

University of Konstanz
Department of Computer and Information Science

Master Thesis for the degree
Master of Science (M.Sc.) in Information Engineering

Quantitative Safety Analysis of UML Models

by

Florian Leitner-Fischer

(Matr.-Nr. 01 / 612538)

1st Referee: Prof. Dr. Stefan Leue

2nd Referee: Prof. Dr. Marc H. Scholl

Konstanz, August 2, 2010

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. Another obstacle is, that the methods often require a profound knowledge of formal methods, which can rarely be found in industrial practice. 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.

Kurzfassung

Bei der Entwicklung sicherheits-kritischer Systeme, ist die Bewertung von verschiedenen Entwurfsalternativen von großer Bedeutung. Eine solche Bewertung ist in den frühen Entwicklungsphasen eines Systems besonders hilfreich. Trotz der großen Anzahl vorhandener Methoden zur formalen quantitativen Analyse, ist es für System- und Software-Architekten immer noch schwierig diese in ihrer tägliche Arbeit anzuwenden. Dieser Umstand lässt sich vor allem auf das Fehlen von Methoden, welche man direkt auf die Architekturmodelle anwenden kann, zurückführen. Ein weiterer Hinderungsgrund ist, dass die Methoden sehr häufig ein tiefgreifendes Verständnis von formalen Methoden fordern welches in der industriellen Praxis kaum zu finden ist.

Unser Ansatz schließt diese Lücke und verbessert die Integration von Methoden zur quantitativen Sicherheitsanalyse in den Entwicklungsprozess. Wir schlagen ein UML Profil vor, welches die Spezifikation von allen Eingangsgrößen, welche für die Analyse benötigt werden, auf der Ebene des UML Modells ermöglicht. Die von uns entwickelte QuantUM Software übersetzt das UML Modell automatisch in das Analyse-Modell. Außerdem, werden die durch die Analyse gewonnen Ergebnisse auf die Ebene der UML Spezifikation gehoben um den Prozess weiter zu unterstützen. Demzufolge wird die Analyse-Ebene, welche das Analyse Modell und die verwendeten formalen Methoden einschließt, vom Benutzer verborgen.

Acknowledgments

I can no other answer make, but, thanks, and thanks.

- *William Shakespeare* -

Firstly, I want to express my gratitude to my supervisor Prof. Stefan Leue for his outstanding guidance and mentoring, his valuable advice and comments and for giving me the opportunity to work in his research group.

I thank Prof. Marc Scholl for agreeing to be the second referee of this thesis.

I thank the whole software engineering group at the University of Konstanz for providing a great and friendly work climate. Particularly, I want to thank my colleagues Husian Aljazzar, Matthias Kuntz and Nafees Ur Rehmann for the great time we had together. Special thanks go also to Dimitar Simeonov for his work on DiPro which made it easier to integrate it into QuantUM.

I am indebted to my family for the support and encouragement during the last years. Many thanks to my father, who taught me how to think like an engineer and for the often endless discussions of my ideas. Many thanks go also to my mother, who had to listen to this discussions too many times.

My special gratitude goes to Viktoria for her love and understanding and for her continuous motivation.

Contents

Abstract / Kurzfassung	3
Acknowledgments	5
1 Introduction	11
1.1 Introduction	11
1.2 Contributions	13
1.3 Structure of the Thesis	13
2 Foundations	15
2.1 Unified Modeling Language	15
2.2 Probabilistic Model Checking	16
3 Quantitative Extension of UML	17
3.1 Motivation	17
3.2 Extension of the UML	18
3.3 UML-Profile for Quantitative Analysis	19
3.4 Discussion	28
4 From Quantitative UML to PRISM	31
4.1 Semantics of the Extension	31
4.2 Automatic Property Construction	38
5 High-Level Probabilistic Counterexamples	41
5.1 Motivation	41
5.2 Probabilistic Counterexamples	42
5.2.1 Counterexamples in Stochastic Model Checking	42
5.2.2 Notion of Counterexamples	42
5.2.3 Generation of Counterexamples	42
5.3 From Probabilistic Counterexamples to Fault Trees	44
5.3.1 Fault Trees and Fault Tree Analysis	44
5.3.2 Mapping of Counterexamples to Fault Trees	46

5.3.3	Complexity of the algorithm	49
5.3.4	Scalability of the approach	50
5.3.5	Correctness and Completeness of the Approach	50
5.4	From Probabilistic Counterexamples to UML	53
5.4.1	UML Sequence Diagrams	53
5.4.2	Mapping of Counterexamples onto UML Sequence Diagrams	54
6	The QuantUM Tool	57
7	Case Studies	59
7.1	Airbag Control Unit	59
7.2	Train Odometer Controller	68
8	Related Work	83
8.1	QuantUM Approach	83
8.2	Quantitative Extension of UML	83
8.3	Automatic Fault Tree Generation	84
9	Conclusion	87
9.1	Conclusion	87
9.2	Future Work	88
10	Appendix	89
10.1	CD	89
	Bibliography	91

List of Figures

1.1	The QuantUM tool chain.	13
3.1	Definition of the <i>QUMComponent</i> stereotype	20
3.2	Class diagram of the light controller system.	21
3.3	Definition of the <i>QUMAttributeRange</i> stereotype	21
3.4	Definition of the stereotypes for abstract and concrete stochastic transitions.	22
3.5	Combination of normal behavior and failure pattern state machines.	22
3.6	State machine representing the normal operation of the <i>switch</i> class.	23
3.7	State machine representing the failure pattern of the <i>switch</i> class.	23
3.8	State machine representing the normal operation of the <i>LightBulb</i> class.	24
3.9	State machine representing the failure pattern of the <i>LightBulb</i> class.	24
3.10	Example of a failure propagation.	25
3.11	<i>QUMPropagationRule</i> for <i>powerDown</i>	25
3.12	Definition of the stereotypes used for repair and spare management.	26
3.13	Light controller example with repair unit and spare management.	27
3.14	State machine representing the failure pattern of the <i>switch</i> class, with <i>QUMAbstractRepairTransition</i>	27
3.15	Definition of the <i>QUMStateConfiguration</i> stereotype.	28
3.16	<i>QUMStateConfiguration</i> "SystemDown" assign to <i>LightBulbBroken</i>	29
4.1	A module in the PRISM language.	32
4.2	PRISM translation rule for <i>QUMComponents</i>	33
4.3	Encoding of the states.	33
4.4	PRISM translation of the state encoding.	34
4.5	PRISM translation rule for transitions.	34
4.6	Translation rule for event handling.	35

4.7	Translation rules for incoming and outgoing propagations.	35
4.8	PRISM translation rule for <i>QUMSpare</i>	36
4.9	PRISM template for repair commands that are added to <i>QUM- Component</i>	37
4.10	Translation rule for a dedicated <i>QUMRepairUnit</i>	37
4.11	Translation rule for a <i>QUMRepairUnit</i> with FCFS strategy.	38
5.1	Fault Tree Elements	44
5.2	Fault Tree Representation of a 2-out-of-3-System Failure	45
5.3	Fault Tree Representation of the running example.	49
5.4	Algorithm sketch of Definition 1	50
5.5	Algorithm sketch of Definition 2	50
5.6	Algorithm sketch of Definition 3	51
5.7	Semantics of fault trees in μ -calculus	51
5.8	Example of a UML sequence diagram.	53
5.9	UML sequence diagram of the running example fault tree.	56
6.1	The QuantUM Tool	57
7.1	Class diagram of the airbag system.	61
7.2	State machine representing the normal behavior of the microcon- troller.	62
7.3	State machine representing the failure pattern of the microcon- troller.	63
7.4	State machine representing the normal behavior of the FASIC.	64
7.5	State machine representing the failure pattern of the FASIC.	64
7.6	Properties of the <i>QUMStateConfiguration</i>	64
7.7	State machine representing the normal behavior of the FET.	65
7.8	State machine representing the failure pattern of the FET.	65
7.9	State machine representing the normal operation of the MainSensor.	66
7.10	Experiment results for T=10, T=100 and T=1000.	67
7.11	Fault tree for the <i>QUMStateConfiguration inadvertent_deployment</i> (T = 10).	69
7.12	UML sequence diagram for the <i>QUMStateConfiguration inadver-</i> <i>tent_deployment</i> (T = 10) (part 1 of 2).	70
7.13	UML sequence diagram for the <i>QUMStateConfiguration inadver-</i> <i>tent_deployment</i> (T = 10) (part 2 of 2).	71
7.14	Class diagram of the train odometer.	72
7.15	State machine representing the normal behavior of the WheelSen- sor.	73
7.16	State machine representing the failure pattern of the WheelSensor.	74

7.17	State machine representing the normal behavior of the RadarSensor.	74
7.18	State machine representing the failure pattern of the RadarSensor.	75
7.19	State machine representing the normal behavior of the Monitor. .	75
7.20	State machine representing the failure pattern of the Monitor. . .	75
7.21	State machine representing the normal behavior of the Observer.	76
7.22	State machine representing the normal behavior of the Speed QUMComponent.	76
7.23	Experiment results for T=10, T=100 and T=1000.	77
7.24	Fault tree for the <i>QUMStateConfiguration</i> unsafe (T = 10). . . .	78
7.25	UML Sequence diagram for the <i>QUMStateConfiguration</i> unsafe (T = 10) (part 1 of 2)	80
7.26	UML Sequence diagram for the <i>QUMStateConfiguration</i> unsafe (T = 10) (part 2 of 2)	81
8.1	Comparison of approaches known from literature	84

Chapter 1

Introduction

1.1 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 [1]. The analysis approach that we used was that of probabilistic Failure Modes Effect Analysis (pFMEA) [2].

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) [3] 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 [4]. 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.
3. The results of the formal analysis may not be easily transferable to the level of the high-level design language.
4. 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.

In [1] we also showed that counterexamples are a very helpful means to understand how certain error states representing hazards can be reached by the

system. While the visualization of the graph structure of a stochastic counterexample [5] helps engineers to analyze the generated counterexample, it is still difficult to compare the thousands of paths in the counterexample with each other, and to discern causal factors during fault analysis. In addition, it is necessary to map the information that is gleaned from the counterexample on the UML level in a manual process.

The objective of this thesis 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 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, and all outputs of the analysis are interpretable on 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. 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. An example of such high-level representation are fault trees [6].

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 [7] 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.
- For this model we compute counterexamples for the probabilistic properties of interest using our extension of PRISM [5], with the name DiPro. A counterexample is a set of execution paths, which are violating the analyzed property.
- The resulting counterexamples can then be represented as a fault tree, that is interpretable on the level of the UML model or they can be mapped onto an UML sequence diagram which is stored in a 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. In Fig. 1.1, a pictorial overview of our approach can be found.

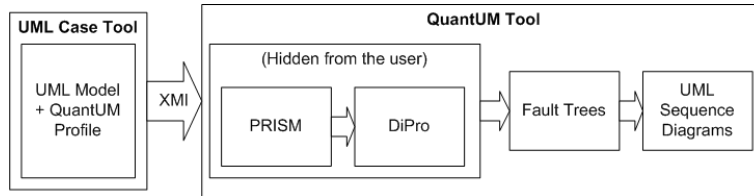


Figure 1.1: The QuantUM tool chain.

1.2 Contributions

The main contributions of this thesis can be summarized as follows:

- We propose an extension of the UML that allows for annotating both structural and behavioral diagrams with quantitative information. We call the resulting notation *QuantUM*.
- We define a translational semantics that enables the translation of the annotated UML models into an analysis model in the PRISM language.
- We present an automatic transformation process from probabilistic counterexamples to fault trees.
- We define a transformation from probabilistic counterexamples to UML sequence diagrams.
- We describe the development of a prototypical tool chain for quantitative safety analysis of UML models, that hides the analysis model from the user.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows: In Chapter 2 the Unified Modeling Language and probabilistic model checking are introduced. The quantitative extension of UML is presented in Chapter 3. Subsequently, the translation from quantitative UML to PRISM is described in Chapter 4. Chapter 5 is devoted to the description of high level representations of probabilistic counterexamples. The prototypical tool chain developed in this thesis is presented in Chapter 6. In Chapter 7 the QuantUM Approach is demonstrated on case studies known from literature. Finally, related work is discussed in Chapter 8 followed by the conclusion and outlook in Chapter 9.

Chapter 2

Foundations

2.1 Unified Modeling Language

The Unified Modeling Language (UML)[3] is a standardized general-purpose modeling language that is widely used in the field of software and system engineering. The standard is managed, and was created by, the Object Management Group. UML includes a set of graphic notation techniques to create visual models of software-intensive systems.

UML 2.3 [3] has 14 types of diagrams divided into two categories. Seven diagram types represent structural information, and the other seven represent general types of behavior, including four that represent different aspects of interactions. For a full overview of the diagrams and elements of the UML we refer to the UML standard specification [3].

In industry, UML is the de-facto standard for system and software architecture modeling. There exist a large number of commercial as well as open source Computer-Aided Software Engineering (CASE) tools which support UML, examples include the commercial tools IBM Rational Software Architect¹, Sparxsystems Enterprise Architect² and the open source tool ArgoUML³. Standards for the development of safety-critical systems like ISO IEC 61508 [8] or ISO CD 26262 [9] highly recommend the usage of UML for specification of the system and software.

¹<http://www.ibm.com/software/rational/>

²<http://www.sparxsystems.com/>

³<http://argouml.tigris.org/>

2.2 Probabilistic Model Checking

Probabilistic Model Checking [10] is an established automated technique used in the analysis of safety-critical systems.

Probabilistic model checking requires two inputs: a description of the system to be analyzed, typically given in some model checker specific modeling language and a formal specification of quantitative properties of the system, relating for example to its performance or reliability that are to be analyzed.

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 transition's occurrence. In the case of continuous-time Markov chain CTMC [11], which we use in this thesis, transitions are assigned positive, real values that are interpreted as the rates of negative exponential distributions. The probabilistic model checker constructs the state space of the model in an exhaustive fashion, based on a systematic exploration of all possible states that can occur.

The quantitative properties of the system that are to be analyzed are specified using a variant of temporal logic. The temporal logic used in this thesis is Continuous Stochastic Logic (CSL) [12, 13]. We give here a short introduction into CSL for a more comprehensive description we refer to [13]. CSL is a stochastic variant of the Computation Tree Logic (CTL) [14] with state and path formulas based on [15]. The state formulas are interpreted over states of a CTMC, whereas the path formulas are interpreted over paths in a CTMC. CSL 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 CTMC. In order to express the time span of a certain path, the path operators until (U) and next (X) are extended with a parameter that specifies a time interval.

In contrast to, for instance discrete-event simulation techniques, which generate approximate results by averaging results from a large number of random samples, probabilistic model checking applies numerical computation to yield exact results.

In this thesis we use the probabilistic model checker PRISM [4], which is open-source and was developed at the University of Oxford.

Chapter 3

Quantitative Extension of UML

3.1 Motivation

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 order to define a quantitative extension of the UML, we first need to specify our requirements on the extension. The main requirement is that all information needed to perform the quantitative analysis, in this case probabilistic model checking, shall be specified by the extension. We base the requirements and the terminology used on the definitions and concepts given in [16, 17]. While [16] gives the main definitions relating to dependability, [17] presents the state-of-the-art of reliability engineering methods and techniques. Additionally, requirements 6-7 are imposed by the authors, since they are crucial for the acceptance of the approach by engineers.

1. The extension shall be applicable on system and software architectures defined in UML.
2. The extension shall provide a way to specify dependability objectives / requirements.
3. The extension shall provide means for the specification of dependability characteristics of the system / software components, such as failure modes

and rates.

4. The extension shall provide means to specify failure propagation paths and dependencies between different system / software components.
5. The extension shall provide means to model safety mechanisms such as redundancy structures and repair management.
6. An experienced user (i.e. software-/safety engineer) must be able to use the extension with a minimum of training.
7. The cost incurred by the additional modeling using QuantUM shall be kept as low as possible.

3.2 Extension of the UML

There are two approaches possible to extend the UML with quantitative expressiveness. The first possibility is an extension or annotation of the UML language itself that can be used directly in the modeling tool. Alternatively, an external description of the required information, for instance in a text file, can be used. If an external description is used, the connection between the model and the external description needs to be ensured, either manually or by a tool, which can be difficult to accomplish. Additionally, the user has to learn how s/he can use the external description. We therefore choose to directly extend the UML language by suitable annotations, because it offers the benefit of being more directly integrated into the model.

The Unified Modeling Language offers two different ways for extension: first, one can extend or alter the UML meta-model in such a way that new UML elements, that is new metaclasses, are created, and second, one can apply a user-defined UML profile to the model. UML profiles allow the definition of stereotypes and constraints on each modeling element. A stereotype [3] is a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends. A metaclass is a class whose instances are classes. Just as a class in object-oriented programming defines the behavior of certain objects, a metaclass defines the behavior of certain classes and their instances. Each stereotype may extend one or more classes through extensions as part of a profile. Any UML model element can be extended by a stereotype. For example, states, transitions, activities, use cases, components, attributes, dependencies, etc. can all be extended with stereotypes. The properties of a stereotype are called tagged values. A stereotype can then be assigned to an instance of the metaclass, for example a class element, it is extending. We

then say the stereotype is *assigned* to the class element, or the class element is *tagged* with the stereotype.

The usage of profiles has two major advantages over extending the meta model: first it is easy to use and "lightweight" compared to extending the meta-model and second, it will not interfere with other important parts of a modeling tool implementing the UML standard, like for instance code generation. Hence, we choose to implement our extension by designing a UML profile.

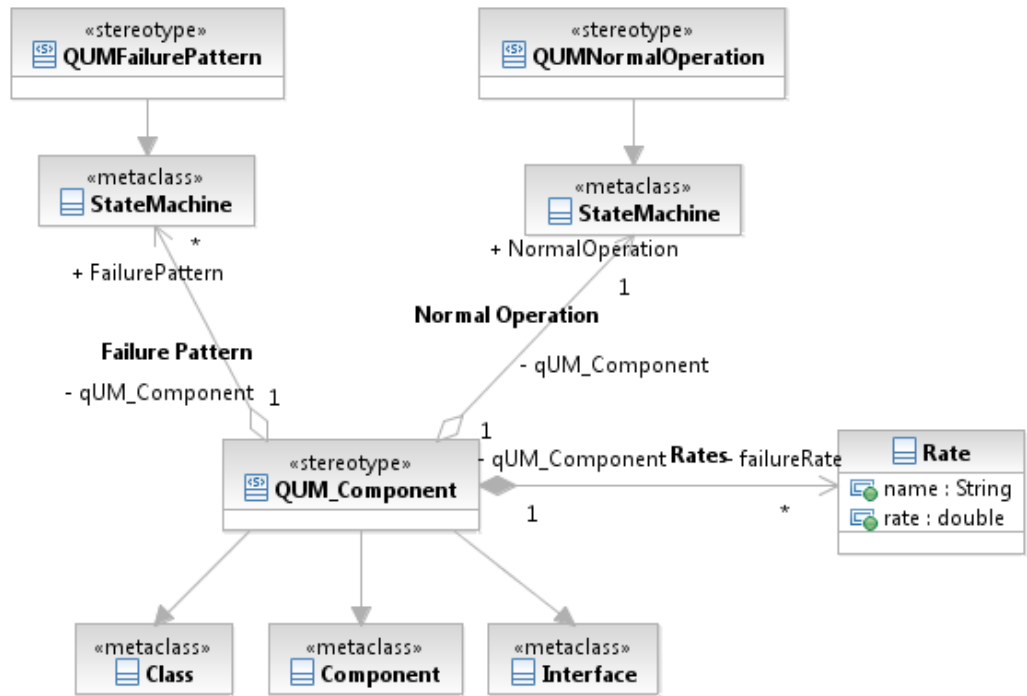
3.3 UML-Profile for Quantitative Analysis

This subsection is devoted to the description of the UML profile for quantitative analysis. We first present the syntactical elements that were introduced in order to specify the required information, then we define the semantics of the extension by specifying translation rules from the UML extended by this profile to PRISM. The dependability terminology that we used here is based on [16].

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. To avoid naming conflicts with other UML profiles that exist in the literature, we use the prefix *QUM* for the names of stereotypes that belong to our profile. In the following we define the stereotypes and their properties that are used to specify the information needed to perform stochastic analysis. We demonstrate the usage of the profile on the running example of a simple light controller system. The light control system consists of a switch and a light bulb, by pressing the switch the light bulb can be switched on or off.

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

Figure 3.1: Definition of the *QUMComponent* stereotype

these state machines saves modeling effort and avoids redundant descriptions. In some cases, the normal behavior of a *QUMComponent* is not of interest for the analysis, for instance when describing failures of external components. In those cases the specification of the failure pattern state machines is sufficient. In addition, each *QUMComponent* comprises a list called *Rates* that contains rates together with names identifying them. The definition of the *QUMComponent* stereotype, its relations and attributes is shown in Figure 3.1.

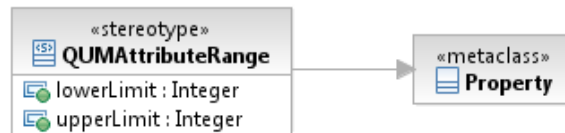
In Figure 3.2 the class diagram of the light controller system can be seen. Both the switch and the light bulb are represented as classes that are tagged with the stereotype *QUMComponent*. The *Switch* class comprehends the operation *switchPressed* and a boolean attribute with the name *value* that represents the state of the switch. The boolean attribute *shining* of the *LightBulb* class indicates whether the light bulb is shining or not. The stereotype *QUMComponent* allows for the association with state machines representing the normal and failure behavior.



