Automated Dependability Analysis of UML Designs

Andrea Bondavalli[1], Istvan Majzik[2] and Ivan Mura[1]

[1] CNUCE/CNR, Via S. Maria 36, 56126 Pisa, Italy, E-mail: a.bondavalli@cnuce.cnr.it, mura@iet.unipi.it
[2] TUB-DMIS, Muegyetem rkp 9, H-1521 Budapest, Hungary, E-mail: majzik@mit.bme.hu

Abstract

This paper deals with the automatic dependability analysis of systems designed using UML. An automatic transformations is defined for the generation of models to capture systems dependability attributes, like reliability. The transformation concentrates on structural UML views, available early in the design, to operate at different levels of refinement, and tries to capture only the information relevant for dependability to limit the size (state space) of the models. Due to the modular construction, these models can be refined later as more detailed, relevant information becomes available. Moreover a careful selection of those, critical, parts to be detailed allows to avoid explosion of the size. An implementation of the transformation is in progress and will be integrated in the toolsets available for the ESPRIT LTR HIDE project.

1. Introduction

Nowadays, an increasing number of critical systems are controlled by means of digital and computer systems to effectively manage the complex control of operations and to provide fast reactions. This pervasive deployment and the growing complexity of computer systems call for effective ways of designing systems and validating designs. This need has contributed to push for the development of standardised and well specified design methods and languages. In this respect, the Unified Modelling Language (UML) [11] is expected to become a de-facto standard for the design of a variety of systems from small control systems to large and complex open systems. An effective design process requires an early validation of the concepts and architectural choices, without wasting time and resources before realising whether the system fulfils its objectives or needs some re-design. The early evaluation of system characteristics like dependability [9], timeliness and correctness, is thus, together with other techniques, necessary to assess whether the system being developed satisfies its targets.

This is the main objective of the European ESPRIT project HIDE. HIDE aims at the creation of an integrated environment where UML-based design toolsets are augmented with modelling and analysis tools. Within this paper, we present one of the activities performed in HIDE, namely an automatic transformation from UML diagrams to Timed Petri Net (TPN) models for model based dependability evaluation. The TPN models output of the transformation can be solved with already available automated tools. To keep the size (state space) of the model at acceptable level and to deal with designs at different levels of refinement, the transformation concentrates on structural UML views, available in the early phases of the design, and tries to capture only the piece of information relevant for dependability. The modular construction of the model favours its extension: the rough, structural model can be refined as the design gets refined, and more detailed, relevant information becomes available. Failure and repair characteristics are assigned by the designer using some extensions of UML.

This paper is organised as follows. Section 2 motivates the need for model-based dependability evaluation, gives the rationale of the transformation and of the opportunity to divide it in two main steps. Section 3, assuming the reader is familiar with UML [11], introduces the limitations to be imposed and the supplementary information required to allow translating the specification into a dependability model. Then, Section 4 introduces the syntax of an intermediate representation and the derivation of the Intermediate model. Section 5 deals with the second step: it defines an abstract TPN syntax and the way the Intermediate model is transformed in a TPN. Finally our concluding remarks and indication of current work follow.

2. Rationale of the transformation

Amongst the approaches commonly adopted to evaluate dependability attributes, analytical modelling has proven to be very useful and versatile. Especially during design, models show their usefulness and potentialities, allowing to compare different architectural and design solutions and to run sensitivity analyses identifying both dependability bottlenecks and critical parameters to which the system is highly sensitive.

Various methods and tools for dependability modelling and analysis have been developed. Among these, Petri nets have been widely accepted in the dependability community because of their powerful representative capabilities, and the relatively cheap solution techniques. Moreover, many automated tools based on Petri Nets are available (e.g. UltraSAN [12], PANDA [3], SPNP [7], GreatSPN [5], SURF2[8]).

Dependability modelling and analysis of complex systems consisting of a large number of components including interactions of redundant hardware and software components as well, pose formidable problems, which arise independently from the design methodology applied, and are thus present in systems designed using UML toolkits. The most important issue to deal with is complexity. The existing tools are not able to deal with the state explosion problems, which plague big size models.

The models for small systems can be obtained by applying a transformation at the fine granularity (e.g. of the Statechart level) of a UML description, which allows to maintain in the model itself other system characteristics like timing aspects and a detailed behavioural description. However, as the systems described grow in size and complexity, this approach is no more viable. To master complexity, a modular modelling methodology is needed so that only the relevant aspects are detailed, still enabling numerical results to be computable.

