## Deliverable 1: Modeling

# Specification of Modeling Techniques

## Esprit Project 27439 - HIDE

## High-level *I*ntegrated *D*esign *E*nvironment for Dependability

**A. Bondavalli, A. Borschet, M. Dal Cin, W. Hohl,**

**D. Latella, I. Majzik, M. Massink, I. Mura**

**Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)**

**Consorzio Pisa Ricerche - Pisa Dependable Computing Centre (PDCC)**

**Technical University of Budapest (TUB)**

**MID GmbH**

**HIDE/D1/FAU/1/v3**

## Contents

# 1. Introduction

This document examines certain aspects of the Unified Modeling Language (UML) relevant to the HIDE framework. The examination is necessary in order to provide a sound basis for a translation of UML-models to models amenable for formal and quantitative analysis. On one side, restrictions of the modeling power of UML are to be identified such that precise transformations become feasible. These restrictions will be relaxed in the future. On the other side, model analysis requires certain extensions to the UML, since additional information is needed depending on the kind of analysis someone wants to perform.

We start by giving a short introduction into UML, its purpose, history and future trends (Section 2, contributors: A. Borschet, MID and W. Hohl, FAU). In Section 3 the examination of the UML-modeling paradigm is provided. We divided this examination into two parts: modeling of structures and modeling of behavior. The section on modeling the structures addresses the problems encountered in dependability modeling and analysis of complex systems (contributor: A. Bondavalli, PDCC). The section on modeling behavior deals with the process view that represents the system's concurrency and synchronization mechanisms. This view is captured mainly in Statecharts and Sequence Diagrams (contributor: M. Dal Cin, FAU). For the purpose of formal verification it is necessary to aim at a precise formal semantics of Statecharts. UML does not yet provide such a semantics. Therefore, in this section the informal semantics of UML-Statecharts is summarized and compared to the semantics of Harel-Statecharts. Then a suitable subset of UML-Statecharts exhibiting a precise semantics is identified (contributor: D. Latella, PDCC).

In Section 4 we discuss how to represent requirements relevant to the analysis of quantitative behavior (contributor: M. Dal Cin, FAU) and those relevant to formal verification (contributor: D. Latella, PDCC). Finally, in Section 5 dependability attributes like safety and liveness which can be checked by formal verification are identified, and their representation is discussed (contributor: D. Latella, PDCC). Likewise, in this section also dependability attributes relevant to the quantitative analysis and their representations are identified (contributor: A. Pataricza, TUB).

# 2. Short Introduction into UML

## 2.1. What is the UML?

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. It is the successor of object-oriented analysis and design (OOA&D) methods that were developed by Booch, Jacobson (OOSE) and Rumbaugh. The UML as a modeling language is incorporated by many companies as a standard into their development process. It covers disciplines such as business modeling, requirements management, analysis & design, programming and testing [21].

The UML was developed at Rational Software. The UML has been submitted to the OMG and has been accepted as a standard contributed by several organizations and a large list of companies.

The UML defines a precise language. It benefits future improvements in modeling concepts. Many advanced techniques can be defined using UML as a base. The UML can be extended

without redefining the UML core. It addresses the needs of user and scientific communities, as established by experience with the underlying methods on which it is based.

In future, component-based development will play a more important role to increase the reuse of classes and frameworks. At the moment there are some efforts by Hewlett-Packard and ICON Computing to bring their technology into the UML standard. The UML, in its current form, is expected to be the basis for many tools, including those for visual modeling, simulation, and development environments. As interesting tool integration are developed, implementation standards based on the UML will become increasingly available.

## 2.2. UML architecture

The Unified Modeling Language itself is a set of documents that consists of the UML Semantics, UML Notation Guide, and UML Extensions documents, plus appendices:

### UML Semantics

The UML Semantics document [19] describes a metamodel that specifies the abstract syntax and semantics of UML object modeling concepts. The specification provides complete semantics for all modeling notations described in the UML Notation document. The UML metamodel is a logical model that emphasizes declarative semantics and suppresses implementation details. It is described in a combination of graphic notation, natural language and formal language.

### UML Notation Guide

The UML Notation Guide [20] describes the UML notation and provides examples. It contains brief summaries of the semantics of UML constructs. It is a reference document that explains the notational representation of the semantic concept, various options for presenting the model information and stylistic guidelines such as fonts, naming conventions, arrangement of symbols, etc. that are not explicitly part of the notation. It also shows the mapping of notation elements to metamodel elements.

Graphical diagrams in UML provide multiple perspectives of the system under analysis or development. The UML defines the following diagrams: *use case* diagram, *class* diagram, *statechart* diagram, *activity* diagram, *sequence* diagram, *collaboration* diagram, *component* diagram, *deployment* diagram.

With those diagrams the UML supports three different views to a system:

- the **requirement model**:
  - *Use case* diagrams visualize the relationships between the actors and use cases. They present an outside view of the system.
  - *Sequence* diagrams and collaboration diagrams show the interactions among societies of objects. They describe how use cases are realized. They are interaction diagrams. A sequence diagram displays interactions between objects arranged in a time sequence. A *collaboration* diagram shows object interactions according to their links to other objects.
- the **static structure** to a system: A *class* diagram shows the existence of a class (of objects) and their relationships in the logical view of a system.
- the **dynamic behavior**: A *state transition* diagram presents the life history of a given class, the events that cause a transition from one state to another and the action that result from a

state change. State transition diagrams are usually needed for objects with significant dynamic behavior.

The physical "world" of a system can be described with *component* diagrams, which illustrate the organization and dependencies among software components. Components may be for example a piece of source code, any run time component or an executable.

The *deployment* diagrams show the configuration at run time. It presents the distribution of components across the enterprise.

**UML Extensions**

User-defined extensions of the UML are enabled through the use of stereotypes, tagged values, and constraints. Therefore a UML Extension is a predefined set of Stereotypes, TaggedValues, Constraints, and notation icons that collectively extend and tailor the UML for a specific domain or process. Two extensions are currently defined:

- Objectory Process
- Business Engineering.

