Technical Report soft-13-02, Chair for Software Engineering, University of Konstanz, Copyright by the Authors 2013

Quantitative Safety Analysis of Non-Deterministic System Architectures

Adrian Beer[1], Uwe Kühne[2], Florian Leitner-Fischer[1],
Stefan Leue[1], Rüdiger Prem[2]

[1]Universität Konstanz, Germany
[2]EADS Deutschland GmbH / Cassidian

Abstract. The QuantUM modeling framework and analysis tool, which allows for the analysis of quantitative aspects of system architectures modeled in UML / SysML, does not offer an adequate treatment of non-determinism. We present an extension of the QuantUM approach based on an interpretation of QuantUM models as Markov Decision processes so that non-determinism in these models is semantically interpreted in an appropriate way. We show that the formal semantic interpretation of the UML / SysML models that we propose coincides with the code generation semantics for a widely used UML / SysML CASE tool. We evaluate the proposed approach by applying it to two industrial strength case studies, an Airport Surveillance Radar system and an Airbag Control Unit.

1 Introduction

Safety-critical software and systems development is subject to special dependability requirements. Early analysis of dependability requirements during the design and development phases is often a statutory condition for the approval of technical systems. For example, the DO-178C / ED-12C [3] and SAE-ARP4761 / 4754A [22,23] standard for aviation systems or the ISO 26262 [24] standard for automotive systems impose such statutory conditions. The recently introduced QuantUM approach [29] allows for the automatic quantitative safety analysis of UML and SysML models. In QuantUM, UML [1] or SysML [18] system models can be annotated using UML profiles that QuantUM provides. QuantUM translates the annotated models automatically into the input language of the probabilistic model checker PRISM [27]. The latter is then used to calculate probabilities for the reachability of system states which can, for instance, represent the occurrence of hazards. QuantUM also generates probabilistic counterexamples using the DiPro [6] tool. These counterexamples can be used to generate probabilistically annotated Fault Trees [36] and UML sequence diagrams in order to visualize the analysis results on the level of the UML/SysML model.

The architecture of complex embedded systems usually comprises concurrently executing subsystems, also referred to as system components. The analysis of these concurrent system architectures comprises the analysis of non-deterministic decisions which are introduced into system models due to the following reasons:

- *Environment behavior*: Since the exact timing of environment interactions at the modeling stage is generally not known, it is typically assumed that any environment interaction of a model can happen non-deterministically at any point in time.
- *Concurrency*: When interpreting concurrency based on an interleaving model, a scheduler will non-deterministically decide at run-time which event out of a set of concurrently enabled events will be executed in the next computation step.
- *Abstraction*: During modeling, some aspects of the system behavior may be replaced by non-deterministic choices between transitions to achieve a certain level of abstraction of the system model. Even though such non-deterministic choices are an artifact that the language definitions of UML and SysML do not include, the UML and SysML models that we analyze may still include such non-determinism since the CASE tools used to edit these models do not enforce the absence of non-determinism. In particular, some details of the design my not be known at a certain stage of the development process. System designers often mask such design incompleteness using non-deterministic choices in the model.

In conclusion, in QuantUM we are faced with all three of the above forms of non-determinism. The probabilistic modeling notations of Continuous-Time Markov Chains (CTMCs) [8] or Discrete-Time Markov Chains (DTMCs) [9] only know probabilistic transitions, which means that they only include transitions that are labeled with transition rates or transition probabilities, respectively. If it is necessary to use non-deterministic transitions for any of the reasons discussed above one has to resort to Markov Decision Processes (MDPs) [34] in the discrete time case, and Continuous Time MDPs (CTMDPs) [32] in the continuous time case, which encompass both non-deterministic and probabilistic transitions.

Currently, QuantUM only supports the modeling and analysis of CTMCs. In order to obtain a more precise analysis of concurrent system architectures we will, hence, introduce a concept of non-deterministic choice into the semantics of QuantUM which means that we will have to map QuantUM models to MDPs. In [26] the use of arbitrary CTMDPs in the context of various property analysis scenarios is shown to be inefficient in terms of runtime and memory consumption. Nevertheless, we want to be able to analyze continuous rates in our model, since the negative exponential distribution rates offered by CTMDPs can easily be mapped to failure rates frequently used in industrial practice. In order to address the above cited efficiency concerns we propose to use an approximation approach [35] often used in practice. With this approximation scheme it is possible to integrate continuous time rates with the discrete semantics of MDPs. The error in the computation of the probabilities that this approximation introduces can be estimated. Our goal is to define a sound translational semantics for the conversion from UML / SysML model to MDPs which is also efficiently implementable in practice. Therefore, we will adopt existing semantics definitions [25,19] for UML / SysML and extend them for use in quantitative safety analysis. Furthermore, we will show that our semantics definition coincides with the code generation semantics of IBM Rational Rhapsody[1] [16] (from now on called Rhapsody), which we consider an important representative of the class of UML / SysML CASE tool used in industrial practice.

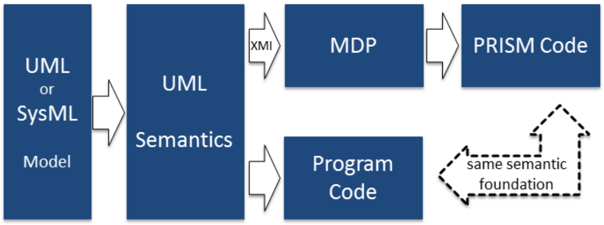

Fig. 1. The new semantics approach

Figure 1 shows the outline of our approach. From the UML / SysML model the program code is generated by Rhapsody using the standard UML / SysML semantics informally given by the OMG [1]. The same UML / SysML model is used by QuantUM to generate the MDP and further the PRISM code to be analyzed. The semantics foundation for both the program code and the PRISM code is the same.

The contributions of this paper can be summarized as follows:

- We present an automatic translation of UML and SysML models that have been annotated using the QuantUM profile into MDPs.
- A semantics definition for the translation is given which coincides, as argued above, with the code generation semantics of Rhapsody.
- Using two industrial case studies we show that the non-deterministic approach models concurrent system architectures more adequately than using CTMCs, as it was previously done in QuantUM.

The remainder of this paper is structured as follows. In Section 2 we show the preliminaries needed in later sections. In Section 3 we present our non-deterministic semantics extension of QuantUM. Section 4 is devoted to the description of the case studies. The evaluation of the results is presented in Section 5. Related work is discussed in Section 6. We conclude in Section 7.

2 Preliminaries

2.1 UML / SysML

UML and SysML are architecture description languages commonly used in the model-based development of systems. While UML [1] is mostly used in software

[1] `http://www-142.ibm.com/software/products/de/de/ratirhapfami/`

engineering, SysML [18] fits the needs for systems engineering more adequately. An example for this domain adaptations are the notion of *classes* and *blocks*. Syntactically classes and blocks are identical. On the semantic level classes represent different software classes which can run concurrently in one piece of software. Blocks can also represent different machines executing in a physically separated way. At the abstraction level of our UML / SysML models we can handle classes and blocks identically since it does not matter for the analysis whether two classes / blocks execute concurrently on the same or on different machines.

The language definitions of UML and SysML encompass a common part which includes Statecharts.

Statechart. A Statechart is a tuple $SC = \{S, Var, A, E, G, Edges\}$. S is the set of states and E is the set of events. Var is a set of variables. Notice that we only allow bounded integer variables to ensure finite-stateness of the model. A is a set of actions which can change the valuation of variables. G is a set of guards. A guard can be constructed using predicate logical expressions on boolean variables or arithmetic comparisons using operators from the set $\{<, \geq, =, \leq, >\}$ on the variables from Var, or integer constants. We use a relation $children : S \rightarrow \mathcal{P}(S)$ to define a partial order on S. This relation defines for each state the set of children of that state. If a state has no children it is called a basic state. Otherwise it is either an AND or an OR-state. *Edges* is a set of transitions. A transition is a tuple (x, e, g, A, y) where $e \in E \cup \{\}$ is the event which triggers the transition, $g \in G$ is the guard for the transition and $A \subseteq Act$ is the set of actions which is executed. $x, y \in S$ are source and target states, respectively. The *scope* of a transition is defined as the lowest OR-node in the hierarchy that contains both the source x and the target node y. A set of statecharts SC_i is called a system.

