Dependability Analysis in the Early Phases of UML Based System Design

Andrea Bondavalli[1], Mario Dal Cin[2], Diego Latella[3], István Majzik[4], András Pataricza[4] and Giancarlo Savoia[5]

[1] University of Firenze, Dip. Sistemi e Informatica, Via Lombroso 6/17, I-50134 Firenze, Italy

[2] Univ. Erlangen-Nürnberg, IMMD3, Martensstraße 3, D-91058 Erlangen, Germany

[3] CNUCE Istituto del CNR, Via V. Alfieri 1, Loc. S. Cataldo, I-56010 Ghezzano (Pisa), Italy

[4] Technical University of Budapest, DMIE, Műegyetem rkp 9, H-1521 Budapest, Hungary

[5] INTECS Sistemi SpA, Via Gereschi 32, I-56127 Pisa, Italy

Abstract

A thorough system specification is insufficient to guarantee that a computer system will adequately perform its tasks during its entire life cycle. The early evaluation of system characteristics like dependability, correctness and performance is necessary to assess the conformance of the system under development to its targets. This paper presents the results achieved so far to develop an integrated environment, where design tools based on the UML (Unified Modeling Language) are augmented with validation and analysis techniques that provide useful information in the early phases of system design. Automatic transformations are defined for the generation of models to capture system behavioral properties, dependability and performance.

Keywords: Dependability analysis, validation and verification, UML, model transformation

1 Introduction

Computer controlled systems are used in many application fields, with different levels of criticality requirements. A common characteristic of such systems is the increasing complexity in internal structure (redundancy, layering of functionality) and operational issues (performance constraints, criticality of the controlled applications, etc.). The increasing need for effective design has contributed to push for the development of standardized and well-specified design methods and languages, which allow system developers to work with a common platform of design tools. The Unified Modeling Language (UML) [1] is a general-purpose visual modeling language that is designed to specify, visualize, construct and document aspects of object oriented systems [2,3]. UML is expected to become a de-facto standard for the design of a variety of systems from small embedded control systems to large and complex open systems.

An effective design process should include an early validation of the concepts and architectural choices underlying system design. The early evaluation of system characteristics like dependability [4], performance and correctness, necessary to assess the compliance of the system under development to its targets, becomes especially important for designing systems supporting critical applications. The simultaneously increasing complexity and dependability requirements of computer controlled systems have, in fact, exposed the limits of the validation techniques traditionally used in industry, like code review, testing, fault trees or Failure Modes and Effects Analysis. Moreover, new technologies such as hardware-software codesign present new challenges for the validation process. The traditional validation techniques are being more and more complemented with advanced validation techniques, such as formal verification, model based dependability evaluation, fault injection. These techniques are not aimed to replace the traditional validation techniques, but should rather be integrated with them.

The validation of designs described using UML was the main objective of our project "High-Level Integrated Design Environment for Dependability" (HIDE) [5]. The purpose of HIDE is to allow the designer to use UML as a front-end for the specification of both the system and the user requirements, and to bridge the gap between a practice-oriented CASE methodology and sophisticated mathematical tools required for dependability validation. Formal verification and quantitative analysis tools will be made available to the designer within the design environment, and the required mathematical models will be derived automatically from the UML specification.

The rest of this paper is structured as follows. Section 2 describes the overall approach of the HIDE project and Section 3 details the structure of the HIDE environment. Section 4 introduces the extensions and constraints applied on the standard UML. Section 5 details the model analysis currently performed in the directions of formal verification, quantitative dependability analysis (where two complementary approaches are followed) and performance analysis. Section 6 describes the assessment of the approach by a demonstrative example. Last section draws some conclusions and identifies our future work.

2 The HIDE approach

While a relevant effort is being devoted to the development of standardized design languages and methods, such as the recently created UML, much less attention has been dedicated up to now to the integration of the design technologies with the validation techniques. At present, the validation of system design is usually performed as a separate project phase postponed to the end of the design process. This practice however suffers from the following weak points:

- Typically, simulation based validation is used for checking the conformance of the system model to the initial specification. However, experimental validation assures only a high likelihood of correctness, but no proof of it.
- No integrated support is currently offered for quantitative validation of dependability and performance, thus leaving these aspects uncovered.
- Pure functional specification leaves the system behavior undefined in the presence of faults. Validation raises the question how to assure dependability in a system built of average quality,

non ultra-reliable components. Moreover, dependability is a crucial cost factor in the after-manufacturing phases of the product life cycle (e.g. maintenance).

While manufacturers of critical systems are prepared and have a considerable experience on the validation of their *products* (driven by the need of certifying and having their systems to be accepted by customers), there are major difficulties and much less expertise in the early validation of system *design*. The use of formal methods for the specification and verification of properties of systems would be one methodological improvement of the system development process, which, together with other techniques, would allow reaching high quality standards. However, the large set of formal analysis techniques and their level of sophistication require a huge investment in staff skill and time. This discourages designers and builders to apply such analysis methods in the phase of the design and implementation process in which they are most effective.

The HIDE project aimed at proposing a convincing and general answer to the need for early validation of system design. It aimed at integration among the design, validation and verification techniques for complex software/hardware systems, through a *transformational approach*. Automatic model transformation techniques from the UML model of the system to the most common formal verification, quantitative and performance analysis tools (like automata, Petri-nets) were elaborated.

The transformations adhere to UML as much as possible. Extensions (using the standard mechanism of UML) are necessary to describe the architecture and properties of the system precisely, so that an automatic tool is able to perform the analysis. The larger model size and run-time requirements of the automatically derived models should not strongly limit the target field of applications in comparison with the manually composed ones.

The results are automatically back-annotated for presentation into the same UML model. The designer is provided with a source of precious information, e.g. highlighting design faults, dependability bottlenecks and identifying to a certain level the possible causes for the design failures and the corresponding recovery actions. The estimation and the prediction of the system properties have an acceptable level of confidence to drive the subsequent design choices. (It has to be emphasized, however, that not all properties related to dependability and performance can be analyzed at the early design phases.)

The entire background mathematics are completely hidden to the designer, thus eliminating the need for both a specific expertise in abstract mathematics and the tedious re-modeling of the system for mathematical analysis. The HIDE framework thus integrates in a user-friendly way the de-facto standard design language UML with a set of validation, verification and evaluation techniques for assuring the quality of service of the system still during the early design phases without the need of special skills or extreme efforts by the designer. This allows to shorten the necessary verification and validation cycle, with a consequent saving in the associated costs. Design refinement is driven by the information gained during the validation process, thus allowing adequate system designs to be produced before implementation and experimental (in the field) validation take place.

3 The HIDE environment

The transformational approach of the HIDE project required an open environment into which different transformations from commercial UML CASE tools towards target mathematical formalisms and corresponding analysis tools can be integrated. In the current phase of the project, the primary target was the definition of a prototyping environment, considering the assurance of a high level of flexibility and a good support of debugging of the algorithms to be implemented. In this phase, both the efficient use of resources and the time requirements of the transformations were of secondary importance. Accordingly, the implementation of the HIDE environment relies (as far as possible) on available tools.