Our approach aims at building first a quite abstract model, which concentrates on the structure of the system and takes information from the structural UML diagrams. This allows for a less detailed but system-wide representation of the dependability characteristics of the analysed systems offering a significant advantage in terms of controlling the size of the models. Furthermore, it represents a means to analyse dependability attributes of a system while it is still being designed. This way preliminary evaluations of the system dependability during the early phases of the design can be provided. Last, it allows to deal with various levels of details, ranging from very preliminary abstract UML descriptions, up to the refined specifications of the last design phases. The UML higher level models, that is the structural 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 well defined interfaces, such models can be augmented by inserting more detailed information coming from refined UML models of the identified critical parts of the system and provided by other HIDE transformations dealing with UML behavioural and communication diagrams (e.g. the Statechart to Petri net transformation).

This transformation is defined in more steps, where the first has the fundamental task of extracting the relevant dependability information from the mass of information available in the UML description. In this step, an Intermediate model is built, in which we can fix the set of basic and derived failure events, the fault activation, propagation and the repair processes. In a sense, the dependability model is built in this step. The next step allows to define a TPN general enough to postpone the choice of the automatic tool to be used for the analysis to a later stage. A small final step can then be easily performed to translate the model to the specific Petri Net tools selected for performing the analysis.

This multi-stage approach looks attractive for several reasons: i) some peculiarities of UML (package hierarchy, composite objects and nodes, different types of dependencies, etc.) can be resolved resulting in a simple and flat model, ii) 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.

3. UML model elements in the transformation

The transformation derives a TPN from a UML specification using mainly structural diagrams, i.e. use case, class, object and deployment diagrams. Moreover, statechart diagrams are taken into account to deal with the management of redundant resources. The other dynamic diagrams are analysed (when available) to identify further structural relations among components.

Since the information on dependability aspects is typically not included into a system design, we prescribe a set of extensions of the UML standard language. Essentially two types of extensions are necessary: one for identifying redundancy (fault tolerance) structures and the other one for defining the dependability parameters and desired measures. UML already provides standard mechanisms to introduce such extensions into the model. *Tagged values* are pseudo attributes assigned in the form of a pair "tag=value". *Stereotypes* classify the meaning/usage of elements, usually requiring also qualification by tagged values.

One fundamental choice has been made regarding the way redundancy has to be expressed in the UML design. We opted for a "class based" redundancy, which prescribes that components of a redundancy structure must be defined as specific classes. Three basic components are allowed, stereotyped for straightforward identification as follows:

- <<redundancy manager>>: indicates classes (objects) being used for redundancy management and providing the service of the redundancy structure;
- <<variant>> indicates classes of variants;
- <<adjudicator>> indicates comparators, voters, etc.

Dependability related parameters are assigned to elements of the UML diagrams as tagged values. Different sets of parameters are associated to different kinds of UML elements: software and hardware, as well as stateful (having internal state), and stateless (purely functional) elements are distinguished by stereotyping [4]. E.g. the tagged values required for an element stereotyped <<hardware>> and <<stateless>> are as follows:

```
- tagged value "FO = x.y"      (fault occurrence)
- tagged value "PP = x.y"      (ratio of permanent faults)
- tagged value "RD = x.y"          (repair delay)
```

The designer can assign a tagged value one (to be used to instanciate the parameter), two (range for a sensitivity analysis), or no values (the value should be *derived* from the parameters of underlying elements in the hierarchy).

4. From UML to the Intermediate model

The main task of this first part of the transformation is to project the entities and relations of the UML design into the Intermediate Model (IM). The definition of the IM and the transformation are inspired by the approach presented in [10], and by the abstraction of a dependability model which consists of the following general parts:

- *Fault activation processes*, which model the fault occurrence in system elements and result in *basic events.*
- *Propagation processes*, which model the consequences of basic events and result in *derived failure events.* The failure of a system is one of the derived events.
- *Repair processes,* which model how basic or derived events are removed from the system.

The IM is defined as an hypergraph, where each node represents an entity described in the set of UML structural diagrams, and each hyperarc represents a relation between entities. IM nodes have attached a set of attributes, describing the fault activation and the repair processes for the node, and a propagation process for a hyperarc.The generic node of the IM is described by :

```
NODE <name> <type> <attributes>.
```

There are six distinct types of nodes, each with a particular set of attributes, as described in Table 1. The fault_occurrence field identifies a random variable, which represents the time needed for a fault to hit the UML entity the node represents. For stateful entities (HW or SW), the occurrence of faults does not immediately lead to the failure of the entity, but it first generates some erroneous internal state, which eventually brings the entity to failure after a latency time. The field error_latency refers to the process with which errors bring to failure.

