ESPRIT Project 27439

H I D E

High-Level Integrated Design Environment for Dependability

On high-level dependability modeling in HIDE

István Majzik
TUB-DMIS, Budapest, Hungary, majzik@mit.bme.hu
CNR-CNUCE, Pisa

Andrea Bondavalli
C.N.R., Ist. CNUCE, Pisa, Italy, a.bondavalli@cnuce.cnr.it
CPR-PDCC, Pisa

September 17, 1998

HIDE Document Nr. HIDE/T1.2/PDCC/4/v1

Abstract

In this report an approach to automatic reliability, availability and safety modeling is proposed. Main features of the approach are as follows: (i) modeling is based on high-level UML descriptions, thus it can be used also in the early phases of the design process when detailed behavioral models are not available, (ii) no assumption is made on fault activation and repair processes, (ii) the "uses service of" and "is composed of" hierarchies of hardware and software elements are taken into account.

1 Introduction

Dependability model of a system (composed of elements) consists of the following general parts: the *fault activation processes* which model the fault occurrence in system elements and results in *basic events*, the *propagation processes* which model the consequences of basic events and results in *derived failure events* and the *repair processes* which model how basic or derived events are removed from the system. This overall structure of the dependability model is shown in Figure 1. The failure of a system is one of the derived events in this model. Note that repair means here a general service restoration (automatic service restoration if underlying faults disappear; explicit diagnosis, repairing or replacing of hardware; restoring the state and re-integration of software etc.).

The fault activation processes are determined by environmental conditions, and physical or computational properties of the elements of the system. The propagation processes are influenced by the structure of the system (e.g. interactions, redundancy, fault tolerance schemes). The repair processes are determined by the (physical or) computational policy implemented in the system.

In our case, the dependability model should be based on a system specification given in UML. Since an UML specification does not cover all non-functional aspects required for dependability modeling (like failure characteristics of model elements), we have to "extend" the specification in order to be able to construct the dependability model, i.e. define the basic and derived events, propagation, failure and repair processes.

In this document, the dependability model is constructed by introducing an intermediate model. The model elements and relations in UML are mapped to elements and relations of the intermediate model, thus defining its syntax. The semantics of the intermediate model should be defined (i) by the extensions attached to the UML model, (ii) utilizing practical experience, (iii) adopting assumptions, characteristics and properties that are usually used by dependability modelers of hardware and software systems (these issues were described in a companion document [5]).

By defining the intermediate model, we can fix (i) the set of basic events, (ii) the propagation processes, (iii) the set of derived events, (iv) the target points of the fault activation processes (the basic events) and finally (v) the target points of the repair processes (basic events as well as a subset of derived events). The question of fault activation and repair processes is left open intentionally, it will be discussed separately.

Accordingly, the target dependability model (in our case, a timed transition Petri-net) is generated in two steps. First the UML model is mapped to the intermediate model, then the

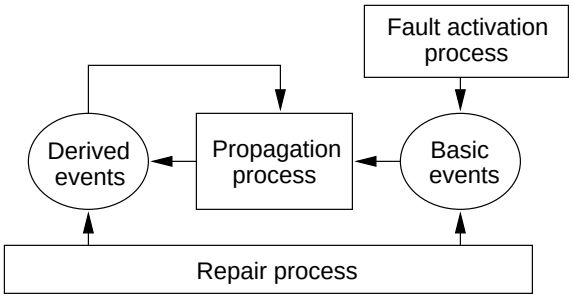

Figure 1: General parts of a dependability model

dependability model of this intermediate model will be generated.

This approach seems to have the following advantages:

- Some peculiarities of UML (package hierarchy, processes, composite objects and nodes, different types of dependencies etc.) can be resolved resulting in a simple and flat model. Moreover, hierarchical and modular modeling can be partially supported by the clever mapping from UML model elements to elements of the intermediate model (to be described later on).
- The second step of the transformation can be defined more easily, based on the limited and well-defined set of elements and relations of the intermediate model. Moreover, the formalism of the target model and the more technical points of the dependability model generation (submodels, implementation) can be changed without re-defining the first step.
- The second step of the transformation (from the intermediate model to the dependability model) can be defined without fixing the syntax of the necessary extensions to be used in the UML model. The model elements, their role and failure semantics are hidden by the intermediate model.

Note, however, that the only goal of the intermediate model is to describe the transformation itself in a more easy-to-use and convenient way. In the final definition of the transformation as well as in its implementation this intermediate step is not necessary.

In Section 2 the subset of UML on which we focus during the dependability modeling is described. Section 3 presents the syntax and "semantics" (characteristics from the dependability point of view) of the intermediate model. Section 4 describes the mapping of elements of the UML subset to the intermediate model. Section 5 and Section 6 describe how a dependability model is constructed based on the intermediate model. Section 7 proposes some refinements which could be elaborated, Section 8 discusses some prerequisite steps of dependability modeling.

2 A subset of UML

We concentrate on high-level, structure-oriented dependability modeling of systems containing software and hardware. The basic failures we take into account at the moment are the failures of objects and nodes. (The characterization of these failures may require lower level modeling, e.g. at the level of states, actions and events. These submodels might be integrated with our model.)

The necessary information is described by the structure level diagrams of UML, i.e. the use case, class, object, component and deployment diagrams.

Behavioral diagrams (sequence, collaboration, statechart diagrams) are analyzed only to derive how the failures of objects and nodes lead to the failure of a (sub)system. It means that behavioral descriptions are not utilized (yet) to construct the lower-level dependability models of system elements.

3 The intermediate model

The intermediate model is a general model of a system composed of multiple elements. The structure of the intermediate model is inspired by the approach presented in [4]. For our purposes, we slightly modify that model. We use a more reduced hierarchy and, for the sake of convenience, we distinguish software and hardware elements.

3.1 The model elements and their relations

The following elements of the intermediate model are distinguished:

- Nodes. Hardware elements are referred to as nodes. They are physical entities providing services (processing resources) to other nodes or software elements. Nodes may have state (e.g. configuration data determining the mode of operation).

- Objects. Software elements are referred to as objects. They provide services for other objects and use services of another objects or nodes. Objects may have state (variables).

  (The notion of objects and nodes is used in order to be "consistent" with the notation used in UML. Note, however, that objects in the intermediate model may represent not only UML objects, but also packages, processes, composite objects etc., a detailed discussion will follow in Section 4.)

