Quantitative Analysis of UML Models

Florian Leitner-Fischer and Stefan Leue

florian.leitner@uni-konstanz.de, stefan.leue@uni-konstanz.de

Abstract: When developing a safety-critical system it is essential to obtain an assessment of different design alternatives. In particular, an early safety assessment of the architectural design of a system is desirable. In spite of the plethora of available formal quantitative analysis methods it is still difficult for software and system architects to integrate these techniques into their every day work. This is mainly due to the lack of methods that can be directly applied to architecture level models, for instance given as UML diagrams. Our approach bridges this gap and improves the integration of quantitative safety analysis methods into the development process. We propose a UML profile that allows for the specification of all inputs needed for the analysis at the level of a UML model. The QuantUM tool which we have developed, automatically translates an UML model into an analysis model. Furthermore, the results gained from the analysis are lifted to the level of the UML specification or other high-level formalism to further facilitate the process. Thus the analysis model and the formal methods used during the analysis are hidden from the user.

1 Introduction

In a recent joint work with our industrial partner TRW Automotive GmbH we have proven the applicability of probabilistic verification techniques to safety analysis in an industrial setting [AFG$^+$09]. The analysis approach that we used was that of probabilistic Failure Modes Effect Analysis (pFMEA) [GCW07].

The most notable shortcoming of the approach that we observed lies in the missing connection of our analysis to existing high-level architecture models and the modeling languages that they are typically written in. TRW Automotive GmbH, like many other software development enterprises, mostly uses the Unified Modeling Language (UML) [Obj10] for system modeling. But during the pFMEA we had to use the language provided by the analysis tool used, in this case the input language of the stochastic model checker PRISM [HKNP06]. This required a manual translation from the design language UML to the formal modeling language PRISM. This manual translation has the following disadvantages: (1) It is a time-consuming and hence expensive process, (2) It is error-prone, since behaviors may be introduced that are not present in the original model. And, (3) the results of the formal analysis may not be easily transferable to the level of the high-level design language. To avoid problems that may result from (2) and (3), additional checks for plausibility have to be made, which again consume time. Some introduced errors may even remain unnoticed.

The objective of this paper is to bridge the gap between architectural design and formal stochastic modeling languages so as to remedy the negative implications of this gap listed above. This allows for a more seamless integration of formal dependability and reliability analysis into the design process. We propose an extension of the UML to capture probabilistic and error behavior information that are relevant for a formal stochastic analysis, such as when performing pFMEA. All inputs of the analysis can be specified at the level of the UML model. In order to achieve this goal, we present an extension of the Unified Modeling Language that allows for the annotation of UML models with quantitative information, such as for instance failure rates. Additionally, a translation process from UML models to the PRISM language is defined.

Our approach can be described by identifying the following steps:

- Our UML extension is used to annotate the UML model with all information that is needed to perform the analysis.
- The annotated UML model is then exported in the XML Metadata Interchange (XMI) format [Obj07] which is the standard format for exchanging UML models.
- Subsequently, our QuantUM Tool parses the generated XMI file and generates the analysis model in the input language of the probabilistic model checker PRISM and the properties to be verified in CSL.
- For the analysis we use the probabilistic model checker PRISM together with our extension of PRISM [AL08] that allows for the computation of probabilistic counterexamples.
- The resulting counterexamples can then either be represented as a fault tree that is interpretable on the level of the UML model, or they can be mapped onto a UML sequence diagram which is stored in an XMI file that can be displayed in the UML modeling tool containing the UML model.

All analysis steps are fully automated and do not require user interaction.

Structure of the paper. The remainder of the paper is structured as follows: In Section 2 we present our QuantUM approach. Section 3 is devoted to the case study of an airbag system. Followed by a discussion of related work in Section 4. We conclude in Section 5.

2 The QuantUM Approach

2.1 Extension of the UML

In our approach all inputs of the analysis are specified at the level of a UML model. To facilitate the specification, we propose a quantitative extension of the UML. The annotated model is then automatically translated into the analysis model, and the results of the analysis are subsequently represented on the level of the UML model. The analysis model and the formal methods used during the analysis are hence hidden from the user.