2.2 Probabilistic Model Checking

Probabilistic model checking [9] requires two inputs. The first is a description of the system to be analyzed, typically given in some model checker specific modeling language. The second input is a formal specification of some quantitative properties of the system that are to be analyzed. They may, for example, relate to its performance or reliability. From the first of these inputs, a probabilistic model checker constructs the corresponding probabilistic model. This model is a probabilistic variant of a state-transition system, where each state represents a possible configuration of the system being modeled and each transition represents a possible evolution of the system from one configuration to another over time. The transitions are labeled with quantitative information specifying the probability and/or timing of the occurrence of the transition. In this paper we will use Markov Decision Processes (MDPs) [9] as the probabilistic model to be analyzed.

Definition 1 (Markov Decision Process).
An MDP is a tuple $M = (S, Act, \mathbf{P}, s_0, AP, L)$ where

- *S is a finite set of states*

- AP *is a set of actions*
- $\mathbf{P} : S \times \mathrm{Act} \times S \to [0,1]$ *is a transition probability function such that for all states* $s \in S$ *and actions* $\alpha \in \mathrm{Act}$*, the sum of all outgoing probabilities is 1:*

$$\sum_{s' \in S} \mathbf{P}(s, \alpha, s') = 1$$

- $s_0 \in S$ *is the initial state.*
- AP *is a set of atomic propositions*
- $L : S \to \mathcal{P}(\mathrm{AP})$ *is a labeling function, i.e. the interpretation of AP.* $\mathcal{P}(\mathrm{AP})$ *is the power set of* AP

Informally, in each state $s \in S$ an MDP chooses the next action to be executed non-deterministically from a set of actions. Afterwards, the target state is selected according to the probability distribution associated with the chosen action.

A *step* in an MDP is an expression of the form $s \xrightarrow{\mu} s'$ where $s \in S$ and $\mu \in P$ and $\mu(s') > 0$. Further a *path* is defined as an infinite sequence of steps starting in the initial state: $s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \xrightarrow{\mu_3} \ldots$ We assume a standard probability measure on the paths, the probability of reaching a state s_n in a path p is the product of all probabilities along the path until s_n is reached: $\mu(p) = \prod_{i=1}^{n} \mu_i$

The quantitative properties of the system that are to be analyzed are specified using Probabilistic Computation Tree Logic (PCTL) [9]. Here we give a short introduction into PCTL for a more comprehensive description we refer to [9]. PCTL is a probabilistic variant of the Computation Tree Logic (CTL) [15] with state and path formulae. The state formulas are interpreted over states of an MDP, whereas the path formulas are interpreted over paths in an MDP. PCTL extends CTL with two probabilistic operators that refer to the steady state and transient behavior of the model. The steady-state operator refers to the probability of residing in a particular set of states, specified by a state formula, in the long run, whereas the transient operator allows us to refer to the probability of the occurrence of particular paths in the MDP. In order to express the time span of a certain path, the path operators until ($\mathcal{U}$) and next ($\mathcal{X}$) are extended with a parameter that specifies discrete time steps in the model. For instance, the PCTL formulae

$$P_{<0.3}\left(\texttt{true}\ \mathcal{U}^{\leq 100}\ \texttt{shutdown}\right)$$

represents the property that the `shutdown`-state is reached within 100 time steps with a probability of less than 0.3.

2.3 The PRISM Language

Since our work involves a translation into the PRISM language [27] we now give a short overview of the input language of the PRISM model checker.

The PRISM language is a state-based, guarded command language that is based on the reactive modules formalism of Alur and Henzinger [7]. For a precise definition of the semantics of PRISM we refer to [28]. A PRISM model is composed of a number of *modules* which can interact with each other. A *module* contains a number of local variables. The values of these variables at any given time constitute the state of the *module*. The global state of the whole model is determined by the local state of all *modules*. The behavior of each module is described by a set of commands. A command takes the form:

$$\begin{aligned}[\text{action\_label}]\ \text{guard} \rightarrow\ & \text{prob}_1 : \text{update}_{1_1}\&...\&\ \text{update}_{1_n} \\ & + \ldots \\ & + \text{prob}_n : \text{update}_{n_1}\&...\&\ \text{update}_{n_n}\end{aligned}$$

The *guard* is a predicate over all the variables in the model. The *update* commands describe multiple transitions which the module can take with certain probabilities if the *guard* is true. Using the *action_label*, synchronization with other modules is possible. A transition is specified by giving the new values of the variables in the *module*, possibly as a function of other variables. Different probabilities, denoted by $prob_{k_n}$, can lead to different target states. The probabilities have to sum up to one in order to form a correct MDP. If different guards are enabled and multiple transitions are executable, one of them is chosen non-deterministically and afterwards a probabilistic decision is made upon the probability distribution of the transition. An example of a PRISM model is given in Listing 1.1. The module named *moduleA* contains two variables: *var1*, which is of type Boolean and is initially *false*, and *var2*, which is a numeric variable and has initially the value 0. If the guard (var2 $<$ 4) evaluates to true, the update ($\text{var2}' = \text{var2} + 1$) is executed with a probability of 0.8 or the update ($\text{var2}' = \text{var2} + 2$) is executed with a probability of 0.2. If the guard ($\text{var2} = 4$) evaluates to true, the update ($\text{var1}' = \textit{true}$) with probability 1.0.

```
module moduleA
   var1: bool init false;
   var2: [0..5] init 0;
   [Count] (var2 < 4) -> 0.8: ( var2'= var2 + 1)
                       + 0.2: ( var2'= var2 + 2);
   [End] (var2 = 4) -> 1.0: ( var1'= true);
endmodule
```

Listing 1.1. A module in the PRISM language.

2.4 The QuantUM Approach

The QuantUM approach [29] was recently introduced in order to support the safety analysis of complex system architectures. QuantUM supports the specification and analysis of dependability requirements at the level of UML and SysML. The information needed for this kind of analysis, such as failure modes, rates and probabilities, can be specified directly in the UML / SysML models using stereotypes [1,18] provided by QuantUM. Stereotypes are meta classes which can be used to add additional information directly to the different UML / SysML constructs. One example for this construction are failure patterns. In QuantUM a failure pattern can be described by additional Statecharts for a component. This means that even if a component has failed it can interact with the entire system and could cause other components to crash, for example, through wrong message passing or providing seemingly correct results on faulty inputs. The translation of the UML / SysML models into the PRISM input language uses the XMI-File exchange format [2] in order to access the model definition in Rhapsody. The PRISM model checker is encapsulated by QuantUM and is made fully transparent to the user, hence lowering the acceptance bar that formal methods often face in industrial engineering practice. The analysis in QuantUM is fully automated, since a manual analysis would be entirely unfeasible, due to the inherent system complexity. The results of the analysis provided by QuantUM are presented to the user in terms of failure probabilities, Fault Trees [36] and UML sequence diagrams.

In this paper we introduce the analysis of non-deterministic models into the QuantUM framework. The Continuous Time Markov Chains (CTMCs) [4] currently supported by QuantUM do not cover non-deterministic behavior.

