Stochastic Dependability Analysis of System Architecture Based on UML Models

István Majzik[1], András Pataricza[1] and Andrea Bondavalli[2]

[1] DMIE, Budapest University of Technology and Economics,
Magyar Tudósok krt. 2, H-1117 Budapest, Hungary
[2]DSI, University of Firenze, Via Lombroso 6/17, I-50134 Firenze, Italy
{majzik,pataric}@mit.bme.hu, a.bondavalli@dsi.unifi.it

Abstract. The work in this paper[1] is devoted to the definition of a dependability modeling and model based evaluation approach based on UML models. It is to be used in the early phases of the system design to capture system dependability attributes like reliability and availability, thus providing guidelines for the choice among different architectural and design solutions. We show how structural UML diagrams can be processed to filter out the dependability related information and how a system-wide dependability model is constructed. Due to the modular construction, this model can be refined later as more detailed information becomes available. We discuss the model refinement based on the General Resource Model, an extension of UML. We show that the dependability model can be constructed automatically by using graph transformation techniques.

1 Introduction

Standardized design methods and tools are available for the designers of complex computer systems in order to increase the effectiveness of the design. UML (Unified Modeling Language [26]), UML based methods and CASE (Computer-Aided Software Engineering) tools are widely used for the design of various systems from small embedded controllers to large information infrastructures.

An effective design process should include an early validation of the architectural choices and concepts underlying the design. The more earlier the bottlenecks and insufficiencies are highlighted, the less is the loss due to the necessary corrections and re-design. Dependability is among the properties to be validated during the system design, especially in the case of critical systems.

Our earlier ESPRIT project HIDE[2] aimed at the creation of an integrated design environment that augmented UML based design tools with mathematical analysis techniques [7]. Model based dependability evaluation was supported by elaboration of

[1] This work was supported partially by ESPRIT Open LTR 27439 'HIDE', the Italian-Hungarian Bilateral Cooperation project I-37/2000, Hungarian NSF T-038027 and the Hungarian Ministry of Education under contract FKFP 0103/2001.

[2] HIDE - High-level Integrated Design Environment for Dependability was carried out by FAU Erlangen, PDCC Pisa, TU Budapest, Intecs Sistemi Pisa and MID GmbH Nuremberg.

an automatic transformation from UML diagrams to Timed Petri Nets (TPN) that could be solved by off-the-shelf analysis tools to get reliability and availability attributes of the system under design. This transformation, together with its extensions and implementation techniques elaborated since that time are the subjects of our paper.

The idea of translating design models into reliability models can be found in several papers. [13] converts UML models to dynamic fault trees. However, in this work the basis of the translation is not the functional design since UML is used mainly as a language to describe error propagation and module substitution. Here also a translator is reported that converts UML descriptions into reliability block diagrams. Similarly, OpenSESAME [34] uses high-level (graphical) diagrams to express dependencies, error propagation and redundancy structure. In this case analysis of availability is performed by a transformation to Generalized Stochastic Petri Nets. In [18], Markov chains are used to derive reliability of middleware architectures described in extended UML.

In our transformation we applied a modular and hierarchical approach. In the early architectural design phase the relevant information is captured from UML structural views and a system-wide dependability model is constructed. Here we use UML as a standard architecture description language (note that our method can be adapted to other architectural languages as well). The critical parts of the model (as shown by the analysis) are extended as the design gets refined and relevant information becomes available. These ideas are also widely accepted in the related literature. It is agreed that architecture evaluation based on analytical dependability modeling deserves attention in the early design phase [14], the modeling approach should be modular [22] and the model should be refined hierarchically as the design includes more and more information [4]. It is observed that the separation of architectural and service concerns allows the dependability analysis from the perspective of different users [29].

The paper is organized as follows. Sect. 2 motivates the model based dependability analysis. Sect. 3 introduces the design conventions and the extensions necessary to include local dependability related parameters in the early phases of the design. Sect. 4 describes the model transformation from the structural view of UML to the TPN dependability model. Sect. 5 discusses the ways of model refinement concentrating especially on the modeling of resources and faults related to the resources. Sect. 6 presents how the model transformation is implemented while Sect. 7 provides assessment of the approach. Sect. 8 concludes the paper.

2 Dependability Modeling and Analysis

This section motivates the need for model-based dependability evaluation, gives the rationale of dependability modeling and introduces a hierarchical and modular modeling approach.

2.1 Purpose of the Model Based Dependability Analysis

Evaluation of *availability* and *reliability* (two attributes of system dependability as defined in [19]) is necessary to assess whether the system being developed satisfies its

targets. Analytical modeling has proven to be useful and versatile to evaluate these attributes in the design phase. Dependability models allow comparing different architectural solutions and design choices and to run sensitivity analysis identifying both dependability bottlenecks and critical parameters to which the system is sensitive.

In our approach, dependability modeling is to be performed in addition to the architectural design based on the extensions of the architectural model of the target system by the parameters needed for the analysis. This approach avoids to build a dependability model from scratch, thus the consistency between the designers' model and the dependability model is guaranteed by the process.

2.2 Components of a Dependability Model

The abstraction of a dependability model consists of the following general parts:

- *Fault activation processes*, which model the fault occurrences in basic system components (especially physical resources).
- *Propagation processes*, which model the consequences of fault occurrences and result in derived failure events. E.g. a failure of a network card results in the failure of an information retrieval service.
- *Repair processes*, which model how basic or derived events are removed from the system. Repair can be implemented by fault treatment and/or error recovery depending on the type of the component.
- *Mapping from architectural level to service level*, which gives how the failures of software and/or hardware components and subsystems result in the failure of a system service (as observable by the user). Different mappings can be used to take into account different service needs (ways of usage).

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, and fault tolerance schemes). The repair processes are determined by the (physical or) computational policy implemented in the system.

2.3 Formalisms and Tools for Dependability Analysis

Among the various formalisms and tools developed for dependability modeling and analysis, Petri nets have been widely accepted because of their expressiveness and powerful solution techniques. Timed and stochastic extensions of Petri nets encompass the class of Generalised Stochastic Petri Nets (GSPN) [2], Deterministic and Stochastic Petri Nets (DSPN) [1] and Markov Regenerative Stochastic Petri Nets (MRSPN) [9]. Many automated tools based on Petri nets are available, e.g. UltraSAN [30], PANDA [3], GreatSPN [8], SPNP [10], SURF2 [31]. In certain cases (e.g. exponential transition firing times) analytic solution is possible, otherwise simulation has to be performed.

2.4 Mastering Complexity

Dependability modeling and analysis of complex systems pose serious problems due to the complexity of the dependability model, which may be out of the range existing tools can deal with. This complexity is due to the large number of components as well as the complex interactions among (redundant) hardware and software entities.

Small systems can be analyzed by modeling the behavior at a fine granularity, e.g. at UML statechart level. However, as the complexity of the system increases, another approach has to be followed. *Modular modeling* and *hierarchic refinement* of a rough (structural) dependability model are typical ways of mastering complexity. First only the relevant aspects of the system are modeled and analyzed, which enables the computation of numerical results and at the same time allows the estimation of the sensitivity of system-level attributes to the parameters of specific components. In this way those components and design decisions can be identified that need a refined analysis. Modular modeling allows replacing components of a rough dependability model with more detailed ones.

In our approach, first we build a quite abstract model, which concentrates on the structure of the system and, accordingly, takes information from the structural UML diagrams. It has the following advantages:

- It results in a system-wide (but less detailed) representation of the dependability characteristics of the system in its entirety.
- The size of the model and thus the time and resource needs of the analysis can be controlled.
- Preliminary evaluations of the system dependability during the early phases of the design can be provided. Usually, the structural UML models, that is the class, object, and deployment diagrams, are available before the detailed low level ones, and the analysis on models derived from the structural view provides indications about the critical parts of the system that require a more detailed representation.

By using appropriate interfaces, the structural dependability models can be augmented by inserting more detailed information coming from refined UML models of the identified critical parts of the system. This way we can deal with various levels of details ranging from very preliminary abstract UML descriptions up to the refined specifications of the last design phases.

Accordingly, dependability modeling and analysis can be performed in several steps. The first step has the fundamental task of extracting the relevant dependability information from the mass of information available in the UML description. In this step, the structural dependability model is built, in which we can fix the fault activation, error propagation and repair processes as well as the mapping to the service level. The dependability model is formalized by a Timed Petri Net (TPN) that can be translated to the input format of the specific Petri net tool selected for performing the analysis.