# 3. Analysis of the UML-Modeling Paradigm

## 3.1. Modeling of structures

UML has been proposed as a unifying modeling language and is expected to become a de-facto standard for the design of many varieties of systems from small control systems to large and complex open systems.

### 3.1.1. Dependability modeling and analysis

Complex and huge systems consisting of a large number of components including interactions of redundant hardware and software components as well, introduce some problems in dependability modeling and analysis. These problems arise independently of the design methodology applied. Thus, they are present also in systems designed using UML toolkits, and must be addressed from any approach to model such systems. Among these problems the most important to solve is complexity. To master complexity a modeling 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. A feasible approach is to start with simple models and make it step by step 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 of the dependability characteristics for small systems can be obtained by applying a transformation at the fine granularity of the behavioral, dynamic (mainly Statecharts) level of a UML description, which allows to maintain in the model itself other system characteristics like timing aspects and a detailed behavioral description. However, as the systems described grow in size and complexity, this approach is no more viable: the available tool's capacity 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.

Thus, there is a role for approaching the dependability modeling from a structural perspective, building (maybe) coarse models by looking primarily to the static structural views. Such models are built so that only the dependability related features are represented filtering from the mass of information contained in the entire specification. They allow to compare different architectural and design solutions and to select the most suitable one. The sensitivity analysis that can be carried out after modeling 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 automated transformation from UML structural diagrams to timed Petri nets serves in the HIDE framework:

i) to provide a means to analyze dependability attributes of the 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 modeling 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.

ii) to allow a less detailed but system-wide representation of the dependability characteristics of the analysed systems. Such coarse models offer a significant advantage in terms of the complexity necessary to perform such an evaluation. Let us observe that, in principle, to analyze 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 behavior of each system component that can be obtained by the behavioral UML models. Due to the limitations of existing tools however, this approach is not viable, the state explosion prevent it. 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.

iii) 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 behavioral and communication diagrams.

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. 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.

### 3.1.2. The Dependability Model

According to this approach, 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. This overall structure

of the dependability model is shown in Fig. 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.).

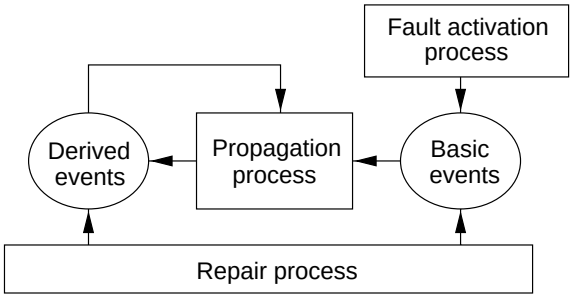


Fig. 1. General parts of a dependability model

The fault activation processes are determined by environmental conditions, and physical or computational properties of the elements of the system. The propagation processes are influenced by the structure of the system (e.g. interactions, redundancy, fault tolerance schemes). The repair processes are determined by the (physical or) computational policy implemented in the system. In our case, the information necessary for developing the dependability model are retrieved from the structural views of the system specification given in UML.

### 3.2. Modeling of behavior

#### 3.2.1. Dynamic models

According to Grady Booch [2] 'the architecture of a software-intensive system can best be modeled by five interlocking views, with each view being a projection into the structure and behavior of the system, and each view focuses on a particular aspect of that system.' These views are: the design (logical) view, the process view, the deployment view, the component view and, last but not least, the use view, Fig. 2.

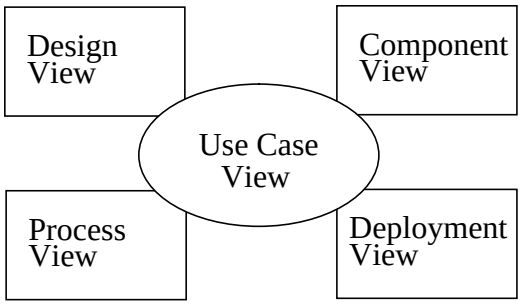


Fig. 2. Views

Section 3.1 deals with the structural view of a model(i.e. with design, deployment and components), whereas the quantitative analysis of the model’s dynamic deals mainly with the process view. 'The process view of a system encompasses the treads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability and throughput of the system' - and its dependability, that is, its reliability

and safety, as we may add. 'With the UML the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the logical view, but with a focus on the active classes that present these threads and processes' [2]. The dynamic aspects of this view are captured in interaction diagrams or sequence diagrams, statecharts and activity diagrams. For short, we refer to the realization of the process view by these diagrams as the *Dynamic Model*.

The automated transformation from the UML dynamic model to Generalized Stochastic Petri Nets (GSPN) serves in the HIDE framework:

i) to provide a means to analyze robustness and fault tolerance of object behavior without dealing with the background mathematical aspects of GSPN modeling and its solution techniques,

ii) to provide likewise a means to derive performance characteristics of object behavior, such as throughput or mean response time,

iii) to provide input parameters for the analysis of the structural UML models.

As mentioned, the dynamic part of a UML-model comprises sequence diagrams, activity diagrams and statecharts.

Within the process view scenarios show typical paths through the state space of the Dynamic Model. (Scenarios are, of course, also instances of use cases). Scenarios are modeled by *sequence diagrams* and collaboration diagrams. Here we will consider sequence diagrams only. Each statechart model gives rise to many scenarios some of which may be performance or reliability critical. The evaluation of these scenarios will deal, for instance, with the probability that a certain scenario can happen, with the mean duration time of a scenario, or with the probability that a scenario does not lead to failure when messages can get lost.

*Statecharts* model the state-driven or reactive behavior of objects. A more detailed introduction to their syntax and semantics is provided in Section 3.2.3. What Statecharts do not show are typical paths through the state space as the system is used, that is, scenarios. The evaluation of object behavior deals, for instance, with the mean duration of a behavioral cycle, the probability for tolerating a state perturbation, or the probability for object failure. However, considering event sequences and event hierarchies one runs into serious problems, if one wants to interpret the modeled object behaviors as Markov processes for quantitative analysis.