In the following we describe our UML profile for quantitative analysis. The dependability terminology that we used here is based on [ALRL04]. UML models consist of two major parts, the structural and the behavioral description of the system. In order to capture the dependency structure of the model, which allows to express how the failure of one component influences the failure of another component, we need to extend the structural description capabilities of the UML. In addition we need to extend the behavioral description to capture the stochastic behavior. In the following we define the stereotypes and their properties that are used to specify the information needed to perform stochastic analysis.

QUMComponent The stereotype *QUMComponent* can be assigned to all UML elements that represent building blocks of the real system, that is classes, components and interfaces. Each element with the stereotype *QUMComponent* comprises up to one (hierarchical) state machine representing the normal behavior and one to finitely many (hierarchical) state machines representing possible failure patterns. These state machines can be either state machines that are especially constructed for this *QUMComponent*, or they can be taken from a repository of state machines describing standard failure behaviors. The repository provides state machines for all standard components (e.g., switches) and the reuse of these state machines saves modeling effort and avoids redundant descriptions. In addition, each *QUMComponent* comprises a list called *Rates* that contains stochastic rates representing, for instance, failure rates, together with names identifying them.

QUMFailureTransition and QUMAbstractFailureTransition In order to capture the operational profile and particularly to allow the specification of quantitative information, such as failure rates, we extend the *Transition* element used in UML state machines with the stereotypes *QUMAbstractFailureTransition* and *QUMFailureTransition*. These stereotypes allow the user to specify transition rates as well as a name for the transition. The specified rates are used as transition rates for the continuous-time Markov chains that are generated for the purpose of stochastic analysis. Transitions with the stereotype *QUMAbstractFailureTransition* are transitions that do not have a default rate. If a state machine is connected to a *QUMComponent* element, there has to be a rate in the *Rates* list of the *QUMComponent* that has the same name as the *QUMAbstractFailureTransition*, this rate is then considered for the *QUMAbstractFailureTransition*. The *QUMAbstractFailureTransition* allows to define general state machines in a repository where the rates can be set individually for each component.

The normal behavior state machine and all failure pattern state machines are implicitly combined in one hierarchical state machine. The combined state machine is automatically generated by the analysis tool and is not visible to the user. Its semantics can be described as follows: initially, the component executes the normal behavior state machine. If a *QUMAbstractFailureTransition* is enabled, the component will enter the state machine describing the corresponding failure pattern with the specified rate. The decision, which of the n FailurePatterns is selected, is made by a stochastic "race" between the transitions.

QUMStateConfiguration The *QUMStateConfiguration* stereotype can be used to assign names to state configurations. In order to do so, the stereotype is assigned to *states* in the state machines. All *QUMStateConfiguration* stereotypes with the same *name* are treated as one state configuration. A state configuration can also be seen as a boolean formula, each state can either be true when the system is in this state or false, when the system is not in this state. The *operator* variable indicates whether the boolean variables representing the states are connected by an *and*-operator (*AND*) or an *or*-operator (*OR*). The name of the state configuration is in the model checking process used to identify the state configurations.

Due to space restrictions we can not describe the extension in full detail here. In addition to the stereotypes discussed above, we have also defined stereotypes allowing for the specification of repair management, failure propagation and spare handling. For a more detailed description of QuantUM, including a demonstration on how the QuantUM profile can be applied to a given UML model, we refer to [LF10].

2.2 From UML to PRISM

We define the semantics of our extensions by defining rules to translate the UML artifacts that we defined into the input language of the model checker PRISM [HKNP06]. This corresponds to the commonly held idea that the semantics of a UML model is largely defined by the underlying code generator. We base our semantic transformation on the operational UML semantics defined in [LMM+99].

Besides the analysis model, the properties to be analyzed are important inputs to the analysis. In stochastic model checking, the property that is to be verified is specified using a variant of temporal logic. The temporal logic used here is Continuous Stochastic Logic (CSL) [AKVR96, BHHK03]. We offer two possibilities for property specification: first we automatically generate a set of CSL properties out of the UML model, and second we allow the user to manually specify CSL properties. This has the advantage, of supporting users with no or little knowledge of CSL by the automatic generation but still offers experts the full possibilities of CSL. Due to space restrictions we can not present the translation rules and CSL property generation here and refer to [LF10].

3 Case Study: Airbag System

We have applied our modeling and analysis approach to a case study from the automotive software domain. We performed an analysis of the design of an Electronic Control Unit for an Airbag system that is being developed at TRW Automotive GmbH, see also [AFG+09]. Note that the used probability values are merely approximate "ballpark" numbers, since the actual values are intellectual property of our industrial partner TRW Automotive GmbH that we are not allowed to publish.