Definition 2 (Continuous Time Markov Chain [4]). *A CTMC is defined by a finite set of states S with s_0 as initial state and a transition rate matrix $\boldsymbol{R}$ which assigns a continuous rate to every transition in the system. Further, a set AP denotes the atomic propositions and $L : S \to \mathcal{P}(AP)$ a labeling function. A* step *in a CTMC is defined as an expression $s \xrightarrow{\lambda} s'$ where $s, s' \in S$ and $\lambda \in \boldsymbol{R}$. A* path *is constructed as an infinite sequence of steps: $s_0 \xrightarrow{\lambda_1} s_1 \xrightarrow{\lambda_2} s_2 \xrightarrow{\lambda_3} \ldots$ The probability measure for the path in a CTMC uses the exponential distribution defined in.*

For every transition in a CTMC a rate has to be defined. If rates are assigned to transitions which do actually not posses stochastic behavior, the probabilities returned by the analysis are too optimistic which means that the returned probability is less than the actual one. This can be explained as follows. Since in a CTMC every transition has to have a rate attached, we set predefined rates for non-failure transitions. These predefined rates are set to be several orders of magnitudes bigger than the failure rates in order to minimize the introduced error. The following example illustrates how this way of modeling non-determinism leads to overtly optimistic probabilities.

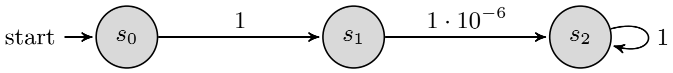

Fig. 2. Example CTMC

Example. In the simple CTMC depicted in Figure 2 there is only one path starting in s_0. The rate $\lambda_1 = 1$ from s_0 to s_1 is the predefined high rate, which is used to express non-stochastic behavior. The problem with this rate is that it will reduce the probability of the whole path in every step. If, for example, the probability of reaching s_1 within one hour is to be computed, we obtain a probability of ≈ 0.63 while one would expect a probability of 1 to represent non-stochastic behavior. This phenomenon can be observed even if the rates representing non-stochastic behavior are set higher. Along very long paths of originally non-stochastic transitions the described phenomenon can have a significant effect on the computed path probability. As opposed to this in a similarly defined Markov Decision Process the probability of reaching state s_1 would be 1 for every time step chosen for the transition. Note that in an MDP we have discrete time steps while in CTMCs they are continuous. We will, however, show in Section 3.4 how to discretize the failure rates for the use in MDPs while leaving non-stochastic transitions always with probability 1.

3 From UML / SysML to MDPs

Our semantics definition for the translation of UML / SysML into MDPs uses notations presented in [25,19]. We propose an adaption of the semantics presented in [25] where a combination of probabilities and rates is covered. We will first give an overview of the code generating semantics of Rhapsody. Afterwards, we will define the formal semantics of translating UML / SysML Statecharts into MDPs with respect to the code generation semantics of Rhapsody.

3.1 A Quantitative Extension of Statecharts

In order to cover quantitative behavior in Statecharts we have to extend the syntax definition of UML / SysML Statecharts given in Section 2.1. We leave the syntax for the definition for the set of states S, the actions Act, the events E and the guards G unchanged. The transition relation is replaced by a quantitative version which handles probabilities and continuous rates:

A quantitative transition either is of the form (X, e, g, P) (P-transitions), where X is the source state, e is the event, g is the guard and P is the discrete probability relation which consists of all target states and a probability to reach the target state. Otherwise, a transition is of the form (X, e, g, R) (R-transitions), where R is the continuous transition rate relation and the rest of the tuple is the same as for the probabilistic case. We impose a restriction on the syntactic form of the UML / SysML Statecharts in order for them to be analyzable with our method: All P-transitions must belong to the same *scope*.

3.2 Rhapsody Semantics

Since we want the QuantUM approach to be applicable to models designed using the Rhapsody tool we have to show that the semantics used in QuantUM for the translation from UML / SysML to PRISM coincides with the code generation semantics assumed by Rhapsody [16]. Otherwise, analysis results that we obtain by QuantUM may not hold for the synthesized program code.

Code Generation in Rhapsody. A Statechart is translated into program code using the following informal programming language independent semantics:

1. Hierarchical Statecharts are flattened and represented by constants in the synthesized program code. The current state of a Statechart is referred to as active state and determined by the value of an additional variable in the generated code.
2. A state can have an entry and an exit action. Enter actions are always executed when the state is entered, after the transition has been fully processed. Exit actions are executed before the next transition is taken.
3. Transitions can have actions. These actions are always executed at the end of the transition to avoid reading inconsistent values.
4. If a substate is active, then its parent state is also active. The event scheduler first tries to send a received event to the active substate. If the event cannot be consumed by that substate, the scheduler tries to send it to the active parent state. This means, if there is more than one transition enabled on different hierarchical levels the lowest level is selected for execution. We call this scheduling strategy *bottom-up priority*. As mentioned before, Rhapsody uses this strategy.
5. When a parent state is left, all exit actions of possible active substates have to be executed as well.

3.3 Step Construction Semantics

We adapt the step construction algorithm from [25] and change it to meet the code generation semantics discussed above.

Definition 3 (Configuration).
A configuration C_i *of a state chart* SC_i *is a set of states that fulfills the following conditions:*

- $\mathrm{root}_i \in C_i$*, the initial state is part of the configuration.*
- *If an OR-state is in* C_i*, then exactly one of its children is in* C_i*.*
- *If an AND-state is in* C_i*, then all of its children are in* C_i*.*

The set of all configurations of SC_i is denoted $conf_i$. The state of SC_i is a tuple (C_i, I_i, V_i) where C_i is a configuration, $I_i \subseteq Events_i$ is a set of events and V_i is a valuation of all variables. An edge between different global states may depend on a guard g. The validity of g depends on the configurations $C_1, \ldots, C_n$ and the valuations $V_1, \ldots, V_n$ of all Statecharts $SC_1, \ldots, SC_n$. We write $g \models (C_{1\ldots n}, V_{1\ldots n})$ *iff* g holds in a state.

In order to match the flattening strategy for hierarchical states used by Rhapsody we use integer constants to identify each state and introduce variables to indicate the active states and substates of a Statechart. Each OR-State is labeled by one additional variable to indicate the active substate. Each AND-state is labeled by as many new variables as the number of its substates. A configuration represents a set of variable valuations over the newly introduced variables. We give an example for this construction in the appendix.

For now, we will only use P-transitions to show the step construction. In Section 3.4 we will show how to translate R-transitions into P-transitions by means of an approximation. As defined above a P-transition is a tuple $t = (X, e, g, P)$. t is enabled if the current configuration C_i contains the source state X, the event e is in the current input set I_i and the guard g holds. We denote the set of enabled transitions by $En(C_i, I_i, V_i)$.

A step is a set of transitions which are concurrently enabled. Which transitions are included in a step depends on the current state (C_i, I_i, V_i) and the guards. A step is subject to four principal constraints:

1. All transitions in a step must be enabled. In the special case where no transition is enabled the step is empty and, thus, represents a deadlock in the system.
2. All transitions must be pairwise consistent which means that they are either identical or their scopes are in different children of AND-states or their descendants.
3. If there are two enabled and pairwise consistent transitions t_1 and t_2 and if $t_2 \in \mathit{children}(t_1)$, then only t_2 is included in the step. In this case t_2 has a higher *priority* than t_1.
4. A step must be maximal. This means that by adding any other transition one of the conditions above is violated. In some cases there might not be a unique maximal step which means that the system is under-specified, i.e., *abstraction* non-deterministic.

If it is not possible to generate a unique maximum it is not possible to generate program code in Rhapsody. It is, nevertheless, possible to conduct a verification of the system since every possible outcome of a non-deterministic decision will be checked by the model checker.

Our step construction algorithm is an adapted version from [25]. We only support the original UML Statechart semantics [1] with the defined bottom-up priority. Notice that we do not have to support the original STATEMATE semantics [21] which assumes a top-down priority for transitions. Thus, we do not have to deal with the ordering problem of probabilities and non-deterministic choices mentioned in [25]. Together with the constraints on P-transitions, we can reduce the algorithm proposed in [25] to a form consistent with the program code generation in Rhapsody.

Step Construction Algorithm

1. Calculate the set of enabled P-transitions $En(C_i, I_i, V_i)$ with respect to all guards.

2. Calculate the set of *steps* possible from $En(C_i, I_i, V_i)$, where *steps* contains the set of enables transitions meeting the four conditions from above.
3. Choose non-deterministically an element from *steps*.