*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, activity charts can be viewed as a subclass of statecharts. Here we introduce another subclass, so-called *Guarded Statecharts*.

Statecharts are widely used to model the dynamic behavior of concurrent embedded systems. Also high level Petri nets are used for this purpose [16]. In attacking the modeling of the behavior of embedded systems such as the production cell of the demonstrator, a trade-off has to be made between the degree of detail in modeling the possible behaviors of the system and the degree of automation of the analysis process [5]. This necessity leads us to define a sub-class of Statechart comprising so-called Guarded Statecharts.

**Guarded Statecharts**: A Guarded StateChart (GSC) is a finite set A of actions with guards and a finite set S of states one of which is the i-state (initial state) of the GSC. Each state and guard event has a name. Actions denote state transition. When state transitions are depicted graphically, they are labelled with labels of the form [guard], where 'guard' is a name of a guard.

- *Guards* are boolean expressions of the predicates *in(<state>)* where *in(<state>)* evaluates to true, if *<state>* is the (actual) i-state of the GSC, i.e. an element of S or of some concurrent GSC.

- *Actions* are of the form:

<guard>*<set_of_states>

where <set_of_states> is a subset of S.

For instance, an action of a feedbelt (see Deliverable 5) could be:

*in*(Table.down) *AND* (*in(*Feed_Belt.move) OR in (Feed_Belt.Stop)*(Feed_Belt-stop)

or, graphically (Fig. 3.):


Fig. 3. Action

The <guard> is a guard expression. We restrict guards of an action by stipulating that, if a guard contains more than one state of S, the predicates of these states are OR-connected. The action is such that it is executed atomically and instantaneously, if its guard evaluates to true. The effect of the execution is that exactly one state of <set_of_states> is chosen non-deterministically as next i-state of the GSC.

A guard expression of GSC may not contain predicates of states of S. If such a guard evaluates to true, the GSC takes one of the target states irrespectively of its actual i-state. That is, the OR-connection of all state predicates of S always evaluates to true. Outputs are considered to be part of the state in which they occur. Guards can be considered as high-level abstractions of synchronization mechanisms.

Using Guarded Statecharts we can abstract continuous signals to discrete signals assuming a finite set of critical values. For example, 'it is only important to observe whether a robot arm is directed in a position allowing for unloading the rotary table, or pointing toward the press; all intermediate positions can be collapsed into a single third value' [5]. This way, we model sensors and actuators via states. A state representing an actuator being active means that the actuator is set; analogously if a component is in a state which represents a sensor, it means that this sensor is set. Moreover, in Guarded Statecharts hardware and software components are only allowed to communicate via such sensor and actuator states. This interaction is expressed via guard expressions containing predicates over sensor or actuator states (public states). Likewise, interactions between tasks of the control software are also modeled by guarded state transitions: this corresponds to an asynchronous synchronization pattern between tasks. This pattern is inherently multi-threaded, because it models a message being passed to another object without the yielding of control [8].

### 3.2.2. Remarks on fault modeling in object behavior

Considering the quantitative analysis of the behavior of embedded systems, it is necessary to model the environment (device models) as well as the control software and their interactions via sensors and actuators (closed loop modeling), since the environment can be the source of faults which can give rise to errors in the execution of the control software.

For a dependability analysis, faults, errors and failures are modeled by message losses, loss of synchronization, erroneous states, and erroneous state transitions. Many of these faults and errors can be modeled by so-called state perturbations. State perturbations include distinguished states corresponding to degraded performance of the modeled system, paths leading to those states, erroneous state transitions, trigger events due to external faults giving rise to erroneous state transitions and the use of guards to express fault-tree like failure conditions. Thus, a wide spectrum of possible errors can be modeled by state perturbations. State perturbations are adequate modeling techniques, particularly when dealing with GSCs.

The fault/error-model for Guarded Statecharts is based on the notion of state perturbations. For example, unintended state transitions are such state perturbations. An unintended transition from state *s* to state *q* may be due to a permanent or temporary fault and *q* may be an erroneous state [4], [15]. An unintended state transition due to a temporary fault occurs at most once in the considered period. An unintended state transition caused by a permanent fault can occur whenever the system is in the state that gives rise to the erroneous transition.

Any binary and reflexive relation over state space S of a Guarded Statechart GSC is called an error of SC. An error is called temporary, if it is “effective only once”; that is, it does not change the transitions of SC. If the error is due to a permanent fault, it changes the transition of the GSC. Then also the error is called permanent. This modified GSC is non-deterministic.

An other type of state perturbations arises, if guards of a Guarded Statecharts are not observed. For example, the guard *in*(*Table.down*) may not be observed by the feed belt; that is, this guard always evaluates to true. This way, sensor and actuator faults can easily be modeled.

Finally, using guards fault trees over component states can be integrated into statecharts. As an example see Fig. 4. (Robot R with two arms feeding two presses).


Fig. 4. Action with Fault Tree

From these Guarded Statecharts certain scenarios can be derived, and, for example, faults modeled as non-observance a guard are represented as lost or spurious messages in sequence diagrams.

### 3.2.3. Introduction to UML statechart diagrams

In the following, the syntax (model elements) and the concepts of the (informal) semantics of UML statecharts are summarized. The detailed description can be found in [19] and [20]. At the end of this section, we introduce the restrictions we adopt in the context of formal verification of object behavior.

**Elements of the notation**

UML statecharts is an (object-oriented) variant of classical Harel statecharts [12]. The statecharts formalism itself is an extension of traditional state transition diagrams including the following additional concepts:

- State hierarchy and concurrency. A state is called a *composite* state if it contains one or more substates. A composite state can be decomposed into mutually exclusive disjoint substates (using OR relationship) or into orthogonal substates (using AND relationship). In the latter case, the composite state is *concurrent* and its direct substates are called *regions*. Regions must be refined by OR relationship. Note that each substate is uniquely owned by its superstate. The statechart diagram itself is the hierarchical decomposition of a single composite top state.