- Packages. Packages are composite elements consisting of objects or nodes (but not both). Packages are not physical entities, they only represent the logical grouping of objects or nodes. We will use packages to describe sets of objects (nodes, respectively) which implement a redundancy scheme.

  More precisely, we adopt the following approach, which leads to a straightforward definition of packages:

  An element (here an object or a node) is redundant if its service can be delivered by an other element in a coordinated and automatic way, without the interaction of the clients. Accordingly, operation of redundant elements presumes the existence of a coordinator (called here controller). Redundant elements are used according to the class-level redundancy approach. A given service is provided by a set of redundant elements, which are coordinated by a controller. The service is available through the controller of the scheme, the redundant elements can not be used separately. An element is participant in a single redundancy scheme only.

  According to this approach, a package is a redundancy scheme consisting of a controller and redundant elements. Other, non-redundant elements which do not belong to the scheme can not be included.

The following relations are defined among the elements:

- "Uses service of": An element may use the service of other elements. It denotes a strict dependency. If an element providing the service fails then also the element which uses that service will fail.

  In our model, objects use the services of other objects, nodes or packages. Nodes use the services of another nodes or packages made up of nodes. Actors (i.e. human users

or external systems which interact directly with the system under investigation) use the services of objects and packages consisting of objects.

A special case of this relation is when an object uses the service of a node or a package of nodes. For the sake of convenience, this relation is referred to as "is deployed on".

- "Is composed of": A composite element is composed of other elements.

  In our model, a package is composed of objects or nodes. From the point of view of dependability modeling, the packages (thus the "is composed of" relations) denote non-trivial dependencies.

- "Interacts with": An element may interact with other elements in order to provide a service. This relation is handled in our model "by default", assuming that all elements are involved in providing the system service, so there is some interaction among them, not defined explicitly in the model.

Based of the relations above, following the approach of [4], two kinds of hierarchy can be defined:

- Hierarchical decomposition based on the "is composed of" relation. In our model, hierarchy is limited to packages which can not contain other packages (but objects in the package may use the service of another packages). Optionally, packages could be defined hierarchically, but now the plain model seems to fit the purposes of dependability modeling.

- Layering, based on the "uses service of" relation. A layer is a set of elements that are at the same level of the hierarchy defined by this relation, i.e. elements of the i-th layer provide services only for the $i-1$-th layer and use service of only the $i+1$-th layer. The uppermost layer is characterized by the fact that it will provide direct service(s) for the actor(s). The lowermost layer is the set of hardware nodes providing basic services without using the service of another nodes.

  It depends on the structure of the system, whether multiple layers can be distinguished or not. In the default case, two layers are distinguished: software elements are on the upper layer and hardware elements are on the lower layer.

The elements of the intermediate model and the possible relations of them are depicted in Figure 2.

3.2 Failure and repair characteristics

Failure and repair characteristics of the elements of the intermediate model are defined in the following[1]. By this analysis, the set of basic and derived events as well as parts of the propagation processes will be defined.

We adopt the following assumptions:

- The execution model is continuous, or demand-driven with a given repetition rate. The input trajectory is random and there is no correlation among successive inputs.

[1] Remember that objects usually represent UML objects, nodes represent UML nodes.

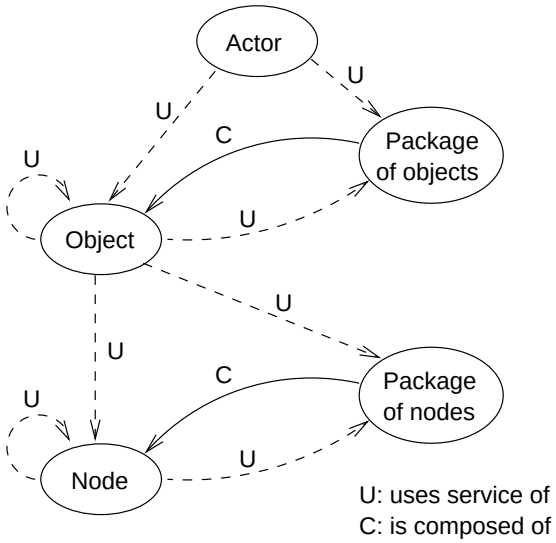

Figure 2: Intermediate model elements and their relations

- Solid software failures are not taken into account (assuming that they were removed before execution by a thorough debugging and fault removal). According to the random input trajectory, activation of software faults is temporary.
- There are no failures which compensate the effects of other ones.

Up to this point, the repair process was considered as a general service restoration. From here, we distinguish automatic (or implicit) and explicit repair.

- Implicit repair: "Repair" of a fault is implicit if it disappears after activation (transient hardware faults and all software faults). Repair of a failure is implicit if it disappears as soon as the underlying faults and failures have been repaired. Stateless objects and nodes are repaired in this way.
- Explicit repair refers to the actions that are planned and scheduled by the designer. Explicit repair may remove (permanent) faults from the system or restore the service of objects or nodes.

The failure characteristics of nodes and objects depend on the fact whether they belong to a package or not. If a node (object) belongs to a package then the distinction of separate and common mode failures is important. For the sake of simplicity of the description, we refer to faults of a node (object) leading to separate failures of that node (object) as independent faults, and faults of a set of nodes (objects) that lead to common mode failures as related faults[2].

The failure and repair characteristics of the model elements are defined as follows.

- Nodes which do not belong to a package. A failure of a node is caused by the events (in OR relation) described below:

[2]In [3] it is argued that also independent faults may lead to common mode failures.

- An activated (transient or permanent) fault of the node which leads to a failure. It is characterized by a fault activation process assigned to the node.
- Failure of an other node which is used by this node.
- Failure of a package of nodes which is used by this node.

Repair of a node (characterized by a repair process) is refined as follows. If the underlying faults and failures are repaired (transient faults disappear automatically, while permanent faults require explicit repair), then the node may remain still faulty, since the state of the node might have been damaged. Thus, the repair process should handle not only the faults of the node but also the failure (i.e. restoring the state). Of course, if some underlying failures are not repaired yet then the failure remains in spite of this latter action.

Accordingly, the node is characterized by its *self-recoverability*, a probability that the implicit repair is successful, i.e. the failure of the node disappears if the underlying faults are repaired. If the node is stateless, then this probability is one.

- Nodes belonging to a package. In this case, a clear distinction should be made between separate and common mode failures.