If in Step (3) there is more than one element in *steps*, then the system is non-deterministic. Notice that the code generator in Rhapsody ignores our stereotypes for probability distributions on transitions. Therefore, the engineer has to select the target state for a probability distribution that shall be used when code is generated. This means that the probabilistic behavior introduced into the UML Model does not change the meaning of the code, but only adds verification possibilities.

We will now define the step execution using the semantics for Markov Decision Processes with respect to the code generation semantics.

Semantics for MDPs. We use a definition that is similar to the one presented in [25]. For a finite set of n Statecharts an MDP is defined by:

- The set of states is constructed from the configurations:

$$S = \mathop{\times}_{i=1}^{n} (\mathit{conf}_i \times \mathcal{P}(E_i) \times \mathit{Val}_i)$$

  where E_i are the events available in a state and Val_i is the valuation of all variables in SC_i.
- The set of action is defined by $\mathit{Act} = \mathcal{P}\left(\bigcup_{i=1}^{n} E_i\right) \setminus \emptyset$ which are the possible events consumable in a state.
- The transition probability function: Let $t_i \in \mathit{steps}((C_i, I_i, V_i))$ a possible step for the state $X_i = (C_i, I_i, V_i)$ in the Statechart SC_i. e is the event chosen non-deterministically for t_i to happen. g is the guard valid in X_i and $\mathbf{P}_{X_i}(a, Y_i)$ is the probability that the target state Y_i is selected. The following SOS rule [33], which needs to be instantiated for every i with $1 \leq i \leq n$, formalizes the transition probability function in the MDP:

$$\frac{\begin{array}{c} X_i = (C_i, I_i, V_i), Y_i = (C'_i, I'_i, V'_i), t_i \in \mathit{steps}(X_i), \\ t_i = X_i \xrightarrow[a]{e[g]} Y_i, \mathbf{P}_{X_i}(a, Y_i) = \mu_i \end{array}}{\mathbf{P}((X_1, \ldots, X_n) \xrightarrow{\alpha} (Y_1, \ldots, Y_n)) = \mu}$$

  where $\alpha = \bigcup_{i=1}^{n}\{e | \exists (X, e, g, a, Y) \in t_i : e \in \mathit{Act}\}$ is the set of events received in this step and $\mu(\alpha, (Y_1, \ldots, Y_n)) = \prod_{i=1}^{n} \mu_i$ is the probability of getting into the global target state $(Y_1, \ldots, Y_n)$ with action α.
- The set of atomic propositions is $AP = \bigcup_{i=1}^{n}\{SC_i.\mathit{active\_state}(x) | x \in S_i\}$, where S_i are the states of SC_i and x is the currently active state in SC_i. In the generated code the atomic propositions are all possible valuations of the additional variable introduced to indicate the currently active state.
- The labeling function L is defined by the following rule:

$$\frac{x \in C_i, \text{ for some } i \in \{1, \ldots, n\}}{SC_i.\mathit{active\_state}(x) \in L((C_1, I_1, V_1), \ldots, (C_n, I_n, V_n))}$$

In the generated MDP, the currently active state is labeled by the current valuation of the variables indicating all active (sub-)states which were mentioned in the step construction.

- The initial state s_0 is the initial configuration of $\{root_i\}$ and V_0, i the initial valuation of the variables in the Statechart SC_i.

Without giving a formal proof we maintain that our semantics coincides with the code generation semantics of Rhapsody. We can give an informal correctness argument following the following steps:

1. All probabilistic choices in the MDP are replaced by non-deterministic choices.
2. When editing the probabilistic decisions in the UML / SysML model using QuantUM, the engineer can decide which outcomes correspond to normal behavior, and which ones correspond to failure behavior. Rhapsody will only generate code for the normal behavior. It is therefore pre-determined which non-deterministic decisions described in the first step will be used for code-generation.
3. Using the previous two steps the MDP has been turned into a Kripke structure, for a similar argument see [9].
4. Assuming that the UML model does not contain any *abstraction* based non-determinism, the resulting Kripke structure is deterministic and code can be directly synthesized from it.

This illustrates that the synthesized PRISM code can be mapped back to the program code synthesized from the Rhapsody model.

An example for the mapping from UML to C-program code and MDPs specified in the PRISM language is given in the appendix.

3.4 Approximation of Rates using Geometric Distributions

We will now show an approximation approach which can be used to convert the continuous rates from the R-transitions into discrete probabilities [35]. We will use the notations presented in [20]. The probability $\mathbf{P}$ that a transition with rate λ is taken within time t is computed using the following integral:

$$\mathbf{P}(X \leq t) = \int_0^t e^{-\lambda \cdot t} = 1 - e^{\lambda \cdot t}$$

where X is the random variable for the time with a value less or equal than t. The approximation of the exponential distribution can be done with a discrete geometric distribution of the form:

$$\mathbf{P}(X \leq k) = 1 - \mathbf{P}(X > t) = 1 - (1-p)^k$$

where k is the number of discrete steps in the model and $p = \delta t \cdot \lambda$ the approximated probability for the R-transition to happen in the time step δt. Accordingly, $1 - p$ denotes the approximated probability of the transition not being executed within δt. A simplified version of the proof in [35] can be given as follows [20]:

$$1 - e^{-\lambda \cdot t} = 1 - lim_{n \to \infty} \left(1 + \frac{-\lambda \cdot t}{n}\right)^n = 1 - lim_{n \to \infty} \left(1 - \frac{\lambda \cdot k \cdot \delta t}{n}\right)^n \approx 1 - (1 - \lambda \delta t)^k$$

The relative approximation error can be computed by [35,20]:

$$\varepsilon(t) = \left| \frac{(1 - e^{-\lambda \cdot t}) - (1 - (1 - \lambda \delta t)^k)}{(1 - e^{-\lambda \cdot t})} \right|$$

To compute the overall relative approximation error we have to determine the smallest rate in the model and calculate the error with the given formula. We know all failure rates of the model and we can determine the discretization step before building the model. This means it is possible to determine the approximation error in advance of the model checking. This information helps to choose an adequate step size so that target probabilities given by statutory conditions lie above the probability computed by the model checking run plus the determined approximation error. Furthermore, the user can choose a coarse step size in order to conduct a very rough, fast analysis, for instance in order to compare different architectures. Alternatively, the engineer can choose a fine grained step size in order to do a more precise analysis to be used, for instance in the course of a certification of the system.

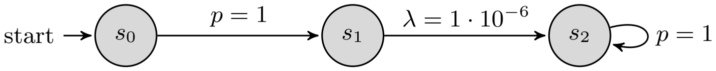

Fig. 3. Example system with a rate on transition $s_1 \to s_2$

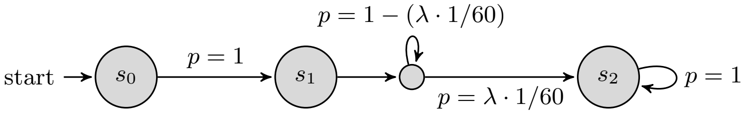

Fig. 4. Example MDP with a discretized rate on transition $s_1 \xrightarrow{\lambda} s_2$

Example. In Figure 3 a system is depicted which consists of two P-transitions and one R-transition with the rate $\lambda = 1 \cdot 10^{-6}$. To turn the system into an MDP the rate has to be discretized. We choose a step size $\delta t = 1/60$. By applying the discretization method we get the discretized version of the MDP presented in Figure 4. Notice that the pseudo state between s_1 and s_2 is only used to implement the delay of the original R-transition. It does neither add nor remove any behavior to or from the system, respectively. The self-loop on the pseudo state represents the non-occurrence of the event within the last discrete time step while the transition from the pseudo state to s_2 shows the occurrence of the event. The relative error introduced by this approximation for a runtime of $t = 1$h and a discretization step size of $\frac{1}{60}$h is $\varepsilon(t) = 8.33 \cdot 10^{-09}$.