- Compound transitions. A *simple transition* indicates that the system may change its state and perform a *sequence of actions* when a specified event occurs and a specified *guard* condition is satisfied. *Compound transitions* have multiple segments separated by pseudostates. *Join segments* originate in multiple states (representing synchronization), *fork segments* are connected to multiple states (representing a splitting of control). *Branch segments* labeled with guards represent different possible paths depending on conditions.

- Interlevel transitions. Transitions that cross the boundaries of composite states can be used to represent an arbitrary state change in the statechart.

- Transitions from/to composite states and history states. A transition drawn to a boundary of a composite state is meant as a transition to its initial substate or to the initial substates of its regions. A transition drawn from a boundary of a composite state means that all of its active substates are exited when the transition is taken (fires). From the semantics point of view, if a transition enters a region of a concurrent state, then the other regions are also entered (explicitly by fork segments or by default entering their initial states). Similarly, if a transition exits a region of a concurrent state then all of the other regions are also exited.
  A transition drawn to a history state is equivalent to a transition to the last active direct substate of the composite state in which the history state resides. Firing of a transition drawn to a deep history state causes the last active substates of the composite state be entered recursively.

- Enriched set of events and actions. Events may have parameters. Actions are distinguished as call, return, send, terminate, create and destroy actions, according to the (object-oriented) software context.

An example is presented in Fig. 5. The statechart is a refinement of $s0$, the top state, where $s1$ is a concurrent state with regions $s4$ and $s5$. Transitions triggered by events $r1$,$r2$ and $a2$ are interlevel ones.

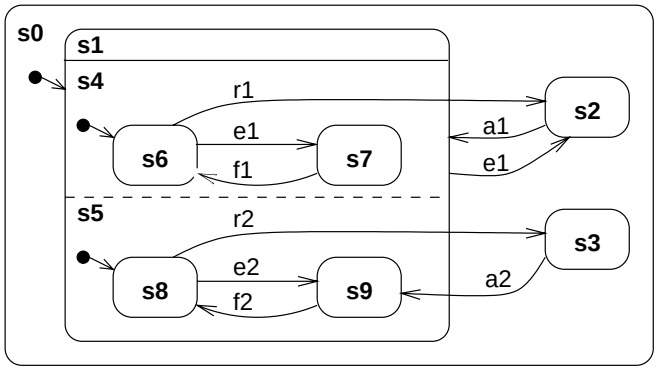


Fig. 5. Example of an UML statechart

**Core concepts of the semantics**

The semantics of UML statecharts are close to that defined for the statechart variant implemented in the tool STATEMATE [13]. However, the adaptation to object-oriented design required changes [14]. The simplifications and modifications introduced in UML can be summarized as follows:

- *Single event processing*. The hypothetical state machine which implements the statechart diagram processes event instances that are selected by an (unspecified) dispatcher from an event queue. Events are processed one after the other. Accordingly, transitions (including compound transitions) are triggered by at most one event.

- *Run-to-completion processing*. An event stimulates a *run-to-completion step*. Transitions that fire have to be fully executed and the state machine has to reach a stable state configuration before it can respond to the next event.

- *Priority concept*. Transitions are in conflict when the intersection of the sets of states they exit is non-empty. Some conflicts can be resolved by using priorities. A transition has higher priority than another transition if its source state is a substate of the source of the other one. If the conflicting transitions are not related hierarchically then there is no priority defined between them, and the conflict is resolved by selecting one of the transitions non-deterministically. Priority of a join transition is according to the priority of its lowest level source state. Notice that the definition of the priority of join transitions in actually not a good definition since in some cases the “lowest level source state” does not exist.

- *Execution step*. The set of transitions that will fire is a maximal set of enabled transitions (i.e. they are triggered by the current event, their guards are satisfied and their source states are active), without conflicting transitions, and such that no enabled transition outside the set has higher priority than a transition in the set. The order in which the transitions in the set fire is not defined. itemize

**A subset of UML statecharts**

During the first phase of the HIDE Project we considered a strict subset of UML statechart diagrams containing though all the interesting conceptual issues related to concurrency in the dy-

namic behavior, like sequentialization, non-determinism and parallelism. Some of the restrictions we imposed can be easily relaxed in the future. These restrictions are in line with formal verification of Statecharts (Restrictions in line with their quantitative analysis are defined in Section 3.2.1.) 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.

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.

# 4. Specification of Requirements

## 4.1. Specification of requirements relevant to qualitative evaluation

In the first phase of the HIDE project, the representation of model behavior by Markov processes is considered as the major analysis technique to be used. These processes can best be specified by Stochastic Petri nets.The designer tool has to produce a Petri net model. A Petri net tool transforms this model to a Markov process and computes state probabilities. By examining these probabilities one can check whether the model satisfies certain performance and/or dependability requirements. These requirements are specified by reward functions and their expected value domain.

An advanced way to specify requirements and to filter information is to use rewards. To this end, a language for specifying result functions and reward measures has been designed and implemented user friendly; it allows the modeler to express the reward measure of interest to be computed in an application oriented syntax. An example is given below.

Therefore, following the outlines from [11], a language for the formulation of reward measures together with an automated computation of these measures from Markov analysis results has been integrated into the tool PANDA [1], [6].

The specification of reward measures is a two-tier process:

- First, states or transitions in the Petri net which represent aspects of interest for the stochastic analysis are selected as *reward functions*. Several reward functions can be assigned to a model. The body of the measures is constructed from basic *result measures*, which in turn are built by applying arithmetic and boolean operations on *characterizing functions*, where the latter access the reachability graph elements.

  Examples:

  - The simplest characterizing functions for rate rewards are
    `mark(placename)` and `enabled(tranition)`

    evaluating the number of tokens in a place, respectively the state of a transition.

