of_modes];
error_state_spec ::=
        defining_error_state_identifier_list : [ initial [ inactive ] ] error state
        [in modes list_of_modes];
error_model_feature_refinement ::= error_event_spec_refinement |
        error_propagation_spec_refinement | error_state_spec_refinement
error_event_spec_refinement ::=
        child_error_event_identifier_list: refine error event
                parent_error_event_identifier
        [ { error_property_association { ; error_property_association }+ } ]
        [in modes list_of_modes];
error_propagation_spec_refinement ::=
        child_error_propagation_identifier_list: refine ( in | out | in out ) error
                propagation parent_error_propagation_identifier
        [ { error_property_association { ; error_property_association }+ } ]
        [in modes list_of_modes];
error_state_spec_refinement ::=
        [ initial [ inactive ] ] child_error_state_identifier_list : refine [ initial [ inactive ] ]
    error state parent_error_state_identifier [in modes list_of_modes];
    error_property_association ::=
        error_property_identifier  =>  error_property_expression ;
```

*Note: This syntax is based on the* Syntax 2*, that includes the* in modes *statement in the error model type declaration.*

Example 3-a shows a basic error model type and Example 3-b shows an extended error model type based on Example 3-a.

**Example 3: Error model type inheritance**

```
error model Basic
features
  Fail: error event;
  No_Data: in out error propagation;
  Error_Free: initial error state;
  Failed: error state;
end Basic;
```

**3-a) Parent error model type**

```
error model More_complete extends Basic
features
  Fail_Stop, Fail_Babbling : refine error event Fail;
  Bad_Data : in out error propagation;
  Stopped, Babbling: refine error state Failed;
end More_complete;
```

**3-b) Child error model type**

#### 5.1.4 Error model implementation inheritance

An error model implementation can extend (only one) another error model implementation. In this case, the child error model implementation inherits the transition and property declarations of its ancestors. By default, the extended error model implementation also inherits the error model type of the parent but the modeller can explicitly associate this extended error model implementation to an error model type that inherits the error model type of the parent. Figure 17 shows these two types of association between extended error model implementations and error model types.

If the child error model implementation corresponds to the same error model type as its parent (case a) from Figure 17), new transitions between error states and triggered by error events or propagations, all declared in the same error model type, can be added in the child error model implementation. It is the modeller's job to ensure the consistence of the child error model implementation, as transitions declared in the parent error model implementation are inherited and are not overridden by newly declared transitions. New error properties can replace the ones declared in the parent error model implementation, i.e., if a property is applied to a feature in a child error model implementation, this new property overrides the one declared in the parent error model implementation.

Conversely, if the child error model implementation is associated to an error model type that inherits the error model type corresponding to the parent error model implementation (case b) from Figure 17), then the modeller can also declare transitions between error states belonging

to the extended set of error states, triggered by error events and propagations belonging to the extended set of error events and propagations.

If some source and / or destination states of transitions declared in the parent error model implementation are refined in the child error model type, then transitions between the refined states are not inherited. Transitions between the refined states must be declared in the child error model implementation. If error events and error propagations declared in the parent error model type are refined in the error model implementation, transitions triggered by them in the parent error model implementation are inherited as a set of transitions between the same error states and triggered respectively by each one of the refined error events or propagations.

New error properties can i) replace the ones existing in the parent error model implementation and ii) be associated to features from the extended error model type.

When features from the parent error model type are refined in the child error model type that is associated to the child error model implementation, the modeller should declare transitions and properties for the refined features in the child error model implementation. Otherwise, it makes no sense to associate a child error model type instead of the parent error model type to a child error model implementation.

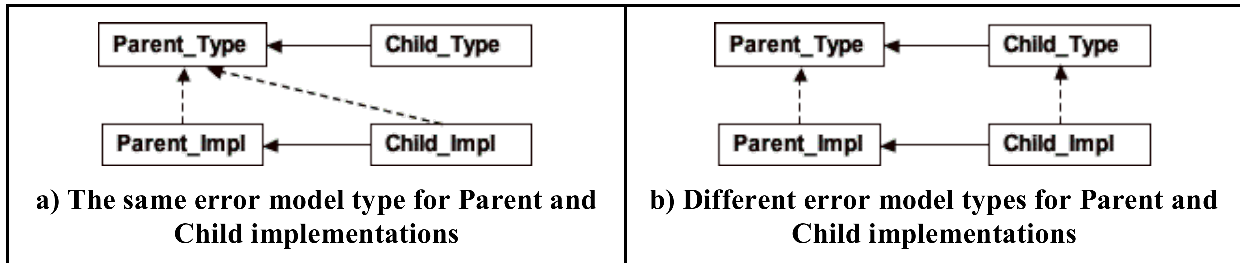



**Figure 17: Error model implementation inheritance**

We propose Syntax 5 for error model implementation inheritance.

**Syntax 5: Error model implementation (inheritance)**

