ESPRIT Project 27493

HIDE

High-Level Integrated Design Environment for Dependability

Deliverable 2: Transformations

Report on the *Specification of Analysis and Transformation Techniques*

A. Bondavalli - CPR/PDCC and CNR/CNUCE
M. Dal Cin - FAU-IMMD3
G. Huszerl - FAU-IMMD3 and TUB
K. Kosmidis FAU-IMMD3
D. Latella - CPR/PDCC and CNR/CNUCE
I. Majzik - TUB
M. Massink - CNR/CNUCE
I. Mura - CPR/PDCC and Univ. of Pisa/Dept. of Inf. Eng.

December 9, 1998

HIDE/T1.2/PDCC/30/v1

Contents

Preface

This is Deliverable 2 of Project ESPRIT 27439 - HIDE (High-level Integrated Design Environment for Dependability). This deliverable is the result of Task 1.2 (Specification of analysis and transformation techniques) of WP1 (Specification of the HIDE Method).

The main contributors to this deliverable have been:

- FAU
- PDCC (Responsible for the deliverable)

This deliverable, together with Deliverable 4, represents the conceptual core of the HIDE project since it contains the detailed description of the main translations from the UML to several validation models for dependability assessment or semantic models for formal verification.

Given the relative independence of each translation from the others this deliverable is organized more as a collection of reports, one per chapter, than as a single document.

Chapter 1, by A. Bondavalli (CPR/PDCC and CNR/CNUCE), M. Dal Cin (FAU-IMMD3), I. Majzik (TUB) and I. Mura (CPR/PDCC and Univ. of Pisa/Dept. of Inf. Eng.) gives a short analysis of existing methods and tools for modeling and assessing quantitative dependability attributes.

Chapter 2, by D. Latella (CPR/PDCC and CNR/CNUCE) gives a short analysis of existing methods and tools for modeling and assessing functional dependability attributes.

The next chapters give the actual definitions of three translations.

Chapter 3, by D. Latella (CPR/PDCC and CNR/CNUCE), I. Majzik (TUB), and M. Massink (CNR/CNUCE) gives the definition of a translation from Statechart Diagrams to Kripke Structures, necessary for formal verification of UML Statechart Diagrams, like model checking, since it actually defines a formal semantics for such diagrams. The translation is fully defined in a formal setting and it is proven to capture the informal requirements on the semantics of UML Statechart Diagrams set in the UML definition book.

The definition of a translation from Structural UML Diagrams to Timed Petri Nets is given in Chapter 4, by A. Bondavalli (CPR/PDCC and CNR/CNUCE), I. Majzik (TUB) and I. Mura (CPR/PDCC and Univ. of Pisa/Dept. of Inf. Eng.). This translation provides a means for assessing quantitative dependability attributes by starting form annotated Structural UML Diagrams. The choice of the domain for the translation allows the specifier to model dependability attributes of systems at a high level of abstraction, so avoiding the (state) explosion problems typical of models based on detailed functional descriptions of systems.

Finally, Chapter 5, by M. Dal Cin (FAU-IMMD3), G. Huszerl (FAU-IMMD3 and TUB), and K. Kosmidis (FAU-IMMD3), describes a translation from dynamic UML diagrams to Generalized Stochastic Petri Nets. The dynamic part of a UML model comprises sequence

diagrams, activity diagrams and statecharts. The translation is defined mainly informally and some sketches of algorithms are given. The subset of statecharts used in the translation comprises also a fault model which allows to evaluate the behaviour of fault-prone systems in their environment.

Analysis of methods and tools for modelling and assessing quantitative dependability attributes

Andrea Bondavalli	CNUCE/CNR and PDCC
Majzik Istvan	TUB
Ivan Mura	University of Pisa and PDCC
Mario Dal Cin	FAU

Abstract

The quantitative analysis of the dependability attributes of computer systems using stochastic modelling is a process that requires ability and experience. Building the model of a system needs the introduction of assumptions, simplifications and abstractions, whose impact on the final results can not be estimated a priori. Also, slight variations in the value of a crucial parameter might cause dramatic changes in the final measures. Moreover, real systems show such a complexity that the definition of the model itself easily becomes an error prone task.

Various methods and tools for dependability modelling and analysis have been developed which provide support to the analyst, during the phases of definition and evaluation of the models. In general, model types used for dependability analysis are in two categories; combinatorial and state-space [13]. In the list below, Markov models and high level approaches which have an underlying Markov model are belonging to state-space models.

1. Combinatorial dependability models

Combinatorial model types include reliability block diagrams, fault trees and reliability graphs. It was shown in [12] that Fault Tree with Repeated Events (FTRE) is the most powerful type. However, the major insufficiency of combinatorial models is that they cannot model several dependencies among model elements, e.g. repair dependency caused by a shared repair facility. These dependencies can be modelled by state-space models. Moreover, algorithms were defined (e.g. in [13]) to transform combinatorial models into state-space models.

2. Markov chains based models

The approach to the modelling based on Markov processes has been widely accepted in the dependability community because of their powerful representative capabilities, and the relatively cheap solution techniques. Constant failure rates are associated with hardware elements and various software failures. This latter is justified by the fact that failed software is not discarded (merely restarted at next execution) thus an equivalent failure rate λ_j can be computed as a product of the constant execution rate of the software and the failure probability:

$$\lambda_j = P_j \lambda$$

where λ is the execution rate, and P_j is a failure probability.

A similar approach for modelling software fault tolerance architectures is adopted in [5]. The service of the system is modelled through execution rates and the fault manifestation process by failure probabilities. The transition rates outputting from the non absorbing states are

$$\lambda_{i,j} = P_{i,j} \lambda_i$$

where i and j denote states, λ_i is the rate associated with the execution in state i, and $P_{i,j}$ represents the probability of the transition from state i to j.

3. Combination of Markov chains and fault trees

A method that combines the simplicity of Fault-Trees with the more powerful representative capabilities of state based approaches has been proposed in [9]. The basic idea of this method is:

- A Markov model describes the system structure (change of configurations due to permanent hardware faults). A single failure state (an absorbing state if repair is not allowed) is included. Parameters of the Markov model are computed using the failure rates of the hardware elements and coverage factors meaning that the system is able to survive and perform the change of configuration (otherwise it goes to the failure state).
- In each configuration (except the failure state) a fault tree details the probability of system failure caused by failures of elements (activation of software and transient hardware faults).

The solution of the Markov chain provides the probability $P_i(t)$ that the system is in state i at time t (n states of the system are encountered). The solution of the fault tree provides $Q_i(t)$, a probability that a failure of the system occurs while the system is in state i. The long-term behaviour (given by the Markov chain) and the short-term behaviour (described by the fault tree) are combined as follows. The probability $Q(t)$ of a system failure at time t is:

$$Q(t) = \sum_{i=1}^{n} Q_i(t) P_i(t)$$

4. High-level modelling tools based on Petri nets

The success of Markov-based approaches for dependability modelling and evaluation has to cope with the increasing complexity of systems, and consequently of the models. The state space combinatorial growth leads to a dramatic increase in the size of Markov chain models, which tend to become difficult to define and computationally intensive to solve. A solution to this issue was offered by modelling tools at a higher level than Markov chains, like Queuing Networks, Stochastic Process Algebras, and Petri nets. Usually, the solution of these models is based on a direct transformation to Markov models. However, high-level models have advantages in the model generation phase, because very compact models can be given even for complex systems, and in the solution phase as well (e.g. state space reduction).

Among these high-level methods and tools, those based on Petri nets models are becoming more and more popular. The reasons of such a success are:

- the natural way in which concurrence, competition and synchronisation are all easily represented within the Petri net formalism;
- the appealing graphical visualisation of the models;
- the ability of Petri nets to deal with different abstraction levels of the analysis.

Many different classes of Petri nets have been proposed over the past decade. The basic untimed class of place/transition Petri nets was augmented with the time for the sake of quantitative analysis of performance/dependability attributes, thus defining the class of Stochastic Petri Nets (SPN) [14]. SPNs only consider activities whose duration is an exponentially distributed random variable. This limitation was overcame with the introduction of Generalised Stochastic Petri Nets [2] (GSPN), which allow for both exponential and instantaneous activities. The stochastic process underlying a SPN and a GSPN model is a discrete space continuous-time homogeneous Markov process. This process must be solved to derive the measures of interest for the system.

Nearly all the tools for dependability modelling and evaluation that are based on Petri net models can be used to define and solve GSPNs. What may be different from one tool to the other is merely a matter of the syntax used to define the model; in this sense, GSPN models can be seen as a standard language which is understood by the majority of the tools for the automated evaluation of dependability attributes, like SURF-2 [11], UltraSAN [1], SPNP [8], GreatSPN [6], TimeNET [10], PANDA [4].

Because of this portability, within the HIDE framework we will be considering the GSPNs as the target class for the quantitative analysis of dependability attributes. Among those cited above, the tool PANDA has been selected as the HIDE front-end tool for this analysis; a brief description of PANDA will be given later in this report. Several extensions of the GSPN class of Petri nets have been farther introduced. These extensions can be distinguished in two classes, depending whether the representative power of the extended models is increased beyond that of GSPNs.

For the extensions that do not increase the representative power, the stochastic processes underlying the models are still Markov processes. In this case the extensions provide useful shorthand notations to represent in a concise way complex dependencies among the elements of the model. The Stochastic Activities Networks (SAN) [15] and the Stochastic Reward Nets (SRN) [16] are two classes of Petri nets that include such extensions. UltraSAN [1] and SPNP [8] are the automated tools for the solution of SANs and SRNs, respectively.

On the other hand, there are classes of Petri nets whose underlying stochastic process is not a simple Markov process. For instance, consider the class of Deterministic and Stochastic Petri Nets (DSPN) [3]. DSPN include transitions having exponentially distributed, immediate and deterministic firing time, and are therefore able to represent models GSPNs can not deal with. The tool TimeNET [10] was especially developed to solve DSPN models. An even more powerful class of Petri nets is represented by the Markov Regenerative Stochastic Petri Nets (MRSPN) [7], which allow for transitions having generally distributed firing times. No automated solution tool exists for MRSPNs, yet.

Of course, the transformation from a UML specification to a Petri net model depends on the particular class of Petri nets we are taking into account. Because of the GSPN class is used, the final models are probably less compact than if a higher expressive class were used. More importantly, as only exponential and immediate transitions are allowed for GSPNs, any activity whose duration does not fit one of these two distributions introduces an approximation in the model.

To leave a leeway for a possible future exploitation of the higher expressiveness and representative power of other classes of Petri nets, in this first phase of HIDE we performed a transformation form UML to an abstract class of timed Petri Nets. This class of Petri nets is quite close to the GSPNs of the target tool PANDA so that its translation into the PANDA model definition language is straightforward. At the same time, this class is sufficiently abstract to represent an intermediate language from which other interesting classes of Petri nets can be targeted.

5. Short description and Assessment of PANDA

PANDA (Petri Net Analysis and Design Assistant [4]) has been developed at the Computer Science Department of the University of Erlangen-Nürnberg, a member of the HIDE consortium. The team of tool developers are available to provide assistance and possibly adapt the source code of the tool for the purposes of the project.

The quantitative, model-based investigation of concurrent interacting systems needs efficient analysis tools. Analysis tools which are based on Stochastic Petri Nets profit from the clear semantics of Petri Nets. PANDA is such a tool. A friendly graphical interface for defining the model is included in the tool. The editor of the net supports an hierarchical description of the models, in which a large size model can be split into nested layers of subnets. Each subnet can be viewed and edited separately. The tool, based on GSPNs implements some syntactical extensions of the GSPN paradigm, like inhibitor arcs, marking-dependent rates and weights of the arcs, enabling functions on the transitions. It also offers the possibility to use different distributions for firing times in order to make the numerical analysis process also amenable to models whose actions do not occur with exponential distributed rates. Hence, phase-type distributed firing times are supported by the numerical analysis process of PANDA. Parametrizing phase-type distributions allows the approximation of arbitrary distributions, for example, that of deterministic actions. PANDA therefore stores additional information within the underlying state space that is unfolded during analysis. This is transparent to the user.

The Markov chains of the GSPN models are directly produced without compilation. Timeless states are eliminated "on the fly". PANDA allows also the qualitative, structural analysis of Petri Nets which can be used for model debugging.

The focus of PANDA is the efficient use of evaluation algorithms. The Multi- Level Method, which is based on state aggregation, was developed for PANDA. This technique is especially suited to evaluate stiff Markov chains, which often occur in the modeling of fault tolerant systems. Through the implementation of that method for shared memory parallel computers, it is possible to take full advance of their memory capacity and acceleration. For the generation of reachability graphs with approximately 8 million arcs and 2.3 million states a speed-up of 5 could be obtained on a Convex Exemplar SPP 1600 with 8 processors. For the model analysis a speed-up of 6.5 was obtained. A PVM-version is currently being developed. For general time distributions there exists also a simulation component.

Since PANDA is primarily used for dependability analysis, a method for integrating (generalized) fault trees, through transforming them into Petri Nets, has also been developed.

Very important for all the transformations is the possibility provided by PANDA to assign state dependent guards and rates to transition firings of the GSPN which represent the actions taking place in the modeled system. Guards assure that a certain action is only possible if the required conditions on the current global system state are met. State dependent rates are needed to have infinite server semantics within the model: the rate at which certain actions occur varies from state to state. Another feature integrated in PANDA are the state dependent arc multiplicities needed, for example, to model synchronizing actions in parallel systems that are non-blocking.

When these guard, rate and multiplicity functions are exhaustively used in the GSPN specification, computation times for state space generation and numerical analysis increase. To compensate for this, PANDA's parallel analysis is very suitable. Moreover as automatic transformations often lead to less efficiently designed models than manual system specifications do, the size of the unfolded state space during numerical analysis becomes even more limiting. This problem is faced by parallelization which in the shared-memory case makes the large global shared memories of modern multipocessors efficiently usable and in the distributed-memory case lets the memories of clustered workstations be combined to store the state space.

At the end of the analysis process the computed results have to be filtered in an suitable way to gain the results that are of interest. Though the GUI for model construction and the analysis components of the PANDA tool are quite efficient, not much has yet been implemented towards an integrated presentation of the analysis results. In earlier releases of the tool, standard GSPN results (e.g. the average number of tokens in a place or the probability of places not being empty) were converted to bar graphs for plotting programs, but this approach seems not very advantageous with PANDA's new, flexible ways of specifying reward measures, and it is certainly not very useful for large Petri Nets.

Therefore, it seems important to spend work for providing a flexible, integrated, easy-to-use facility that enables the modeler to specify which analysis results (e.g. which of the reward measures computed by PANDA) are selected for presentation and how they should be formatted. The result-presentation module should be operable from within the GUI, and it should include an online graphical display of the selected result data as well as output to files in formats that can be post-processed or printed with standard programs (e.g. Gnuplot or PostScript). The first step would be to implement an extension of to PANDA language for specifying result function and reward measures; this extension should allow the modeler to annotate the reward measures of interest for presentation. Then, appropriate forms for controlling technical details of output data formatting and the online display have to be added.

References

[1] "UltraSAN," Center for Reliable and High-Performance Computing Coordinated Science Laboratory, University of Illinois, Urbana, USA. User Manual, 1994.

[2] 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.

[3] 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.

[4] S. Allmaier and S. Dalibor, "PANDA - Petri net ANalysis and Design Assistant," in Proc. Tools Description, 9th Int. Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Saint Malo, France, 1997, pp.

[5] J. Arlat, K. Kanoun and J.C. Laprie, "Dependability Modelling and Evaluation of Software Fault-Tolerant Systems," IEEE Transactions on Computers, Vol. 39, pp. 540-513, 1990.

[6] G. Chiola, "GreatSPN 1.5 Software Architecure," in Proc. 5th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, Torino, Italy, 1987, pp.

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

[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, pp.

[9] J. B. Dugan and M. R. Lyu, "Dependability modeling for fault-tolerant software and systems," in "Software fault-tolerance", M. R. Lyu Ed., Wiley & Sons, 1995, pp. 109-137.

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

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

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

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

[14] M. K. Molloy, "Performance analysis using stochastic Petri nets," IEEE Transactions on Computers, Vol. 31, pp. 913-917, 1982.

[15] W. H. Sanders and L. M. Malhis, "Dependability evaluation using composed SAN-based reward models," Journal of parallel and distributed computing, Vol. 15, pp. 238-254, 1992.

[16] L. A. Tomek and K. S. Trivedi, "Analyses using stochastic reward nets," in "Software fault-tolerance", M. R. Lyu Ed., Wiley & Sons, 1995, pp. 139-166.

Analysis of methods and tools for modeling and assessing functional dependability attributes

D. Latella - CPR PDCC and CNR Ist. CNUCE

In this section we shall briefly discuss methods and tools for formal modeling and verification of systems which are relevant for HIDE. It is outside the scope of this deliverable to give a comprehensive overview on the subject. The reader interested in such an overview is referred to the excellent papers [2, 3], which also contain a very rich bibliography on the subject.

Nowadays, society is highly dependent on computer systems and with no doubt it can be stated that in the near future complex, multimedia, computer-based systems will more and more permeate our society and our activities, including the most critical ones. There is therefore need for higher quality computer systems, both from the reliability point of view and from the performance one. The use of formal methods for the specification and verification of properties of systems is one methodological improvement of the system production process, which, together with other techniques, can make it possible to reach high quality standards.

The study of formal methods for the specification, design, and analysis of distributed systems has been an important research topic over the past decade. Initially, the research in this area has concentrated on the dynamic, functional aspects of such systems, like their observable behaviour, control flow, and synchronization as properties in relative time.

More recently, formal methods for the representation and analysis of functional properties in combination with quantitative aspects of system behaviour have come into focus. They allow the specification of the delay of activities (or, actions) or the probability of actual occurrence of actions.

1 Specification

There are nowadays several notations available for formally specifying the desired behaviour of systems. Among them we can mention Z and VDM, mainly suited for the specification of sequential system, and *process algebras* (CSP, CCS, LOTOS, ACP, etc), *temporal logic* and Statecharts, which instead focus on the behaviour of concurrent systems.

We shall not further elaborate here on specification methods since the UML already provides a notation for that, mainly Statechart Diagrams. We only want to point out that

- within the HIDE framework, in order to use powerful verification tools based on model checking (see below) it is necessary to extend the set of HIDE notations with *temporal logic* ones, to be used as *requirements* specification notations, and
- in order to further pursue the objectives of HIDE it is necessary that, during the second phase, possible deterministically-timed, stochastically-timed and probabilistic extensions of Statechart Diagrams and related semantic models are studied and developed.

1.1 Temporal Logics

Several different temporal logics with different expressive power have been proposed in the literature. Essentially two categories can be mentioned; *Linear Time* TL (LTL) and *Branching Time* TL (BTL). Roughly speaking, in the LTL framework the behaviour of a system is modeled as the set of all the runs of the system where a run is the sequence of states the system resides during a computation. In this context a formula is interpreted on runs and it is satisfied by the system if it holds for all runs. On the other hand, in the BTL framework the behaviour of a system is modeled as a tree on which formulas are interpreted, so that the branching structure of the behaviour due to non-determinism is maintained. During the first phase of HIDE temporal logics is not an issue *per se* since no definitive choice has been made both in the particular temporal logics to be used and on the related model-checking tools. Nevertheless, as a matter of fact, examples have been given using a simple linear time temporal logic.

During the second phase it is important to take into consideration at least one LT logic and one BT logic because of the above mentioned complementary expressive power.

1.2 Quantitative Extension

In recent years a considerable amount of work has been done in the area of extensions of formal specification notations and models with information related to non-functional aspects of system behaviour. Notable examples are deterministically-timed process algebras, stochastically-timed process algebras, probabilistic process algebras, probabilistic and timed temporal logics. The general aim is the definition of general notations and models where functional and non-functional issues are integrated within the same formal framework in such a way that a sound mathematical link is provided between formal specification/verification (of functional properties) and assessment of non-functional parameters (like performance or fault-tolerance attributes).

In order for such extensions to be of any practical use, it is *essential* that the underlying notations provide powerful abstraction mechanisms. This is indeed the case for process algebras and temporal logics.

Within the context of HIDE it is very important that similar studies be performed (at least) with respect to Statechart Diagrams, which also offer high abstraction mechanisms. This actually means equipping Statechart Diagrams with annotations concerning time/probability/stochastic-variables and enrich the operational semantics proposed in this deliverable in order to cope with such extensions.

Timed extensions of statecharts have already been proposed in the literature, but the semantics specificity of UML Statechart Diagrams requires further study and/or adaptation of such results.

The benefits of such studies should be obvious since they not only match the overall goal of HIDE but also allow for integration *within* the same kind of diagrams and in a sound way with respect to formal semantics and formal verification.

2 Verification

Many approaches and tools are nowadays available for formal verification as well. Two main categories of techniques and related tools can be mentioned. Namely *model-checking* and

theorem proving. In the following we shall focus on model checking since theorem proving still requires quite some interation with the (skilled!) user, which is not so much in line with the HIDE objectives.

2.1 Model-checking

In this technique, a finite model of the behaviour of the system is checked in order to verify that a certain property holds for that model. The model has usually the form of a state-transitions graph. Depending on the way the property is expressed, different kind of model-checking techniques can be used:

- *LTL model-checking*
  Properties are expressed as LTL formulas
- *BTL model-checking*
  Properties are expressed as BTL formulas
- *behavioural relations model-checking*
  Properties are expressed as another state-transitions graph

The choice between LTL model-checking and BTL model-checking may depend both on requirements on the expressive power of the unrelying logic and on the availability of efficient tools, in combination with the kind of system modeling language such tools support. In the case of LTL one of the most successful tools is the SPIN model checker. It provides a C-like specification language, PROMELA, plus a simple LTL for the specification of requirements, where basic predicates are specified essentially as PROMELA boolean expressions. The model checker embodies powerful state compression techniques and related search algorithms. It embodies also an approximate representation technique which allows to store almost as many states as it is the size, in *bits*, of the machine main memory. A friendly graphical user interface is available.

During the first phase of HIDE some preliminary study on the implementation of the translation from Statechart Diagrams to Kripke Structures as a translation from Statechart Diagrams to PROMELA has been done. The detailed and rigorous (i.e. proven correct whenever possible) systematic implementation of such a translation is one of the main tasks for the second phase.

In the case of BTL, there are several tools available, using different state-space representation technologies. In particular, the use of binary decision diagrams (BDD) allows for the verification of systems with up to 10^{120} states.

It is worth mentioning here that a model-checker for an action based BTL is available within the JACK toolset, developed at PDCC-CNR/IEI and that a new BDD implementation of such a tool is under way. Several specification languages, including LOTOS, and a graphical interface are also available.

Some study on how to use the results of the work on the translation from Statechart Diagrams to Kripke Structures in the framework of BTL would be of great benefit to HIDE during its second phase since this way the power of LTL model-checking would be complemented by that of BTL, BDD-based model-checking.

Essentially all temporal model-checker provide a counter-example whenever a certain logic formula is not satisfied by a model.

In behavioural relations model-checking, two models of behaviors are compared according to some criterion, expressed as a formal algebraic relation of such models.

For instance one could (naively) compare the set of runs of two behaviours and conclude that a certain behaviour as a sub-behaviour of the other one if the runs of the first one are contained in the runs of the second one. Unfortunately, things are not so simple and the relation between sub-behaviours and behavioural relations is still subject of research. An account of such a research can be found in [1].

Nevertheless, it must be pointed out that such a study is *essential* for providing the UML, and in particular its behavioural part, including Statechart Diagrams, with a notion of sub-behaviour which is based on solid mathematical foundations. Moreover, certain behavoiural relations, namely congruences, are also essential as a formal basis for *reusability* which is another key issue within the UML (and software engineering in general).

Several tools are available for behavioural model-checking. We can mention here the Concurrency Workbench, and, again the JACK toolset which also includes AUTO, a tool for behavioural model-checking developed at INRIA.

We close this section by mentioning the fact that tools for timed/hybrid extensions of the above mentioned models are also available nowadays. Here we can mention UPPAAL, HyTech and KRONOS as examples. These tools should be considered for enriched translations from quantitative extensions of Statechart Diagrams to models like timed automata or hybrid automata.

References

[1] H. Bowman and J. Derrik. A junction between state based and behavioural based specifications. In P. Ciancarini and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*. Kluwer, 1999. (To appear).

[2] E. Clarke, J. Wing, and et. alt. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[3] R. Cleaveland, S. Smolka, and et.alt. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):606–625, 1996.

From Statechart Diagrams to Kripke Structures

D. Latella - CPR PDCC and CNR Ist. CNUCE
I. Majzik - TUB
M. Massink - CNR Ist. CNUCE

1 Informal Description of the translation

In this section we shall present an informal description of the translation from statechart diagrams to Kripke Structures. We shall start with some motivations for the translation, which will be done in Sect. 1.1, followed by a short description of the subset of the UML for which the translation is currently available (Sect. 1.2) and a brief discussion on possible extensions of the UML in order to fully exploit the potentials of the translation (Sect. 1.3). The intuitions behind the translation will be illustrated by means of a small example in Sect. 1.4. For the terminology peculiar to the UML (like event queue, step etc.) we refer to the UML literature [1]. An extended abstract of the work described here can be found in [2].

1.1 Purpose

Formal verification is a hot topic nowadays in the field of software engineering, specially for the development of critical dependable systems.

The use of formal methods for the specification and verification of properties of systems is one methodological improvement of the system production process, which, together with other techniques, can make it possible to reach high quality standards.

The purpose of the translation from statechart diagrams to Kripke Structures is to define a reference formal operational semantics for statechart diagrams within HIDE. Formal semantics are obviously necessary whenever formal verification is at issue: they are a necessary prerequisite for any sensible formal verification or analysis. In particular, the Kripke Structure resulting from the translation can be conveniently used as a basis for model checking, which is a major and widely used technique for formal automatic verification.

A nice aspect of the semantics definition proposed in this deliverable is that it is *parametric* in aspects which are not (yet) completely defined for UML, like the management of the event queue and the priorities. In particular, parametricity of our semantics definition w.r.t. priorities makes it suitable for describing the behaviour of systems under different priority schemas. All the results on the semantics are preserved since they do not depend on the particular priority schema, provided the notion of conflict and orthogonality satisfy the general constraints which are usually satisified by meaningful priority schemas.

1.2 Constraints

During the first phase of the project we considered a strict subset of UML statechart diagrams containing though all the interesting conceptual issues related to concurrency in the dynamic behaviour, like sequentialization, non-determinism and parallelism. Some of the restrictions

we imposed can be easily relaxed in the future, others require some deeper research. In the following we list the restrictions:

- States: History, deep history states as well as action and activity states (and corresponding completion transitions and completion events) are not allowed. Initial (final) pseudostates are used only to identify the initial and final states, their outgoing (incoming) transitions can not have actions.

- Events: Events are restricted to signal and call events without parameters (method execution is not modeled). Time and change events, object creation and destruction events as well as deferred events are not allowed.

- Transitions: Branch segments are not allowed[1]. In the following, compound transitions mean transitions containing join and/or fork segments but no branch segments.