  - Likewise for impulse rewards, the characterizing functions
    `fire(transition)` and `rate(transition)`

  An example for the use of such simple result functions together with arithmetic operations is

```
reward_func1(
    (mark(place1) + mark(place5)) * enabled(trans1)
)
```

  returning the number of tokens in place 1 and place 5 if transition 1 is enabled (and 0 otherwise).

  Far more complex result functions can be formulated, including more elaborated characterizing functions or even nested if-then-else clauses for sophisticated conditional queries of the reachability graph. An example would be:

```
reward_func2(
 IF (enabled(trans1) AND (mark(place5) < mark(place8)) THEN (
          summark
     ) ELSE (
```

```
            IF (minmark < 3) THEN (
                probfire(transition3)
            ) ELSE (
                probfire(transition2)
            )
        )
    )
```

evaluating to either the normalized firing probability of transition 3 or 2 (depending of whether there are at least 3 tokens in the net) or the sum of all tokens, under the condition that transition 1 is enabled and there are fewer tokens in place5 than in place 8.

Result functions are tagged with identifiers, and variables can be assigned to complicated expressions or to terms which are to be reused. Nesting of result functions is also possible for ease of use.

- In the second step, stochastic operators can be evaluated for each defined result measure. In the current implementation, this can be the standard functions expected value and variance; more complicated stochastic functions can be built by defining ``result functions'' which consist of arithmetic operations of standard operators.

  Classical GSPN analysis results (like expected number of tokens in a place, throughput of a transition etc.) have been predefined as result functions for convenience.

  Examples:

  ```
  result_func1(
      E(reward_func1, NULL) + ENTOKEN(place5)
  )
  ```

  computes the expected value of the reward function defined above and adds the average number of tokens in place 5.

  The covariance of two impulse reward functions can be computed as

  ```
  covariance(
      E(NULL, product_impulse) - E(NULL, factor1) * E(NULL, factor2)
  )
  ```

  with the definition

  ```
  product_impulse(
      factor1 * factor2
  )
  ```

The goal of the stochastic analysis of Petri nets is to get answers for model-specific questions; however, the results obtained from the solution of the Markov process described by the Petri net state space are not directly related to the input net. The computed results have to be filtered in a suitable way as to gain the results of interest.

### 4.2. Specification of requirements relevant to verification

In the first phase of the HIDE project, model checking is considered as the major verification technique to be used. In such a technique the system designer produces a model of the behavior 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 behavior 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 (LTL in the sequel).

An informal description of the logics is given below. Examples of its use are given in Deliverable 2.

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,...,sn)* meaning that the current configuration contains *all* the states listed in the *in* predicate. Moreover, the notation [*e1,..., en*] will be used for denoting the fact that the events in the current event queue are *e1,..., en,* where *e1* is the first element in the queue and *en* is the last one (here a FIFO discipline is assumed)[1].

Predicates of the form *in(s)* and [*e1,..., en*] form the *atomic formulas*. The syntax of a generic formula is given by the following grammar:

```
LTL_formula ::= atomic_formula

        | LTL_formula AND LTL_formula

        | LTL_formula OR LTL_formula

        | NOT LTL_formula

        | [] LTL_formula

        | <> LTL_formula

        | LTL_formula U LTL_formula
```

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

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

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

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 could be rewritten as []*f*, or as [](p==>q) where:

p=[*e*10,*e*25,*e*0]

q=*in(s1, s2)*

f=p==>q

# 5. Identification of Dependability Attributes and their Representation in UML

## 5.1. Identification of dependability attributes for qualitative evaluation

The following attributes of dependability are defined in [17]: *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 unauthorized access. In this document reliability, availability and safety modeling will be considered.

### 5.1.1. Dependability parameters and system structure

“A dependability model of an integrated fault tolerant system must include at least three different factors: computation errors, system structure and coverage modeling” [9]. In the following we concentrate on the first two, by investigating how system structure and (failure and related) parameters of system elements can be captured in dependability models.

### 5.1.2. Reliability analysis

Reliability of a system is a measure of the time to failure. Reliability analysis investigates how failures of system elements lead to a failure of the system.

Both hardware and software elements are characterized by failure parameters as follows:

- *Independent failures.* Independent failures are caused by unrelated faults of the elements. Hardware elements are affected by physical faults and design faults, software elements are affected by design faults.
- *Common mode failures.* Specification mistakes or dependencies in the design and implementation cause related faults. Related faults manifest themselves as similar errors which lead to common mode failures. Note that independent faults or distinct errors may also lead to common mode failures.

A failure is characterized by a probabilistic parameter which expresses whether a corresponding fault leads to the failure during execution or not. In this way, error handling and fault treatment at the level of individual components is taken into account. The kind of parameter depends on the execution model of the system. In the case of continuous service, failures are characterized by a *failure rate* (i.e. arrival rate of active faults) while in the case of demand-driven service, *failure probability* (i.e. a probability that a fault is activated during the execution) is used.

If the state of the element is taken into account, i.e. the causes of failures are to be modeled, then the notion of *error* is introduced. Faults are resulting in erroneous state of the element,

which may lead to failure of the element. This process is characterized by *fault activation rate* and *error latency.*

The key point in reliability modeling is the identification of *redundant elements* and their use to tolerate failures of (other) system elements. To represent the fault tolerance in the dependability model the understanding of the following points is necessary:

- Identification of redundant elements in the system. Redundant elements are present in the form of replicas or variants.
- Identification of the dynamics of the redundancy scheme, i.e. how the redundant elements are used.

The tolerance of faults is characterized by a *coverage factor,* which is a probability that the given fault (a failure of a system element) will be tolerated by the system, i.e. the system provides its services in spite of the fault (maybe in a degraded mode).

The restoration of failed system elements during the normal operation of the system is discussed and parameterized below.