4 Case Studies

4.1 Airbag System

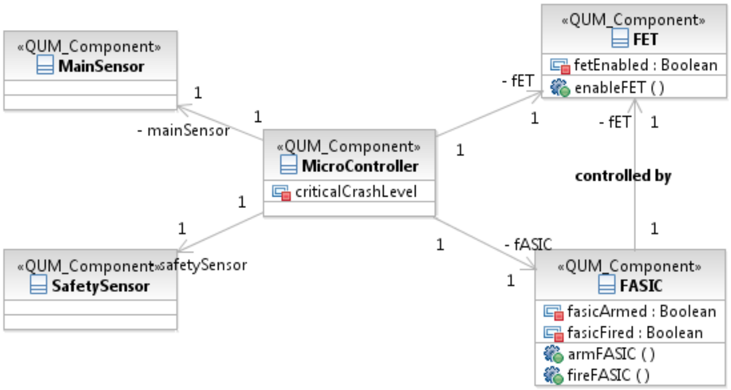

Fig. 5. The UML class diagram of the architecture of the AECU.

The case study of an Airbag Electronic Control Unit (AECU) system from the automotive domain that we consider here was first presented in [5].The architecture of the system was modeled in the UML. The architecture of the airbag system is depicted in Fig.5. The airbag system consists of two acceleration sensors whose task it is to detect front or rear crashes, one micro controller to perform the crash evaluation, and an actuator that controls the deployment of the airbag. Although airbags save lives in crash situations, they may cause fatal behavior if they are deployed inadvertently. Therefore, we will check the system for the probability of an inadvertent deployment of the airbag within 100 hours.

4.2 Airport Surveillance Radar

The Airport Surveillance Radar (ASR) is developed at EADS Cassidian[2]. The high-level architecture of the physical components of the ASR system together with high-level behavior of the embedded software were specified by a model in the SysML and first presented in [10]. The ASR monitors the airspace in the vicinity of an airport. It is used by air traffic controllers who guide aircraft according to their flight plan. The system consists of two redundant channels with 5 components in each channel as well as two different radar units:

1. The Primary Surveillance Radar (PSR) which uses radar signals that are reflected from the surface of an object, which in the normal case is an aircraft that is to be tracked, in order to locate the object.

[2] EADS Deutschland - Cassidian: `http://www.cassidian.com/`

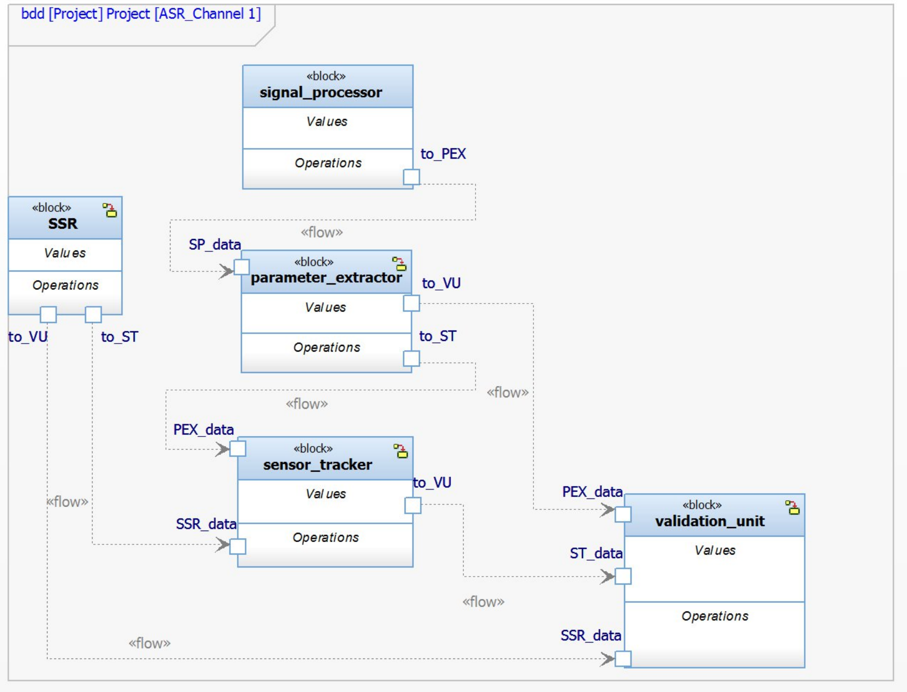

Fig. 6. Overview block-definition-diagram of the ASR system.

2. The Secondary Surveillance Radar (SSR) which communicates with the transponder of the aircraft to identify the aircraft using the transponder code. Furthermore, the SSR is used to determine, among others, the location to confirm the findings of the PSR.
3. The internal structure of the ASR consists of two identical processing channels. Each channel processes the data provided by the PSR and the SSR and creates tracks representing the flight paths of the aircraft:
   (a) A *receiver* that receives the analog PSR signal.
   (b) A *signal* processor which converts the analog PSR signal into a digital signal.
   (c) A *parameter extractor* that extracts plots from the digital signal.
   (d) A *sensor tracker* which takes the SSR signal and the plots generated by the *parameter extractor* and combines them into a track representing the flight path of an aircraft.
   (e) A *validation unit* that consolidates the tracks generated by the sensor tracker.

The architecture of one channel of the ASR system is depicted in Figure 6. Even though a complete failure of the system is highly critical for the safety of the system, the displaying of wrong data to the air traffic controller is considered to be the most critical hazard that can occur. Therefore, we analyze the probability of wrong information being displayed to the air traffic controller within 1 hours.

Similarly to the AECU system, the probabilities and rates used in this study are not the real numbers, since the actual values are intellectual property of the corresponding companies.

5 Experimental Evaluation

After the models are processed by QuantUM they are automatically converted into the PRISM input language containing the MDP model as well as PCTL formulae automatically generated by QuantUM. All experiments were performed on an Intel i7 processor with 3.3Ghz and 24GB of RAM. In the introduction we discussed the three forms of non-determinism occurring in QuantUM models. In both case studies all three forms of non-determinism are present. The UML / SysML models all contain failure descriptions as well as environmental behavior interacting with the system, concurrent system components, and in both case studies we used non-deterministic choices as abstraction in order to reduce the state space of the models.

	CTMC ($\lambda = 1.0$)	CTMC ($\lambda = 100.0$)	MDP (non-det.)
Airbag (time)	0.1 sec.	258.1 sec.	3.94 sec.
Airbag (prob.)	$2.0 \cdot 10^{-4}$	$2.7 \cdot 10^{-4}$	$9.98 \cdot 10^{-4} (\pm 8.33 \cdot 10^{-6}\%)$
ASR	22.57 min	68.88min	277.27min
ASR	$8.8 \cdot 10^{-22}$	$8.231 \cdot 10^{-20}$	$4.81 \cdot 10^{-13} (\pm 1.39 \cdot 10^{-7}\%)$

Table 1. Computation times and probabilities for the two case studies with different rates and the new non-deterministic approach.

Analysis of the AECU System. The PRISM MDP model of the AECU contains 8019 states and 49959 transitions. The discretization step length was set to 1 minute per step. The property we analyze is the probability of an inadvertent deployment of the Airbag within 100 hours. The computed maximum failure probability is $9.98 \cdot 10^{-4}$ and the time needed to compute this probability is 3.94 seconds. The memory used is 818.9KB. The relative error for the probability is $8.33 \cdot 10^{-6}\%$ which is a significantly number of magnitudes smaller than the calculated failure probability. Since we are computing probabilities on non-deterministic models, minimum probabilities have to be discussed as well. In our models the minimum failure probability is always 0 which means no failure is occurring. We will, thus, not discuss this case further. A comparison of the new method to the former approach is shown in Table 1. We have analyzed the properties in three different settings. First we used the CTMC based approach attaching a rate $\lambda = 1$ to all non-stochastic transitions (first column). The repetition of the experiment with $\lambda = 100$ shows that the overall probability raises as λ increases. Notice that the computation time for properties in CTMCs depends exponentially on the difference of the smallest and the largest rate. In the experiments this can be clearly seen by comparing the first with the second column. This means that the further λ increases, the higher the computation time will be.