  Completion transitions (without trigger) are not allowed. A transition (characterized by its source and target states, trigger event, guard and action sequence) may appear at most once in a statechart. Interlevel transitions are allowed in our subset.

- Transition labels: In guards, only Boolean combinations of predicates about the current state configuration are allowed, variables and data dependency are excluded. Actions are restricted to generate global events (termination, creation and destruction of objects as well as send clauses are not allowed). Synchronous calls should be modeled by explicit wait states.

- Internal actions of states: Common internal actions as well as "do" actions are not allowed in states.

  A further simplification applies to special internal actions. In the UML semantics, upon taking a transition, the following actions have to be executed in order: *exit actions* of states that are exited explicitly or by default (in the order of the exit hierarchy, i.e. first the lower level ones), normal actions assigned to the transition (in the syntactical order) and then the *entry actions* of states entered explicitly or by default (in the order of the entry hierarchy, i.e. first the higher level ones). Note that the order of entering or exiting regions of a concurrent composite state is not defined.
  We abstract from entry and exit actions of states and handle them in the following together with the normal actions as a (single) sequence of actions executed when the transition fires. Methodologically, it is easy to consider the exit and entry actions as the dynamic semantics keeps track of states that are exited and entered.

- Single statechart: The translation applies to a single statechart diagram. Collections of diagrams must first be reduced to a single one, usually by means of enclosing them into a single parallel state.

- No class hierarchies: In this version of our work we do not deal with more "object-oriented" features like class hierarchies, etc.

[1] They could be resolved by replacing each possible path of segments from the source state to targets with a simple transition. The guard of this transition is the conjunction of the guards on the segments, the action sequence of this transition is the sequence of actions along the segments, following their linear order.

The subset of UML we considered is rather small. Many features which we did not consider are not of conceptual importance from the semantics definition point of view. Others, like the more "object oriented" ones (e.g. object management, inheritance) are not to be considered as slight extensions of the ideas presented in this paper: they need further research. On the other hand, we consider the semantics presented here as an essential first step towards a more complete model for statecharts.

1.3 Extensions

The translation as such does not require any extension to the UML. When using the translation for formal verification, depending on the kind of verification one wants to perfom there might be the need of additional information. For the purpose of this deliverable and as an example of how our semantics could be used, we assume model checking as the verification technique to be used. In such a technique the system designer produces a model of the behaviour of the system or subsystem (s)he has to design and the model checking tool checks if such a model satisfies a certain requirement. In the context of HIDE, the system behaviour is modeled by a statechart diagram. In the context of model checking the requirement is to be specified as a Temporal Logics formula. In this deliverable we consider a Linear Time Temporal Logic.

An informal description of the logics is given below. Examples of its use are given in the next section.

Given a statechart diagram, we assume there exist a predicate $in(s)$ for each state s of the statechart. The meaning of $in(s)$ is that state s is in the current *state configuration* (simply *configuration* in the sequel). We will also use the generalization of the predicate $in(s1, \ldots, sn)$ meaning that the current configuration contains *all* the states listed in the in predicate. Moreover, the notation $[e1, \ldots, en]$ will be used for denoting the fact that the events in the current event queue are $e1, \ldots, en$, where $e1$ is the first element in the queue and en is the last one (here a FIFO discipline is assumed)[2].

Predicates of the form $in(s)$ and $[e1, \ldots, en]$ will be called *atomic formulas*. A *formula* can be either an *atomic formula* or a composition of *formulas*. We assume usual boolean composition operators so, if $f1$ and $f2$ are formulas, then also $f1$ AND $f2$, $f1$ OR $f2$, NOT $f1$ and $f1 ==> f2$ are formulas and their meaning is the standard one.

For example, the formula $in(s0)$ AND $[e0]$ means that state $s0$ is in the current configuration and $e0$ is the only event currently in the event queue.

As a second example, suppose state $s2$ is a substate of state $s1$. Then the formula $[e10, e25, e0] ==> in(s1, s2)$ is violated if the current queue is composed by events $e10, e25$, and $e0$ with $e10$ ($e0$) being the first (last) element and $s1$ or $s2$ are not in the current configuration. In the following, we shall call a pair (configuration, event queue) a *status*.

The set of formulas of interest for us is enriched as follows (where f is any formula). $[]f$ (to be read as "f forever") informally means "f holds in every status of every run of the system". $<> f$ (to be read as "evetually f") means "In every run of the system there is a status in which f holds". Finally $f1\ U\ f2$ (to be read as "f1 until f2") means "In every run of the system there is a status in which $f2$ holds and in all the previous statuses (in the same run) $f1$ holds".

So, for instance, $[][e10, e25, e0] ==> in(s1, s2)$ means that we require that *whenever* the

[2] More interesting predicates on the queue can be defined, but we leave them out here for simplicity reasons.

queue is composed by the events $e10, e25$, and $e0$, the current configuration must contain states $s1$ and $s2$.

Obvioulsy formulas containing the above *temporal* connectives $[], <>, U$ can in turn be composed using logical as well as temporal connectives.

For the sake of readability, it is often convenient to assign names to formulas by means of defining equations and then use such names (recursive definitions are not allowed here). For example, the above formula coud be rewritten as $[]f$, or as $[](p ==> q)$ where:
$p = [e10, e25, e0]$
$q = in(s1, s2)$
$f = p ==> q$

1.4 Example

In this section we shall informally describe the translation by means of a simple example. Consider the statechart diagram of Figure 1.

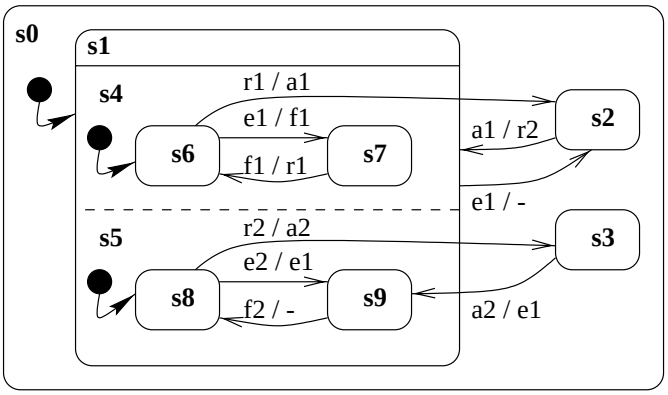


Figure 1: Example of an UML statechart

The first step of our translation is a purely syntactical one and consists in translating the statechart diagrams into what is usally called an extended hierarchical automaton. Extended Hierarchical Automata can be seen as an *abstract syntax* for statechart diagrams in the sense that they abstract from the purely syntactical/graphical details and describe only the essential aspects of the statechart. Thus they are composed of simple sequential automata related by a *refinement function*. A state is mapped via the refinement function into the set of (parallel) automata which refine it.

Our sample sample statechart diagram is mapped into the extended hierarchical automaton of Fig 2.

It should be already clear that the extended hierarchical automaton of Fig. 2 can be taken as an alternative representation for the statechart of Fig. 1. In fact there is a clear correspondence between the states of the two structures. Also the refinement of a state into one or more substates in the statechart is properly represented by the refinement function ρ; in our example we have $\rho s1 = \{A1, A2\}$ and $\rho s = \emptyset$ for any other state s. In the figure this is represented by dotted arrows.

Non-interlevel transitions are represented in the obvious way. Consider now the interlevel transition from $s6$ to $s2$ in Fig. 1. Such a transition is represented in the extended hierarchical automaton by the transition from $s1$ (the highest ancestor of $s6$ "crossed" by the transition

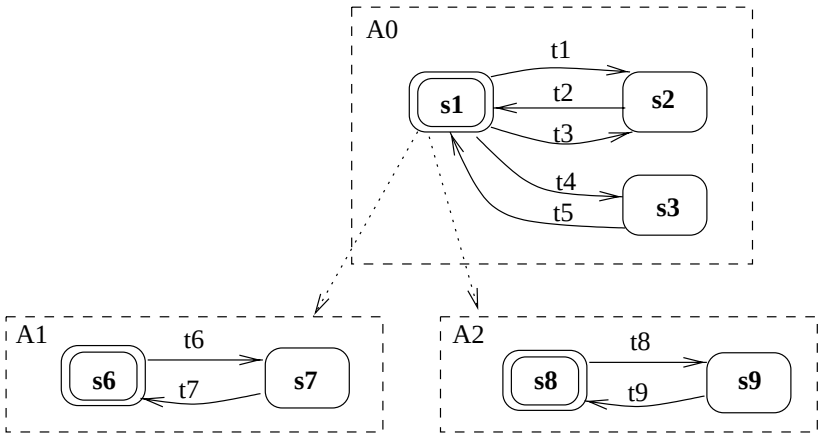


Figure 2: Example of an Extended Hierarchical Automaton

in the statechart) to $s2$, named $t1$. The indication of the fact that the real "origin" of such a transition is state $s6$ is coded in the *label* of the transition (not shown in the figure). In particular, it is coded in what is called the *source restriction* of the transition. The source restriction of $t1$ will be $s6$. In general, for join transitions the source restriction will be a set of pairwise orthogonal states. In the label we will also find the event which *triggers* the transition and the corrensponding *actions* to be performed when the transition is fired. Finally, in the label of a transition, we also find the so called *target determinator*. The target determinator explicitly lists *all* the basic states which must be reached when a transition is fired. For example, the transition from $s3$ to $s9$ in Fig. 1 is represented in Fig. 2 by the transition labeled $t5$, the target determinator of which is $\{s6, s9\}$.

The complete information related to the transition labels for our extended hierarchical automaton is given by the table below:

t	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	$t8$	$t9$
$EV\ t$	$r1$	$a1$	$e1$	$r2$	$a2$	$e1$	$f1$	$e2$	$f2$
$SR\ t$	$\{s6\}$	$\emptyset$	$\emptyset$	$\{s8\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$TD\ t$	$\emptyset$	$\{s6, s8\}$	$\emptyset$	$\emptyset$	$\{s6, s9\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$AC\ t$	$a1$	$r2$	ϵ	$a2$	$e1$	$f1$	$r1$	$e1$	ϵ

Table 1: Transition Labels

In the following we shall refer to the extended hierarchical automaton of Fig. 2. The initial *configuration* is the set of states in which the system resides in the beginning of any run, namely $\{s1, s6, s8\}$. Suppose initially the event queue contains only event $e2$ which is then selected. Then transition $t8$ is fired, event $e1$ is generated and the system will move to configuration $\{s1, s6, s9\}$. In our semantics, the event generated ($e1$) is put back into the event queue. At this point both transition $t3$ and $t6$ are anabled, but $t6$ will fire since, according to the UML statechart diagrams priority rule, it has priority over $t3$.

The above procedure can be modeled by using an automaton. The states of such an automaton are the statuses of the system, i.e. pairs (configuration, event queue). Its transitions represent the *steps* of the system and, for easyness of exposition, here they are labeled by the name(s) of the transition(s) of the extended hierarchical automaton which are fired in the

corresponding steps. The automaton for our sample system is given in Fig. 3.

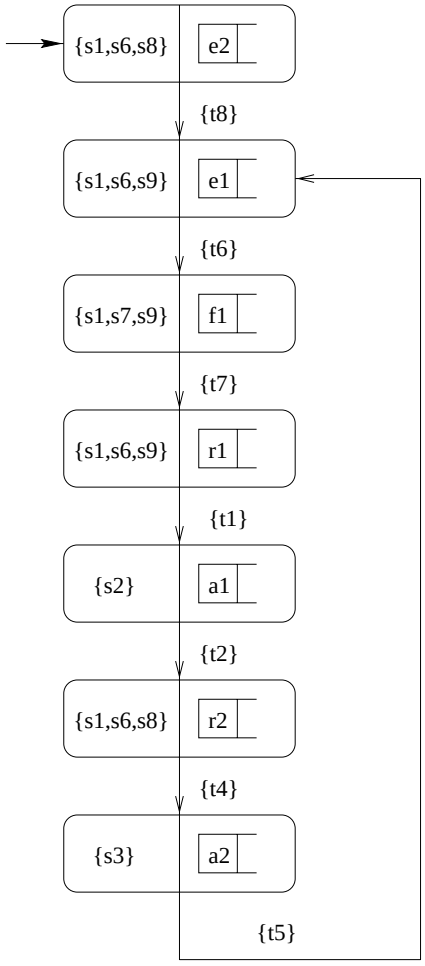


Figure 3: Example of an Extended Hierarchical Automaton

In the formal semantics, the existence of a step from status $(\mathcal{C}, \mathcal{E})$ to status $(\mathcal{C}', \mathcal{E}')$ is modeled as an assertion which needs to be proven within a formal system of logical deduction. So the semantics definition amounts to a set of deduction rules. All the relevant details are given in Sect.2.

We close this section by stating some typical temporal logics properties related to our sample statechart and its semantics automaton.

A first simple property, which is satisfied by our system is $[e2]$ AND $in(s1, s6, s8)$. It simply states that the initial status of the system is $(\{s1, s6, s8\}, e2)$ which is indeed the case.

The following is a typical response property stating that whenever $s2$ is entered, sooner or later (starting from that point in the run) $s3$ is reached: $[](in(s2) ==><> in(s3))$. You can imagine $s2$ as a state entered immediately after a request of some service is issued and $s3$ as the state entered immediately after such a request is granted. So the above requirement means that every request must eventually be granted and our system fulfills it.

Suppose now we want to model the following requirement: "Every request must be preceded by a distinct reception of event $e2$". It should be easy to understand that the preceding statement is equivalent to the following (maybe more tedious but certainly more precise): "In

every run of the system, and in every status reached during the run, call it S, the following must hold: it must not be the case that a request is made without $e2$ having occurred from status S to the status in which the request is made". This statement is coded into the following formula: $[]NOT(NOT[e2]Uin(s2))$. This formula is not satisfied by our system since the formula $NOT(NOT[e2]Uin(s2))$ is not satisfied by the statuses in the loop.

We conclude this section by pointing out that the way we have expressed temporal logics formulas here is probably not the most user friendly one can conceive. There are ways for expressing the above formulas in which the user does not need to be cercerned with the notational details. In fact there are also graphical notations to serve the purpose of formulas specifications. All these issues are to be dealt with at the level of user interface design and this is the reason why we do not deal with them here. Here we simply wanted to sketch the main concepts of model checking in a as much intuitive fashion as possible.

2 Formal Description of the translation

In the following we will present a formal definition of the translation from Statechart Diagrams to Kripke Structures. Following the approach proposed in [4], we will proceede in two steps: we first map Statechart Diagrams into (a slightly modified variant of) *Extended Hierarchical Automata,* which provide essentially an abstract syntax for diagrams, and then we define a formal operational semantics for Extended Hierarchical Automata, which amount to a translation of Extended Hierarchical Automata to Kripke Structures.

In Section 2.1 the intermediate model is introduced. Section 2.2 defines the translation from a subset of UML statecharts to Extended Hierarchical Automata. Section 2.3 introduces our formal operational semantics of Extended Hierarchical Automata.

All proofs are omitted here. They can be found both in [3] and in Deliverable 4 of Task 3.1 of the HIDE Project.

The results presented in this Deliverable are also summarized in the paper "Towards a Formal Operational Semantics of UML Statechart Diagrams" by D. Latella (CNR-PDCC, Pisa), I. Majzik (TUB, Budapest) and M. Massink (CNR, Pisa) to appear in the proceedings of the Third IFIPTC6/WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems to be held in Florence, Italy, on February 15-18, 1999.

2.1 Extended Hierarchical Automata

In this section we recall the notion of Extended Hierarchical Automata defined in [4], although our notation is slightly different from that used therein. We start by the notion of (sequential) automaton[3].

Def. 1 (Sequential Automata) *A sequential automaton A is a 4-tuple $(\sigma_A, s^0_A, \lambda_A, \delta_A)$ where σ_A is a finite set of* states *with $s^0_A \in \sigma_A$ the* initial state, *λ_A is a finite set of* transition labels *and $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ is the* transition relation.

[3] In the following we will freely use a functional-like notation in our definitions where: (i) currying will be used in function application, i.e. $f\ a_1\ a_2 \ldots\ a_n$ will be used instead of $f(a_1, a_2, \ldots, a_n)$ and function application will be considered left-associative; (ii) for function $f : X \longrightarrow Y$ and $Z \subseteq X$, $f\ Z = \{y \in Y \mid \exists x \in Z.\ y = fx\}$, $rng\ f$ denotes the *range* of f and $f_{|Z}$ is the restriction of f to Z.

We shall use a particular structure for the labels in λ_A which will be described later. For sequential automaton A let functions $SRC, TGT : \delta_A \longrightarrow \sigma_A$ be defined as $SRC(s, l, s') = s$ and $TGT(s, l, s') = s'$. Extended Hierarchical Automata [4] are defined as follows:

Def. 2 (Extended Hierarchical Automata) *An extended hierarchical automaton H is a 3-tuple (F, E, ρ), where F is a finite set of sequential automata with mutually disjoint sets of states, i.e. $\forall A_1, A_2 \in F.\ \sigma_{A_1} \cap \sigma_{A_2} = \emptyset$ and E is a finite set of* events*; the* refinement function *$\rho : \bigcup_{A \in F} \sigma_A \longrightarrow 2^F$ imposes a tree structure to F, i.e. (i) there exists a unique root automaton $A_{root} \in F$ such that $A_{root} \notin \bigcup rng\ \rho$, (ii) every non-root automaton has exactly one ancestor state: $\bigcup rng\ \rho = F \setminus \{A_{root}\}$ and $\forall A \in F \setminus \{A_{root}\}.\ \exists_1 s \in \bigcup_{A' \in F \setminus \{A\}} \sigma_{A'}.\ A \in (\rho\ s)$ and (iii) there are no cycles: $\forall S \subseteq \bigcup_{A \in F} \sigma_A.\ \exists s \in S.\ S \cap \bigcup_{A \in \rho s} \sigma_A = \emptyset$.*

We say that a state s for which $\rho s = \emptyset$ holds is a *basic* state. An example of an extended hierarchical automaton is presented in Figure 2. Here $F = \{A0, A1, A2\}$, and state $s1$ of the root $A0$ is refined by $A1$ and $A2$: $\rho\ s1 = \{A1, A2\}$. All states except $s1$ are basic. Initial states are indicated by double boxes.

In the sequel we will implicitly make reference to a generic extended hierarchical automaton $H = (F, E, \rho)$.

Every sequential automaton $A \in F$ characterizes an extended hierarchical automaton in its turn: intuitively, such a extended hierarchical automaton is composed by all those sequential automata which lay below A, including A itself, and has a refinement function ρ_A which is a proper restriction of ρ.

Def. 3 *For $A \in F$ the* automata, states, *and* transitions under *A are defined respectively as $\mathcal{A}\ A = \{A\} \cup (\bigcup_{A' \in \left(\bigcup_{s \in \sigma_A} (\rho_A s)\right)} (\mathcal{A}\ A'))$, $\mathcal{S}\ A = \bigcup_{A' \in \mathcal{A}\ A} \sigma_{A'}$, and $\mathcal{T}\ A = \bigcup_{A' \in \mathcal{A}\ A} \delta_{A'}$*

The following lemmata state some useful properties of $\mathcal{A}\ A, \mathcal{S}\ A$ and $\mathcal{T}\ A$.

Lemma 1 *For $A, A', \bar{A}, \bar{A}' \in F$, $s \in \mathcal{S}\ H$, the following holds: (i) $A' \in \mathcal{A}\ A$ implies $\mathcal{A}\ A' \subseteq \mathcal{A}\ A$, $\mathcal{S}\ A' \subseteq \mathcal{S}\ A$, and $\mathcal{T}\ A' \subseteq \mathcal{T}\ A$. (ii) $A, A' \in (\rho s), A \neq A', \bar{A} \in (\mathcal{A}\ A), \bar{A}' \in (\mathcal{A}\ A')$ implies $\mathcal{A}\ \bar{A} \cap \mathcal{A}\ \bar{A}' = \mathcal{S}\ \bar{A} \cap \mathcal{S}\ \bar{A}' = \mathcal{T}\ \bar{A} \cap \mathcal{T}\ \bar{A}' = \emptyset$. (iii) $s \in (\mathcal{S}\ A)$ implies $\exists_1 A' \in (\mathcal{A}\ A).\ s \in \sigma_{A'}$.*

Lemma 2 *For $A, A' \in F$, $s \in \sigma_A$, $s' \in \sigma_{A'}$ the following holds: $s' \in \mathcal{S}\ (\rho s) \ \Rightarrow\ A' \in (\mathcal{A}\ A)$.*

The definition of sub-extended hierarchical automaton follows:

Def. 4 (Sub-Extended Hierarchical Automata) *For $A \in F$, (F_A, E, ρ_A), where $F_A = (\mathcal{A}\ A)$ and $\rho_A = \rho_{|(\mathcal{S}\ A)}$, is the extended hierarchical automaton characterized by A.*

In the sequel for $A \in F$ we shall refer to A both as a sequential automaton and as the sub-extended hierarchical automaton of H it characterizes, the role being clear from the context. H will be identified with A_{root}. Sequential Automata will be considered a degenerate case of Extended Hierarchical Automata. In Figure 2, automaton $A0$ refers to both the sequential automaton $A0 = (\{s1, s2, s3\}, s1, \lambda_A, \{t1, t2, t3, t4, t5\})$ and the extended hierarchical automaton $H = (\{A0, A1, A2\}, E, \rho)$ where $\rho\ s1 = \{A1, A2\}$.

Def. 5 (State Precedence) *For $s, s' \in \mathcal{S}\ H$, $s \prec s'$ iff $s' \in \mathcal{S}\ (\rho\ s)$. Let also $\preceq$ denote the* reflexive *closure of $\prec$.*

Proposition 1 *Relation* $\preceq$ *is a partial order.*

The following holds of $\preceq$:

Lemma 3 *For* $s, s', \bar{s} \in (\mathcal{S}\ H)$ *the following holds:* $(s \preceq \bar{s}) \wedge (s' \preceq \bar{s}) \Rightarrow (s \preceq s') \vee (s' \preceq s)$

Lemma 4 *For all* $A, A' \in F$ *and all* $s \in \sigma_A$, $s' \in \sigma_{A'}$ *the following holds:* $s \preceq s' \Rightarrow (\mathcal{S}\ A) \cap (\mathcal{S}\ A') \neq \emptyset$

Def. 6 (Orthogonal States) *Two states* $s, s' \in \mathcal{S}\ H$ *are* orthogonal, *written* $s \ || \ s'$, *iff* $\exists s'' \in (\mathcal{S}\ H), A, A' \in (\rho s''). A \neq A' \wedge s \in \mathcal{S}\ A \wedge s' \in \mathcal{S}\ A'$

Obviously $s \ || \ s'$ implies $s \neq s'$. Orthogonal states in Figure 2 are, among others, $s6$ and $s8$, since $s6 \in \mathcal{S}\ A1$, $s8 \in \mathcal{S}\ A2$ and there is $s1$ for which $A1, A2 \in \rho\ s1$.
It is easy to see that orthogonal states enjoy the following property:

Lemma 5 *For all* $s, s' \in \mathcal{S}\ H$ *the following holds:* $s \ || \ s' \Rightarrow s \not\preceq s'$

We say that $S \subseteq \mathcal{S}\ H$ is a *set of pairwise orthogonal states* iff $\forall s, s' \in S.\ (s \neq s' \Rightarrow s \ || \ s')$. An obvious consequence of the above lemma is that for $S \subseteq \mathcal{S}\ H$ a set of pairwise orthogonal states, the following holds: $s, s' \in S$ and $s \preceq s'$ implies $s = s'$. The following definition lifts $\preceq$ to *sets of* states:

Def. 7 *For all* $S, S' \subseteq \mathcal{S}\ H$, $S \preceq^s S'$ *iff* $\forall s \in S.\ \exists s' \in S'.\ s \preceq s'$

Notice that $\preceq^s$ is only a preorder. Take for instance $S = \{s1, s2\}$ and $S' = \{s2\}$ with $s1 \preceq s2$. Now $S \preceq^s S'$ and $S' \preceq^s S$, but $S \neq S'$. The following proposition holds:

Lemma 6 *For* $S, S' \subseteq \mathcal{S}\ H$ *sets of* pairwise orthogonal states $S \preceq^s S' \wedge S' \preceq^s S$ *implies* $S = S'$.

For the purpose of representing statechart diagrams using Extended Hierarchical Automata we shall require transition labels of transitions t of sequential automata $A \in F$ be 5-tuples (sr, ev, g, ac, td) where (i) the *source restriction* $sr \subseteq \mathcal{S}\ (\rho(SRC\ t))$ is a set of pairwise orthogonal states; (ii) $ev \in E \cup \{-\}$ is the event which *triggers* the transition, with $-$ representing that no event is required for triggering the transition; (iii) g Is the *guard*, i.e. a boolean expression on states (which we shall not further specify in this paper); (iv) $ac \in E^*$ is the sequence of events to be generated when the transition is fired, i.e. the *sequence of actions* to be executed; and the *target determinator* $td \subseteq \mathcal{S}\ (\rho(TGTt))$ is a *maximal* (under set inclusion) set of pairwise orthogonal *basic* states.

The role of target determinator and source restriction will be clear when the transformation from UML statecharts to Extended Hierarchical Automata is introduced (Section 2.2). Here we only mention that compound and interlevel transitions of UML statecharts will be represented by simple transitions at the level of uppermost states they exit and enter, and the original sources (resp. targets) of these transitions will be represented in the source restriction (resp. target determinator) of such simple transitions.

In the sequel we shall use the following functions SR, EV, G, AC, TD defined in the obvious way: for transition $t = (s, (sr, ev, g, ac, td), s')$, $SR\ t = sr, EV\ t = ev, G\ t = g, AC\ t =$

$ac, TD\ t = td$. Finally, for transition $t \in \delta_A$ for $A \in F$ let $ORIG\ t$ be defined as follows:

$$ORIG\ t = \{s \mid s \in (SRC\ t) \wedge (SR\ t) = \emptyset\} \cup (SR\ t)$$

The following definition establishes when two transitions are *conflicting*:

Def. 8 *For $t, t' \in (\mathcal{T}\ H)$, t is conflicting with t', written $t\#t'$, iff $t \neq t'$ and $(SRC\ t \preceq SRC\ t') \vee (SRC\ t' \preceq SRC\ t)$*

The following lemma relates orthogonality and conflict:

Lemma 7 *For $t, t' \in (\mathcal{T}\ H)$ the following holds: $(SRC\ t)\ ||\ (SRC\ t')$ implies $\neg(t\#t')$.*

The following definition characterizes those structures which can be used for imposing priorities on transitions.

Def. 9 *[Priority Schema] A* Priority Schema *is a triple $(\Pi, \sqsubseteq, \pi)$ with $(\Pi, \sqsubseteq)$ a* parial order *and $\pi : (\mathcal{T}\ H) \to \Pi$ such that: $\forall t, t' \in (\mathcal{T}\ H).\ (\pi t \sqsubseteq \pi t') \wedge t \neq t' \Rightarrow t\#t'$ We say that t has lower priority than (equal priority as) t' iff $\pi t \sqsubseteq \pi t'$.*

The following lemma relates orthogonality and priority:

Lemma 8 *For $t, t' \in (\mathcal{T}\ H)$ the following holds: $(SRC\ t)\ ||\ (SRC\ t')$ implies $\pi t \not\sqsubseteq \pi t'$.*

The priority system we use in this paper is based on the origin of transitions. Let $PWO = \{X \subseteq (\mathcal{S}\ H) \mid X\ pairwise\ orthogonal\}$ and function f defined as $ft = ORIG\ t$.