```
error_model_implementation ::=
    error model implementation
    error_model_type_identifier . defining_error_model_implementation_identifier
        [extends parent_error_model_type_identifier .
            unique_parent_error_model_implementation_identifier]
    transitions ( { error_transition }+ | none_statement ) [in modes list_of_modes];
    [ properties ( { implementation_error_property_association }+ | none_statement ) ]
        [in modes list_of_modes];
    end error_model_type_identifier . defining_error_model_implementation_identifier ;
    error_transition ::=
        source_error_state_identifier { , source_error_state_identifier }*
        -[ error_transition_label ]-> destination_error_state_identifier ;
    error_transition_label ::=
        event_or_propagation_identifier { , event_or_propagation_identifier }*
    event_or_propagation_identifier ::=
        error_event_identifier | ( in | out ) error_propagation_identifier | activate |
    deactivate
    implementation_error_property_association ::=
        error_property_association applies to feature_identifier ;
    none_statement ::= none;
```

*Note: This syntax is based on the* Syntax 3*, that includes the* in modes *statement in the error model implementation declaration.*

As in the case of error model types, error model implementation extensions form a hierarchy as an error model implementation can inherit from another error model implementation that inherits another error model implementation.

Example 4-a shows a simple error model implementation that corresponds to the error model type *Basic* shown in Example 3-a.

Example 4-b shows an error model implementation that extends the one shown in Example 4-a and that corresponds to the same error model type as its parent, *Basic* (shown in Example 3-a). This child implementation replaces the value of the Occurrence property declared in the parent implementation.

Example 4-c shows an error model implementation that extends one shown in Example 4-a and that corresponds to the error model type *More_complete* shown in Example 3-b, which is a child of the error model *Basic* (shown in Example 3-a).

**Example 4: Error model implementation inheritance**

```
error model implementation Basic.Nominal
transitions
  Error_Free -[Fail, in No_Data]-> Failed;
  Failed –[ out No_Data ]-> Failed;
properties
  Occurrence => poisson 10E-4 applies to Fail;
  Occurrence => fixed 1 applies to No_Data;
end Basic.Nominal;
```

**4-a) Parent error model implementation**

```
error model implementation Basic.Enriched extends Basic.Nominal
properties
  Occurrence => poisson 10E-2 applies to Fail;
end Basic.Enriched;
```

**4-b) Child error model implementation - the same error model type**

```
error model implementation More_complete.Enriched extends Basic.Nominal
transitions
  Error_Free -[Fail_Stop, in No_Data]-> Stopped;
  Error_Free -[Fail_Babbling, in Bad_Data]-> Babbling;
  Stopped –[ out No_Data ]-> Stopped;
  Babbling –[ out Bad_Data ]-> Babbling;
properties
  Occurrence => poisson 10E-4 applies to Fail_Stop;
  Occurrence => poisson 10E-6 applies to Fail_Babbling;
  Occurrence => fixed 0.6 applies to No_Data;
  Occurrence => fixed 0.4 applies to Bad_Data;
end More_complete.Enriched;
```

**4-c) Child error model implementation - different error model type**

# Annex 2: Duplex system case study – AADL architecture

```
--
********************************************************************************
--
-- This is intended to be a VERY HIGH_LEVEL model of the duplex system:
-- Two computers and two replicas of the same software, each one running on one
computer
-- at every moment, one of the software replicas is the primary software
-- and its output is active. The other replica is the backup.Its output is not
active.
-- The fault tolerance strategy is the following:
-- The replicas switch places (backup becomes primary and primary becomes
backup)
--        - After an error in the primary software component,
--        error non tolerated by the local fault tolerance mecanisms,
--
--        - After a crash of the primary software replica, due to the failure
--        of the hardware which hosts it.
-- In the end, the new backup replica is restarted immediately if the failure
-- was a software failure or after the repair of the hardware if the failure was
-- due to a hardware failure.