  Thus, instead of the (general) node failure described above, these nodes are characterized by separate and common mode failures as follows (note that these nodes are not used by another nodes):

  - A separate failure of a node is caused by the events (in OR relation) described below:
    - An activated independent fault of the node. It is characterized by a fault activation process.
    - Failure of an other node which is used by this node.
    - Failure of a package of nodes which is used by this node.
  - Common mode failures of nodes: Each (meaningful) subset of the set of nodes belonging to a package may have related faults that, when activated, lead to common mode failures. Related faults are also characterized by fault activation processes. It is assumed that common mode failures induced by shared nodes in the "uses service of" hierarchy are also covered by these processes.

  Repair of a separate node failure is characterized by a repair process (similarly to general node failures). Common mode failures are temporary ones, i.e. they disappear if the underlying faults are deactivated. However, they may lead to permanent separate failures of the nodes involved in the common mode failure. This case is characterized also by the self-recoverability of the nodes.

- Objects which do not belong to a package. The following events (in OR relation) cause a failure of an object:

  - An activated software fault of the object itself. It is characterized by a fault activation process.
  - Failure of an object or package of objects which is used.

– Failure of a node or package of nodes on which the object is deployed.

From the point of view of repair, a software fault disappears automatically (implicit repair). However, the state of the object may be damaged due to an activated software fault as well as an underlying failure. Similarly to the nodes, self-recoverability of an object is a probability that the implicit repair is successful, i.e. the failure of the object disappears if the software faults disappear and all of the underlying failures are repaired.

Explicit repair is applied to handle directly the failure of the object. However, it may remain active in spite of the explicit repair if there is an underlying failure which was not repaired.

- Objects belonging to a package. In this case, a distinction is made between separate and common mode failures.

  – Separate failure of an object. It is caused by the following events (OR relation):
    * An activated independent software fault of the object itself. It is characterized by a fault activation process.
    * Failure of an other object or package of objects which is used by this object.
    * Failure of a node or a package of nodes on which it is deployed.
  – Common mode failure of objects. Each (meaningful) subset of the set of objects belonging to a package may have related faults that, when activated, lead to common mode failures. They are characterized by a fault activation process. It is assumed that common mode failures induced by shared nodes or shared objects in the "uses service of" hierarchy are also covered by this process.

  As before, software faults disappear automatically (implicit repair). Separate object failures are repaired in the same way as the object failures described before (characterized by the recoverability). Common mode failures disappear as the underlying fault is deactivated (implicit repair), but they may lead to permanent separate failures of the objects involved, in the same way as in the case of separate failures.

- Package. Failure of a package means that the controller(s) of the package are not able to provide the service required. Since the packages represent redundancy schemes, the failure of a package is not the usual OR relation meaning that if any of the elements is faulty then the package is faulty. Instead, the failure of a package is characterized by a complex submodel. (Usually, the failure of a package can be described as a Boolean combination of the failures of the elements belonging to the package, thus a fault tree can be constructed [2]. In the following, let us refer to the dependability submodel of a package as a fault tree).

  – Packages consisting of objects: Underlying events are separate and common mode failures of objects belonging to the package.
  – Packages consisting of nodes: Underlying events are separate and common mode failures of the nodes belonging to the package.

  Repair of packages is implicit, i.e. their faulty $\rightarrow$ non-faulty state transition is a consequence of the repair actions performed on the elements included in the fault tree. The repair process of an element also covers its re-integration.

- System. A system is faulty if at least one of the elements at its uppermost layer is faulty (OR relation). Elements of the lower layers are taken into account through the "uses service of" and "is deployed on" relations.

3.3 Safety related characteristics

Safety modeling requires additional considerations. We adopt the usual assumption that undetected element failures (at the uppermost level) lead to catastrophic failure of the system. Accordingly, here we should concentrate on characteristics related to undetected failures of nodes, objects, packages and of the system.

In comparison with the characteristics used for availability modeling, fault characteristics remain the same: activated (general, independent or related) hardware faults and activated (general, independent or related) software faults are used in the same way. However, instead of failures we will use *undetected failures* to characterize the nodes, objects and packages. They will be derived applying internal (local) and external error detection coverage parameters as follows.

Internal error detection parameters characterize the ability of an element to detect its errors that are caused by (local) faults or underlying failures which were not detected. Similarly, external error detection coverage corresponding to an element characterizes the ability of an other element (monitor) to detect errors of the element which were not detected yet (by the internal mechanism). Note that these coverages are given relative to undetected errors, i.e. the "overlapping" of internal and external detection is eliminated by the parameter assignment[3]. Moreover, multiple external error detectors can be assigned to an element; in this case they are characterized by separate error detection coverages related to the given element.

Accordingly, the following modifications apply:

- Nodes. Underlying undetected failures (determined by the "uses service of" relations) and local faults (in OR relation) result in errors of the node. These errors may be detected by the internal mechanism (characterized by the internal error detection coverage) or an external mechanism (characterized by external error detection coverage). The errors which are not detected lead to undetected failures of the node.

- Objects. The similar considerations apply as for nodes.

- Packages. Here special fault trees (usually different from the ones described in the previous section) are used to combine the underlying (separate or common mode) undetected failures to get errors of the package, which, if undetected by the controller (internal detection) or by external monitors (external detection), become undetected failures of the package.

- System. Underlying undetected failures are combined (OR relation) to get catastrophic failure of the system.

Repair processes can be applied in the same way as described in the previous section.

[3]We could say that (i) detected underlying failures are explicit signals that are processed separately, preventing the "normal" computation and error detection in the element and similarly (ii) if an error is detected internally then it results in an explicit error signal which is not checked by the external monitors

4 From UML to the intermediate model

Table 1 sketches the translation from UML model elements to the intermediate model[4].

Note that the hierarchical composition of UML model elements and the corresponding mapping to the elements of the intermediate model enables a modular, hierarchical approach. E.g. a use case or a package can be mapped to a single object if we want to abstract from its internal details. However, it can also be mapped to a set of objects/nodes if a more detailed analysis is necessary. The mapping of UML elements to the elements of the intermediate model hides the level of resolution selected and provides a uniform "interface" towards the generation of the dependability model.

UML element or relation	Intermediate element or relation
Object	Object
Composite object	"uses service of" sub-objects
Composition of objects	"uses service of" sub-objects
Association	"interacts with"
Generalization	-
Dependency "uses"	"uses service of" objects
Node	Node
Composite node	"uses service of" sub-nodes
Composition	"uses service of" sub-nodes
Association	"uses service of" nodes
Package	Package "is composed of" elements (redundancy)
	Set of elements (no redundancy, content is specified)
	Object (content is unspecified)
Dependency	"uses service of" package/elements
Component	Set of objects (if its content is specified)
	Object (if its content is unspecified)
Dependency	"uses service of" objects
Use case	Set of elements (if its content is specified)
	Object (if its content is unspecified)
Generalization "uses"	"uses service of" elements

Table 1: Translation from UML to the intermediate model

5 From the intermediate model to availability and reliability models

The general parts of the dependability model are reviewed on the basis of the intermediate model. Then the corresponding submodels in the target formalism (Petri-nets) are described.