Proposition 2 *$(PWO, \preceq^s, f)$ is a priority schema.*

2.2 Translation of UML statecharts to extended hierarchical automata

The translation maps a UML statechart to an extended hierarchical automaton $H = (F, E, \rho)$ by defining the set of sequential automata F, the composition function ρ and the set of events E. For the sake of simplicity and readability, here we give just an informal sketch of the translation.

Set of sequential automata. Each automaton $A \in F, A = (\sigma_A, s^0_A, \lambda_A, \delta_A)$ is defined as follows.

- States. States of the statechart are uniquely mapped to states of sequential automata.
  - Root automaton H. If the (composite) top state s_0 of the statechart is concurrent then it is mapped to the single (initial) state of a degenerate root automaton H. Otherwise the direct substates of the top state are mapped to states σ_H of the root automaton H.
  - Sub-automata in $\mathcal{A}\ H$. Each non-concurrent composite substate s of the statechart defines the states of a unique sequential automaton A_s, as direct substates of s are mapped to states of σ_{A_s}. Note that regions (direct substates of a concurrent composite state) are not mapped to any state in the extended hierarchical automaton.
- Initial state. The initial state s^0_A of an automaton A is the state that corresponds to the state of the statechart marked by an initial pseudostate.

- Transitions. In order to define the mapping of the transitions, we need the following definitions. A transition of the statechart is characterized by its least common ancestor (LCA) state, which is the lowest level *non-concurrent* state that contains all the source states and target states (here the definition of [1] is slightly modified). The *main source* (*main target*) of a transition is the direct substate of its LCA that contains the sources (targets). According to the above rules, main sources and main targets are always transformed to states of the same automaton.

  Each transition τ in the statechart is mapped to a unique transition t of the extended hierarchical automaton as follows. The source $SRC\ t$ (target $TGT\ t$) of t is the state that corresponds to the main source (main target) of τ. This means that a compound or interlevel transition of the statechart is mapped to a transition of the automaton containing the states corresponding to its main source and main target (this automaton is a sub-automaton of the state representing the LCA). The original source and target states will be included in the label of the transition in the form of source restriction and target determinator, as described below.

- Transition labels. The label of a transition t is of the form $(SR\ t, EV\ t, G\ t, AC\ t, TD\ t)$. $SR\ t$ and $TD\ t$ are generated using the source(s) and target(s) of τ, while the $EV\ t$, $G\ t$ and $AC\ t$ of t are inherited from τ:

  - Source restriction. If the set of states that corresponds to the source(s) of τ is the same as $SRC\ t$, then $SR\ t$ must be empty, otherwise it is such a set of source(s).
  - Target determinator. $TD\ t$ is the normalized set of states that corresponds to the target(s) of τ. Normalizing means computing the maximal set of orthogonal basic states that are substates of the states entered by τ explicitly or by default. In this way, $TD\ t$ explicitly contains all the states which have to be entered when the transition is fired, while some of these states are not explicitly pointed to by τ. The following is a sketch of a normalization algorithm which visits the states reached by (segments of) τ, starting from its main target:
    * If a basic state is reached then it is added to $TD\ t$ and recursion stops.
    * If a composite state is reached at its boundary then the algorithm is applied recursively to its initial substate, or to the initial substate of each of its regions.
    * If a non-concurrent composite state is reached and its boundary is crossed then the algorithm is applied recursively to its direct substate where the transition continues (note that branch segments are not considered in this paper).
    * If a concurrent composite state is reached and its boundary is crossed then the algorithm is applied recursively to (i) the direct substate(s) of those regions where the transition continues and (ii) the initial substates of the other regions.
  - Trigger events. In UML statecharts, each transition (including compound transitions) can have at most one trigger event, since join, fork and branch segments can not have a trigger. Accordingly, $EV\ t$ is exactly the trigger event of τ.
  - Guards. Since fork and joint segments have no guards, each transition may have a single guard (note that branch segments are not considered in this paper). Accordingly, $G\ t$ is exactly the guard of τ.
  - Actions. $AC\ t$ is exactly the sequence of actions of τ.

Composition function. ρ is determined by the substate relationships of composite states. If a composite state s is non-concurrent and it is not a region then its direct substates form the states of A_s, a sub-automaton of s, where $\{A_s\} = (\rho\ s)$. If a composite state s is concurrent then every one of its regions forms a sub-automaton of s, in such a way that this automaton contains the direct substates of the region.

Set of events. E is defined as the union of two (not necessarily distinct) sets: the set of events used in the statechart as triggers of the transitions and the set of events generated by actions. In open systems, the set of events generated by the environment is also included.

Figure 2 together with Table 1 is the result of applying the translation to the statechart in Figure 1.

2.3 UML Formal Operational Semantics of Extended Hierarchical Automata

In this section we develop a formal semantics for Extended Hierarchical Automata which is different from that proposed in [4] in that it has to deal with the peculiarities of UML statechart diagrams. The main difference is the need to deal explicitly with priorities since UML priority rules do not directly match the hierarchical structure of Extended Hierarchical Automata, as is the case with classical statecharts. Moreover, the environment is treated differently.

2.3.1 Operational Semantics Rules

We first define *configurations*. A configuration denotes a global state of an extended hierarchical automaton, composed of local states of component sequential automata.

Def. 10 (Configurations) *A* configuration *of H is a set $C \subseteq (\mathcal{S}\ H)$ such that (i) $\exists_1 s \in \sigma_{A_{root}}.\ s \in C$ and (ii) $\forall s, A.\ s \in C \wedge A \in \rho\ s \Rightarrow \exists_1 s' \in A.\ s' \in C$*

For $A \in F$ the set of all configurations of A is denoted by Conf_A. Possible configurations of the extended hierarchical automaton of Fig 2 are: $\{s2\}$, $\{s1, s6, s8\}$, $\{s1, s7, s9\}$ whereas $\{s1\}$ is not (it is not downward closed), as well as $\{s7\}$ (no state from the root) or $\{s1, s2\}$ (two states belonging to the same sequential automaton). The following result easily follows from the definitions:

Proposition 3 *For $A \in F$ and $A' \in \rho_A\ \sigma_A$: $\mathcal{C} \in \mathrm{Conf}_A \wedge \mathcal{C} \cap \sigma_{A'} \neq \emptyset \Rightarrow \mathcal{C} \cap \mathcal{S}\ A' \in \mathrm{Conf}_{A'}$*

The operational semantics of an extended hierarchical automaton will be defined as a Kripke structure, which is a set of states related by a (transition) relation. Usually, the states are called *statuses* and the transition relation is called the *STEP relation*. Each status is composed by a configuration and the current *environment* with which the extended hierarchical automaton is supposed to interact. While in classical statecharts the environment is modeled by a set, in the definition of UML statechart diagrams the particular nature of the environment is not specified (actually it is stated to be a *queue*, but the management policy of such a queue is not defined). We choose *not* to fix any semantics such as a set, or a bag or a FIFO queue etc. for the environment. In the following definition we will then assume that for set X, ΘX denotes the set of all structures of a certain kind (like FIFO queues, or bags, or sets) over X and we shall assume to have basic operations for inserting and removing elements from such structures. In particular $(add\ \mathcal{E}\ e)$ will denote the structure obtained by adding e to environment $\mathcal{E}$. Similarly, $(join\ \mathcal{E}\ \mathcal{E}')$ denotes the environment obtained by merging $\mathcal{E}$ with $\mathcal{E}'$. Moreover, by $(Sel\ \mathcal{E}\ e\ \mathcal{E}')$ we mean that $\mathcal{E}'$ is the environment resulting from selecting e from $\mathcal{E}$, the selection policy depending on the choice for the particular semantics of the environment. Finally, *nil* is the empty structure and given sequence $r \in X^*$, $(new\ r)$ is the structure containing the elements of r (again, the existence and nature of any relation among the elements of $(new\ r)$ depends on the semantics of the particular structure).

So, for instance, if sets are chosen, then $(add\ \mathcal{E}\ e) = \mathcal{E} \cup \{e\}$, $(join\ \mathcal{E}\ \mathcal{E}') = \mathcal{E} \cup \mathcal{E}'$ and, for $e \in \mathcal{E}$, $(Sel\ \mathcal{E}\ e\ \mathcal{E}') \equiv (\mathcal{E}' = \mathcal{E} \setminus \{e\})$. Details like what is the result of attempting to select an event from an empty environment etc. are left unspecified here since they are part of the semantics of the environment and will be specified when such a semantics is fixed.

Def. 11 (Operational semantics of Extended Hierarchical Automata) *The operational semantics of an extended hierarchical automaton H is a Kripke structure $\mathbf{k} = (\mathbf{S}, \mathbf{s^0}, \overset{STEP}{\longrightarrow})$ where (i) $\mathbf{S} = \mathrm{Conf}_H \times (\Theta\ E)$ is the set of statuses of $\mathbf{k}$, (ii) $\mathbf{s^0} = (\mathcal{C}_0, \mathcal{E}_0) \in \mathbf{S}$ is the initial status, and (iii) $\overset{STEP}{\longrightarrow}$ is the transition relation defined in the sequel.*

A transition of **k** is a maximal set of non-conflicting transitions of the sequential automata of H which respect priorities. As in [4], we shall define the $\stackrel{STEP}{\longrightarrow}$ relation by means of a deduction system, and we shall do this both for the case in which the environment can be manipulated from outside the system specified by H (open systems semantics) and for the case in which this is not allowed (closed systems semantics). The rules follow:

Def. 12 (Closed Systems)

$$\frac{(Sel\ \mathcal{E}\ e\ \mathcal{E}'') \quad (1) \qquad H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \stackrel{L}{\longrightarrow} (\mathcal{C}', \mathcal{E}') \quad (2)}{(\mathcal{C}, \mathcal{E}) \stackrel{STEP}{\longrightarrow} (\mathcal{C}', (join\ \mathcal{E}''\ \mathcal{E}'))}$$

Def. 13 (Open Systems)

$$\frac{(Sel\ \mathcal{E}\ e\ \mathcal{E}'') \quad (1) \qquad H \uparrow \emptyset :: (\mathcal{C}, \{e\}) \stackrel{L}{\longrightarrow} (\mathcal{C}', \mathcal{E}') \wedge \mathcal{E}' \subseteq \mathcal{E}''' \quad (2)}{(\mathcal{C}, \mathcal{E}) \stackrel{STEP}{\longrightarrow} (\mathcal{C}', (join\ \mathcal{E}''\ \mathcal{E}'''))}$$

In the above rules we make use of an auxiliary relation, namely $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \stackrel{L}{\longrightarrow} (\mathcal{C}', \mathcal{E}')$. The relation $\stackrel{L}{\longrightarrow}$ models labeled transitions of the extended hierarchical automaton A, and L is the set containing the transitions of the sequential automata of A which are selected to fire. We shall call $\stackrel{L}{\longrightarrow}$ *step* transitions in order to avoid confusion with transitions of sequential automata. P is a set of transitions. It represents a constraint on each of the transitions fired in the step, namely that it must not be the case that there is a transition in P with a higher priority. So, informally, $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \stackrel{L}{\longrightarrow} (\mathcal{C}', \mathcal{E}')$ should be read as "A, on status $(\mathcal{C}, \mathcal{E})$ can perform L moving to status $(\mathcal{C}', \mathcal{E}')$, when required to perform transitions with priorities not smaller than any in P". Obviously, no restriction is made on the priorities for H, but, as we shall see later, set P will be used to record the transitions a certain automaton can do when considering its sub-automata. More specifically, for sequential automaton A, P will cumulate (the priority information of) all transitions which are enabled in the ancestors of A. In the sequel we shall formalize all the above concepts by means of defining a deduction system for relation $\stackrel{L}{\longrightarrow}$. We first need a few auxiliary definitions.

Def. 14 (Enabled Transitions) *For $A \in F$, set of states $\mathcal{C}$ and environment $\mathcal{E}$,*
(i) the set of all the enabled local *transitions of A in $(\mathcal{C}, \mathcal{E})$, $LE_A\ \mathcal{C}\ \mathcal{E}$ is defined as follows*[4]*:*

$$LE_A\ \mathcal{C}\ \mathcal{E} = \{t \in \delta_A \mid \{(SRC\ t)\} \cup (SR\ t) \subseteq \mathcal{C} \wedge (EV\ t)\ \in\ \mathcal{E} \wedge (\mathcal{C}, \mathcal{E}) \models (G\ t)\}$$

(ii) the set of all enabled *transitions of A in $(\mathcal{C}, \mathcal{E})$ considered as an extended hierarchical automaton, i.e. including those of descendents of A, $E_A\ \mathcal{C}\ \mathcal{E}$ is defined as follows:*

$$E_A\ \mathcal{C}\ \mathcal{E} = \bigcup_{A' \in (\mathcal{A}\ A)} LE_{A'}\ \mathcal{C}\ \mathcal{E}$$

Moreover, $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \stackrel{L}{\longrightarrow}$ will stand for: there exists $\mathcal{C}'$ and $\mathcal{E}'$ such that $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \stackrel{L}{\longrightarrow} (\mathcal{C}', \mathcal{E}')$. Finally, for state s and set $S \subseteq \mathcal{S}\ (\rho\ s)$, such that $s \preceq s''$ for all $s'' \in S$, the *closure* of S, $(\mathbf{c}\ s\ S)$, is defined as the set $\{s' \mid \exists s'' \in S.\ s \preceq s' \preceq s''\}$.

[4] $(\mathcal{C}, \mathcal{E}) \models g$ means that guard g is true of status $(\mathcal{C}, \mathcal{E})$. Its formalization is immaterial for the purposes of the present paper. In the deduction rules, we will relax the requirement $\mathcal{C} \in \mathrm{Conf}_A$ and we will assume $\mathcal{C} \in \mathrm{Conf}_H$. This allows the use of guards which make reference to non local states.

Def. 15 (Progress rule) *If there is a transition of A enabled and the priority of such a transition is "high enough" then the transition fires and a new status is reached accordingly:*

$$\frac{\begin{array}{ll} t \in LE_A\ \mathcal{C}\ \mathcal{E} & (1) \\ \nexists t' \in P \cup E_A\ \mathcal{C}\ \mathcal{E}.\ \pi t \sqsubset \pi t' & (2) \end{array}}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\{t\}} (\mathbf{c}\ (TGT\ t)\ (TD\ t), new(ACt))}$$

The rule essentially says that a (local) transition t of sequential automaton A can fire if it is enabled in the current configuration (1) and there is no higher priority transition in P (so t is "high enough" for P, or "respects" P), or in the set of all the currently enabled transitions of A or of any descendent of A.

Once transition t is taken, a new configuration is entered and proper actions are performed. For instance, in our example, when $\{s3\}$ is the current configuration and $a2$ is offered by the environment, the above rule can be used for firing transition $t5$, which will result in generating event $e1$ and entering configuration $\{s1, s6, s9\}$

Def. 16 (Composition Rule) *This rule establishes how automaton A delegates the execution of transitions to its sub-automata and these transitions are propagated upwards.*

$$\frac{\begin{array}{ll} \{s\} = \mathcal{C} \cap \sigma_A & (1) \\ \rho_A\ s = \{A_1, \ldots, A_n\} \neq \emptyset & (2) \\ \bigwedge_{j=1}^{n} A_j \uparrow P \cup LE_A\ \mathcal{C}\ \mathcal{E} :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L_j} (\mathcal{C}_j, \mathcal{E}_j) & (3) \\ \left(\bigcup_{j=1}^{n} L_j = \emptyset\right) \Rightarrow (\forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t') & (4) \end{array}}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\bigcup_{j=1}^{n} L_j} (\{s\} \cup \bigcup_{j=1}^{n} \mathcal{C}_j, join_{j=1}^{n} \mathcal{E}_j)}$$

First of all notice that the sub-automata are required to perform their step-transitions under the new set $P \cup LE_A\ \mathcal{C}\ \mathcal{E}$ which includes all the enabled local transitions of A (3) so that, in order to be selected, the transitions of such sub-automata must have a priority which is not lower than any of those of the enabled local transitions of A (and A's ancestors, recursively upwards ...). Notice also that if no transition of the sub-automata can be fired then the rule is applied *only* if also no local transition of A can fire (4), thus propagating the empty set of transitions upwards (see below). The new configuration will still include the current state of A but the possible new states of the sub-automata and related actions are recorded in the new status.

Def. 17 (Stuttering Rule) *If there is no transition of A enabled and with priority "high enough" and moreover no sub-automata exist to which the execution of transitions can be delegated, then A has to "stutter":*

$$\frac{\begin{array}{ll} \{s\} = \mathcal{C} \cap \sigma_A & (1) \\ \rho_A\ s = \emptyset & (2) \\ \forall t \in LE_A\ \mathcal{C}\ \mathcal{E}.\ \exists t' \in P.\ \pi t \sqsubset \pi t' & (3) \end{array}}{A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{\emptyset} (\{s\}, nil)}$$

In our example, from status $(\{s1, s6, s8\}, new\ e1)$ automaton $A2$ can only stutter. Moreover, in the above status, automaton $A1$ can fire transition $t6$ and, via the progress rule it can generate a $\xrightarrow{\{t6\}}$ step-transition. Notice also that although transition $t3$ of $A0$ is enabled the progress rule cannot be applied just because of the above step-transition of $A1$ ($\pi t3 \sqsubset \pi t6$). On the other hand, the composition rule can be applied to $A0$ which will propagate the step of $A1$ and the stuttering of $A2$ at the level of a step transition of $A0$.

Notice that in general the progress rule and the composition rule have not mutually exclusive conditions, so that when both rules are applicable non-determinism arises and results in separate step-transitions from the same status. Another source of non-determinism is of course the presence of different enabled local transitions in the same sequential automaton which are selected by different applications of the progress rule. Finally notice that condition (4) of the composition rule prevents the propagation of stuttering above A when there are transitions of A which can fire.

2.3.2 Properties of the Operational Semantics

In the sequel we present a few results which show that the operational semantics we propose meet the informal requirements stated in the definition of UML [1].

We let $A \in F, \mathcal{C} \in \mathrm{Conf}_H, \mathcal{E} \in (\Theta E), P \in 2^{(\mathcal{T}\ H)}$ be respectively a generic automaton, a configuration, an environment and a set of transitions.
The following proposition guarantees that after firing a transition again a status is reached.

Proposition 4 *For all $L \in 2^{(\mathcal{T}\ H)}, \mathcal{C}', \mathcal{E}'$ the following holds:*
$A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} (\mathcal{C}', \mathcal{E}') \Rightarrow ((\mathcal{C}' \in \mathrm{Conf}_A) \wedge (\mathcal{E}' \in (\Theta E)))$.

The next lemma expresses a safety property w.r.t. P: it essentially states that only transitions with a "high enough" priority are fired.

Lemma 9 *For all $L \in 2^{(\mathcal{T}\ H)}, t \in L$ the following holds:*
$A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L} \Rightarrow \nexists t' \in P.\ \pi t \sqsubset \pi t'$

The following result shows that our operational semantics satisfies the requirements informally defined in [1].

Theorem 1 *For all $L \subseteq (\mathcal{T}\ A)$, $A \uparrow P :: (\mathcal{C}, \mathcal{E}) \xrightarrow{L}$ if and only if L is a maximal set, under set inclusion, which satisfies all the following properties: (i) L is conflict-free, i.e. $\forall t, t' \in L.\ \neg t \# t'$); (ii) all transitions in L are enabled in the current status, i.e. $L \subseteq E_A\ \mathcal{C}\ \mathcal{E}$; (iii) there is no transition outside L which is enabled in the current status and which has higher priority than a transition in L, i.e. $\forall t \in L.\ \nexists t' \in E_A\ \mathcal{C}\ \mathcal{E}.\ \pi t \sqsubset \pi t'$; and (iv) all transitions in L respect P, i.e. $\forall t \in L.\ \nexists t' \in P.\ \pi t \sqsubset \pi t'$*

References

[1] Rational Software * Microsoft * Hewlett-Packard * Oracle * Sterling Software * MCI Systemhouse * Unisys * ICON Computing * IntelliCorp * i Logix * IBM * ObjecTime * Platinum Technology * Ptech * Taskon * Reich Technologies * Softeam. *UML Semantics, version 1.1*, 1997.

[2] D. Latella, I. Majzik, M. Massink. *Towards a Formal Operational Semantics of UML Statechart Diagrams.* Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems, Florence, Italy, Feb. 15-18, 1999. Kluwer Publications, (accepted for publication)

[3] D. Latella, M. Massink, and I. Majzik. A Simplified Formal Semantics for a Subset of UML Statechart Diagrams. Technical Report HIDE/T1.2/PDCC/5/v1, ESPRIT Project n. 27439 - High-Level Integrated Design Environemnt for Dependability HIDE, 1998. Available in the HIDE Project Public Repository (https://asterix.mit.bme.hu:998/).

[4] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *Third Asian Computing Science Conference. Advances in Computing Sience - ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, 1997.

From Structural UML diagrams to timed Petri nets

Andrea Bondavalli CNUCE/CNR and PDCC
Majzik Istvan TUB
Ivan Mura University of Pisa and PDCC

1 Introduction

In this chapter we describe the transformation from structural UML specifications to Petri net models for the quantitative evaluation of dependability attributes. We first discuss the motivations that led to the idea of introducing such a transformation in HIDE and its rationale. Then, we detail the limitations to be imposed on the UML designer to allow translating the specification into a dependability model. This restrictions are mainly related to the introduction of redundancy into the system under design, for which particular structures are to be utilised to permit the identification of the crucial points of the dependability analysis. Since the information on dependability aspects are typically not included into a system design, we prescribe a set of extensions of the UML standard language in order to create towards the designer a controlled interface for the input of parameters, the selection of the desired measures, and the choice of the fault-tolerance structures to be included in the system.

We proceed with the definition of the syntax of 2 intermediate representations of the system, used to divide the entire transformation in sequential phases. Last the two main steps of the transformation are described. The first step takes the UML model and produces an Intermediate model in which the dependability related features are filtered from the entire specification. The second, starting from the “dependability” oriented description provided by the Intermediate model produces a timed Petri net, which is described using still an abstract representation. A final step can then be easily performed to translate the model according to the syntax adopted by specific PN tools selected for performing the analysis. This approach of performing the transformation in several stages looks attractive for several reasons which will be discussed at the end of the Section 2.

This document gives a precise and detailed description of the transformation procedure, starting from the UML structural diagrams, and proceeding towards Petri net models. It is important to point out that not only the UML diagrams that form the input of our transformation do not have a formal semantics, but also the specification this set provides might be incomplete or ambiguous. Therefore, the aim of this report is not to provide a “formalization” of the transformation in the sense of formal correctness, but only to precisely describe the steps and the models involved in the transformation itself.

2 Purpose and Rationale

2.1 Purpose

Dependability modelling and analysis can be useful for system understanding and assessment in all phases of the system life cycle as summarised in [16].
During design phases, those of interest in the HIDE context, models allow to compare different architectural and design solutions and to select the most suitable one. The sensitivity analysis that can be carried out after modelling allows 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.
The following attributes of dependability, as defined in [12], might be of interest for a UML designer: availability is the measure of the delivery of correct service with respect to the alternation of correct and incorrect service, reliability is a measure of the continuous delivery of the correct service, safety is the non-occurrence of catastrophic consequences, security is the non-occurrence of unauthorised access. Often additional dependability-related attributes are also defined (e.g. in [14]). Performability attributes originate from a combination of performance and dependability models by taking into account performance in degraded system states. Integrity is defined as the avoidance of improper alterations of system service (information provided by the system). Confidentiality means the non-occurrence of unauthorised disclosure of system service. Maintainability is the ability to undergo repairs and evolution. Testability is the ability to test for certain attributes within the system.
Complex systems consisting of a large number of components including interactions of redundant hardware and software components as well, introduce some problems in modelling and analysis. These problems arise independently of the design methodology applied, thus are present also in systems designed using UML, and must be addressed from any approach to model such systems. Among these problems the most important to solve is complexity (sate explosion). To master complexity a modelling methodology is needed so that only the relevant aspects are detailed, still enabling numerical results to be computable. Simplifying hypotheses are often necessary to keep the model manageable. Since the assumptions may lead to (inaccurate) approximations of the system behaviour, the resulting errors 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. A feasible approach is to start with simple models and make them more and more complex and detailed by releasing those assumptions having unacceptable impact on the results. An other problem is that models need many parameters whose meaning is not always intuitive for the designers. Moreover, it may be very difficult to assign values to the parameters (usually by way of experimental tests).
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: the capacity of available tools is by far exceeded by the state space explosion associated to system-wide models of such detailed view. Moreover, the complete set of statecharts for the system might not be available till the design has reached an advanced development stage, whereas some still partial and not yet very precise analysis may provide useful hints much before.

These are the main motivations for approaching the modelling from a structural perspective.

The automated transformation from UML structural diagrams to timed Petri nets serves in the HIDE framework:

- to provide a means to analyse dependability attributes of the a system while it is still being designed. This way, a designer can easily verify whether the system that is being built satisfies predefined requirements on dependability attributes, without dealing with the background mathematical aspects of Petri net modelling and solution. The results of the dependability model evaluation are automatically back-annotated into the UML diagrams. This choice allows the transformation to provide preliminary evaluations of the system dependability during the early phases of the design.
- to allow a less detailed but system-wide representation of the dependability characteristics of the analysed systems. This models offer a significant advantage in terms of controlling the size of the models.
- to deal with various level of details, ranging from very preliminary abstract UML descriptions, up to the refined specifications of the last design phases. On one side the UML higher level models, that is the structural diagrams, are available before the detailed, low levels ones and the analysis on models derived from the structural view provides indications about the critical parts of the system which require a more detailed representation. On the other side, 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).

2.2 Rationale

During this first phase of the project, we defined the transformation for a reduced set of the dependability attributes defined above. In particular, we restricted our attention to Reliability and Availability. With this approach other attributes (and the above ones at a more refined precision) can be analysed depending on the amount of relevant information provided by the designer.

To analyse the dependability figures of systems of large size one could ideally build a model of the system accounting for all the details, the fine grained behaviour of each system component

that can be obtained by the behavioural UML models. Due to the limitations of existing tools, this approach is not viable (state explosion). Therefore the model to build must be of a reduced size where only the features relevant to dependability are captured and all other information is left aside. On the other hand, it is very difficult to define a priori which are the relevant dependability related characteristics to be captured and represented in the model. Our approach aims at building first a quite abstract model, maybe too coarse for representing with due precision the real dependability to be expected. This model concentrates on the structure of the system and takes information from the structural UML diagrams. However, the modular construction of the model does not prevent, rather favours its extension by offering the possibility to substitute in the model 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. Moreover, for some parts, such as the redundancy management of redundant resources, we take into account from the right beginning the behavioural description by analysing statecharts of selected components and use them to derive dependability-related static relations among objects associated with them, later transformed to subnets of our model. Thus, the idea is to start with a broad system-wide model and to refine it by plugging detailed description of those parts which result to be the critical ones (this selection might be guided by the analyses performed on the coarse model itself).