Type	**Attributes**
Stateless HW (SLE-HW)	fault_occurrence, repair_delay, permanent/transient
Stateful HW (SFE-HW)	fault_occurrence, error_latency, repair_delay, permanent/transient
Stateless SW (SLE-SW)	fault_occurrence
Stateful SW (SFE-SW)	fault_occurrence, error_latency, repair_delay
Fault-tolerance structures (FTS)	fault-tree
System (SYS)	measure_of_interest

Table 1: Description of IM nodes

The repair_delay attribute specifies a random variable representing the time needed to perform the repair (fault-treatment and/or error recovery, depending on the type of the node) of the UML entity the node represents. The permanent/transient field specifies the relative percentages of permanent/transient faults affecting HW components. The fault-tree [4] field associated to an FTS node describes the way the failures of the elements composing the structure propagate, possibly resulting in the failure of the whole structure. The measure to be evaluated from the final dependability model (either reliability or availability) is associated to only one out of the SYS nodes of the IM.

IM nodes are linked by hyperarcs, described as follows:

```
HYPERARC <type > <from_node>
      <to_node1, to_node2, ..., to_nodek> <attributes>
```

where from_node is the originating node, and to_node1, to_node2,..., to_nodek are the names of the destination nodes of the hyperarc.

Type	**Link**	**Attributes**
Uses the service of (U)	one-to-one	propagation probability
Is composed of (C)	one-to-many	-

Table 2: Description of IM hyperarcs

There are two distinct types of hyperarcs, described in Table 2 together with the respective type of link and the attributes. The type U hyperarc represents a unidirectional client-server relation between node_1 and node_2. Nodes involved in such relation are coupled in terms of failure propagation: whenever the server node_2 fails, the client node_1 may fail with probability given by the field propagation_probability. Also, the U hyperarc prescribes a constraint for the repair of a node. The repair of a node can not be completed until all the used nodes are fully operational themselves. The type C hyperarc links a FTS (or SYS) node to the set of SWEs or HWEs it is composed of. The C relation is used to identify the non-trivial dependencies between a FTS (or SYS) node and its composing elements.

The IM is practically built by projecting the UML entities into IM nodes, the structural UML relations into IM hyperarcs and the tagged values into attributes. Because of the limited space, we only give a flavour of this projection, by sketching it for deployment diagrams and for the interesting case of fault-tolerance structures. A formal and complete definition of the projection is given in [4] in terms of the metamodel elements of the UML.

4.1. Projection of deployment diagrams

Deployment diagrams (Figure 1) show nodes (HW resources) and the deployment of components (run-time SW entities) on them. Nodes are projected into HW elements, objects are projected into SW elements of the IM. Components are projected into single SW elements or into a set of SW elements (if the realisation is available).

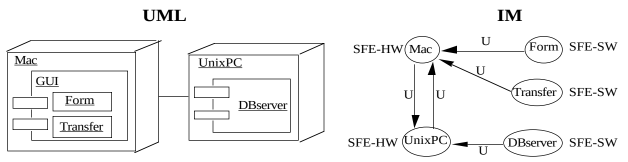

Figure 1: Projection of a deployment diagram into the IM

Deployment and realisation relations (indicating potential error propagation paths) are projected into U hyperarcs. Associations indicate communication and thus potential bi-directional error propagation paths. Each of them is projected into a pair of U hyperarcs of the IM.

4.2. Projection of redundancy

A redundancy structure (identified by the redundancy manager, Figure 2) is projected into an FTS node of the IM connected, using a C hyperarc, to the elements representing the redundancy manager, the adjudicators, and the variants.

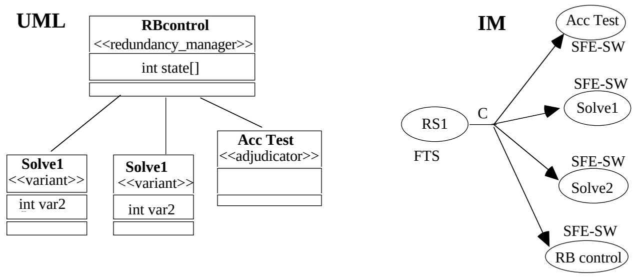

Figure 2: Projection of a simple redundancy structure (recovery block)

The error propagation is described by a fault tree, which can be automatically derived as described in [4].