Figure 3.2: Class diagram of the light controller system.

QUMAttributeRange

The range of integer attributes of *QUMComponents* can be specified by this stereotype. This is necessary since finite state verification methods, such as probabilistic model checking, require variables to be defined over finite domains.

Figure 3.3: Definition of the *QUMAttributeRange* stereotype

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 *QUMAbstractStochasticTransition* and *QUMStochasticTransition* (see Fig. 3.4). 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 *QUMAbstractStochasticTransition* 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 *QUMAbstractStochasticTransition*, this rate is then considered for the *QUMAbstractStochasticTransition*. The *QUMAbstractStochasticTransition* allows to define general state machines in a repository where the rates can be set individually for each component. The stereotypes *QUMAbstractFailureTransition*, *QUMAbstractRepairTransition*, *QUMFailureTransition* and *QUMRepairTransition* are specializations of *QUMAbstractStochasticTransition* and *QUMStochasticTransition*, respectively.

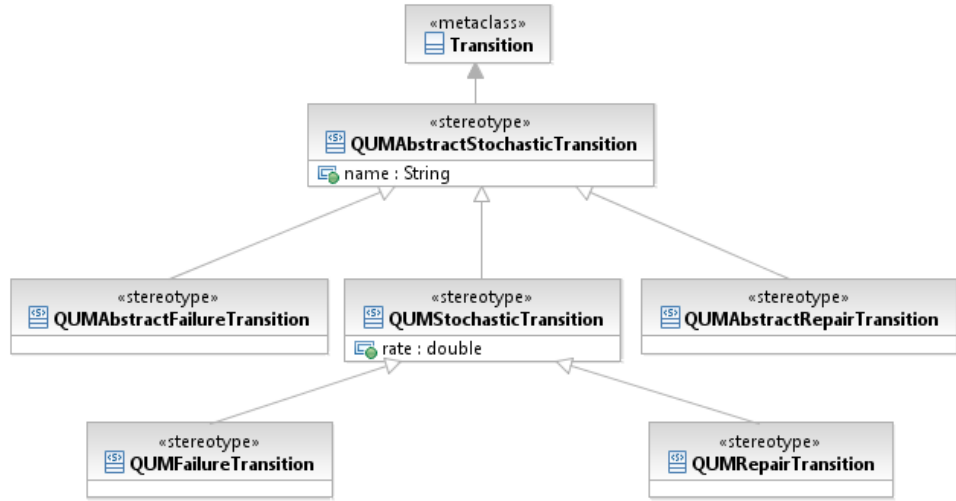


Figure 3.4: Definition of the stereotypes for abstract and concrete stochastic transitions.

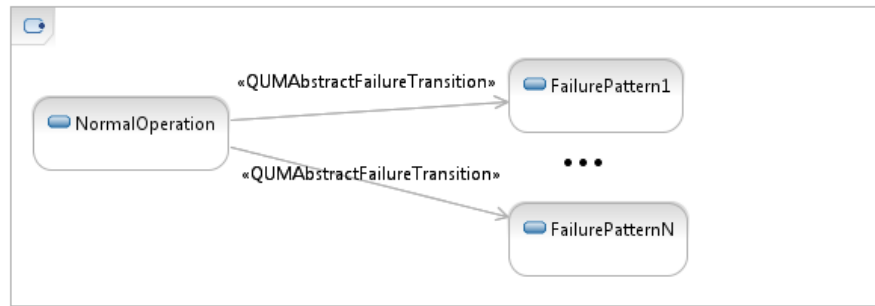


Figure 3.5: Combination of normal behavior and failure pattern state machines.

The normal behavior state machine and all failure pattern state machines are implicitly combined in one hierarchical state machine, cf. Figure 3.5. 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.

In the light controller example there are two classes that are tagged with the *QUMComponent* stereotype, namely the *Switch* class and the *LightBulb* class. The state machine that represents the normal behavior of the *Switch* class is

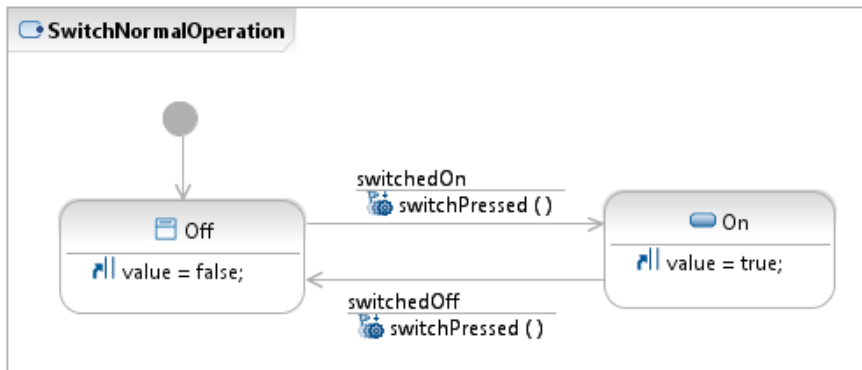


Figure 3.6: State machine representing the normal operation of the *switch* class.

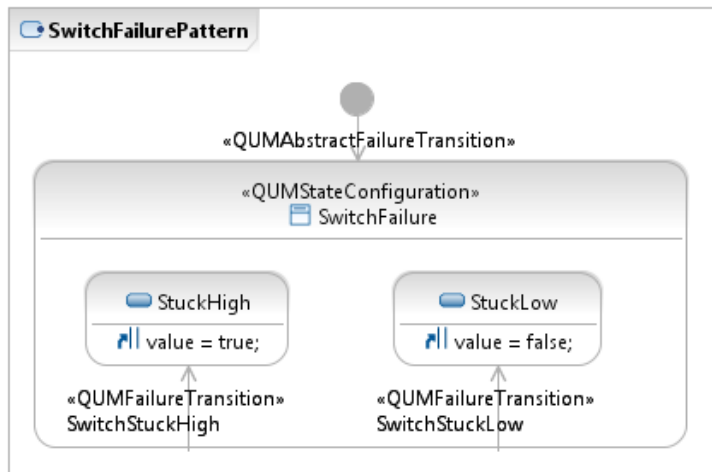


Figure 3.7: State machine representing the failure pattern of the *switch* class.

shown in Figure 3.6. It can be either in the state *Off* or *On*, and changes its state whenever the operation *switchPressed* is executed. On entering the *Off* state, the *value* variable is set to false and on entering the *On* state the *value* variable is set to true.

Figure 3.7 shows the state machine representing the failure pattern of the *Switch* class. If the switch enters the failure state *SwitchFailure*, it can either go to the state *StuckHigh*, where the value of the switch is always high (i.e. entry action sets *value* to true), or it can go to the state *StuckLow*, where the value of the switch is always low (i.e. entry action sets *value* to false).

The normal behavior of the *LightBulb* class is specified by the state machine shown in Figure 3.8. The *LightBulb* class can be in two states when in normal operation, namely the state *LightOff* where the light bulb is not shining and

therefore the variable *shining* is set to false in the entry action and the state *LightOn* where the light bulb is shining and the variable *shining* is set to true in the entry action. The transitions from one state to another are triggered by the change of the *value* variable of the *switch* class. The transition from state *LightOff* to *LightOn* is taken whenever *value* changes its value to true. The reverse transition from state *LightOn* to *LightOff* is taken whenever the value of the variable *value* changes to false. The state machine describing the failure pattern of the *LightBulb* class is displayed in Fig. 3.9. It consists of one failure state where the light bulb is broken and hence will not shine regardless of whether the switch is pressed or not. Consequently, the variable *shining* is set to false by the entry action of the state *LightBulbBroken*.

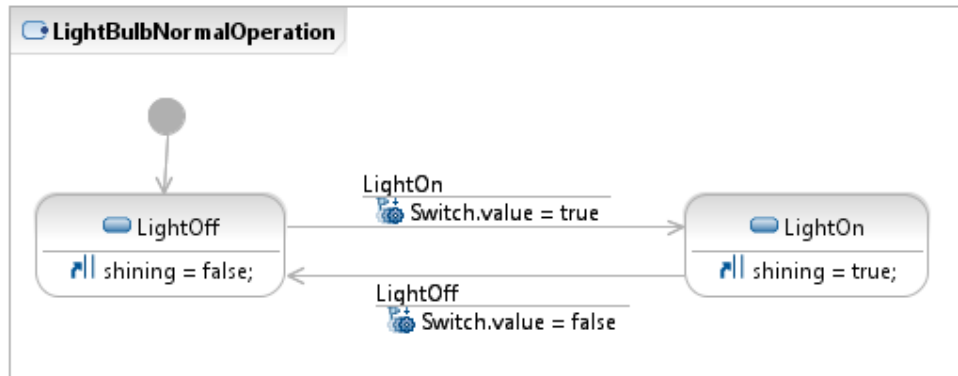


Figure 3.8: State machine representing the normal operation of the *LightBulb* class.

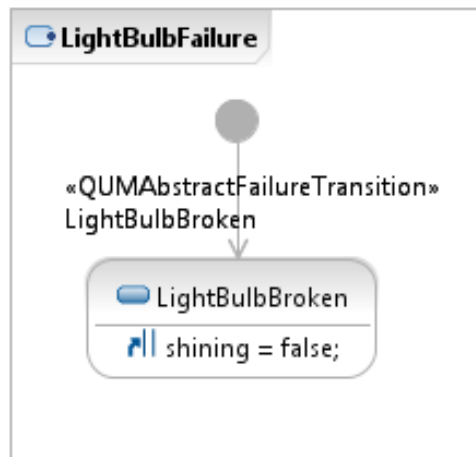


Figure 3.9: State machine representing the failure pattern of the *LightBulb* class.

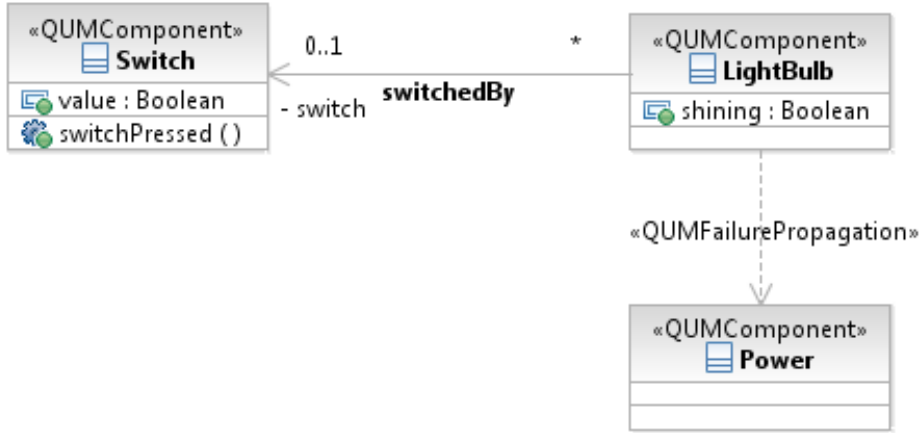


Figure 3.10: Example of a failure propagation.

activatorName	powerDown
rate	1.0
targetTransitionName	LightBulbBroken

Figure 3.11: *QUMPropagationRule* for *powerDown*

QUMFailurePropagation

There are two possible ways how a failure propagation can be specified with the extension. First, it might be possible to specify the propagation solely by using the state machines specifying the behavior of the components: In the running example, a failure of the *Switch* class automatically propagates to the *LightBulb* class, if for instance the *Switch* is in the failure state *StuckLow* the variable *value* is set to false and the *LightBulb* class enters the state *LightOff*, if it is not already in that state, and will stay in that state. When this is impossible, we propose the use of the *QUMFailurePropagation* stereotype. Whenever the component that propagates the failure executes a *QUMFailureTransition* that matches the *activatorName* of one of the *QUMPropagationRules*, the component that is the propagation target executes the *QUMFailureTransition* which matches the *targetTransitionName* with the specified rate. Figure 3.10 gives the example of a failure propagation in the light control system. The corresponding *QUMPropagationRule* is shown in Fig. 3.11 and specifies that whenever the *Power* component executes the *PowerDown* transition, the light bulb will execute the *LightBulbBroken* transition.

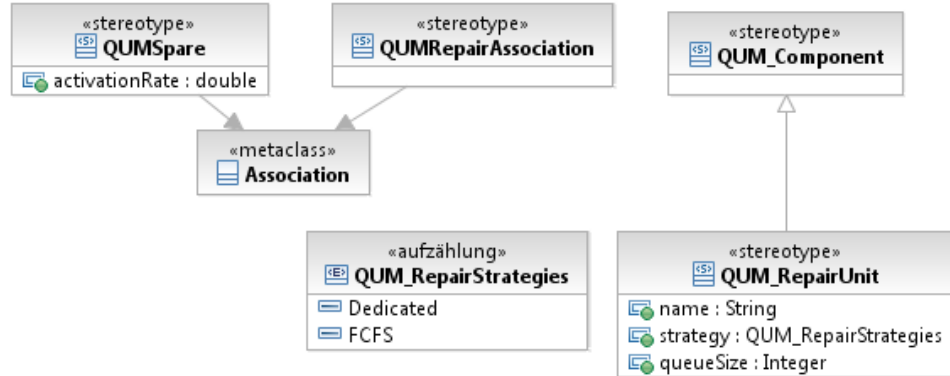


Figure 3.12: Definition of the stereotypes used for repair and spare management.

Repair and Spare Management

Our extension also allows for the specification of repair management strategies, in which a repair unit can be defined that is able to repair system components whenever they have failed. Additionally, spare components can be defined for each system component. Spare components are activated whenever their master component fails, cf. Figure 3.12. The stereotype *QUMSpare* can be assigned to associations in order to specify that the associated component acts as a spare and is activated after a failure of the main component with the rate specified in *activationRate*.

The *QUMRepairUnit* is a specialization of the *QUMComponent* stereotype and can be associated to another *QUMComponent* by an association with the *QUMRepairAssociation* stereotype. The *QUMRepairUnit* repairs the associated *QUMComponents* according to one of the *QUMRepairStrategies* specified in *strategy*. At the moment two repair strategies are implemented. *Dedicated* is the strategy that uses exactly one repair unit per one component. *First come first serve* (FCFS) is a strategy according to which more than one component can be repaired by one repair unit. Similarly to the specification of the failure patterns, the *QUMRepairUnit* comprises a set of rates, representing the repair rates for the component. The corresponding transitions are specified in the failure pattern with the *QUMRepairTransition* and *QUMAbstractRepairTransition* stereotypes (cf. Fig.3.14).

Figure 3.13 shows the light controller example, with a *SwitchRepairUnit* which is attached to the *Switch* with a *QUMRepairAssociation* association. Furthermore, the *LightBulb* component has a self-referencing association with the stereotype *QUMSpare*. This means that there are a total of two spare *LightBulb* instances that will be activated, after each other, with the specified *activa-*

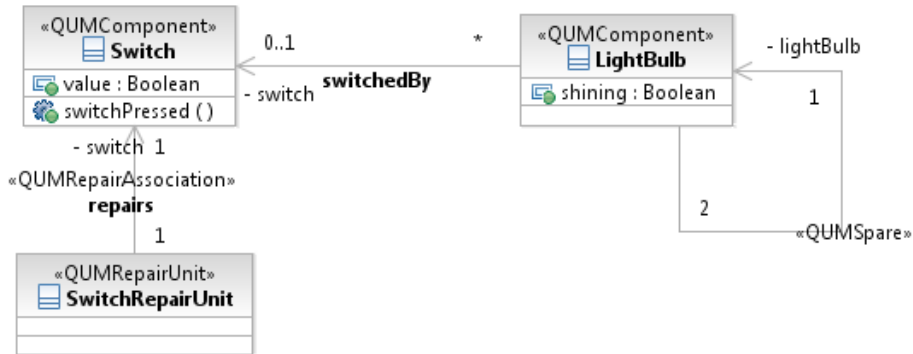


Figure 3.13: Light controller example with repair unit and spare management.

tionRate as soon as the *LightBulb* executes a transition that is tagged with the *QUMFailureTransition* or *QUMAbstractFailureTransition* stereotype. The *SwitchRepairUnit* repairs the *Switch* class and executes the *QUMAbstractRepairTransition* leading to the normal behavior state machine with the specified repair rate.

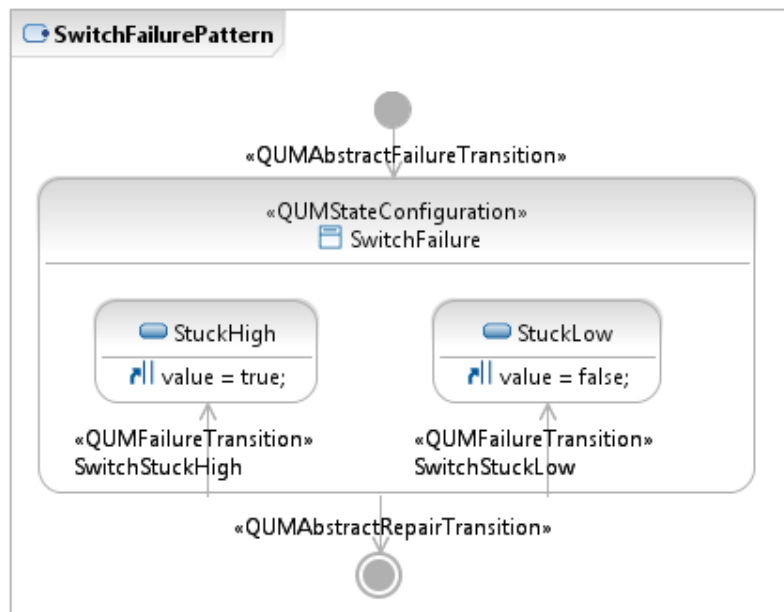


Figure 3.14: State machine representing the failure pattern of the *switch* class, with *QUMAbstractRepairTransition*

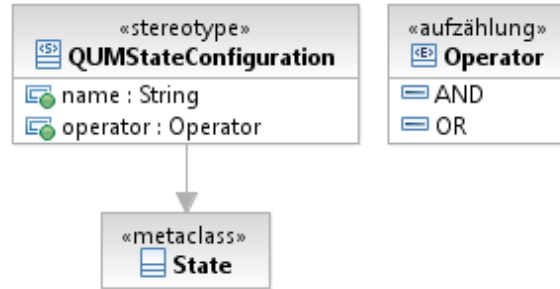


Figure 3.15: Definition of the *QUMStateConfiguration* stereotype.

QUMStateConfiguration

The *QUMStateConfiguration* (cf. Fig. 3.15) 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.

In our running example we assign the *QUMStateConfiguration* with the name *SystemDown* to the states *SwitchFailure* and *LightBulbBroken* (Fig. 3.16) and select the *or*-operator to connect these two states. Therefore, the state configuration *SystemDown* is true whenever the system is either in the *SwitchFailure* or *LightBulbBroken* state or in both.

3.4 Discussion

This section is devoted to a discussion whether the QuantUM approach fulfills the requirements stated in Section 3.1.

The requirement (1) "The extension shall be applicable on system and software architectures defined in UML" is fulfilled, since our extension is applicable to the UML elements that are used to specify system and software architectures. In order to fulfill requirement (2) "The extension shall provide a way to specify dependability objectives / requirements.", we have introduced the stereotype *QUMStateConfiguration* which allows for the specification of state configurations, which are then used in the QuantUM tool to specify depend-

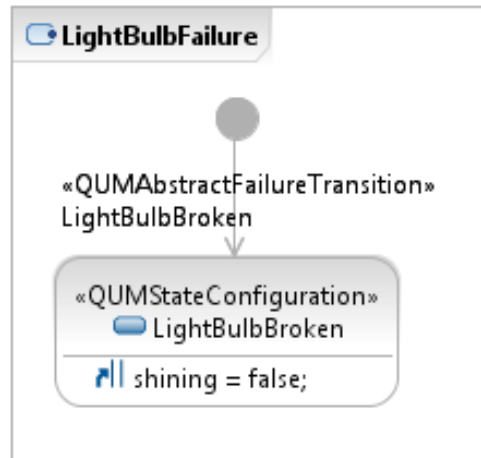


Figure 3.16: *QUMStateConfiguration* "SystemDown" assign to *LightBulbBroken*

ability objectives and requirements (cf. Section 4.2). Our profile allows for the specification of all dependability characteristics that are according to [17] needed for the analysis. Hence the requirement (3) "The extension shall provide means for the specification of dependability characteristics of the system / software components, such as failure modes and rates" is fulfilled. Requirement (4) "The extension shall provide means to specify failure propagation paths and dependencies between different system / software components" is achieved by the introduction of the stereotype *QUMPropagationRule*. The stereotypes *QUMSpare*, *QUMRepairUnit*, *QUMRepairAssociation* and *QUMRepairStrategies* allow for the specification of the information that is needed to satisfy requirement (5): "The extension shall provide means to model safety mechanisms such as redundancy structures and repair management." The requirement 6 "An experienced user (i.e. software-/safety engineer) must be able to use the extension with a minimum of training." is satisfied, because on the one hand, the definitions and concepts used are standard in industry, hence the engineers are already familiar with them and on the other hand, the engineers are already familiar with the usage of the UML and the UML modeling tool. Since the extension was designed to be applicable to the original architectural model, that is it is not necessary to construct an additional analysis model, the requirement (7) "The cost incurred by the additional modeling using QuantUM shall be kept as low as possible." is also fulfilled.

Chapter 4

From Quantitative UML to PRISM

4.1 Semantics of the Extension

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 [4]. 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 [18].