The dependability model of a system (composed of elements) consists of the following general parts: the *fault activation processes* which model the fault occurrence in system elements and results in *basic events*, the *propagation processes* which model the consequences of basic events and results in *derived failure events* and the *repair processes* which model how basic or derived events are removed from the system.

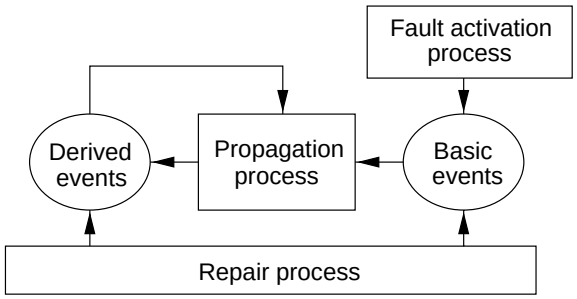


Figure 2.1: General parts of a dependability model

This overall structure of the dependability model is shown in Figure 2.1. The failure of a system is one of the derived events in this model. Note that repair means here a general service restoration (automatic service restoration if underlying faults disappear; explicit diagnosis, repairing or replacing of hardware; restoring the state and re-integration of software etc.).

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

The repair processes are determined by the (physical or) computational policy implemented in the system.

We tried to keep at the minimum the set of assumptions made for this broad model (notice that specific assumptions can be made with reference to a specific system when more information is made available). The following general assumptions (which do not form a complete set) have been made:

- 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 which 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 the failure disappears as soon as the underlying faults and failures have been repaired. Stateless SW elements and HW elements are repaired in this way.
- Explicit repair refers to the actions that are planned and scheduled by the designer. Explicit repair may remove (permanent) faults from the system or restore the service of SW or HW elements.

This transformation is defined in more steps, where the first is 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 (i) the set of basic events, (ii) the propagation processes, (iii) the set of derived events, (iv) the target points of the fault activation processes and finally (v) the target points of the repair processes. In a sense, the dependability model is built in this step.

The next step allows to define a *timed Petri net* general enough to postpone the choice of the automatic tool to use for the analysis to a later stage. Once the tool is selected the construction of the model that can be directly processed by the tool involves a simple syntactic manipulation.

The dependability model will be built in a modular and incremental way. Several studies are known in the literature which propose modular modelling approaches based on Petri net models. For instance, the work in [10] addresses the dependability analysis of the new architecture of the French air traffic control system by exploiting the composability of Petri net submodels connected over a set of well-specified interface points. Another method based on a modular and hierarchical modelling approach which combines different layers of Stochastic Activity Network model [18] has been presented in [16], for the study of the Italian ANSALDO railway interlocking systems.

One fundamental choice has been made in defining the transformation regarding 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).

It is also important to notice that this choice favours the construction of a fault-tolerance library, another component of the HIDE environment. Building a fault-tolerance library means that many important critical elements or schemes are available from the library. The construction of such library can be integrated with the dependability modelling 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. For systems adopting only redundant structures taken from the library, no behavioural information is then required, dependability models are obtained considering the structure level view only.

3 UML: model elements used, additions and constraints

Here we first list the set of UML constructs that are considered in the derivation of our transformation. Then we describe the restrictions to be imposed on the UML designer. These restrictions are mainly related to the introduction of redundancy into the system under design, for which particular structures are to be utilised to permit the identification of redundancy (fault tolerance), i.e. the crucial points of the dependability analysis.
Last we prescribe a set of extensions of the UML standard language for dependability parameters to be provided.

3.1 UML model elements used

As already stated, this transformation is deriving a timed Petri net from UML using mainly structural diagrams. Nevertheless, as anticipated, it may (or must, as the case might be) use also behavioural diagrams for some specific component or for a more detailed modelling of critical situations. We now summarise the role UML diagrams and elements considered in our model derivation (for a more detailed analysis, refer to Section 5.1).

- **Use case diagrams:** The role of use case diagrams is to identify system level relations at the top level of the hierarchy. Actor(s) identify the use cases which represent the top level service of the system, this way also defining the system level failure. The design should contain a use case diagram with (at least one) use case and actor.
- **Class diagrams:** Class diagrams are used to identify relations, which are traced to objects (instanciated from the given classes, represented in other diagrams). By default, each class is instanciated by a single object. Multiple objects of the same class should be identified on collaboration, sequence or deployment diagrams.
- **Object diagrams:** Object diagrams are used to identify objects (as basic software elements) and the relations among them.
- **Collaboration diagrams**: Collaboration diagrams can be used to identify objects and their relations. (Note that in some of the UML-based tools, objects can be included only on collaboration, sequence or deployment diagrams.)

- **Sequence diagrams:** Sequence diagrams can be used to identify objects and their relations. Messages may identify the direction of relation.
- **Component diagrams**: Component diagrams are used to identify the relations among components, and in this way among objects realised by the components. Note that the components (and their objects) are instanciated on the deployment diagrams.
- **Deployment diagrams:** Deployment diagrams are used to identify nodes (as basic hardware elements) and relations among software and hardware elements. Relations among nodes (e.g. communication) are also described here.
- **Statechart diagrams:** Statechart diagrams are used basically only in the case of redundancy structures, to derive the non-trivial relations among participants of the structure.

3.2 Representation of redundancy (fault tolerance) structures

We adopt the class-based approach of redundancy (fault tolerance) structures.
In general, an element (here SW or HW) is redundant if its service can be delivered by an other element in a coordinated and automatic way, without the interaction of the client(s). Accordingly, operation of redundant elements 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 elements (objects) called here *variants*, which are coordinated by the redundancy manager: the service is available through the redundancy manager and the redundant elements can not be used separately. An element is participant in a single redundancy structure only. Other, non-redundant elements can not be included.
Accordingly, redundancy structures must be composed of objects instanciated from the following types of classes:

- redundancy manager;
- variant;
- adjudicator, which can be further refined by various subtypes e.g. tester, voter or comparator.

This constraint on the way redundancy is expressed allows first to identify redundancy in the design and also gives the opportunity to identify the specific relations among the components (which can be quite complex). These relations can be conveniently represented by a fault-tree [4], which can be generated automatically provided that some conventions are applied (Section 3.3.3), as it will be shown in Section 5.4.

3.3 Review of the extensions of UML

Since an UML specification does not cover all non-functional aspects required for dependability modelling (like failure characteristics of model elements), we have to ask the designer to "extend" the specification in order to be able to construct the dependability model, i.e. define the basic and derived events, propagation, failure and repair processes. UML provides the fa-

cilities to introduce such extensions into the model. The following extensions can be applied directly to any model element:

Tagged values. Tagged values are pseudo-attributes assigned in the form of a tag (name of a property) and a value.

Constraints. Constraints are Boolean expressions given mainly in the Object Constraint Language OCL [17]. Note that constraints can be applied also to the system structure, since the constraint language provides mechanisms to describe the structure of model elements.

Stereotypes. Stereotypes introduce a new class of modelling elements introduced at modelling time. A high-level classification (meaning/usage) of elements can be described. Usually, a stereotype qualifies the base class with additional constraints (that must be satisfied) and tagged values (that must be present). Stereotypes are generalisable, i.e. subtypes and hierarchy can be defined.

Comments. Comments are arbitrary, unstructured annotations.

Extensions are necessary for the following purposes:

- Identifying redundancy (fault tolerance) structures.
- Assignment of dependability related parameters to elements of the UML to be projected into elements of the IM.
- Identifying states and events in statecharts of redundancy managers.

The role and form of these extensions will be reviewed in the following sections.

3.3.1 Identifying redundancy structures

As described earlier, the class-based approach is adopted in order to include redundancy structures in the design. The three basic components of a redundancy structure are the redundancy manager, the variants and the adjudicator. The classes (or directly the objects) of the structure are 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, testers etc.).

3.3.2 Assignment of parameters

The model parameters can be included in UML models as standard extensions in the form of tagged values. The use of tagged values can be prescribed by stereotypes assigned to model elements (e.g. classes of critical objects). Since tagged values can not be applied to a group of model elements, the common parameters like common mode failure rates have to be distinguished by using other mechanisms.

Consider the case that, at a given stage of the design process (more or less advanced), an element is a basic one in the sense that no further elements in hierarchy levels under this are represented (not yet or there are no elements at all). If at this point the designer wants to perform an analysis of the design he/she should set the parameters of that element (as required for the dependability model). Whenever an element is further decomposed then its parameters can be derived during the analysis, using the parameters assigned to the underlying basic elements.
Software elements and hardware elements have different sets of parameters.

Hardware elements

Stateless and stateful hardware elements are distinguished by stereotypes. These stereotypes are constrained to have a set of tagged values storing the actual set of parameters. The designer can assign a tagged value with one, two, or no values. In the first case, the value is intended to be used to instanciate the parameter, in the second the two values specify a range for a sensitivity analysis, and in the third, when no value is assigned by the designer, then the parameter should be derived.

- Stereotype <<stateless>> indicates a stateless element. The necessary tagged values are the following:
- tagged value “FO = x.y“ -- fault occurrence
- tagged value “PP = x.y“ -- percentage of permanent faults
- tagged value “RD = x.y“ -- repair delay
- Stereotype <<stateful>> indicates a stateful element. The necessary tagged values are the following:
- tagged value “FO = x.y” -- fault occurrence
- tagged value “EL = x.y” -- error latency
- tagged value “PP = x” -- percentage of permanent faults
- tagged value “RD = x.y” -- repair delay

These stereotypes can be applied to nodes (in deployment diagrams) of UML.

Software elements

Similarly, stateless and stateful software elements are distinguished by stereotypes. These stereotypes are constrained to have a set of tagged values storing the actual set of parameters. If a tagged value does not have a value assigned by the designer then it means that this parameter should be derived.

- Stereotype <<stateless>> indicates a stateless element. The necessary tagged value is the following:
  - tagged value “FO = x.y” -- fault occurrence
- Stereotype <<stateful>> indicates a stateful element. The necessary tagged values are the following:
  - tagged value “FO = x.y” -- fault occurrence

- tagged value “EL = x.y” -- error latency
- tagged value “RD = x.y” -- repair delay

These stereotypes can be applied to the following model elements of UML:

- Use cases. If a use case is assigned numerical dependability parameters then it means that its refinement is not relevant for the dependability model.
- Classes. In this case, all objects instanciated from the class should be assigned the same set of parameters.
- Packages. If a package is assigned numerical dependability parameters then it means that its refinement is not relevant for the dependability model.
- Objects.
- Components. If a component is assigned numerical dependability parameters then it has to be considered in the dependability model as a software element.

Relations

Relations indicating error propagation paths are assigned propagation-related parameters. A stereotype is used for this purpose:

- Stereotype <<propagation>> indicates an error propagation path, with the following parameter:
  - τagged value “PP = x.y” -- propagation probability

This stereotype can be applied to the following model elements of UML (for a detailed analysis, refer to Section 5.1):

- Generalisation relationship <<extends>> and <<uses>> between use cases.
- Association between classes. In general, an association denotes a bidirectional error propagation path, this way both association ends may have stereotypes. In case of aggregation or composition, only one association end (with the special adornment) can have stereotype.
- Dependency <<uses>> between classes.
- Dependency <<uses>> between packages.
- Links between objects.
- Set of messages in sequence diagrams. Since the parameters characterise not the messages but the communication path, it is enough to assign the parameters to one of the messages along a given path.
- Actions of statecharts (in the same way as messages).
- Dependency <<calls>> between components.
- Deployment relations (graphical nesting or composition associations) between nodes and components/objects.
- Association between nodes.

A well-formed model requires that a type form (class, association etc.) and its instance form (object, link etc.) should not have different dependability parameters. Such contradictions can be resolved by considering the parameters assigned to instances (more refined elements) valid, according to the object-oriented approach.

3.3.3 Conventions in statecharts

In order to derive the non-trivial relations in redundancy structures (i.e. the fault tree assigned to the redundancy structure), the statechart of the redundancy manager has to be analysed. This analysis is supported by stereotyping the states and events in the statechart as follows:

- Stereotype <<failure>> of a state indicates that it is an explicit failure state.
- Stereotype <<failure>> of an event indicates that it is an explicit failure notification towards the client(s).
- Stereotype <<response>> of an event indicates that it is a normal response of the object towards the client(s).

Moreover, a more detailed distinction of adjudicators (indicated by the stereotype <<adjudicator>> introduced above) has to be defined:

- Stereotype <<tester>> indicates a tester object called by the redundancy manager to perform (acceptance) tests on the results of the variant(s).
- Stereotype <<comparator>> indicates a comparator object which is called by the redundancy manager to compare the results of variants.

In redundancy structures, the behaviour in the presence of faults is determined not only by the individual failure/repair parameters of the elements but also by their common mode failure characteristics. Similarly, detection coverage of adjudicators is an important parameter of the structure. Accordingly, the following tagged values are used:

- tagged value “CF = x.y” -- common mode failure occurrence
- tagged value “DC = x.y” -- detection coverage

Since tagged values can be assigned to single elements only, the above tagged values has to be assigned to a *comment* associated with the elements for which they are defined. Note that this information is used only in redundancy structures, when the fault tree corresponding to the failure of the structure is generated.

4 The intermediate representations

4.1 Intermediate model

The intermediate model is a general model of a system composed of multiple elements. The structure of the intermediate model is inspired by the approach presented in [13]. For our purposes, we slightly modify that model. We use a more reduced hierarchy and, for the sake of

convenience, we distinguish between stateless (purely functional) and stateful (having internal state) elements.

Practically speaking, the intermediate model is an hypergraph G=(N,A), where each node in N represents an entity described in the set of UML structural diagrams, and each hyperarc represents a relation between elements, that is a bit of the structure itself, as it has been projected from the UML diagrams. Both the nodes and the hyperarcs are labelled, that is they have attached a set of attributes completing their description. These attributes are obtained from the UML diagrams. We now give the semantic of the intermediate model G, by describing the sets N and A and what they represent.

The generic elements of set N are described by the following list:

```
NODE <name> <type of node> <list of attributes>
```

There are six distinct types of nodes, each with a particular set of attached attributes:

Stateless hardware elements (type SLE-HW). They represent purely functional hardware elements. The attributes for the SLE-HW type of nodes are the following ones:

```
<fault_occurrence>
<permanent/transient>
<repair_delay>
```

The `fault_occurrence` field identifies a random variable, which represents the time needed for a fault to hit the hardware component the element represents. The field permanent/transient specifies the relative percentages of the two type of faults. The `repair_delay` attribute specifies a random variable representing the time needed to perform the repair of the hardware element in the case it has been hit by a permanent fault. This time to repair includes the time for fault treatment. Whenever any of these parameter is not specified, the a more detailed submodel is to be included in final dependability model. Notice that for the time being we have considered a single failure process for both the transient or permanent faults affecting a hardware element. This will be probably refined in the next phase of the project. Anyway, it is worthwhile observing that if an accurate knowledge of the fault occurrence processes is available, then the fault-occurrence field and the permanent/transient field as well may be left unspecified, and a detailed fault submodel can be included in the final dependability model, as already specified.

Stateful hardware elements (type SFE-HW). They represent hardware components of the system, which do have internal state. The attributes for the SFE-HW type of nodes are the following ones:

```
<fault_occurrence>
<error_latency>
<permanent/transient>
<repair_delay>
```

The field `fault_occurrence` is defined as for the SLE-HW elements. Because of the presence of an internal state, 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 fields `error_latency` then plays the same role as `fault_occurrence`, but it 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 hardware element in the case it has been hit by a permanent fault. This time to repair includes for SFE-HW elements the time for fault-treatment plus the time necessary to perform the error recovery. Indeed, the internal state of the element may have been corrupted by the effects of the fault activation, and needs to be recovered.

Stateless software elements (type SLE-SW). They represent purely functional software elements. The attributes for the SLE-HW type of nodes are the following ones:

```
<fault_occurrence>
```

The `fault_occurrence` field represents the time needed for a fault to hit the software element. Let us remind that faults affecting a software component are only of transient nature, therefore there is no need to perform fault-treatment actions. Moreover, since the component is stateless, there is no need to perform error recovery, neither. The repair of a SLE-SW elements is thus implicit.

Stateful software elements (type SFE-SW). They represent software components of the system, which do have internal state (variables). The attributes for the SFE-SW type of nodes are the following ones:

```
<fault_occurrence>
<error_latency>
<repair_delay>
```

The fields listed above are defined as for the SFE-HW elements. However, the meaning of the `repair_delay` attribute is slightly different. Indeed, faults may lead to a corruption of the internal state of the stateful software component, thus making necessary a state restoration. However, since faults affecting the software are transient, for a SFE-SW type of element it accounts for the time needed to perform the error recovery solely, without any fault-treatment.

Fault-tolerance structures (type FTS). FTSs are composite elements consisting of SLEs or SFEs. FTSs are not physical entities, they only represent the logical grouping of SLEs or SFEs that implement a redundancy structure. An FTS has the following attached attribute:

```
<fault-tree>
```

which 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 description of the fault-tree is obtained:

1. From the fault-tolerance library, if the UML designer has selected the fault-tolerance scheme from the list of those predefined made available in the library. In this case the fault-tree has been defined and associated with the name of the fault-tolerance scheme.
2. With the analysis procedure specified in Section 5.4, if the designer has defined a new fault-tolerance scheme not already present in the library.

System (type SYS). SYS nodes are introduced in the intermediate model to represent the components of the system whose dependability attributes are the object of the evaluation and to represent compound elements as well. In particular, a SYS node may represent the system itself, that is the entity that provides the whole set of functionalities (use cases) to the final users (the actors). Also, it may represent any UML entity about which the designer is interested in estimating the dependability figures for a particular design. In this case the attribute list for the SYS element is:

```
<measure_of_interest>
```

which specifies the particular dependability attribute of interest for the analysis, that is one among pointwise reliability, steady-state reliability (MTBF), pointwise availability, steady-state availability.

A SYS node does not necessarily corresponds to a particular entity appearing in some UML diagrams. For instance, SYS nodes may also be used to represent sets of system elements, usually below a high-level compound one (like a use case), which have interactions with other system elements as a whole. In this case, the attribute list of the SYS node is empty.

The nodes of the intermediate model G are linked by the hyperarcs in the set A. The generic element of set A is described by the following list:

```
HYPERARC <type of hyperarc>
     <from_node> <to_node_1, to_node_2,...,to_node_n>
     <list of attributes>
```

where the field `from_node` gives the name of the originating node, and the list `to_node_1, ...,to_node_n` gives the names of the destination nodes of the hyperarc. There are the following two distinct type of hyperarcs, describing two different types of relations among nodes of the intermediate model:

Uses the service of (type U). The type U hyperarc is a simple arc connecting element `node_1` and `node_2` of the intermediate model. It represents a client-server relation between `node_1` and `node_2`, a relation that is unidirectional (not reflexive). In the intermediate model, SW elements (either SFE or SLE) use the services of other SWs, HWs or FTSs. HWs use the services of another HWs or FTSs made up of HWs. Actors (i.e. human

users or external systems which interact directly with the system under investigation) use the system, that is (the collection of) use cases. Elements involved in such relation are coupled in terms of failure propagation: whenever the server `node_2` fails, there is a possibility (non-zero probability) that the client `node_1` fails (or reaches an erroneous state) as a result. A failure of the client `node_1` might as well result in a error/failure of the server `node_2`, for instance if an illegal request is generated by `node_1`, and `node_2` is not able to trap it, but we did not consider such level of detail during this first phase.

Also, the U relation prescribes a constraint for the repair of a node. Indeed, the repair actions needed to recover a node from a failure can be logically split into two parts. The first part is related to the fault treatment/error recovery that can be performed locally at the node, and the second is related to the nodes the node is using, that is external environment of the node, which must be correctly behaving as well before completing the recovery. Therefore, the repair of a node can not be completed until all the used nodes are fully operational themselves.

The list of attributes field for the U type of hyperarc is as follows:

`<prop_prob>`

The field `prop_prob` gives a measure of the probability that a failure of the server `node_2` will result in an error/failure on the client side, that is in `node_1`.

Is composed of (type C). The type C hyperarc links an FTS node to the set SWEs or HWEs it is composed of. The C relation is used to denote the non-trivial dependencies between the FTS and its composing elements, dependencies that are described in the `fault_tree` field of the FTS node. This relation is also used to link the nodes marked SYS with the set of intermediate model nodes they consist of. No attributes are foreseen for the type C hyperarcs.

In the following, in order to avoid confusion with UML nodes (representing hardware entities), nodes of the intermediate model will often be referred to as “elements” of the IM.

4.2 Timed Petri Nets

The class of timed Petri nets we consider hereafter is not given a particular name in the literature. We shall keep referring to them as timed Petri nets, to emphasise that they allow modelling activities whose duration is a random time. However, we are not interested in a rigorous specification of the class of distribution from which these random times can be drawn, because in this first phase of the project we do not want to restrict the set of automated tools that could be utilised for the solution of the models.

We give in the following a short description of timed Petri nets. A timed Petri net model is formally a five-tuple (P, T, I, O, S), where:

- $P=\{P_1,P_2,...,P_n\}$ is the set places, each graphically represented as a small circle. Each place is described as follows:

```
PLACE <name> <tokens> <bound>
```

The field `name` identifies the place in the net. The place may contain a possibly bounded (the threshold is given by field `bound`) non-negative integer number of tokens, which represent some entities of the system. The number of tokens in a place P_i is called the marking of the place, and is given by the field `tokens`. The n-sized vector that collects the marking of all the places of the net is called the marking of the net.

- $T=\{t_1,t_2,...,t_m\}$ is the set of transitions, each of them graphically represented as a small bar. A transition models the delay necessary to complete a certain activity of the system, and is described as follows:

```
TRANSITION <name> <random_variable> <memory_policy> <guard> <priority>
```

Each transition has a name, an associated random variable, and a memory policy. The random variable is a set of fields specifying the distribution of the delay necessary to perform the associate activities, and the memory policy specifies a rule for the sampling of the successive random delays from the distribution. Both the random variable distribution and the memory policy may be dependent from the marking of the net. Also, any transition may optionally have associated a guard, that is a Boolean function of the net marking. Immediate transitions may optionally have a priority, which is used to solve the competition among immediate transitions. The competition among immediate transitions can be also probabilistically solved by associating a probability to the transition through the field `random_variable`.

- I is the set of input arcs, contained in PxT, each described as follows:

```
INPUT_ARC <from_place> <to_transition> <weight>
```

The two fields `from_place` and `to_transition` specify the name of the originating place and of the transition destination, respectively. An integer non-zero weight is associated to the arc. An input arc having a negative weight is called an inhibitor arc, otherwise it is called an ordinary input arc. The places linked by an ordinary input arc to transition t_i are called input places for that transition, and the places linked to transition t_i by an inhibitor arc are called inhibitor places for t_i. The weight on an input arc may be dependent from the marking of the net.

- O is the set of output arcs contained in TxP. An output arc is as follows:

```
OUTPUT_ARC <from_transition> <to_place> <weight>
```

All the places linked by an output arc to transition t_i are called output places for that transition. The weight on an output arc may be dependent from the marking of the net.

- S is a list of subnets. A subnet can be described with the same five-tuple (P, T, I, O, S), we used above to define the whole timed Petri net model, that is a subnet may contain places, transitions, arcs and other nested subnets, as follows:

```
SUBNET <name>
  <list of local places>
  <list of local transitions>
  <list of inner subnets>
  <list of input arcs>
  <list of output arcs>
END SUBNET
```

A subnet has a name and may contain other nested subnets. The following convention is used for the naming of the objects of the model. Objects' names are always local to the sub-net where the objects are defined. However, an arc may link two objects defined in different subnets. In this case, the arc itself is at a higher level than both the two subnets. The name of the two objects is identified by prefixing them with the names of the nested subnets, until the subnet they are defined in.

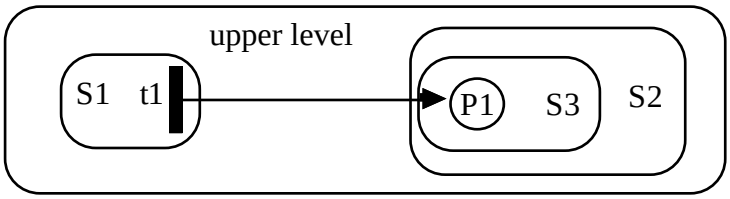


Figure 4.1: Convention for the naming of objects

For instance, consider the case in Figure 4.1, where the output arc is linking transition t1 defined in subnet S1, to place P1 defined inside subnet S3, which is nested in subnet S2. At the topmost level, the output arc will be described as follows:

```
OUTPUT_ARC <from_transition=S1.t1> <to_node=S2.S3.P1> <weight=?>
```

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. Very useful and convenient in the context of HIDE, the possibility of having nested subnets allows the combination of models at the different level of detail. Starting from a coarse model of a system, some parts of it that are of particular interest can be subsequently substituted with a whole more refined subnet, and this procedure can be iteratively repeated until the desired level of detail is achieved. Such a substitution procedure is relatively easy, provided that the interface points between a subnet and the rest of the model have been clearly specified.

It is worthwhile observing that the possibility of defining subnets in the timed Petri net models once again does not imposes any constraints on the automated tools. Passing from the hierarchical view of the model to a flat one is indeed a straightforward procedure,

whereas going to the opposite direction may be a formidable problem. On the other hand, there are tools, like UltraSAN [18], that allow the hierarchy to be exploited, during both the model definition phase and the model solution phase. For those tools, the translation from the timed Petri net paradigm to the specific language definition of the models can keep the information on the hierarchy to best exploit the features of the tool.

The preceding elements define the static structure of a time Petri net. Besides, there is a dynamic behaviour of the model, which evolves from the initial marking M_0 to reach new markings. The basic rules for the evolution of the model are as follows. A transition t_i is said to be *enabled* at time t, if and only if at that time the following conditions are all satisfied:

1. Each of its input places holds at least as many tokens as the weight of the input arc connecting it to t_i.
2. Each of the inhibitor places holds less tokens than the absolute value of the weight of the inhibitor arc connecting it to t_i.
3. The guard for t_i evaluates true for the current marking of the net.

As soon as t_i gets enabled, a random delay is chosen according to the distribution and the memory policy associated to t_i, and a timer starts counting from that delay down to zero. If t_i stays enabled until the timer reaches zero, the transition t_i is said to fire. In this case, the marking of the net is changed as follows:

- Each of the input places is subtracted a number of tokens equal to the weight of the corresponding input arc.
- Each of the output places is added a number of tokens equal to the weight of the corresponding output arc.

On the contrary, if any of the conditions 1), 2), 3) stops to be fulfilled anymore before the delay elapses, then the transition is disabled. The residual value of the timer is held or deleted, depending on the memory policy selected for the transition. Priorities are used to decide which immediate transition is to fire first whenever two immediate transitions get enabled simultaneously: the one with higher priority fires the first.