The airbag system architecture that we consider is depicted in Figure 1. The two acceleration sensors, MainSensor and SafetySensor, measure the acceleration of the car in order to detect front or rear crashes. The acceleration values are read by the microcontroller which performs the crash evaluation. The deployment of the airbag is secured by two redundant protection mechanisms. The Field Effect Transistor (FET) controls the power supply for the airbag squibs. If the Field Effect Transistor is not armed, which means that the FET-Pin is not high, the airbag squib does not have enough electrical power to ignite the airbag. The second protection mechanism is the Firing Application Specific Integrated Circuit (FASIC) which controls the airbag squib. Only if it receives first an arm command and then a fire command from the microcontroller it will ignite the airbag squib.

Although airbags save lives in crash situations, they may cause fatal behavior if they are inadvertently deployed. This is because the driver may lose control of the car when this deployment occurs. It is therefore a pivotal safety requirement that an airbag is never deployed if there is no crash situation. In order to analyze whether the considered system architecture, modeled with the CASE tool IBM Rational Software Architect, is safe or not we annotated the model with our QuantUM extension and performed an analysis with the QuantUM tool.

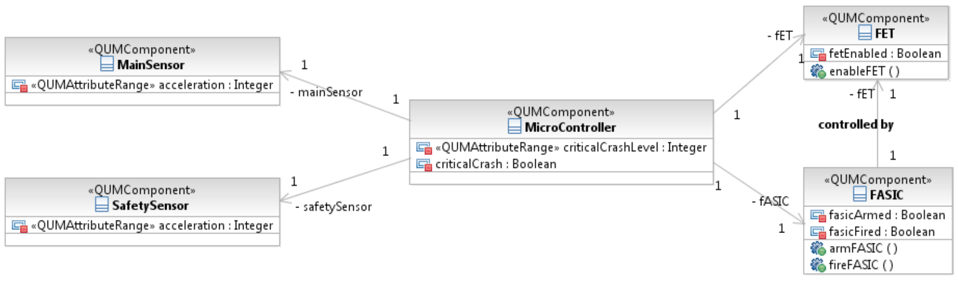

Figure 1: Class diagram modeling the airbag system.

After all annotations were made, we exported the model into an XMI file, which was then imported by the QuantUM tool and translated into a PRISM model. The import of the XMI file and translation of the model was completed in less than two seconds. Without the QuantUM tool, this process would require hours of work of a trained engineer.

The resulting PRISM model consists of 3249 states and 15390 transitions. The QuantUM tool also generated the CSL formula. $P_{=?}[(\text{true})U^{<=T}(\mathit{inadvertent\_deployment})]$ where *inadvertent_deployment* is replaced by the state formula which identifies all states in the *QUMStateConfiguration inadvertent_deployment* and T represents the mission time. The mission time specifies the driving time of the car. Since, the acceleration value of the sensor state machines is always zero, the formula $P_{=?}[(\text{true})U^{<=T}(\mathit{inadvertent\_deployment})]$ calculates the probability of the airbag being deployed, during the driving time T, although there is no crash situation.

Therefore, if the QuantUM tool is used, the only input which has to be given by the user is the mission time T. Whereas, without the QuantUM tool the engineers would also have to specify the CSL formula.

We computed the probability for the mission time T=10, T=100, and T=1000 and recorded the runtime for the counterexample computation (Runtime CX), the number of paths in the counterexample (Paths in CX), the runtime of the fault tree generation algorithm (Runtime FT) and the numbers of paths in the fault tree (Paths in FT) in Figure 2. The experiments where performed on a PC with an Intel QuadCore i5 processor with 2.67 Ghz and 8 GBs of RAM.

T	Runtime CX (sec.)	Paths in CX	Runtime FT (sec.)	Paths in FT
10	646.425 (approx. 10.77 min.)	738	2.86	5
100	664.893 (approx. 11.08 min.)	738	3.52	5
1000	820.431 (approx. 15.67 min.)	738	2.98	5

Figure 2: Experiment results for T=10, T=100 and T=1000.