The PRISM language is a state-based, guarded command language that is based on the reactive modules formalism of Alur and Henzinger [19]. We use the PRISM language in order to specify continuous-time Markov chains [11] which are then used for the probabilistic model checking. Continuous-time Markov chains are used for the analysis because their rates are interpreted as the rates of negative exponential distributions. This is important for our type of analysis, because the probability distributions for all failure rates given by the manufacturers of electronic components or failure rates that can be found in engineering standards for failure rates [20, 21] are exponential.

We present here an introduction of the basic elements of the PRISM language, for a precise definition of the semantics we refer to [22]. 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 samplemodule
  var1: bool init false;
  var2: [0..11] init 0;
  [Count] (var2 < 2) -> 0.8: ( var2'= var2 + 1);
  [End] (var2 = 2) -> 1.0: ( var1'= true);
endmodule

```

Figure 4.1: A module in the PRISM language.

module is described by a set of commands. A command takes the form:

$$[\text{transition_label}] \text{guard} \rightarrow \text{rate}_1 : \text{update}_1 \& \dots \& \text{update}_n;$$

The *guard* is a predicate over all the variables in the model (including those belonging to other *modules*). Each *update* describes a transition which the module can make if the *guard* is true. A transition is specified by giving the new values of the variables in the *module*, possibly as a function of other variables. Each *update* is also assigned a *rate* which will be assigned to the corresponding transition. An example of a PRISM *module* is given in Fig. 4.1. The module named *samplemodule* 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 ($\text{var2} < 2$) evaluates to true, the update ($\text{var2}' = \text{var2} + 1$) is executed with the rate 0.8. If the guard ($\text{var2} = 2$) evaluates to true, the update ($\text{var1}' = \text{true}$) is executed with the rate 1.0.

In the following we present translation rules that allow the translation of the annotated UML model into the PRISM language. We use the following notation to specify the transition rules: everything that is enclosed by % characters, such as in *%module_id%*, will be rewritten by the QuantUM tool. Statements enclosed by << ... >>, such as in << $\&$ (*%action%*) >>, are optional and will only be rewritten when the enclosed element does have a value, for instance if *%action%* is not null. All other notational elements are part of the PRISM language, see [4].

QUMComponent

Each *QUMComponent* is translated into one PRISM *module*, as shown in Fig. 4.2. The ... are replaced by the PRISM code of the state machine, propagation rules etc. belonging to this *QUMComponent*. The placeholder *%module_id%* represents a unique id that identifies the corresponding UML element. For each attribute of the *QUMComponent* the line *%module_id%-%attribute_name%: %type_definition% init %init_value%;* is added where *%attribute_name%* is the name of the attribute, *%type_definition%* is replaced by *bool* for boolean vari-

```

module %module_id%
  //for each attribute
  %module_id%_%attribute_name%: %type_definition% init %init_value%;
  ...
endmodule

```

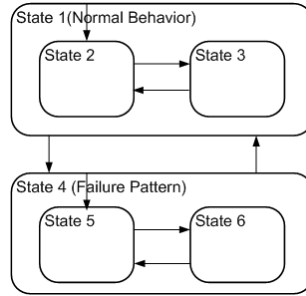
Figure 4.2: PRISM translation rule for *QUMComponents*.

Figure 4.3: Encoding of the states.

ables or $[%range_lower\%..%range_upper\%]$ for integer variables. The values for $%range_lower\%$ and $%range_upper\%$, which are representing the lower and upper value of the range of possible values for this variable, need to be specified in the model by the *QUMAttributeRange* stereotype. For boolean variables the placeholder $%init_value\%$, which represents the initial value, is set to *false*, while for integer variables it is set to $%range_lower\%$.

State Machines

The state machines describing the normal behavior and the failure patterns of a *QUMComponent* are first combined into one hierarchical state machine as shown in Fig. 4.3 and then translated to PRISM. For the purpose of translation, the states are first numbered and then encoded in an integer variable. States representing the normal behavior are always assigned a value between 0 and the total number of states representing normal behavior ($\#normstate$). All failure states are identified by a number that is greater than $\#normstate$. Each parent state together with its sub-states can be represented by the range of the state numbers, for instance 1 to 3 for the normal behavior state in Figure 4.3. This state encoding by ranges allows for the translation of hierarchical state machines. The placeholders $\#\#normstate\%$ and $\#\#failstates\%$ represent the number of states in the normal behavior state machine and failure pattern state machines respectively and will be replaced with their actual values by the translation tool. The variable $%module_id\%-state$ is used to represent the state of the state

```
%module_id%_state: [0..%#normstate% + %#failstates%] init 0;
```

Figure 4.4: PRISM translation of the state encoding.

```
[%module_id%_%transition_name%]
  ((%module_id%_state >= %state_id_parent%)
   & (%module_id%_state <= %state_id_substate_{n}%)
  )|(%module_id%_state = %transition_source_id%) << & (%guard%) >>
-> %rate%: ( %module_id%_state' = %transition_target_id%)
  << & (%actions%) >> << & (%events_fired%) >>;
```

Figure 4.5: PRISM translation rule for transitions.

machines, according to the state encoding explained above (cf. Fig. 4.4). All transitions are translated into PRISM commands. A transition is enabled and will be taken with the rate specified when the following conditions are fulfilled:

- The state machine is in a state in which the transition is an out-going transition, or in a sub-state of this state. This condition can be represented by the following expression

$$\begin{aligned} & ((\text{state_id_parent} = \text{transition_source_id}) \wedge \\ & (\text{state_id_parent} \leq \text{module_id_state} \leq \text{state_id_substate}_n)) \\ & \vee (\text{module_id_state} = \text{transition_source_id}) \end{aligned}$$

where *module_id_state* represents the current active state of the state machine, *state_id_parent* represents the id of a parent state *p*, *state_id_substate_n* represents the sub-state of *p* that has the highest id and *transition_source_id* represents the id of the state where the transition starts. If the state does not have sub-states the condition $(\text{module_id_state} = \text{transition_source_id})$ suffices. In Fig. 4.3 the transition going from state 1 (normal behavior) to state 4 (failure pattern) has a *transition_source_id* = 1 and is hence enabled in states 1, 2 and 3.

- The event causing an execution of the transition has been fired.
- The transition guard evaluates to true.

The corresponding PRISM code for transitions is shown in Fig. 4.5. In UML the synchronization of state machines of different *QUMComponents* can be achieved through events. In PRISM synchronization is achieved by using the same transition labels for two or more commands. Transitions with identical transition names are executed atomically by both processes. For each event we add a

```

%module_id%_%eventname_active%: bool init false;
[%eventname%] (true)
-> 1.0: (%module_id%_%actionname%'=false);>>

```

Figure 4.6: Translation rule for event handling.

```

//For all outgoing propagations
[%Propagation_Id%]
  (%module_id%_state = %id_prop_state%)
-> %rate%: (true);>>
//For all incoming propagations
[%Propagation_Id%]
  (true)
-> %rate%: ( %module_id%_state'= %id_target%)
  << & (%yaction%) >> << & (%events_fired%) >>; >>

```

Figure 4.7: Translation rules for incoming and outgoing propagations.

boolean variable representing the availability of this event. If the event is fired by a *QUMComponent* the value of this variable is set to true, else it is false. Subsequently, a transitions with the name of the event as transition label, that is guarded by that variable, becomes enabled. Now, all transitions of the other components which are triggered by the event and thus have the event name as their transition label will be executed if there guards evaluate to true. Figure 4.6 shows a translation rule that adds the variable and the additional command for synchronization.

QUMFailurePropagation

The *QUMFailurePropagation* can be translated by adding a synchronization command to the state machine propagating the failure, and by adding the same synchronization command to the state machine receiving the failure (see Fig. 4.7). The propagating command is enabled as soon as the state machine is in the failure state, the receiving command then forces the transition into the specified state.

QUMSpare

The *QUMSpare* stereotype is translated by adding a counter that counts the active spares, and one transition command that sets the module to the initial state whenever a failure state is entered and there still is a spare left that can be activated, see Figure 4.8. The placeholder *%#spares%* represents the number of spares (if any) that are associated with this element.

```

%module_id%_activespares: [0..#spares%]; >>
[%module_id%_SpareActivated]
  (%module_id%_state > %normstate%)
  & (%module_id%_activespares < %module_id%_nuofspares)
-> %rate%: ( %module_id%_state = %id_init%)
  & (%module_id%_activespares = %module_id%_activespares + 1); >>

```

Figure 4.8: PRISM translation rule for *QUMSpare*

Repair Management

Associations tagged with the stereotype *QUMRepairAssociation*, identify the *QUMComponents* that can be repaired by a *QUMRepairUnit*. The task of the *QUMRepairUnit* is to queue repair requests and then executed the repair transitions of the *QUMComponent* with the specified repair rate. The synchronization between a *QUMRepairUnit* and its associated *QUMComponent* is established via transition labels. Hence, the PRISM code for the *QUMRepairUnit* must be generated and the transitions needed to request and execute a repair need to be added to the *QUMComponent*.

For each *QUMRepairUnit* a new module representing this repair unit is created. In case the element tagged with the *QUMRepairUnit* stereotype is also tagged with the *QUMComponent* stereotype, there already exists a PRISM *module* representing the *QUMComponent* and the repair commands are added to this *module*. In the following *%repairmodule_id%* represents the identifier of the *QUMRepairUnit*, and *%module_id%* represents the module being repaired. The commands that are added to the *QUMComponent* that can be repaired by the *QUMRepairUnit* are shown in Fig. 4.9. There is one synchronized command to request a repair (*[%module_id%_RequestRepair]*) and one synchronized command to perform the repairing (*[%module_id%_Repaired]*). In Fig. 4.10 the translation rule for a *QUMRepairUnit* that is dedicated to one *QUMComponent* is represented. Here the command *[%module_id%_RequestRepair]* is synchronized with the command with the same transition label in *QUMComponent* and receives repair requests. Whenever a repair request was made, the transition with the transition label *[%module_id%_Repaired]* becomes enabled and will be taken with the specified repair rate (*%repairrate%*). This transition is synchronized with the transition *[%module_id%_Repaired]* in the corresponding *QUMComponent*, that will set the component to its initial state.

The translation rule for a *QUMRepairUnit* with first come first serve (FCFS) strategy is shown in Fig. 4.11. The variable *order* indicates which repair request will be executed next. The variable *repairRequests* represents the number of the pending repair request, whereas the variable *repaired* represents the number of

```

[%module_id%_RequestRepair]
  (%module_id%_state > %normstate%)
  -> 1.0: (true);
[%module_id%_Repaired]
  (%module_id%_state > %normstate%)
  -> 1.0: ( %module_id%_state = %id_init%);

```

Figure 4.9: PRISM template for repair commands that are added to *QUMComponent*

```

module %repairmodule_id%_RepairUnit
  //Repair
  repairRequested: bool init false;
  [%module_id%_RequestRepair]
    (true)
    -> 1.0: (repairRequested' = true);
  [%module_id%_Repaired]
    (repairRequested = true)
    -> %repairrate%: (repairRequested' = false);
endmodule

```

Figure 4.10: Translation rule for a dedicated *QUMRepairUnit*.

repair requests already processed. The variable $\%module_id\%_order$ represents the queue position of the module with the id in $\%module_id\%$. There is one $\%module_id\%_order$ variable, for each module that is assigned to this *QUMRepairUnit*. The variable $order$, $repairRequests$, $repaired$ and $\%module_id\%_order$ are all limited by the size of the queue ($\%queue_size\%$). Whenever the queue is full, the repair unit will not accept any repair requests until all repair requests in the queue are processed. Whether the repair unit is ready to executed a request is indicated by the boolean variable $ready$. When a repair request is made, the module requesting the repair is assigned a position in the queue, that is $\%module_id\%_order$ is set to the current value of $repairRequests$ which represents the last position in the queue. The queue is then processed first come first serve order, that is lowest $\%module_id\%_order$ first.

Whenever $\%module_id\%_order$ has the same value than $order$, the transition with the transition label $[\%module_id\%_Repaired]$ becomes enabled and will be taken with the specified repair rate ($\%repairrate\%$). This transition is synchronized with the transition $[\%module_id\%_Repaired]$ in the corresponding *QUMComponent*, that will set the component to its initial state.

```

module %repairmodule_id%_RepairUnit

    order: [0..%queue_size%] init 1;
    repairRequests: [0..%queue_size%] init 1;
    repaired: [0..%queue_size%] init 0;
    %module_id%_order: [0..%queue_size%] init 0;
    ready: bool init true;