[4]It should be elaborated in a more precise way, by referring to metamodel elements and relations among them.

5.1 General parts of the dependability model

The failure/repair characteristics of the elements of the intermediate model determine the basic events, derived events, propagation processes, fault activation and repair processes used in the dependability model.

- Basic events. Basic events are the activated faults of objects or nodes as follows.
  - Activated hardware fault of a node. It is characterized by a fault activation process and an explicit repair process.
  - Activated independent (related, respectively) hardware fault of a node. It is characterized by a fault activation process and an explicit repair process.
  - Activated software fault of an object. It is characterized by a fault activation process and an implicit repair process.
  - Activated independent (related, respectively) software fault of an object. It is characterized by a fault activation process and an implicit repair process.
- Propagation processes. They are used to compose the basic events in order to get the derived events.
  - OR relations. Each node and object induces (through the "uses service of" and "is deployed on" relations) an OR relation.
  - Fault tree. Each package induces (through the "is composed of" relations) a fault tree.
- Derived events. Derived events are composed of basic events and other derived events using one of the propagation processes described above.
  - Failure of a node. An OR relation is used, the events are the activated hardware fault of the node itself and the derived events determined by the "uses service of" relations.
  - Separate failure of a node. An OR relation is used, the events are the activated independent hardware fault of the node itself and the derived events determined by the "uses service of" relations.
  - Common mode failure of a set of nodes. A degenerate OR relation is used, since the single event is the activated related hardware fault of the nodes in the set.
  - Failure of a package consisting of nodes. A fault tree is used, the events are the derived events (separate and common mode failures) determined by the "is composed of" relations.
  - Failure of an object. An OR relation is used, the events are the activated software fault of the object itself and the derived events determined by the "uses service of" and "is deployed on" relations.
  - Separate failure of an object. An OR relation is used, the events are the activated independent software fault and the derived events determined by the "uses service of" and "is deployed on" relations.

  - Common mode failure of a set of objects. A degenerate OR relation is used, the event is the activated related software fault of the objects in the set.
  - Failure of a package consisting of objects. A fault tree is used, the events are the derived events (separate and common mode failures) determined by the "is composed of" relations.
  - Failure of the system. An OR relation is used, the events are the derived events of the elements (objects, packages) of the uppermost layer.

- Fault activation processes. They are assigned to the basic events. They are not fixed in the model, since numerous types of fault activation processes are possible depending on the system and environment to be modeled. Usually, independence of basic events is assumed and the exponential approach is adopted, this way, each basic event is characterized by a constant rate. (In the next phase of the project, within reasonable limits, this assumption should be released.)

- Repair processes. Implicit repair processes are used in the following cases:

  - Software faults.
  - Failures of packages. Practically, the repair process of a package is represented by the complementary of the fault tree corresponding to the propagation process. (The propagation process keeps track of the failed elements while the repair process keeps track of the repaired elements.)
  - Failures of stateless nodes and objects, common mode failures and system failures. Here, similarly, the complementary of the OR relation corresponding to the propagation process is used, i.e. an AND relation of the underlying events.
  - Failures and separate failures of objects and nodes with state. In this case, the self-recoverability parameter determines whether an implicit repair is successful (restores the service) or not.

  Explicit repair processes are used in the following cases:

  - Remove hardware faults from the system. Since transient and permanent faults are not distinguished in this document (to avoid mixing of time and logic dimensions of events in the propagation process), this way an explicit repair process is required in general.
  - Removing failures of objects and nodes with state. In this case, the explicit repair process is used to restore the service. It can be scheduled independently of the implicit repair process or even triggered by the unsuccessful implicit repair (detection of the permanent failure).

  Numerous policies of explicit repair are possible for the hardware and software layers. As an illustration, we list the ones presented in [6] for the hardware layer:

  - Repair without dependencies (each node is repaired independently from the others, in parallel).
  - First-come-first-served repair (nodes are repaired by shared facilities in the order of failure).

- Preemptive Resume Priority repair (nodes are repaired by shared facilities in the order of their priorities; a high priority node preempts the repair of a low priority one.).
- Non-preemptive Priority repair (nodes are repaired by shared facilities in the order of their priorities, but there is no preemption).
- Processor Sharing repair (each failed node perceives the repair facility to be slowed by a given factor).

In the software layer, objects may be repaired, for example, independently; or objects belonging to the same packages, or being deployed on the same node etc. could be repaired together.

5.2 Subnets of the target model

The above described "compositional" nature of the events (as the event representing the system failure is composed of derived events, which themselves are composed of derived events and basic events) and the separation of propagation, failure and repair processes suggests a clear decomposition of the target Petri net into subnets. The interfaces among these subnets are the (basic and derived) events. For the sake of the easy construction and composition of the subnets, each event e is represented by a pair of places F_e and H_e: a token in F_e means that the event is active (i.e. a failure of an element has occurred), a token in H_e means that the event is inactive (there is no failure). The sum of tokens in these two places is always one.

The subnets are defined as follows:

- The fault activation processes corresponding to the basic events are represented by a subnet. In the case of a basic event e, this subnet moves the token from H_e to F_e. Assuming independent basic events, this subnet is decomposed into a set of subnets each representing a fault activation process of a basic event and consisting of a single timed transition.

- Similarly, the explicit repair processes are represented by a subnet. Since the repair processes corresponding to different events are not necessarily independent (e.g. shared repair), this way this subnet might not be decomposed into independent subnets corresponding to the different events. Further work is required to develop the subnets corresponding to different repair policies. (Note that the subnets developed for different repair policies in [6] can be re-used without modifications.)

- Subnets representing implicit repair processes of software faults. For each event e, a timing transition is used that removes a token from F_e and puts it to H_e.

- Propagation processes and implicit repair. OR relations with implicit or conditional implicit repair, fault trees with implicit repair and general propagation processes (when the derived event is not a simple Boolean combination of underlying events) can be distinguished.