Figure 2 shows that the computation of the fault tree is finished in several seconds, whereas the computation of the counterexample takes several minutes. While the different running times of the counterexample computation algorithm seem to be caused by the different values of the running time T, the variation of the running time of the fault tree computation seems to be due to background processes on the experiment PC.

Figure 3 shows the fault tree generated from the counterexample for T=10. While the counterexample consists of 738 paths, the fault tree comprises only 5 paths. It is easy to see by which basic events, and with which probabilities, an inadvertent deployment of the airbag is caused. There is only one single fault that can lead to an inadvertent deployment, namely *FASICShortage*. The basic event *MicroControllerFailure*, for instance, can only lead to a inadvertent deployment if it is followed by one of the following sequences of basic events: *enableFET*, *armFASIC*, and *fireFASIC* or *enableFET*, and *FASICStuckHigh*. In the Fault Tree this is expressed by the Priority-AND symbol that connects those events. If the basic event *FETStuckHigh* occurs prior to the *MicroControllerFailure* the sequence *armFASIC* and *fireFASIC* occurring after the *MicroControllerFailure* event suffices.

The case study shows that the fault tree is a compact and concise visualization of the counterexample. It allows for an easy identification of the basic events that cause the inadvertent deployment of the airbag, as well as their corresponding probabilities. If the order of the events is important, this can be represented in the fault tree by the *PAND*-gate. Without this automation one would have to manually compare the order of the events in all 738 paths of the counterexample, which is a tedious and time consuming task.

Figure 4 shows the first part of the UML sequence diagram which visualizes the counterexample for T=10. The generation of the XMI code of the sequence diagram took less then one second. We imported the XMI code into the UML model of the airbag system in the CASE tool IBM Rational Software Architect. This allows us to interpret the counterexample directly in the CASE tool.

An additional benefit of our QuantUM implementation is a visualization of the counterex-

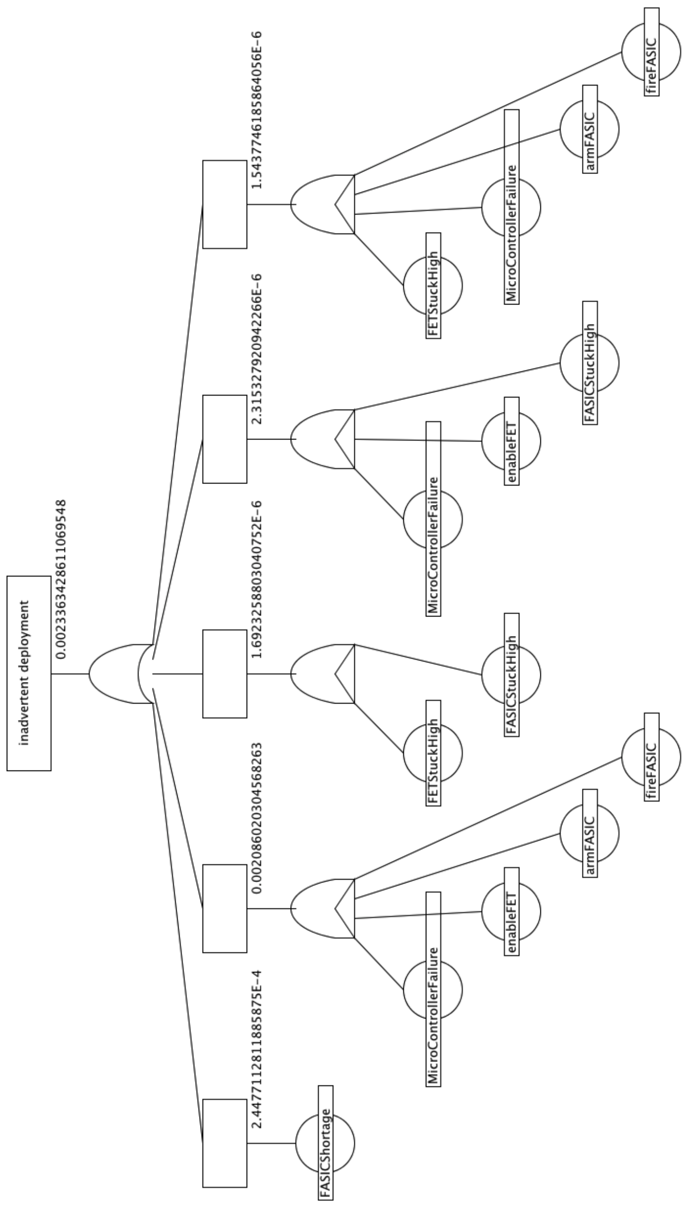

Figure 3: Fault tree for the *QUMStateConfiguration inadvertent_deployment* (T = 10).

ample as sequence diagram in which operation calls can be shown. In the lower *alt*-compartment of the automatically synthesized sequence diagram in Figure 4 for instance, it is easy to see how after a failure of the microcontroller the operations *enableFET()*, *armFASIC()*, and *fireFASIC()* are called.