Analysis of the ASR System. The PRISM MDP model of the ASR system consists of approximately 196 million states and 2.1 billion transitions. The discretization step size is set to 10 seconds per time step. The property we analyze is the probability of wrong information being displayed to the air traffic controller within 1 hour. The computed maximum failure probability is $4.81 \cdot 10^{-13}$ and it takes 277.27 min to calculate the results. The memory consumption is 4.8GB. The relative error for the resulting probability is $1.39 \cdot 10^{-7}\%$. The longer runtime for the ASR model can be explained with the bigger size and the higher precision with which the probability was computed compared to the Airbag model. The comparison of the new results to the former approach in Table 1 shows that the relative difference between the probabilities computed with the CTMC based approach and the new MDP approach is higher than for the airbag system. This can be explained with the higher complexity of the ASR system and the resulting much longer error paths. As explained in Section 2.4, when using the CTMC approach the probability of a path decreases with the length.

Summary. As shown in Table 1 the CTMC based approach does not give reliable results when modeling non-deterministic or non-stochastic behavior. The analysis of the ASR system showed that the difference between the first and the second column lies within two orders of magnitude. With the new MDP based approach it is possible to calculate probabilities within a reasonable amount of time even if analysis time grows with the size of the model and a more fine grained discretization step size. The analysis of the airbag model with the MDP based approach showed that with the rough discretization step size of 1 minute the calculation of the probabilities is almost as fast as it is possible with the CTMC based approach. The relative computation error is low even for the rough discretization step size.

6 Related Work

In [17], Debbabi et al. present a method for automated model checking of SysML activity diagrams. In [31] Majzik et al introduce a UML profile to verify UML diagrams. Additionally, in [11] an approach is presented to check dependability properties on UML models using the MARTE profile[3]. These approaches are using Timed Petri Nets to verify the models. Timed Petri Nets (TPNs) support non-determinism by definition and, thus, are suitable to model concurrent systems. Although, disadvantage of these techniques is that there is no automated tool support for building the TPNs. As a consequence a manual conversion of the model is necessary which is an error prone process.

[3] `http://www.omgmarte.org/`

In [25] Jansen proposes a stochastic extension of Statecharts. The transitions in the Statecharts are tagged with rates and the system is converted into alternating probabilistic transitions systems, a sub-set of MDPs also supporting non-determinism. With this approach the behavior of a system can be analyzed while the structure is disregarded for the verification. In QuantUM we want to be able to analyze the structure, represented for example in UML Class-Diagrams, as well which means that we can not use the approach by Jansen directly. For Jansen's approach there is no tool support available, which means the building of the transition system is a manual process. A manual process is prone to introduce errors in the system.

In [20] the authors present the formal modeling framework SAML. The framework allows the tool-independent specification of formal models for a model-based safety analysis. SAML uses finite state automata with discrete-time and non-deterministic steps (MDP). Nevertheless, continuous-time rates are supported by applying the approach presented in [35]. A disadvantage of the SAML approach is that models have to be specified manually in the SAML language which makes the process error prone. Another drawback is the failure description of systems modeled in SAML. They only consider the possibility of a component error in general but not a specific failure behavior which can interact with the rest of the system. Additionally, probabilistic as well as stochastic transitions cannot occur in the normal behavior of the system, because they are limited to pre-defined failure behavior patterns.

Other formal verification frameworks are the Arcade [12] framework for dependability analysis, the COMPASS [13] framework for formal analysis of aerospace systems and NuSMV3 [14] for analyzing requirements and hybrid systems using SAT solving and the symbolic model checker NuSMV. These approaches are combining formal verification tools in one framework providing a unified input language. However, the models have to be built manually without any automatic translation from SysML or UML models which is what we try to achieve with our QuantUM framework.

7 Conclusion

In this paper we have extended the QuantUM approach with a new analysis for non-deterministic systems. We have defined a formal semantics definition for the automatic translation of UML / SysML models into MDPs which coincides with the program code generation of IBM Rational Rhapsody, one representative of the class of UML / SysML CASE tools used in industrial practice. In two industrial strength case studies we showed that the new semantics definition models the systems more adequately than the former approch using CTMCs already implemented in QuantUM.

In future work we plan to extend the automatic fault tree generation technique of QuantUM to the new non-deterministic approach presented in this paper. Currently the scalability of the fault tree generation method is limited for large models like the ASR. A new approach described in [30] improves the scalability of the fault tree generation for CTMCs significantly. In future work we plan to extend the approach from [30] to MDPs.

References

1. Unified modelling language, specification 2.4.1 (2010), http://www.uml.org/
2. Xml metadata interchange, specification 2.4.1 (August 2011), http://www.omg.org/spec/XMI/
3. DO-178C/ED-12C: Software considerations in airborne systems and equipment certification (2012)
4. A. Aziz, K. Sanwal, V. Singhal, R. K. Brayton: Verifying Continuous-Time Markov Chains. In: Proc. of CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer (1996)
5. Aljazzar, H., Fischer, M., Grunske, L., Kuntz, M., Leitner-Fischer, F., Leue, S.: Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples. In: Proc. of QEST 2009. IEEE Computer Society (2009)
6. Aljazzar, H., Leitner-Fischer, F., Leue, S., Simeonov, D.: Dipro - a tool for probabilistic counterexample generation. In: Proceedings of the 18th International SPIN Workshop. LNCS, vol. 6823, pp. 183–187. Springer (2011)
7. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design 15(1), 7–48 (1999)
8. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-Checking Continuous-Time Markov Chains. ACM Trans. Comput. Logic 1(1), 162–170 (2000)
9. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
10. Beer, A., Kühne, U., Leitner-Fischer, F., Leue, S., Prem, R.: Analysis of an Airport Surveillance Radar using the QuantUM approach. Technical Report soft-12-01, Chair for Software Engineering, University of Konstanz (2012), http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-12-01.pdf
11. Bernardi, S., Merseguer, J., Petriu, D.: A dependability profile within marte. Software and Systems Modeling 10, 313–336 (2011), http://dx.doi.org/10.1007/s10270-009-0128-1, 10.1007/s10270-009-0128-1
12. Boudali, H., Crouzen, P., Haverkort, B.R.H.M., Kuntz, G.W.M., Stoelinga, M.I.A.: Architectural dependability evaluation with Arcade. In: Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Anchorage, USA. pp. 512–521. IEEE Computer Society, Los Alamitos (2008)
13. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP. Lecture Notes in Computer Science, vol. 5775, pp. 173–186. Springer (2009), http://dx.doi.org/10.1007/978-3-642-04468-7_15
14. Cimatti, A., Roveri, M., Susi, A., Tonetta, S.: Formalizing requirements with object models and temporal constraints. Software and System Modeling 10(2), 147–160 (2011)
15. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
16. Corporation, I.: Rational rhapsody in c code generation guide (2009), http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/codegenc.pdf