5. From the IM to TPNs

The second step of our transformation builds a TPN dependability model, by generating a set of subnets for each element of the IM. Due to the limited space, we only give a flavour of the model generation. A detailed description can be found in [4]. A TPN model is composed of set of elements among those listed in Table 3, enclosed between BEGIN-END delimiters.

Element	**Description**
SUBNET	a nested TPN model
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>

Table 3: Elements of a TPN model

Subnets are a convenient modelling notation to make the models clearer. They encapsulate portion of the whole net, thus allowing for a modular and hierarchical definition of the model. The possibility of having nested subnets allows the combination of models at the different levels of detail. Places, transitions and subnets all have a name, which is local to the subnet where they are defined in. Transitions are described by a random_variable and a memory_policy field, which specify the distribution of the delay necessary to perform the associate activities, and a rule for the sampling of the successive random delays from the distribution, respectively. 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.

Notice that the class of TPNs so defined is quite general. It encompasses the class of Generalised Stochastic Petri Nets (GSPN) [1] Deterministic and Stochastic Petri Nets (DSPN) [2] and Markov Regenerative Stochastic Petri Nets (MRSPN) [6]. If the TPN model only contains instantaneous and exponential transitions, then it is a GSPN that can be easily translated into the specific formalism for any of the automated tools able to solve it. 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, 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 of 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 well-specified interface places. For each node of the hypergraph, one or two subnets (basic subnets hereafter) 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.

The basic subnets include a set of places and transitions that are interfaces towards other subnets of the model. Failure subnets contain places called H and F to model the healthy and failed state of the element. Failure subnets of stateful elements also include a place E, to represent the erroneous (not yet faulty) 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), modelled with the introduction of a token in E. After a latency period, modelled by transition latency, this error brings to the failure.

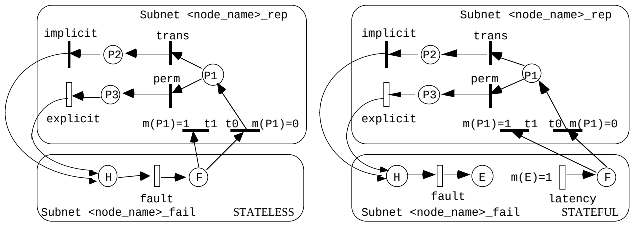

Figure 3: Basic repair subnets for HW nodes

The repair subnet of a node is activated by the failures occurred in the failure subnet of the same node. For instance, Figure 3 shows the pair of failure-repair subnets for stateless and stateful HW nodes. The two transitions implicit and explicit defined inside the repair subnets represent the two different kinds of repair which are needed as a consequence of a transient and permanent hardware fault, respectively. In case the node is stateless, the final effect of the repair is the insertion of a token in place H. In case it is stateful, the repair also removes a token from place E, modelling the error recovery activity.

For each FTS and SYS node, a basic failure subnet containing 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 modelled through the subnets of their 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 UML element object of the dependability evaluation is considered failed.

Notice that all the parameters needed to define the subnets are found in the IM in the obvious fields. If a parameter is not found in the IM, then a more refined submodel is to be included in the final TPN. The nested subnet must exhibit a proper structure for the inclusion to be possible, that is it must show explicit interface elements.

Up to now, the basic subnets of a node are completely separate from the subnets of other nodes. By examining the hyperarcs of the IM, the transformation generates a set of propagation subnets, which link the basic sub-nets. 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 to the server B may propagate to the client A, corrupting its internal state. The propagation subnet B->A shown in Figure 4 models this phenomenon.

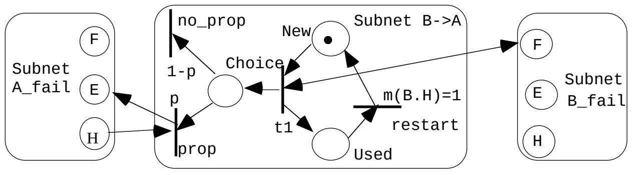

Figure 4: Error of A from the failure of B

The propagation subnet becomes enabled only after element B has failed. At that time, a token is put into place B.F, and the failure propagation subnet is activated. The subnet moves with probability p the token which might be 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.