  - The subnet representing and OR relation and the corresponding implicit repair process is depicted in Figure 3. If the implicit repair is conditional (i.e. its success is determined by the self-recoverability parameter), then the subnet presented in Figure 4 is used.

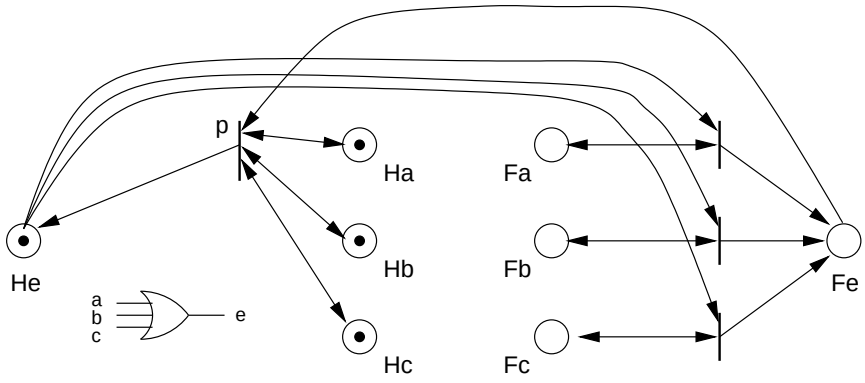

Figure 3: Subnet representing OR relation with implicit repair

- The subnet representing a fault tree and the corresponding implicit repair process is constructed by an algorithm which is similar to the one presented in [6] (but not the same, since the resulting nets are different).

  The algorithm starts from the top gate of the tree. The output event of this gate is represented by existing interface places. If an input event of this gate is a basic event then it is also represented by existing interface places. If an input event is an intermediate one (used only in this fault tree) then the corresponding pair of places is created. The transitions and the flow relation are constructed depending on the type of the gate as Figure 5 shows (each gate is assigned a subnet representing the gate and its complementary part). Then the algorithm is called recursively for the lower level gates of the fault tree. An example is presented in Figure 6 (here failure and repair processes of the basic events are represented by timed transitions).
- If the propagation process is not a fault tree, then a general subnet can be used which is defined in the FT library (attached to the redundancy scheme used by the designer) or derived automatically using the behavioral description of the package. In order to include this subnet in the dependability model, it should follow the interface rules.

Note that the subnets representing propagation processes do not change the marking of underlying events.

Examples of the composition of subnets are presented in Figure 7, where four derived events are shown.

The system has a failure if there is a token in the place F_{system} representing the derived event "system failure". The analysis of the net is as follows.

- Reliability modeling. If there is a token in F_{system} then the computation of the net can be terminated. In GSPN, this can be implemented by using inhibitor arcs from this place to all timed transitions. In SAN, a halting condition can be formulated. The reliability function can be computed by transient analysis regarding the marking of F_{system}.
- Availability modeling. In this case no halting condition is used. The availability function

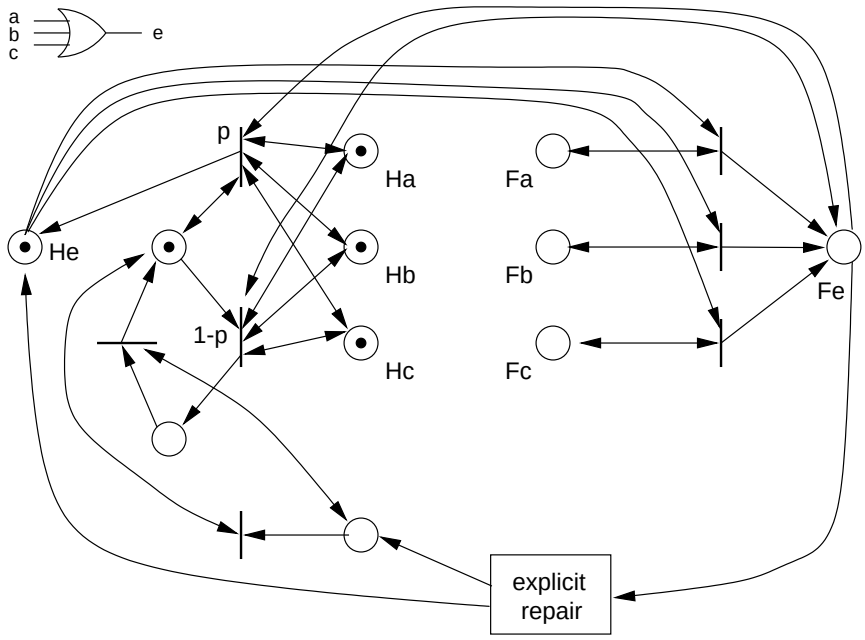

Figure 4: Subnet representing OR relation with conditional implicit repair

can be computed by transient analysis regarding the marking of F_{system}. Asymptotic availability can be computed by steady-state analysis.

6 From the intermediate model to the safety model

Safety models are derived in a similar way as in the case of availability models. The basic events correspond to faults, the subnets representing the fault activation processes and repair processes are the same as the subnets in the reliability model. The differences can be summarized as follows:

- Derived events represent undetected failures of elements.
- In the propagation process, the same relations are used to derive errors, and then, additionally, internal and external error detection is applied to derive the undetected failures of the element.

  Accordingly, the same subnets as used in availability models are connected to the interface places (us usual, pairs of places) representing the errors. These interface places are connected to a (series of) subnet(s) representing error detection coverage(s). The last subnet of this series is connected to the pair of places representing the derived event, i.e. the undetected failure.

  The subnet used to represent error detection coverage is depicted in Figure 8. An example propagation process is shown in Figure 9.