Notice that the class of timed Petri nets 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 timed Petri net 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, like PANDA [3], GreatSPN [5], SPNP [7], UltraSAN [18], TimeNET [9], Surf2 [11]. If deterministic transitions are included as well, then the model is a DSPN which can be analytically solved with specific tools like UltraSAN, TimeNET. If other kinds of distributions of the transition firing times are included, then the simulation can be used to solve the timed Petri net model. Alternatively, a transformation technique as the one involving fictitious stages [19] can be applied to translate a general distribution into a sequence of exponential stages

(Coxian or phase-type distribution). After that, the model can still be solved by using tools for GSPN models.

5 From UML to the Intermediate model

The main task of this part of the transformation is to project the elements and relations of the UML design to the elements and relations of the intermediate model (IM). From this point of view, first the analysis of the role of the different views and model elements of UML is performed (Section 5.1). The redundancy (fault tolerance) structures are analysed in Section 5.2. After this analysis, the projection is formalised in terms of the UML metamodel in Section 5.3.

5.1 Analysis of UML

The UML model views (diagrams) are analysed in order to identify the *system elements* and relations to be represented first in the Intermediate model and the in the dependability.

- Software or hardware elements are distinguished as listed in the definition of the IM. According to the high-level approach, not only the “natural” software elements as objects, tasks, processes etc. are 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 system elements. The representation of a compound UML element (like a package or a use case) 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 element in the IM. If it is refined and its refinement is relevant then its subcomponents are represented as simple elements and the compound as a whole is represented by a SYS element in the Intermediate model.
- Relations are identified which may result in *error propagation* among the model elements identified above. According to the high-level structural approach, 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 will discuss diagram by diagram the role of the diagrams (from the point of view of the dependability model) and the projection of its elements. We start from the highest level diagram and then look at more and more refined ones. Implementation diagrams will follow at the end.

5.1.1 Use case diagrams

The role of use case diagrams is to identify system level relations. Model elements of use case diagrams include actors, use cases, communication associations among actors and use cases, and generalisations 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 single software element. If a use case is refined and the refinement is relevant, then it is projected into a system element of the IM, which relates the elements resulting from the projection of the UML submodel of the use case.
- Actors represent (roles of) users or external entities which interact directly with the system. Being an external user, an actor is not projected into the IM.
- Communication associations among actors and use cases identify the top level service of the system. If a use case is connected directly to external actor(s) then it is projected into a top level system element of the IM. A communication association is an error propagation path from the system to the actor, however, it is not projected into the IM (since actors are not projected into the IM).
  Usually, a real system is composed of several use cases, many of them being connected directly to actors. From the point of view of dependability, the designer (and user) of a system is usually interested in the dependability of a given service, i.e. of a use case, of the system. Accordingly, dependability measures of such use cases can be computed separately, by a set of dependability models assigned to each use case. This way, each IM includes only a single top level system element. However, all services of the system can also be composed in a single dependability model, computing then measures corresponding to multiple top level system elements.
- Relationships among use cases are represented in UML by generalisation relations with stereotype <<extends>> and <<uses>>.
  - An extends relationship means that a use case includes the behaviour implemented by an other one. It indicates and error propagation path in the direction of the relationship, thus it is projected into the IM.
  - A uses relationships means a similar inclusion of the behaviour, thus it will be projected similarly into an error propagation path in the IM.

An example of a use case and the corresponding IM is shown in Figure 5.1.

5.1.2 Class diagrams

Class diagrams represent the types (descriptors) of the objects of the system. Several declarative extensions and relations are also included. Class diagrams may also contain objects (we will describe objects separately, in object diagrams).

- Since classes are declarative (not implementation) elements, they are not projected directly to elements of the IM. They only identify the relations (described below) among the objects instanciated from the given class. Similarly, inheritance hierarchy of classes is utilised only

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

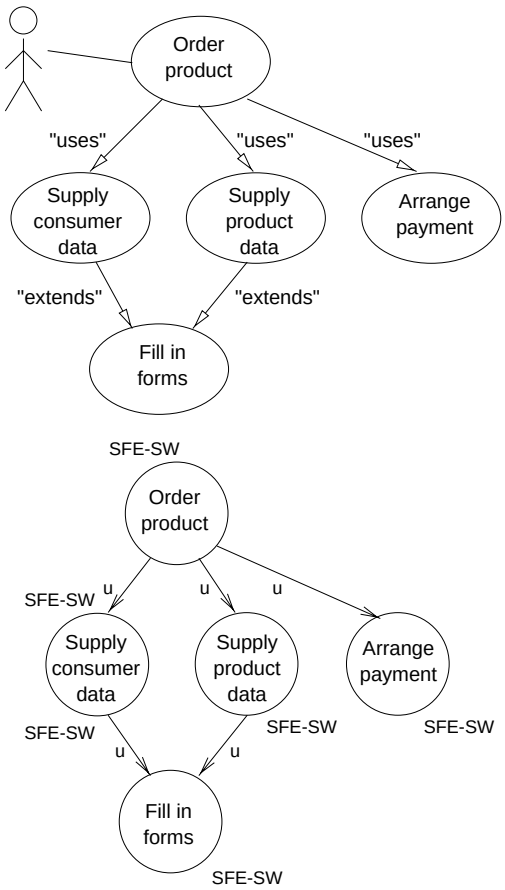


Figure 5.1: A use case diagram and its projection into the IM

Features of classes (attributes, operations, methods) are not projected to any element of the IM. (In a more refined model, attributes can be thought as composite elements of a class.) If a class has attributes, then it can be assumed that it has state, i.e. its objects will be projected to stateful software elements of the IM.

- An interface is a specifier of the externally visible operations of a class (does not have implementation). It is not projected to any element of the IM.
- Associations are binary or n-ary relations among classes. In general, associations mean that the objects instanciated from the corresponding classes know (can name) each other. Usually (especially in collaboration diagrams) communication among objects is possible along the associations.

  Accordingly, an association indicates a potential bidirectional error propagation path among these objects, thus it is projected into the IM. The following additional features might be taken into account:

  - Or-associations (indicating a situation when only one of several possible associations may be valid) are all projected into the IM (a worst case model is generated).

- Multiplicity of association ends are taken into account only in the “default” instantiation of classes (see below).
- Navigability of an association end denotes whether the instance is directly reachable via the association. However, it does not give precise information about the direction of the potential error propagation, since through return values also an unidirectional navigation may result in bidirectional error propagation. Accordingly, each association is projected by default to bidirectional error propagation; further refinement is possible on the basis of behavioural diagrams (collaboration or sequence diagrams).
- Aggregation (as a special association) is projected into an unidirectional error propagation path: the aggregate uses the service of the aggregated elements.
- Composition (as a special association meaning strong ownership and coincident life-times) is projected similarly into an unidirectional error propagation path: the composite uses the service of the sub-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 associations (one object to itself) are not considered.
- Association classes are handled as separate classes having associations with the classes at the endpoints of the association.
- 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. (In general, the designer is asked to include only binary associations, since in this way the assignment of the dependability-related parameters of associations is more easy.)
- Qualifiers are attributes that are used to partition a set of objects associated with an other object across an association. They are not projected into relations of the IM, since they are not relevant from the point of view of error propagation.

• Generalisation is the relationship between a more general element (class) and a more specific one. Generalisation does not indicate an error propagation path, thus it is not projected into the IM (but the inheritance of relations defined by generalisations is taken into account).
• Dependency means a semantic relationship between classes. From the point of view of dependability modelling, 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 an other element for its correct functioning) indicates an error propagation path in the direction of the dependency, thus it is projected into the IM.
• Derived elements are shown in UML only for the sake of clarity, they are not projected into any element of the IM.

Packages are included mainly on class diagrams, this way their projection is described here. Relations among packages have to be taken into account.

- A package is a grouping of model elements. Packages may be nested within other packages. The entire system description can be considered as a single high-level package. Any model element of UML can be included in a package.

  If the content of a package is not described or it is not relevant then the package is projected into a single software element. If the content of the package is described and relevant then it is not projected into any element of the IM, since its content is projected separately.
- Similarly to classes, the <<uses>> dependency of packages may indicate and error propagation path, in the direction of the dependency. If a package is refined, then this dependency is inherited to all elements contained by the package. These dependencies are projected into the IM.

As mentioned earlier, classes (and packages of classes) are declarative elements, they are not projected to elements of the IM. However, in the early phases of the design the instantiation of the model (in the form of object, collaboration, sequence or deployment diagrams) is not available. It might be useful for the designer to have a “default” instantiation of these diagrams, in order to compute rough, preliminary dependability measures. We can establish the following simple rules for the default instantiation:

If a class has multiplicity specification (corresponding to an association) then the value or the lower bound of its range (if it is not equal to zero) can be taken into account.

- If a class has no multiplicity specification or the lower bound of its range is zero, then by default a single instance is taken into account.
- Metaclasses, type classes and parameterised classes (i.e. templates) are not instanciated.

An example of a class diagram and the IM corresponding to its default instantiation is shown in Figure 5.2.

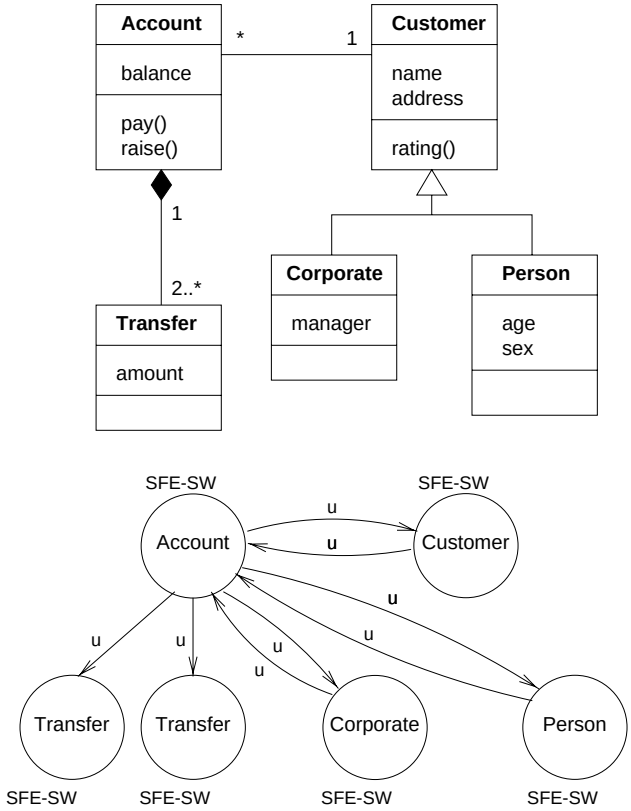


Figure 5.2: A class diagram and its projection into the IM

•*5.1.3 Object diagrams*

Object diagrams include instances, i.e. objects and data values. Some tools do not support object diagrams, objects should be represented in class diagrams (a class diagram with objects only is an object diagram) or in other diagrams (sequence, collaboration, deployment).

- An object is a particular instance of a class. It has identity, its own attribute values (state). An object is projected into a software element of the IM.
- A composite object represents a high-level object made of tightly bound parts. As a composite object is an instance of classes related by composition, its projection is the same as described above, in the case of composition: the composite object and each sub-object are projected into software elements of the IM, with unidirectional error propagation paths from the sub-objects to the composite one.
- Links between objects are instances of associations between classes. They are projected into relations of the IM (between elements representing the objects) in the same way as the associations in the class diagram. By default, a bidirectional error propagation path is included in the IM.

5.1.4 Collaboration diagrams

Collaboration diagrams with messages show the interaction among objects and their links to each other, collaboration diagrams without messages show only the context. Accordingly, collaboration diagrams can be used to identify objects as well as relations among them. Moreover,

the direction and other message properties may indicate the direction of the potential error propagation. Note that use cases are often refined to collaborations, and thus also actors may appear in collaboration diagrams. It helps to identify (or refine) the top level service of the system.

- Objects (object roles) are projected into software elements of the IM. Named objects are projected separately, while unnamed objects (object roles, general class references) are bound to all instances of the given class. Active and passive objects are not distinguished.
- Multiobjects represent sets of objects. If the (minimal) cardinality of the set is known then a multiobject is projected into a set of software elements of the IM. Otherwise, it is projected into a single software element only. Individual objects from the set, that are represented separately (using composition association) are projected separately.
- Links are projected into error propagation paths. If a link is shown on the collaboration diagram without messages then (in the worst case approach) it indicates a bidirectional error propagation path. Messages shown among the links may determine the direction of the error propagation as follows:
  - Signals indicate an unidirectional error propagation from the sender to the receiver.
  - Operations, i.e. procedure calls indicate in general a bidirectional error propagation between the receiver and the sender (the sender uses a service of the receiver but also may cause errors in the receiver by changing its state). Exception is the case when he receiver is stateless, then the error propagation is unidirectional, from the receiver to the sender (only the service of the receiver is used).
  - Guards do not modify the projection (a worst case approach is adopted).
- Pattern structures, parameterised collaborations are not used in the subset of the UML to be analysed. Similarly, object destruction, termination and creation are not allowed.

An example of a collaboration and the corresponding IM is shown in Figure 5.3.

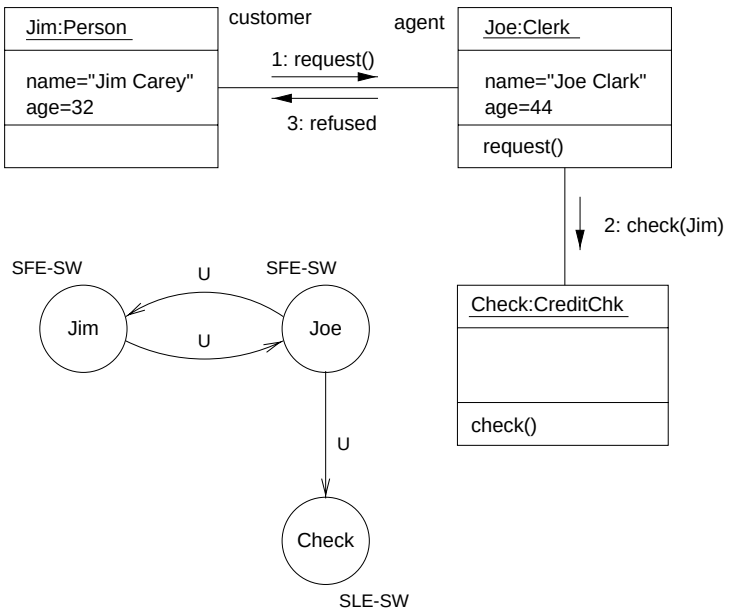


Figure 5.3: A collaboration diagram and its projection into the IM

5.1.5 Sequence diagrams

The role of sequence diagrams is similar to the role of collaboration diagrams, however, in sequence diagrams the links among objects are not shown, instead, the diagram concentrates on the temporal sequence of messages. A generic form describes all possible sequences while an instance form describes only one actual interaction pattern.

From the point of view of high-level, structural dependability modelling, the time sequence of messages is irrelevant, thus the diagram is used to identify objects and indicate the error propagation paths. The projection is the same as in the case of collaboration diagrams:

- Objects (object roles) are projected into software elements of the IM.
- Messages are projected into error propagation paths in the same way as in the case of collaboration diagrams.
- Lifelines, activations and transition times are irrelevant from the point of view of error propagation. However, activations help to identify call actions (procedure calls).
- Object destruction, termination and creation are not enabled in the subset of UML to be analysed.

5.1.6 Statechart diagrams

Statecharts diagrams show the internal behaviour of an object, i.e. the set of states that an object goes through in response to events, together with its actions.

The role of statechart diagrams in high-level dependability modelling is to identify error propagation paths (not shown in other diagrams). This can be done by the analysis of the actions shown in the statechart. The sequence of states and the actual behaviour is not relevant.

- Actions expressed in the form of send clauses indicate an error propagation path. A send clause contains the destination expression (which evaluates to an object or a set of objects), a message name and its parameters. Accordingly, there is (are) error propagation path(s) between the given object and the destination one(s).
- Actions without send clauses can be mapped only indirectly to error propagation paths. These actions result in events (signal events or call events), which may trigger changes in other objects, as described in the statecharts of these objects. Accordingly, there is a potential error propagation path among the given object and all others which include these events as triggers.

A message (resulting from an action) may be a signal or an operation (procedure call). Accordingly, the direction of the error propagation path is refined like in the case of messages in collaboration diagrams.

The special analysis of statecharts in the case of redundancy managers (to derive the static relations of objects involved in a redundancy structure) is described in Section 5.4.

Activity charts are considered as a special case of statecharts.

5.1.7 Component diagrams

Component diagrams show the dependencies among software components. They have only a type form without instances. From the point of view of dependability modelling, they can be used to identify the relations among objects realised by the components.

- A component type represents a piece of implementation of the system. Only executable components are relevant.
- Dependencies are relevant only among run-time components. The <<calls>> dependency indicates a potential error propagation path among the objects realised by the components, the direction is from the called to the caller (the same as the direction of the dependency).

If a component is not instanciated on deployment diagrams then a default instantiation (a single component) may be useful.

5.1.8 Deployment diagrams

Deployment diagrams show hardware components and the configuration of run-time components on them. Deployment diagrams have instance form, this way component instances are on node instances, and objects may reside in component instances.

- Nodes are run-time physical objects, usually hardware resources. They are projected into hardware elements in the IM.
- Objects realised by components are projected into software elements of the IM.

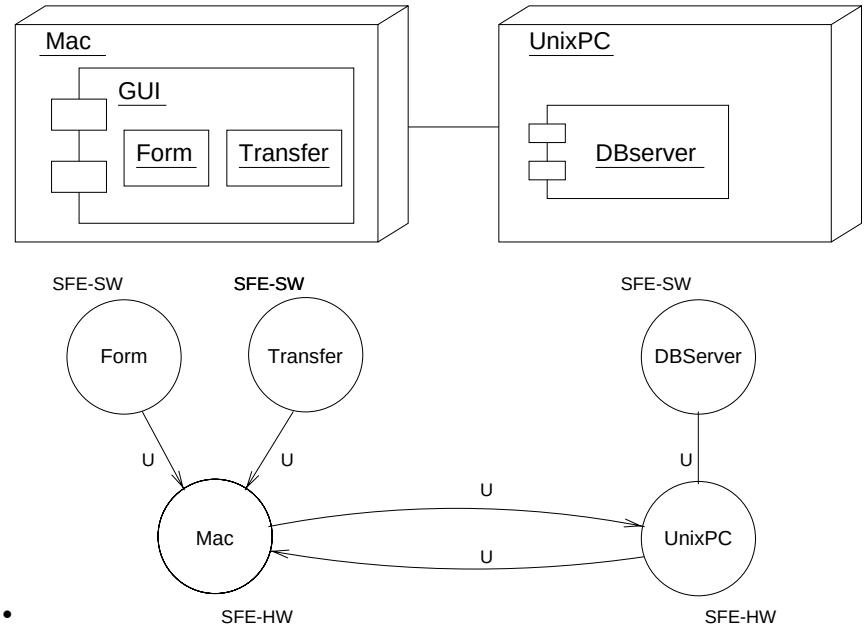


Figure 5.4: A deployment diagram and its projection into the IM

Components represent pieces of run-time software. If a component is refined, i.e. the set of objects realised by the component is given, then the component is not projected into a separate software element of the IM (note that the set of objects is projected into software elements). If a component is not refined then it is projected into a single software element of the

IM. Deployment relations among nodes and components and 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.

An example of a deployment diagram and its projection into the IM are shown in Figure 5.4.

5.2 Analysis of structures

Redundancy structures require a non-trivial projection into the IM, which is different from the one described in the previous subsections.

Redundancy structures are identified by stereotyped classes (or objects): the redundancy manager, the adjudicators and the variants are stereotyped as <<redundancy manager>>, <<adjudicator>> and <<variant>>, respectively.

A redundancy structure is projected into the IM in the following way.

- The redundancy manager, the adjudicators and the variants are projected into software or hardware elements of the IM.
- The structure as a whole (identified by the redundancy manager) is projected into an element called “fault tolerant structure” (FTS) element. This element is connected to the elements representing the redundancy manager, the adjudicators and the variants using special relations “is composed of”, collectively represented in the Intermediate model by means of a type C hyperarc.

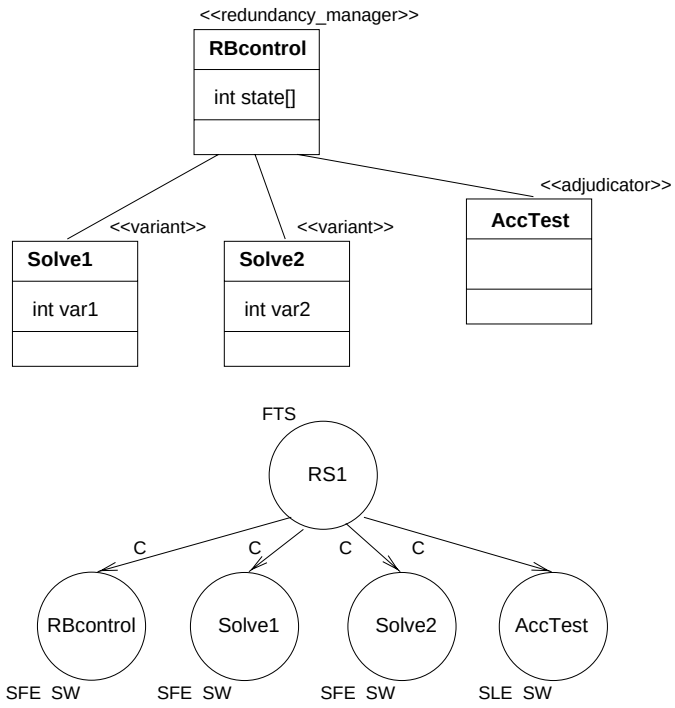


Figure 5.5: Projection of a simple redundancy structure (recovery blocks)

Due to the redundancy scheme implemented by the manager, the “is composed of” relations indicate nontrivial error propagation paths from the elements of the structure towards the clients (which use the service of the structure as a whole). In most of the cases, this error propagation can be described by a fault tree. If the redundancy structure is obtained from the fault-tolerance library, then the fault tree is available from the library as well (generated by dependability experts). In the case of user-defined structures, the fault tree should be derived automatically, as part of the projection into the IM. This subtask will be described later, in Section 5.4.
An example of a simple redundancy structure and its projection are depicted in Figure 5.5.

5.3 Definition of the projection

A (more formal) definition of the projection can be given in terms of metamodel elements of UML. (Note that the UML design database usually implements the metamodel itself, this way the implementation of the projection should also be based on metamodel definition.)
In comparison with Section 5.1, here we follow an inverse direction and list the UML metamodel elements which are projected into software elements (Table 5.1), hardware elements (Table 5.2) and “uses service of” relations (Table 5.3). Note that an unidirectional “uses the service of” relation (from the client to the server) indicates an error propagation path in the reverse direction (i.e. from the server to the client).

Metamodel element	**Remark**
UseCase	If its refinement is not relevant (it is stereotyped)
Class	By a default instantiation
Package	If its refinement is not relevant (it is stereotyped)
Object	Usually only objects are projected
Component	By a default instantiation

Table 5.1 UML metamodel elements projected into software elements of the IM

Metamodel element	**Remark**
Node	

Table 5.2 UML metamodel elements projected into hardware elements of the IM

Note that redundancy structures are projected, as defined in Section 5.2, by introducing an additional element (FTS) into the IM, connected to the objects involved in the structure by “is composed of” relations.

Metamodel element	Remark
Generalisation with stereotype <<extends>> or <<uses>>	Direction from *supertype* to *subtype*
Dependency with stereotype <<uses>> or <<calls>>	Direction from *client* to *supplier*
Association (general)	Bidirectional
Association having AssociationEnd with attribute “aggregate”	Direction from the aggregate end to the normal one
Association having AssociationEnd with attribute “composite”	Direction from the composite end to the normal one
Link (general)	Bidirectional
Link having LinkEnd as instance of an AssociationEnd with attribute “aggregate”	Direction from the aggregate end to the normal one
Link having LinkEnd as instance of an AssociationEnd with attribute “composite”	Direction from the composite end to the normal one
Message	Bidirectional, except when the receiver is stateless, in which case its direction is from the *sender* to the *receiver*
SendAction	Direction from the *target* or to all elements having context with the dispatched Signal to the context having the Action
CallAction	Bidirectional between the context having the Action and the *target* or all elements having context with the dispatched Operation, except when the target is stateless, in which case its direction is from context of the Action to the *target*

Table 5.3 UML metamodel elements projected into “uses service of” relations of the IM

The algorithmic description of the projection is as follows.

1. Projection of hardware elements (Table 5.2)

Note that hardware elements with stereotype <<redundancy manager>>, <<variant>> or <<adjudicator>> are projected separately.

- For each Node with stereotype <<stateless>>, add to the IM an element “SLE-HW”, with parameters copied from the tagged values.

- For each Node with stereotype <<stateful>>, add to the IM an element “SFE-HW”, with parameters copied from the tagged values.

2. Projection of software elements (Table 5.1)

The values of the parameters are copied from the tagged values of the stereotype. Note that software elements with stereotype <<redundancy manager>>, <<variant>> or <<adjudicator>> are projected separately.

- For each UseCase, do the following:
  - if its stereotype is <<stateless>> then add to the IM an element “SLE-SW”.
  - if its stereotype is <<stateful>> then add to the IM an element “SFE-SW”.
  - ιf there is no stereotype, then add to the IM a subsystem element “SYS”.

- For each Package, do the following:
  - if its stereotype is <<stateless>> then add to the IM an element “SLE-SW”.
  - if its stereotype is <<stateful>> then add to the IM an element “SFE-SW”.

- For each Object, do the following:
  - if its stereotype is <<stateless>> then add to the IM an element “SLE-SW”.
  - if its stereotype is <<stateful>> then add to the IM an element “SFE-SW”.
  - if there is no stereotype then look at the ancestor classes; the lowest level class with stereotype determines the projection:
    - if the stereotype is <<stateless>> then add to the IM an element “SLE-SW”.
    - if the stereotype is <<stateful>> then add to the IM an element “SFE-SW”.

- For each Class without instantiation (no Object of the Class is available on the diagrams), do the following:
  - if the stereotype is <<stateless>> then add to the IM exactly N elements “SLE-SW”, where N is the maximum of the lower bounds of the Multiplicity values corresponding to the Class.
  - if the stereotype is <<stateful>> then add to the IM exactly N elements “SFE-SW”, where N is the maximum of the lower bounds of the Multiplicity values corresponding to the Class.

- For each Component without refinement, do the following:
  - if the stereotype is <<stateless>> then add to the IM an element “SLE-SW”.
  - if the stereotype is <<stateful>> then add to the IM an element “SFE-SW”.

3. Projection of relations (Table 5.3)

Here we start listing the (already projected) elements which are in the IM and look for the "uses service of" (U) relations, which indicate error propagation paths (in reverse direction). In order to avoid the duplicate projection of UML relations, they have to be marked after the first projection. The parameters of the relations are copied from the tagged values assigned to the relations in UML. If a relation is not stereotyped as <<propagation>> or the tagged values indicate a zero probability, then the projection is skipped. Note that relations among elements with stereotype <<redundancy manager>>, <<variant>> or <<adjudicator>> are projected separately.