**Safety analysis**

Safety of a system is a measure of the time to catastrophic failure. In general, catastrophic failures are identified by the designer, depending on the application task and environment of the system. Detection of a failure of a system element is a prerequisite to handle it in a safe way by the system. Thus, detected failures are often termed as benign failures, undetected failures are termed as catastrophic failures. This way the major difference between a reliability and a safety model is the further categorization of the system failure.

Error detection capability can be captured at the level of individual elements as well as at the level of the system. A proportion of failures is detected by the element itself, while other failures can be detected only by external system elements (e.g. a monitor).

Individual system elements are characterized by a *local detection coverage,* which is a probability that a failure of the element is detected by itself.

Focusing on the system structure, those redundant elements are considered which are assigned to other system elements to detect their failure (often without the ability to tolerate it). These redundant elements are characterized by a *(global) detection coverage,* which is a probability that a failure of the assigned element (which is not detected locally) is detected, but not tolerated.

**Availability analysis**

Availability of a system is the probability that the system provides a correct service. The long-term ability of the system to recover from failures is taken into account.

Detected failures of system components trigger fault treatment. The fault is judged to be permanent or transient.

Theoretically, from the point of view of temporal persistence, faults can be categorized as being either permanent (presence is not related to internal or external pointwise conditions) or temporary ones (present for a limited amount of time). Temporary faults are called transient or intermittent faults, according to whether their conditions are defined by the external environment or some internal system activity. Practically, faults are declared to be hard (requiring passivation) or soft faults (with negligible probability of recurrence, therefore no need of passivation) by a decision mechanism [3]. A hard fault may be due to a permanent fault, a dangerously fre-

quent intermittent fault or a transient fault. Soft faults are due to intermittent or transient faults. A more precise diagnosis may further categorize hard faults to be permanent or transient.

If a hard fault is diagnosed to be permanent then the system is degraded and an explicit *repair* action is required to restore the original state. If it is declared as soft or diagnosed to be as transient then the system element may be *reintegrated* by some mechanism. Repair and reintegration will be referred to as maintenance in the following.

Maintenance can be characterized a follows:

- The *proportion of faults* of an element that are diagnosed to be permanent.
- The *repair rate* of the element in the case of permanent faults.
- The *reintegration rate* of an element in the case of faults diagnosed to be *not* permanent (i.e soft faults and hard faults diagnosed to be transient).
- The key point related to the system structure in availability modeling is the identification of *maintenance dependency* among elements, i.e. shared repair facilities.

The proposed model parameters and distinguished elements of the system structure are summarized in Table 1.

| Model type | Parameters | Distinguished elements |
|---|---|---|
| Reliability | Separate failure rate<br>Common mode failure rate<br>Fault activation rate<br>Error latency | Redundant elements (variant, redundancy manager) |
| Availability | Proportion of permanent faults<br>Repair (reintegration) rate | |
| Safety | Error detection coverage (global or local) | Error detector |

Table 1. Model parameters and distinguished elements of the structure

### 5.1.3. Identifying system structure

The granularity level of redundancy in object oriented systems can be described as operation, object or class level [22]. In practice, the class level redundancy is the most widely used for implementing fault tolerance schemes. The following types of implementations are well-known:

- **Predefined (library-based) class hierarchy:**
  Predefined system classes are responsible for the implementation of a given solution. Non-functional characteristics (persistence, recovery, fault tolerance) are inherited from these system classes [7]. Abstract classes may be defined for *variants*, *adjudicator* and *redundancy manager.*
- **Reflective languages and metaobject protocols:**
  A reflective language can manipulate a representation of its own behavior, which is called the system's meta-level [18]. In object oriented languages, the execution of an object is controlled by a meta-object: that represent both the structural and computational aspects of the original one. The use of reflection helps to improve the transparency of fault tolerance

mechanisms, the mixing of functional and non-functional programming is eliminated, the programmer does not have to be aware of which fault tolerance mechanism is used and how to use it.

- **Pattern-based design**:
  Design patterns are widely used by practitioner engineers to solve common problems which arise at the level of the system. Basic fault tolerant structures can be provided as general, reusable design patterns.

Since the reflective approach depends heavily on the implementation language, we focus on the approach using a (predefined) class hierarchy.

In class level redundancy schemes the identification of the scheme requires the identification of the distinguished components (e.g. variants, adjudicator, redundancy manager) and identification of the dynamics of the scheme. Structure level diagrams do not contain enough information to identify the dynamics, this way either behavioral level modeling or the identification of predefined behavior is required:

- In most of the cases, the dynamics of the scheme can be assigned to a distinguished object, the redundancy manager, which is responsible to coordinate the execution of the other objects. It can be characterized as follows:
  - If the redundancy manager implements a predefined scheme then it can be stereotyped to indicate that scheme. This way the dynamics can be estimated automatically, based on a library of predefined schemes and the corresponding managers. The behavioral level modeling is not necessary, the designer can select the redundancy manager of a predefined scheme fitting to the needs of the application.
  - In a more general way (covering also cases when the redundancy manager implements a special scheme which is not available in the library of predefined ones) the designer should describe the redundancy manager by behavioral level (activity or statechart) diagrams. This way, in order to identify the dynamics of the scheme, the behavioral view has to be analyzed. Note that the behavioral view of the other elements of the scheme is not necessary (corresponding to the approach of partially refined models where only critical parts of the system should be refined).
- In some cases the dynamics of the scheme can not be assigned to a single object (e.g. in a distributed scheme like NSCP in which there is no distinguished controller responsible to coordinate the other objects). However, the set of objects implementing the scheme can be characterized in a similar way as in the previous case.

The above approaches can be unified to a methodology which uses library based classes for redundancy managers and objects of predefined schemes and also enables the behavioral level description of special schemes not available in the library. The designer can decide to use one of the schemes available in the library or describe the detailed behavior of objects.

### 5.2. Identification of dependability attributes for formal verification

In the context of formal verification, important properties are those of *safety, liveness* and *precedence*. A very convenient way for expressing these properties is by means of Temporal Logics formulas. It is outside the scope of this deliverable to discuss in detail these classes of properties and how they can be expressed in Temporal Logics. The interested reader is referred to the valuable tutorial [10].