[%module_id%_RequestRepair]
    (%module_id%_order = 0) & (repairRequests < %queue_size%)
    -> 1.0: (%module_id%_order' = repairRequests)
        & (repairRequests' = repairRequests + 1);
[RepairQueue]
    (repairRequests > order) & (repaired = order) & (ready = true)
    -> 1.0: (order' = order + 1) & (ready' = false);
[ResetQueue]
    (order = repairRequests)
    -> 1.0: (order' = 1) & (repairRequests' = 1) & (repaired' = 0);
[%module_id%_Repaired]
    (%module_id%_order = order)
    -> %repairrate%: (repaired' = %module_id%_order)
        & (%module_id%_order' = 0) & (ready' = true);
endmodule

```

Figure 4.11: Translation rule for a *QUMRepairUnit* with FCFS strategy.

4.2 Automatic Property Construction

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 in this thesis is Continuous Stochastic Logic (CSL) [12, 13]. 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.

In the following we describe how state formulas can automatically be generated by the QuantUM tool.

Probability of the failure of a specified *QUMComponent*

A *QUMComponent* is failed whenever it has entered a failure pattern state machine. Hence, whenever the value of the variable *%module_id%_state* is greater than *normstate* the component is failed. Therefore, the resulting state

formula representing the failure of a component is:

$$(\%module_id\%_state > \%\#normstate\%)$$

Thus, the CSL formula

$$P_{=?}[(true)U(\%module_id\%_state > \%\#normstate\%)]$$

can be used to determine the probability of a failure of the *QUMComponent* with the specified module id.

Probability of the failure of any *QUMComponent*

As already explained $(\%module_id\%_state > \%\#normstate\%)$ is the state formula that represents the failure of one component. Consequently, the state formula

$$\begin{aligned} &(\%module_id_1\%_state > \%\#normstate\%) \quad | \\ &(\%module_id_2\%_state > \%\#normstate\%) \quad | \dots | \\ &(\%module_id_n\%_state > \%\#normstate\%) \end{aligned}$$

represents the failure of any of the components with module ids $\%module_id_1\% \dots \%module_id_n\%$. Similarly to the above, the CSL formula

$$\begin{aligned} P_{=?}[(true)U(&(\%module_id_1\%_state > \%\#normstate\%) \quad | \\ &(\%module_id_2\%_state > \%\#normstate\%) \quad | \dots | \\ &(\%module_id_n\%_state > \%\#normstate\%))] \end{aligned}$$

can be used to determine the probability of a failure of any of the *QUMComponents*.

Probability of a *QUMStateConfiguration*

As already explained, each *QUMStateConfiguration* can also be interpreted as a boolean formula, each state can either be true when the system is in this state or in one of its sub-states, 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 states are identified by the state encoding explained above, hence the boolean expressions

$$in_state_id = (state_id \leq \%module_id\%_state \leq state_id_substate_n)$$

can be used to determine whether a *QUMComponent* is in the state with the state id in *%state_id%*, or in one of its sub-state, if the state with *%state_id%* does not have sub-states, the expression

$$\text{in_state_id} = \%module_id\%_state = \%state_id\%$$

suffices. Hence, to obtain the whole state configuration we connect the individual formulas by the operator defined. Consequently, for the *and*-operator we get

$$\varphi = (\text{in_state_id}_1 \& \text{in_state_id}_2 \& \dots \& \text{in_state_id}_n)$$

as state formula representing the *QUMStateConfiguration*, for the *or*-operator we get

$$\varphi = (\text{in_state_id}_1 | \text{in_state_id}_2 | \dots | \text{in_state_id}_n)$$

respectively. In analogy to the previous cases, the CSL formula

$$P_{=?}[(\text{true})U(\varphi)]$$

can be used to determine the probability of reaching the *QUMStateConfiguration*.

Chapter 5

High-Level Representations of Probabilistic Counterexamples

5.1 Motivation

In [1] we showed that counterexamples are a very helpful means to understand how certain error states representing hazards can be reached by the system. While the visualization of the graph structure of a stochastic counterexample [5] helps engineers to analyze the generated counterexample, it is still difficult to compare the thousands of paths in the counterexample with each other, and to discern causal factors during fault analysis.

In order to facilitate the counterexample interpretation, we propose the following strategy: first the mapping of counterexamples onto fault trees and second the mapping of paths of the counterexample in the fault tree onto UML sequence diagrams.

In safety analysis, fault tree analysis (FTA) [6] is a well-established method to break down the hazards occurring in complex, technical systems into a combination of what is referred to as basic events, which represent system component failures. The main drawback of fault tree analysis is that it relies on the ability of the safety engineer to identify all possible component failures that might cause a certain hazard. In this chapter we present a method that automatically generates a fault tree from a probabilistic counterexample. Our method provides a compact and concise representation of the system failures using a representation that is well known by the safety engineers.

Although, fault trees are common in industrial practice and known by en-

gineers, we propose an additional mapping of the paths of the counterexample in the fault tree onto UML sequence diagrams and thus lift the counterexample on the level of the UML model.

In the following we first briefly introduce precursory work on counterexample generation for stochastic model checking (Section 5.2). Then we present in Section 5.3 our mapping of probabilistic counterexamples to fault trees and finally we present the mapping of the counterexample paths belonging to the fault tree to UML sequence diagrams in Section 5.4.

5.2 Probabilistic Counterexamples

5.2.1 Counterexamples in Stochastic Model Checking

Just like in traditional model checking, given an appropriate system model and a CSL property, stochastic model checking tools such as PRISM [4] or MRMC [23] can verify automatically whether the model satisfies the property. If the model refutes the property, a counterexample usually helps engineers to comprehend the reasons of the property violation and to devise arrangements to fix the error. The computation of counterexamples in stochastic model checking has recently been addressed in [24, 25, 26, 27, 28, 29, 30, 31, 32].

5.2.2 Notion of Counterexamples

For our purposes in this thesis it suffices to consider upper bounded properties, which require the probability of a property offending behavior not to exceed a certain upper probability bound. In CSL such properties can be expressed by formulas of the form $P_{\leq p}(\varphi)$, where φ is path formula specifying undesired behavior of the the system. Any path which starts at the initial state of the system and which satisfies φ is called a *diagnostic path*. A counterexample for an upper bounded property is a set X of diagnostic paths such that the accumulated probability of X violates the probability constraint $\leq p$. If the formula $P_{=?}(\varphi)$ is used, the probability of the path formula φ to hold is computed and the counterexample contains all paths fulfilling φ .

5.2.3 Generation of Counterexamples

We use the heuristic state space search algorithm XK* [33], which is an optimized version of the K*[34] algorithm, to efficiently compute the set CX of diagnostic paths. XK* finds the k shortest paths in a given directed graph G for a start vertex and a set of target vertices. The following are the main ideas underlying the algorithmic structure of XK*:

1. We apply the directed search algorithm A^* [35] on G in order to determine a shortest path tree T on G .
2. All edges from G which A^* explores, are inserted into a special graph structure called path graph $PG(G)$. The details about the structure of $PG(G)$ are beyond the scope of this thesis, here it suffices to know that $PG(G)$ is used to construct the k shortest paths in G . This is accomplished by a shortest path search on $PG(G)$, for example using Dijkstra's algorithm [36].
3. Dijkstra's search on $PG(G)$ is performed concurrently with A^* on G . Consequently, Dijkstra will be able to deliver solution paths before G is completely searched by A^* .

In order to apply XX^* to the generation of counterexamples, we use a probabilistic variant of A^* on the state transition graph of the Markov model. The paths found by XX^* are added into a diagnostic subgraph, which is then considered as the counterexample. The probability of the counterexample is computed using a stochastic model checker whenever the diagnostic subgraph grows by q per cent (in our experiments $q = 20$). When the probability of the diagnostic subgraph is sufficient to violate the probability bound, then CX is provided as a counterexample.

5.3 From Probabilistic Counterexamples to Fault Trees

5.3.1 Fault Trees and Fault Tree Analysis

Fault trees (FTs) [6] have been used extensively, in particular in fault analysis, to illustrate graphically under which conditions systems can fail, or have failed. In our context, we need the following elements of fault trees:

1. Basic event: represents a basic, atomic failure event.
2. *AND*-gate: represents a failure, if all of its input elements fail.
3. *OR*-gate: represents a failure, if at least one of its input elements fails.
4. Priority-*AND* (*PAND*-gate): represents a failure, if all of its input elements fail in the specified order; where the required input failure order is usually read from left to right.
5. *Intermediate Event*: represents a failure event that is caused by its child nodes. The probability of the intermediate event to occur is denoted by the number in the lower right corner. The top event is a special case of Intermediate Event, representing the system hazard.

The graphical representation of these elements can be found in Fig. 5.1. For an

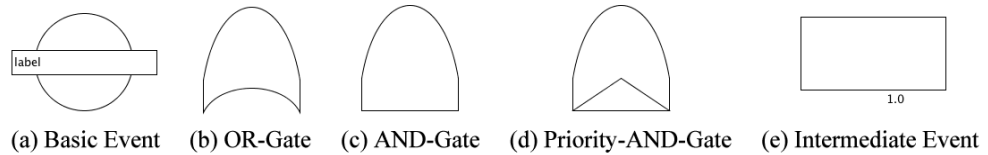


Figure 5.1: Fault Tree Elements

in-depth discussion of fault trees we refer the reader to [6].

Example 1. In Figure 5.2 we present an FT describing the conditions under which a simple 2-out-of-3 fault-tolerant, redundant system is failed. To be failed, components A and B or A and C or B and C , or A , B and C have to be failed. FTs without *PAND* can be rewritten as Boolean formula. The FT of the 2-out-of-3 fault tolerant system can be represented by the following boolean formula: $(A \wedge B) \vee (A \wedge C) \vee (B \wedge C) \vee (A \wedge B \wedge C)$.

By assigning probabilities to the basic events, it is also possible to compute the failure probability of the system. The FT can be analyzed by using simple

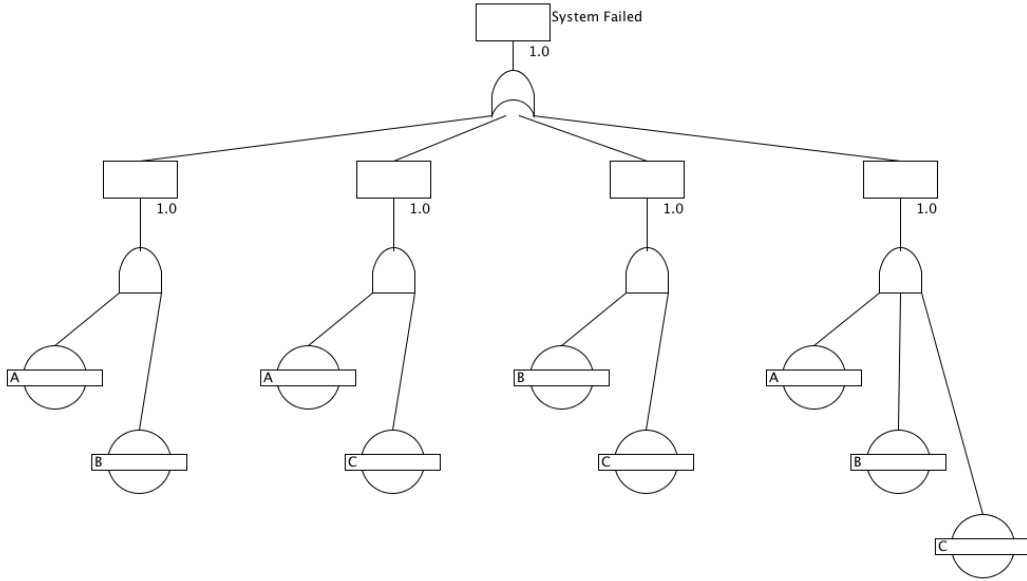


Figure 5.2: Fault Tree Representation of a 2-out-of-3-System Failure

Boolean algebra, set algebra and probability theory if it does not contain *PAND*-gates:

- Let Q be the top event of an *OR*-gate with inputs A and B , then the probability of Q , $P(Q)$, can be computed as follows: $P(Q) = P(A) + P(B) - P(A \cap B)$. If we assume that A and B are independent, this reduces to $P(Q) = P(A) + P(B) - P(A) \cdot P(B)$.
- Now, let Q be the top event of an *AND*-gate, with inputs A and B , the probability of Q , $P(Q)$, is then, assuming A and B to be independent: $P(Q) = P(A) \cdot P(B)$.

Example 2. Consider the 2-out-of-3 system of Example 1. The probability of the top event $Q := \text{system failed}$, $P(Q)$, can be computed as follows, using set algebra and basic probability theory:

$$\begin{aligned}
 P(Q) &= \\
 & (P(A) \cdot P(B)) + (P(A) \cdot P(C)) + (P(B) \cdot P(C)) \\
 & - 2 \cdot (P(A) \cdot P(B) \cdot P(C))
 \end{aligned}$$

If the FT contains at least one *PAND*-gate, this straightforward analysis is no longer feasible since the order of events cannot be captured by the set algebraic approach. In these cases, the FT has to be transformed into a CTMC for

analysis. Another solution to this problem is offered in [37], where an algorithmic approach for the probabilistic analysis of dynamic FTs [38] is presented.

5.3.2 Mapping of Counterexamples to Fault Trees

In order to enable the automatic generation of the fault trees we need to identify what is commonly referred to as basic events. Those are events causing a certain hazard. Subsequently, the combinations of basic events leading to a hazard need to be determined. In the PRISM model that was generated from the UML model, all events, including the basic events, are encoded by the transition labels. Hence, we need to find those basic events in the counterexample that are causing the hazard. The counterexample contains all paths from an initial state to the hazard state. Therefore we can assume that *all* events encountered on the shortest path of the counterexample are necessary to happen in order to cause the hazard. Otherwise, it would be possible to find a shorter path leading to a hazard which omits some of the events of the longer path. Based on this observation we will define the definitions 1 and 2 below. In addition to the identification of the basic events and their combinations, it needs to be checked whether the order of the events to occur is important for the causation of the hazard, or not. This is accomplished by definition 3. Additionally, the path probabilities computed by the stochastic model checker need to be mapped to the fault tree. The rules for the probability mapping are given together with definitions 1, 2 and 3.

In order to compute the fault tree, the QuantUM tool checks for each path in the counterexample whether it is in the fault tree or not. In the following we denote a path in the counterexample either by $p := e_0, e_1, \dots, e_k$, where e_i is the transition label at position i in the path or by the set of transition labels e_0, e_1, \dots, e_k . We define the $=$ operator used in our subsequent definitions as follows: For any paths p_1 and p_2 , $p_1 = p_2$ is true if and only if $\forall s[s \in p_1 \leftrightarrow s \in p_2]$. The \subseteq operator is defined as follows: $p_1 \subseteq p_2$ is true if and only if $\forall s[s \in p_1 \rightarrow s \in p_2]$ Consequently, $p_1 \subset p_2$ is true if and only if $p_1 \subseteq p_2$ is true and $p_1 = p_2$ is false. We define $p_1 \cup p_2 \cup \dots \cup p_n$ to be the union of the set of transition labels e_i of the paths $p_1 \dots p_n$, with each transition label only occurring once. For example:

$$\{\text{A_Failed}, \text{B_Failed}\} \cup \{\text{A_Failed}, \text{C_Failed}\} = \{\text{A_Failed}, \text{B_Failed}, \text{C_Failed}\}$$

We demonstrate the automatic fault tree generation on the running example of a system consisting of the three components A, B and C. The system is down whenever two out of three components A, B and C failed. The generated counterexample comprises the following paths: $\{\text{B_Failed}, \text{C_Failed}\}$, $\{\text{B_Failed},$

C_Failed}, {B_Failed, A_Failed}, {C_Failed, B_Failed}, {A_Failed, B_Failed}, {C_Failed, A_Failed}, {A_Failed, C_Failed}, {B_Failed, C_Failed, A_Failed }, {C_Failed, B_Failed, A_Failed }, {B_Failed, A_Failed, C_Failed },

A path has to fulfill the two definitions defined below in order to belong to the fault tree. Informally, we define the minimal subset definition as follows: If there is a path $p := e_0, e_1, \dots, e_k$ that belongs to the counterexample, it holds for all paths p' in the counterexample that $p' \not\subseteq p$. If $p \subset p'$, then p is added to the fault tree and the probability of p is set to $\text{Prob}(p) = \text{Prob}(p) + \text{Prob}(p')$. More formally, we define:

Definition 1 (Minimal Sub Sets) *Let $CX(TLE)$ be the counterexample for the top level event TLE , and let $p := e_0, e_1, \dots, e_k$ be a path in the counterexample. We define the set of paths belonging to the fault tree of TLE :*

$$FT(TLE) = \{p \in CX(TLE) \mid \forall p' \in CX(TLE)(p' \subseteq p \Rightarrow p' = p)\} \quad (5.1)$$

After applying Def. 1 to our example, the following paths are added to the fault tree: {B_Failed, C_Failed}, {B_Failed, A_Failed} and {C_Failed, A_Failed}. Def. 1 successfully identifies the minimal combinations of basic events that cause the top-level event.

In order to find all combinations of basic events that cause the top level event, we introduce definition 2. Let FT be the set of all paths in the fault tree and $p := e_0, e_1, \dots, e_k$ the path that needs to be checked. p is added to the fault tree if there is no path p' in the fault tree for which $p' = p$ holds, and if there are at least two paths p'_1 and p'_n that are already in the fault tree and for which $p'_1 \subset p \wedge p'_2 \subset p$ and $(p'_1 \cap p'_2) = p$ hold.

Definition 2 (Combinations of basic events) *Let $p := e_0, e_1, \dots, e_k$ be a path in the counterexample, let $SUB(p)$ be the set of all paths $p'_1 \dots p'_n$ that fulfill definition 1 and for which $p'_1 \subset p \dots p'_n \subset p$ holds.*

$$p \in FT(TLE) \text{ iff } \neg(\exists p' \in FT(TLE)(p' = p)) \wedge (\exists k(p'_1 \in SUB(p), \dots, p'_k \in SUB(p) \wedge ((p'_1 \cap \dots \cap p'_k) = p))) \quad (5.2)$$

In our example, Definition 2 adds the path {B_Failed, C_Failed, A_Failed} to the counterexample. Hence, the following paths are now in the fault tree: {B_Failed, C_Failed}, {B_Failed, A_Failed}, {C_Failed, A_Failed} and {B_Failed, C_Failed, A_Failed} .

All paths fulfilling one of the above mentioned definitions are stored in a list. For each path in the list, we check whether the order of the basic events to occur is important or not. Let p be a path that belongs to the counterexample. For all possible subsets of the path p with more than one element, we check whether this subset consisting of labels e_0, \dots, e_k appears in all other paths, belonging to the counterexample, in the same order as in p or not. If for all possible subsets of p the order is irrelevant, that is there exists for each possible order of labels at least one path containing that order, all paths p' that are only permutations of p and hence have the same length as p , are removed from the fault tree and the probability of p is set to $Prob(p) = Prob(p) + Prob(p')$. If for at least one subset of p , consisting of labels e_0, \dots, e_k , the order is relevant, that is for all paths that contain the labels e_0, \dots, e_k , the labels e_0, \dots, e_k appear in the same order as in p we mark the subset to be ordered. The probability of this path was already set by the subset computation.

More formally we define:

Definition 3 (Event Ordering) *Let $CX(TLE)$ be the counterexample for the top level event TLE , let $p := e_0, e_1, \dots, e_k$ be a path in the counterexample and e_0, e_1, \dots, e_k be transition labels out of p and $I(e, p)$ the position of e in p . The order of a set of transition labels e_0, e_1, \dots, e_k with $I(e_0, p) < I(e_1, p) < \dots < I(e_k, p)$ is relevant if and only if*

$$\begin{aligned} & (\forall p' \in CX(TLE) \setminus \{p\}) \\ & ((e_0, e_1, \dots, e_k \notin p') \\ & \vee (I(e_0, p') < I(e_1, p') < \dots < I(e_k, p'))) \end{aligned} \quad (5.3)$$

In our example, Def.3 is not satisfied for any path, since all possible interleaving of the paths are contained in the counterexample. Consequently, the order of the events A_Failed, B_Failed and C_Failed to occur is not relevant, for causing the top level event.

In the fault tree, paths with a length of 1 and hence consisting of only one basic event are represented by the respective basic event. A path with length > 1 , that has no subset of labels marked as ordered is represented by an *AND*-gate. This *AND*-gate connects the basic events belonging to that path. If the whole path is marked as ordered, the path is represented as *PAND*-gate that connects the basic events. If one or more real subsets of the path are marked as ordered, the path is represented by an *AND*-gate that connects the basic events not in the marked subsets and a *PAND*-gate that connects the basic events in the subset marked as ordered.

Figure 5.3 shows the fault tree of the running example.

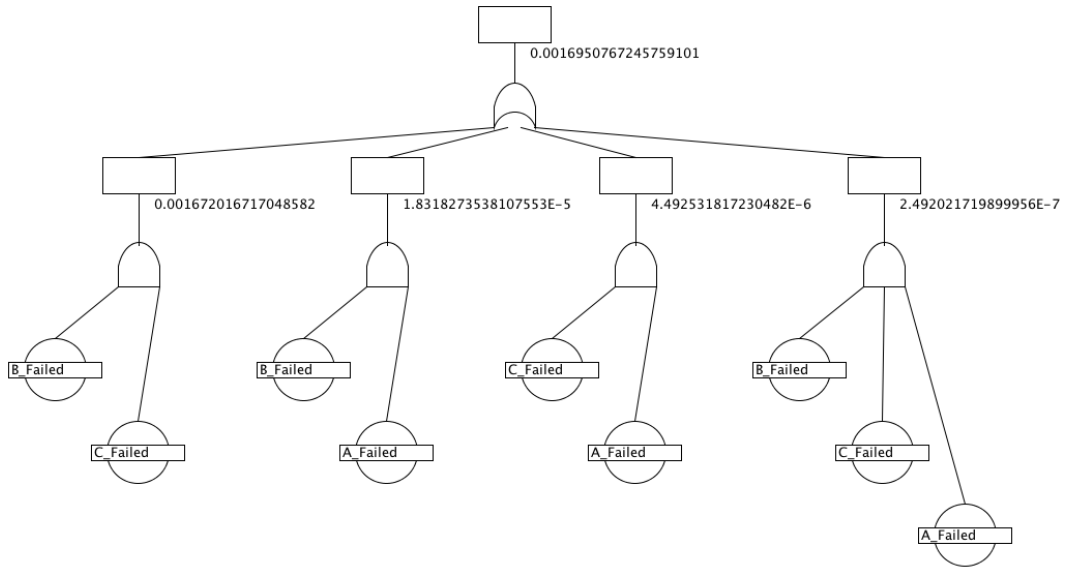


Figure 5.3: Fault Tree Representation of the running example.

The probability values of the *AND*-gates are the corresponding probabilities of the paths that they represent. In order to display the probabilities in the graphical representation of the fault tree, we add an intermediate event as parent node for each *AND*-gate. The resulting intermediate events are then connected by an *OR*-gate that leads to the top event, representing the hazard. Since the path probabilities are calculated for a path starting from an initial state to the hazard state, the probability of the *OR*-gate is the sum of the probability of all child elements.

5.3.3 Complexity of the algorithm

In Figure 5.4 a sketch of the algorithm, which is used to check Def. 1 is shown. The worst case runtime of the algorithm is in $O(n^2)$ where n is the number of paths in the counterexample. A sketch of the algorithm which checks Def. 2 is shown in Fig. 5.5. Since it is theoretically possible that all paths of the counterexample are in the fault tree The worst case runtime of this algorithm is also in $O(n^2)$ where n is the number of paths in the counterexample. The same worst case running time holds for the algorithm used to check Def. 2 which is shown in Figure 5.6. Consequently, the worst case complexity of the whole algorithm is in $O(n^2)$.

```

List FaultTree;

//Check Definition 1
FORALL(paths p in CX)
{
  bool isMin = true
  FORALL(paths p' in CX)
  {
    IF p'.subsetOF(p)
      -> isMin = false
  }
  IF isMin -> FaultTree.add(p)
}

```

Figure 5.4: Algorithm sketch of Definition 1

```

//Check Definition 2
FORALL(paths p in CX)
{
  Set Of Paths: SubSet;
  FORALL(paths p' in FaultTree )
  {
    IF p'.subsetOF(p) -> SubSet.add(p')
  }
  FOR(i = 2 TO length(p))
  {
    CartesianProduct = SubSet
    FOR(u = 0 TO i)
    {
      CartesianProduct = (CartesianProduct
        X CartesianProduct)
    }
    IF p isIn(CartesianProduct) -> FaultTree.add(p)
  }
}

```

Figure 5.5: Algorithm sketch of Definition 2

5.3.4 Scalability of the approach

Although, the worst case running time of the algorithm is quadratic in the size of the counterexample, our case studies presented in Chapter 7 show that the fault tree computation finishes in several seconds, while the computation of the counterexample took several minutes. Hence, the limiting factor of our approach is the time needed for the computation of the counterexample.

5.3.5 Correctness and Completeness of the Approach

In the following we show that the generated fault tree is indeed a correct and complete fault tree. In [39] a formal semantics for fault trees in modal μ -calculus is given (see Fig. 5.7). We will give here a brief introduction in the fragment of the modal μ -calculus used, for a more comprehensive description we refer to [40, 41]. The model μ -calculus is temporal logic to express behavioral properties. The formula $(a_1 \wedge a_2) \Rightarrow \text{even}(a_3)$ says whenever a_1 and a_2 holds, this implies that eventually a_3 will hold.

We prove the correctness of our fault tree generation algorithm by proving the consistency of our mappings with the modal μ -calculus semantics. In the following we assume that if a path p is in the counterexample, there is no path $p' = p$ that is not in the counterexample, because this would imply that a path is at the same time fulfilling the formula φ , describing the top level event, and not fulfilling the formula φ which is a contradiction.

5.3. FROM PROBABILISTIC COUNTEREXAMPLES TO FAULT TREES 53