17. Debbabi, M., Hassaïne, F., Jarraya, Y., Soeanu, A., Alawneh, L.: Verification and Validation in Systems Engineering Assessing UML/SysML Design Model. Springer Berlin / Heidelberg (Nov 2010)
18. Engineering, I.C.o.S.: Systems Modelling Language, Specification 1.2 (Jun 2010), http://www.sysml.org/specs
19. Eshuis, R., Wieringa, R.: Requirements-level semantics for uml statecharts. Formal Methods for Open Object-Based Distributed Systems IV pp. 121–140 (2000)
20. Güdemann, M., Ortmeier, F.: Probabilistic Model-Based Safety Analysis. ArXiv e-prints (Jun 2010)
21. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Trans. Softw. Eng. Methodol. 5(4), 293–333 (Oct 1996), http://doi.acm.org/10.1145/235321.235322
22. International, S.: ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (1996)
23. International, S.: ARP4754A: Guidelines for Development of Civil Aircraft and Systems (2010)
24. International Organization for Standardization: Road vehicles – functional safety, ISO 26262 (2011)
25. Jansen, D.N.: Extensions of statecharts : with probability, time, and stochastic timing. Ph.D. thesis, University of Twente, Enschede (October 2003), http://doc.utwente.nl/58230/
26. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Performance Evaluation 68(2), 90–104 (2011), http://www.sciencedirect.com/science/article/pii/S0166531610000660
27. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806, pp. 585–591. Springer (2011)
28. Kwiatkowska, M., Norman, M., Parker, D.: The semantics of the prism language, http://www.prismmodelchecker.org/doc/semantics.pdf
29. Leitner-Fischer, F., Leue, S.: QuantUM: Quantitative safety analysis of UML models. In: Proceedings Ninth Workshop on Quantitative Aspects of Programming Languages (QAPL 2011). EPTCS, vol. 57, pp. 16–30 (2011), http://www.inf.uni-konstanz.de/soft/research/publications/pdf/qapl2011.pdf
30. Leitner-Fischer, F., Leue, S.: Synergy of probabilistic causality computation and causality checking. Technical Report soft-13-01, Chair for Software Engineering, University of Konstanz (2013), available from http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-13-01.pdf
31. Majzik, I., Pataricza, A., Bondavalli, A.: Architecting dependable systems. In: Architecting dependable systems, chap. Stochastic dependability analysis of system architecture based on UML models, pp. 219–244. Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1768179.1768192
32. Neuhäußer, M., Zhang, L.: Time-bounded reachability in continuous-time Markov decision processes. In: Proceedings of the 2010 Seventh International Conference on the Quantitative Evaluation of Systems. pp. 209–218. QEST (Oktober 2010), http://dx.doi.org/10.1109/QEST.2010.47
33. Plotkin, G.D.: A structural approach to operational semantics (1981)
34. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, NY, USA, 1st edn. (1994)

35. Reich, M.: Asymptotical exponentiality and the approximation of queueing distributions for the pattern in random sign chains. Ph.D. thesis, Hannover: Univ. Hannover, Fachbereich Mathematik (Diss.) (2004)
36. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault Tree Handbook (2002), http://handle.dtic.mil/100.2/ADA354973

A Appendix

In this explanatory appendix we give a example for the translation of a Statechart (Figure 7) into C-code (Listing 1.3) and PRISM code (Listing 1.2).

A.1 Example for the Translation into Program and PRISM Code

In Figure 7 the UML Statechart of the micro controller for the airbag system from the case studies is presented. The Statechart consists of the three states *Idle, EvaluateCrash* and *Crash*. The states *EvaluateCrash* and *Crash* are OR-states which in turn consist of 6 and 3 substates, respectively. The Statechart first is in the *Idle* state. When the trigger event passedtime_20ms is received the microcontroller changes its active state to *EvaluateCrash* which activates the OR-state. When evaluating a situation the micro controller checks whether the acceleration provided by the MainSensor is high enough for a critical situation. If this is the case another check is conducted which checks for the SafetySensor also providing a critical acceleration. If the second checks passes the micro controller sets the criticalCrash variable to `true` and goes into the *EvaluationDone* state. If no critical acceleration is detected the micro controller directly goes into the *EvaluationDone* state. If in an evaluation cycle a crash situation is detected the only transition that is enabled when in the *EvaluationDone* state is the parent transition from *EvaluateCrash* to *Crash*. When the micro controller is in the *Crash* state the firing sequence is initiated and the airbag is deployed.

In Listing 1.3 the generated C-code from Rhapsody is presented. We only give a exemplary part of the code due to its overall size. The PRISM code generated by QuantUM is shown in Listing 1.2. We will now compare the three example state implementations shown in the C-code part. Notice, that in the C-code the current active state is checked through a `switch / case` block (Listing 1.3, line 3):

***Idle* state.** If the system is in the *Idle* state it has to receive the passedtime_20ms to do the next evaluation cycle. In the C-code (line 6-10) we can see that if the variable NormalOperation_active has the value *Idle* which represents the *Idle* state. If an event of the type passedtime_20ms is available, the method which handles the entry procedure for the OR-state *EvaluateCrash* is called and the event is set to be consumed.

The corresponding PRISM code (Listing 1.2, line 20-21) shows that first the root state of *EvaluateCrash* is activated and afterwards the state *Evaluation* is activated. Since, *Evaluation* has no entry action there is no more code needed to represent this transition.

***EvaluationDone* state.** As explained above there are two possible outgoing transitions in the *EvaluationDone* state. In the first case the variable criticalCrash is set to `false` (Listing 1.3, line 18-24). Here the parent state *EvaluationDone* is left and *Idle* is the new active state. If in the other case criticalCrash is set to `true`, the state *EvaluateCrash* is left via the parent transition to *Crash*. The dedicated function `EvaluateCrash_handleEvent()` (Listing 1.3, line 57-72) handles the leaving of the *EvaulateCrash* state by executing all exit actions (line 64) and executing the entry actions of the *Crash* state which includes the sending of the armFasic event (line 65).

The corresponding PRISM code (Listing 1.2, line 22-24 and 49-50), implements the explained behavior. First the code checks whether the criticalCrash is set to `true` (line 23) and then executes the entry action of the *ArmFasic* state (line 50). This action contains a sending of the event armFasic.

***ArmFasic* state.** If the firing sequence of the airbag is initiated first the *amFasic* event is sent in the entry action of the *ArmFasic* state. When leaving this state (Listing 1.3, line 39-40) the entry action of the *EnableFet* state is executed (line 42) which consists of the sending of the enableFet event. The corresponding PRISM code for this state can be found in Listing 1.2 line 51-52.

```
// discretization step size 1 minute
const double TIME_STEP = 1/60;

module MicroController

     // variables
     MicroController_criticalCrashLevel: [0..4] init 0;
     MicroController_criticalCrash: bool init false;

     // active (sub-)states
     NormalOperation_active: [0..18] init 0;

     // additional variables for message handling
     fireFASIC_event: bool init false;
     armFASIC_event: bool init false;
     enableFET_event: bool init false;

     // normal behavior of MicroController
     [](NormalOperation_active = 0) -> ( NormalOperation_active '= 1); //ROOT -> Idle
     [passedtime_20ms](NormalOperation_active = 1)-> ( NormalOperation_active '= 3); //Idle -> EvaluateCrash_ROOT
     [](NormalOperation_active = 3) -> ( NormalOperation_active '= 4); //EvaluateCrash_ROOT -> Evaluation
     [](NormalOperation_active >= 2) & (NormalOperation_active <= 9)
        & (  MicroController_criticalCrash = true )
        -> ( NormalOperation_active '= 11); //Evaluate Crash -> Crash_ROOT

     // sub-Statechart of EvaluateCrash
     [](NormalOperation_active = 4)  & (  MainSensor_acceleration < 3 )
        -> ( NormalOperation_active '= 9) & (MicroController_criticalCrashLevel '=0); //Evaluation -> NotCritical
     [](NormalOperation_active = 4)  & (  MainSensor_acceleration >= 3 )
        -> ( NormalOperation_active '= 5); //Evaluation -> MainSensorCritical
     [](NormalOperation_active = 5)  & (  SafetySensor_acceleration >= 3 )
        & (MicroController_criticalCrashLevel <= 4- 1)
        & (MicroController_criticalCrashLevel >= 0)
         -> ( NormalOperation_active '= 6) &
            (MicroController_criticalCrashLevel '=MicroController_criticalCrashLevel+1);
           //MainSensorCritical -> SafetySensorCritical
     [](NormalOperation_active = 5)  & (  SafetySensor_acceleration < 3 )
         -> 1.0: ( NormalOperation_active '= 8); //MainSensorCritical -> EvaluationDone
     [](NormalOperation_active = 6)  & (  MicroController_criticalCrashLevel >=3 )
        -> ( NormalOperation_active '= 7) & (MicroController_criticalCrash '=true); //SafetySensorCritical -> Crash
     [](NormalOperation_active = 6)  & (  MicroController_criticalCrashLevel < 3 )
        -> ( NormalOperation_active '= 8); //SafetySensorCritical -> EvaluationDone
     [](NormalOperation_active = 7) -> ( NormalOperation_active '= 8); //Crash -> EvaluationDone
     [](NormalOperation_active = 8)  & (  MicroController_criticalCrash = false )
        -> ( NormalOperation_active '= 1); //EvaluationDone -> Idle
     [](NormalOperation_active = 9)-> ( NormalOperation_active '= 8); //NotCritical -> EvaluationDone

     // sub-Statechart of Crash
     [](NormalOperation_active = 11)
        -> ( NormalOperation_active '= 12) & (armFASIC_event '=true); //Crash_ROOT -> ArmFasic
     [](NormalOperation_active = 12)
        -> ( NormalOperation_active '= 13) & (enableFET_event '=true); //ArmFasic -> EnableFet
     [](NormalOperation_active = 13)
        -> ( NormalOperation_active '= 14) & (fireFASIC_event '=true); //EnableFet -> FasicFire

     // event handling
     [fireFASIC](fireFASIC_event=true) -> 1.0: (fireFASIC_event '=false);
     [armFASIC](armFASIC_event=true) -> 1.0: (armFASIC_event '=false);
     [enableFET](enableFET_event=true) -> 1.0: (enableFET_event '=false);

     //failure pattern
     [] (NormalOperation_active = 16)-> ( NormalOperation_active '= 17) & (armFASIC_event '= true);
     [] (NormalOperation_active = 17)-> ( NormalOperation_active '= 18) & (fireFASIC_event '= true);
     [] (NormalOperation_active = 15)-> ( NormalOperation_active '= 16) & (enableFET_event '= true);
     [] (NormalOperation_active >= 0 & NormalOperation_active < 15)
        ->       (1E-5*TIME_STEP): ( NormalOperation_active '= 15)
            + 1-(1E-5*TIME_STEP): ( NormalOperation_active '= 19);
     // selfloop for failure transition
     [] (NormalOperation_active = 19)
        ->       (1E-5*TIME_STEP): ( NormalOperation_active '= 15)
            + 1-(1E-5*TIME_STEP):true;



endmodule
```