3.1 Architecture of the HIDE environment

The HIDE environment is built up from 3 main components:

- UML CASE tool. The user-end modeling platform is an (arbitrary) UML CASE tool, in which the designer can build up his/her UML model. All tool-provided features like code generation, round-trip engineering and documentation generation, can be used without modifications. However, during the creation of the UML model some required (standard) extensions and restrictions defined by HIDE must be observed in order to be able to access the analysis tools.
- Analysis tools. These tools are off-the-shelf components. The HIDE core acts as an end-user toward the analysis tools by supplying the input for the tool and then by processing and back-annotating the result. One of the challenges of the project was the selection of the proper modeling formalisms and supporting tools in order to be able to perform the analysis required by the designer.
- HIDE core. The HIDE core establishes a bridge from the UML CASE tool to the various analysis tools that require different input formats (based on different mathematical formalisms). The common way to implement this bridge is a set of direct model transformations from the repository of the UML CASE tool to the input formalism of each analysis tool. Instead of implementing each transformation separately, the HIDE environment incorporates a common representation, the so-called HIDE database, in order to develop the model transformations in a uniform framework. The structure of this database corresponds to the structure of the UML metamodel, which was made available for end users [1]. Accordingly, full compliance with the UML standard and a straightforward mapping from the product-dependent repository of the CASE tool is assured. Based on the database representation of the UML model, specialized scripts written in standard database language (PL/SQL) implement the model transformations to the input formalisms of the analysis tools (Figure 1). This approach has the advantage that the transformation scripts are independent of the CASE tool.

  It should be pointed out that the target formalism of the analysis tools might also have an efficient and straightforward database representation. In this way, the core part of the model transformation can be performed as a series of database operations within the HIDE database.

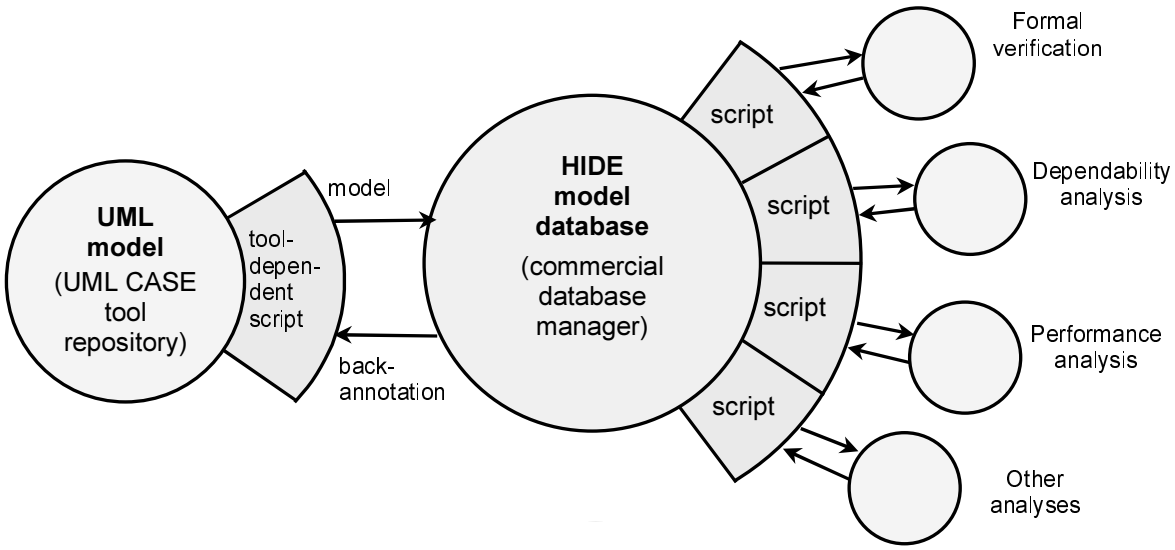


Figure 1. The transformation flow in the HIDE environment

3.2 The HIDE roundtrip

According to the architecture of the HIDE environment, an analysis initiated by the designer consists of several steps (hidden from the designer):

1. The UML model is exported from the repository of the CASE tool to the HIDE model database.
2. The model transformation is implemented partially as a database transformation (from the HIDE model database to the database structure of the target formalism) and partially as a code generation (from the database representation of the target model to the specific input format of the analysis tool).
3. The analysis tool is then called and its result is back-annotated into the HIDE model database.
4. The HIDE model database is imported into the CASE tool and this way the results of the analysis are visualized. Note that the restricted support of the CASE tool may require external visualization modules accessing directly the (back-annotated) HIDE model database, e.g. to show dynamic analysis traces if the CASE tool lacks a simulation engine.

The HIDE roundtrip, which is based on theoretically well-founded model transformations, assures both the faithfulness of the source UML model and the target analysis model, and the unambiguous interpretation of the analysis results.

4 UML modeling - extensions and constraints

At present UML is merely a semiformal language. It provides a set of diagrams to describe different views of object-oriented designs. Its syntax is well-defined, precise rules (given by diagrammatic techniques and textual constraints) specify conditions under which constructs are allowed, but little formal semantics is attached to the individual diagrams and there is no formally defined semantics for the integration of them [6]. Moreover, there are no standard ways to describe non-functional (e.g. dependability) aspects of the design. Accordingly, it is impossible to apply rigorous analysis

techniques to evaluate a UML model as it is. In other words, in order to evaluate an UML-based visual design, the model has to be enriched with additional information and then be transformed to a model with a precise semantics.

UML provides standard mechanisms to *introduce the required extensions* into the model. As these extensions are parts of UML, they are adopted by the CASE tools. We applied the following extensions:

- *Tagged values* are pseudo attributes assigned to model elements in the form of a pair "tag=value". In our case, tagged values are assigned to objects to define dependability parameters like fault activation rate, error latency, failure propagation probability, and repair delay. E.g. repair delay is assigned in the form "`RD=0.02`". Tagged values with empty value field may indicate measures to be computed by the analysis. States and transitions in statechart diagrams are assigned tagged values to define performance-related parameters like transition rates, branching probabilities or general reward functions.
- *Stereotypes* introduce a new class of modeling elements at modeling time by defining a high-level classification (e.g. meaning, usage) of model elements. On structural diagrams, we use stereotypes to identify non-functional properties and roles of objects. For example, stereotypes like `<<stateless hardware>>`, `<<variant>>`, and `<<adjudicator>>` were introduced. On dynamic diagrams, stereotypes indicate failure states and faulty transitions.
- *Constraints* introduce rules that can be checked against the design, thus providing a way to express requirements. In our case, constraints are used to define behavioral requirements (e.g. in the form of linear temporal logic formulae assigned to scenarios or use cases).

In the HIDE environment a strict subset of UML diagrams is considered, containing though the most important conceptual issues interesting from the point of view of dependability analysis. From the point of view of the structure, the UML design is considered as a static set of interacting objects, the dynamic creation and destruction of objects is not allowed. Inheritance is utilized only to derive the properties of the static objects. Redundancy in the design is restricted to classes (objects) and hardware elements. From the point of view of dynamic behavior, the subset includes all aspects related to concurrency like sequentialization, non-determinism and parallelism. Minor peculiarities like deferred events, time and change events, history states, exit and entry actions, activity states etc. are not covered. These latter simplifications were introduced essentially to keep the level of complexity at an acceptable level, since they do not have any strong impact on the semantics. On the other hand, dynamic object-oriented features require further study.