A system has catastrophic failure if there is a token in the place representing the derived event "catastrophic system failure". If there is a token in this place, then further computation of

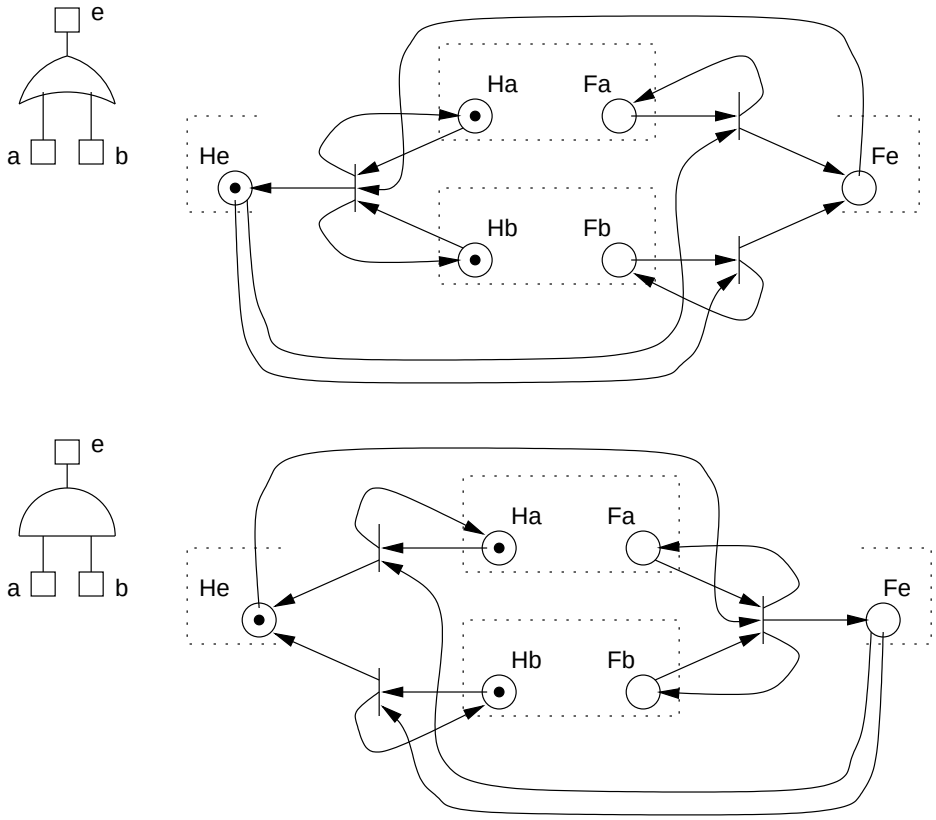

Figure 5: Subnets representing gates in the fault tree

the net can be terminated by using inhibitor arcs (GSPN) or halting condition (SAN). The safety function can be computed by transient analysis regarding the marking of this place.

7 Extensions and refinements

One extension of the very basic model was already introduced when implicit and explicit repair were distinguished. The characteristics of the intermediate model (and accordingly the target model itself) can be further refined by applying the following extensions:

- Coverage of element failures in packages. A package implementing a redundancy scheme can be assigned a coverage which is a probability that the elements of a package can be coordinated in such a way that the failure is tolerated (e.g. the failure is detected and a redundant object can provide the service).

- Sensitivity of an element (node or object) to an underlying failure. It is a probability that the underlying failure will result in a failure of the element. It can cover local fault tolerance (e.g. an object may tolerate underlying transient hardware failures by self-checking and retry; in this case, this parameter characterizes the proportion of transient underlying failures and the fault tolerance coverage of the object), usage and timing dependency etc.

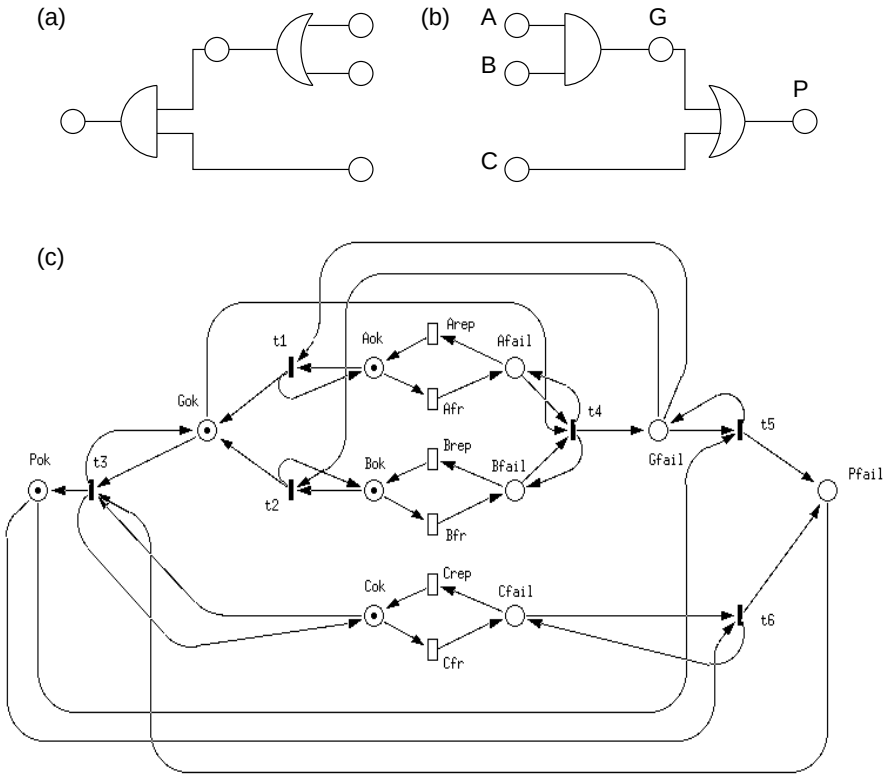

Figure 6: A PN subnet corresponding to a fault tree

- Detected and undetected errors in availability models. The repair process can be refined by introducing the notion of a detected error, since (usually) a detected error triggers the repair process. The error detection is characterized by the (internal and/or external) detection coverage (like in safety models).
- Modeling the fault to error to failure chain in details. Main characteristics of this chain are as follows:
  - A transient fault may result in a single error.
  - A permanent fault may produce errors (characterized by a rate).
  - An error may produce further errors (by a rate).
  - An error may result in a failure (by a given latency).
  - An error may become detected (by a given latency). Detected errors may trigger repair processes.
- Transient and permanent hardware failures. Since transient hardware failures can be tolerated by techniques being different from the ones that tolerate permanent (or both transient and permanent) failures, it may be useful to make explicit difference between transient and permanent failures. This way, separate activation, propagation and repair processes can be assigned to transient and permanent failures.