```

//Check Definition 3
FORALL(paths p in FaultTree with length(p) > 1)
{
  bool orderRelevant = true
  FORALL(paths p' in CX)
  {
    FOR(i = 0 TO length(p))
    {
      FOR(u = 0 TO length(p))
      {
        IF p'.contains(p.labelAt(i)) & p'.contains(p.labelAt(u))
          & ((p'.indexOf(p.labelAt(i)) <= p'.indexOf(p.labelAt(u)) & i > u)
            | (p'.indexOf(p.labelAt(i)) > p'.indexOf(p.labelAt(u)) & i <= u))
          -> orderRelevant = false
      }
    }
  }
  p.setOrderRelevant(orderRelevant)
}

```

Figure 5.6: Algorithm sketch of Definition 3

$$\begin{aligned}
\llbracket +(in_1, in_2, out) \rrbracket &\stackrel{\text{def}}{=} (in_1 \vee in_2) \Rightarrow \mathbf{even}(out) \\
\llbracket \bullet(in_1, in_2, out) \rrbracket &\stackrel{\text{def}}{=} (in_1 \wedge in_2) \Rightarrow \mathbf{even}(out)
\end{aligned}$$

Figure 5.7: Semantics of fault trees in μ -calculus

Correctness of the fault tree

The semantics of an AND-gate (\bullet) is defined by the modal μ -calculus formula

$$\llbracket \bullet(in_1, in_2, out) \rrbracket \stackrel{\text{def}}{=} (in_1 \wedge in_2) \Rightarrow \mathbf{even}(out)$$

Consequently, an AND-gate connecting the events in_1 , in_2 and the top event out is correct if all paths containing the events $e_1 = in_1$ and $e_2 = in_2$ eventually lead to the event out . We generate an AND-gate connected to all events in a path p that fulfills the definitions 1 and 2. This path p leads to the top level event out , because otherwise it would not be in the counterexample. Hence for each path p containing the events $e_1 = in_1, e_2 = in_2, \dots, e_n = in_n$ the formula $(in_1 \wedge in_2 \wedge \dots \wedge in_n) \Rightarrow \mathbf{even}(out)$ holds. Therefore, it is proven that all AND-gates in the fault tree are correct.

The semantics of a PAND-gate is the same as the AND-gate semantics with the additional constraint on the order of the events. If the event in_1 always occurs before in_2 and they both can be connected by a valid AND-gate, the AND-gate is also a valid PAND-gate.

Similarly, the semantics of an OR-gate ($+$) is defined by the modal μ -calculus

formula

$$\llbracket +(in_1, in_2, out) \rrbracket \stackrel{\text{def}}{=} (in_1 \vee in_2) \Rightarrow \text{even}(out)$$

Therefore, an OR-gate connecting the events in_1 , in_2 and the top event out is correct if all paths containing at least one of the events $e_1 = in_1$ and $e_2 = in_2$ eventually lead to the event out . We generate an OR-gate that connects all AND-gates, representing the paths leading to the top level event TLE . Therefore the formula,

$$\begin{aligned} & ((in_{1,1} \wedge in_{1,2}) \Rightarrow \text{even}(TLE)) \vee \\ & ((in_{2,1} \wedge in_{2,2}) \Rightarrow \text{even}(TLE)) \vee \dots \vee ((in_{n,1} \wedge in_{n,2}) \Rightarrow \text{even}(TLE)) \end{aligned}$$

holds.

Consequently, the formula

$$\begin{aligned} & ((in_{1,1} \wedge in_{1,2}) \vee (in_{2,1} \wedge in_{2,2}) \vee \dots \vee (in_{n,1} \wedge in_{n,2})) \Rightarrow \text{even}(TLE) = \\ & (\text{even}(TLE) \vee \text{even}(TLE) \vee \dots \vee \text{even}(TLE)) \Rightarrow \text{even}(TLE) \end{aligned}$$

holds. Therefore, it is proven that all OR-gates in the fault tree are correct.

Completeness of the fault tree.

We define a fault tree to be complete, if it consists of all basic events that can lead to the top level event. All paths leading to the top level event are in the counterexample. The definitions 1 and 2 add the minimal paths containing the basic events to the fault tree. We assume that there is an incomplete fault tree generated by our algorithm. Hence, there has to be a path in the counterexample, that is not marked as belonging to the fault tree and which contains a combination of basic events that are all needed to cause the top level event. This path will only be marked as not belonging to the fault tree if a subset of the path is already in the fault tree, hence not all basic events of the path are required to cause the top level event. Consequently the fault tree is complete.

5.4 From Probabilistic Counterexamples to UML

5.4.1 UML Sequence Diagrams

A sequence diagram in Unified Modeling Language (UML) is a kind of interaction diagram, similar to Message Sequence Charts [42], that shows how processes interact with one another and in what order. An example of a UML sequence diagram can be found in Figure 5.8. In a sequence diagram different processes

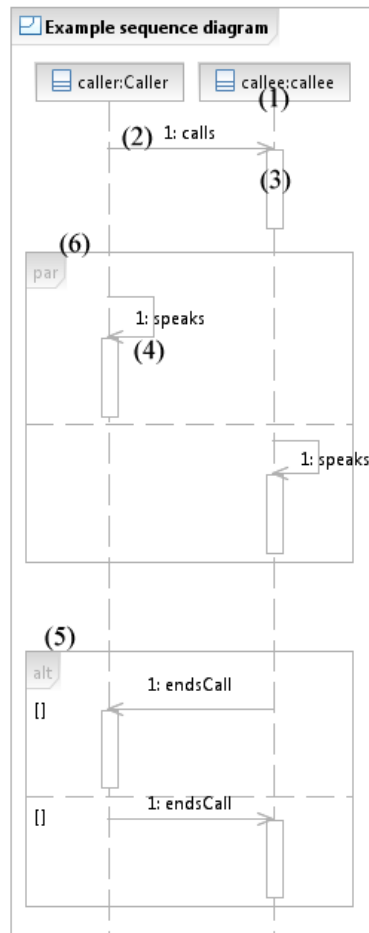


Figure 5.8: Example of a UML sequence diagram.

or objects that live simultaneously are represented as parallel vertical lines, so called lifelines (cf. (1) in Figure 5.8). The messages exchanged between those objects are represented by horizontal arrows with the message name written above them (cf. (2) in Figure 5.8). Activation boxes, or method-call boxes, are opaque rectangles drawn on top of lifelines to represent that processes are being performed in response to the message (cf. (3) in Figure 5.8). Objects calling

methods on themselves use messages and add new activation boxes on top of any others to indicate a further level of processing (cf. (4) in Figure 5.8). Usually, the order of the messages is sequential from top to bottom and indicated by a number in front of the message name. So called combined interaction fragments can be used to change this sequential order and to display alternative (*alt*) sequences (cf. (5) in Figure 5.8) or sequences that run concurrently (*par*) (cf. (6) in Figure 5.8).

Since, paths in a counterexample represent sequences of the system executions, UML sequence diagrams are an obvious choice to display probabilistic counterexamples.

5.4.2 Mapping of Counterexamples onto UML Sequence Diagrams

Theoretically it is possible to map all paths of the counterexample directly to the sequence diagram. But because this mapping would result in a sequence diagram with hundreds of alternative sequences, it is necessary to extract those paths from the counterexample that represent the minimal failure configurations (i.e. basic events causing the error). This extraction is done by our fault tree computation algorithm.

Each path in the counterexample is an alternative execution path of the system with the probability P . In order to express alternative executions in sequence diagrams, an *alt* combined interaction fragment is used. The probability of the path is given in the name of the corresponding *alt* combined interaction fragment. For each *QUMComponent* of the model we add a lifeline representing the component. We add a function *transition*("source state", "target state") to each component, in order to visualize the transitions that are taken inside the component. A call to the *transition* function is added for each transition in the fault tree. If the transition is an operation call to an operation of a *QUMComponent*, an operation call from the caller component to the callee component is added to the sequence diagram. If the paths of the counterexample would be mapped directly to the sequence diagram, all paths, including the possible interleavings of the paths, would be added to the sequence diagram. When the fault tree computation is used to filter the paths, all paths that are only an interleaving of each other are represented by one *AND*-gate. In order to reflect the interleaving in the sequence diagram, *AND*-gates are represented by *par* combined fragments where each compartment represents one branch of the *AND*-gate. The compartments of the *par* combined fragment can be either executed in parallel or in any other possible concurrent interleaving. *PAND*-gates are mapped to a sequence of events representing the order of the events in the

PAND-gate.

To illustrate the mapping we show the sequence diagram of our running example with components A, B and C in Fig. 5.9. The corresponding fault tree is shown in Fig. 5.3.

In order to display the sequence diagrams, the QuantUM tool appends the XMI code of the diagram to the XMI file that contains the UML model. Thus, it can easily be imported in the UML CASE-tool.

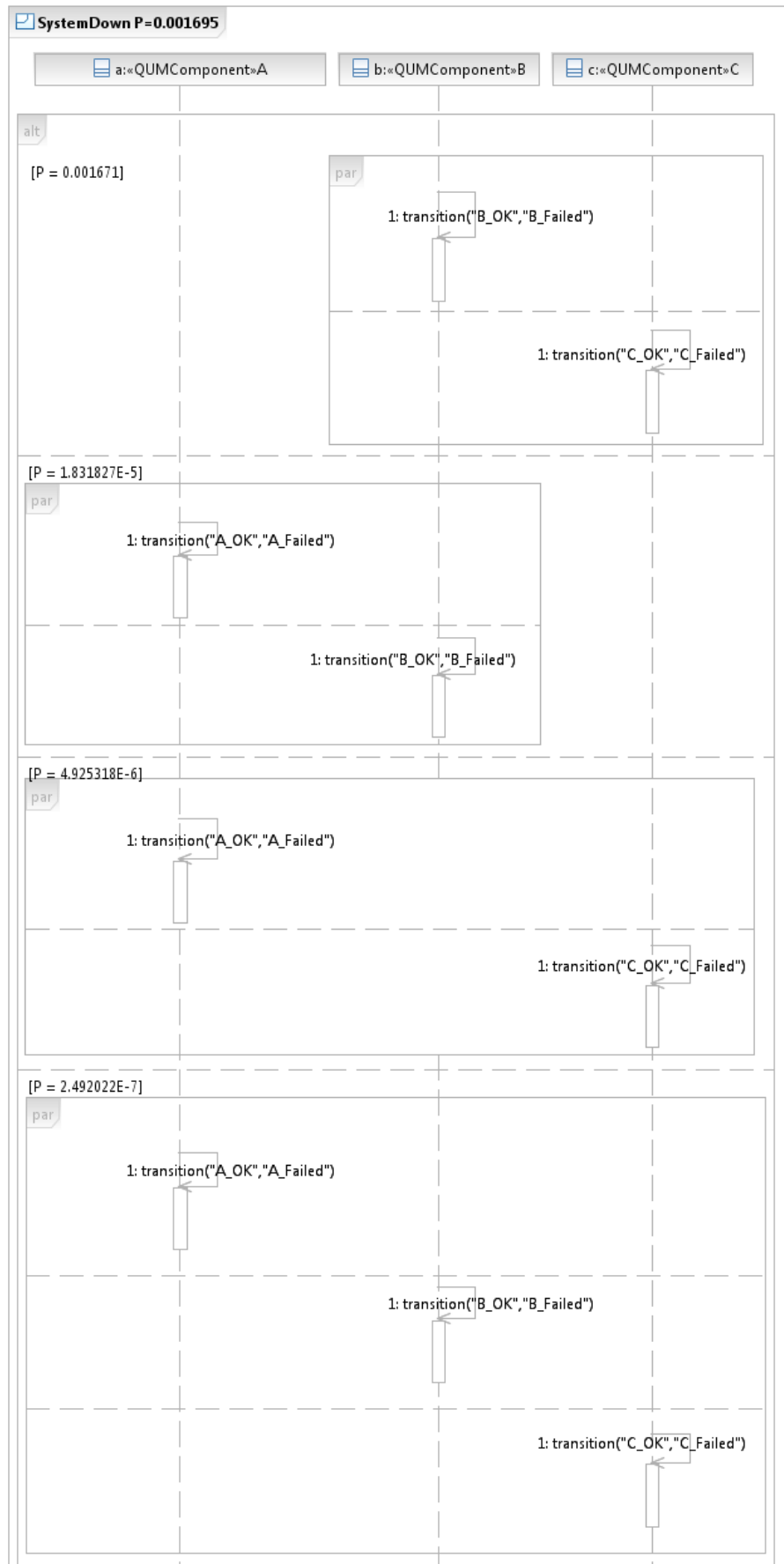


Figure 5.9: UML sequence diagram of the running example fault tree.

Chapter 6

The QuantUM Tool

The architecture of the QuantUM tool is depicted in Fig. 6.1. The UML model is specified in a UML CASE tool, where it can also be annotated with the QuantUM profile. Subsequently the annotated model is exported in the XML Metadata Interchange (XMI) format [7] which is the standard format for exchanging UML models. In our case studies we used the IBM Rational Software Architect¹, for the UML modeling part, but in principle any other UML tool that can export a model to XMI can be used. The QuantUM tool then parses the XMI file and translates the model into the PRISM language. Additionally it generates a set of CSL formulas that can be used for the analysis. To facilitate the analysis we integrated an instance of the probabilistic model checker PRISM into the QuantUM tool. Furthermore, our counterexample generation tool DiPro [5] is directly integrated into QuantUM. This allows us to directly compute the fault tree. Both the execution of PRISM and DiPro are hidden from the user and require no user interaction. The resulting fault tree can then be mapped on a UML sequence diagram which is stored in the XMI format. The file containing the UML sequence diagram can then be displayed in the UML CASE tool.

¹<http://www.ibm.com/software/rational/>

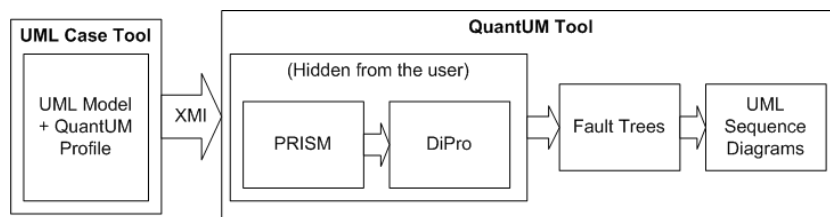


Figure 6.1: The QuantUM Tool

Chapter 7

Case Studies

7.1 Airbag Control Unit

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 [1]. 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. An airbag system can be divided into three major parts: sensors, crash evaluation and actuators. An impact is detected by acceleration sensors (front/rear/side impact) and additional pressure sensors (side impact). Angular rate or roll rate sensors are used to detect rollover accidents. The sensor information is evaluated by a microcontroller which decides whether the sensed acceleration corresponds to a crash situation or not. The deployment of the airbags is only activated if the microcontroller decides that there was indeed a critical crash.

The airbag system architecture that we consider consists of two acceleration sensors whose task it is to detect front or rear crashes, one microcontroller to perform the crash evaluation, and an actuator that controls the deployment of the airbag.

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 microcon-

troller 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 7.1 shows the class diagram describing the structure of the system. The *MainSensor*, *SafetySensor*, *MicroController*, *FET* and *FASIC* are modeled as classes and are tagged with the stereotype *QUMComponent*.

The *MicroController QUMComponent* comprises one state machine representing its normal behavior (cf. Fig. 7.2) and one state machine representing the failure pattern, which is shown in Fig. 7.3. If the *MicroController* is in the normal state machine, it evaluates every 20ms whether there is a crash or not. If the *MainSensor* and the *SafetySensor* where above the threshold (*acceleration* > 3) for three consecutive evaluations, there is a crash and the state *Crash* is entered. In the *Crash* state, first, the *FASIC* is armed, then the *FET* is enabled and, finally, the *FASIC* is fired. The failure pattern of the *MicroController* can be entered at any time, with the rate specified for the *QUMAbstractFailureTransition*. As soon as it is entered it will execute the fire sequence: enable *FET*, arm *FASIC*, and fire *FASIC*.

