Automatic Dependability Analysis for Supporting Design Decisions in UML

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

1) CNR Ist. CNUCE, via S. Maria 36, 56126 Pisa, Italy, E-mail (a.bondavalli, ivan.mura}@cnuce.cnr.it
2) TUB-DMIS, Muegyetem rkp 9, H-1521 Budapest, Hungary, E-mail: majzik@mit.bme.hu

Abstract

Even though a thorough system specification improves the quality of the design , it is not sufficient to guarantee that a system will satisfy its reliability targets. Within this paper, we present an application example of one of the activities performed in the European ESPRIT project HIDE, aiming at the creation of an integrated environment where design toolsets based on UML are augmented with modeling and analysis tools for the automatic validation of the system under design. We apply an automatic transformation from UML diagrams to Timed Petri Nets for model based dependability evaluation. It allows a designer to use UML as a front-end for the specification of both the system and the user requirements, and to evaluate dependability figures of the system since the early phases of the design, thus obtaining precious clues for design refinement. The transformation completely hides the mathematical background, thus eliminating the need for a specific expertise in abstract mathematics and the tedious remodeling of the system for mathematical analysis.

1 Introduction

The pervasive deployment of computer systems we are experiencing, and their growing complexity, are increasing the need for effective design. This need has contributed to push for the development of standardized and well-specified design methods and languages, which allow system developers to work with a common platform of design tools. In this respect, the Unified Modeling Language (UML) [17] 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 should also include an early validation of the concepts and architectural choices underlying system design. The early evaluation of system characteristics like dependability [11], timeliness and correctness, necessary to assess whether the system being developed satisfies its targets, becomes especially important for designing systems supporting critical applications.

The validation of designs described using UML is the main objective of the European ESPRIT project HIDE. HIDE aims at the creation of an integrated environment where design toolsets based on UML are augmented with modeling and analysis tools and techniques for the validation and verification of the system design. Designers can thus use UML as a front-end for the specification of both the system and the user requirements. Analysis models are then derived automatically from the UML specification, and solved by the tools available within the HIDE environment. The transformation bridges the gap between a practice-oriented CASE methodology and sophisticated mathematical tools without requiring any knowledge of the mathematical background.

HIDE provides the UML designer with a set of analysis tools, to evaluate the various dependability attributes of the system under design. One of the activities performed in HIDE, namely an automatic transformation from UML Statechart diagrams to Generalised Stochastic Petri Nets models, is described in [9]. In this paper we deal with another HIDE tool, which automatically transforms UML diagrams into Timed Petri Net (TPN) models. The two transformations have been both included in the HIDE environment for the sake of model based dependability evaluation. That in [9] has been conceived for the fine-grained analysis of specific parts of system, and as such it requires the detailed information which become typically available during the later stages of the design refinement. The transformation we will be considering here heads towards a system-level modeling instead, and aims at providing a coarser but overall picture of the system dependability, which can be obtained since the early design phases. The two transformations have thus complementary roles and objectives, and can be flexibly interfaced within HIDE to obtain an accurate modeling only of the critical parts of the system, while keeping an abstract view of the parts that are unrelevant for dependability analysis.

To discuss the benefits offered to the designer by our transformation from UML to TPN, we describe in the following the experience we had in applying the transformation to two versions of the UML design of the production cell example [13] The objective of this study is twofold:

- to understand whether our transformation, the constraints put on the UML designer, and the additions defined, are adequate for properly deriving models and for conducting appropriate (sensitivity) analyses;
- to show the usefulness of such early model based quantitative dependability analyses during the design refinement, in providing hints for changes, identification of dependability bottlenecks, comparisons of alternative choices.

The paper is organized as follows. Section 2, after a short motivation for model based dependability analysis, recalls the transformation, the limitations imposed to the designer, and the supplementary information required. Section 3 introduces the example, both in its basic formulation and in one modified version, and provides the UML design of such systems. Section 4 describes the application of the transformation to the examples, thus deriving the Petri net models to be evaluated. Then, Section 5 is devoted to the evaluation of the models. Sensitivity analyses are performed, which show the deep understanding of the system that can be gained and used in further design refinements. Last, Section 6 concludes the paper.