Consider now a type C hyperarc, linking a FTS node P with its composing elements. 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 elements are recovered. Thus, the fault-tree is translated into a failure propagation subnet, and its dual counterpart is translated into a repair propagation subnet. These two propagation subnets are linked to their duals as well as to the basic failure subnets of all the elements connected by the C hyperarc. Notice that the SYS nodes do not have associated fault-trees, therefore we implicitly associate to them a simple fault-tree representing the OR relation of the failures of all the composing elements.

6. Conclusions

In this paper we described a transformation from structural UML specification to TPN models for the quantitative evaluation of dependability attributes.

Our transformation is an attempt to define the guidelines for the automated generation of models with tractable dimensions, where only those features relevant to dependability are included, and all other information is left aside. We utilised mainly the structural views of UML specifications, to build at first quite abstract models, which can be subsequently refined and enriched by substituting the coarse representation of some elements with a more detailed and precise one, obtained, maybe later in the design process, by some other transformation or analysis technique.

At present, the transformation is being implemented and integrated with the other transformations on a prototype version of the HIDE environment, using Panda [3] for model solution. Experimental evaluations are being conducted on some case study, to assess the efficiency in terms of the computation time needed for the model solution. From a preliminary estimate we can claim that the size of the models automatically generated is proportional to that of the hand-made ones, and proportional to the size of the UML specification representing the input of the transformation as well.

The results of this experimental phase are expected to provide the indications for possible refinements of the transformation, pointing out the parts which need a more accurate treatment. The back-annotation of the results into the UML specification is a matter of our future work, and represents a relevant step towards the complete automation of the transformation. Another point that deserves a deeper investigation is the optimisation of the transformation procedure, particularly the stage of the TPN model solution. Many tools, e.g. [5, 12] could be conveniently employed to efficiently solve TPN models, by exploiting specific repetitive structures that often arise in the models.

Acknowledgements

This work was partially supported by the EC in the framework of ESPRIT LTR 27493 HIDE "High-Level Integrated Design Environment for Dependability". URL: https://asterix.mit.bme.hu:998/. Partners are FAU, Erlangen, Germany, CPR-PDCC, Italy, Technical University of Budapest, Hungary, Mid, Germany, Intecs Sistemi, Italy.

References

[1] M. Ajmone Marsan, G. Balbo and G. Conte, "A Class of Generalized Stochastic Petri Nets for the Performance Analysis of Multiprocessor Systems," ACM TOCS, Vol. 2, pp. 93-122, 1984.

[2] M. Ajmone Marsan and G. Chiola, "On Petri nets with deterministic and exponentially distributed firing times," Lecture Notes in Computer Science, Vol. 226, pp. 132-145, 1987.

[3] S. Allmaier and S. Dalibor, "PANDA - Petri net analysis and design assistant," in Proc. Performance TOOLS'97, Saint Malo, France, 1997.

[4] A. Bondavalli, I. Majzik and I. Mura, "From structural UML diagrams to Timed Petri Nets," European ESPRIT Project HIDE, Deliverable 2, Section 4, 1998.

[5] G. Chiola, "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.

[6] H. Choi, V. G. Kulkarni and K. S. Trivedi, "Markov regenerative stochastic Petri nets," Performance Evaluation, Vol. 20, pp. 337-357, 1994.

[7] G. Ciardo, J. Muppala and K. S. Trivedi, "SPNP: stochastic Petri net package," in Proc. International Conference on Petri Nets and Performance Models, Kyoto, Japan, 1989.

[8] LAAS-CNRS, "SURF-2 User guide," 1994.

[9] J.C. Laprie, "Dependability: a Unifying Concept for Reliable Computing and Fault Tolerance," in "Dependability of Resilient Computers", T. Anderson Ed., BSP Professional Books, 1989, pp. 1-28.

[10] J. C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Architectural issues in software fault-tolerance," in "Software fault-tolerance", M. R. Lyu Ed., Wiley & Sons, 1995, pp. 47-80.

[11] Rational Software * Microsoft * Hewlett-Packard * Oracle* Sterling Software * MCI Systemhouse * Unisys * ICON Computing* IntelliCorp * i-Logix * IBM * ObjecTime * Platinum Technology * Ptech * Taskon * Reich Technologies and Softeam "Object Constraint Language Specification," version 1.1, 1997.

[12] W. H. Sanders, W. D. Obal II, M. A. Qureshi and F. K. Widjanarko, "The *UltraSAN* modeling environment," Performance Evaluation, Vol. 21, pp. 1995.