The subsequent steps can refine the structural model by replacing modules of the structural dependability model (i.e. the corresponding TPN) with sub-models constructed on the basis of the refined UML models (e.g. also behavioral diagrams like statecharts and message sequence charts that describe the interaction of components more precisely. For the purpose of this sub-model construction, other methods of processing UML diagrams are provided (Section 5).

In this paper we will show the construction of the structural dependability model, identify the interfaces to extend this model and make reference to approaches available to construct the refined sub-models.

2.5 Model Parameters and Validation of the Dependability Model

Dependability modeling - especially in the case of rough structural models constructed in the early design phases - should be based on a number of assumptions and simplifications. Since the assumptions and simplifying hypotheses may lead to wrong approximation of the system behavior, the resulting error should always be estimated either through sensitivity analysis or by comparing the results obtained by the model containing the assumption and by a model where it has been released.

Another problem is that models need many (aggregate) parameters whose meaning is not always intuitive for the designers. Obviously, values for such parameters may be difficult to provide in early phases of design. The ideal source would be to provide them through experimental tests on prototypes (unlikely to be available). Alternatives are data from similar systems (modules) or data derived from designers' experience.

Experimental results have to be provided at later stages to validate the numerical results gained by the solution of the dependability model. It has however to be emphasized that instead of the concrete numerical dependability measures, the outcomes of the sensitivity analysis and the comparison of design choices are the most beneficial results of a model based dependability evaluation.

3 System Modeling in the Early Phases of Design

In the early phases of the design we assume that an architectural description of the system is available. The software architecture is specified by class, object and collaboration diagrams. The allocation of the software elements to hardware units and resources is described by static deployment diagrams, no dynamic resource utilization is specified. The dependability-related attributes of components are aggregate ones that combine performance-related and dependability-related attributes (like component activation probability and error detection coverage are aggregated to error propagation probability).

In the subsequent design steps the model of the system is to be refined. From the point of view of the dependability modeling, the management of redundancy and the specification of dynamic resource usage play a crucial role. Accordingly, more refined UML models are assumed that separate usage (conformant with the corresponding UML resource modeling profile [25]) and fault activation/error propagation. The aspects of this kind of model refinement are detailed in Sect. 5.

3.1 Attributes of the System and its Components

The system-level dependability attributes, i.e. availability and reliability are computed by the solution of the dependability model based on the *local dependability attributes*

of the various components of the system. In the abstract structural model, the local (aggregate) attributes of basic components are values characterizing fault activation, error propagation and repair processes (Sect. 2.2) as follows:

- Fault activation is characterized by the *fault occurrence* (random variable representing the time needed for the activation of a fault), the *error latency* (random variable representing the time needed to bring the component to a failure after the fault generates an erroneous internal state) and the *ratio of permanent and transient faults*. Naturally, in stateless (purely functional) components there is no error latency, while in software components only permanent (design) faults occur.
- Error propagation is characterized by the *error propagation probability* which is assigned to a pair of interconnected components whenever the failure of a server-like component results in the failure of the another, client-like component. Two components can be connected in terms of failure propagation bi-directionally if any one of them may influence the failure of the other component. In this case error propagation probability is assigned in both directions.
- Repair is characterized by the *repair delay* (random variable representing the time needed to perform the repair). Note that error propagation prescribes a constraint for the repair of a component: the repair of a component can not be completed until all the used components are fully operational.
- Mapping from architecture to service levels is characterized typically by Boolean logic expressions describing what combinations of component failures can constitute the failure of the service. These combinations are visualized by a *fault tree*.

The structure of this dependability model is inspired by the approach presented in [20]. We have slightly modified that model: we use a more reduced hierarchy and, for the sake of convenience, we distinguish between stateless and stateful components. The distinction between them is important from the point of view of the potential erroneous state, error latency and propagation. Similarly, the distinction of hardware components is necessary from the point of view of the transient faults.

As it partially turns out from the above set of relevant attributes, we have introduced a set of general assumptions for the dependability model:

- Solid software failures are not taken into account (assuming that they were removed before execution by a thorough debugging and fault removal).
- There are no failures that compensate the effects of other ones.
- "Repair" is implicit if the fault disappears after activation (transient hardware faults and all software faults). Repair of a derived failure is implicit if it disappears as soon as the underlying faults and failures have been repaired.
- 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 system components.

In the following we detail the restrictions to be imposed on the UML designer to allow translating the specification into a dependability model. These restrictions are mainly related to the introduction of redundancy into the system, for which particular structures are to be utilized to permit the identification of the crucial points. Since the information on dependability aspects is typically not included into a UML design, we prescribe a set of extensions of the standard UML in order to create controlled interface towards the designer for the input of parameters, the selection of desired measures, and the choice of the fault-tolerance structures to be included into the system.

3.2 Redundancy Structures

One fundamental choice has been made in defining the way redundancy has to be expressed in the UML design. We opted for the so called "class based" redundancy which prescribes that elements of a redundancy structure must be defined as instances of specific classes (based on templates and stereotypes) [35]. It is important to notice that this choice supports the use of design patterns collected in a fault-tolerance library. Moreover, the construction of such library can be integrated with the dependability modeling in the sense that it will be possible to associate to the elements of the library their dependability sub-models which will be derived only once, thus building at the same time a library of dependability sub-models.

In general, a system component is redundant if its service can be delivered by another component in a coordinated way, without the interaction of the client(s). Accordingly, operation of redundant components presumes the existence of a coordinator (called here *redundancy manager*) and some type of *adjudicator*. A given service is provided by a set of redundant components called here *variants*, which are coordinated by the redundancy manager: the service is available through the redundancy manager and the redundant components can not be used separately. A component may be a participant in a single redundancy structure only. Other, non-redundant components can not be included in a redundancy structure (but the variants may use the service of another components).

Accordingly, redundancy structures must be composed of objects instantiated from the following types of classes: redundancy manager, variant, and adjudicator (which can be further refined by various subtypes e.g. tester, voter or comparator).

The specific conditions of the failure of the redundancy structure (which can be quite complex) are either available in the library of dependability-related design patterns or they have to be derived by analyzing the UML diagrams describing the behavior of the redundancy structure.

It has to be emphasized that the construction of the structural dependability model relies on behavioral diagrams only in the case of redundancy managers. Otherwise, behavioral diagrams are used only in the subsequent phases to refine the structural dependability model (Sect. 5).

3.3 Specifying Dependability Related Properties and Requirements

Basic UML focuses on capturing the complex functionality of the system but neglects non-functional aspects such as quality of service (QoS). To allow dependability analysis of designs, UML has to be extended with a notation for describing the quantitative properties of model elements and the required properties of the system to be analyzed.

There are some ongoing activities to extend UML for dealing with such kind of data. The OMG proposal [25] describes a general approach to classify model elements by stereotypes and bind performance characteristics to them by using tagged values. We followed this approach, because this annotation does not change the UML metamodel and thus it is conformant with existing CASE tools. However, it may be inconvenient for working with a large number of elements (some of them having identical attributes). Because of its generality and object-oriented nature, the QoS specification

language QML [12] is a potential extension. It was adapted to quantitative model analysis with respect to the OMG proposal in [17].

Accordingly, in our approach standard extensions of UML, i.e. stereotypes and tagged values are used to identify the elements of redundancy structures and to assign dependability attributes (Sect. 3.1) to components and relations.

The classes in redundancy structures, namely the redundancy manager, variant and adjudicator are stereotyped as `<<redundancy manager>>`, `<<variant>>` and `<<adjudicator>>`, respectively.

Stateless or stateful software and hardware components are stereotyped accordingly. The local dependability attributes are described by tagged values as follows:

- Fault occurrence: `FO=x`
- Error latency: `EL=y`
- Ratio of permanent faults: `PP=v`
- Repair delay: `RD=z`

The designer can assign a single value (here x, y, z, v are used to instantiate the parameter), two values (range for a sensitivity analysis), or no values (the parameter should be derived based on the parameters of underlying elements in the hierarchy). Different sets of tagged values are assigned to different types of components according to Table 1.