2 Background

Amongst the approaches commonly adopted to evaluate dependability attributes, analytical modeling has proven to be very useful and versatile. Modeling can be used for system assessment in all phases of the system life cycle. However, it is especially during the design phase that models show their usefulness and potentialities, allowing for the comparison of different design solutions and for the selection of the most suitable one. Also, the sensitivity analyses that can be carried out after modeling allow to identify dependability bottlenecks, thus highlighting problems in the design, and to identify the critical parameters (out of the many that are usually employed at this stage), those to which the system is highly sensitive.

Various methods and tools for dependability modeling and analysis have been developed [14], which provide support to the analyst, during the phases of definition and evaluation of the models. Among these, Petri nets [15, 16, 18] have been widely accepted in the dependability community. Moreover, many automated tools based on Petri Nets are available (e.g. UltraSAN [19], SURF-2 [4], SPNP [8], PANDA [1], TimeNET [10], GreatSPN [7]). Dependability modeling 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, most importantly related to the computational complexity.

To effectively master complexity , the transformation defined in [6] concentrates on higher level UML diagrams, that is the structural views, and tries to capture only the piece of information relevant for dependability analysis. This allows for a less detailed but system-wide representation of the dependability characteristics of the analyzed systems, offering a significant advantage in terms of controlling the size of the models. Furthermore, the structural UML diagrams are available since the earliest development phases, much before the design process has come to a detailed description of system behaviour. This way, preliminary evaluations of the system dependability can be provided [2], and the analyses of models derived from the structural views employed to get precious indications about the critical parts of the system that require a more detailed representation.

Our transformation allows to deal with various levels of details, ranging from preliminary abstract UML descriptions, up to the refined specifications of the last design phases. Indeed, the dependability model of the system is built in a modular way, which provides a good degree of modifiability and extendibility. Rough structural models can be refined later on as more detailed, relevant information becomes available in the design process. A careful selection of those critical parts to be detailed, possibly guided by the analyses performed on the coarse model itself, allows to avoid explosion of the size of the models.

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 the set of basic and derived failure events, the fault activation, propagation and the repair processes are captured. In a sense, the dependability model is built in this step. The next step allows to define dependability models expressed in TPNs, 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 TPN models according to the syntax adopted by specific Petri net tools selected for performing the analysis. Because of the limited space, only a short description of the transformation is recalled. A more detailed and complete definition is given in [5, 6].

2.1 From UML specifications to Petri Net Models

As already stated, the transformation derives a TPN dependability model from a UML specification using mainly structural diagrams, that is *use case, class, object, and deployment diagrams*. Moreover, for some parts, such as the management of redundant resources, the UML behavioral description is taken into account as well, by analyzing UML *statechart diagrams* to identify the relations in the redundancy scheme.

Since the information on dependability aspects is typically not included in a design based on UML, minor extensions of the standard language are needed. First, the designer is *constrained* to identify redundant resources in a predefined way. Next, *extensions* are necessary to provide the designer a controlled interface for the input of parameters and the selection of the desired measures.

UML itself provides standard mechanisms to introduce the required extensions into the model. *Tagged values* are pseudo attributes that can be assigned to UML model elements in the form of a pair “tag = value”. *Stereotypes* introduce a high-level classification (meaning/usage) of model elements. Usually, a stereotype qualifies a base class with tagged values (that must be present).

2.1.1. Design constraints

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 [20], which prescribes that components of a redundant structure must be defined as specific classes qualified by stereotypes. Three basic components of redundancy structures are allowed in a UML design, stereotyped as follows:

- stereotype `<<redundancy manager>>` indicates classes (or objects) being used for redundancy management;
- stereotype `<<variant>>` indicates classes (or objects) of variants;
- stereotype `<<adjudicator>>` indicates adjudicators (comparators, voters, etc.).

According to this approach, a redundancy structure consists of a redundancy manager, variants and adjudicators. Other model elements that do not belong to these types can not be included. The service is available through the redundancy manager, and the redundant elements can not be used separately. An element is participant of a single redundancy scheme only. This restriction allows for a straightforward identification of the redundancy structures, crucial points of the dependability analysis. Future work will be devoted to relax these constraints.