Listing 1.2. The generated PRISM code for the Microcontroller of the Airbag System in Figure 7

```
IOxfReactive::TakeEventStatus MicroController::NormalOperation_processEvent() {
    IOxfReactive::TakeEventStatus res = eventNotConsumed;
    switch (NormalOperation_active) {
        case Idle:
        {
            if(IS_EVENT_TYPE_OF(passedtime_20ms))
                {
                    EvaluateCrash_entDef();
                    res = eventConsumed;
                }
        }
        break;
        case EvaluationDone:
        {
             if(IS_EVENT_TYPE_OF(OMNullEventId))
                {
                //## transition 2
                if(criticalCrash = false)
                    {
                        EvaluateCrash_exit();
                        NormalOperation_subState = Idle;
                        rootState_active = Idle;
                        res = eventConsumed;
                    }
                }

            if(res == eventNotConsumed)
                {
                    res = EvaluateCrash_handleEvent();
                }
        }
        break;
        case ArmFasic:
        {
            if(IS_EVENT_TYPE_OF(OMNullEventId))
                {
                    popNullTransition();
                    pushNullTransition();
                    NormalOperation_Crash_subState = EnableFet;
                    NormalOperation_active = EnableFet;
                    //#[ state ROOT.NormalOperation.Crash.EnableFet.(Entry)
                    enableFET();
                    //#]
                    res = eventConsumed;
                }
        }
        break;
        // rest of the handling code
        .
        .
        .

    }
    return res;
}

IOxfReactive::TakeEventStatus MicroController::EvaluateCrash_handleEvent() {
    IOxfReactive::TakeEventStatus res = eventNotConsumed;
    if(IS_EVENT_TYPE_OF(OMNullEventId))
        {
            //## transition 10
            if(criticalCrash = true)
                {
                    EvaluateCrash_exit();
                    NormalOperation_Crash_entDef();
                    res = eventConsumed;
                }
        }


    return res;
}
```

Listing 1.3. The generated C-code for a Part of the Microcontroller of the Airbag System in Figure 7

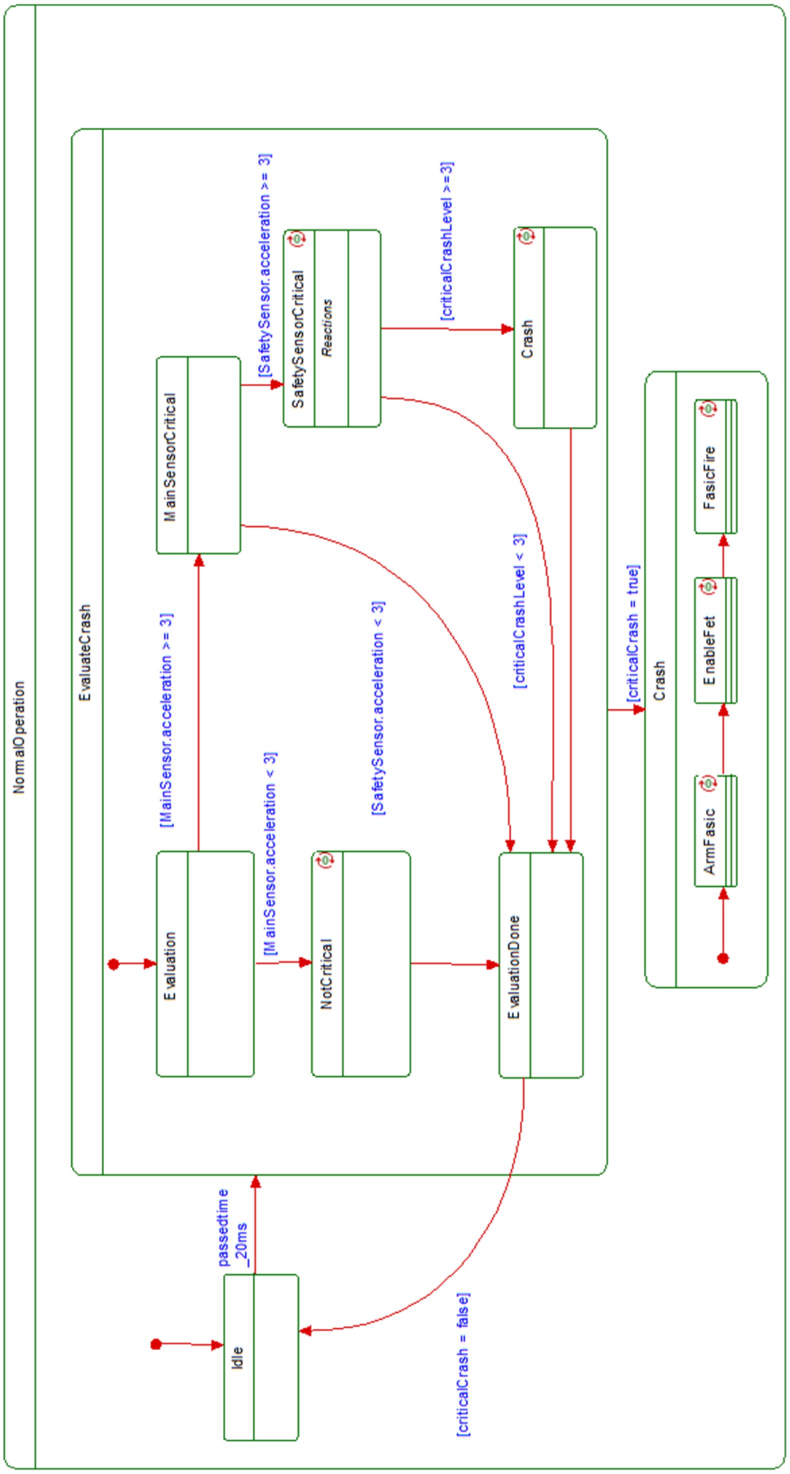

Fig. 7. The UML-Statechart of the airbag microcontroller.