5 Model analysis by transformations

Up to now three transformations were elaborated in the HIDE environment. The first one targets formal verification of dependability-related attributes like freedom from deadlocks, avoidance of unsafe system states. The next two transformations target quantitative analysis of dependability attributes like reliability or availability, and performance measures like throughput or average response time.

Formal verification is a hot topic nowadays in the field of system engineering, especially for the development of critical dependable systems. For a complete software system, formal verification is almost impossible to afford due to complexity issues. It is only performed for critical aspects, or parts, of the system. In HIDE, the dynamic behavior of critical objects, specified by statechart diagrams, is to be analyzed. A transformation has been defined [10] which maps a subset of UML statechart diagrams to Kripke structures (or in general transition systems) for formal verification of functional properties using the model checker SPIN [7]. This transformation is based on a reference formal operational semantics for UML statechart diagrams within HIDE.

Amongst the approaches commonly adopted to evaluate dependability attributes, analytical modeling has proven to be very useful and versatile. Especially during design, models show their usefulness and potentialities. They allow to compare different architectural and design solutions and to run sensitivity analyses identifying both dependability bottlenecks and critical parameters to which the system is highly sensitive. Two complementary approaches have been followed for providing quantitative analyses of dependability attributes. A trade-off has to be made between the degree of details in modeling and the degree of possible automation of the analysis process. The transformations have been defined from UML subsets to Timed Petri Nets (TPN) and Stochastic Reward Nets (SRN) [8]. On one hand, this approach allows using the elaborate and well established Petri Net tools for the quantitative analysis of UML models. On the other hand, it integrates the use of Petri Nets into the object-oriented modeling paradigm of UML. For example, the generated models can be extended by modeling aspects like the integration of fault models, difficult to express directly in UML. Both these two transformations use PANDA [9], as Petri Net analysis tool. It allows to annotate transitions with guards and to use state dependent capacities for arcs. Moreover, PANDA accepts not only exponential distribution functions, but also non-exponential ones. Dependability measures can be specified by reward functions, built from so-called characterizing functions like the number of tokens of a place. PANDA computes the expected value of a reward function at a point in time (e.g. availability) as well as accumulated rewards. Reward functions can be utilized to derive performance related parameters (like throughput, average response time).

5.1 Formal verification

When performing formal verification of dependability related behavioral properties, the scope of investigation is (the set of) UML statechart diagrams. By using statecharts, the behavior of classes (objects) and hardware nodes can be specified. A mandatory prerequisite for formal verification is to map the statechart diagrams to a formal semantics model. The verification technique we aim at is model checking, thus we use Kripke structures. This formal semantics model should satisfy the properties of the UML semantics given informally in [1].

To define the mapping from UML statechart diagrams to Kripke structures, an intermediate model is introduced, which is a modified variant of Extended Hierarchical Automata (EHA) introduced for a similar purpose in [12]. EHA can be considered as an abstract syntax for UML statechart diagrams, since the transformation to EHA is straightforward and purely syntactical. It abstracts from some graphical details and describes only the essential aspects of statechart diagrams, namely the state hierarchy and concurrency. EHA are composed of simple sequential

automata related by a refinement function. A state can be mapped by the refinement function to a single automaton (representing a nested state machine in the statechart diagram) or to a set of automata (representing a concurrent composite state in the statechart diagram). There is a clear correspondence of the states and non-interlevel transitions of the two structures (Figure 2), other details are encoded in the labels of transitions.

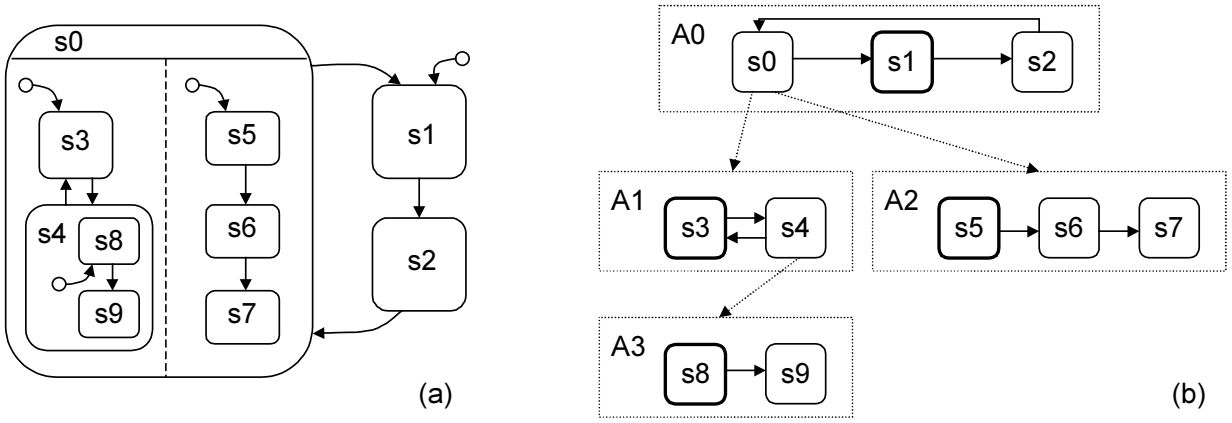


Figure 2. An example of an UML statechart diagram (a) and the corresponding EHA (b)

The next step is the elaboration of the UML semantics of EHA. When defining the semantics, the peculiarities of UML should be taken into account. The fundamental semantics of UML assumes that events are processed by the underlying state machine one by one, where each event stimulates a run-to-completion step. However, the selection of the event from the event queue representing the external environment is defined in the UML semantics only partially. Accordingly, our transformation is parametric with respect to the semantics of the environment (random selection, FIFO ordering of events etc. can be set by the modeler). Selected events enable state transitions, some of them may also be in conflict if the intersection of the set of states they exit is not empty. UML resolves some conflicts by defining that a transition emanating from a substate has higher priority than a transition emanating from a containing state. To cover the priority issues, we use a notion of *priority scheme*, on which the semantics is parametric, and then instantiate it with the UML one. The parametric nature of our semantics definition with respect to transition priorities and the semantics of the environment allows to use the transformation under different priority schemes and for specific run-time environments.

The UML operational semantics of EHA is defined as a Kripke structure that is a set of states (called statuses) related by a transition relation (called step relation). The status is composed by the state configuration of the EHA (i.e. the statechart) and the current environment, the latter can also be manipulated from outside of the automata. The step relation is defined by means of a deduction system. Basically, each sequential automaton of the EHA might contribute in three ways in the step relation.

- Progress: If the automaton has a transition which is enabled and its priority is high enough, then the transition fires and a new status is reached accordingly. The priority is checked on the basis of (1) a cumulative set of transitions which are enabled in the ancestors of the automaton and (2) the set of enabled transitions in the descendants of the automaton.

- Composition: The automata can delegate the execution of transitions to its sub-automata (under the constraints of the set containing also the local enabled transitions) and these transitions are propagated upwards. Note that non-determinism may arise both from conflicting transitions of the same automaton and from the possibility of applying progress or composition.
- Stuttering: If the automaton has no transition that is enabled and no sub-automata exist to which the execution can be delegated then the state of the automaton is not changed.