Table 1. Stereotypes and tagged values

Component type	Dependability attributes	Stereotypes	Tagged values
Stateless hardware	Fault occurrence, ratio of permanent faults, repair delay	<<stateless>>, <<hardware>>	FO, PP, RD
Stateful hardware	Fault occurrence, error latency, ratio of permanent faults, repair delay	<<stateful>>, <<hardware>>	FO, EL, PP, RD
Stateless software	Fault occurrence	<<stateless>>, <<software>>	FO
Stateful software	Fault occurrence, error latency, repair delay	<<stateless>>, <<software>>	FO, EL, RD

These stereotypes and corresponding tagged values can be applied to UML objects, classes (in this case all objects instantiated from the class should be assigned the same set of parameters), nodes and components.

Stereotype `<<propagation>>` indicates an error propagation path, with the tagged value `PP=x` to assign propagation probability. This stereotype can be applied to links between objects, associations between classes or nodes, deployment relations, dependencies, and generalization relationships.

In order to derive the non-trivial relations in redundancy structures automatically, further extensions are required. In statechart diagrams of the redundancy managers, failure states are distinguished by stereotypes. Similarly, stereotypes identify the specific types of adjudicators (e.g. testers and comparators). Tagged values are used to assign common mode fault occurrences to components. The detailed description of these extensions can be found in [5].

4 Construction of the Dependability Model

The structural dependability model is constructed first in the form of an intermediate model (IM) which is a hypergraph representing the components and their relations relevant from the point of view of the dependability analysis. In this way some peculiarities of UML (e.g. package hierarchy, composite objects and nodes, different types of dependencies) can be resolved which results in a simple and flat model with a limited set of elements and relations.

4.1 The Intermediate Model (IM)

The IM is a hypergraph G=(N,A), where each node in N represents an entity, and each hyperarc in A represents a relation between these entities. Both the nodes and the hyperarcs are labeled, that is they have attached a set of attributes completing their description. A generic node of the IM is described by a triple consisting of the fields `<name>`, `<type>` and `<attributes>`. We now give the semantic of the intermediate model by describing the sets N and A and what they represent.

Nodes represent the stateful or stateless hardware/software components described in the set of UML structural diagrams. Four types of nodes are used: SFE-SW (stateful software), SLE-SW (stateless software), SFE-HW (stateful hardware) and SLE-HW (stateless hardware). Attributes of the nodes characterize the fault activation and the repair processes according to Table 1.

The fault tolerance (redundancy) structures are represented by composite nodes FTS. The system service is represented by another specific node SYS to which the system-level attributes (measures of interest) are assigned. It has to be emphasized that these nodes represent the composite structures and not individual components.

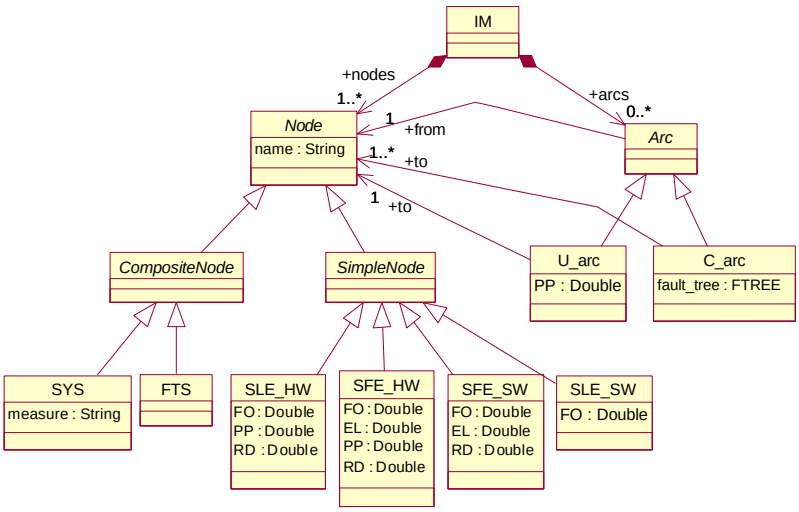

Fig. 1. Metamodel of the IM

Hyperarcs among the nodes represent two kinds of relations. "Uses-the-service-of" (U) type of hyperarc indicates an error propagation path between a server-like component (typically a resource) and a client-like component. A U hyperarch is a one-to-one relation directed from a node representing the client to the node representing the server. Error propagation may occur in the opposite direction, i.e. from the server to the client. The assigned attribute is the error propagation probability.

Another type of hyperarc represents the "is-composed-of" (C) relation in the case of fault tolerance (redundancy) structures, i.e. FTS nodes, and system services, i.e. SYS nodes. This type of hyperarc is a one-to-many relation directed from the FTS or SYS node to the nodes representing the constituent parts. The hyperarc is assigned a fault tree describing the conditions of error propagation from the parts to the composite structure.

The metamodel of the IM (in the form of class diagrams) is shown in Fig. 1.

4.2 Dependability Related Information in UML Diagrams

The dependability model is built by projecting the UML model elements into nodes, and the UML structural relations to hyperarcs of the IM. The types of nodes are determined by UML stereotypes, while the attributes are projected from tagged values of the corresponding UML model elements (Table 1). The types of hyperarcs are determined by the stereotypes assigned to the key elements of the composite structures (i.e. the redundancy managers).

According to the high-level approach, not only the “natural” software elements as objects, tasks, processes etc. can be identified but also higher-level, compound elements as use cases or packages. As the UML design is hierarchical, intermediate levels of the hierarchy can be represented by SYS nodes. The representation of a compound UML element (like a package) depends on the level of detail described or selected by the designer. If a compound UML element is not refined, or its refinement is not relevant for the dependability analysis (as selected by the designer) then it is represented by a simple software or hardware node in the IM. If it is refined and its refinement is relevant then its subcomponents are represented as simple nodes and the compound as a whole is represented by a SYS node. In the case of hyperarcs, all potential propagation paths are taken into account thus the structure of the model represents worst-case error propagation. The fine-tuning is left to the actual parameter assignment.

Now we summarize the role UML diagrams and elements considered in our model derivation. As already stated, we focus on the structural UML diagrams to construct the dependability model. Nevertheless, it may be necessary to use also behavioral diagrams for a more detailed modeling of redundancy structures.

- Use case diagrams identify actors and top-level services of the system (this way also identify the system level failure).
- Class diagrams are used to identify relations (associations) that are traced to objects. By default, each class is instantiated by a single object.
- Object, collaboration and sequence diagrams are used to identify objects (as basic components) and their relations. Messages identify the direction of the relations.

– Component diagrams are used to identify the relations among components, and in this way among objects realized by the components. Note that the components are instantiated on the deployment diagrams.
– Deployment diagrams are used to identify hardware elements and deployed-on (a specific case of "uses-the-service-of") relations among software and hardware elements. Relations among hardware elements (e.g. communication) are also described here.
– Statechart diagrams are used basically only in the case of redundancy structures, to derive the non-trivial relations among participants of the structure.

In the following we sketch the projection in the case of object and class diagrams. Other projections are described in [5].

4.3 Projection of the Model Elements of Object and Class Diagrams

Object diagrams (and also collaboration diagrams) include instances that are represented by nodes in the IM. Simple objects are projected into simple nodes of the IM. Composite objects are projected into a set of nodes, with unidirectional error propagation paths from the sub-objects to the composite one.

Since each object is a particular instance of a class, the relations of objects and their type information can be deduced by the analysis of the class diagrams. Model elements of class diagrams are utilized as follows.

Inheritance hierarchy of classes is utilized to identify the relationships: if an object is instantiated from a given class then the relationships of this class and also of its ancestors have to be taken into account.

Associations are binary or n-ary relations among classes. In general, associations mean that the objects instantiated from the corresponding classes know (i.e. can name) each other. Associations may be instantiated by links on object and collaboration diagrams, and communication among objects is possible along these links. Accordingly, an association indicates a potential bi-directional error propagation path among these objects thus it is projected into the IM. The following additional features might be taken into account.