2.1.2. Additional information required

Dependability related parameters are assigned to elements of the UML diagrams as tagged values. The use of tagged values can be prescribed by stereotypes. This way, different sets of parameters can be associated to different types of UML elements. Software and hardware, stateful (having internal state), and stateless (purely functional) elements are distinguished by stereotyping. As an example, we list the tagged values required for an element stereotyped `<<hardware>>` and `<<stateless>>`:

– “`FO = ...`“ (fault occurrence rate)
– “`PP = ...`“ (percentage of permanent faults)
– “`RD = ...`“ (repair delay)

The complete list of tagged values required for all types of elements can be found in [5]. The designer can assign one value intended to instantiate the parameter, or a range for a sensitivity analysis.

2.1.3. The transformation

The main task of the first part of the transformation is to project the elements and relations of the UML design into the Intermediate model (IM, hereafter) which is used to capture the dependability related information. The definition of the IM and the transformation are inspired by the approach presented in [12], and by the abstraction of a dependability model consisting of the following parts:

Fault activation processes, which model the fault occurrence in system elements and result in *basic failure events*. They are determined by environmental conditions, and physical or computational properties of the system.

- *Propagation processes*, which model the consequences of basic events and result in *derived failure events*. They are influenced by the structure of the system, that is interactions, redundancy, fault tolerance schemes. The failure of a system is one of the derived events in this model.
- *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 somewhere in the set of UML structural diagrams, and each hyperarc represents a relation between elements, that is a bit of the structure itself. IM nodes have a set of attached attributes, characterizing the fault activation and the repair processes for a node, and the propagation process for a hyperarc.

The generic node of the IM is described as follows:

NODE <name> <type> <attributes>.

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
F-T structures (FTS)	fault-tree
System (SYS)	measure_of_interest

Table 1: Description of the IM nodes

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

The repair_delay attribute specifies a random variable representing the time needed to perform the repair of the UML entity the node represents. This time to repair covers the time for fault-treatment and/or error recovery, depending on the type of the node. The fault-tree [3] field associated to a FTS node describes the way the failures of the elements composing the structure propagate, possibly resulting in the failure of the whole structure if the fault-tolerance provisions are not able to tolerate them. The measure to be evaluated from the final dependability model (either reliability or availability) is associated to one out of the SYS nodes of the IM.

Nodes of the IM are linked by hyperarcs. An hyperarc is described by the following list:

HYPERARC <type> <from_node>
<to_node_1, to_node_2,...,to_node_n> <attributes>

where from_node is the originating node, and to_node_1, to_node_2,...,to_node_n are the names of the destination nodes of the hyperarc. 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 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. The type C hyperarc links a FTS (or SYS) node to the set of SW or HW nodes representing the entities 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.

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

Table 2: Description of the IM hyperarcs

The IM is built by projecting the UML entities into IM nodes, and the structural UML relations into IM hyperarcs.

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

The second step of our transformation builds a TPN dependability model by examining the hypergraph representing the IM, and generating a set of subnets for each IM element. A TPN model is syntactically composed of the set of elements listed in Table 3. Subnets 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. Transitions are described by a random_variable and a memory policy field, which specify the distribution of the delay necessary to perform the associated 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 possible conflicts. The weights on input and output arcs may be dependent from the marking of the net.

Taking advantage from the modularity allowed by the TPN, the transformation generates 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 entity appearing in the IM, and model all the events that happen locally to the entity , as the failure occurrence and repair processes. At the end of this generation process, the basic subnet(s) of a node are completely disjoint from the subnets of other nodes. Then, by examining the hyperarcs of the IM, the transformation generates a set of propagation subnets, which link the basic subnets. For each pair of nodes for which an hyperarc exists in the IM, a failure propagation subnet and a set of arcs is added to the TPN model. Depending on the type of the hyperarc, a repair propagation subnet can also be generated.

3 The production cell

The production cell [13] has been adopted in the literature as a benchmark for the modeling of reactive systems.