The formalization of these three rules constitutes the deduction system, which defines the step relation of the Kripke structure, i.e. the formal operational semantics of our subset of UML statechart diagrams [10]. Such semantics have been proven to satisfy requirements specified in the official definition of UML [1]. This simple deduction system forms the basis of the transformation from UML statechart diagrams to PROMELA, input language of the model checker SPIN. The transformation is formally specified and proven correct in [11]. We selected SPIN, since it is one of the most efficient tools available, and PROMELA allows the specification of both state variables and communication actions [7]. However, the recursive nature of the semantics posed problems in the transformation, since PROMELA has very limited recursion capabilities. Accordingly, we restricted the use of recursion only in the transformation function definition in order to exploit the static tree-structure of the statechart diagrams in transformation time. On the other hand, the hierarchical and recursive nature of the above rules helped a lot in carrying on correctness proofs of the semantics: all interesting properties of the statechart diagrams were proven inductively [11].

Requirements to be checked by formal verification are expressed by a linear temporal logic formula or by another automaton transformed to a Kripke structure. In the first case the formula can be included in the form of a constraint (however, not the standardized constraint language of UML is used, since it does not provide an explicit notation for temporal logic). In the second case, the requirement can be expressed again as a (simple) statechart diagram and its resulting semantics can be used for model checking the original statechart diagram.

If a requirement fails, SPIN produces a counter-example in the form of a trace. Traces can be back-annotated into the UML model in the form of sequence diagrams, thus the designer may be able to identify the reason of the failure. However, as the common UML CASE tools do not support guided simulation, the dynamic (step-by-step) presentation of the trace is not solved yet.

The approach we followed shares the relative simplicity of the work proposed in [12] for classical statecharts, but it takes into consideration the peculiarities of the UML relevant for the considered subset of the notation. Several other approaches have been proposed in the literature for the definition of a formal semantics of UML statechart diagrams, e.g. [13,14], and much more has been done for classical statecharts. State refinement is not allowed and transition priorities are not dealt with in [13]. In [14] all interesting aspects of UML statechart diagrams semantics are covered. Unfortunately, no correctness result for the proposed semantics is provided, more emphasis is put on implementation related issues. In [15] a different approach to model checking of UML statechart diagrams (based on the semantics proposed in [10]) has been investigated, which uses branching time temporal logic.

5.2 Dependability analysis of structural design

The first quantitative transformation focuses on the early, system-level dependability modeling based on high-level, structural descriptions. The transformation maps UML structural diagrams (use case, class, object, and deployment diagrams) to TPN [16,17]. This transformation copes with the state explosion by starting from simple high-level models, and making them more and more complex and detailed by refining the relevant parts of the system. Higher level UML models (structural diagrams) are usually available before the detailed, low levels ones. The analysis on these rough models provides indications about the critical parts of the system, which require a more detailed representation. In addition, 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. These might be provided by other transformations dealing with UML behavioral and communication diagrams, like the transformation to be described in Section 5.3.

Accordingly, our transformation:

- allows a less detailed but system-wide representation of the dependability characteristics of the analyzed system,
- provides early, preliminary evaluations of the system dependability during the early phases of the design,
- deals with various levels of detail, ranging from very preliminary abstract UML descriptions, up to the refined specifications of the last design phases, capturing only the features relevant to dependability.

The analysis covers the so-called *class-based redundancy* applied in the UML design. It prescribes that components of a redundancy structure must be defined as specific classes identified by stereotypes. The three basic components are as follows. Variants (stereotype `<<variant>>`) implement the same functionality by replication or design diversity. Accordingly, hardware faults can be tolerated by deploying replicated objects on different hardware, while software bugs can be tolerated by diversity. An adjudicator (`<<adjudicator>>`) is used to check (compare, test, vote) the results of the variants. A single redundancy manager of the structure (`<<redundancy manager>>`) controls the execution of the variants. It may implement specific techniques as well as common schemes like recovery blocks, N-version programming etc. The service of the redundancy structure is available through the redundancy manager and the variants cannot be used separately.

The transformation is defined in more steps. The first one has the fundamental task of extracting the relevant information from the UML diagrams necessary to fix the set of basic and derived failure events like fault activation, failure propagation and repair processes. In this step a general system model called here Intermediate Model (IM) is built. The next step defines a TPN model in a formalism general enough to postpone the choice of the automatic tool. Further steps allow to translate the model to the input format of a specific tool (in our case PANDA).

During the first step of the transformation, UML model elements (objects, nodes, packages, and components) are mapped to simple elements, while redundancy structures are mapped to composite elements of the IM. Hardware and software components, with or without internal states, are distinguished by the type of the element (the distinction is important from the point of view of errors and error propagation). UML relations (associations, compositions, dependencies, deployments) are mapped to "uses service of" (in a special case "deployed on") and "composed of" relations of the IM. Finally, the resulting IM will be a hypergraph, where each node represents an element described in the UML diagrams, and each hyperarc represents a relation between elements.

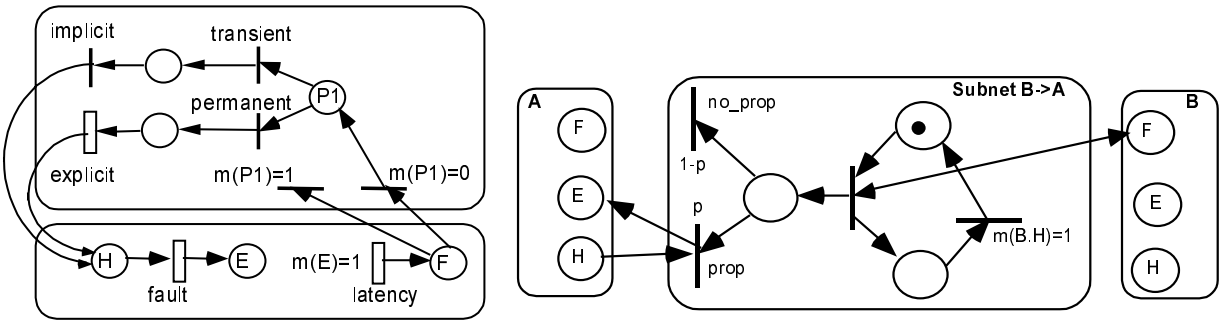
Failure of a composite element, i.e. a redundancy structure, is not only an OR relation of the failures of the simple elements belonging to the given composite one. Usually, a fault tree can be used to describe the effects of separate and common mode failures of the elements. In our approach, it is available either directly from the library of redundancy schemes (constructed by dependability experts), or derived by analyzing the behavioral description of the redundancy manager belonging to the scheme. In the latter case, the designer is required to provide the complete behavior of the redundancy manager in the form of an annotated UML statechart diagram. In such diagrams the paths (trajectories) from initial state leading to failure states or events are followed and the incoming events and conditions are examined, deriving in this way the fault tree automatically.

The second step of our transformation builds the TPN dependability model, by generating a set of subnets for each element of the IM. The subnets include well-defined set of places and transitions that are interfaces towards other subnets of the model. Subnets are a convenient notation to make the model clearer, encapsulating portion of the whole net, thus allowing for a modular and hierarchical definition of the model. Moreover, nested subnets allow the combination of models at the different levels of details.