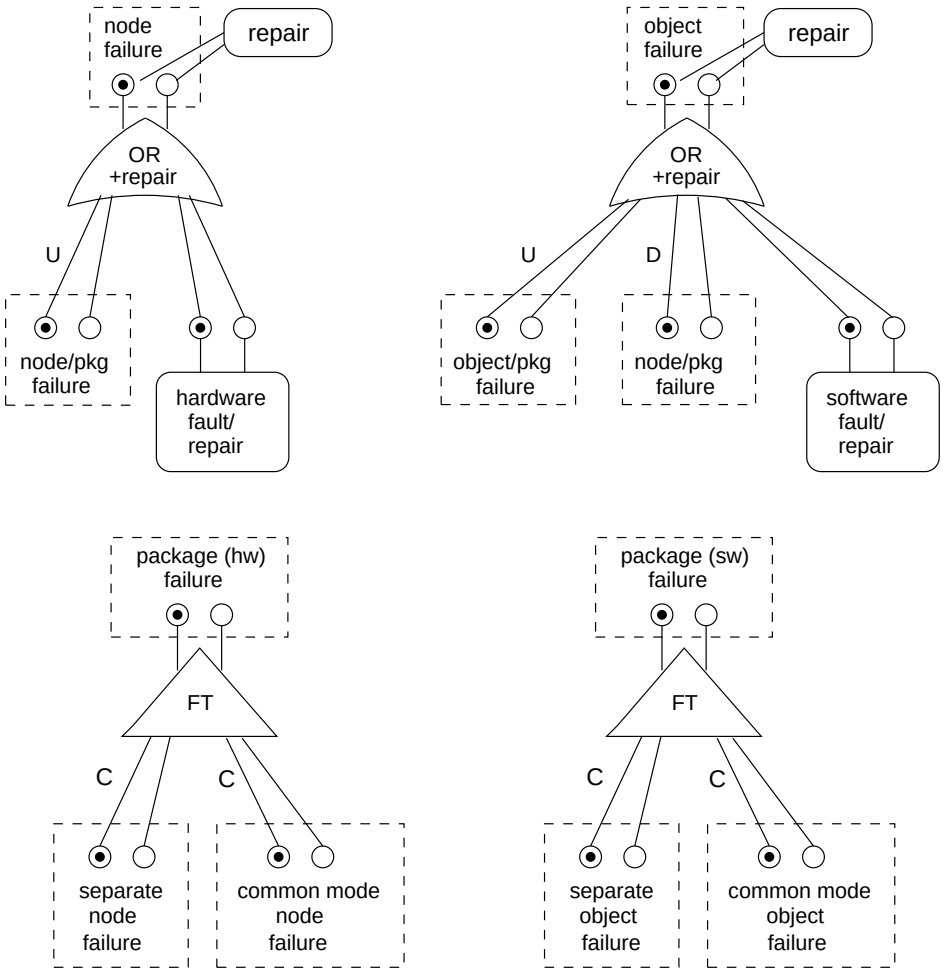

Figure 7: Composition of submodels

The first three extensions can be implemented in the target model by the application of the subnet presented in Figure 8. The fourth extension requires detailed modeling of activation and error detection processes. The fifth extension requires new subnets representing failure, propagation and repair processes of transient and permanent failures.

8 Derivation of fault trees by the analysis of the behavioral description

Derivation of the fault trees is a prerequisite step of the construction of the system dependability model.

If some conventions are applied then the fault trees corresponding to packages (implementing simple redundancy schemes) can be generated automatically, by the analysis of the behavioral description of the controller.

The following conventions are identified:

- Failure states should be marked (e.g. by stereotyping).

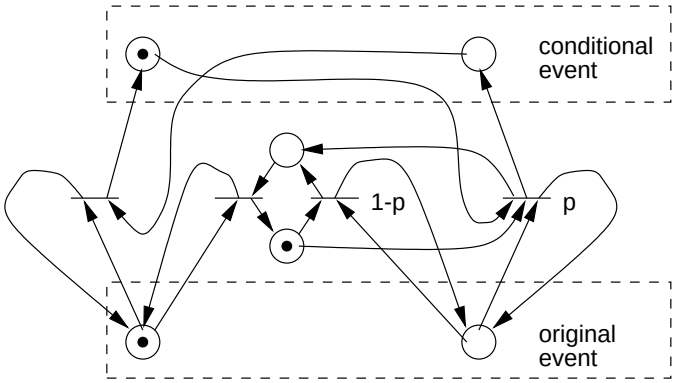

Figure 8: A subnet representing a conditional event

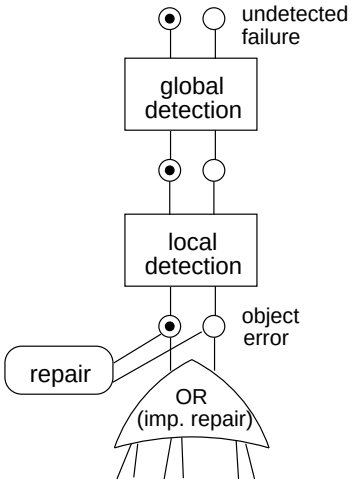

Figure 9: Propagation process with internal and external error detection

- Events representing failures (explicit error notification generated by the package) should be marked. These events will be referred to in the following as *failure events*.
- Normal responses of the package (called in the following *response events*) should be marked.

The fault tree is generated by the composition of the subtrees of (i) response events, (ii) failure events and (iii) failure states. The following steps are performed (in [1], a similar algorithm is proposed for reliability modeling):

- Backward reachability analysis from response events generates the set of trajectories leading from initial state to the response event. The OR relation of the subtrees of the trajectories form the fault tree corresponding to the error of the response.

  On a trajectory, the incoming events identify the objects which contribute to the response, while guards identify the test and compare actions of the controller(s) or adjudicator(s).

  Usually the following roles of objects can be distinguished (an object might play several roles):

- contributor: providing some (partial) results which are used to get the response (but not related to the task of redundancy management);
- tester: performing tests on results of (other) objects;
- comparator (voter): comparing results of two or more objects.

The fault tree corresponding to a trajectory is the OR relation of the events which may lead to the error of the response on the given trajectory. The following events are combined:

- Separate failure of a contributor without testing.
- Common mode failure of contributors without testing.
- Common mode failure of a contributor and its tester.
- Failures of a contributor escaping the test.
- Common mode failure of contributors which are compared.
- Common mode failure of contributors and the comparator.
- Separate failure of the controller (providing the response).

- Backward reachability analysis from explicit failure events generates the set of trajectories leading from initial state to the failure event. The OR relation of the subtrees of the trajectories form the fault tree corresponding to the failure event.

  On a trajectory, the events and conditions identify the objects, of which failures lead to the failure event. The fault tree corresponding to a trajectory is the AND relation of these failures. The following events are combined:

  - Separate failure of a contributor detected by a tester.
  - Common mode failure of contributors detected by a tester.
  - Separate or common mode failures of contributors detected by a comparator.