-- computers are modeled as systems, so are the software replicas
-- computers and software are linked through connections
-- a connection links the 2 software replicas
--
*******************************************************************************
--
-- author: Ana RUGINA: aerugina@laas.fr - LAAS-CNRS - France --

-- a bus to support the connection between sub-systems
bus LAN_bus
end LAN_bus;

-- a simple model of a computer: a system with an out event port and an out data
port
system computer
features
     computer_to_appli_interrupt: out event port;
     computer_to_appli_data: out data port;
     computer_repairman_event: in out event port;
end computer;

system implementation computer.basic
annex Error {**
```

```
     Model => Mymodels::HW.HWSW_HWHWdep;
        **};
end computer.basic;

-- a simple software replica model with an in event port and an in data port
system software
features
     interrupt_from_computer: in event port;
     data_from_computer: in data port;
     communicate: in out data port;
     notification: in out event port;

-- we need this to model a mode switch triggered by an internal event
-- as a mode switch happens only when an event comes through a port
-- and the name of the port is used in specification of mode transition
     inp: in event port;
     outp: out event port;
end software;



-- generic implementation for the software
-- this implementation does not take into account the behaviour of the component
-- modes will be declared in implementations that inherit this generic
implementation
system implementation software.basic
annex Error {**
     Model => Mymodels::SW.HWSW_SWSWdep;
-- vote-in: when 2 transitions SW_I_Am_Dead come in => both SW are dead
     Vote_In => SW_Both_Dead when inp[SW_KO] and notification[SW_KO]
              --SW_The_Other_Is_Dead when inp and notification[SW_KO]
              --SW_I_Am_Dead when inp[SW_KO] and notification
        applies to inp, notification;
-- vote-in: when the HW is not in Repair state, it means it is perceived as OK
by the software
     Vote_In => HW_OK when not interrupt_from_computer[HW_In_Repair]
        applies to interrupt_from_computer;
-- mode transitions allowed only when the SW_KO or SW_I_Am_Restarted propagation
occurs
-- this prepares the behaviour description and could be included in both
inheriting implementations
-- instead of here, but this would be included twice
     Vote_Transition =>
          inp[SW_KO] or notification[SW_I_Am_Restarted]
        applies to inp;
     Vote_Transition =>
          notification[SW_KO] or inp[SW_I_Am_Restarted]
```

```
        applies to notification;
     **};
end software.basic;



-- we distinguish 2 implementations for the two software components belonging to
our system:
-- The difference between the two is made according to the behaviour of each
replica
-- one of them is initially in a Primary mode, the other one is initially in
Backup mode
-- both implementations inherit from the basic implementation software.basic
system implementation software.basic_primary extends software.basic
modes
     primary: initial mode;
     backup: mode;
     primary-[inp] -> backup;
     backup-[notification] -> primary;
end software.basic_primary;

system implementation software.basic_backup extends software.basic
modes
     primary: mode;
     backup: initial mode;
     primary-[inp] -> backup;
     backup-[notification] -> primary;
end software.basic_backup;

-- a sub-system of the primary-backup global system
system sub_system
features
     communicate: in out data port;
     repair: in out event port;
     notification: in out event port;
     Network: requires bus access;
end sub_system;



-- we distinguish 2 implementations for the sub-systems: one correponding
-- to the primary and the other corresponding to the backup
-- The one uses the software.basic_primary implementation and the other
-- uses the software.basic_backup implementation
system implementation sub_system.basic_primary
subcomponents
     HW: system computer.basic;
     SW1: system software.basic_primary;
```