- For each software element (SFE-SW or SLE-SW) projected from a UseCase:
  - For each Generalisation from the UseCase with stereotype <<uses>> or <<extends>>, add to the IM an U relation from the element corresponding to the UseCase *supertype* to the element corresponding to the UseCase *subtype*.
  - For each UML element which is in the context of a UseCase without stereotype <<stateful>> or <<stateless>>, add to the IM an "is composed of" (C) relation from the SYS element corresponding to the UseCase to the IM element corresponding to the UML element.

- For each software element (SFE-SW or SLE-SW) projected from a Package:
  - For each Dependency from the Package with stereotype <<uses>>, add to the IM an U relation from the element corresponding to the *client* to the element corresponding to the *supplier*. If such element does not exist in the IM, then do it for each IM element corresponding to the UML elements of its refinement or instantiation.

- For each software element (SFE-SW or SLE-SW) projected from an Object:
  - For each Dependency from the Object or from one of its ancestors, with stereotype <<uses>>, add to the IM an U relation from the element corresponding to the *client* to the element corresponding to the *supplier*. If such an element does not exist in the IM, then do it for each IM element corresponding to the UML elements of its refinement or instantiation.
  - For each ancestor Class of the Object, for each Association with attribute "composite" or "aggregate" on the AssociationEnd at the Class, add to the IM an U relation from the element corresponding to the Object to the instance of the target class.
  - For each ancestor Class of the Object, for each Association without attribute "composite" or "aggregate" on the AssociationEnd at the Class, add to the IM an U relation from the element corresponding to the Object to the instance of the target Class of the Association, and also an other one in the reverse direction.

- For each Link of the Object with attribute “composite” or “aggregate” on the AssociationEnd at the ancestor Class, add to the IM an U relation from the element corresponding to the Object to the element corresponding to the target object.
- For each Link of the Object without attribute “composite” or “aggregate” on the AssociationEnd at the ancestor Class, add to the IM an U relation from the element corresponding to the target element of the association (at the other AssociationEnd) to the element corresponding to the Object, and also an other one in reverse direction.
- For each Message sent by the Object, add to the IM an U relation from all elements corresponding to Objects receiving the message to the element corresponding to the Object, and also an other one in reverse direction. Exception is the case when a receiver is represented in the IM by a *stateless* element, which means that the direction of the U relation is only from the IM element corresponding to the Object to the receiver.
- For each ancestor Class of the Object, for each SendAction in the context of the Class, add to the IM an U relation from all elements corresponding to instances having context including the dispatched Signal to the element corresponding to the Object.
- For each ancestor Class of the Object, for each CallAction in the context of the Class, add to the IM an U relation from all elements corresponding to instances having context including the dispatched Operation (targets) to the element corresponding to the Object, and also an other one in reverse direction. Exception is the case when a target is represented in the IM by a *stateless* element, which means that the direction of the U relation is only from the IM element corresponding to the Object to the target.

- For each software element (SFE-SW or SLE-SW) projected from a Class by default instantiation: do the same as in the case of elements corresponding to Objects.

- For each software element (SFE-SW or SLE-SW) projected from a Component (by default instantiation):
  - For each Dependency from the Component, with stereotype <<calls>>, add to the IM an U relation from the element corresponding to the *client* to the element corresponding to the *supplier*. If such an element does not exist in the IM, then do it for each IM element corresponding to the UML elements of its refinement.

- For each hardware element (SFE-HW or SLE-HW) projected from a Node:
  - For each UML element deployed on the node, add to the IM an U relation from the IM element representing the UML element to the element representing the node.
  - For each Association, add to the IM an U relation from the element representing the target element of the Association (at the other AssociationEnd) to the element representing the Node, and also an other one in reverse direction.

4. Projection of redundancy structures

Note that during the projection, parameters of the IM elements are copied from the tagged values assigned to the elements in UML.

- For each Class or Object with stereotype <<redundancy manager>>, add to the IM a fault-tolerance element (FTS).

- For each Class or Object with stereotype <<redundancy manager>> and <<stateless>> add to the IM an element “SLE-SW”. In the case of a Node, add an element “SLE-HW”.
- For each Class or Object with stereotype <<redundancy manager>> and <<stateful>> add to the IM an element “SFE-SW”. In the case of a Node, add an element “SFE-HW”.

- For each Class or Object with stereotype <<variant>> and <<stateless>> add to the IM an element “SLE-SW”. In the case of a Node, add an element “SLE-HW”.
- For each Class or Object with stereotype <<variant>> and <<stateful>> add to the IM an element “SFE-SW”. In the case of a Node, add an element “SFE-HW”.

- For each Class or Object with stereotype <<adjudicator>> and <<stateless>> add to the IM an element “SLE-SW”. In the case of a Node, add an element “SLE-HW”.
- For each Class or Object with stereotype <<adjudicator>> and <<stateful>> add to the IM an element “SFE-SW”. In the case of a Node, add an element “SFE-HW”.
- For each FTS element in the IM, add to the IM an “is composed of” (C) relation from the FTS element to the elements corresponding to the UML elements with stereotype <<redundancy manager>>, <<variant>> or <<adjudicator>>.

5.4 Automatic derivation of fault trees of redundancy structures

The generation of a fault tree corresponding to a redundancy structure (not included in the library of schemes) requires a non-trivial analysis of the behaviour, 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 special states and events in the statechart, as described in Section 3.3. Recalling shortly, the following stereotypes and extensions are used:

- Failure states (with stereotype <<failure>>).
- Failure events (with stereotype <<failure>>),
- Response events (with stereotype <<response>>),
- Adjudicators distinguished as tester (with stereotype <<tester>>) or comparator (with stereotype <<comparator>>).
- Additional tagged values characterising common mode failures and error detection coverage of adjudicators.

The fault tree is generated by the composition of the subtrees of (i) response events, (ii) failure events and (iii) failure states. The following steps are performed (in [8] a similar algorithm is proposed for reliability modelling):

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

  On a trajectory, the incoming events identify the objects contributing to the response, while guards identify the test and compare actions of the redundancy manager(s) or adjudicator(s). The fault tree corresponding to a trajectory is the OR relation of the events which may lead to the error of the response on the given trajectory. The following events are combined:
  - Separate failure of a variant without testing.
  - Common mode failure of variants without testing.
  - Common mode failure of a variant and its tester.
  - Failure of a variant escaping the test.
  - Common mode failure of variants which are compared.
  - Common mode failure of variants and the comparator.
  - Separate failure of the redundancy manager (providing the response).
- Backward reachability analysis from explicit failure events generates the set of trajectories leading from initial state to the failure event. The OR relation of the subtrees of the trajectories forms the fault tree corresponding to the failure event.

  On a trajectory, the events and conditions identify the objects, failures of which lead to the failure event. The fault tree corresponding to a trajectory is the AND relation of these failures. The following events are combined:
  - Separate failure of a variant detected by a tester.
  - Common mode failure of variants detected by a tester.
  - Separate or common mode failures of variants detected by a comparator.
- Failure states usually provide explicit failure events, this way they are covered by the previous point. If a failure state is without failure events (fail stop) then it can be handled in the same way as failure events: Backward reachability analysis from the failure state generates the set of trajectories leading from initial state to the failure state. The OR relation of the subtrees of the trajectories forms the fault tree corresponding to the failure state. On a trajectory, the events and guards identify the objects whose failures lead to the failure state. The fault tree corresponding to a trajectory is the AND relation of these failures.

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

variants fail then the scheme will fail as well, reporting this towards the client. The statechart of the redundancy manager CW is presented in Figure 5.6.

Response events are `resp1` and `resp2`, while failure events are `failure1` and `failure2`. The following trajectories and failures can be recognised as shown in Figure 5.7:

Trajectory to response event `resp1`: Event `respw1` is the result of object W1, its testing is made by CW. The response also requires CW, the redundancy manager. Failures along this path are the common mode failure of W1 and CW, and the separate failure of CW. (The failure of CW covers the case when both W1 and CW fail simultaneously.) Tr1 accounts for these events.

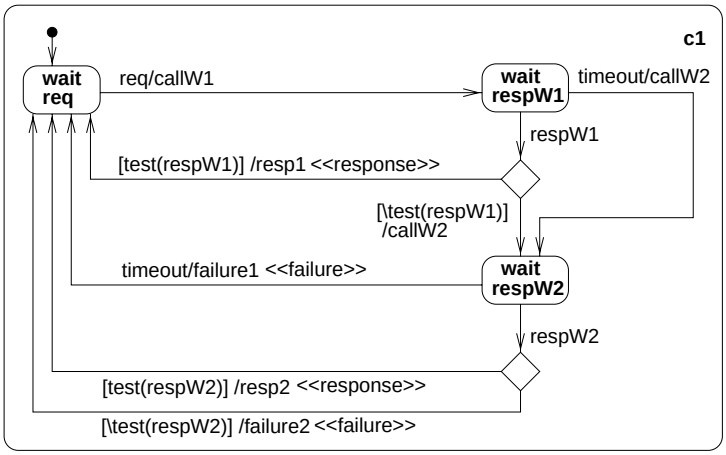


Figure 5.6: Statechart of the redundancy manager

Trajectory to response event `resp2`: The guard condition shows that W1 fails. Event `respw2` is the result of object W2, its testing is made by CW. The response also requires CW, the redundancy manager. The fault tree Tr2 accounts for the failures along this path: the separate failure of CW as well as the common mode failure of W2 and CW (in the case of the failure of W1).

Trajectory to failure event `failure1`: The guard condition shows that W1 fails (tested by CW), the event `timeout2` shows that also W2 fails. The fault tree Tr3 accounts for the separate failures of W1 and W2 the failures along this path.

Trajectory to failure event `failure2`: The guard conditions show that both W1 and W2 fail (tested by CW). Tr4 describes the separate failures of W1 and W2.

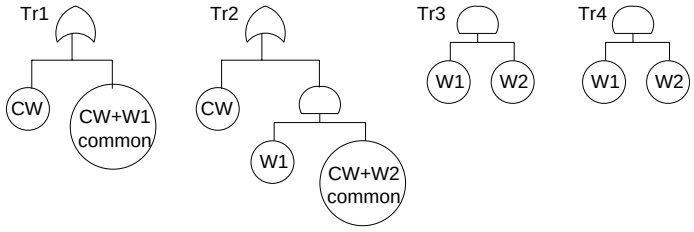


Figure 5.7: Fault trees corresponding to the trajectories of the redundancy manager

6 From the Intermediate model to timed Petri nets

The timed Petri Net dependability model is generated and istanciated only by using the information contained in the intermediate model, without any further need of the UML system specification. The final model is built in two steps:

1 A set of subnets (basic subnets hereafter) are generated for the elements of the intermediate model. Also, a set of input/output arcs are generated at this step which link places and transitions that are inside the basic subnets. These arcs are at the highest hierarchy level of the timed Petri net model.
2 A set of failure/repair propagation subnets plus another set of input/output arcs are generated for the hyperarcs of the intermediate model. These arcs link among them basic and propagation subnets, and are at the highest hierarchy level of the timed Petri net model

We take advantage from the modularity of the timed Petri net models defined above, to build the whole model as a collection of subnets, linked by input and output arcs over well-specified interface places. These input/output arcs linking subnets are all defined at the highest level of the model. Notice that in the following some of the fields of the timed Petri net specification that are not needed to understand the transformation are left unspecified.

6.1 Basic subnets

A set of simple basic subnets are generated for the elements of the intermediate model. These basic subnets represent:

1 The failure and repair processes for the SLE-HW, SFE-HW, SLE-SW, SFE-SW element types of the intermediate model.
2 A set of interface places for FTS and SYS element types of the intermediate model.

Let us now describe the basic subnets that are generated for each of the type of elements of the intermediate model. The basic subnets for SLE-HW type of element includes a failure subnet and a repair subnet. The failure subnet for a SLE-HW element is the one shown in Figure 6.1, which consists of:

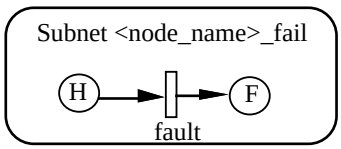


Figure 6.1 Basic failure subnet for a SLE-HW element

- Two interface places, namely H, and F. These places are interfaces towards the other subnets of the model. A token in place H represents the healthy state of the element, where no faults have appeared yet. A token in place F means that a fault has lead to a failure of the element. Obviously the absence of an internal state prevents the element from having errors.

- One single transitions, called fault. The transition fault represents the failure of the element as soon as it is hit by a fault. The firing time of the transition fault is obtained from the field <fault_occurrence> of the intermediate model element description.
- A single token, which occupies place H at the beginning.

Whenever this can be helpful for a better understanding, we also give timed Petri net description of the subnets we are defining. The timed Petri net description of the subnet SLE-HW is as follows:

```
SUBNET <name=<element_name>_fail>
  PLACE <name=H> <tokens=1> <bound=1>
  PLACE <name=F> <tokens=0> <bound=1>
  TRANSITION <name=fault> <random_variable=?> <memory_policy=?>
        <guard=TRUE>
  INPUT_ARC <from_place=H> <to_transition=fault> <weight=1>
  OUTPUT_ARC <from_transition=fault> <to_place=F> <weight=1>
END SUBNET
```

The basic repair subnet for a SLE-HW element is the one called <element_name>_rep shown in Figure 6.2, which consists of:

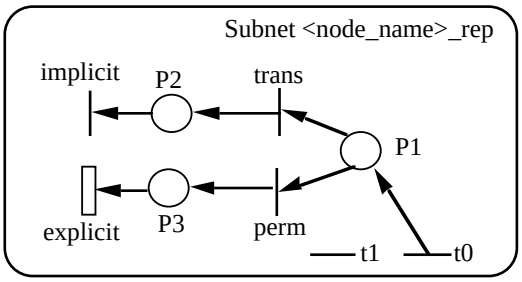


Figure 6.2 Basic repair subnet for a SLE-HW element

- Three places, namely P1, P2, and P3. The two places P2 and P3 are interface places towards other repair subnets, as it will be explained in the following. A token in P1 means that a failure of the element has occurred and therefore the repair actions need to be performed. A token in place P2 and P3 means that an implicit and explicit repair action is being performed, respectively. Note that for a hardware stateless component, the repair is implicit if a transient fault occurred, and explicit (fault-treatment) if the fault is permanent.
- Six transitions. Transitions trans and perm discriminate between transient and permanent faults, according to the relative occurrence probabilities given in field <permanent/transient> of the intermediate model element description. The other four transitions are all interface transitions towards other subnets. Transition implicit and explicit model the immediate execution of an implicit and explicit repair, respectively. Additional input arcs coming form the basic failure subnets of other elements will be added further to the transition implicit and explicit, to represent the awaiting for the repair of the other elements. Transition t0 and t1 are used to trigger the repair activities, by inserting a token in place P1, if there is not yet.
- No tokens are in the subnet <element_name>_rep at the beginning.

The detailed timed Petri net description of the repair subnet is the following one:

```
SUBNET <name=<element_name>_rep>
  PLACE <name=P1> <tokens=0> <bound=1>
  PLACE <name=P2> <tokens=0> <bound=1>
  PLACE <name=P3> <tokens=0> <bound=1>
  TRANSITION <name=t0> <random_variable=instantaneous>
       <memory_policy=?> <guard=(m(P1)=0)> <priority=1>
  TRANSITION <name=t1> <random_variable=instantaneous>
       <memory_policy=?> <guard=(m(P1)=1)> <priority=1>
  TRANSITION <name=trans>
         <random_variable=instantaneous probability=transient>
         <memory_policy=?> <guard=TRUE> <priority=0>
  TRANSITION <name=perm>
         <random_variable=instantaneous probability=permanent>
         <memory_policy=?> <guard=TRUE> <priority=0>
  TRANSITION <name=implicit> <random_variable=instantaneous>
         <memory_policy=?> <guard=TRUE> <priority=0>
  TRANSITION <name=explicit> <random_variable=?> <memory_policy=?>
         <guard=TRUE>
  INPUT_ARC <from_place=P1> <to_transition=trans> <weight=1>
  INPUT_ARC <from_place=P1> <to_transition=perm> <weight=1>
  INPUT_ARC <from_place=P2> <to_transition=implicit> <weight=1>
  INPUT_ARC <from_place=P3> <to_transition=explicit> <weight=1>
  OUTPUT_ARC <from_transition=t0> <to_place=P1> <weight=1>
  OUTPUT_ARC <from_transition=trans> <to_place=P2> <weight=1>
  OUTPUT_ARC <from_transition=perm> <to_place=P3> <weight=1>
END SUBNET
```

Note that transition t0 and t1 have a priority lower than the default value for immediate transitions, that is one. The reason for such a choice will become clear when the failure propagation subnets will have been described.
A set of arcs is then added, which link the failure and the repair subnet of the SLE-HW element. These arcs are defined as follows:

```
INPUT_ARC <from_place=<element_name>_fail.F>
       <to_transition=<element_name>_rep.t0> <weight=1>
INPUT_ARC <from_place=<element_name>_fail.F>
       <to_transition=<element_name>_rep.t1> <weight=1>
OUTPUT_ARC <from_transition=<element_name>_rep.implicit>
       <to_place=<element_name>_fail.H> <weight=1>
OUTPUT_ARC <from_transition=<element_name>_rep.explicit>
       <to_place=<element_name>_fail.H> <weight=1>
```

These arcs link the two basic subnets of the element as graphically shown in Figure 6.3, where only the interface elements involved in the linking are depicted. Notice that transitions t0 and t1 always get simultaneously enabled: their guards select which one between them has to fire depending on the marking of place P1.

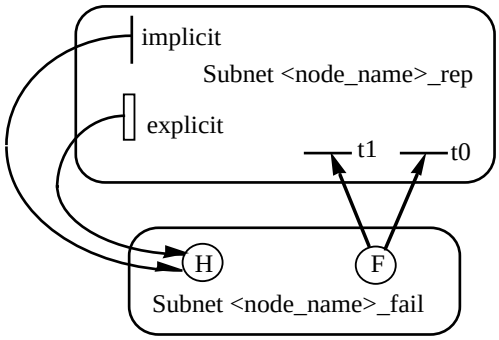


Figure 6.3: Arcs linking the two basic subnets for a SLE-HW element

Consider now a SFE-HW type of element. In this case, the basic failure subnet in the one shown in Figure 6.4.

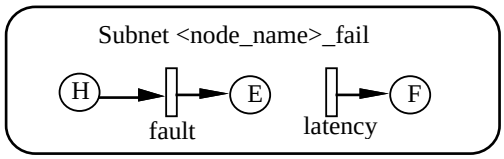


Figure 6.4: Basic failure subnet for a SFE-HW type of element

This subnet contains:

- Three interface places, namely H, E, and F. A token in place H represents the healthy state of the element, where no faults have appeared yet. A token in place E represents the presence of an erroneous internal state, and a token in place F means that an erroneous internal state has lead to a failure of the element.
- Two transitions, namely fault and latency. The transition fault represents the corruption of the internal state of the element due to the activation of a fault. The transition latency models the time needed for errors to generate a failure of the element.
- A single token at the beginning, which occupies place H (the element is in a healthy state at the beginning of operations).

The timed Petri net description of the subnet shown in Figure 6.4 is as follows:

```
SUBNET <name=<element_name>_fail>
  PLACE <name=H> <tokens=1> <bound=1>
  PLACE <name=E> <tokens=0> <bound=1>
  PLACE <name=F> <tokens=0> <bound=1>
  TRANSITION <name=fault> <random_variable=?> <memory_policy=?>
           <guard=TRUE>
  TRANSITION <name=latency> <random_variable=?> <memory_policy=?>
           <guard=(m(E)=1)>
  INPUT_ARC <from_place=H> <to_transition=fault> <weight=1>
  OUTPUT_ARC <from_transition=fault> <to_place=E> <weight=1>
  OUTPUT_ARC <from_transition=latency> <to_place=F> <weight=1>
END SUBNET
```

Notice that transition latency has a guard which enables it as soon as a token reaches place E. Once an error has occurred, transition latency remains enabled until the token is removed from

place E (the error recovery is performed). During its enabling period, many tokens can reach the place F.

The basic repair subnet for a SFE-HW element is exactly the same as the one for the SLE-HW type of elements. However, the arcs that link the two subnets are now as follows:

```
INPUT_ARC <from_place=<element_name>_fail.F>
       <to_transition=<element_name>_rep.t0> <weight=1>
INPUT_ARC <from_place=<element_name>_fail.F>
       <to_transition=<element_name>_rep.t1> <weight=1>
INPUT_ARC <from_place=<element_name>_fail.E>
       <to_transition=<element_name>_rep.implicit> <weight=1>
INPUT_ARC <from_place=<element_name>_fail.E>
       <to_transition=<element_name>_rep.explicit> <weight=1>
OUTPUT_ARC <from_transition=<element_name>_rep.implicit>
       <to_place=<element_name>_fail.H> <weight=1>
OUTPUT_ARC <from_transition=<element_name>_rep.explicit>
       <to_place=<element_name>_fail.H> <weight=1>
```

The two subnets are therefore linked as shown in Figure 6.5. The two transitions implicit and explicit remove the token from the place E, modelling the error recovery of the element.

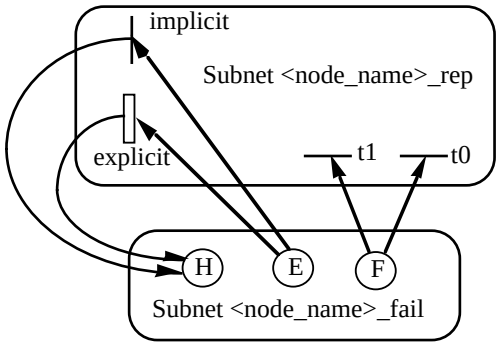


Figure 6.5: Arcs linking the two basic subnets for a SFE-HW element

For a SLE-SW and SFE-SW elements, the basic failure subnets are the same as for the SLE-HW and SFE-HW elements, respectively. The place P1 is for both the two cases an interface place for other repair subnets. The basic repair subnets in the two cases are as follows:

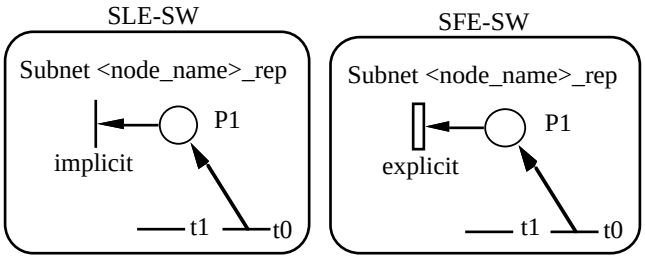


Figure 6.6: Basic repair subnets for the SLE-SW and SFE-SW elements

The repair for a stateless software is only implicit, therefore a single immediate transition is sufficient for the repair subnet model. For the stateful software, the repair is only explicit, be-

cause the error recovery must be necessarily performed. The repair subnets are linked to the corresponding basic failure subnets as shown in Figure 6.7.

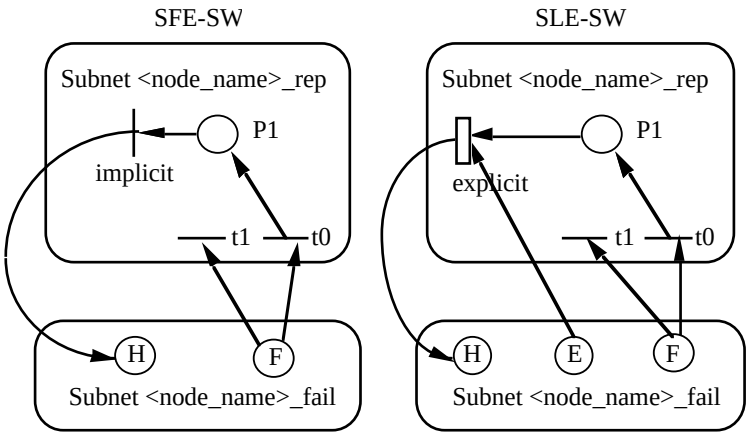


Figure 6.7: Arcs linking the basic subnets for software elements

Now, consider the case of a element of type FTS or SYS. In this case, we only generate a basic subnet with two interface places, as shown in Figure 6.8. Also, for a SYS element whose attrbute list indicates a dependability measure to be evaluated, a directive is generated for the evaluation purposes, specifying the measure to be evaluated by analysing the final dependability timed Petri net model. The measure of interest is always evaluated through the stochastic process describing the probability for a token to be in place F of the SYS basic subnet. A token is put in place H at the beginning.

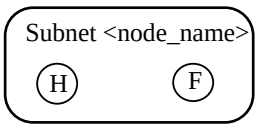


Figure 6.8: Basic subnet for a FTS or SYS type element

All the subnets listed above are generated one after the other by inspecting the list of elements of the intermediate model. Notice that all the parameters needed to define the subnets are found in the intermediate model in the obvious fields. The only case in which the parameters are not found in the elements of the intermediate model is when a more refined submodel is to be included in the final timed Petri net. The directives for such an inclusion are implicitely given in the intermediate model, by leaving parameters unspecified. Consider for instance a stateful element (either hardware or software) for which the process of fault occurrence must be modelled through the inclusion of a subnet, while the error latency does not.

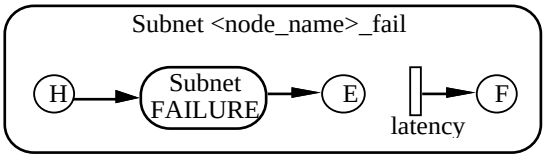


Figure 6.9: Inclusion of a refined submodel into a basic subnet

The basic subnet will be accordingly built to include a nested subnet representing the refined submodel, as shown in Figure 6.9. This nested subnet is linked with input and output arcs to the other local objects of the basic subnet. Obviously, the structure of the nested subnet must be such that the inclusion can be performed automatically, that is the interface points are to be precisely specified and declared to the external.
Accordingly, the timed Petri net description of the subnet shown in Figure 6.9 would be as follows:

```
SUBNET <name=<element_name>_fail>
     PLACE <name=H> <tokens=1> <bound=1>
     PLACE <name=E> <tokens=0> <bound=1>
     PLACE <name=F> <tokens=0> <bound=1>
     TRANSITION <name=latency> <random_variable=?> <memory_policy=?>
            <guard=(m(E)=1)>
     SUBNET <name=FAILURE>
       <description of the subnet>
     END SUBNET
     INPUT_ARC <from_place=H>
          <to_transition=FAILURE.{a transition of subnet FAILURE}>
          <weight=1>
     OUTPUT_ARC <from_transition=FAILURE.{a transition of subnet
          FAILURE}> <to_place=E> <weight=1>
     OUTPUT_ARC <from_transition=latency> <to_place=F> <weight=1>
END SUBNET
```

Notice the prefixed names of the elements inside the included subnet. This inclusion procedure is exactly the same for a refined model specifying other activities of the basic subnets, as the latency or repair processes.

6.2 Failure/Repair propagation subnets

During the previous step of the model generation, a set of basic subnets were built. The basic subnets of a element are up to now completely disjoint from the subnets of other elements. By examining the hyperarcs of the intermediate model, we will now describe the generation of a set of propagation subnets, which link the basic sub-nets of elements among them.
For each pair of elements A and B for which an hyperarc exists in the intermediate model, a failure propagation subnet and a set of arcs is added to the timed Petri net model. Depending on the type of hyperarc, a repair propagation can also be added to the model.