Figure 7.4 shows the normal behavior state machine of the *FASIC*. It stays in the *Idle* state until the operation *armFASIC()* is called, which executes the transition into the *Armed* state, where the variable *fasicArmed* is set to true. If the state machine is in the *Armed* state, the operation *fireFASIC()* is called, and the variable *fetEnabled* is true, the transition to the state *Fired* is taken and the variable *fasicFired* is set to true. The state machine representing the failure pattern of the *FASIC* is shown in Figure 7.5. The state machine will be entered with the rate specified for the *QUMAbstractFailureTransition*. If it is entered, one of the three failure modes will be entered. The failure mode *FASICShortage* is entered with the rate specified in the *QUMFailureTransition FasicShortage* and models a short circuit of the *FASIC*, where the protection mechanism of the *FET* is disabled and the airbag is fired. *FASICStuckLow* represents the failure mode, where the variable *fasicFired* is always false and hence the airbag will not be fired. This failure mode is entered with the rate specified in the *QUMFailureTransition FasicStuckLow*. The failure mode *FASICStuckHigh* represents the failure mode, where the variable *fasicFired* is set to true and hence the airbag will be fired, if the *FET* was enabled before. Hence, this failure mode is entered with the rate specified in the *QUMFailureTransition FasicStuckHigh*,

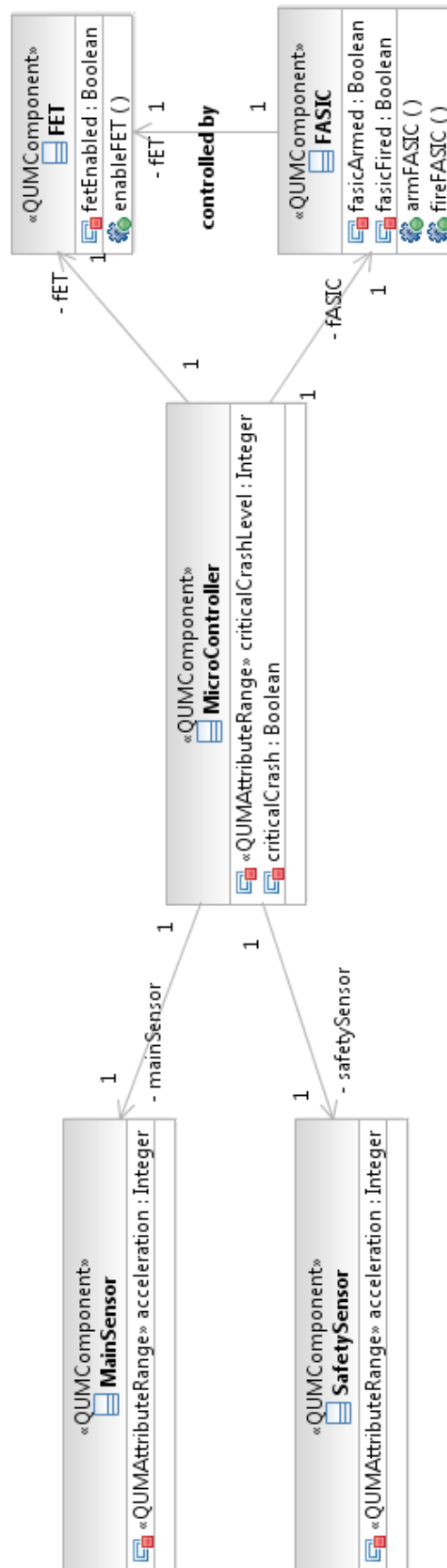


Figure 7.1: Class diagram of the airbag system.

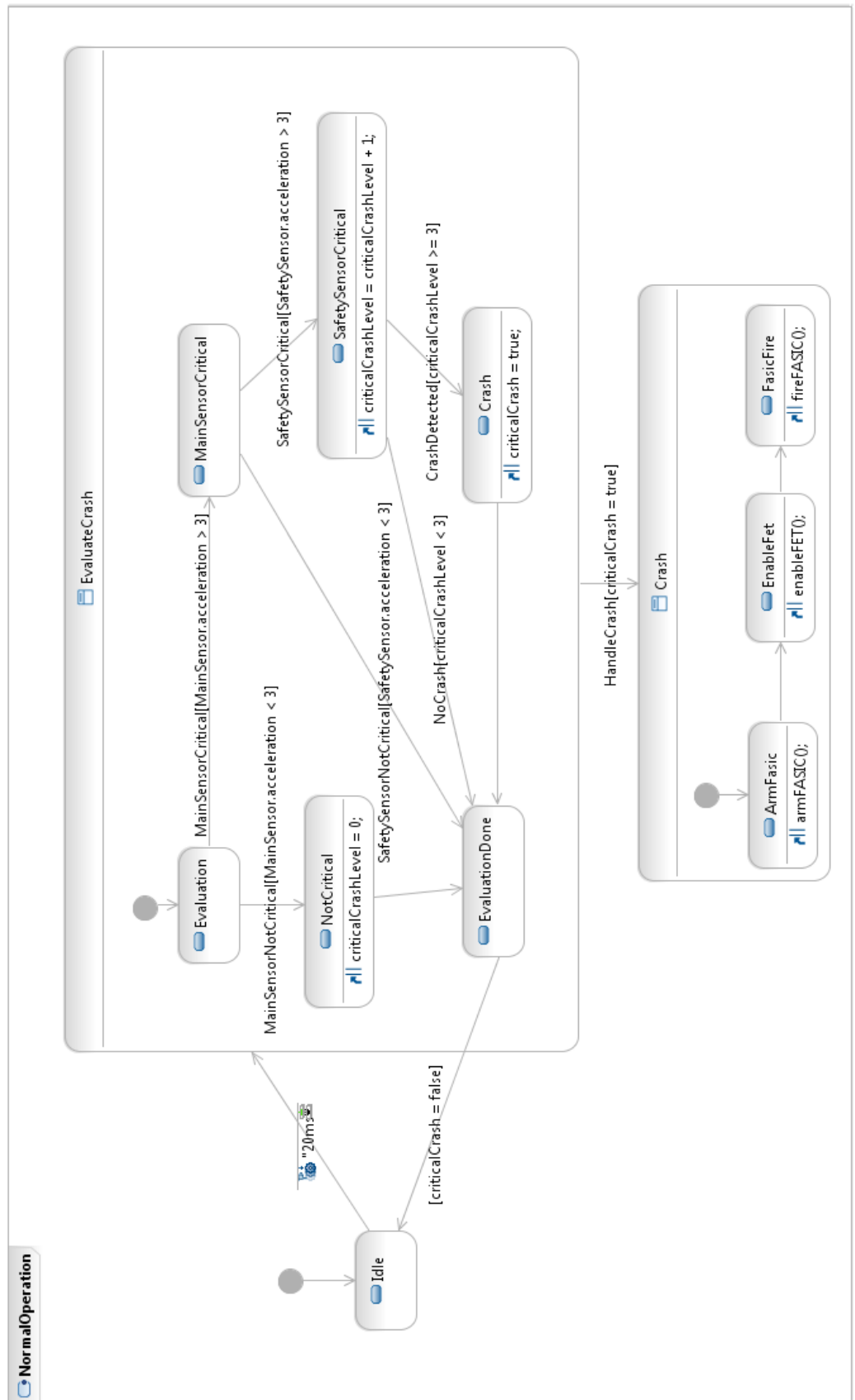


Figure 7.2: State machine representing the normal behavior of the microcontroller.

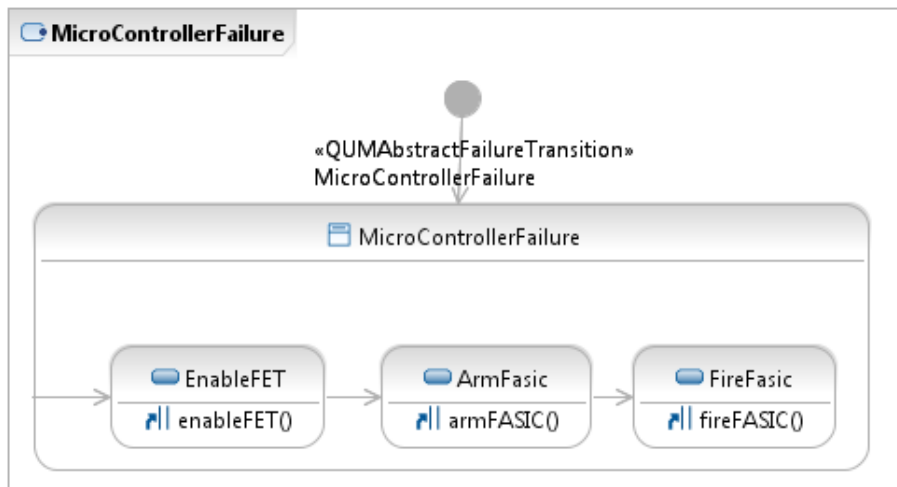


Figure 7.3: State machine representing the failure pattern of the microcontroller.

if the variable *fetEnabled* has been set to true before.

Since, we are interested in the analysis of an inadvertent deployment of the airbag, that is the airbag is fired although there is no crash, we tagged all states, where the variable *fasicFired* is set to true, with the stereotype *QUM-StateConfiguration* with the name *inadvertent_deployment* and the *or*-operator (cf. Figure 7.5 and 7.6). Consequently, the *QUMStateConfiguration* is true whenever the *FASIC* component is in a state where the variable *fasicFired* is true.

The state machines of the *FET* are show in Figure 7.7 (normal behavior) and Figure 7.8 (failure pattern). If the *FET* is in the normal behavior state machine, it is initially in the *Disabled* state, where the variable *fetEnabled* is false. If the operation *enableFET()* is called, when the *FET* is in the *Disabled* state it will take the transition to the *Enabled* state, where the variable *fetEnabled* is set to true. If the operation *enableFET()* is called, when the *FET* is in the *Enabled* state it will take the transition to the *Disabled* state, where the variable *fetEnabled* is set to false. The failure pattern state machine will be entered with the rate defined in the *QUMAbstractFailureTransition*. There are two possible failure modes, namely *FETStuckHigh*, where the variable *fetEnabled* is set to true and *FETStuckLow* where the variable *fetEnabled* is set to false. One of theses failure modes will be entered with the rate specified in the corresponding *QUMFailureTransition*.

The *MainSensor* and *SafetySensor* are only acting as environment input and hence only comprise one normal behavior state machine, each. The state machine of the *MainSensor*, which is identical to the one of the *SafetySensor*

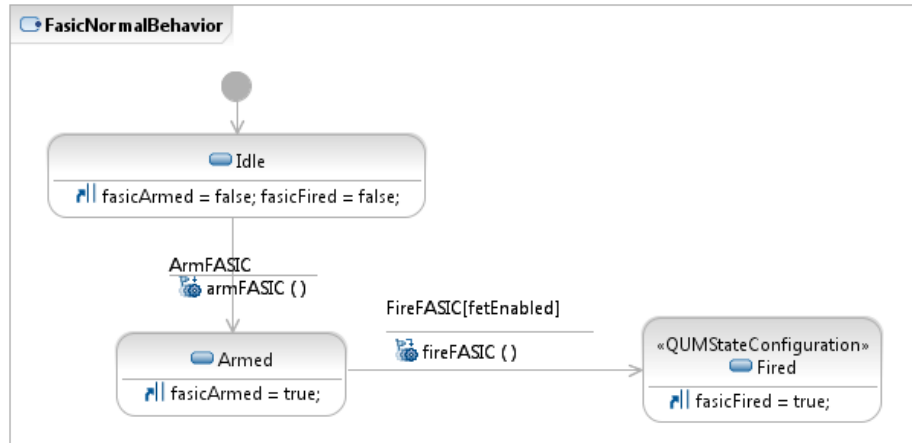


Figure 7.4: State machine representing the normal behavior of the FASIC.

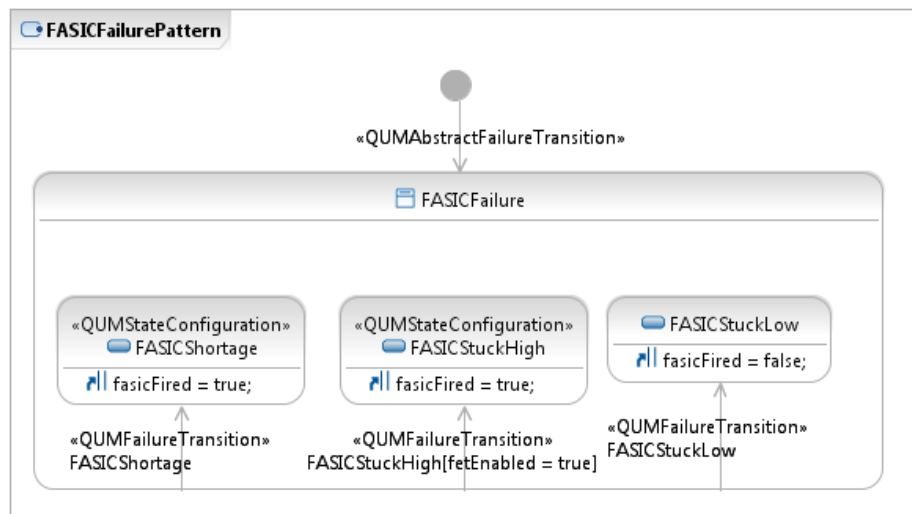


Figure 7.5: State machine representing the failure pattern of the FASIC.

QUMStateConfiguration	
name	inadvertent_deployment
operator	1 - OR

Figure 7.6: Properties of the QUMStateConfiguration.

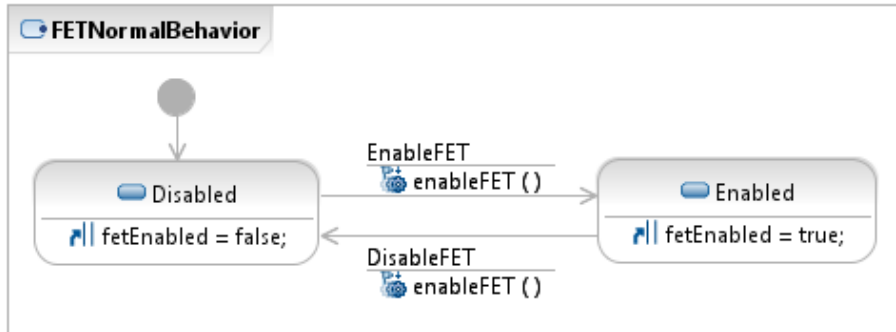


Figure 7.7: State machine representing the normal behavior of the FET.

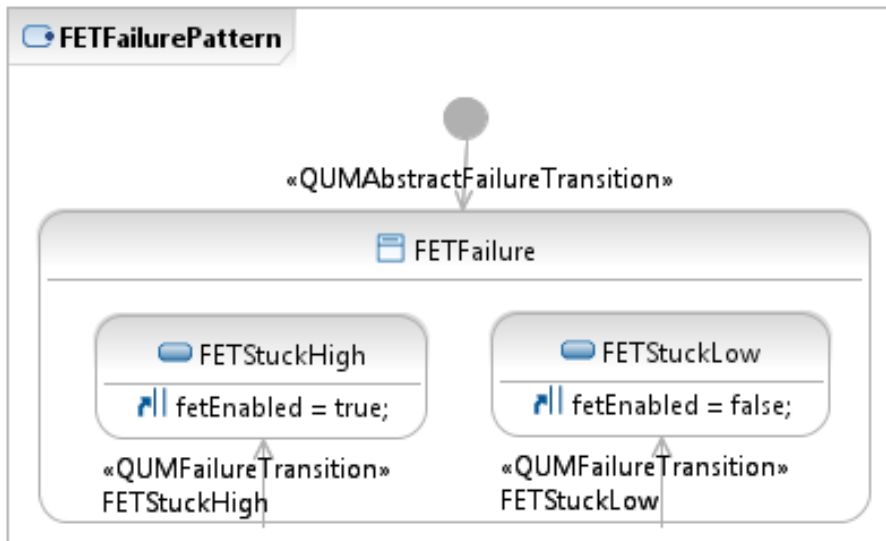


Figure 7.8: State machine representing the failure pattern of the FET.

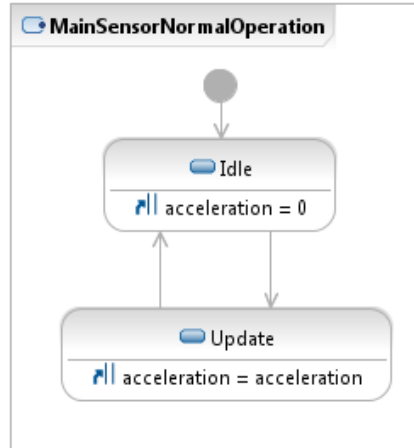


Figure 7.9: State machine representing the normal operation of the MainSensor.

except for the name, is shown in Figure 7.9. We do not specify any failure patterns for the sensors, because we only want to analyze the correct behavior of the *MicroController*, *FASIC* and *FET*. Since, we further are interested in the analysis of an inadvertent deployment of the airbag, that is the airbag is fired although there is no crash, we modeled the state machine in such a way that the acceleration is always zero.

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

$$P_{=?}[(\text{true})U^{<=T}(\text{inadvertent_deployment})]$$

where *inadvertent_deployment* is replaced by the state formula

$$(\text{FASIC_states} = 3 | \text{FASIC_states} = 9 | \text{FASIC_states} = 11)$$

which identifies all states in the *QUMStateConfiguration* *inadvertent_deployment* and *T* represents the mission time. Since, the acceleration value of the sensor state machines is always zero, the formula

$$P_{=?}[(\text{true})U^{<=T}(\text{inadvertent_deployment})]$$

calculates the probability of the airbag being deployed, during the mission 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 7.10. The experiments were 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 7.10: Experiment results for $T=10$, $T=100$ and $T=1000$.

Figure 7.10 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 seems 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 caused by background processes on the experiment PC.

Figure 7.11 shows the fault tree generated from the counterexample for the formula $P_{=?}[(\text{true})U^{<=10}(\text{inadvertent_deployment})]$. 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* 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*. If the basic event *FETStuckHigh* occurs prior to the *MicroControllerFailure* the sequence *armFASIC*, and *fireFASIC* occurring after the *MicroControllerFailure* event suffices.

It is also easy to see that the combination of the basic events *FETStuckHigh* and *FASICStuckHigh* only lead to an inadvertent deployment of the airbag if the basic event *FETStuckHigh* occurs prior to the basic event *FASICStuckHigh*. This is due to the fact that the FET controls the power supply for the airbag

squibs. Consequently, if the FET is not armed, that is the FET-Pin is not high, the airbag squib does not have enough electrical power to ignite the airbag.

The case study shows that the fault tree is a compact and concise visualization of the counterexample which allows for an easy identification of the basic events that cause the inadvertent deployment of the airbag and their corresponding probabilities. If the order of the events is important, this can be seen in the fault tree by the *PAND*-gate, in the counterexample one would have to manually compare the order of the events in all 738 paths, which is a tedious and time consuming task.

The Figures 7.12 and 7.13 show the UML sequence diagram which visualizes the counterexample for the formula $P_{=?}[(\text{true})U^{<=10}(\text{inadvertent_deployment})]$. The generation of the XMI code of the sequence diagram took less than 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 the visualization of the counterexample as sequence diagram, is that operation calls can be shown. In the lower *alt*-compartment of Figure 7.12 for instance, it is easy to see how after a failure of the microcontroller the operations *enableFET()*, *armFASIC()*, and *fireFASIC()* are called.

7.2 Train Odometer Controller

This case study of a train odometer system taken from [43]. The train odometer system consists of two independent sensors used to measure the speed and position of a train. A wheel sensor is mounted to an unpowered wheel of the train to count the number of revolutions. A radar sensor determines the current speed by evaluating the Doppler shift of the reflected radar signal. We consider transient faults for both sensors. For example water on or beside the track could interfere with the detection of the reflected signal and thus cause a transient fault in the measurement of the radar sensor. Similarly, skidding of the wheel affects the wheel sensor. Due to the sensor redundancy the system is robust against faults of a single sensor. However, it needs to be detectable by other components in the train, when one of the sensors provides invalid data. For this purpose a monitor continuously checks the status of both sensors. Whenever either the wheel sensor or the radar sensor are failed, this is detected by the monitor and the corresponding status variable (*wsensor* or *rsensor*) is set to false. This information can be used by other train components that have to disregard temporary erroneous sensor data. Due to the robustness against single faults and since both sensor faults are transient the system even can recover completely from such a situation. If both sensors fail the monitor initiates an

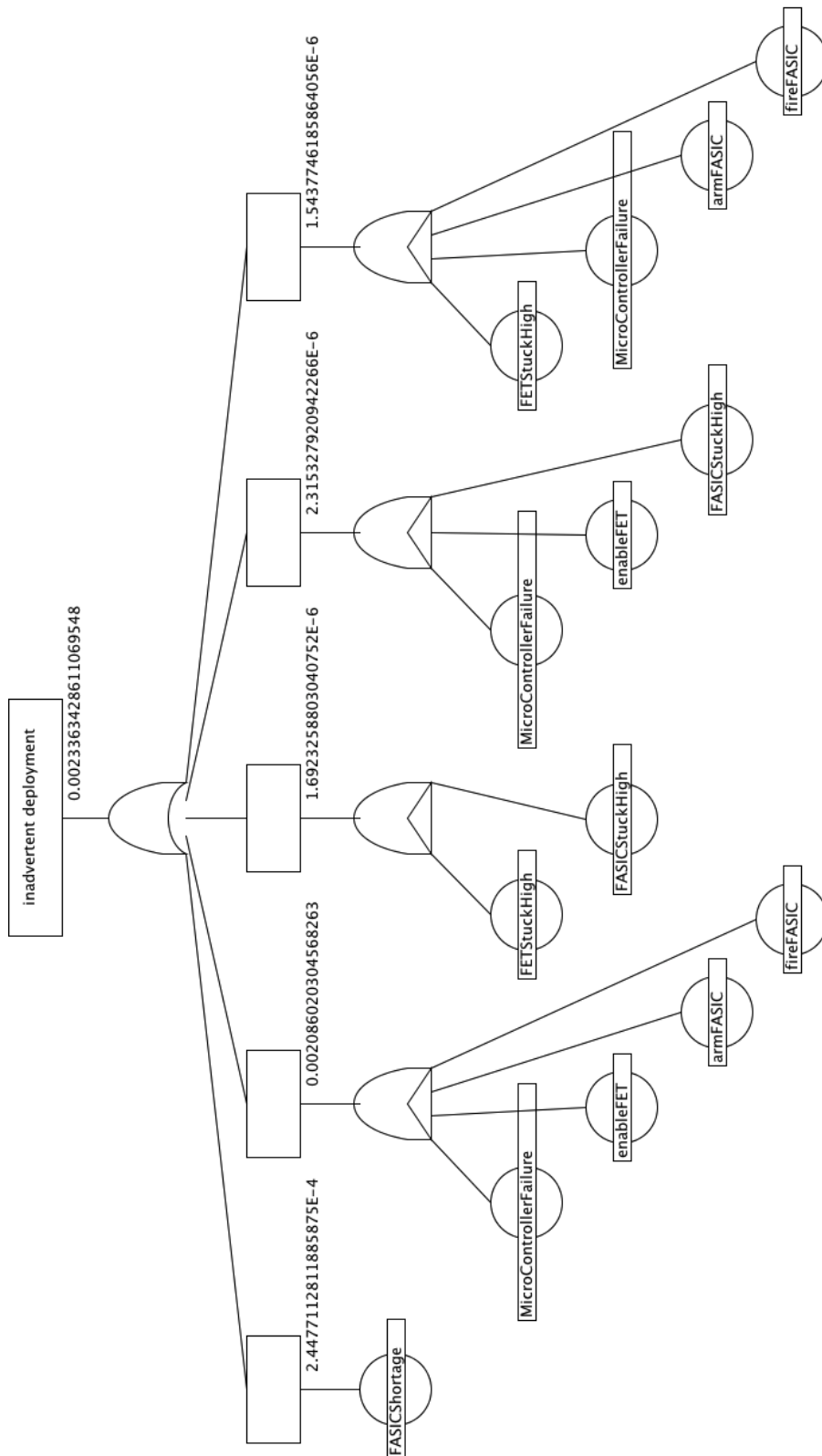


Figure 7.11: Fault tree for the *QUMStateConfiguration inadvertent_deployment* ($T = 10$).

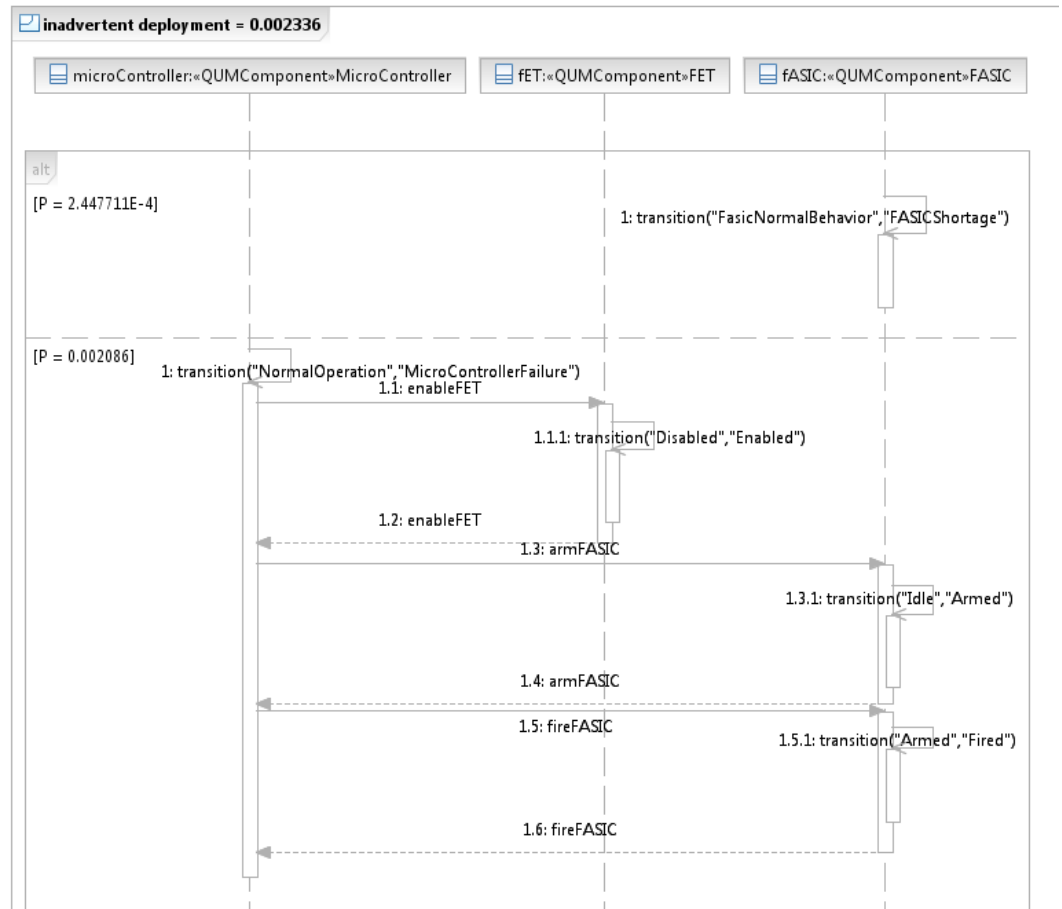


Figure 7.12: UML sequence diagram for the *QUMStateConfiguration inadvertent_deployment* ($T = 10$) (part 1 of 2).

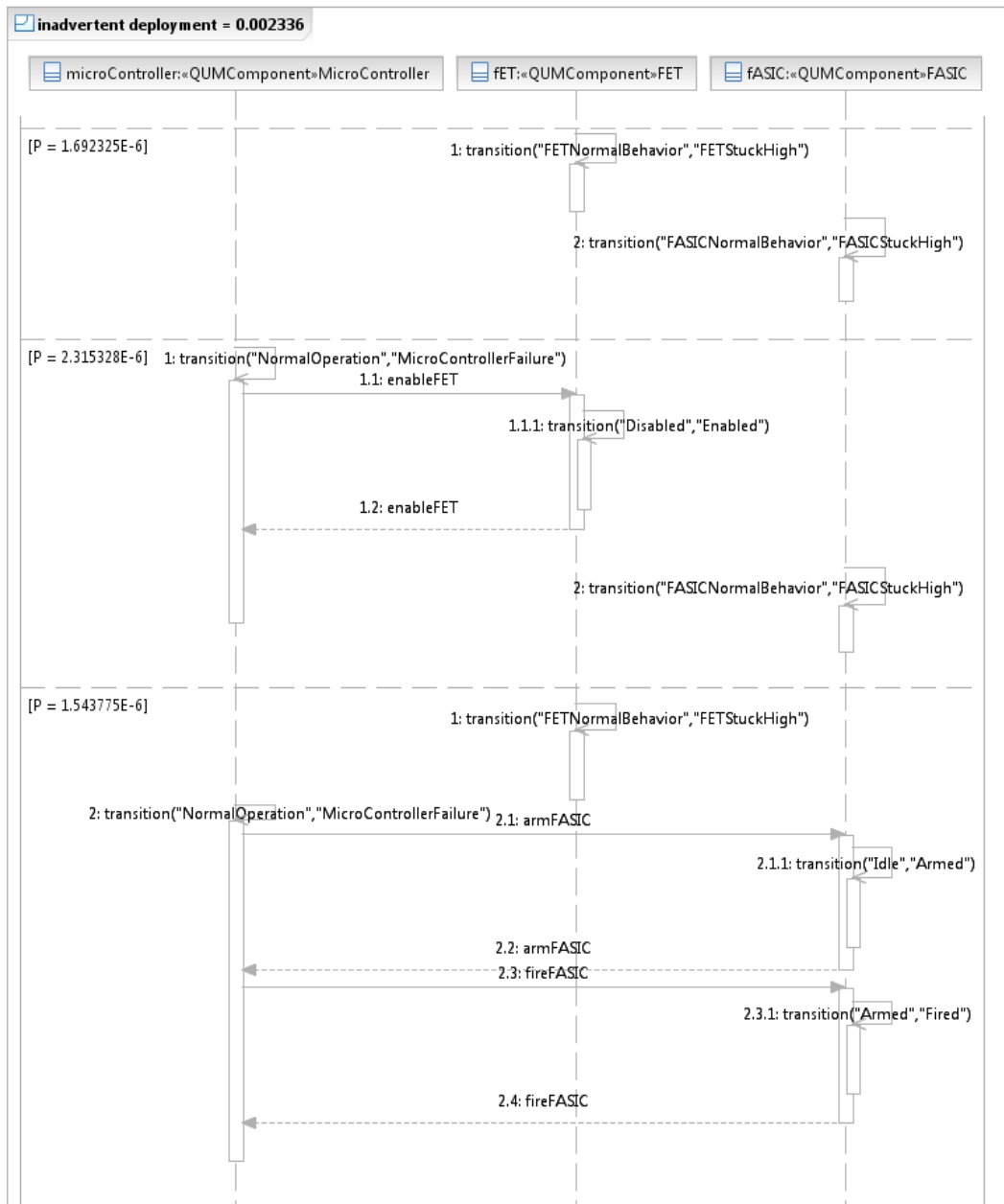


Figure 7.13: UML sequence diagram for the *QUMStateConfiguration inadvertent_deployment* (T = 10) (part 2 of 2).

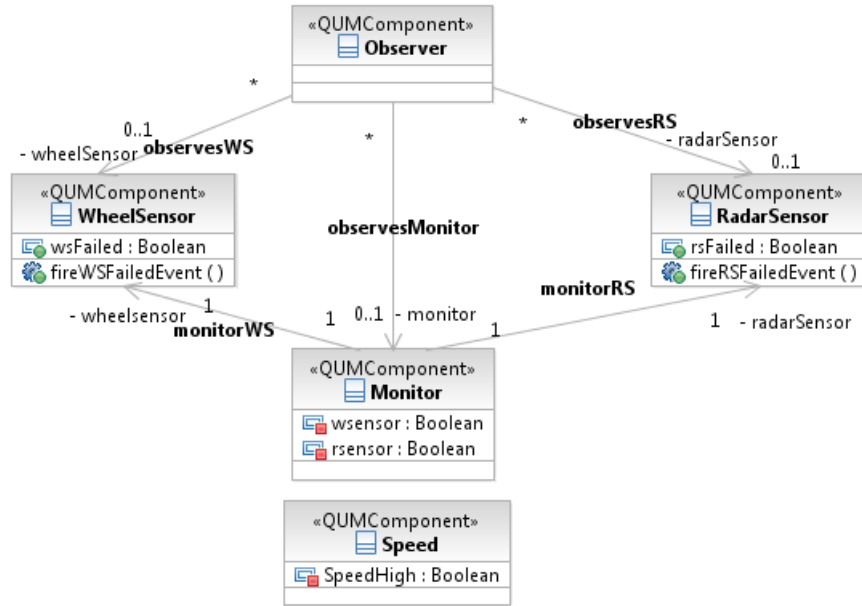


Figure 7.14: Class diagram of the train odometer.

emergency brake maneuver, and the system is brought into a safe state. Only if the monitor fails, any subsequent faults in the sensors will no longer be detected. Since now the train may be guided by invalid speed and position information such situations are safety critical.

We modeled the train odometer with the CASE tool IBM Rational Software Architect. The class diagram of the system can be seen in Figure 7.14. In addition to the *QUMComponents* *WheelSensor*, *RadarSensor* and *Monitor*, we added the *QUMComponent* *Speed* and *Observer*. The *QUMComponent* *Speed* is needed, since the failure of the *WheelSensor* is depending on the current speed of the train. The *Observer* components observes the *WheelSensor*, the *RadarSensor* and the *Monitor* and checks whether the failure of one of the sensors was correctly recognized by the *Monitor*.

The *WheelSensor* *QUMComponent* comprises one state machine representing its normal behavior (Figure. 7.15) and one state machine representing the failure pattern (Figure. 7.16). Since we are not interested in the functionality of the sensor itself, but only in the correct error detection, we can model the normal behavior of the *WheelSensor* with one state indicating that the *WheelSensor* is working. The failure rates of the *WheelSensor* are depending on speed of the train. If the speed is above a certain threshold, represented by *SpeedHigh* = *true*, the *WheelSensor* enters with the rate specified by the *QUMFailureTransition* *Wait_W_Fail_F* the failure state *WF*, where the variable *wsFailed* is set to

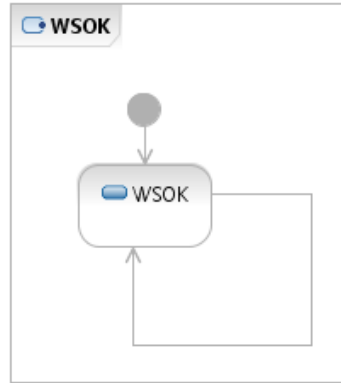


Figure 7.15: State machine representing the normal behavior of the WheelSensor.

true and the operation *fireWSFailedEvent()* is called. Similarly, if the speed is below a certain threshold (*SpeedHigh = false*) the *WheelSensor* enters the failure state *WF* with the rate specified by the *QUMFailureTransition Wait_W_Fail.S*. Once, the *WheelSensor* is in the failure state *WF*, it will return to the normal operation state machine with the rate specified in the *QUMStochasticTransition Wait_W_OK*. On exit of the state *WF*, the variable *wsFailed* is set to false, to indicate that the normal operation of the *WheelSensor* is recovered.