```
connections
     data port HW.computer_to_appli_data -> SW1.data_from_computer;
     event port HW.computer_to_appli_interrupt -> SW1.interrupt_from_computer;
     event port SW1.notification -> notification;
     event port SW1.inp -> SW1.outp;
     event port HW.computer_repairman_event -> repair;
     data port communicate -> SW1.communicate;
end sub_system.basic_primary;

system implementation sub_system.basic_backup
subcomponents
     HW: system computer.basic;
     SW2: system software.basic_backup;
connections
     data port HW.computer_to_appli_data -> SW2.data_from_computer;
     event port HW.computer_to_appli_interrupt -> SW2.interrupt_from_computer;
     event port SW2.notification -> notification;
     event port SW2.inp -> SW2.outp;
     event port HW.computer_repairman_event -> repair;
     data port communicate -> SW2.communicate;
end sub_system.basic_backup;

-- the repairman is seen as a system interacting with the two computers
system repairman
features
     repair_computer: in out event port;
     Network: requires bus access;
end repairman;

system implementation repairman.basic
annex Error {**
     Model => Mymodels::Repairman.Simple;
     **};
end repairman.basic;

-- the global system
system global_system
end global_system;

system implementation global_system.basic
subcomponents
-- system1 is initially the primary while system2 is initially the backup
     system1: system sub_system.basic_primary;
     system2: system sub_system.basic_backup;
     repairman: system repairman.basic;
```

```
    LAN: bus LAN_bus;
connections
    data port system1.communicate -> system2.communicate
{Actual_Connection_Binding => reference LAN;};
    event port system1.notification -> system2.notification
{Actual_Connection_Binding => reference LAN;};
    event port repairman.repair_computer -> system1.repair
{Actual_Connection_Binding => reference LAN;};
    event port repairman.repair_computer -> system2.repair
{Actual_Connection_Binding => reference LAN;};
    event port system1.repair -> repairman.repair_computer
{Actual_Connection_Binding => reference LAN;};
    event port system2.repair -> repairman.repair_computer
{Actual_Connection_Binding => reference LAN;};
    bus access LAN -> system1.Network;
    bus access LAN -> system2.Network;
    bus access LAN -> repairman.Network;
end global_system.basic;
```

# Annex 3: Duplex system case study – AADL Error Model

```
-- this annex defines error models for components inside the duplex
system's AADL architecture model --



-- **** basic error model for the hardware **** --
-- Specifications:
-- Permanent and Temporary faults are activated following poisson
distributions.
-- The errors generated by temporary faults do not need to be handled.
They
-- will dissapear after a very short time. An error generated by a
permanent fault
-- is either detected or not detected.
-- If such an error has been detected, the hardware component is repaired
within
-- a certain time. If not, the failure is perceived after a time called
error latency.
-- After that, the repair process can start as in the case stated before.

error model HW
features
-- states
     HW_Error_Free: initial error state;
     HW_Activation_Fault, HW_Temporary_Erroneous_State,
HW_Permanent_Erroneous_State,
     HW_End_of_Error_Detection_Action, HW_Error_Non_Detected, HW_In_Repair:
error state;
-- events
     HW_Fault, HW_Perm_Fault, HW_Temp_Fault,
     HW_Error_Detection_Action, HW_Failure_Perceived,
     HW_Perm_Fault_Detected, HW_Perm_Fault_Non_Detected,
     HW_Repair_Temp, HW_Repair_Perm: error event;
-- propagations (not declared in the first modelling phase, when error
models are isolated)
-- propagations for HW-SW dependency (modelling phase 2)
     HW_KO, HW_Permanent_Non_Detect, HW_Temporary: out error propagation;
-- propagations for HW-HW dependency (modelling phase 3)
     HW_I_Am_Repaired: out error propagation;
     Repairman_I_Repair_You: in error propagation;
-- **** --
end HW;

-- **** basic error model for the software **** --
-- Specifications:
```

```
-- Faults are activated following a poisson distribution.
-- An error will be detected or not with two different rates.
-- A detected error is handled by the exception handling mechanisms.
Either effects of
-- the error are eliminated, or the software needs to be restarted
-- The effects of a non detected error can disappear after a period of
time or can
-- be perceived after a period of time.