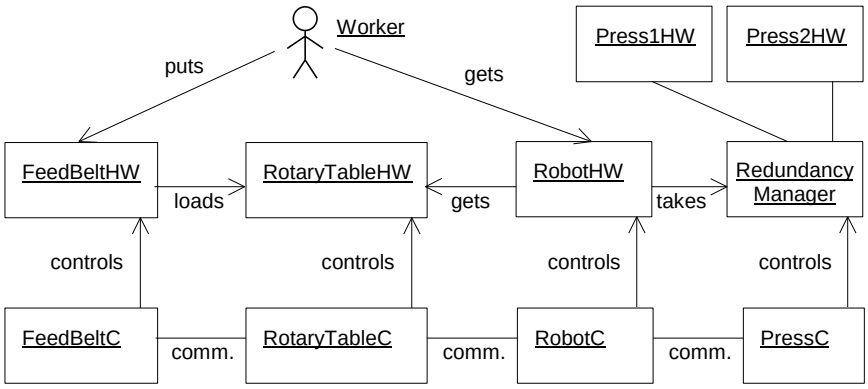

Fig. 2. Object diagram of a production cell

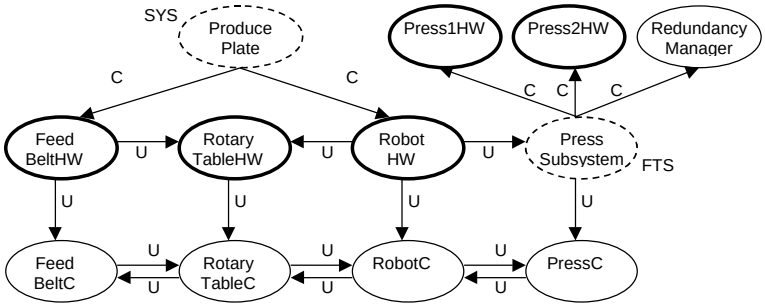

Fig. 3. Projection of the UML object diagram in Fig. 2. into the IM

- Navigability of an association end denotes whether the instance can be directly reached. However, it does not give precise information about the direction of the potential error propagation, since through return values also a unidirectional navigation may result in bi-directional error propagation. Accordingly, each association is projected by default to bi-directional error propagation.
- Or-associations (indicating a situation when only one of several possible associations may be valid) are all projected into the IM, in this way a worst case dependability model is constructed.
- Multiplicity of association ends are taken into account only when the classes are not instantiated on object diagrams. Indeed, in the early phases of the design the instantiation of the model may not be available. It might be useful for the designer to have a default instantiation in order to compute rough dependability measures. Accordingly, if a class has multiplicity specification then the value or the lower bound of its range can be taken into account. If it is missing or equal to zero then by default a single instance is taken into account. Metaclasses, type classes and parameterized classes (i.e. templates) are not instantiated.
- Aggregation (composition) is projected into a unidirectional error propagation path: the aggregate (composite, respectively) uses the service of the aggregated elements.
- Unary associations (both ends attached to the same class) denote associations among objects of the same class. According to the structural (worst case) approach, they are projected into the IM denoting error propagation paths from each object to each other. Reflexive paths are not considered. N-ary associations are projected into the IM as a set of binary associations, where each possible pair of classes included in the n-ary form is taken into account.
- Association classes are handled as separate classes having associations with the classes at the endpoints of the association.

Generalization is the relationship between a more general element (class) and a more specific one. Generalization does not indicate an error propagation path, thus it is not projected into the IM (but the inheritance of relations defined by generalizations is taken into account).

Dependency means a semantic relationship between classes. From the point of view of dependability modeling, those dependencies are relevant which relate also the instances (not only the classes themselves, like `<<refine>>` or `<<trace>>` relationships). This way in the set of the predefined types of dependencies, only the

<<uses>> dependency (meaning that an element requires the presence of another element for its correct functioning) indicates an error propagation path, thus it is projected into the IM.

An example of an object diagram of a production cell (described in [6]) and the corresponding IM are shown in Fig. 2 and Fig. 3. Note that the objects Press1HW, Press2HW and RedundancyManager form a redundancy structure (Sect. 4.5). The external actor in Fig. 2 identifies the top level components providing the system service (Sect. 4.6).

4.4 Projection of Resource Usage

UML deployment diagrams show instances of hardware components (nodes) and the configuration of run-time components on them.

- *Nodes* are run-time physical objects, usually hardware resources. They are projected into hardware nodes in the IM.
- *Objects* realized by components are projected into software nodes of the IM.
- *UML components* represent pieces of run-time software. If a component is refined, i.e. the set of objects realized by the component is given then the component is not projected into a separate software node of the IM. If a component is not refined then it is projected into a single software node of the IM.
- *Deployment relations* among nodes and components and realization relations among components and objects (both shown by graphical nesting or composition associations) indicate potential error propagation paths with direction from the nodes to the objects. They are projected into the IM.

Note that the conventions introduced in the General Resource Model (GRM [25]) enable a more refined modeling of resource usage than in the case of deployment diagrams (see Sect. 5).

4.5 Projection of Redundancy

A redundancy structure (identified by the redundancy manager, Fig. 4) is projected into an FTS node of the IM connected, by using a C hyperarc, to the elements representing the redundancy manager, the adjudicators, and the variants.

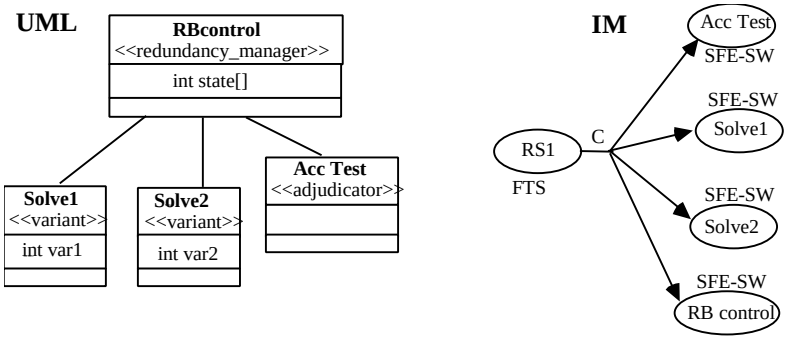

Fig. 4. Projection of a simple redundancy structure

In simple cases the error propagation is described by a fault tree. The construction of a fault tree corresponding to a redundancy structure (not included in the library of schemes) requires the analysis of the behavior, i.e. the statechart diagram, of the redundancy manager. This kind of analysis is supported by the designer, as he/she identifies (by stereotyping) the failure states and events in the statechart. The fault tree is constructed by a reachability analysis enumerating the paths in the statechart that can lead to failure states (i.e. failure of the redundancy structure) [5]. The incoming events on these paths identify the failures of variants and/or adjudicators that are considered as basic events in the fault tree. Repair is taken into account in a default way by using the dual counterpart of the fault tree.

In the case of sophisticated recovery and repair policies, the statechart of the redundancy manager is transformed directly to a TPN subnet (Sect. 5.1). This approach results in a more complex model but allows the analysis of non-trivial scenarios and temporal dependencies.

4.6 Mapping from Architectural to Service Level

In UML, the services of the system are identified on use case diagrams. Model elements of these diagrams include actors, use cases, communication associations among actors and use cases, and generalizations among use cases.

- A *use case* represents a coherent functionality of the system. Usually, each use case is refined by interactions of objects. However, it may happen that in the early phases of the design only some (important or critical) use cases are refined, the others are not. Accordingly, if a use case is not refined or the refinement is not relevant then it is projected into a simple IM node, otherwise it is projected into a SYS node of the IM, which relates the nodes resulting from the projection of the other UML diagrams belonging to this use case.
- *Actors* represent (roles of) users or entities that interact directly with the system. Being external entities from the point of view of a given service, actors are not projected into the IM.
- *Communication associations* among actors and use cases identify the services of the system. If a use case is connected directly to external actor(s) then it is projected into a top-level SYS node of the IM. Usually, a real system is composed of several use cases, more of them being connected directly to actors. Dependability measures of such use cases can be computed separately, by a set of dependability models assigned to each use case. However, all services of the system can also be composed in a single dependability model, computing the measures corresponding to multiple SYS nodes.
- *Relationships* among use cases are represented in UML by generalizations with stereotype `<<extend>>` and `<<include>>`. Extend relationships mean that a use case augments the behavior implemented by another one. It indicates an error propagation path in the direction of the relationship, thus it is projected into the IM. Include relationships mean a containment relation thus they will be projected (in the reverse direction) similarly into error propagation paths.

4.7 Construction of the Analysis Model

On the basis of the IM a second step of the transformation builds a TPN dependability model, by generating a subnet for each model element of the IM [5].