- Failure states usually provide explicit failure events, this way they are covered by the previous point. If a failure state is without failure events (fail stop) then it can be handled in the same way as failure events: Backward reachability analysis from the failure state generates the set of trajectories leading from initial state to the failure state. The OR relation of the subtrees of the trajectories form the fault tree corresponding to the failure state. On a trajectory, the events and guards identify the objects whose failures lead to the failure state. The fault tree corresponding to a trajectory is the AND relation of these failures.

The approach is illustrated by an example. Objects CW, $W1$ and $W2$ form a fault tolerance structure, where $W1$ and $W2$ are variants and CW is their controller, which implements a recovery block scheme. CW checks the results of the variant $W1$ (it is assumed that the coverage of the check is 100%), and if it is error-free then this result is accepted. Otherwise $W2$ will be executed, its result will be checked again and accepted if it is error-free. If both variants fail then the scheme will fail as well, reporting this towards the client.

The statechart of the controller CW is presented in Figure 10.

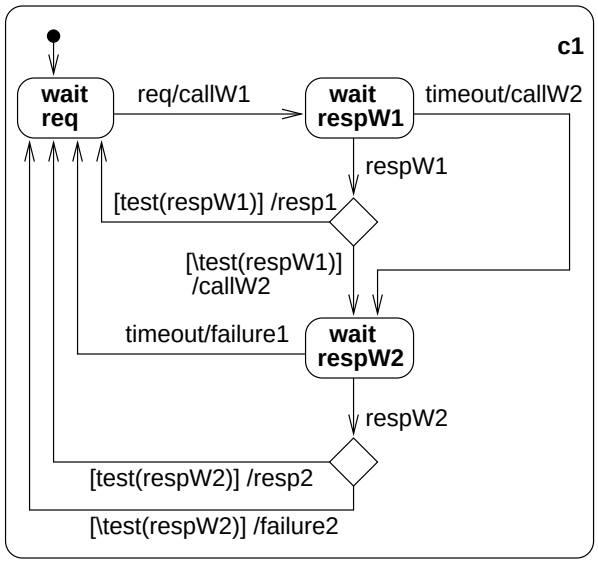

Figure 10: Statechart of the controller

Response events are `resp1` and `resp2`, while failure events are `failure1` and `failure2`. There is no failure state in this example. The following trajectories and failures can be recognized:

Trajectory to response event `resp1`: Event `respw1` is the result of object $W1$, its testing is made by CW. The response also requires CW, the controller. This way the common mode failure of $W1$ and CW and the separate failure of CW have to be taken into account. (The failure of CW covers the case when both $W1$ and CW fail simultaneously.) The fault tree corresponding to the trajectory is shown in Figure 11 (Tr1).

Trajectory to response event `resp2`: The guard condition shows that $W1$ fails. Event `respw2` is the result of object $W2$, its testing is made by CW. The response also requires CW, the controller. This way the separate failure of CW as well as the common mode failure of $W2$ and CW (in the case of the failure of $W1$) have to be taken into account (Figure 11 (Tr2)).

Trajectory to the failure event `failure1`: The guard condition shows that $W1$ fails (tested by CW), the event `timeout2` shows that also $W2$ fails. This way the separate failures of $W1$ and $W2$ have to be taken into account (Figure 11 (Tr3)).

Trajectory to the failure event `failure2`: The guard conditions show that both $W1$ and $W2$ fail (tested by CW). This way the separate failures of $W1$ and $W2$ have to be taken into account (Figure 11 (Tr4)).

9 Conclusion

The approach sketched in this document aims at the (automatic) construction of a dependability model. The main features of the approach are described as follows:

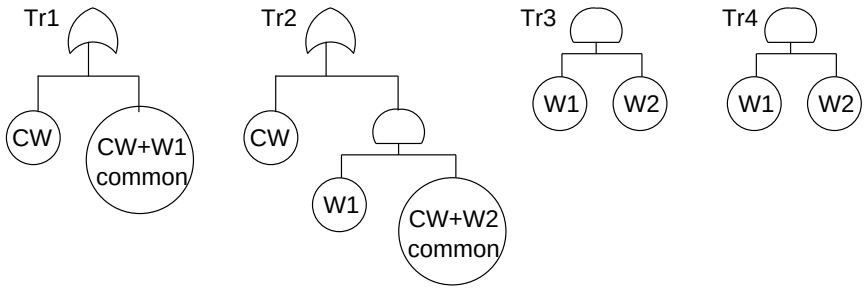

Figure 11: Fault trees corresponding to the trajectories of the controller

- The set of basic events, the set of derived events and some (trivial) propagation processes were defined by introducing an intermediate model, which is the result of a mapping from the UML model.

  The parameters, which are used in the semantics of the intermediate model, are the ones described in the companion document [5] (sometimes a more general or shorter name is used here). They are summarized in Table 2.

Model type	Parameters
Reliability	Separate failure rate Common mode failure rate (Coverage factor and sensitivity)
Safety	External error detection coverage Internal error detection coverage
Availability	Proportion of permanent faults (here self-recoverability) Repair rate

Table 2: Model parameters

- The non-trivial propagation processes are determined by (i) the sub-models stored in FT library together with FT schemes or (ii) sub-models constructed automatically on the basis of the behavioral description of (controllers of) FT schemes.

  It turned out that fault trees or Petri-net submodels with predefined interfaces are useful for the above purposes (to be stored in the FT library or to be derived automatically).

- The fault activation process as well as the repair policy should be derived on the basis of behavioral descriptions or extensions (constraints or stereotypes) assigned to the UML model.

References

[1] J. Davis, J. Scott, J. Sztipanovits, and G. Karsai. Integrated analysis environment for high impact systems. Research report, Measurement and Computing Systems Laboratory, Vanderbilt University, http://mcsl.vuse.vanderbilt.edu, 1997.

[2] J. B. Dugan and M. R. Lyu. Dependability modeling for fault-tolerant software and systems. In M. R. Lyu, editor, *Software Fault Tolerance*, volume 3 of *Trends in Software*, pages 109–137. Wiley & Sons, New York, 1995.

[3] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 23(7):39–51, July 1990.

[4] J.-C. Laprie and K. Kanoun. Software reliability and system reliability. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, pages 27–69. McGraw Hill, New York, 1995.

[5] I. Majzik and A. Bondavalli. Dependability analysis in the hide framework. HIDE research note, CNR-CNUCE, Pisa, Italy, 1998.

[6] M. Malhotra and K. S. Trivedi. Dependability modeling using Petri-nets. *IEEE Transactions on Reliability*, 44(3):428–440, September 1995.