The state machine shown in Fig. 7.17 represents the normal behavior of the *RadarSensor*. For the same reason like for the *WheelSensor*, one state indicating that the *RadarSensor* is working suffices. The failure pattern of the *RadarSensor* is entered with the rate specified by the *QUMFailureTransition Wait_R_Fail*. When the state *RF* is entered the variable *rsFailed* is set to true and the operation *fireRSFailedEvent()* is called. The *RadarSensor* recovers from a failure with the rate specified in the *QUMStochasticTransition Wait_R_OK*. When this transition is taken, the exit action of the state *RF* sets the variable *rsFailed* to false.

The *Monitor* is initially in the state *AllOK* of its normal behavior state machine, which is shown in Figure 7.19. Whenever the operation *fireRSFailedEvent()* or *fireWSFailedEvent()* is called, the state *RSFailed* or *WSFailed*, respectively is entered. In the states *RSFailed* and *WSFailed* the corresponding variable indicating the availability of the sensor is set to false. Once one of the states is reached, either the transition to the state *ALLOK* is taken as soon as the sensor has recovered or the EmergencyBrake maneuver is initiated if the second sensors also fails. The failure pattern of the *Monitor* is modeled by one state indicating the non availability of the *Monitor* and is entered by the rate specified by the *QUMFailureTransition WAIT_MON_FAIL*. When the failure

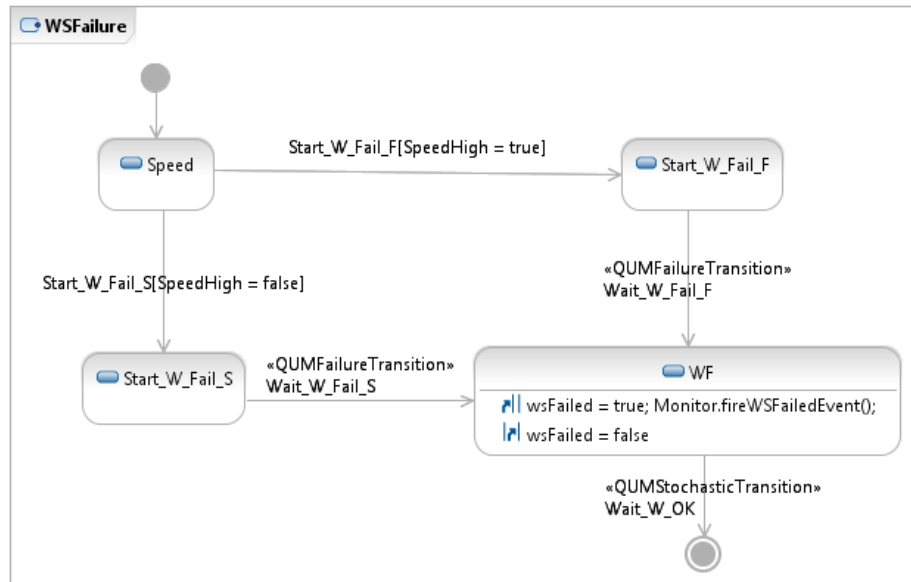


Figure 7.16: State machine representing the failure pattern of the WheelSensor.

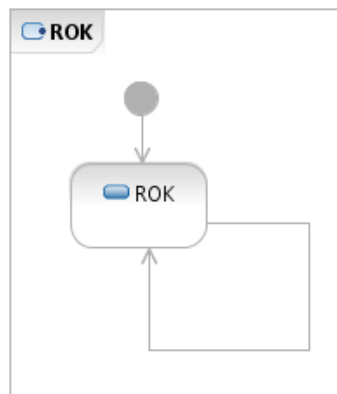


Figure 7.17: State machine representing the normal behavior of the RadarSensor.

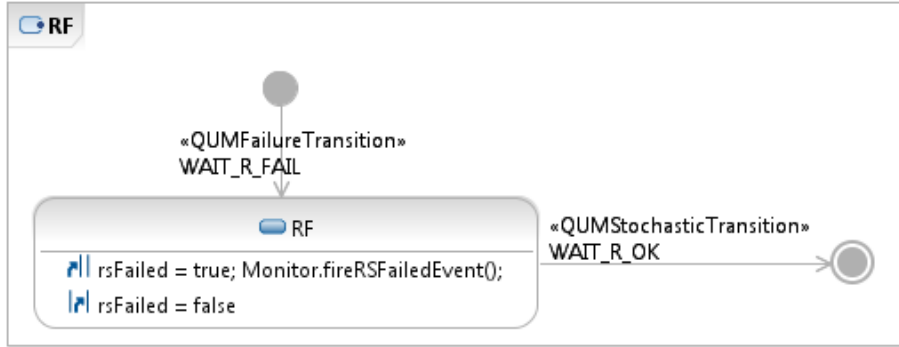


Figure 7.18: State machine representing the failure pattern of the RadarSensor.

state is entered, the variables *rsensor* and *wsensor* are set to true.

The *Observer* component continuously checks whether one of the sensors is in a failure state, indicated by one of the variables *wsFailed* and *rsFailed* being true and whether this was recognized by the *Monitor* indicated by the variables *wsensor* and *rsensor*. Whenever one of the sensors failed and the *Monitor* nevertheless indicates that this sensor is available the state *Wait* is entered. If after a second check the *Monitor* is still indicating that the failed sensor is available the *Unsafe* state is entered. If the *Monitor* recognized in the meantime that the sensor is down, the transition to the state *Safe* is taken. We are interested in the probability of reaching the *Unsafe* state, hence we tag this state with a *QUMStateConfiguration* with the name *unsafe*.

The *Speed QUMComponent* sets the variable *SpeedHigh* to either true or false. This variable is then used in the failure pattern state machine of the *WheelSensor*.

We exported the above described UML 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. The resulting PRISM model consists of 11722 states and 66262 transitions. The QuantUM tool also generated the CSL formula

$$P_{=?}[(\text{true})U^{<=T}(\text{unsafe})]$$

where *unsafe* is replaced by the state formula

$$(\text{Observer_states} = 3)$$

which identifies state *Unsafe* that was tagged with the *QUMStateConfiguration* and *T* represents the mission time. We computed the probability for the mission

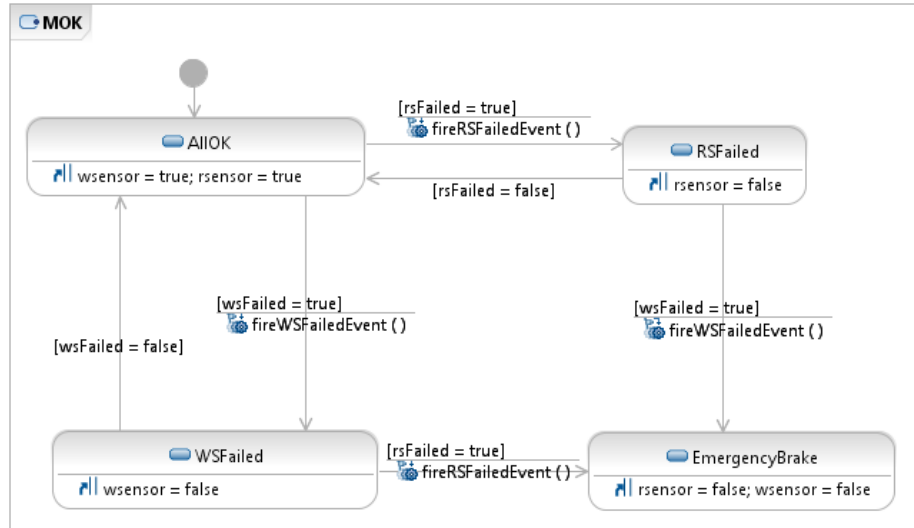


Figure 7.19: State machine representing the normal behavior of the Monitor.

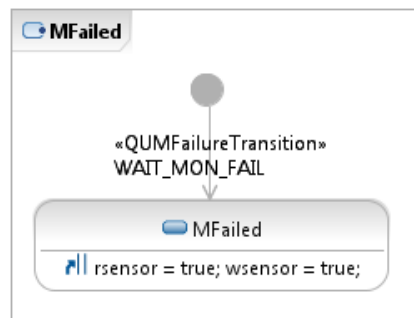


Figure 7.20: State machine representing the failure pattern of the Monitor.

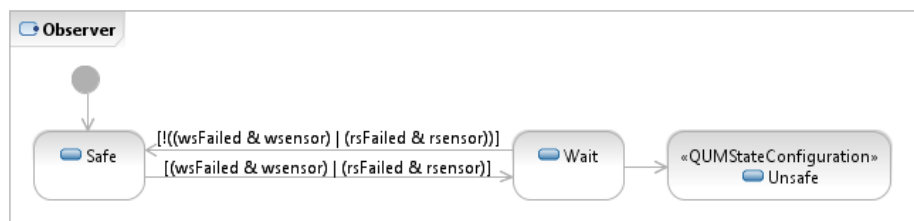


Figure 7.21: State machine representing the normal behavior of the Observer.

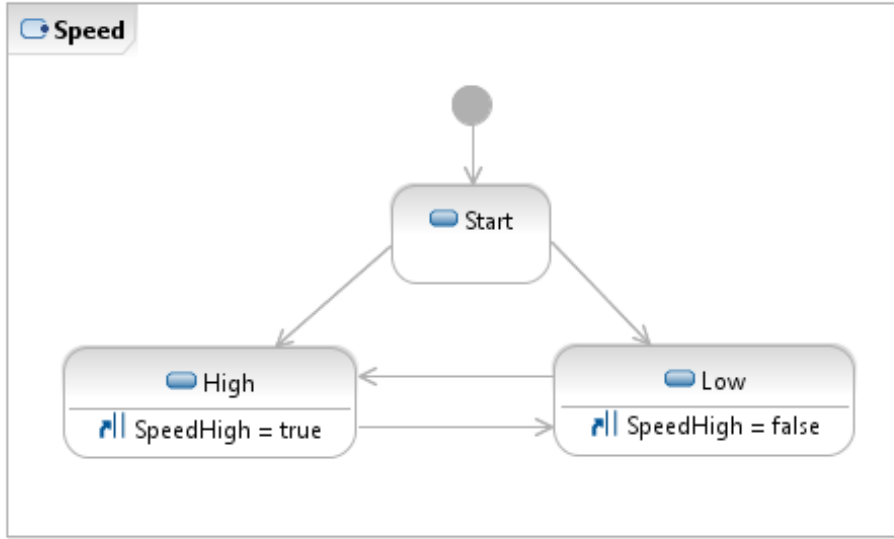


Figure 7.22: State machine representing the normal behavior of the Speed QUM-Component.

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 7.23. The experiments were 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	379.681 (approx. 6.33 min.)	108	0.030	5
100	505.081 (approx. 8.42 min)	108	0.026	5
1000	1074.831 (approx. 17.91 min.)	108	0.043	5

Figure 7.23: Experiment results for $T=10$, $T=100$ and $T=1000$.

Figure 7.23 shows that the computation of the fault tree is finished in under one second, whereas the computation of the counterexample takes several minutes. While the different running times of the counterexample computation algorithm seems 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 caused by background processes on the experiment pc.

Figure 7.24 shows the fault tree generated from the counterexample for the formula $P_{=?}[(\text{true})U^{<=10}(\text{unsafe})]$. While the counterexample consists of 108 paths, the fault tree comprises only 5 paths. In the fault tree it is easy to see

that all paths contain the basic event *WAIT_MON_FAIL* and a number of basic events representing a failure of the wheel sensor, or of the radar sensor, or of both sensors. Again, if our fault tree method would not be used, the same conclusion would require to compare all 108 paths manually.

The UML sequence diagram, which was generated from the counterexample for the formula $P_{=?}[(\text{true})U^{<=10}(\text{unsafe})]$ and imported into the CASE tool IBM Software Architect is shown in Figures 7.25 and 7.26. In the sequence diagram it is easy to see that the event *WAIT_MON_FAIL*, that is the transition from *MOK* to *MFailed*, together with a sequence representing the failure of one or both of the sensors leads to the *unsafe* state. The *par* combined fragments show that the *WAIT_MON_FAIL* event can either happen before, in parallel or directly after the sensor failure sequence.

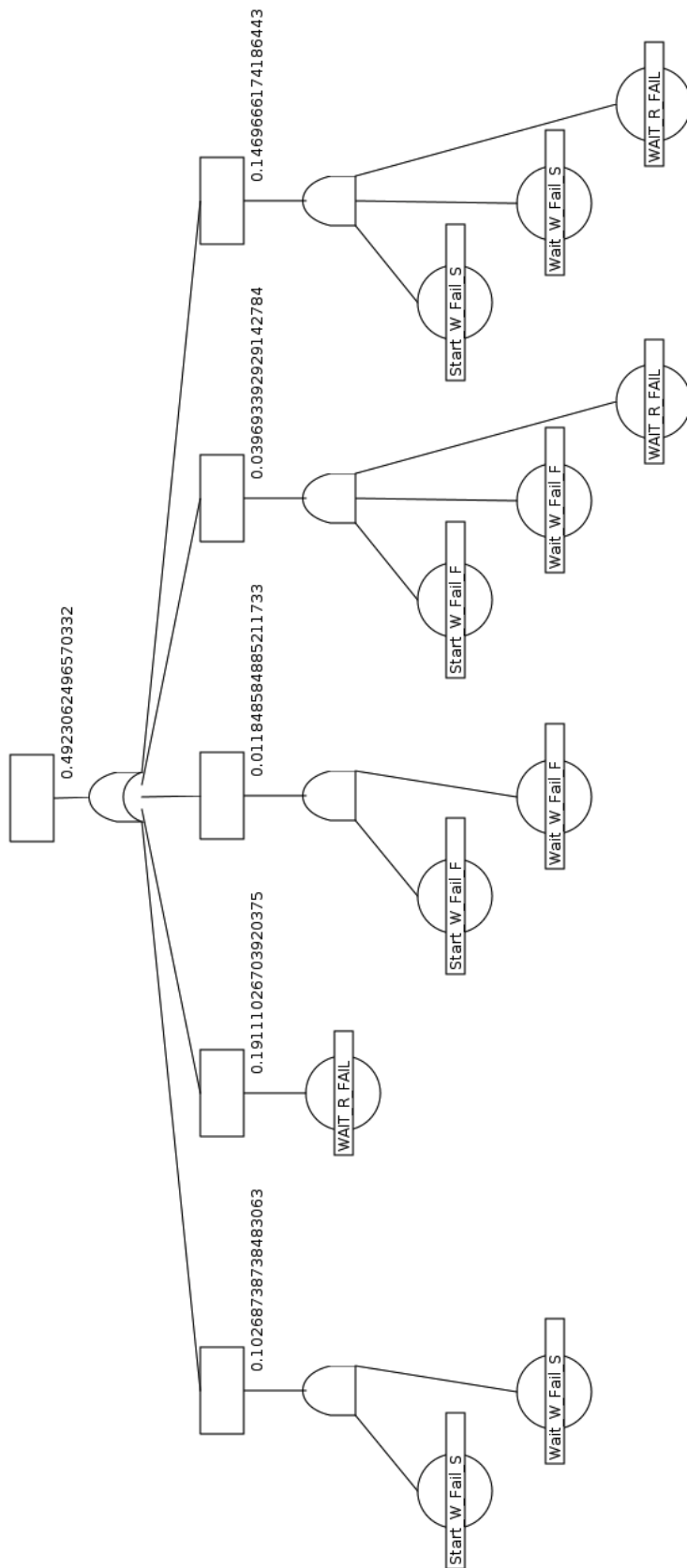


Figure 7.24: Fault tree for the *QUMStateConfiguration unsafe* (T = 10).

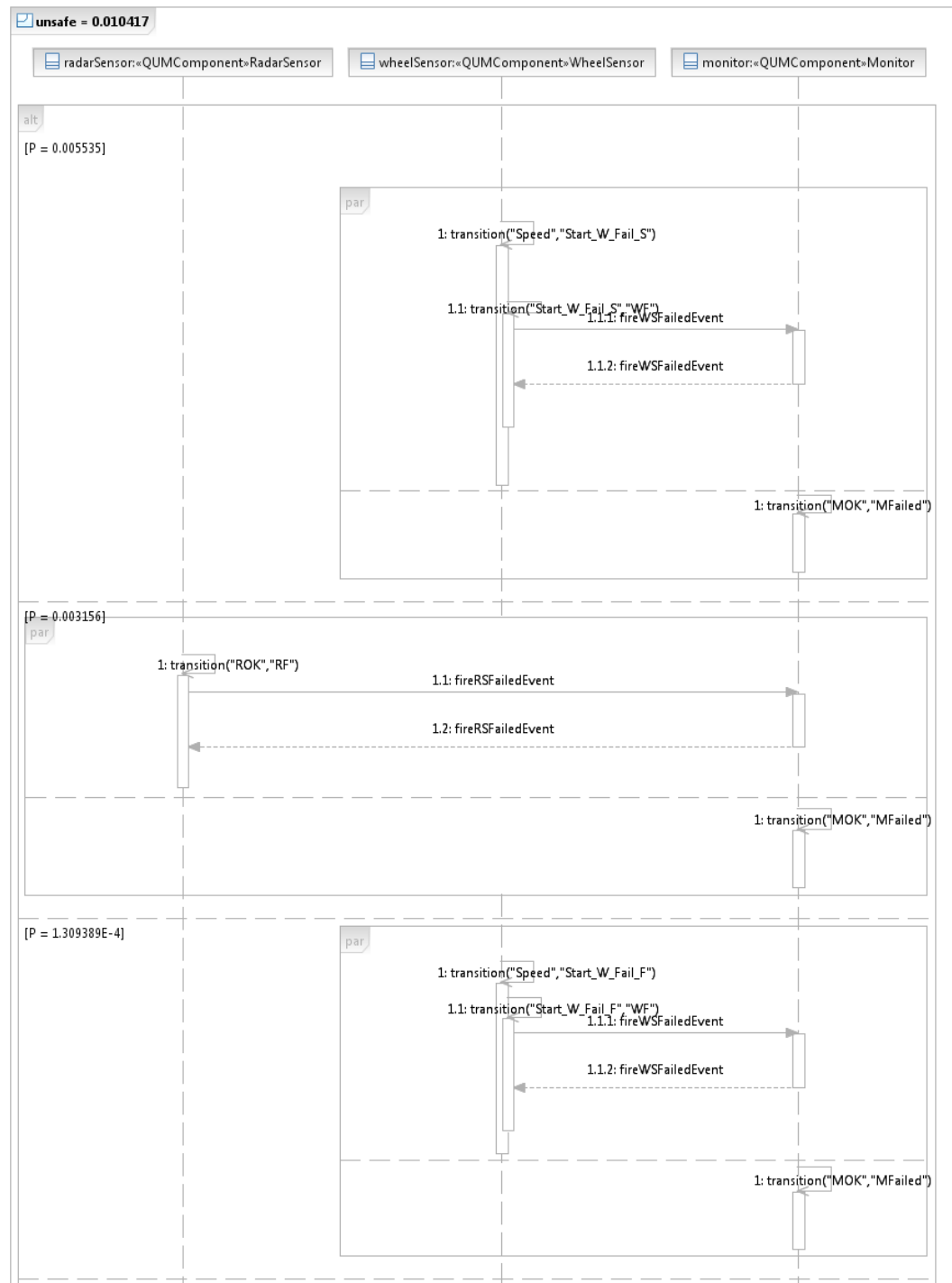


Figure 7.25: UML Sequence diagram for the *QUMStateConfiguration* unsafe ($T = 10$) (part 1 of 2)

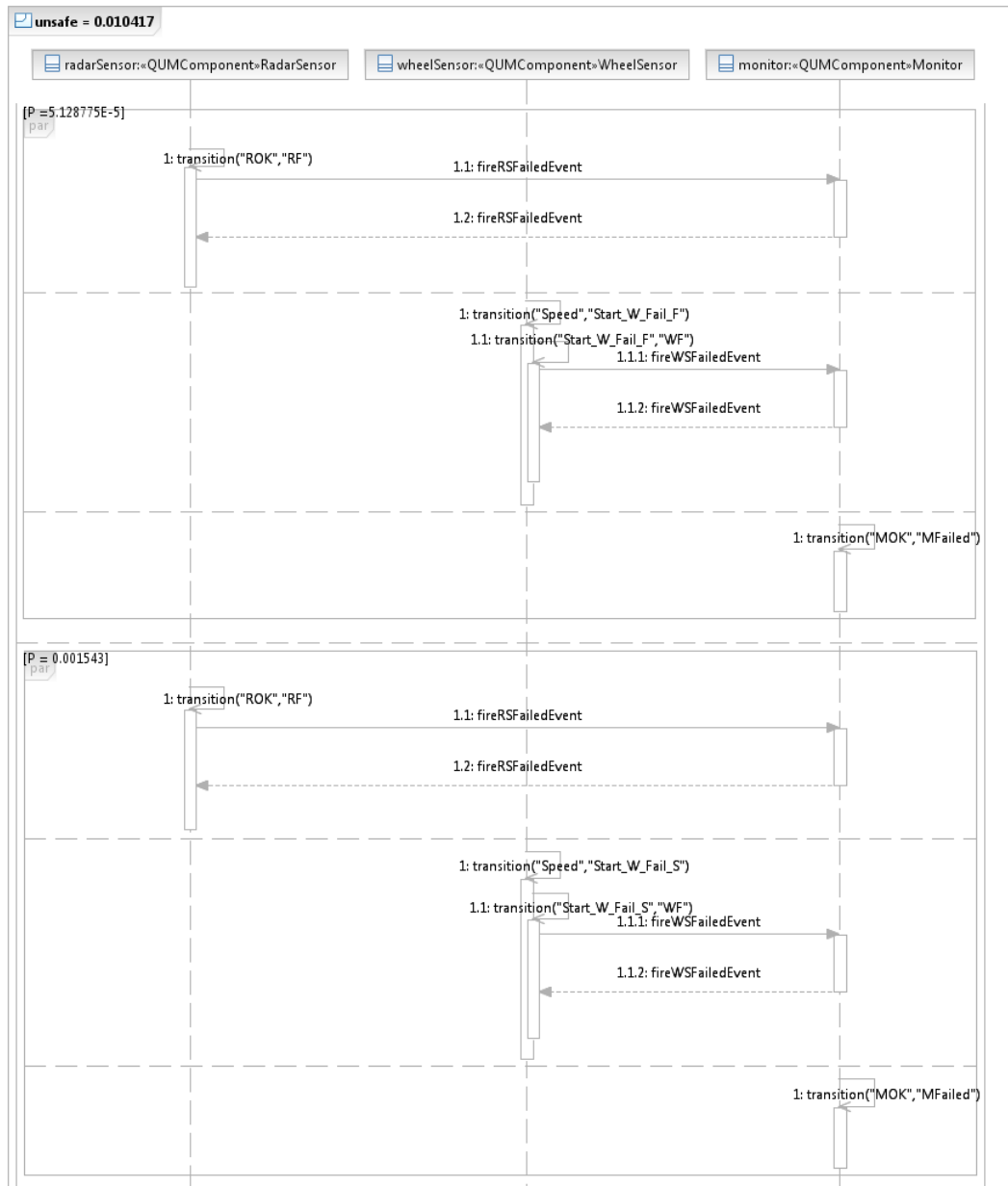


Figure 7.26: UML Sequence diagram for the *QUMStateConfiguration* unsafe ($T = 10$) (part 2 of 2)

Chapter 8

Related Work

8.1 QuantUM Approach

There exists work in the literature, that allows for the quantitative analysis of UML models or the automatic generation of fault trees. We will discuss this work and show how it relates to the QuantUM approach in the following two sections. We are to the best of our knowledge not aware of any approach, that integrates these two approaches together with automatic CSL property construction and a mapping of probabilistic counterexamples into one integrated approach.

8.2 Quantitative Extension of UML

The idea of using UML models to derive models for quantitative safety analysis is not new. In [44] 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 drawback is the introduction of unnecessary redundant information in the UML model, since sometimes the joint use of more than one stereotype is required. In [45] the authors extend the UML Profile for Modeling and Analysis of Real Time Embedded Systems [46] 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 drawback of this approach is that the translation from the annotated UML model into the Deterministic and Stochastic Petri Nets (DSPN) [47] 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 [48] presents a stochastic extension of the Statechart notation, called StoCharts. The StoChart approach suffers from the following drawbacks. 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 [49] the architecture dependability analysis framework Arcades is presented. While Arcade is very expressive and was applied to hardware, software and infrastructure systems, the main drawback 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.

In Figure 8.1 we compare the QuantUM approach, based on the requirements defined in Section 3.1, with other approaches known from literature. We used + if the requirement is completely fulfilled, \pm if it is only partially fulfilled and - if it is not fulfilled.

Approach / Req.:	(1)	(2)	(3)	(4)	(5)	(6)	(7)
QuantUM	+	+	+	+	+	+	+
Majzik et al. [44]	+	\pm	\pm	\pm	\pm	\pm	\pm
Bernardi et al. [45]	\pm	\pm	+	+	-	-	-
Jansen [48]	-	-	\pm	-	-	-	-
Arcade [49]	+	+	+	+	+	-	-

Figure 8.1: Comparison of approaches known from literature

8.3 Automatic Fault Tree Generation

There exist a number of approaches for automatic fault tree generation in the literature. In recent work [50, 51], algorithmic strategies for fault tree analysis of reactive systems that are implemented in the FSAP/NuSMV-SA platform have been presented. [52] discusses an approach that generates fault trees with a symbolic model checker. The automatic synthesis of fault trees based on annotated UML models is presented in [53]. In [54] a method to automatically generate fault trees from Little-JIL process definitions is proposed. Further automatic fault tree generation methods are presented in [55, 56, 57]. The main drawback of the above mentioned approaches compared to our approach is the lack of an automatic probabilistic assessment of the fault tree nodes, which is what our method provides.

In [43], a method is described that uses minimal cut sets for the computation of the probabilities to reach a safety critical state. The authors present

an approach that aims at computing the probabilities of different combinations of component failures to reach a safety-critical state. According to the method that they propose it is possible to identify those system components whose failures contribute most to the probability of reaching a safety-critical state. The system is modeled using extended Statecharts. Information about the failure transition labels, the failure transitions, and the relevant time bounds have to be provided either as an annotation to the Statecharts or as a separate specification in a language different than extended Statecharts. The fault configurations of the system that lead to a safety-critical state are represented as cut sets. Our work extends and improves the approach of [43] in the following ways: (1) We use a single system specification and modeling language, namely the UML, for describing the behavior of the system, the failure model and the relevant time bounds in an integrated fashion. The approach of [43] uses a large number of different tools and modeling notations. (2) By using the PRISM model checker and the counterexample computation capabilities that we integrated into PRISM, it is possible to use the full expressiveness of CSL for the specification of probabilistic measures of interest, whereas the approach of [43] only allows for the analysis of timed reachability properties. (3) Whereas in the approach of [43] only minimal cut sets are generated, we generate complete fault trees. We believe that our visual rendering of fault trees is essential for the system designer to identify the critical system paths. (4) Finally, by allowing PAND-gates we support the full set of fault tree operators defined in [6], which is not the case for the approach of [43].

Chapter 9

Conclusion

9.1 Conclusion

We have presented an UML profile that allows for the annotation of UML models with quantitative information, together with a tool that automatically translates the model into the PRISM language and performs the analysis with PRISM. In addition to the translation of the model, the tool also allows for the automatic generation of CSL properties.

Furthermore, we have developed a method that automatically generates a fault tree from a probabilistic counterexample. The resulting fault trees were significantly smaller, then the probabilistic counterexamples and hence easier to understand, but still contain all important information. The fault tree generation method is not limited to models that were generated with our QuantUM tool, it can also be applied to counterexamples generated from a manual constructed PRISM model.

Additionally, we presented a mapping of probabilistic counterexamples to UML sequence diagrams and thus make the counterexamples interpretable inside the UML modeling tool. We believe that this is particularly useful for system and software architects which usually can interpret an UML sequence diagram much better as fault trees, whereas safety engineers might prefer the visualization of the counterexamples as fault trees.