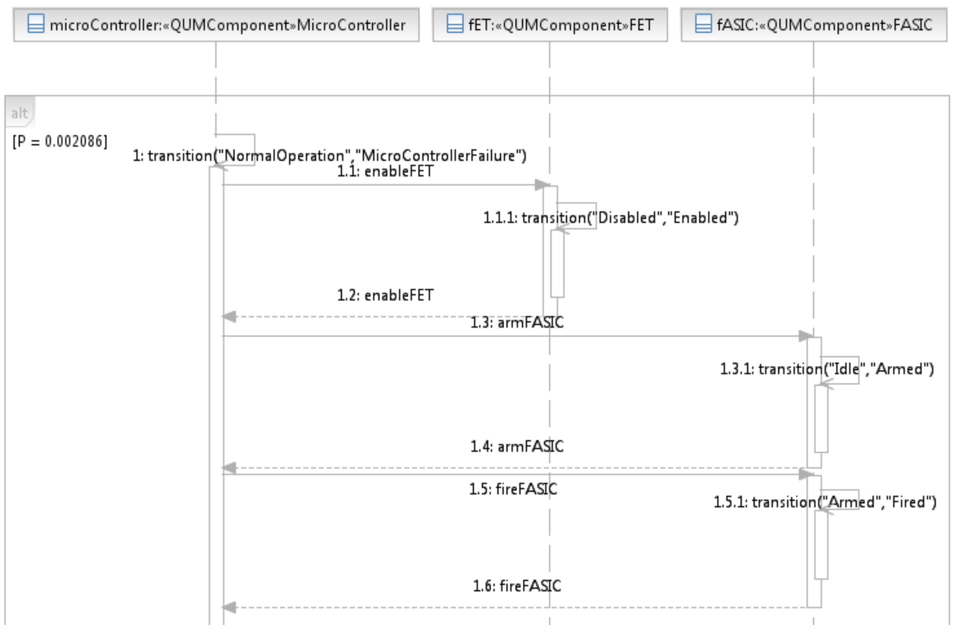
microController:«QUMComponent»MicroController
fET:«QUMComponent»FET
fASIC:«QUMComponent»FASIC
alt
[P = 0.002086]
1: transition("NormalOperation","MicroControllerFailure")
1.1: enableFET
1.1.1: transition("Disabled","Enabled")
1.2: enableFET
1.3: armFASIC
1.3.1: transition("Idle","Armed")
1.4: armFASIC
1.5: fireFASIC
1.5.1: transition("Armed","Fired")
1.6: fireFASIC

Figure 4: First part of the UML sequence diagram for the *QUMStateConfiguration inadvertent_deployment* (T = 10) .

4 Related Work

The idea of using UML models to derive models for quantitative safety analysis is not new. In [MPB03] the authors present a UML profile for annotating software dependability properties. This annotated model is then transformed into an intermediate model, that is then transformed into Timed Petri Nets. The main drawbacks of this work is that it merely focuses on the structural aspect of the UML model, while the actual behavioral description is not considered. Another shortcoming is the introduction of unnecessary redundant information in the UML model, since sometimes the joint use of more than one stereotype is required. In [BMP09] the authors extend the UML Profile for Modeling and Analysis of Real Time Embedded Systems [Obj08] with a profile for dependability analysis and modeling. While this work is very expressive, it heavily relies on the use of the MARTE profile, which is only supported by very few UML CASE tools. Additionally, the amount of stereotypes, tagged values and annotations that need to be made to the model is very large. Another disadvantage of this approach is that the translation from the annotated UML model into the Deterministic and Stochastic Petri Nets (DSPN) [MC87] used for analysis is carried out manually which is, as we argue above, an error-prone and risky task for large UML models. The work defined in [Jan03] presents a stochastic extension of the Statechart notation, called StoCharts. The StoChart approach suffers from the following limitations. First, it is restricted to the analysis of the behavioral aspects of the system and does not allow for structural analysis. Second, while there exist some tools that allow to draw StoCharts, there is no integration of StoCharts into UML models available. In [BCH+08] the architecture dependability analysis framework Arcades is presented. While Arcade is very expressive and was applied to hardware, software and infrastructure systems, the main restriction is that it is based on a textual description of the system and hence would require a manual translation process of the UML model to Arcade.