In the following we shall give some examples of typical safety, liveness and precedence properties and their representation in linear time temporal logics. To our knowledge, there is currently no direct way for expressing Temporal Logics formulas as requirements within the UML. We refer to Sect. 4 of this deliverable for a short introduction to the (limited) temporal logics considered during the first phase of HIDE. For UML related terminology we refer to [19].

**Safety Properties**

Informally, a safety property states that “nothing bad happens” with the system.

A typical safety property is *partial correctness* which states that if the initial status of the system satisfies a certain precondition $\phi_p re$ then if the system reaches a final status, it will satisfy a certain postcondition $\phi_p ost$. Assuming that formula I characterizes the initial status and formula F characterizes a final status, the above property can be written as

$$(\mathrm{I}\ AND\ \phi_p re) \Longrightarrow ([](F \Longrightarrow \phi_p ost))$$

Another common safety property is *mutual* **exclusion**. Supposing that two subsystems both access a shared resource and that formula $CS_j$ characterizes those statuses where subsystem $j$ is in the critical section, for $j$=1, 2, the property can be formalized as

$$[]NOT(CS_1\ AND\ CS_2)$$

**Liveness Properties**

Informally, a liveness property stipulates that “eventually something good will happen”.

A typical liveness property is *total correctness* which states that if the initial status of the system satisfies $\phi_p re$ then the system will eventually reach a a final status which will satisfy formula $\phi_p ost$. Assuming again that formula *I* characterizes the initial status and formula *F* characterizes a final status, the above property can be written as

$$(I\ AND\ \phi_p re) \Longrightarrow \langle\rangle(F\ AND\ \phi_p ost)$$

Another useful liveness property is *responsiveness*. Suppose predicate *Req* characterizes those statuses where a certain request is issued and *Grant* characterizes those statuses where the request is granted. Then the fact that whenever a request is made eventually a response is given can be formalized as

$$[](Req \Longrightarrow \langle\rangle\ Grant)$$

**Precedence Properties**

Precedence properties ensure a certain ordering of events or situations. They are of the form

$$NOT((NOT\ p)\ U\ q)$$

and the intended meaning is that the first occurrence of a status satisfying *q*, if any, must be

preceded by an occurrence of a status satisfying *p*. One can think of *p* as a formula expressing for instance a request and *q* denoting a response.

### 5.3. Representation of dependability attributes in UML

An UML specification does not cover all non-functional aspects required for dependability modeling. The specification should be extended in order to be able to construct the dependabil-

ity model. UML provides the following facilities to introduce such extensions assigned 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. E.g. coverage=78%
- *Constraints.* Constraints are (Boolean) expressions given e.g. in the Object Constraint Language OCL. Note that constraints can be applied also to the system structure, since the constraint language provides mechanisms to browse the structure of model elements.
- *Stereotypes.* Stereotypes introduce a new class of modeling elements introduced at modeling 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 generalizable, i.e. subtypes and hierarchy can be defined.
- *Comments.* Comments are arbitrary, unstructured annotations.

Extensions can be used to identify specific structure (e.g. redundancy structures) or to assign dependability related parameters.

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

### 5.3.1. Identifying redundancy structures

If a class-based redundancy approach is adopted, then elements and dynamics of the redundancy structure have to be identified.

- In the case of a library-based scheme, the name (ID) of the scheme (as stored in the library) has to be given.
- In the case of a scheme specified by the designer, the roles of system elements (redundancy manager, variant, adjudicator) have to be identified. Moreover, since the dynamics of the redundancy scheme has to be derived based on the behavioral description of the redundancy manager, some elements of its statechart has to be distinguished. States and events are stereotyped as follows:
  - Failure state: the redundancy manager detects that the set of objects in the scheme can not provide the service.
  - Failure event: the redundancy manager notifies the client(s) that the service of the scheme is not available.
  - Response event: the redundancy manager provides the service of the scheme to the client(s).

  For the sake of the automatic analysis of the behavior, the role of adjudicator can be further refined as *comparator*, *voter*, etc.

The precise form of the stereotypes and the corresponding model elements will be specified in Deliverable 2. Here a short summary is presented in Table 2.

| Stereotype | Role |
|---|---|
| <<redundancy manager>> | General identification of classes or objects of redundancy schemes |
| <<variant>> | |
| <<adjudicator>> | |
| <<comparator>> | Subtypes of <<adjudicator>> in user-defined redundancy schemes |
| <<voter>> | |
| <<failure>> | State in the statechart of redundancy manager |
| <<failure>> | Events in statechart of the redundancy manager |
| <<response>> | |

Table 2. Stereotypes used to identify system structure

### 5.3.2. 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. Since tagged values can not be applied to a group of model elements, the common parameters like common mode failure rates have to be given by using other mechanisms. A possible solution is to place the tagged value into a comment, and attach the comment to multiple model elements.

Tagged values can represent the following characteristics:

- fault occurrence
- common mode failure occurrence
- percentage of permanent faults
- error latency
- detection coverage
- propagation probability
- repair delay.

The form of these stereotypes and the corresponding UML model elements are specified in Deliverable 2.

As an example, the structure of stereotypes and tagged values of a general software element (object or component) is presented in Fig. 6. Similar stereotypes and tagged values are applied in the case of hardware elements (Fig. 7.).