The production cell processes metal plates, which are taken to the cell by a worker. A plate is conveyed to a rotary table by a feed belt. The rotary table is used to move the plate to a position that is proper for a robot to take the plate and place it into a press. The press forges the plate, which is then removed by the robot and given back to the worker. The various elements of the production cell are controlled by software modules, which run on a single computer.

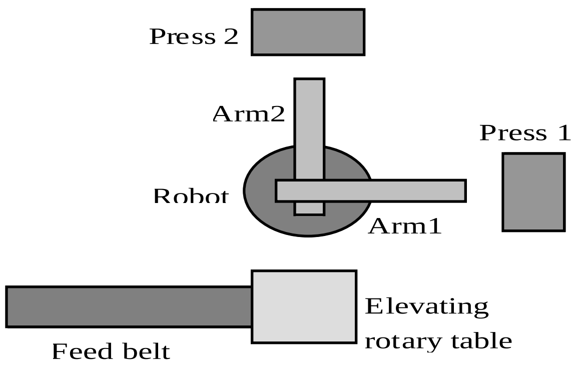

Figure 1: The Production Cell

Here we give the UML description of the basic production cell system, and also the one of a modified production cell , where, in order to tolerate the failure of a press, two redundant presses are used (Figure 1).

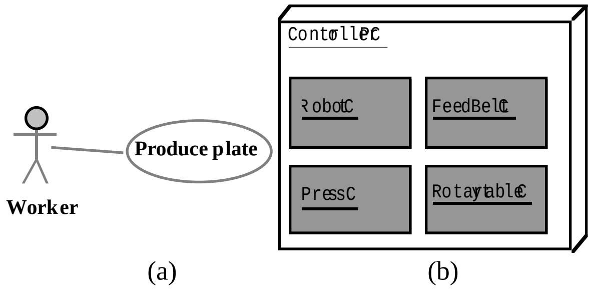

Figure 2: UML use case (a) and deployment (b) diagrams of the production cell

The functionality of the basic production cell system is described by the use case diagram of Figure 2 (a). The four objects of the control software are deployed on a single computer as defined by the deployment diagram shown in Figure 2. (b)

The production cell is modeled by a set of objects described in Figure 3, each representing either a hardware unit (e.g. `RotaryTableHW` is the rotary table, including its sensors and actuators) or part of the controller software (e.g. `FeedBeltC` is a piece of the control software responsible for the feed belt).

The intuitive meaning of the links is that (i) the machines have states (e.g. positions) and operations which are set and sensed by the control software (ii) the software components cooperate to control the safe and efficient operation of the cell and (iii) the machines interact by performing operations on a plate. The object Worker is included to show the interaction with the environment. Each object is an instantiation of a separate class (class diagrams are not included here). The objects are assigned tagged values to describe their dependability parameters. For example, the object `RobotHW` is stereotyped as `<<stateful>>` and `<<hardware>>`, the tagged values prescribed by these stereotypes are `FO=0.004`, `EL=0.0`, `RD=0.0`, `PP=1.0` (since the repair facility is not modeled, the latter two parameters are not used). Similarly, links are assigned parameters of the error propagation. For example, the tagged value of the directed link from `RobotC` to `RobotHW` is `PP=0.9` (propagation probability). For the sake of simplicity, these parameters are not shown in Figure 3.