According to the above described structure and parameters of the IM, the target TPN model consists of the following types of sub-nets:

- Basic failure subnets represent the internal state of the UML element corresponding to each node of the IM, and model the failure processes. Places H and F model the healthy and failed state, occurrence of a fault is modeled by a timed transition. Stateful nodes also include a place E for the erroneous state, and an other timed transition representing error latency (Figure 3a).
- Repair subnets are activated by the failures occurred in the failure subnet of the same node. Two different kinds of repair are modeled as a consequence of transient and permanent faults (Figure 3a). In the case of stateful nodes, the error recovery activity should also be modeled.
- Propagation subnets are generated by examining the "uses the service of" hyperarcs of the IM. They link the basic subnets modeling that a failure occurring in a node may propagate to other node(s) by corrupting its internal state. Propagation subnets are interfaced through the places H, E and F of the basic failure subnets (Figure 3b).
- Fault tree subnets (as special propagation subnets) express the logic with which the failures of composing elements propagate towards the failure of a redundancy structure. Also, the dual of the fault tree represents the way the redundancy structure becomes healthy when the composing elements are recovered. These subnets are both induced by the "is composed of" hyperarcs of

the IM, and generated by mapping the logic of fault trees to TPN [18]. Note that the system failure is usually a simple fault tree consisting of an OR gate connecting the failures of the elements (on the uppermost layer) of the system.



(a) (b)

Figure 3. (a) Basic failure subnet (bottom) and repair subnet (top) of a stateful hardware element, (b) a propagation subnet

Probability and timing parameters of the transitions are available (through the parameters of the IM nodes) in tagged values of the source UML model. Note that if all timing distributions are exponential ones, then the TPN model results in a Generalized Stochastic Petri Net (GSPN [19]). If there are also transitions with deterministic timing then the model is Deterministic and Stochastic Petri Net (DSPN [20]). Both classes are well supported by analysis tools.

The result of the analysis is usually a single value, e.g. reliability or availability of the system. It can be back-annotated as a UML tagged value (assigned to the given use case). In the case of sensitivity analysis, the visualization of the series of computed parameters requires external tools.

5.3 Dependability analysis of dynamic behavior

Another quantitative transformation focuses on UML dynamic model comprising use cases, sequence and statechart diagrams. A rigorous mapping has been worked out from the semi-formal language of the subset of UML diagrams to the formalism of SRN possessing a precise semantics [21]. We dealt with a specific sub-class of statecharts, the so-called Guarded Statecharts (GSC), which are suitable for modeling dependable embedded systems. The control software in these systems is usually a multitasking (multithreading) system receiving sensor signals and omitting actuator signals. Continuous signals are abstracted to discrete signals assuming a finite set of critical values. Events are signal changes.

Guarded statecharts are composed of states and transitions. Transitions are labeled with guards, trigger events and the set of target states. Moreover, transitions are annotated with state transition probabilities, branching probabilities (weights) and/or timing information (rates). Guards are Boolean expressions of predicates *in(state)*, which predicate evaluates to true if *state* is the current state of the GSC or of some concurrent GSC. If a transition fires, the next state is chosen non-deterministically from the set of target states (this way non-determinism required to model external events or fault effects can be modeled). Output is considered as part of the state in which it occurs.

GSCs are not hierarchic – rather, all hierarchy levels (except the bottom level) describe concurrent behavior (concurrent tasks, or threads of the control software). Guarded state transitions can be considered as high-level abstractions of an asynchronous synchronization pattern between tasks. This pattern is inherently multithreaded, because it models a message being passed to another object without the yielding of control. Specific states, so-called public states, describe the events, statecharts generate or respond to.

Augmenting the system model with a realistic fault model is the basis for dependability analysis. Our error model for GSCs is based on the notion of state perturbations. For example, unintended state transitions are state perturbations. An unintended state transition caused by a fault can occur whenever the system is in the state that gives rise to the erroneous transition. Signal losses can cause that guards (referring to public states) are not observed, the guard then evaluates to constant true/false. This way, also sensor and actuator stuck-at faults or loss of messages can be modeled by state perturbations. Finally, using guards also safety requirements, expressed as Boolean expressions over GSC states, can be integrated into the model (describing e.g., collision of certain devices).

For a dependability analysis, GSC models are transformed to SRN models. The main transformation steps are as follows:

- Private states (which do not appear in guard expressions) are transformed to SRN places.
- Public states (states denoting sensor and actuator signals) are transformed to *pairs of* places (Figure 4). Place *State* models the state of the signal and place *State_PUB* models the presence of the signal (referred to in guard expressions). Arcs with state dependent capacities are used to model faults (transitions *F1* and *F2*). Normally, both *State* and *State_PUB* have the same number of tokens. If only *State* contains a token then it means that a fault 'signal lost' has been occurred. If only *State_PUB* contains a token then a spurious signal has been injected (transition *S* can not fire, but the guard corresponding to the signal evaluates to true). The modeler has to provide the failure rates *e1* and *e2*.
- State transitions with time delay are transformed to timed transitions.
- Time-less transitions (labeled with weights) are transformed to immediate transitions (with the same weight).
- Guards and triggers become guards of transitions.

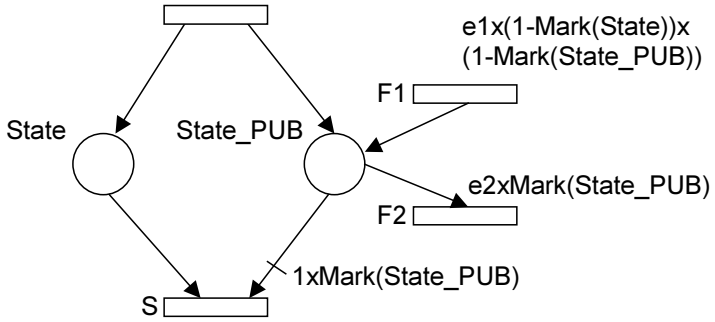


Figure 4. Modeling fault occurrence in SRN models

Thus, we made it possible by the transformation to model communication errors explicitly within the SRN, which fits well to the modeling of embedded systems where the communication between several components is of importance. The dependability measures can be specified by reward functions.

The analysis of detailed GSC models is very time consuming and needs high-performance (parallel) computers. Full GSC models of realistic applications are usually above the complexity modern tools and computers can handle. Thus, quantitative analysis should be focused on certain system components such as embedded controllers. They can be modeled in more detail, while the other system components need not be modeled in details. Here the connection with the system-level structural dependability analysis (Section 5.2) should be useful: system-level sensitivity analysis can identify critical components, while the analysis of dynamic behavior provides parameters useful in the computation of (system-level) dependability attributes.

5.4 Quantitative analysis of performance

The SRN model resulted from the previous transformation (with or without fault models) is amenable also to the analysis of performance. The tool PANDA supports various distribution functions (exponential and non-exponential ones) of time delay, and performance measures can be defined as reward functions.

To handle complexity problems arising during performance analysis, it may be necessary to model and analyze only certain scenarios of the design. Some scenarios may be available as UML sequence diagrams (provided by the designer), some others can be deduced from the statechart diagrams. Usually these sequence diagrams are much less complex than the statechart model itself. The sequence diagrams are transformed to SRN (Figure 5). It can be checked whether the scenario is deadlock-free, and various measures can be computed (e.g. the probability that the scenario terminates after a given time, or distribution functions of time required for specific activities).