error model SW
features
-- states
     SW_Error_Free: initial error state;
     SW_Activation_Fault, SW_End_of_Error_Detection_Action,
     SW_Error_Non_Detected, SW_Error_Detected,
     SW_End_of_Exception_Handling, SW_In_Restart: state;
-- events
     SW_Fault, SW_Detection_Action, SW_Detected, SW_Non_Detected,
     SW_Non_Detected_Disappear, SW_Non_Detected_Perceived,
SW_Error_Detected_Handling,
     SW_Error_Temp, SW_Error_Perm, SW_Restart: error event;
-- propagations (not declared in the first modelling phase, when error
models are isolated)
-- propagations for HW-SW dependency (modelling phase 2)
     HW_Permanent_Non_Detect, HW_Temporary, HW_OK, HW_KO: in error
propagation;
-- propagations for the SW-SW dependency (modelling phase 3)
-- the following two propagations are in out as both SW componennts can
send and receive them
     SW_KO, SW_I_Am_Restarted: in out error propagation;
     Tempo: error event;
-- this is just an in propagation as it is generated in a Vote_In
property.
-- It is never sent out explicitly
     Both_SW_Dead: in error propagation;
-- **** --
end SW;

-- **** error model implementation for the hardware part
-- taking into account the structural dependency between hardware and
software
-- and the repair dependency between the hardware parts**** --
-- the "repairman" sub-system is taken into account here. it must have an
error model itself

-- Temporary faults into hardware can propagate errors (with a certain
probability)
-- to the hosted software component.
```

```
-- Permanent faults into hardware cause hardware failures. Consequently, the hosted
-- software component stops. The repair and restart actions must then be synchronized
-- as the restart of software can be done only if the hardware component has been repaired.

-- The repairman is a shared ressource for the two software components.
-- Consequently, if the repairman is busy repairing one HW component while the other one fails,
-- the more recently failed system has to wait for the repairman to be free.

error model implementation HW.HWSW_HWHWdep
features
-- we extend the In_Repair state: it is formed of three states: Needs_Repair
-- (where it propagates out the I_Am_Dead event), Repairing and Repaired.
-- When the repairman comes to repair the HW component,
-- it goes in the Repairinng state which lasts for a while.
-- When it is repaired, it goes to Repaired state
     HW_Needs_Repair, HW_Repairing, HW_Repaired: error state extends HW_In_Repair;
transitions
-- transitions triggered by internal events (modelling phase 1)
     HW_Error_Free-[HW_Fault] -> HW_Activation_Fault;
     HW_Activation_Fault-[HW_Perm_Fault] -> HW_Permanent_Erroneous_State;
     HW_Activation_Fault-[HW_Temp_Fault] -> HW_Temporary_Erroneous_State;
     HW_Temporary_Erroneous_State-[HW_Repair_Temp] -> HW_Error_Free;
     HW_Permanent_Erroneous_State-[HW_Error_Detection_Action] -> HW_End_of_Error_Detection_Action;
     HW_End_of_Error_Detection_Action-[HW_Perm_Fault_Detected] -> HW_In_Repair;
     HW_End_of_Error_Detection_Action-[HW_Perm_Fault_Non_Detected] -> HW_Error_Non_Detected;
     HW_Error_Non_Detected-[HW_Failure_Perceived] -> HW_In_Repair;
     HW_In_Repair-[HW_Repair_Perm] -> HW_Error_Free;

-- transitions triggered by propagations (this should be done by extension of impl. HW.Simple)
-- **** specific transitions for the HW-SW dependency **** --
-- propagations may propagate out more than one time.
-- the arrival state is the same as the source state
     HW_Temporary_Erroneous_State-[out HW_Temporary] -> HW_Temporary_Erroneous_State;
     HW_Error_Non_Detected-[out HW_Permanent_Non_Detect] -> HW_Error_Non_Detected;
     HW_In_Repair-[out HW_KO] -> HW_In_Repair;
-- **** specific transitions for the HW_HW repair dependency **** --
```