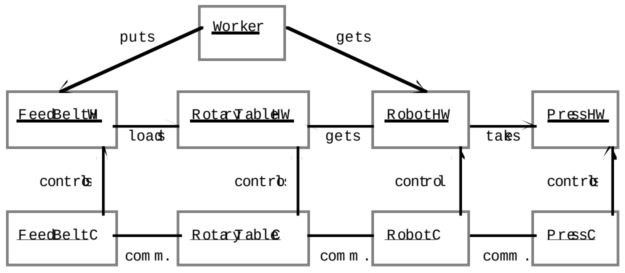

Figure 3: UML Object diagram of the production cell

In order to tolerate the failure of a press, two redundant presses are used in the cell. It is the task of the controller software to hide the failure of a press from the other parts of the system. A separate object (Redundancy `Manager`) is used to perform the task of selecting the available press and forwarding the control to it, this way the pure functional control can be performed by the same object `PressC`. In our case, the redundancy manager implements a cold redundancy scheme: it checks the operation of the first press and in the case of a failure it switches to the redundant second one. The signals from/to the controller are forwarded to/from the active press. The modifications to the object model of this version of the cell are as follows. The `PressHW` (in the basic cell) is replaced by a redundancy structure consisting of the two presses and the redundancy manager. They are identified by stereotypes as follows: objects `Press1HW`, `Press2HW` are both stereotyped as `<<variant>>`, object `RedundancyManager` is stereotyped as `<<redundancy manager>>`.

4 Obtaining TPNs from UML designs

In the first step of the transformation, the UML design is projected into the IM, which represents the relevant entities of the system, their dependability-related parameters and relations. In the second step, the TPN subnets are derived and composed.

The IM of the basic production cell is depicted in Figure 4. The following nodes and relations are used:

- The service of the system is represented by a `SYS` node ("Produce Plate"). The relations of type C identify the nodes that represent objects used directly by the worker. The failure of the system can be recognized when the feed belt or the robot provides improper service (the plate is not taken or no/wrong plate is returned).
- The components of the system are represented either by stateful hardware or stateful software nodes.
- The links among the objects are represented by relations of type `U`.
- The deployment of the software is projected to a set of (unidirectional) `U` relations.

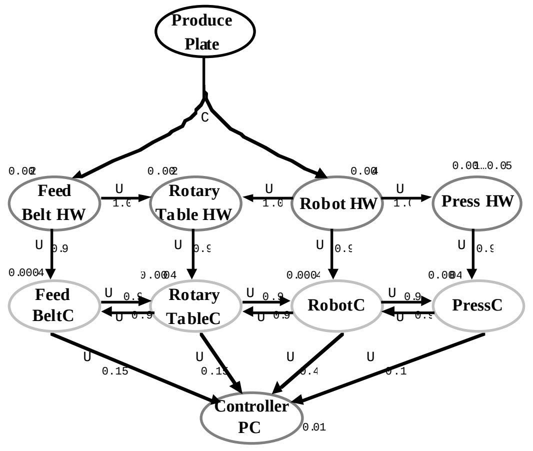

Figure 4: IM of the basic production cell

Each element of the IM is attached a set of attributes, copied from the tagged values of the corresponding UML element, such as failure rate and error propagation probability parameters (included in Figure 4).

Nodes and relations of the IM are translated into subnets of the TPN model, not shown here (see [5]).

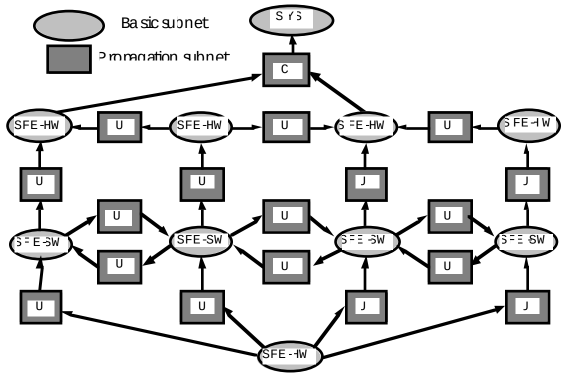

Figure 5: Overall structure of the TPN model

The overall composition of the subnets is presented in Figure 5 (where arrows show the direction of the error propagation). Note that its structure is similar to that of the intermediate model.

In case of the production cell with two redundant presses, the redundancy structure is identified on the basis of the stereotypes. In the IM, this structure is represented by using a `FTS node`, a C hyperarc to link the participants of the scheme (Figure 6 (a)), and a fault tree to model the failure propagation (Figure 6 (b)). In the TPN model a specific subnet represents the Fault tree [5]. In our case, the redundancy structure fails if either the redundancy manager fails or both presses fail.