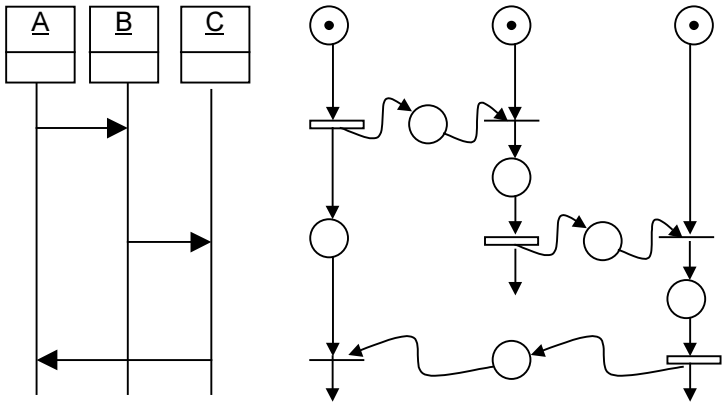


Figure 5. Transformation from a sequence diagram to Petri net

6 Implementation and assessment

Though all the mentioned transformation procedures have been worked out, only some of them were implemented in the HIDE environment. Moreover, no examples of applying the

transformations for large systems are available yet. Neither to provide large UML specifications to exercise the transformation, nor the optimization of the transformation procedures (from a quantitative viewpoint) was in the scope of the HIDE project. Accordingly, we are far from being able to provide real quantitative assessment of the transformation. However, some remarks on the qualitative assessment are possible.

6.1 Qualitative assessment

The qualitative criteria for the assessment of the transformations are (i) adherence to UML modeling techniques, (ii) semantic truthfulness and (iii) availability of the formal definition of the transformations.

From the viewpoint of adherence to UML, it can be stated that all transformations use well-defined subsets of UML diagrams, and the new parameters are defined using the standard extension mechanisms of UML.

Semantic truthfulness and the comparison of automatically generated models and that of generated by hand are assessed as follows:

- Formal verification: The transformation is defined formally. The formal proofs of the interesting properties of EHA and their semantic model, including the correctness of such a model with respect to the informal definition of UML statechart diagrams, show that the transformation is sound from a semantical point of view [11]. The generated number of states in the PROMELA code has only a small overhead in comparison with the number of statuses of the corresponding Kripke structure. It is due to the intermediate states generated by SPIN, and grows only linearly with the size of the automaton.
- Dependability analysis: The UML diagrams that form the input of this transformation do not have a formal semantics and also the specification this set provides might be incomplete or ambiguous, so formal correctness of this transformation cannot be provided. The number of places and transitions in the generated TPN is proportional to the number of model elements in the UML diagrams: the IM is result of a direct mapping, and all IM elements are translated to predefined subnets. No more concise TPN can be used to represent the given failure/repair scenario.
- Analysis of dynamic diagrams: These transformations are based on a subset of the UML dynamic diagrams. The generated SRN exhibit the same behavior as their UML counterparts. Here the number of elements of the SRN is the same as the number of modeling elements in the UML model (except the additional states and transitions introduced to model faults).

The transformation required for formal verification is defined formally [11], the others are given informally but in a rigorous way.

6.2 Implementation of the transformations

The three transformations were implemented in the HIDE environment by PL/SQL scripts. The HIDE database structure of an UML design consists of 121 tables (corresponding to the architectur of the UML metamodel).

The PROMELA code generation for formal verification is approximately 2000 lines, while the generation of the SRN for the analysis of dynamic diagrams is about 1000 lines. The internal representation of these models in the HIDE database comprises only 4 and 6 tables, respectively. The system-level dependability model is generated by PL/SQL scripts and a separate Java application.

6.3 Pilot application

We demonstrated the transformational approach by a small example. It is a variant of the production cell model [22]. The system contains a press that processes metal plates, a robot with an extensible arm for loading and unloading the press, a rotary table for moving blank plates and a feed belt for transferring plates. The production cell was extended to be fault tolerant by inserting a redundant press and a worker being able to repair faulty components (Figure 6).

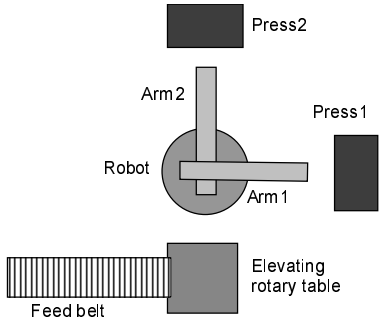


Figure 6. The production cell

The UML model of the production cell was manageable. A single use case (producing plates) was elaborated including 2 typical scenarios (sequence diagrams), all structural diagrams (object diagrams with 11 objects and a deployment diagram) and all dynamic behavioral diagrams (11 statechart diagrams, 73 states and 89 transitions). The main object diagram is depicted in Figure 7, while one of the statechart diagrams (describing the behavior of the rotary table controller) is presented in Figure 8. All software objects were deployed on a single controller PC.

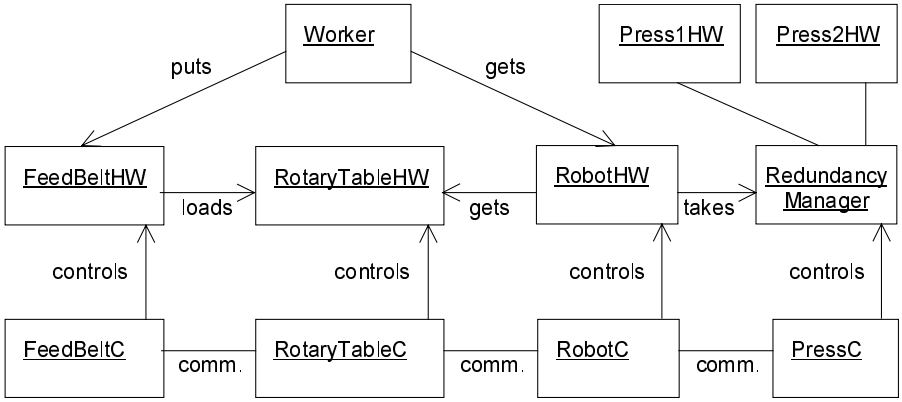


Figure 7. Object model of the production cell

The UML model of the production cell model was analyzed from the viewpoint of functional correctness and dependability.

The transformation from the UML statechart diagrams to PROMELA code for verification required about 3 minutes (including database export). The verification was performed using SPIN version 3.2.3 running on an Intel Celeron 400MHz Linux PC. The deadlock checking of the model was performed in about 1 sec. (generating 1320 states in 441329 atomic steps) and required 14 Mbytes of memory. The liveness checking of the model was started to look for non-progress cycles (here progress means that a processed plate is removed from the production cell). The verification resulted in the failure of the liveness property due to possible cycles of failure-repair of a press without progress in processing the blank plates. Only 181 states (in 29866 atomic steps) were visited during this analysis.