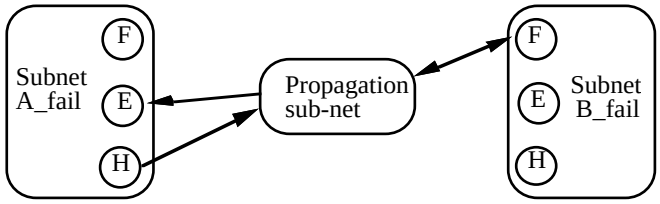


Figure 6.10: Subnet for a failure propagation from B to A

Let us start with the case of a hyperarc of type U. The following Figure (6.10) shows the role of a propagation subnet in the case the stateful element A uses the stateful element B (hardware or software is not relevant). Only the interface places are shown outside the basic subnets of the two elements. The propagation subnet (probabilistically) moves a token from place A.H to A.E, depending on the state of the used element, namely the marking of B.F. In Figure 6.10, the bi-directional arc is a shorthand graphical notation for a pair of input/output arcs linking a place and a transition.

More precisely, for each hyperarc of type U from element A to element B in the intermediate model, the propagation subnet called B->A shown in Figure 6.11 is added to the list of subnets of the timed Petri net model:

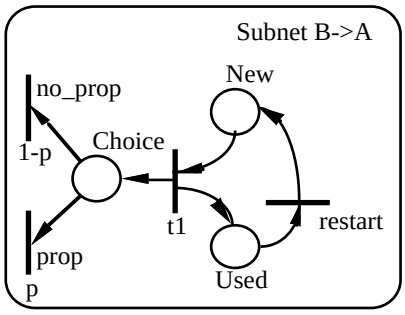


Figure 6.11: Propagation subnet for U type of hyperarc

Parameter p of the subnet is obtained from the parameter prop_prob of the intermediate model hyperarc. The timed Petri net description of this subnet is as follows:

```
SUBNET <name=B->A>
  PLACE <name=New> <tokens=1> <bound=1>
  PLACE <name=Used> <tokens=0> <bound=1>
  PLACE <name=Choice> <tokens=0> <bound=1>
  TRANSITION <name=restart> <random_variable=instantaneous>
         <memory_policy=?> <guard=(m(B.H)=1)> <priority=0>
  TRANSITION <name=t1 <random_variable=instantaneous>
         <memory_policy=?> <guard=TRUE> <priority=0>
  TRANSITION <name=prop>
         <random_variable=instantaneous, probability p>
         <memory_policy=?> <guard=TRUE> <priority=0>
  TRANSITION <name=no_prop>
         <random_variable=instantaneous, probability 1-p>
         <memory_policy=?> <guard=TRUE> <priority=0>
  INPUT_ARC <from_place=New> <to_transition=t1> <weight=1>
  INPUT_ARC <from_place=Used> <to_transition=restart> <weight=1>
  INPUT_ARC <from_place=Choice> <to_transition=prop> <weight=1>
  INPUT_ARC <from_place=Choice> <to_transition=no_prop> <weight=1>
  OUTPUT_ARC from_transition=restart> <to_place=New> <weight=1>
  OUTPUT_ARC <from_transition=t1> <to_place=Used> <weight=1>
  OUTPUT_ARC from_transition=t1> <to_place=Choice> <weight=1>
END SUBNET
```

Besides this propagation subnet, a set of arcs linking that propagation subnet to the basic subnets of element A and element B are added to the model. These arcs are as follows:

```
INPUT_ARC <from_place=B.F> <to_transition=B->A.t1> <weight=1>
INPUT_ARC <from_place A.H> <to_transition=B->A.prop> <weight=1>
OUTPUT_ARC <from_transition=B->A.t1> <to_place=B.F> <weight=1>
OUTPUT_ARC <from_transition=B->A.prop> <to_place=A.E> <weight=1>
```

After this linking the whole model including the basic subnets and the propagation subnet looks like the one shown in Figure 6.12. The propagation subnet basically tries to move the token which might be in place A.H to place A.E, modelling the introduction of an error in element A. The two immediate transitions prop and no_prop of the propagation subnet become enabled only immediately after element B has failed. At that time, a token is put into place B.F, and the failure propagation subnet is activated. Notice that the failure propagation subnet does not remove the token from F. However, the propagation subnet is activated only for each failure of B, because the token in place Used needs to be moved back to place New in order to enable again transition B->A.t1. This re-enabling is performed by transition B->A.restart, whose guard is satisfied only when the element B is repaired (a token is put in place B.H).

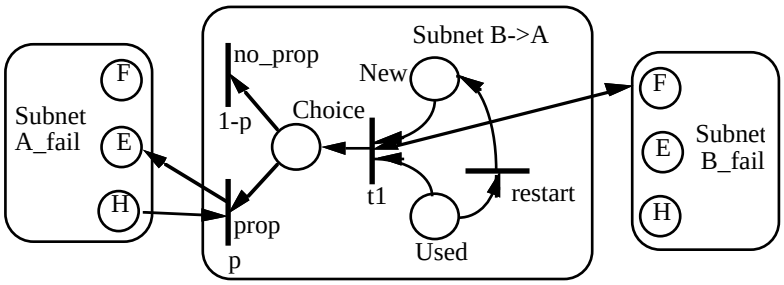


Figure 6.12: Error of A as a result of the failure of B

The propagation subnet shown in Figure 6.11 is to be replicated for each of the type U hyperarc that appears in the intermediate model. Each replicated propagation subnet must be linked to the basic subnets as shown above, except in the case when the element A that is using B is a stateless element. In that case, the propagation subnet must be linked to the basic subnet of A in a way that the token removed from place A.H is moved to place A.F. Indeed, for a stateless component, a failure propagates immediately without generating internal errors. Notice that transition t1 has the highest priority (default value 0). This transition has therefore a priority higher than the two transitions t0 and t1 inside the basic repair subnet of B. As a result of this priority assignment, the token in F is first “sensed” by the failure propagation subnet to trigger the propagation, and then it is removed from F to trigger the repair.

Last, for each of the “uses the service of” relation specified by a U hyperarc, input and output arcs are added to the timed Petri net model, which link the basic repair subnet of the using element to the basic repair subnets of the used element. For instance, suppose a hardware element (either stateful or stateless) is linked by U type hyperarc to the hardware (either stateful or stateless) element B. Then, two pairs of inhibitor arcs with weight -1 are added between the basic repair subnet A_rep and the basic repair subnet B_rep. Figure 6.13 shows those arcs, together with the interface components of the two subnets that are involved in the links. Let us

remind that, as shown in Figure 6.2, a token stays either in place B.P2 or B.P3 until the repair of B is not completed. These arcs impose a synchronisation constraint on the repair activities of element A, blocking them until the used component B does not complete its own repair. Since this synchronisation constraint is imposed for each of the U hyperarcs that link element A to an other element, in the final model timed Petri net model the repair of A is constrained to the repair of all the used elements.

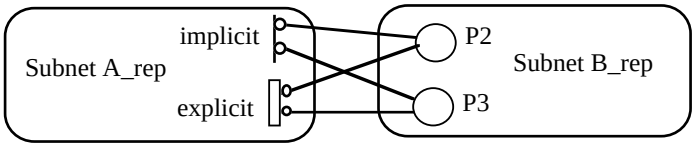


Figure 6.13: Repair of element A conditioned to the repair of a used element B

In the case the used element B is a FTS type element, then the inhibitor arcs link place B_fail.H (contained in the basic failure subnet of B) to the two transitions A_rep.implicit and A_rep.explicit. The case when the using element is a FTS node will be considered later on.

Notice that if a cycle formed by U hyperacs exists in the intermediate model, this constraint on the repair process may potentially lead to a deadlock of the repair subnets. In this case, none of the elements in the cycle is able to perform its repair because everyone is waiting for the completion of the someone else's repair. However, it is worthwhile observing that a situation like that is typically representative of a context in which other repair strategies, more refined than the simple ones we have been considering, needs to be defined. For instance, in a distributed environment, the communication protocols follows specific strategies for deadlock avoidance. If these strategies are modelled into the UML design, then they can be transformed into more refined timed Petri net models, and the deadlock of the timed Petri net model will not occur.

Let us know consider the case of an hyperarc of type C, which links an FTS type of element of the intermediate model to the set of elements that realise a fault-tolerance scheme. The failure propagation inside a fault-tolerance structure can not be modelled with the simple net shown above for the type U hyperarcs, because in this case special provisions have been taken to detect and confinate errors. Rather, the propagation follows a path which is conveniently represented by a fault-tree whose leaves are the failures of the elements involved in the fault-tolerance scheme, and the root is the failure of the whole scheme.

Whenever a type C hyperarc is found in the intermediate model, a fault-tree representing the failure propagation inside the scheme must be found to derive the associate failure propagation subnet. This fault-tree can be provided together with the library of pre-defined classes for fault-tolerance schemes, or can be built with the procedure described in Section 5.4 for a user-defined fault-tolerance scheme. Once the fault-tree is available, translating it into a failure propagation subnet can be performed with an automatic procedure. For instance, consider the simple fault-tree shown in Figure 6.14, which describes how the failure of the three composing elements A, B, and C propagate towards the composed FTS element P. The meaning of the fault-

tree is the following. Each leaf represents an event (in our context a failure event). The events are connected to the gates, which represent the Boolean operators AND and OR. The events propagate according to the Boolean logic: the propagation through an AND gate requires the occurrence of all the incoming events. For instance, the AND gate in Figure 6.14 generates an event of type G if and only if both the events A and B occur. Similarly, the propagation through the OR gate follows the Boolean logic of the OR operator, generating an outcoming event if at least one of the incoming events occurs.

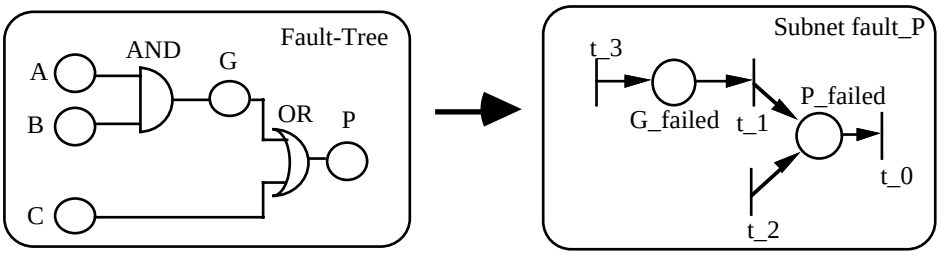


Figure 6.14: Failure propagation subnet corresponding to a fault-tree

The subnet representing the failure propagation process as described by the fault tree is constructed by an algorithm which is similar to the one presented in [15] (but not the same, since the resulting nets are different).

First of all the algorithm generates the following timed Petri net definitions:

```
SUBNET <name=fault_P>
  TRANSITION <name=t_0> <random_variable=instantaneous>
         <memory_policy=?> <guard=TRUE?> <priority=0>
```

Then the algorithm calls the procedure failure_subnet, with arguments the fault-tree and the transition t_0. The procedure looks at the top event of the tree, and if a leaf event is found, the procedure returns. Otherwise, the procedure adds a place which represents a stage of the failure propagation, like P_failed and G_failed in Figure 6.14, and an input arc connecting that place with the transition t_0 previously generated. Then the gate connected to that event is examined. If the gate is an OR gate, as it is in the example, the algorithm generates as many immediate transitions as the number of subtrees connected to the gate. These immediate transitions are connected by output arcs to the place generated for the top event. If the gate is an AND gate, then only one immediate transition is generated, connected to the place for the top event. Then the algorithm is recursively called on each of the subtrees connected to the gate.

A more formal definition of the subnet generation procedure is the following one, where the variable i used to generate the names of transitions is assumed to be a global variable which is incremented by one each time a new transition is added to the failure propagation subnet by the procedure add_to_subnet.

```
PROCEDURE failure_subnet(ft: fault-tree, destination: transition);
BEGIN
  event:=top_event(ft);
```

```
    IF {the event is a leaf of ft}
    THEN return
    ELSE
       BEGIN
          add_to_subnet(PLACE <name=event_failed> <tokens=0>
               <bound=1>);
          add_to_subnet(INPUT_ARC <from_place=event_failed>
               <to_transition=destination> <weight=1>);
          gate:={the gate whose output is event}
          IF (gate=AND)
          THEN
             BEGIN
                add_to_subnet(TRANSITION <name=t_i>
                  <random_variable=instantaneous>
                  <memory_policy=?> <guard=TRUE> <priority=0>);
                add_to_subnet(OUTPUT_ARC <from_transition=t_i>
                     to_place=event_failed> <weight=1>);
                FOR EACH {subtree sft of ft} DO
                  failure_subnet(sft,t_i};
             END
          ELSE
             FOR EACH {subtree sft of ft} DO
                BEGIN
                  add_to_subnet(TRANSITION <name=t_i>
                    <random_variable=instantaneous>
                    <memory_policy=?> <guard=TRUE> <priority=0>);
                  add_to_subnet(OUTPUT_ARC <from_transition=t_i>
                    <to_place=event_failed> <weight=1>);
                  failure_subnet(sft,t_i};
                END;
       END;
END;
```

When the procedure completes, the failure propagation subnet is almost completely defined in the timed Petri net language. The algorithm only needs to add the END SUBNET line.

The so-defined propagation subnet must be linked to the basic subnets of the element composing the fault-tolerance scheme, A, B, and C in the example, and to the basic subnet generated for the element FTS, P in the example, as shown by Figure 6.15:

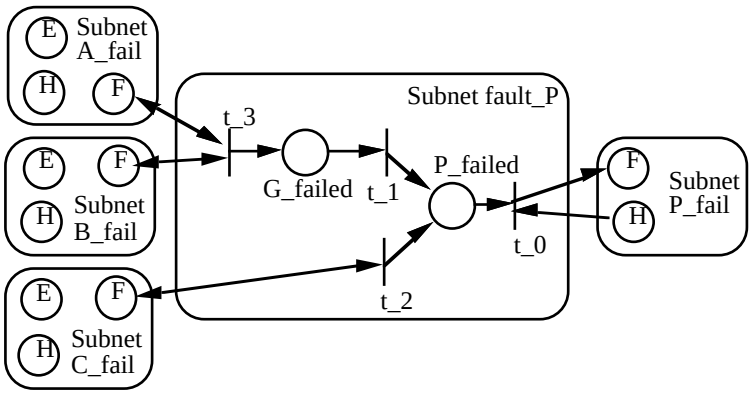


Figure 6.15: Failure propagation subnet connected to the basic subnets of the elements composing a fault-tolerant scheme

The final effect of the propagation is therefore the movement of a token from place P.H to place P.F. The input and output arcs necessary to link the failure propagation subnet to the basic subnets of the composing elements can be conveniently generated by the procedure failure_subnet described above, whenever a leaf event of the fault-tree is found.
The evolution of the failure propagation subnet so defined is not completely defined yet, because its behaviour has to be controlled by its dual counterpart, the repair subnet, which we shall define in the following.

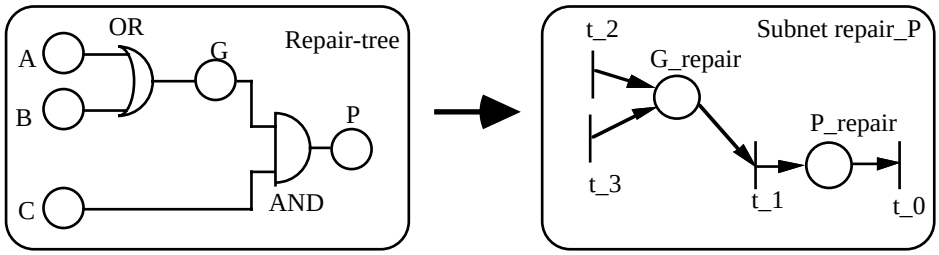


Figure 6.16: Repair propagation subnet corresponding to repair-tree

Starting from the fault-tree of the failure propagation, we have also to generate a subnet for the repair of the fault-tolerant scheme. Indeed, an implicit repair of the scheme takes place as the composing elements get repaired. In this sense, we must model a propagation of the repair from the composing elements to the composed element. To explain the procedure to generate the repair subnet, we again take as an example the fault-tree in Figure 6.14. First of all, the dual fault-tree is defined, by exchanging each AND gate with an OR gate and vice versa, thus obtaining the repair-tree shown in Figure 6.16. Then the algorithm sketched above is applied to this repair-tree, to generate the repair subnet shown in the left side of Figure 6.16. Each of the places of the subnet is assigned a token at the beginning. This subnet is linked to the basic subnets of A, B, C, and P, as shown in Figure 6.17. The repair subnet moves a token from place P.F to P.H, bringing the fault-tolerant scheme in an healthy state. Note that, according to the logic expressed by the repair-tree, not all the elements composing the fault-tolerant scheme need to be repaired for the repair to be propagated to the composed element.

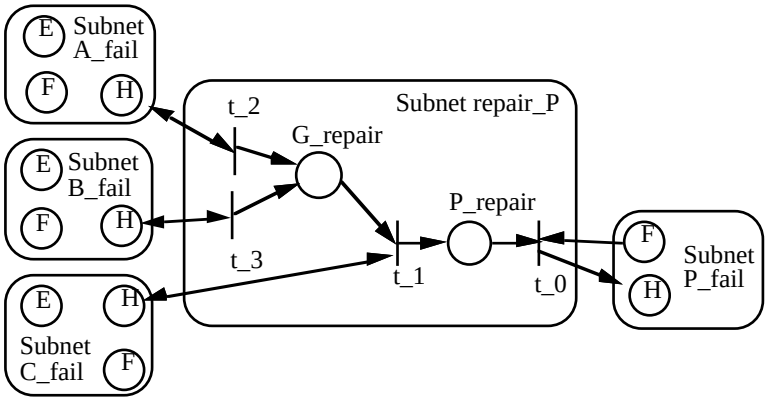


Figure 6.17: Repair propagation subnet connected to the basic subnets of the elements composing a fault-tolerant scheme

Finally, we need to add to the timed Petri net model a set of input arcs which link the failure propagation subnet failure_P to the repair propagation subnet repair_P, and vice versa, in order to completely define their evolution. These input arcs are included in the model according to the following two complementary rules, which we give in an algorithmic form:

```
FOR EACH {place X_failed in failure_P} DO
  FOR EACH{transition t_1,t_2,...,t_n connected to X_failed by an
         output arc} DO
    add_to_model(INPUT_ARC <from_place=repair_P.X_repair>
           <to_transition=t_i>   <weight=1>);


FOR EACH {place Y_repair in repair_P}   DO
  FOR EACH{transition t_1,t_2,...,t_n connected to Y_repair by an
         output arc} DO
    add_to_model(INPUT_ARC <from_place=fail_P.Y_failed>
         <to_transition=t_i> <weight=1>);
```

Among the hyperarcs of type C of the intermediate model, that one linking the element of type SYS with the elements the SYS is composed of, is very particular. Indeed, note that the SYS element does not have an associated fault-tree, therefore the procedure explained above needs an additional input. We associate to the SYS element, as a default value, a very simple fault-tree representing the OR of all the composing elements. Therefore, in the final timed Petri net model, the failure of the "system" is determined by the failure of any of the composing element, and the repair is conditioned to the repair of all the composing elements.

Last, let us consider the constraint to be imposed on the repair of a FTS element named P using the services of element B. In case the used element B is either a HW or SW element (no matter if stateful or stateless), then inhibitor arcs are to be added to the timed Petri net model, which link the repair propagation subnet of the FTS element with the basic repair subnet of B. For instance, suppose B is a hardware element. Then, we add the inhibitor arcs are shown in Figure 6.18, to impose a synchronisation between the FTS element and the used element B. In case the used element B is a FTS element itself, then the arcs are added which link the repair propagation subnet of P to the basic subnet of B, as already explained.

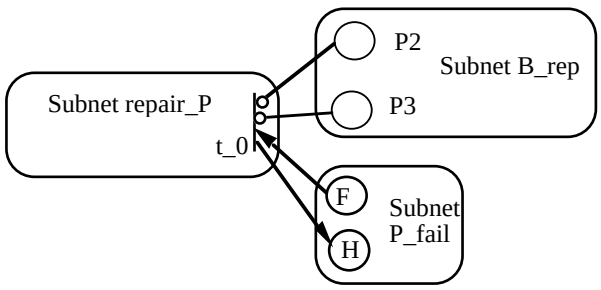


Figure 6.18: Repair of a FTS element conditioned to that of a used element

Conclusions

In this report we have described in an algorithmic form the transformation from structural UML specification to Petri net models for the quantitative evaluation of dependability attributes. Although this description is precise and detailed it has not been formally proven. The reason is the lack of dependability-related semantics of the UML structural diagrams that form the input of our transformation.

To analyse the dependability figures of systems of large size one could ideally build a model of the system accounting for all the details, i.e. the fine grained behaviour of each system component that can be obtained by the behavioural UML diagrams. However, this approach is not viable due to the state explosion and the limitations of existing tools. Therefore the model to build must be of a reduced size where only the features relevant to dependability are captured and all other information is skipped.

Our approach, resorting mainly to the structural views of UML specifications, allows to build at first quite abstract models, maybe too coarse for representing with due precision the real dependability to be expected. However, the modular construction of the model does not prevent, rather favours its extension by offering the possibility to substitute in the model the coarse representation of some elements with a more detailed and precise ones, obtained, maybe later in the design process, by some other transformation or analysis technique. The long term objective is to start with a broad system-wide model and to refine it by plugging in a detailed description of those parts which result to be the critical ones (this selection might be guided by the analysis performed on the coarse model itself).

With this approach, dependability attributes can be analysed depending on the amount of relevant information provided by the designer. In any case the standard sub-models used to build the model of the system, which are those to be used when no specific information is available, can always be substituted by more detailed models derived when information is available.

We defined the syntax of 2 intermediate representations used to divide the entire transformation in sequential steps. The first step takes the UML model and produces an Intermediate model in which the dependability related features are filtered from the entire specification. The second, starting from the “dependability” oriented description provided by the Intermediate model produces a timed Petri net, which is described using still an abstract representation. A final step can then be easily performed to translate the model according to the syntax adopted by specific PN tools selected for performing the analysis.

This approach needs some further work to be refined and to optimise the solution and analysis of the resulting Petri nets.

References

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

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

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

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

[5] Chiola G., “GreatSPN 1.5 software architecture”, in Fifth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, 1991, Torino, Italy.

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

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

[8] Davis J., Scott J., Sztipanovits J. and Karsai G., Integrated analysis environment for high impact systems. 1997, Measurement and Computing Systems Laboratory, Vanderbilt University.

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

[10] Kanoun K., Borrel M., Morteveille T. and Peytavin A., “Modeling the dependability of CAUTRA, a subset of the french air traffic control system”, in IEEE 26-th International Symposium on Fault-Tolerant Computing (FTCS26), 1996, Sendai, Japan: IEEE Computer Society Press.

[11] LAAS-CNRS, SURF-2 User guide. 1994, LAAS-CNRS, Toulouse France.

[12] Laprie J. C., “Dependability-Its Attribues, Impairments and Means”, in Predictably Dependable Computing Systems, Randell B., Laprie J.C., Kopetz H.andLittlewood B., Editor. 1995, Springer-Verlag: pp. 3-24.

[13] Laprie J. C. and Kanoun K., “Software reliability and system reliability”, in Handbook of Software Reliability Engineering, Lyu M.R., Editor. 1996, McGraw-Hill: New York. pp. 27-69.

[14] Lyu M. R., ed. Handbook of Software Reliability Engineering. 1996, McGraw-Hill: New York.

[15] Malhotra M. and Trived K. S., “Dependability modeling using Petri nets” IEEE Transactions on Reliability, 1995. Vol. 44 (3): pp. 428-440.

[16] Nelli M., Bondavalli A. and Simoncini L., “Dependability modelling and analysis of complex control systems: an application to railway interlocking”, in EDCC2, 1996, Taormina, Italy: Springer-Verlag.

[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 * Softeam, Object Constraint Language Specification. 1997, version 1.1.

[18] Sanders W. H., Obal II W. D., Qureshi M. A. and Widjanarko F. K., “The UltraSAN modeling environment” Performance Evaluation, 1995. Vol. 21 (Special Issue “Performance Evaluation Tools”).

[19] Singh H., Billington R. A. and Lee S. Y., “The method of stages for non-Markov models” IEEE Transactions on Reliability, 1977. Vol. 26 (6): pp. 135-137.

From Dynamic UML-Diagrams to Generalized Stochastic Petri Nets

M. Dal Cin, - FAU-IMMD3
G. Huszerl, - FAU-IMMD3 and TUB
K. Kosmidis - FAU-IMMD3

1 Introduction

In this document we first present an informal description of the translation from the diagrams of the Dynamic Model to Generalized Stochastic Petri Nets (GSPNs). The dynamic part of a UML-model comprises sequence diagrams, activity diagrams and statecharts. Collaboration diagrams are equivalent to sequence diagrams and, hence, are not considered here further. We start with the informal description of the translation of sequence and activity diagrams, since their translation is straightforward. We then discuss informally the translation of statecharts. Due to the specific semantics of UML-statecharts we have not yet completely defined the transformation. We rather selected a subset of UML statecharts (Guarded Statecharts) for which the translation is available and which is particularly suited for modeling embedded systems with their environment. This subset comprised also a well defined fault model [Dal Cin 1998]. Thus, it provides the possibility to model and evaluate, for example, the behavior of systems prone to faults in their environment. This is, of course, necessary for any dependability evaluation of critical systems.

We then discuss the transformations more formally. The intuitions behind the transformations will be illustrated by means of very small examples. The demonstrator (Deliverable 5: Software and Report on the Demonstrator) provides a much more elaborate example of the transformations, the modeling technique by means of Guarded Statecharts and of the fault model.

The purpose of the transformation of the Dynamic Model to Stochastic Petri Nets is to be able to provide within HIDE automatically an analytical model which is consistent with the Dynamic Model and which is amenable to a quantitative evaluation. To this end, the Dynamic Model has to be annotated with stochastic parameters. Generalized Stochastic Petri-Nets (GSPN) provide a concise possibility to specify Markov chains. That is, they are equivalent to Markov processes and can be transformed automatically to Markov chains. [Ajmone Marsan et al.]. Moreover, it is possible to extend GSPNs such that non-markovian behavior can be modeled and evaluated as well. An important aspect is that the transformations are open in the sense that they do not presume a specific evaluation tool as long as the modeling capacity of the class of Petri Nets supported by the tool is high

enough. Hence, the most appropriate tool among the many tools available can be selected for our evaluation.

The information gained by the evaluation of the Dynamic Model complements the quantitative evaluation of the structural UML specification. Moreover, input parameters required by the evaluation of the structural UML specification can also be provided by the evaluation of the Dynamic Model.

2 Informal Description of the Transformations

In this section we discuss the transformation on an informal basis and explain our choices.