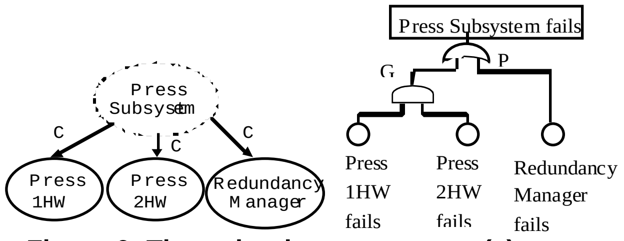

Figure 6: The redundancy structure (a) and the fault tree (b)

5. Evaluation and analyses

The production cell is intended to work continuously for a period of 10 hours each day, followed by a maintenance period. While designing the system, it is important to check the reliability of the production cell during the working time (a 10 hours mission), since the failure of the cell might cause the stoppage of the whole factory. If the reliability is not satisfactory, then the bottleneck has to be found and component(s) of higher reliability should be selected or some methods of fault tolerance (redundancy in hardware, software, information or time) has to be introduced. Such design decisions require, among others, (i) sensitivity analyses of the reliability parameters of system components and (ii) comparison of alternative implementations (structures) of the system.

These analyses are supported by our environment, as the reliability of the production cell can be computed automatically. The transient analysis of the TPN is performed, computing the probability of the failure of the SYS node, i.e. the probability that a token is moved into the place representing the failure of this node. If the repair processes were modeled, then also availability measures could be derived by performing steady-state analysis. It has to be emphasized that these technical issues are hidden from the designer, as he/she is working only at the level of the UML model by defining the parameters and the measure of interest.

To illustrate the kind of analysis performed, we provide the sensitivity of the system to the reliability of the press and compare the reliability of the basic and of the alternative production cell. This does not mean that the press has been identified as the primary dependability bottleneck, there might be other components which show to be even more critical. PANDA [1] was used for the analysis. The reliability of the basic system for various values of the failure rate of the press is presented in Figure 7. It turns out that the reliability of the basic system shows to be sensitive to the reliability of the press, and this dependency can be reduced if two presses are used.

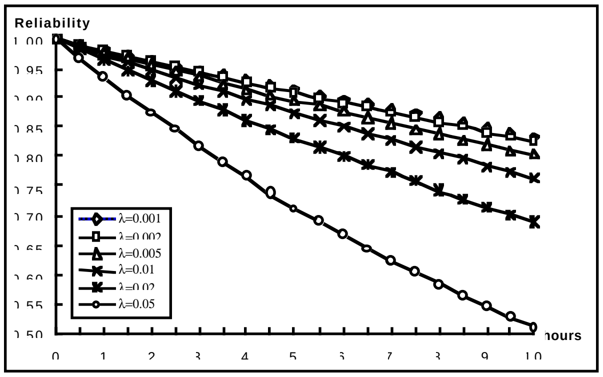

Figure 7: Reliability of the basic production cell

The comparison of the reliability in the two cases is shown in Figure 8, where the reliability of the system at the end of the mission is presented. From this comparison it appears that adding the second press brings significant advantages in reliability only if the failure rate of the press is at least 0.01 per hour.

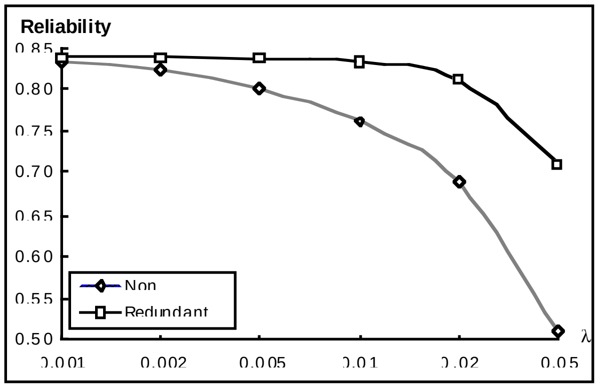

Figure 8: Comparison of the two alternatives of the production cell

To take decisions on the final design, i.e. the choice on which (if any) component to make redundant, requires to analyze many alternatives and to consider also the cost of the redundant press versus the time/money lost due to the failure of the cell, the performance issues of applying two presses also in the fault-free operation, etc. Our transformation focused on the dependability analysis: reliability and availability measures can be provided. However, it appears quite straightforward to include performability measures, which improve the support offered to the designer for the refinement of the system design. This will be part of our future work.