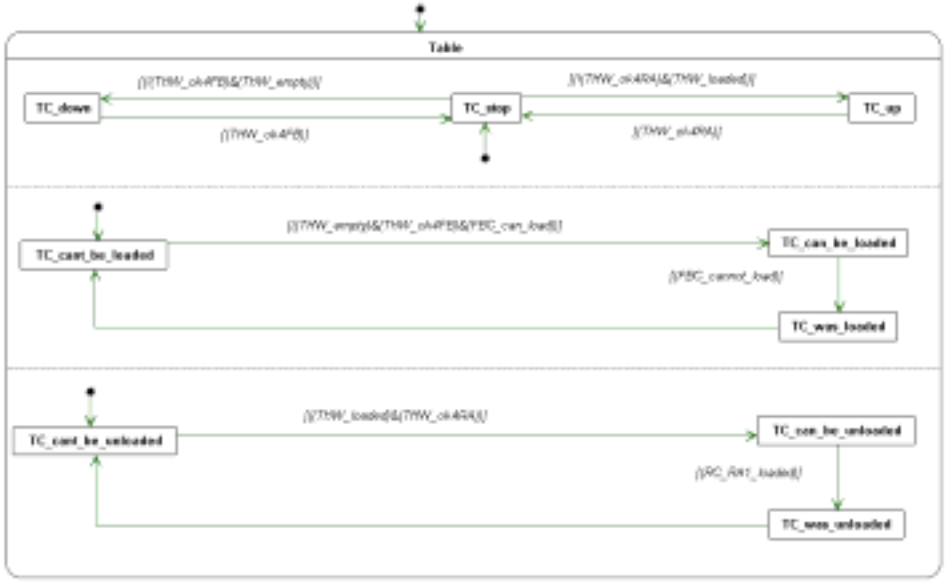


Figure 8. The GSC model of the rotary table controller

The system level reliability model of the production cell was derived by the transformation from the UML object and deployment diagrams extended with failure rates (failure/hour) and error propagation probabilities. The overall structure of the resulting TPN reliability model and the parameters are presented in Figure 9. Here only reliability is concerned, thus repair is not modeled. The basic and propagation subnets were introduced in Figure 3 (here SFE-HW and SFE-SW denote stateful hardware and software elements, respectively). The fault tree corresponding to the press subsystem (subnet C at the upper right corner of Figure 9) and its TPN representation are detailed in Figure 10. The model consisted of 37 places and 37 transitions, the underlying Markov-chain contained 1344 states. The reliability of the production cell during the working time (a 10 hours mission) was computed varying the failure rate of the presses. The solution of the model required 33 seconds, using PANDA on the same machine as above. The result is depicted on Figure 11. The thick dashed line shows the reliability of the reference production cell with a single press only (failure rate is 0.05/hour). It turns out that the sensitivity of the production cell to the reliability of the press can be significantly reduced if two presses are used. (This, of course, does not prove that the press is the dependability bottleneck in the system, since other components should be checked as well.)

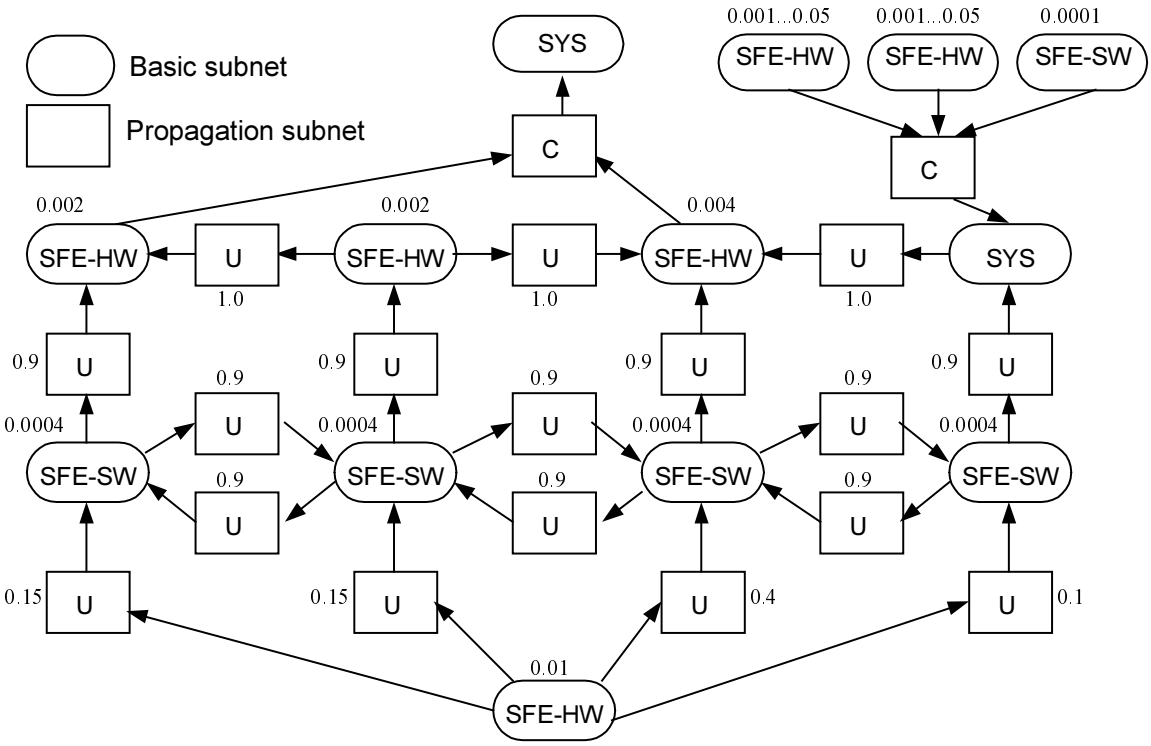


Figure 9. Overall structure of the SRN dependability model

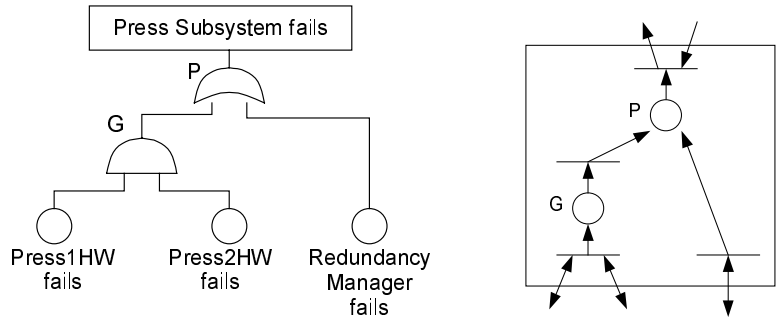


Figure 10. Fault tree and its Petri-net representation

The performance analysis of the system required the transformation of the behavioral models (guarded statecharts extended with timing information) to an SRN model. The transformation required a few minutes; the resulting SRN had 133 places and 98 transitions (83 guarded and 15 unguarded). Performance measures like utilization of the worker (repairman) and the throughput of the production cell were computed [21]. The typical scenarios (sequence diagrams) were transformed to SRN separately. A sequence diagram describing the failure of a robot arm and its repair (including 15 messages) resulted in an SRN with only 51 places and 31 transitions. By solving these SRNs simplified performance measures like distribution of the duration of a scenario could be computed [21].