A TPN model is composed of a set of elements as listed in Table 2. Places, transitions and subnets all have a name, which is local to the subnet where they are defined. Transitions are described by a random variable (specifying the distribution of the delay necessary to perform the associated activity) and a memory policy field (a rule for the sampling of the successive random delays from the distribution). A transition has a guard, that is a Boolean function of the net marking, and a priority used to solve the conflict. The weights on input and output arcs may be dependent from the marking of the net. Subnets are a convenient modeling notation to encapsulate portion of the whole net, thus allowing for a modular and hierarchical definition of the model.

Table 2. Elements of the TPN model

Element	Description
Place	<name> <initial tokens>
Transition	<name> <random variable> <memory policy> <guard> <priority>
Input arc	<from place> <to transition> <weight>
Output arc	<from transition> <to place> <weight>
Subnet	Nested TPN sub-model

Notice that the class of TPNs defined here is quite general. If the TPN model contains only instantaneous and exponential transitions, then it is a GSPN that can be easily translated into the specific notation of the automated tools able to solve it [8,10]. If deterministic transitions are included as well, then the model is a DSPN, which under certain conditions can be analytically solved with specific tools like UltraSAN, and TimeNET. If other kinds of distributions of the transition firing times are included, then simulation can be used to solve the TPN model.

We take advantage from the modularity of the TPN models defined above, to build the whole model as a collection of subnets, linked by input and output arcs over interface places. For each node of the hypergraph, one or two subnets are generated, depending from node type. The basic subnets represent the internal state of each element appearing in the IM, and model the failure and repair processes.

Fault Activation Subnets. The fault activation subnets (called basic subnets hereafter) include a set of places and transitions that are also interfaces towards other subnets of the model. Places called H and F model the healthy and failed state of the component represented by the IM node (Fig. 5). Fault activation subnets of stateful elements also include place E to represent the erroneous state of the component. For a stateless node, transition `fault` models the occurrence of a fault and the consequent failure of the node. For a stateful node, the occurrence of a fault generates first a corrupted internal state (error), modeled by the introduction of a token in E. After a latency period, modeled by transition `latency`, this error brings to the failure.

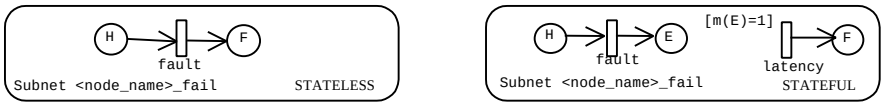

Fig. 5. Fault activation subnets for stateless and stateful nodes

For each FTS and SYS node, a basic failure subnet containing only two interface places, namely H and F, is generated in the TPN model. Indeed, FTS and SYS nodes represent composite elements, and their internal evolution is modeled through the subnets of their composing elements.

Propagation Subnets. By examining the U hyperarcs of the IM, the transformation generates a set of propagation subnets, which link the basic subnets. For instance, suppose node A is linked by a U hyperarc to node B in the IM. In this case, we want to model the fact that a failure occurred in the server B may propagate to the client A, corrupting its internal state. The propagation subnet B->A shown in Fig. 6 models this phenomenon (immediate transitions are depicted by thin bars).

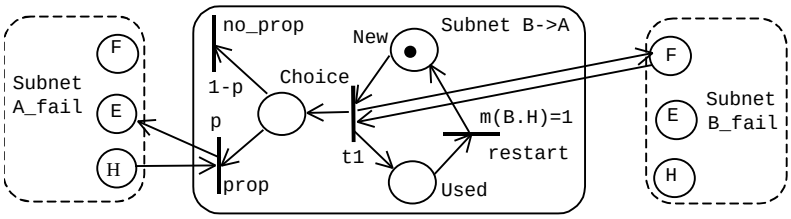

Fig. 6. Error propagation from node B to node A

The propagation subnet becomes enabled only after element B has failed. At that time, a token is put into place B.F, and the propagation subnet is activated. The subnet moves with probability p the token being in place A.H to place A.E. This models the introduction of an error in element A. A single token circulates among the two places New and Used, to allow the propagation subnet to be activated only once for each failure of B (t1 cannot fire until a token is moved from place Used to New).

Consider now a type C hyperarc, linking a FTS node P with its composing nodes. The fault-tree associated with the arc expresses the logic with which the failures of the composing elements propagate towards the failure of P. Also, the dual of the fault-tree (obtained by exchanging each AND gate with an OR gate and vice versa) represents the way the composed element P gets repaired when the composing element are re-covered. Thus, the fault-tree is translated into a failure propagation subnet, and its dual counterpart is translated into a "repair propagation" subnet. These two subnets are linked to the basic subnets of all the elements connected by the C hyperarc. Note that the Boolean conditions of the fault trees can be represented in Petri nets either by enabling conditions of transitions or by explicit subnets corresponding to AND or OR gates [5].

Repair Subnets. The repair subnet of a node is activated by the failure occurred in the fault activation subnet of the same node. For instance, Fig. 7 shows the repair subnets

for stateless and stateful hardware nodes. The two transitions `implicit` and `explicit` represent the two different kinds of repair which are needed as a consequence of a transient and permanent hardware fault, respectively. If the node is stateless then the final effect of the repair is the insertion of a token in place H. If it is stateful then the repair also removes a token from place E, modeling the error recovery activity.

Notice that all the parameters needed to define the subnets are found in the IM in the obvious fields.

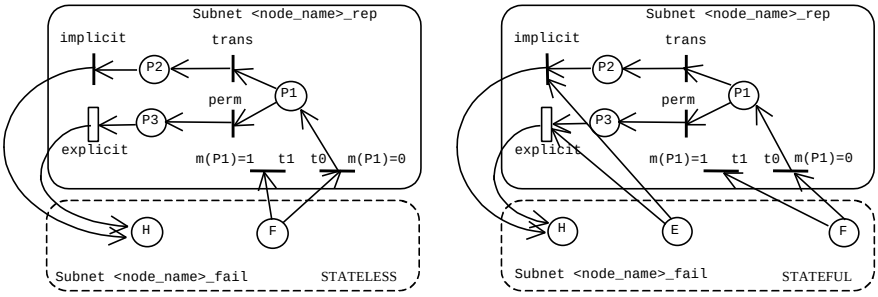

Fig. 7. Repair subnets of hardware nodes

Subnets for Mapping from Architecture to Service Level. C hyperarcs linking SYS nodes are handled in the same way like C hyperarcs linking FTS nodes. When a SYS node does not have an associated fault tree, then we implicitly associate to it a simple fault tree representing the OR relation of all the composing elements.

The markings of places H and F for the SYS node at the top level of the IM define a partition between proper and improper service: whenever a token reaches place F, the service (the object of the dependability evaluation) is considered as failed. Accordingly, the solution of the dependability model should provide the average number of tokens in H to get asymptotic availability figures and the time of the first occurrence of a token in F to get reliability figures.

5 Refinement of the Dependability Model

We identified and elaborated the following options to perform refinement in the structural dependability model as the UML model becomes more detailed:

- Mapping the behavioral diagrams describing redundancy management directly to TPN subnets.
- Constructing subnets on the basis of the refined modeling of usage and fault activation in the used resources.

These options will be described in the following sections.

5.1 Refined Modeling of Redundancy Management

The behavioral diagrams describing redundancy and resource management are natural candidates to be used during refinement since they describe the core logic of the dependability model. In Sect. 4, fault trees were constructed on the basis of the statechart diagram of the redundancy manager in a class based redundancy scheme. The fault tree representing logic conditions (as a static snapshot) can be integrated with the fault activation and repair subnets by using the Petri net places E, F and H representing the state of the components. However, in this way sophisticated repair/recovery scenarios and temporal dependencies could not be taken into account.

In the refinement step, also the dynamics of replica management (sequence of failure events and repair actions) can be considered. Statecharts of selected objects are mapped directly to Petri nets by a model transformation that preserves the dynamic semantics of the statechart [15]. This way the designer is allowed to use the full power of statecharts (state hierarchy, concurrency etc.) to describe application-dependent replication and recovery strategies. It has to be emphasized that only the statecharts of the objects responsible for replica management and recovery are considered in this refinement step. They are interfaced with the other parts of the system by events describing failure occurrences (error reports), and actions initiating repairs. Accordingly, instead of using the places corresponding to the *states* of the components (as in the case of fault trees), the integration of the resulting subnets requires additional TPN places representing the *changes* in the system as follows:

- Input places of the refined subnets represent events. By default, the basic subnet responsible for fault activation puts a token into the place representing the corresponding failure event when it occurs (i.e. state changes from H to F). In a further refinement step, more precise error detection and coverage can also be modeled (similarly by statecharts).
- Output places of the refined subnets represent actions. The subnet belonging to the statechart puts a token into a place representing a repair action when it is generated. By default, this action triggers a TPN transition corresponding to the explicit repair in the component (i.e. target of the action) affected by the repair. Again, based on this interface the repair actions can be refined in subsequent steps.