We are to the best of our knowledge not aware of any approach, that allows for the automatic generation of the analysis model and the automatic CSL property construction.

5 Conclusion

We have presented a UML profile that allows for the annotation of UML models with quantitative information as well as a tool called QuantUM that automatically translates the model into the PRISM language, generates the CSL properties, and automatically performs a probabilistic analysis with PRISM. Furthermore, we have developed a method that allows us to automatically generates a fault tree from a probabilistic counterexample. We also presented a mapping of probabilistic counterexamples to UML sequence diagrams and thus make the counterexamples interpretable inside the UML modeling tool that is being used during design. We have demonstrated the usefulness of our approach on a case study known from the literature. The case study shows that tasks that previously required hours of work of trained engineers, like the translation of an UML model into PRISM or the formulation of CSL formulas, are now fully automated and completed within several seconds.

In future work we plan to extend the expressiveness of the QuantUM profile, to integrate methods to further facilitate automatic stochastic property specification, and to apply our approach on other architecture description languages such as, for instance, SysML.

References

[AFG+09] Husain Aljazzar, Manuel Fischer, Lars Grunske, Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples. In *QEST '09: Proceedings of the Sixth International Conference on Quantitative Evaluation of Systems*, pages 299–308, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[AKVR96] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying Continuous-Time Markov Chains. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102, pages 269–276, New Brunswick, NJ, USA, 1996. Springer Verlag LNCS.

[AL08] H. Aljazzar and S. Leue. Debugging of Dependability Models Using Interactive Visualization of Counterexamples. In *QEST '08: Proceedings of the Fifth International Conference on the Quantitative Evaluation of Systems*, pages 189–198. IEEE Computer Science Press, 2008.

[ALRL04] A. Avižienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, pages 11–33, 2004.

[BCH+08] H. Boudali, P. Crouzen, B.R. Haverkort, M. Kuntz, and M.I.A. Stoelinga. Architectural Dependability Modelling with Arcade. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 512–521, 2008.

[BHHK03] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Transactions on Software Engineering*, 29(7), 2003.

[BMP09] S. Bernardi, J. Merseguer, and D.C. Petriu. A dependability profile within MARTE. *Software and Systems Modeling*, pages 1–24, 2009.

[GCW07] Lars Grunske, Robert Colvin, and Kirsten Winter. Probabilistic Model-Checking Support for FMEA. In *QEST '07: Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems*, pages 119–128, Washington, DC, USA, 2007. IEEE Computer Society.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS '06: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 441–444. Springer, 2006.

[Jan03] David Nicolaas Jansen. *Extensions of statecharts : with probability, time, and stochastic timing*. PhD thesis, University of Twente, October 2003.

[LF10] Florian Leitner-Fischer. Quantitative Safety Analysis of UML Models. Master's thesis, University of Konstanz, 2010.

[LMM+99] D. Latella, I. Majzik, M. Massink, et al. Towards a formal operational semantics of UML statechart diagrams. In *IFIP TC6/WG6*, volume 1, pages 331–347. Citeseer, 1999.

[MC87] M. Marsan and G. Chiola. On Petri nets with deterministic and exponentially distributed firing times. *Advances in Petri Nets 1987*, pages 132–145, 1987.

[MPB03] I. Majzik, A. Pataricza, and A. Bondavalli. Stochastic dependability analysis of system architecture based on UML models. *Architecting dependable systems*, page 219, 2003.

[Obj07] Object Management Group. XML Metadata Interchange (XMI), v2.1.1. http://www.omg.org/technology/documents/formal/xmi.htm, December 2007.

[Obj08] Object Management Group. UML Profile for Modeling and Analysis of Real Time Embedded Systems. http://www.omgmarte.org/, June 2008.

[Obj10] Object Management Group. Unified Modeling Language. Specification v2.3. http://www.uml.org, May 2010.