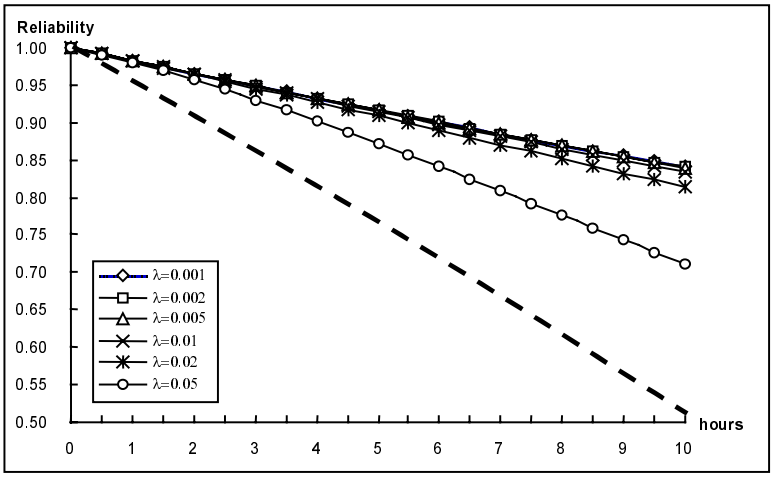


Figure 11. Reliability of the production cell

This kind of multi-aspect, automatic analysis of the same model provides to the designer consistent and detailed information that can help to take decisions like the choice of redundancy, selection of the appropriate hardware components, deployment of software components. However, it has to be emphasized that our analyses focussed on dependability-related parameters, and the final decision needs also checking of other aspects like costs, time required for implementation etc.

7 Conclusion

This paper reports the results of our exploratory HIDE project. The project aimed at proposing a convincing and general answer to the need for early validation of system design, supporting the design and validation strategy of (embedded) dependable systems. It intended to contribute to the integration among the design, validation and verification techniques for complex software/hardware systems, through a transformational approach that targets the most common analysis tools.

The project was of special interest for our industrial partners since it tries to extend the application area of UML toward dependable, critical systems. It promises a speed-up in the design of such systems and an increase of the quality of service during the product life cycle, both with a consequent saving in the associated costs. The support of model-based analysis is of competitive advantage in the field of CASE tools, especially if the application of formal methods does not require specific training and experience.

Part of the work performed so far has been described, namely the transformations for achieving formal verification and quantitative dependability analysis, where two complementary approaches were followed. Work is currently in progress to provide a more general transformation methodology [23] and investigation started for dealing with timeliness analyses in the HIDE context.

Acknowledgements

The work described in this paper was supported by the EC in the framework of ESPRIT LTR 27439 HIDE "High-Level Integrated Design Environment for Dependability". Parterns were FAU (Erlangen, Germany), CPR-PDCC (Pisa, Italy), TUB (Budapest, Hungary), MID GmbH

(Nuremberg, Germany), and Intecs Sistemi SpA (Pisa, Italy). The authors express their thanks to all people involved in the project, especially to K. Kosmidis and W. Hohl (FAU), I. Mura (CPR-PDCC), Gy. Csertán, Cs. Szász, J. Jávorszky, G. Huszerl (TUB), and A. Borschet (MID).

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 and Softeam, "UML Notation Guide", "UML Semantics", version 1.1, 1997.
2. J. Rumbaugh, I. Jacobson and G. Booch, "The Unified Modeling Language Reference Manual". Addison-Wesley, 1999.
3. M. Fowler and K. Scott, "UML Distilled. Applying the Standard Object Modeling Language". Addison-Wesley, 1997.
4. J. C. Laprie, "Dependability - Its Attributes, Impairments and Means". In B. Randell, J. C. Laprie, H. Kopetz and B. Littlewood (Eds), "Predictably Dependable Computing Systems", pp 3-24, Springer Verlag, 1995.
5. A. Bondavalli, M. Dal Cin, D. Latella and A. Pataricza, "High-level Integrated Design Environment for Dependabiliy (HIDE)". In Proc. Fifth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS-99), Monterey, California, pp. 87-92, 1999.
6. M. D. Fraser, K. Kumar and V.K. Vaishnavi, "Strategies for Incorporating Formal Specifications in Software Development". Comm. of the ACM, Vol. 37, pp. 74-86, 1994.
7. G. Holzmann, "The Model Checker SPIN". IEEE Transactions on Software Engineering, Vol. 23, pp 279-295, 1997.
8. G. Ciardo, A. Blakemore, P. Chimento, J. Muppala and K. Trivedi, "Automated Generation and Analysis of Markov Reward Models using Stochastic Reward nets". In Linear Algebra, Markov Chains and Queueing Models, Springer Verlag, 1992.
9. S. Allmaier and S. Dalibor, "PANDA - Petri Net Analysis and Design Assistant". In Proc. Performance TOOLS'97, Saint Malo, France, 1997.
10. D. Latella, I. Majzik, and M. Massink, "Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini and R. Gorrieri, editors, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems, Kluwer Academic Publishers, 1999.
11. D. Latella and I. Majzik and M. Massink, "Automatic Verification of UML Statechart Diagrams using the SPIN Model-Checker". Formal Aspects of Computing, The International Journal of Formal Methods. Vol. 11, No. 6, pp. 637-664, Springer, 1999.

12. E. Mikk, Y. Lakhnech and M. Siegel, "Hierarchical Automata as Model for Statecharts". In R. Shyamasundar and K. Euda (eds), "Third Asian Computing Science Conference ASIAN'97, LNCS-1345, Springer Verlag, pp. 181-196, 1997.

13. R. J. Wieringa and J. Broersen, "A Minimal Transition System Semantics for Lightweight Class and Behavior Diagrams". In Proc. ICSE98 Workshop on Precise Semantics for Software Modeling Techniques, Munich, Germany, 1998.

14. J. Lilius and I. Paltor Porres, "The Semantics of UML State Machines". Technical Report 273, Turku Centre for Computer Science, Abo Academy University, Turku, Finland, 1999.

15. S. Gnesi, D. Latella and M. Massink, "Model Checking UML Statechart Diagrams Using JACK". In A. Williams, editor, Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE'99), pp 46-55, IEEE Computer Society Press, 1999.

16. A. Bondavalli, I. Majzik and I. Mura, "Automated Dependability Analysis of UML Designs". In Proc. ISORC'99, the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Saint Malo, France, pp. 139-144, 1999.

17. A. Bondavalli, I. Majzik and I. Mura, "Automatic Dependability Analysis for Supporting Design Decisions in UML". In A. Williams, editor, Proc. Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE'99), IEEE Computer Society Press, 1999.

18. M. Malhotra and K. S. Trivedi, "Dependability Modeling Using Petri Nets". IEEE Transactions on Reliability, Vol. 44, no. 3, pp 428-440, September 1995.

19. 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.

20. 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, Springer Verlag, 1987.

21. M. Dal Cin, G. Huszerl and K. Kosmidis, "Quantitative Evaluation of Dependability-Critical Systems Based on Guarded Statechart Models". In A. Williams, editor, Proc. Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE'99), pp. 37-45, IEEE Computer Society Press, 1999.

22. C. Lewerentz, Th. Lindner (eds), "Formal Development of Reactive Systems". Lecture Notes in Computer Science, Vol. 891, Springer Verlag, 1994.

23. D. Varró, G. Varró and A. Pataricza, "Designing the Automatic Transformation of Visual Languages". In H. Ehrig and G. Taentzer, editors, Proc. GRATRA 2000, Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems, Berlin, pp. 14-21, 2000.