This approach was introduced first in [16] and adapted to the fault tolerance infrastructure defined by FT-CORBA [23] in [21]. In this adaptation, first a high-level dependability model is constructed that allows the analysis of the fault tolerance strategies and properties directly supported by the standard infrastructure. In the refinement step, the detailed behavioral model, i.e. the UML statechart of the Replication Manager is transformed to a TPN. This subnet forms the core of the dependability model by replacing the original, generic subnet. In this way the analysis of application-dependent, specific replication strategies and recovery policies can be supported.

5.2 Rationale of the Refined Fault Modeling

The modeling and analysis methodology described in Sect. 4 is able to deliver a first, system-level estimate of the dependability attributes based on aggregate measures, like fault occurrence and error propagation rates. However, as error propagation depends both on the attributes of the corresponding components and on the *workload distribu-*

tion in the system, a more refined model in the later phases of the design process should reflect these factors separately.

Frequently, a timeliness and/or performance analysis is performed on the system prior to dependability evaluation, especially if the system has to satisfy severe temporal constraints. We sketch here a dependability modeling style which can reuse the models constructed for performance analysis.

The main scope of the refined methodology is the modeling of the effects of permanent and transient operational faults in resources, dominantly implemented in hardware, and the propagation of errors via hardware, software components, and messages, respectively. A crucial problem results from structure altering faults and error propagation effects not included in a functional model of the system. For instance, two functionally independent threads sharing a resource may interact through a parasitic coupling if a fault affects this resource. (The theoretical background would even allow for modeling faults in the very computer core, like the CPU-memory setup. However, such a fine granular modeling is practically infeasible due to the complexity of the model.) Fortunately, each critical application has some well-defined damage confinement regions designed to limit the propagation of errors within them. Accordingly, an appropriate modeling style assures a proper handling of structure altering faults. The main rule is to describe explicitly all the resource sharing within of a damage confinement region. Direct computational faults are restricted in our model to transient faults local to components, and redundancy-based solutions are modeled explicitly.

5.3 The General Resource Model (GRM)

OMG did elaborate the General Resource Model (GRM [25]), providing a standardized notation to describe resource types, their static or dynamic interaction with the system, together with their management. The services required from and delivered by the resources are characterized by means of QoS parameters defined according to the actual analysis objective. GRM is a (sub)profile defined for transformation based analysis. The standard profile offers several notations to describe schedulability, performance, and time. GRM can be extended for several analysis objectives, including dependability assessment.

For the sake of completeness, we shortly summarize the main concepts of GRM (Fig. 8).

In case of *static resource usage modeling* the dynamics of the client-server interaction is neglected, and only a comparison of the *QoS values* required by the *client* and offered by the *resource instance* is performed. A typical system level static quantitative QoS analysis task is a worst case assessment of the sufficiency of the total capacity offered by the resources matched against the total of the requests.

Dynamic usage modeling explicitly describes the order and timing of the client-resource interaction steps. A *resource instance* may offer multiple different kinds of *services*. Each kind of a service use is represented by a *scenario*, a temporally ordered series of *action* executions, as *steps* using specific service(s) offered by a resource. *QoS values* are assigned to the service invocations (*required QoS value*), and to the resource service instances (*offered QoS*).

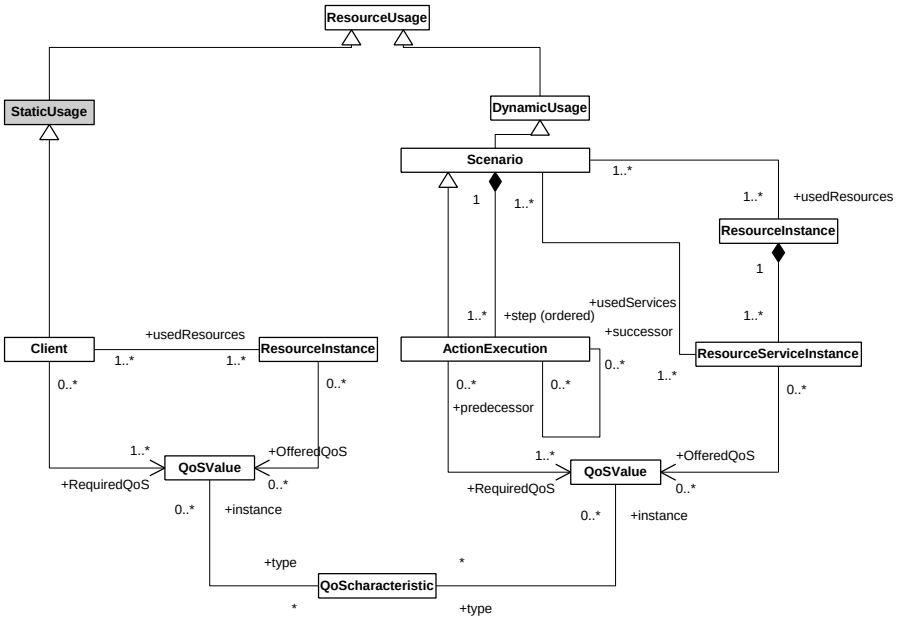

Fig. 8. The OMG General Resource Model

Resources can be classified according to the *protection kind* used either as *unprotected* or *protected*. The latter offers *services* to be used simultaneously only by a single client under an *access control policy* administered by a *resource broker*. A *resource manager* supervises the creation and administration of (typically software) resources according to a *resource control policy*. GRM introduces further technical classifications on resources, like by *purpose* (processor, communication resource, or device), or by activeness (whether they require a trigger or not).

5.4 Modeling of Fault Occurrences

A GRM-based model, as a starting point describes the logic of interactions between the application and the underlying platform. The initial structure of this model is frequently the byproduct of the previous performance analysis. However, dependability analysis necessitates the simultaneous tracing of the information flow in the fault-free and in a faulty case (or all candidate faulty cases) of the system in order to estimate the probabilities of the observable difference in their behavior.

This way the interaction model has to be extended in order to cover all the anticipated faulty cases. This is done by *qualitative fault modeling.* This uninterpreted type of modeling uses a small set of *qualitative values* from an enumerated type, like *{good, faulty, illegal}* to construct an abstract model reflecting the state of the resources and the potential propagation of errors through the system via invocations and messages. The value of *illegal* serves here to model the fault effects resulting in a violation of functional constraints. For instance, a data may not conform to a type re-

lated restriction due to a memory fault forcing it out of the legal range or catastrophic distortions of the control flow may cause a crash. The designer can select the set of qualitative values arbitrarily, according to the aim of the analysis. For instance values of *early* and *late* can extend the domain of qualitative values for timeliness analysis in the potentially faulty case.

Stateful components in the system, like resources can take their actual state from this domain of qualitative values. They change the state upon internal fault activation, repair and error propagation.

- Temporal and permanent operational faults in the resources originate in external effects occurring independently from the processes internal to the system. The appearance of the selected fault or fault mode (in case if a resource can be affected by different kinds of faults) at the fault site is triggered by a separate *fault activation process* independently of the internal functioning of the system (Fig. 9). The fault activation process has a direct access to the resources in order to activate the selected fault(s) by forcing a transition of the state of the component to *faulty*. Frequently, fault activation is restricted, for instance by a single fault assumption. All these restrictions can be included into the fault activation process.
- *External repair actions* can be included into the model as independent processes, as well. Built-in *automated recovery actions* can be modeled as special service invocations forcing the state of the resource to *good*.

Rates can be associated to steps of fault activation and repair processes as QoS values. Their rule is identical to that in the corresponding TPN subnets described in Sect. 4. The changes of fault states in resources are reflected in the activation of different scenarios for service invocations.