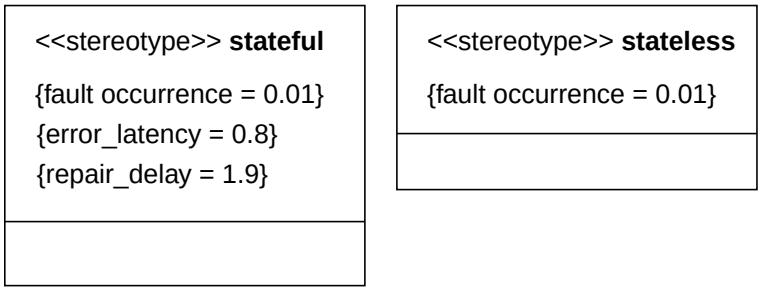


Fig. 6. Stereotypes and tagged values of software components

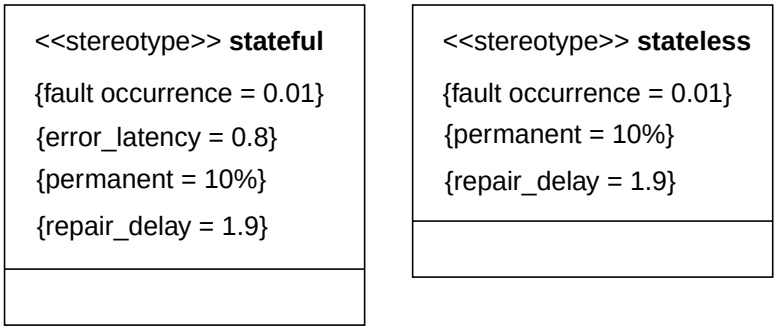


Fig. 7. Stereotypes and tagged values of hardware components

### 5.3.3. Definition of requirements

The easiest way to define requirements in UML diagrams is the application of constraints. For a single UML element, the constraint may be placed near to the (name of the) symbol. For two elements, a dashed line connecting the elements is labeled by the constraint. For three or more elements, the constraint is placed into a comment, which is attached to the elements.

The following requirements can be expressed in the form of constraints:

- requirements relevant to verification (in the form of a Linear Time Temporal Logic)
- requirements relevant to quantitative dependability analysis (i.e. reliability or availability measures)
- performance (timing) requirements.

The form and semantics of the requirements are specified in Section 4 and in Deliverable 2.

**References**

[1] Allmaier, S.: A reward-based result measure concept for GSPNs. Technical Report, IMMD III, Friedrich-Alexander Universität Erlangen-Nürnberg, 1998.

[2] Booch, G.: The visual modeling of software architecture for the enterprise. Rational Rose Magazine, 1998.

[3] Bondavalli, A.; Chiaradonna, S.; Di Giandomenico, F.; and Grandoni, F.: Discriminating Fault Rate and Persistency to Improve Fault Treatment. In: Proc. 27th International Symposium on Fault-Tolerant Computing (FTCS-27), 1997, pp 354-362.

[4] Dal Cin, M.: Checking Modification Tolerance. In: Proc. IEEE High Assurance Systems Engineering (HASE 98), Bethesda, Maryland, Nov. 13-14, 1998.

[5] Damm W.; Hungar H.; Kelb P.; Schlör R.: Statecharts: Using graphical specification languages and symbolic model checking in the verification of a production cell. In: Lewerentz, C.; Lindner, Th. (eds.): Formal Development of Reactive Systems. Springer 1995, LNCS 891, pp. 131-149.

[6] Deseive, S: Einführung von Reward-Maßen bei der Modellierung mit stochastischen Petri-Netzen. Diploma Thesis, IMMD III, Friedrich-Alexander Universität Erlangen-Nürnberg, 1998.

[7] Detlefs, D.; Herlihy, M. P. and Wing, J. M.: Inheritance of Synchronisation and Recovery Properties in Avalon/C++, IEEE Computer, 1988, vol 21, no 12, pp. 57-69.

[8] Douglass R.P.: Real Time ULM: Developing efficient objects for embedded systems. Addison Wesley 1998.

[9] Dugan, J. B. and Lyu, M. R.: Dependability Modeling for Fault-Tolerant Software and Systems. In: M. R. Lyu (editor), Software Fault Tolerance, Wiley & Sons, 1995, pp. 109-137, series Trends in Software, vol 3, New York.

[10] Emerson, E.: Temporal and modal logics. In J. van Leeuwen (ed.) Handbook of Theoretical Computer Science - Vol. B: Formal Models and Semantics, pp. 995--1072. Elsevier Science Publishers B.V., 1990.

[11] German, R.: SPNL: Processes as language-oriented building blocks of stochastic Petri net. In Proc. IEEE Int. Workshop Petri Nets and Performance Models (PNPM '97), St. Malo, France, 1997. IEEE Comp. Soc. Press.

[12] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming. Elsevier, 8(3):231--274, 1987.

[13] Harel, D.: The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering and Methodology, 5(4):293--333, 1996.

[14] Harel, D.: Executable object modeling with statecharts. IEEE Computer 30(7):31--42, 1997.

[15] Huszerl, G.: Formal Verification of Fault-tolerant Systems (A Relational Approach to Model Checking). Diploma Thesis, IMMD III, Friedrich-Alexander Universität Erlangen-Nürnberg and Technical University of Budapest, 1998.

[16] Kosmidis, K.: Konzeption und Implementierung eines auf H-L-PN basierten Steuerungssystems für flexible Produktionszellen. Diploma Thesis, IMMD III, Friedrich-Alexander Universität Erlangen-Nürnberg, 1998.

[17] Laprie, J. C.: Dependability: Basic Concepts and Terminology, 1992, Springer Verlag, series Dependable Computing and Fault Tolerant Systems, vol 5

[18] Maes, P.: Concepts and Experiments in Computational Reflection, ACM SIGPLAN Notices, 1987, vol 22, pp 147-155, Proc. of OOPSLA'87

[19] UML Semantics, version 1.1 (1. September 1997), The Object Management Group, doc. no. ad/97-08-04

[20] UML Notation Guide, version 1.1 (1. September 1997), The Object Management Group, doc. no. ad/97-08-05

[21] www.omg.org/news/pr97/umlprimer.html

[22] Xu, J.; Randell B.; Rubira-Calsavara, C. M. F. and Stroud, R. J.: Towards an Object-Oriented Approach to Software Fault Tolerance, University of Newcastle upon Tyne, 1994, PDCS Technical Report, no 140