```
     HW_Needs_Repair-[out HW_KO] -> HW_Needs_Repair;
     HW_Needs_Repair-[in Repairman_I_Repair_You] -> HW_Repairing;
     HW_Repairing-[HW_Repair_Perm] -> HW_Repaired;
     HW_Repaired-[out HW_I_Am_Repaired] -> HW_Error_Free;
-- ************************************************ --
-- properties => stochastic (and temporal) parameters for transitions
properties
-- **** properties for internal events (modelling phase 1) **** --
-- a fault occurs following a Poisson distribution
     Occurrence => poisson 10e-2 applies to HW_Fault;
-- fixed probabilities are associated with occurrence
-- of Temporary and Permanent Faults
     Occurrence => fixed 0.98 applies to HW_Temp_Fault;
     Occurrence => fixed 0.02 applies to HW_Perm_Fault;
-- if a Temporary Fault occurs, it disappears after a very short time
     Occurrence => poisson 10e+3 applies to HW_Repair_Temp;
-- The time needed by the error detection mechanisms
     Occurrence => poisson 10e+2 applies to HW_Error_Detection_Action;
-- The permanent faults are detected or not with the following
probabilities:
     Occurrence => fixed 0.75 applies to HW_Perm_Fault_Detected;
     Occurrence => fixed 0.25 applies to HW_Perm_Fault_Non_Detected;
-- if a fault is not detected it will be perceived after the error
latency:
     Occurrence => poisson 10e+4 applies to HW_Failure_Perceived;
-- the repair duration is given by:
     Occurrence => poisson 10e-1 applies to HW_Repair_Perm;

-- **** specific properties for the dependencies **** --
-- **** HW-SW dependency **** --
-- a permanent fault propagates to the software with a given probability
     Occurrence => fixed 0.65 applies to HW_Permanent_Non_Detect;
-- a temporary fault is propagated with a given probability
     Occurrence => fixed 0.85 applies to HW_Temporary;
-- when the hardware fails (HW_In_Repair), it certainly stops interacting
with the software
     Occurrence => fixed 1 applies to HW_KO;
-- **** HW-HW dependency **** --
     Occurrence => fixed 1 applies to HW_I_Am_Repaired;
-- ************************************************ --
end HW.HWSW_HWHWdep;

-- **** end HW error model implementation (HW-SW dependency) and (HW_HW
dependency)**** --

-- the error model of the repairman
```

```
-- the repairman has 2 states: Free or Working. When the repairman gets
the I_Am_Dead
-- event from a computer, it goes to the Working state. When the computer
is repaired,
-- the repairman goes to the Free state.

error model Repairman
features
-- propagations => Petri Net transions towards / from states of another
component
-- We define propagations for the repair dependency between HW comp.
     Repairman_I_Repair_You: out error propagation;
     HW_KO, HW_I_Am_Repaired: in error propagation;
-- states => Petri Net states
     Repairman_Free: initial error state;
     Repairman_Working: error state;
end Repairman;

error model implementation Repairman.Simple
transitions
     Repairman_Free-[in HW_KO] -> Repairman_Working;
     Repairman_Working-[out Repairman_I_Repair_You] -> Repairman_Working;
     Repairman_Working-[in HW_I_Am_Repaired] -> Repairman_Free;
-- properties => stochastic parameters for transitions
properties
     Occurrence => fixed 1 applies to Repairman_I_Repair_You;
end Repairman.Simple;
-- **** end basic error model Repairman **** --


-- **** error model implementation for the software
-- taking into account the dependency between hardware and software
-- and the dependency between the two software components**** --

-- Temporary faults into hardware can propagate errors (with a certain
probability)
-- to the hosted software component.
-- Permanent faults into hardware cause hardware failures. Consequently,
the hosted
-- software component stops. The repair and restart actions must then be
synchronized
-- as the restart of software can be done only if the hardware component
has been repaired.

-- The software-software dependency deals with the mode switch between
replicas if
-- the primary fails.
```

```
error model implementation SW.HWSW_SWSWdep
features
-- we refine even more than for the HW-SW dependency
-- the state SW_In_Restart. it is needed for the repair-restart and
reconfiguration process
-- we also extend the state SW_In_Restart. it is needed for the repair-
restart process
     SW_Needs_Restart, SW_Now_Restart, SW_Restarting,
SW_Restarting_Both_Dead, SW_Both_Dead, SW_Restarted: error state refines
SW_In_Restart;
transitions
-- transitions triggered by internal events
     SW_Error_Free-[SW_Fault] -> SW_Activation_Fault;
     SW_Activation_Fault-[SW_Detection_Action] ->
SW_End_of_Error_Detection_Action;
     SW_End_of_Error_Detection_Action-[SW_Detected] -> SW_Error_Detected;
     SW_End_of_Error_Detection_Action-[SW_Non_Detected] ->
SW_Error_Non_Detected;
     SW_Error_Detected-[SW_Error_Detected_Handling] ->
SW_End_of_Exception_Handling;
     SW_Error_Non_Detected-[SW_Non_Detected_Disappear] -> SW_Error_Free;
     SW_Error_Non_Detected-[SW_Non_Detected_Perceived] -> SW_In_Restart;
     SW_End_of_Exception_Handling-[SW_Error_Temp] -> SW_Error_Free;
     SW_End_of_Exception_Handling-[SW_Error_Perm] -> SW_In_Restart;
     SW_In_Restart-[SW_Restart] -> SW_Error_Free;
-- **** specific transitions for the dependencies **** --