6 Conclusions

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

Our transformation defines 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. It mainly uses the structural views of UML specifications and prescribes a few constraints and additions to UML, to build at first quite abstract models, which can be subsequently refined and enriched. In particular, we have described the experience we made in applying the transformation to two versions of the UML design of the production cell example [13]. The tool PANDA [1] has been applied in working this example out, though the transformation engine can easily be adapted to any other Petri Net tool.

It turned out that the constraints put on the UML designer, and the additions defined are adequate for allowing the models to be properly derived and appropriate analyses to be conducted. We have shown how quantitative dependability analyses at early stages of design do provide hints for changes, identification of dependability bottlenecks, comparisons of alternative choices, and are a valuable help for design refinement.

At present, the transformation is being implemented and integrated with the other transformations on a prototype version of the HIDE environment. Experimental evaluations are being conducted on further case studies, 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 (number of entities) of the UML specification representing the input of the transformation as well.

The results of this experimental phase are expected to provide indications for possible refinements of the transformation, pointing out the parts that need a more accurate treatment and the extensions that appear most desirable (such as the inclusion of reward information).

Acknowledgement

This work has been partially supported by the CEC in the framework of the ESPRIT LTR 27493 HIDE “High-Level Integrated Design Enviroment for Dependability”.

References

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

[2] A. Avizienis, "Building Dependable Systems: How to Keep Up with Complexity," in Proc. IEEE 25-th International Symposium on Fault-Tolerant Computing Systems, Pasadina, U.S.A., 1995, pp. 4-14.

[3] R. E. Barlow, J. B. Fussel and N. D. Singpurwalla, "Reliability and Fault-Tree Analysis," Philadelphia, PA, USA, Society for Industrial and Applied Mathematics, 1975.

[4] C. Beounes, M. Aguera, et al., "SURF-2: a program for dependability evaluation of complex hardware and software systems," in Proc. IEEE FTCS'23, Fault-Tolerant Computing Symposium, Toulouse, France, 1993, pp. 668-673.

[5] A. Bondavalli, I. Majzik and I. Mura, "From structural UML diagrams to Timed Petri Nets," European ESPRIT Project 27439 HIDE, Deliverable 2, Section 4, 1998.

[6] A. Bondavalli, I. Majzik and I. Mura, "Automated dependability analysis of UML designs," in Proc. ISORC'99 2-nd IEEE International Symposium on Object-oriented Real-time distributed Computing, Saint-Malo, France, 1999, pp. 139-144.

[7] 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.

[8] 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.

[9] M. Dal Cin, G. Huszerl and K. Kosmidis, "Evaluation of Safety-Critical Systems based on Guarded Statecharts," in Proc. HASE'99 Fourth IEEE International Symposium on High Assurance Systems Engineering, Washington DC, USA, 1999.

[10] R. German, C. Kelling, A. Zimmermann and G. Hommel, "TimeNET: a toolkit for evaluating non-Markovian stochastic Petri nets," Performance Evaluation, Vol. 24, 1995.

[11] 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.

[12] 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.

[13] C. Lewerentz and T. Lindner, "Formal development of Reactive Systems," Springer Verlag, 1995.

[14] M Malhotra and K. S. Trivedi, "Power-hierarchy among dependability model types," IEEE Transactions on Reliability, Vol. 43, pp. 493-502, 1994.

[15] M. Malhotra and K. S. Trivedi, "Dependability modeling using Petri nets," IEEE Transactions on Reliability, Vol. 44, pp. 428-440, 1995.

[16] J. L. Peterson, "Petri net theory and the modeling of systems," Englewood Cliffs, NJ, Prentice-Hall, 1981.

[17] * 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.

[18] W. Reisig, "Petri nets: an introduction," Springer Verlag, 1985.

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

[20] J. Xu, B. Randell, C. M. F. Rubira-Calsavara and R. J. Stroud, "Towards an object-oriented approch to software fault-tolerance," University of Newcastle Upon Tyne, PDCS-2 Technical Report, 1994.