5.5 Modeling of Propagation

The *propagation of faulty* and *illegal* values in the scenarios of dynamic usage models (i.e. in message sequence charts or statecharts) represents error propagation during an interaction between a component and resource.

Usage scenarios have to be extended by including fault states as branch conditions if the interaction potentially exposes different behaviors on *good* and *faulty* data or resource states. Usually this extension transforms the scenarios to non-deterministic ones, for instance to express probabilistic error manifestation. The arrival of a *faulty* input data to a stateful resource or component may trigger a *good* to *faulty* transition. Similarly, the invocation of a *faulty* component may result in a *faulty* output delivered by the service.

Quantitative measures can be associated to the different scenarios, including the input request arrival frequencies (rates) of the different kinds of data, probabilities assigned to the non-deterministic branches.

Please note, that the main benefit of using this detailed model is that the quantitative dependability model is cleanly decomposed into the workload model (rate of service invocation), activation and repair (separate processes), and error manifestation (interaction scenarios).

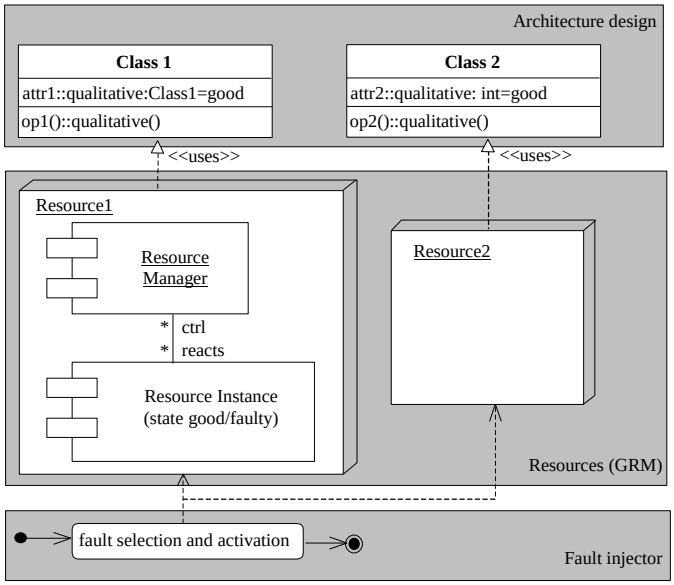

Fig. 9. The fault modeling architecture according to the GRM

The model transformation technology that maps fault activation and usage scenarios to dependability sub-models is the message sequence chart and statechart to TPN transformation introduced in Sect. 5.1 [15].

6 Implementation of the Model Transformation

In HIDE the enriched UML model of the target design was transformed to the mathematical model (TPN for dependability evaluation) by a custom-built model transformation. Later, to avoid the impreciseness of an ad-hoc implementation, the mathematically precise paradigm of graph transformations was selected as the foundation of the definition and implementation of the model transformation. A general framework called VIATRA (Visual Automated model TRAnsformations [11]) was worked out that is also flexible enough to cover the changing UML standard.

In VIATRA, both the UML dialect (standard base UML and its profiles, on one hand restricted by modeling conventions and on the other hand enriched by dependability requirements and local dependability attributes) and the target mathematical notation are specified in the form of UML class diagrams [32] following the concepts of MOF metamodeling [24]. Metamodels are interpreted as type graphs (typed, attributed and directed graphs) and models are valid instances of their type graphs [33]. The transformation steps are specified by graph transformation rules in the form of a 3-tuple (LHS, N, RHS) where LHS is the left-hand side graph, RHS is the right-hand side graph and N is an optional negative application condition. The application of the rule rewrites the initial model by replacing the pattern defined by LHS with the pattern of the RHS. Fig. 10 shows, for example, how an UML object of a class stereotyped as "stateful hardware" becomes an IM node of type SFE_HW with the same name (the

source and target objects are linked together by a reference node in the figure). The operational semantics of the transformation (i.e. the sequence of rules to be applied) is given by a control flow graph. As both LHS and RHS can be specified visually (in the form of UML class diagrams), we have an expressive and easy-to-understand notation for the transformation designer.

In the case of dependability modeling, VIATRA is applied to the UML dialect described in Sect. 3 in order to generate first the IM model according to the metamodel given in Fig. 1 then (in a second step) the TPN model itself. The generic TPN description is then tailored to the input language of the selected tool by a minor post-processing step. The first step of the model transformation from the point of view of the transformation designer is depicted in Fig. 11.

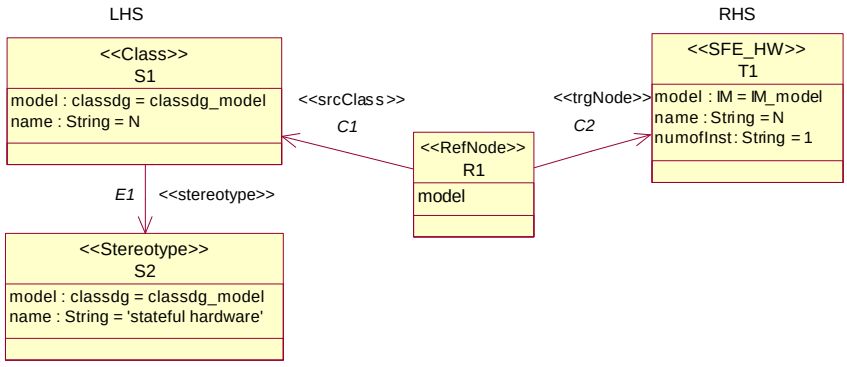

Fig. 10. Definition of a simple transformation rule

The modeler does not have to deal with these internal representations, since the executable program generated from the rules will perform the transformation automatically [32]. The general technological concept of VIATRA is the use of the XMI standard (for arbitrary MOF-based metamodel). Accordingly, the UML CASE tool is interfaced with the transformator by utilizing its XMI export facility. The output of the transformation is again in XMI format, thus the final syntax tailoring steps can be performed by XSLT or dedicated programs.

The first step of our model transformation (from UML structural diagrams to the IM) was defined by only slightly more than 50 rules (e.g. the processing of class diagrams required 28 rules). The subsequent step of constructing the various modules of the TPN can be performed either by similar graph transformation rules or by a Java program [28].

VIATRA is used to construct the fault trees from the statechart diagrams of the redundancy managers as well. Moreover, the generic transformation from UML statecharts to TPN was also developed in this framework.

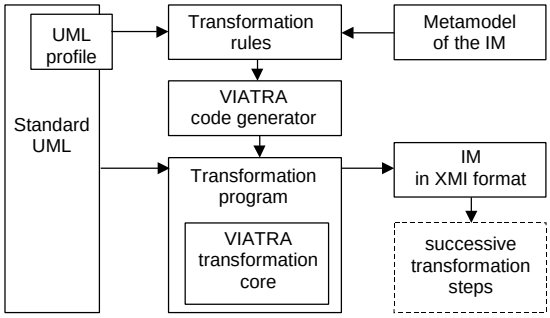

Fig. 11. Application of VIATRA in the first transformation step

7 Assessment of the Approach

The input of the dependability analysis is the set of UML structural diagrams. These diagrams were enriched to allow the designer to identify redundancy structures and provide the parameters needed for the definition of the dependability model. The extensions adhere to standard UML and to the OMG GRM profile. The restrictions imposed on the designer concern only the modeling of redundancy structures. The class based redundancy approach correlates with the usual architecture of distributed fault tolerant object-oriented systems (e.g. Fault Tolerant CORBA [23]).

The semantic correctness of the dependability model relies on the abstraction represented by the IM. In the IM, each element is assigned an internal fault activation and repair process, while relations specify the way for the propagation of failure and repair events. These local effects are represented in the TPN by separate subnets that can be checked formally (contrary to the above UML diagrams, TPN has formal semantics).

The number of model elements in the transformation is in the same order of magnitude as the number of model elements in the UML design. This statement derives from the projection defined when the IM is constructed, since the TPN subnets belonging to IM elements have a fixed number of model elements. A hand-made model could save on the number of modeling elements at the expenses of the modularity.

8 Conclusion

In this paper we described a transformation from structural UML diagrams to TPN models for the quantitative evaluation of availability and reliability. Our transformation is an attempt to build first a quite abstract system-level dependability model with tractable dimensions that can be subsequently enriched and refined by substituting coarse representation of some elements with a more detailed one.