The QuantUM tool that we have developed, allows for an easy-to-use and fully automated probabilistic analysis of UML models, where the analysis level is complete hidden from the user.

We have demonstrated the usefulness of our approach on two case studies known from literature. The case studies showed 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.

9.2 Future Work

In future work we plan to further extend the expressiveness of the QuantUM profile, for example by adding more repair strategies. In addition, we will add constraints in the Object Constrain Language (OCL) [58] to the profile, which will enable the CASE tool to perform consistency checks of the annotations.

Furthermore, we are planning to further automated the translation from UML to PRISM by the integration of methods to automatically determine the ranges of variables. We also plan to integrated specification pattern systems like ProProST [59] in order to further facilitate automatic stochastic property specification. In addition, we plan to further extend our method for fault tree generation, to support the generation of dynamic fault-trees [38].

The SysML [60] language is very similar to the Unified Modeling Language. Therefore we plan to investigate whether our approach can also be adapted to the SysML language.

For the time being, the factor limiting scalability of the approach, is the algorithm used for counterexample generation. Our recent work [33] indicates that a better performance of the algorithm can be achieved by using heuristics. For this reason, we plan to investigate how we can automatically generate heuristics for the counterexample search.

At the moment the approach can be applied to all system and software architecture models, that do not make use of complex programming code in the entry-, during- or exit-action of the states. To further extend our approach we plan to work on methods for the probabilistic analysis of object-oriented source code.

Chapter 10

Appendix

10.1 CD

Bibliography

- [1] 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: QEST '09: Proceedings of the Sixth International Conference on Quantitative Evaluation of Systems, Los Alamitos, CA, USA, IEEE Computer Society (2009) 299–308
- [2] Grunske, L., Colvin, R., Winter, K.: Probabilistic model-checking support for fmea. In: QEST '07: Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems, Washington, DC, USA, IEEE Computer Society (2007) 119–128
- [3] Object Management Group: Unified Modeling Language. Specification v2.3. <http://www.uml.org> (2010)
- [4] Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: 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, Springer (2006) 441–444
- [5] Aljazzar, H., Leue, S.: Debugging of dependability models using interactive visualization of counterexamples. In: QEST '08: Proceedings of the Fifth International Conference on the Quantitative Evaluation of Systems, IEEE Computer Society Press (2008)
- [6] U.S. Nuclear Regulatory Commission: Fault Tree Handbook. (1981) NUREG-0492.
- [7] Object Management Group: XML Metadata Interchange (XMI), v2.1.1. <http://www.omg.org/technology/documents/formal/xmi.htm> (2007)
- [8] IEC 61508: IEC(International Electrotechnical Commission) Functional safety of electrical/electronic/programmable electronic safety-related systems (2004)
- [9] ISO 26262: International Organization for Standardization, Road Vehicles Functional Safety (Committee Draft) (2008)
- [10] Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic Model Checking. In: Modeling and verification of parallel processes. Summer school. (2001) 189–204

- [11] Kulkarni, V.: Modeling and analysis of stochastic systems. Chapman & Hall/CRC (1995)
- [12] A. Aziz, K. Sanwal, V. Singhal, R. K. Brayton: Verifying continuous-time Markov chains. In: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification. Volume 1102., New Brunswick, NJ, USA, Springer Verlag LNCS (1996) 269–276
- [13] Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering* **29** (2003)
- [14] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* **8** (1986) 244–263
- [15] Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. *ACM Trans. Comput. Logic* **1** (2000) 162–170
- [16] Avižienis, A., Laprie, J., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing* (2004) 11–33
- [17] Birolini, A.: Reliability engineering: theory and practice. Springer Verlag (2007)
- [18] Latella, D., Majzik, I., Massink, M., et al.: Towards a formal operational semantics of UML statechart diagrams. In: IFIP TC6/WG6. Volume 1., Citeseer (1999) 331–347
- [19] Alur, R., Henzinger, T.A.: Reactive modules. *Formal Methods in System Design* **15** (1999) 7–48
- [20] Siemens AG: (Siemens Norm SN29500, Ausfallraten Bauelemente)
- [21] Telcordia Technologies: Sr-332 Reliability Prediction Procedure for Electronic Equipment (2001)
- [22] Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: The prism language - semantics. (Available from URL <http://www.prismmodelchecker.org/doc/semantics.pdf>)
- [23] Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The Ins and Outs of The Probabilistic Model Checker MRMC. In: QEST '09: Proceedings of the Sixth International Conference on Quantitative Evaluation of Systems, IEEE Computer Society (2009) 167–176
- [24] Aljazzar, H., Leue, S.: Extended directed search for probabilistic timed reachability. In: FORMATS '06: In Proceedings of the 4th International Conference on Formal Modelling and Analysis of Timed Systems. Lecture Notes in Computer Science, Springer (2006) 33–51
- [25] Aljazzar, H.: Directed Diagnostics of System Dependability Models. PhD thesis, Universität Konstanz, Universitätsstr. 10, 78457 Konstanz (2009)

- [26] Aljazzar, H., Leue, S.: Generation of counterexamples for model checking of markov decision processes. In: QEST '09: Proceedings of the Sixth International Conference on Quantitative Evaluation of Systems, Los Alamitos, CA, USA, IEEE Computer Society (2009) 197–206
- [27] Aljazzar, H., Leue, S.: Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Transactions on Software Engineering* (2009)
- [28] Andrés, M.E., D'Argenio, P.R., van Rossum, P.: Significant Diagnostic Counterexamples in Probabilistic Model Checking. In: Haifa Verification Conference. Volume 5394 of Lecture Notes in Computer Science., Springer (2008) 129–148
- [29] Fecher, H., Huth, M., Piterman, N., Wagner, D.: Hintikka games for PCTL on labeled Markov chains. In: QEST '08: Proceedings of the Fifth International Conference on the Quantitative Evaluation of Systems, Washington, DC, USA, IEEE Computer Society (2008) 169–178
- [30] Han, T., Katoen, J.P., Damman, B.: Counterexample generation in probabilistic model checking. *IEEE Transactions on Software Engineering* **35** (2009) 241–257
- [31] Damman, B., Han, T., Katoen, J.P.: Regular Expressions for PCTL Counterexamples. In: QEST '08: Proceedings of the Fifth International Conference on the Quantitative Evaluation of Systems, IEEE Computer Society (2008) 179–188
- [32] Schmalz, M., Varacca, D., Völzer, H.: Counterexamples in Probabilistic LTL Model Checking for Markov Chains. In: Proceedings of the 20th International Conference on Concurrency Theory. Volume 5710 of Lecture Notes in Computer Science., Springer (2009) 587 – 602
- [33] Aljazzar, H., Kuntz, M., Leitner-Fischer, F., Leue, S.: Directed and heuristic counterexample generation for probabilistic model checking: a comparative evaluation. In: QUOVADIS '10: Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems, New York, NY, USA, ACM (2010) 25–32
- [34] Aljazzar, H., Leue, S.: K^* : A directed on-the-fly algorithm for finding the k shortest paths. Technical Report soft-08-03, University of Konstanz, Gemany (2008) submitted for publication.
- [35] Pearl, J.: Heuristics – Intelligent Search Strategies for Computer Problem Solving. Addison–Wesley (1986)
- [36] Dijkstra, E.W.: A note on two problems in connexion with graphs. *Numerische Mathematik* **1** (1959) 269–271
- [37] Crouzen, P.: Compositional Analysis of Dynamic Fault Trees using Input/Output Interactive Markov Chains. Master's thesis, University of Twente, Enschede (2007)
- [38] Dugan, J., Bavuso, S., Boyd, M.: Dynamic Fault Tree Models for Fault Tolerant Computer Systems. *IEEE Transactions on Reliability* **41** (1992) 363–377

- [39] Bruns, G., Anderson, S.: Validating safety models with fault trees. In: SafeComp. Volume 93. (1993) 21–30
- [40] Kozen, D.: Results on the propositional μ -calculus. Theoretical Computer Science **27** (1983) 333–354
- [41] Stirling, C.: Temporal logics for CCS. Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency (1989) 660–672
- [42] ITU-TS recommendation Z.120: Message sequence chart (msc) (1996)
- [43] Böde, E., Peikenkamp, T., Rakow, J., Wischmeyer, S.: Model based importance analysis for minimal cut sets. In: ATVA '08: Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis, Berlin, Heidelberg, Springer-Verlag (2008) 303–317
- [44] Majzik, I., Pataricza, A., Bondavalli, A.: Stochastic dependability analysis of system architecture based on UML models. Architecting dependable systems (2003) 219
- [45] Bernardi, S., Merseguer, J., Petriu, D.: A dependability profile within MARTE. Software and Systems Modeling (2009) 1–24
- [46] Object Management Group: UML Profile for Modeling and Analysis of Real Time Embedded Systems. <http://www.omgmarTE.org/> (2008)
- [47] Marsan, M., Chiola, G.: On Petri nets with deterministic and exponentially distributed firing times. Advances in Petri Nets 1987 (1987) 132–145
- [48] Jansen, D.N.: Extensions of statecharts : with probability, time, and stochastic timing. PhD thesis, University of Twente (2003)
- [49] Boudali, H., Crouzen, P., Haverkort, B., Kuntz, M., Stoelinga, M.: Architectural Dependability Modelling with Arcade. In: Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. (2008) 512–521
- [50] Bozzano, M., Cimatti, A., Tapparo, F.: Symbolic fault tree analysis for reactive systems. In: ATVA '07: In Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis, Springer (2007) 162–176
- [51] Bozzano, M., Villaflorita, A.: Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform. Computer Safety, Reliability, and Security (2003) 49–62
- [52] Liggesmeyer, P., Rothfelder, M.: Improving system reliability with automatic fault tree generation. In: Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on. (1998) 90–99
- [53] Pai, G.J., Dugan, J.B.: Automatic synthesis of dynamic fault trees from uml system models. In: ISSRE '02: In Proceedings of the 13th International Symposium on Software Reliability Engineering. Volume 0., Los Alamitos, CA, USA, IEEE Computer Society (2002) 243

- [54] Chen, B., Avrunin, G., Clarke, L., Osterweil, L.: Automatic fault tree derivation from little-jil process definitions. *Software Process Change* (2006) 150–158
- [55] Ratan, V., Partridge, K., Reese, J., Levenson, N.: Safety analysis tools for requirements specifications. Available from URL http://www.safeware-eng.com/system_and_software_safety_publications/SafAnTooReq.pdf (1996)
- [56] Cha, S., Leveson, N., Shimeall, T.: Safety verification in Murphy using fault tree analysis. In: *ICSE '88: Proceedings of the 10th International Conference on Software Engineering*, IEEE Computer Society Press (1988) 377–386
- [57] McKelvin Jr, M., Eirea, G., Pinello, C., Kanajan, S., Sangiovanni-Vincentelli, A.: A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems. In: *Proceedings of the 5th ACM international conference on Embedded software*, ACM (2005) 246
- [58] Object Management Group: Object Constraint Language (OCL), v2.2. <http://www.omg.org/spec/OCL/2.2/> (2010)
- [59] Grunske, L.: Specification patterns for probabilistic quality properties. In: *ICSE '08: Proceedings of the 30th international conference on Software engineering*, New York, NY, USA, ACM (2008) 31–40
- [60] Object Management Group: SysML. Specification v1.2. <http://www.sysml.org> (2010)