2.1 Transformation of Sequence Diagrams to Generalized Stochastic Petri-Nets

Sequence diagrams describe the exchange of messages within an interaction of objects arranged in a time sequence. Sequence diagrams may exist in generic form or in instance form. In generic form they describe a set of message exchange sequences among a set of classes; in instance form they describe one actual message exchange sequence consistent with the generic form. The instance form does not include repetition sequences (loops) or conditional sequences (branches). It is mainly this form that is the relevant for the quantitative analysis of model behavior. For example, it may be of interest to compute and compare the cumulative distribution functions of the time to process a blank (in our production cell example) for different patterns of message exchange between the controller tasks. The differences in the exchange of messages may arise from the implementation of different task synchronization mechanisms, such as spin locks or barrier synchronization. Then it may be important to investigate their effect on the throughput of the production cell.

Graubmann et. al. indicate how to transform sequence diagrams into Petri Nets, specifically into Labeled Occurrence Nets. They provide transformation rules for basic sequence diagram constructs and cover also structural concepts like co-regions and sub-diagrams.

Occurrence Nets are cycle free and conflict free Petri Nets which, therefore, suitably describe the actual message exchange sequences within distributed systems. Their labeling is used to relate the elements of the Occurrence Net to elements of the corresponding sequence diagram. For example, labels serve to identify which places of the Occurrence Net correspond to which messages in the sequence diagram. Thus, the labeling establishes the actual semantic link between the UML-diagrams and the Petri Nets [Graubmann et al.].

2.2. Transformation of Activity Diagrams to Stochastic Petri-Nets

Activity charts best describe the concurrent behavior of objects and their interactions. They can be viewed as a combination of statecharts and Petri Nets. Hence, the

transformation of activity diagrams to Generalized Stochastic Petri Nets is straightforward. Actions are represented by places and transitions by timed Petri Net transitions. Some minor details of the transformations are discussed in Section 2.
Recently, activity diagrams became very popular for modeling business processes and workflows, and it is commonly felt, that the quantitative evaluation of the attributes of theses processes is an important issue for further development of modeling environments [Versteegen 1998]. The transformation from activity diagrams to Stochastic Petri Nets can provide the basis for such an analysis.

2.3. Transformation of Statecharts to Generalized Stochastic Petri-Nets

Statecharts (state diagrams) represent finite state machines. They describe the behavior of objects in response to external stimuli, such as sensor signals. In Deliverable 1: 'Specification of Modeling Techniques' we define a subset of statecharts comprising so-called Guarded Statecharts. The main restrictions for Guarded Statecharts are: (a) they are essentially flat, (b) no history states are allowed, (c) transitions may not have an event expression, entry and exit events are allowed, (d) guards are Boolean expressions of state predicates (That is, guards are essentially tests of concurrent states). Stubbed transitions and complex transitions (e.g., forks and joints) are allowed but not considered here. Hence, Guarded Statecharts form a strict subset of UML-statecharts. Nevertheless, Guarded Statecharts are useful for modeling the behavior of a collection of systems of relatively low complexity, for example, for modeling the behavior of several interacting objects of the Dynamic Model. Complex system behaviors have to be modeled by hierarchical statecharts. Moreover, as the systems described grow in complexity and size their analysis should be approached from the structural perspective defined in the chapter 'From Structural UML Diagrams to Timed Petri Nets' of this deliverable.

Guarded Statecharts

Guarded Statecharts (GSC) are suited to model embedded systems with their environment, see Deliverable 1: 'Specification of Modeling Techniques'. That is, embedded systems can often be modeled by the AND-composition of concurrent GSCs. Moreover, with Guarded Statecharts also non-deterministic behavior can be modeled. Although the software of embedded systems is completely deterministic, the system can not know if and when external events or faults will happen. For instance, a task can be requested at any time and peripherals may react to controls with unpredictable delays due to faults.

Such non-deterministic behavior is transformed to a stochastic behavior of the Petri Nets. This requires the specification of state transition rates and branch probabilities. The Petri Net class supported by PANDA allows to directly transform a Guarded Statechart to a Generalized Stochastic Petri Net. State transitions with time delay are transformed to timed Petri Net transitions, those without time delay to immediate Petri Net transitions. Guards become guards of Petri Net transitions.

Hierarchical Statecharts

When dealing with large dynamic models it becomes mandatory to use hierarchy in statechart models. We will handle these models by first transforming them to Extended Hierarchical Automata (see Deliverable 1: Specification of Modeling Techniques). The subautomata of an Extended Hierarchical Automaton are then interpreted as interrelated Markov-processes. They are transformed to Stochastic Petri Nets in a straightforward manner where events are substituted by transition rates.

The specification and implementation of the transformation from hierarchical statecharts to Petri Nets will be considered in future research.

3 Algorithmic Description

In the following we present some details of the transformation of the Dynamic Model to GSPNs. No proofs are given. They will become available in [Dal Cin et a. 1998].

The Generalized Stochastic Petri-Nets we use are 9-tuples (*P, T, In, Out, G, O, p, w, M_0),* where *P* and *T* are disjunct, non-empty sets of places and transitions, respectively. There are two disjunct subsets of *T* : *T_i* and *T_t* where $T\_i \cup T\_t = T$; *T_i* is the set of immediate transitions and *T_t* is the set of timed transitions. $In \subseteq P \times T$ is the set of input arcs and $Out \subseteq T \times P$ is the set of output arcs. G is the set of transition functions:
g_M: $T \rightarrow \{True, False\}$, where *M* corresponds to the actual marking of the Petri Net,
O is the set of arc multiplicity functions *m_M* : $In \rightarrow \{0, 1\}$, where *M* again corresponds to the actual marking of the Petri Net.

Furthermore, *p* is the transition probability function *p*: $T\_i \rightarrow R+$; *w* is the transition rate function *w*: $T\_t \rightarrow R+$ and *M_0* is the function *M_0*: $P \rightarrow \{0,1\}$ of the initial marking.
The 'probabilities' are not normalized, so they can be greater 1 and their sum is not constrained. (For the sake of clarity we call these probabilities weights).

Places are drawn as circles, transitions as bars or boxes, input arcs as arrow-headed arcs from places to transitions, output arcs as arrow-headed arcs from transitions to places. The guard functions are written in square brackets at the given transition, the multiplicity functions are written in round brackets at the given arc, weights and rates are annotations of the given transitions. The actual marked places contain dots.

3.1. Transformation of Sequence Diagrams to Generalized Stochastic Petri-Nets

According to Graubmann, a Labeled Occurrence Net (*B,E,F,l*) is an Occurrence Net (*B,E,F*), which is a cycle free and conflict free Petri Nets. *B* is the set of places, *E* the set of transitions and *F* the cycle free flow relation, with a labeling function

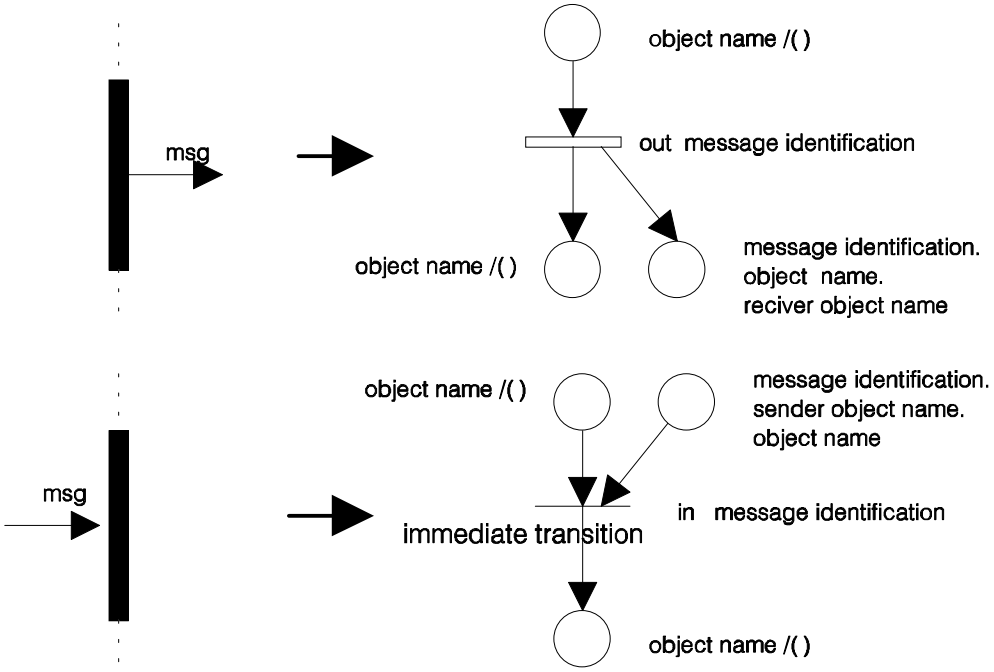
$$l : B \cup E \rightarrow LABLE$$

where *LABEL* is an arbitrary set of strings (labels). The labeling establishes the actual semantic link between the sequence diagrams and the corresponding occurrence net. During transformation these labeling strings are analyzed. For the quantitative analysis we view the Occurrence Nets as subclass of Generalized Stochastic Petri Nets.

Here we only illustrate the transformation algorithm by a few examples. For formal definitions employing a suitable Sequence Diagram Grammar see [Graubmann et al.].

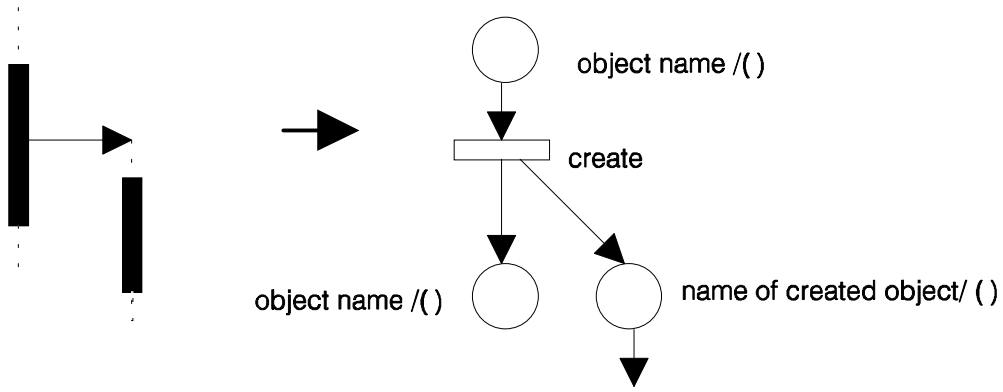
The fragments of Occurrence Nets consist of a transition with one input and one output place. Interaction between live lines is provided by an additional place linked with the transition. The composition process constructs the live lines of the sequence diagram out of such fragments. Here we show the transformations for message input and output and for the creation of objects.

Message input and output:



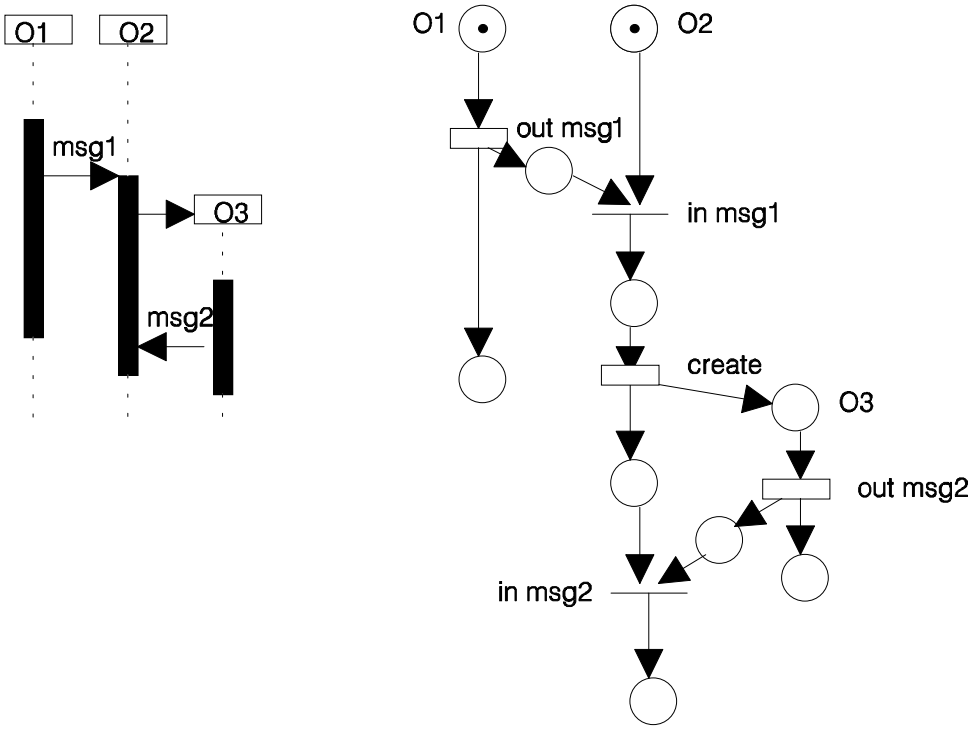
The place label "message identifier, sender object name, object name" is identical to the place label "message identifier, object name, receiver object name", if the message is exchanged between the sender object and the receiver object.

Create live line:



The place label "name of created object" identifies the initial place of the created object.

Small example:



Enhancement: Each message is annotated with a mean send time

3.2. Transformation of Activity Diagrams to Stochastic Petri-Nets

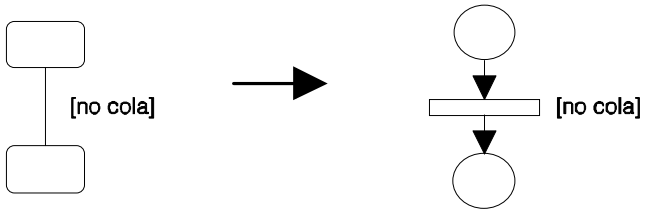
As mentioned, activity diagrams can be viewed as a combination of statecharts and Petri Nets.

Transformation:

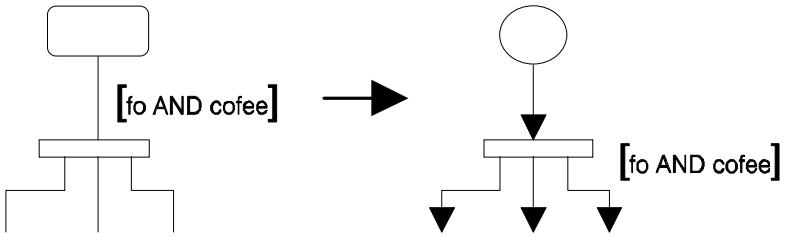
(1) Preparation: (a) Implicit control arcs (due to object flow) are made explicit.
(b) Short hands for decisions are resolved.

(2) An action state models a step in the execution of an algorithm. Each action state is represented as a place. The place of the initial action state gets a token.

(3) Immediate transitions from an action state to an other action state are replace by timed Petri Net transitions and corresponding arcs. The guards of the transitions become the guards of the timed Petri Net transitions.



(4) Each explicit fork and joint transition is represented as timed Petri Net transition. The guard of the transition becomes the guard of the immediate Petri Net transition.



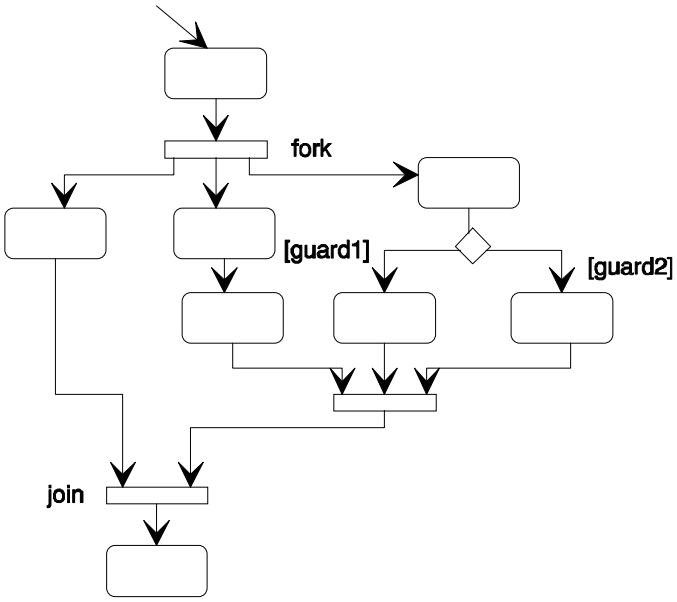
Activities, stop states, object flow and swim lanes are ignored, since they are irrelevant for the quantitative analysis.

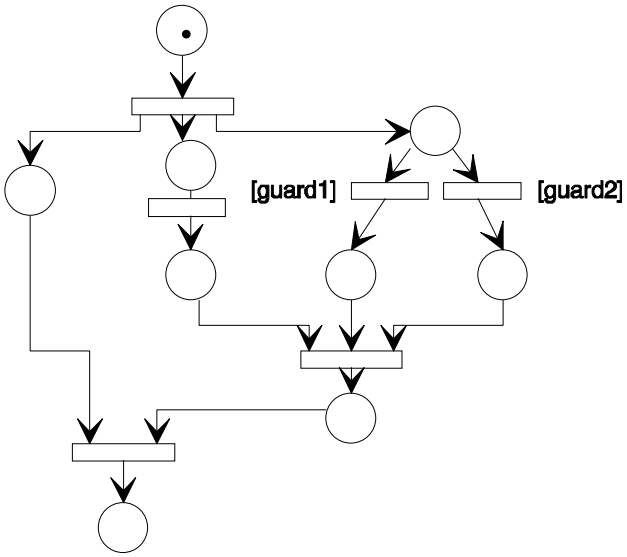
Enhancements:

(a) For each direct transition a firing rate is required. This rate is determined from the mean duration time of the preceding action.

(b) For each fork and joint transition a firing rate is required. This rate is determined from the maximum of the mean duration times of the preceding actions.

Small example:





3.3. Transformation of Guarded Statecharts to Generalized Stochastic Petri-Nets

As mentioned already, Guarded Statecharts (GSC) form a sub-class of statecharts which is suited for modeling embedded systems with their environment. That is, embedded systems can often be modeled by the AND-composition of concurrent GSCs. On the other hand, the Petri Nets supported by PANDA exhibit several properties that make the transformation of Guarded Statecharts to these Stochastic Petri Nets easy. For instance, in PANDA it is possible to annotate transitions with guards and to use state dependent capacities for arcs.

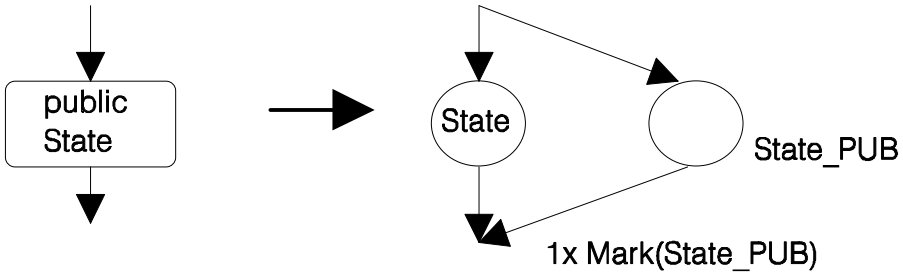
The main objects of an (enhanced) Guarded Statechart are states (container states, base states, initial states) and transitions with guards and labels. The labels describe timing information (arrival distribution of signals) or static information (probabilities of possible outcomes of a given choice).

Guarded Statecharts are not hierarchic - rather, all hierarchy levels (except the bottom level) describe concurrency. The transformation neglects all these concurrent container states, since they have no counterparts in the Petri Net structure. The base states are represented as places. The PN-place holds the name of the GSC-base state. The initial marking of the place is 1, if there is an initial transition in the GSC leading to this state, and its initial marking is 0 else. Since our GSCs model mainly embedded systems, where the communication between several components is of importance, we made it possible by the transformation to model communication errors explicitly within the Petri Nets. Within the statecharts communication errors are modeled by substituting guards by *True* (e.g., the sensor signal is stuck-at-active) or *False* (e.g., the actuator signal is stuck-at-inactive or not observed by the hardware). This way, lost or spurious signals can be modeled. State perturbations are modeled by additional states and/or additional state transitions.

The main transformation steps are:

(1) 'Private' states, e.g., states which do not appear in guard expressions are transformed to places.

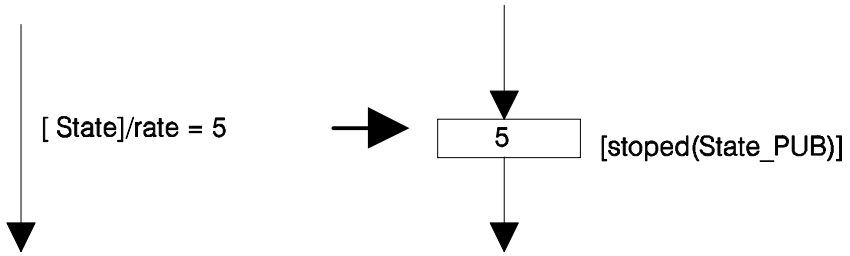
(2) 'Public' states, e.g. sensor and actuator states, are transformed into a pair of places, see figure.



Function *Mark* delivers the number of tokens in State_PUB. The arc annotation *1xMark(..)* determines a state dependent capacity of the arc. The duplication of public states serves

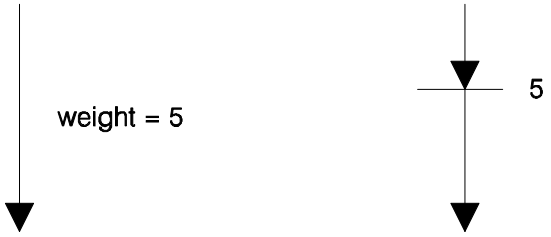
mainly to model communication faults, e.g. lost or spurious sensor or actuator signals. Such a fault occurs when *State* and *State_PUB* have different markings.

(3) State transitions labeled with rates are transformed to timed Petri Net transitions with the same rates, see figure.



Function *stoped* evaluates the predicate *in*(State_PUB) and returns TRUE, if this predicate evaluates to TRUE and all transitions out of state State_PUB are disabled (in conformance with the UML semantics).

(4) State transitions labeled with weights are transformed to immediate Petri Net transitions with the same weight, see figure. The weights of conflicting immediate transitions are normalized by PANDA such that they become branching probabilities.

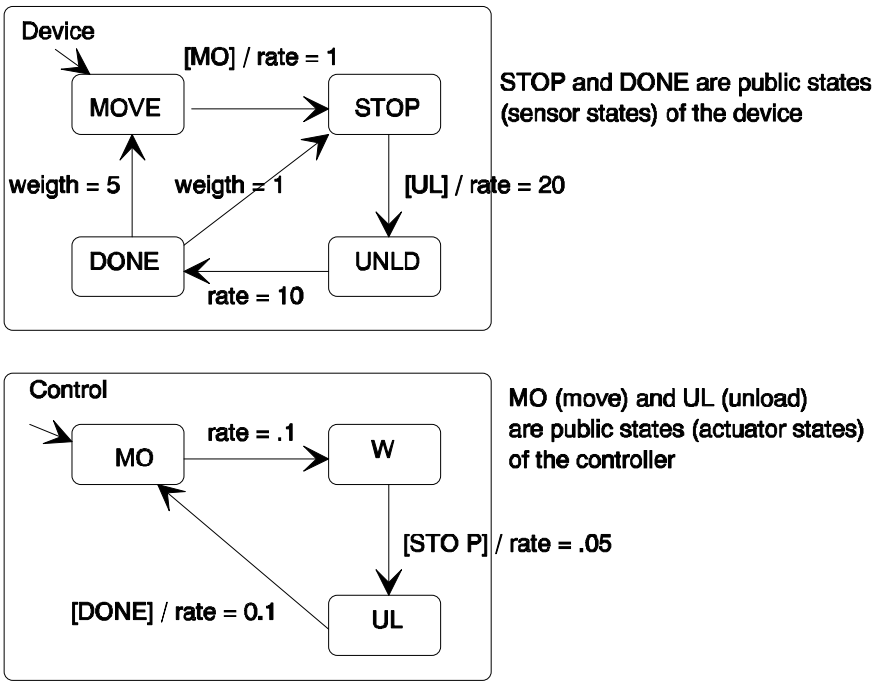


At present, weighted transitions can not have guards as well.

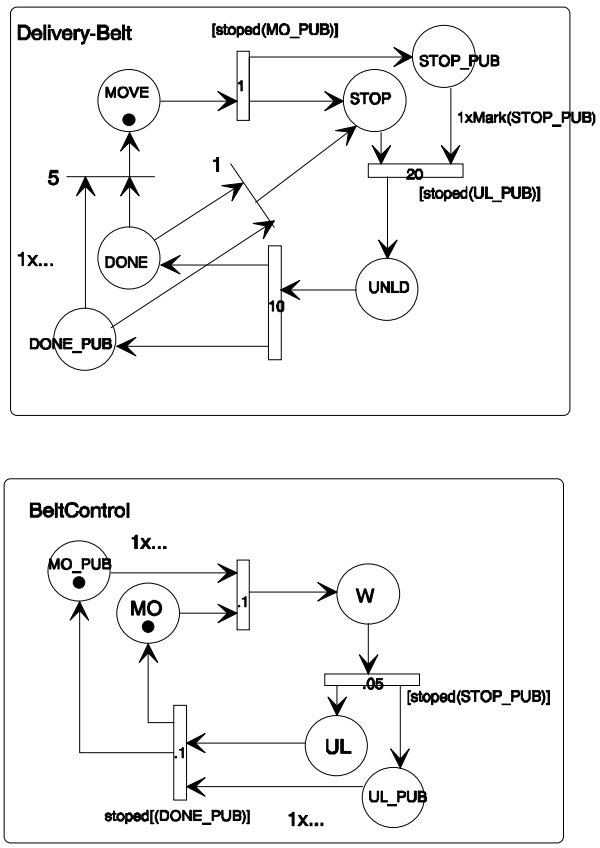
Enhancements:
(1) State transitions with time delay require transition rates.
(1) State transitions without time delay require weights.

Again we show a simple example. This example shows two interacting statecharts exchanging sensor and actuator signals.

The statecharts:



The corresponding Generalized Stochastic Petri Net:



References

M. Ajmone Marsan, G. Balbo, G. Conte, Performance Models of Multiprocessor Systems, The MIT Press 1986

S. Allmaier, S. Dalibor: PANDA -- Petri net ANalysis and Design Assistant, Tools Descriptions, 9th Int. Conference on Modeling Techniques and Tools for Computer Performance Evaluation, St. Malo, 1997

M. Dal Cin, Modeling Fault-Tolerant System Behavior, in Systems: Theory and Practice, Advances in Computing Science Springer , pp. 213-234, 1998

M. Dal Cin, Checking Modification Tolerance, Proceedings of the Third IEEE Interanational High-Assurance Systems Engineering Symposium, HASE 98pp. 4-12, 1998

M. Dal Cin, G. Huszerl, K. Kosmidis, University of Erlangen-Nürnberg
IMMD 3, Quantitative Analysis of Dynamic Models, Technical Report IMMD3 1998

P. Graubmann, E. Rudolph, J. Gabrowski., Towards a Petri Net based semantics definition for message sequence charts, Siemens AG ZFE, Technical Report

G. Versteegen, Geschäftsabläufe objektorientiert optimiert: Fit mit UML, iX 12/1998 pp. 143-147