We identified two points in the model where the refined sub-models can be acquired: both the behavioral diagrams describing redundancy and the resource usage (based on the GRM) are candidates of model transformations that result in refined sub-models to be included in the dependability model.

Besides the structural dependability modeling of the production cell benchmark [6], we successfully applied the model refinement approach in the case of FT-CORBA architectures [21].

In addition, further work is ongoing to refine the methodologies described here and to collect evidence on the feasibility and usefulness of the approach. This work is performed in the frame of the PRIDE project, supported by the Italian Space Agency, where the transformation described in this paper is being implemented as a part of the 'HRT UML Nice' toolset specifically tailored for the design and validation of real-time dependable systems.

References

1. Ajmone Marsan, M., and G. Chiola: On Petri nets with deterministic and exponentially distributed firing times. Lecture Notes in Computer Science, Vol. 226, pp. 132-145, 1987.
2. Ajmone Marsan, M., G. Balbo and G. Conte: A Class of Generalized Stochastic Petri Nets for the Performance Analysis of Multiprocessor Systems. ACM TOCS, pp. 93-122, 1984.
3. Allmaier, S., and S. Dalibor: PANDA - Petri net analysis and design assistant. In Proc. Performance TOOLS'97, Saint Malo, France, 1997.
4. Betous-Almeida, C., and K. Kanoun: Dependability Evaluation - From Functional to Structural Modeling. In Proc. SAFECOMP 2001, pp 239-249, Springer Verlag, 2001.
5. Bondavalli, A., I. Majzik and I. Mura: From structural UML diagrams to Timed Petri Nets. European ESPRIT Project 27439 HIDE, Del. 2, Sect. 4, http://www.inf.mit.bme.hu/, 1998.
6. Bondavalli, A., I. Majzik, I. Mura: Automatic Dependability Analysis for Supporting Design Decisions in UML. Proc. Fourth IEEE Int. Symposium on High-Assurance Systems Engineering (HASE'99), November 17-19, 1999, Washington DC, 1999, pages 64-71.
7. Bondavalli, A., M. Dal Cin, D. Latella, I. Majzik, A. Pataricza and G. Savoia: Dependability Analysis in the Early Phases of UML Based System Design. International Journal of Computer Systems - Science & Engineering, Vol. 16 (5), Sep 2001, pp. 265-275.
8. Chiola, G.: GreatSPN 1.5 software architecture. In Proc. Fifth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Torino, Italy, 1991, pp. 117-132.
9. Choi, H., V. G. Kulkarni and K. S. Trivedi: Markov regenerative stochastic Petri nets. Performance Evaluation, Vol. 20, pp. 337-357, 1994.
10. Ciardo, G., J. Muppala and K. S. Trivedi: SPNP: stochastic Petri net package. In Proc. International Conference on Petri Nets and Performance Models, Kyoto, Japan, 1989.
11. Csertán, Gy., G. Huszerl, I. Majzik, Zs. Pap, A. Pataricza, and D. Varró: VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In Proc. 17th Int. Conference on Automated Software Engineering (ASE 2002), Edinburgh, Scotland, 23-27 September 2002, IEEE CS Press, 2002.
12. Frolund, S., J. Koistinen: Quality of Service Specificaton in Distributed Object Systems Design. In Proc. of the 5th USENIX Conf. on Object-Oriented Technology and Systems (COOTS), May 3-7, San Diego, California, USA, 1999, pp 69-89.
13. Ganesh, J. P. , and J. B. Dugan: Automatic Synthesis of Dynamic Fault Trees from UML System Models. Proc. of the IEEE Int. Symp. on Software Reliability Engineering, 2002
14. Goseva-Popstojanova, K., and K. S. Trivedi: Architecture Based Software Reliability. In Proc. of the Int. Conf on Appplied Stochastic System Modeling (ASSM 2000), Kyoto, Japan, March 2000.
15. Huszerl, G., and I. Majzik, A. Pataricza, K. Kosmidis, M. Dal Cin: Quantitative Analysis of UML Statechart Models of Dependable Systems. The Computer Journal, Vol 45(3), May 2002, pp. 260-277

16. Huszerl, G., and I. Majzik: Modelling and Analysis of Redundancy Management in Distributed Object-Oriented Systems by Using UML Statecharts. In: Proc. of the 27th Euromicro Conference, pp. 200-207., Warsaw, Poland, 4-6. September 2001.
17. Huszerl, G., K. Kosmidis: Object Oriented Notation for Modelling Quantitative Aspects. In Proc. Workshop of the International Conference on Architecture of Computing Systems (ARCS 2002), Karlsruhe, Germany, 2002, VDE Verlag Berlin, pp. 91-100.
18. Issarny, V., C. Kloukinas, and A. Zarras: Systematic Aid for Developing Middleware Architectures. In Communications of the ACM, Issue on Adaptive Middleware, Vol 45(6), pp 53-58, June 2002.
19. Laprie, J.-C. (editor): Dependability: Basic Concepts and Terminology. Series Dependable Computing and Fault Tolerant Systems, volume 5, Springer Verlag, 1992
20. Laprie, J.-C. and K. Kanoun: Software Reliability and System Reliability. In M. R. Lyu (editor), Handbook of Software Reliability Engineering, pp 27-69, McGraw Hill, New York, 1995
21. Majzik, I., and G. Huszerl: Towards Dependability Modeling of FT-CORBA Architectures. In A. Bondavalli, P. Thevenod-Fosse (eds.): Dependable Computing EDCC4. Proc. 4th European Dependable Computing Conference, Toulouse, France, 23-25 October 2002, LNCS 2485, Springer Verlag, Berlin Heidelberg, pp. 121-139, 2002.
22. Nelli, M., A. Bondavalli, and L. Simoncini: Dependability Modelling and Analysis of Complex Control Systems: An Application to Railway Interlocking. In Proc. 2nd European Dependable Computing Conference (EDCC-2), pp. 93-110, Springer Verlag, 1996.
23. Object Management Group: Fault Tolerant CORBA. CORBA 2.6, Chapter 25, formal/01-12-63, OMG Technical Committee, http://www.omg.org/, 2001.
24. Object Management Group: Meta Object Facility Version 1.3, http://www.omg.org/, September 1999.
25. Object Management Group: UML Profile for Schedulability, Performance, and Time. Final adopted specification. http://www.omg.org/, 2001.
26. Object Management Group: Unified Modeling Language. Specification v1.4, http://www.uml.org/, 2000.
27. Pataricza, A.: From the General Resource Model to a General Fault Modeling Paradigm? Workshop on Crititcal Systems Development with UML at UML 2002, Dresden, Germany.
28. Poli, S.: Dal Linguaggio di Specifica UML ai modelli a rete di Petri stocastiche: generazione per la valutazione di Dependability. Master thesis (in Italian), University of Pisa, 2000.
29. Rabah, M., and K. Kanoun: Dependability Evaluation of a Distributed Shared Memory Multiprocessor System. In Proc. 3rd European Dependable Computing Conference (EDCC-3), pp. 42-59, Springer Verlag, 1999.
30. Sanders, W. H., W. D. Obal II, M. A. Qureshi and F. K. Widjanarko: The UltraSAN modeling environment. Performance Evaluation, Vol. 21, pp. 1995.
31. SURF-2 User guide. LAAS-CNRS, 1994.
32. Varró, D., and A. Pataricza: Metamodeling Mathematics: A Precise and Visual Framework for Describing Semantic Domains of UML Models. In Proc. UML 2002, International Conference on UML, Dresden, Germany, pp. 18-33, LNCS-2460, Springer Verlag, 2002.
33. Varró, D., G. Varró, and A. Pataricza: Designing the Automatic Transformation of Visual Languages. Science of Computer Programming, 44(2002):205-227, 2002.
34. Walter, M., C. Trinitis, and W. Karl: OpenSESAME: An Intuitive Dependability Modeling Environment Supporting Inter-Component Dependencies. In Proc. of the 2001 Pacific Rim Int. Symposium on Dependable Computing, pp 76-84, IEEE Computer Society, 2001.
35. Xu, J., B. Randell, C.M.F. Rubira-Calsavara and R. J. Stroud: Toward an Object-Oriented Approach to Software Fault Tolerance. In D.K. Pradhan and D.R. Avresky (eds.): Fault-Tolerant Parallel and Distributed Systems. IEEE CS Press, pp.226-233, 1994.