-- **** HW-SW dependency **** --
-- if the software is error free, the propagation leads it in the
SW_Err_State state.
-- the error is then processed as the internal ones
-- if the software is in another state (internal SW error followed shortly
by a propagation)
-- we consider that the HW fault cannot propagate (it is a very unlikely
case)
     SW_Error_Free-[in HW_Temporary] -> SW_Activation_Fault;
     SW_Error_Free-[in HW_Permanent_Non_Detect] -> SW_Activation_Fault;
-- if the hardware is KO, the software moves to SW_Needs_Restart state
     SW_Error_Free-[in HW_KO] -> SW_Needs_Restart;
-- additional state Needs_Restart needed because 2 events
-- (a propagation and an event cannot be attached to the same transition
between two states)
     SW_Activation_Fault-[in HW_KO] -> SW_Needs_Restart;
     SW_Error_Detected-[in HW_KO] -> SW_Needs_Restart;
     SW_Error_Non_Detected-[in HW_KO] -> SW_Needs_Restart;
-- ............................................
-- to see how to represent better SW_Needs_Restart -> SW_Error_Free IF HW
```

```
NOT IN HW_In_Repair STATE
     SW_Needs_Restart-[in HW_OK] -> SW_Restarting;
     SW_Restarting-[SW_Restart] -> SW_Error_Free;

-- **** SW-SW dependency **** --
     SW_Needs_Restart-[out SW_KO] -> SW_Needs_Restart;
     SW_Needs_Restart-[Tempo] -> SW_Now_Restart;
     SW_Needs_Restart-[in Both_SW_Dead] -> SW_Both_Dead;
     SW_Now_Restart-[in HW_OK] -> SW_Restarting;
     SW_Both_Dead-[in HW_OK] -> SW_Restarting_Both_Dead;
     SW_Restarting_Both_Dead-[SW_Restart] -> SW_Restarted;
     SW_Restarting_Both_Dead-[in SW_I_Am_Restarted] -> SW_Error_Free;
     SW_Restarted-[out SW_I_Am_Restarted] -> SW_Error_Free;
-- ************************************************ --
-- properties => stochastic (and temporal) parameters for transitions
properties
-- **** properties for internal events (modelling phase 1) **** --
-- a fault occurs following a poisson distribution
     Occurrence => poisson 20e-1 applies to SW_Fault;
-- The error detection mechanisms need some time
     Occurrence => poisson 10e+2 applies to SW_Detection_Action;
-- fixed probabilities associated with detection and non detection
     Occurrence => fixed 0.7 applies to SW_Detected;
     Occurrence => fixed 0.3 applies to SW_Non_Detected;
-- The effects of a non detected error can dissapear after a while or be
perceived
     Occurrence => poisson 10e+10 applies to SW_Non_Detected_Disappear;
     Occurrence => poisson 10e+6 applies to SW_Non_Detected_Perceived;
-- Exception handling mechanisms need some time
     Occurrence => poisson 10e+2 applies to SW_Error_Detected_Handling;
-- The error is recovered (with a given probability) or
-- the software must be restarted (with the complementary probability)
     Occurrence => fixed 0.98 applies to SW_Error_Temp;
     Occurrence => fixed 0.02 applies to SW_Error_Perm;
-- The restart takes some time
     Occurrence => poisson 10e+2 applies to SW_Restart;

-- ** properties for out propagations and events in SW-SW dependency **--
     Occurrence => fixed 1 applies to SW_KO;
     Occurrence => fixed 1 applies to SW_I_Am_Repaired;
     Occurrence => poisson 10e+10 applies to Tempo;
end SW.HWSW_SWSWdep;
-- ** end SW error model implementation (HW-SW and SW-SW dependency) **--
```