Technical Report soft-12-01, Chair for Software Engineering,
University of Konstanz, Copyright by the Authors 2012

Analysis of an Airport Surveillance Radar using the QuantUM approach

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

[1]Universität Konstanz,
[2]EADS Deutschland GmbH

March 5, 2012

Abstract

We report on the modeling and formal analysis of reliability requirements in the context of an *Airport Surveillance Radar* system using SysML and probabilistic model checking. The system is modeled using the QuantUM modeling tool which uses the PRISM model checker as an analysis back-end. We illustrate how a complex system architecture can be modeled, what challenges have to be addressed when performing the automated property verification, and to what extent the presented automated analysis method can support the system engineering. We expect that the use of these methods will also have a strong impact on the certification process in future.

1 Introduction

The development of safety-critical software and systems is subject to special reliability and dependability requirements. The analysis and verification of such systems is, therefore, of high interest to systems and software engineers. Only few tools and notations supporting the automatic verification of reliability and

dependability requirements [20] for such systems at the architecture level are available. An example of such a notation is the graphical modeling language UML [3]. The QuantUM tool presented in [22] supports the specification and analysis of system dependability requirements for UML models. The QuantUM approach has recently been extended to SysML [21]. Using QuantUM it is possible to begin the verification and analysis process of an UML or SysML model directly at the modeling stage of a system. It is a benefit of this approach that users do not have to understand the entire formal verification process in detail. Also, QuantUM can cooperate with many industrial practice UML tools. This means that users can most likely use a modeling tool and notation that they are familiar with and do not have to familiarize themselves with formal notations or languages needed to operate formal verification tools. The Analysis that QuantUM offers is automated, which promises a much less time consuming verification process than manual analysis.

It is the objective of this report to illustrate the application of the QuantUM modeling and the related automated analysis technology to an industrial case study. We report on work that was recently performed in a joint project between the Chair for Software Engineering of the University of Konstanz and Cassidian[1]. The system that we consider as a case study is an *Airport Surveillance Radar* (ASR) system which consists of a pure radar signal based *primary radar* component, and a transponder communication based *secondary radar* component. The detected radar echoes are tracked by the system and graphically displayed to air space controllers, which are the typical users of this system. The system is safety-critical in nature and hence very stringent requirements with respect to system reliability, availability and dependability apply. The system is also subject to certification requirements imposed by the German Government. The project therefore aimed to assess the overall scalability of a modeling and verification approach based on SysML and a formal analysis based on probabilistic model checking. At the same time it was a prime objective to analyze how this approach can be used to compile safety cases such as they needed in system certification. The project used the design and architecture of an existent system of which first prototypes have been developed, in this sense the work described here was not performed concurrently with the actual system development.

[1] `http://www.cassidian.com/`

Structure of the report. In Chapter 2 and 3 the basis of the QuantUM approach is shown, while in the following chapters the actual modeling process and verification will be discussed (4). In the last chapter a perspective for the future is given (6). In the Appendix there is an extra chapter for radar-related abbreviations and explanations for those readers who are not familiar with the radar-terminology 7.

1.1 Contributions (Executive Summary)

1. We provide a QuantUM-Model of the ASR system.
2. The possible failure behavior of the ASR system is captured and incorporated in the QuantUM model.
3. A probabilistic failure analysis of both the one channel and the two channel ASR architecture for the case of a failure of the SSR component is performed.
4. The comparison of the results for the one channel and the two channel models illustrates that the proposed analysis can be used to perform design time architecture variant analysis with respect to reliability and dependability properties.
5. An approximative, incomplete Fault Tree Analysis (FTA) has been performed and we have estimated the remaining runtime in order to obtain a complete fault tree.
6. We have analyzed the use of functional model checking to increase the scalability of the FTA and plan to pursue this as a goal for future research.

1.2 Related Work

In [13], Debbabi et al. present a method for automated model checking of SysML activity diagrams. Since in SysML, an extension of the UML, it is possible to tag activity diagrams directly with probabilities, it is not necessary to introduce stereotypes into the model. A disadvantage of this approach is that only single diagrams are checked and dependency of other components or diagrams in the system are disregarded. This dependency problem is addressed in [11] where

Bernardi et al. have extended the UML profile for modelling and analyzing real time embedded systems (MARTE [2]) with an additional profile for dependability analysis and modeling. A drawback of this method is that it relies on the MARTE profile, which is provided by the OMG and not well supported in most commercial UML tools. Another disadvantage of this approach lies in the high number of stereotypes that have to be applied to the model to actually prepare it for the property checking process. The conversion of the UML model to the Petri Nets [26] that is required in this work has to be done manually, which is an error prone and time consuming process. The application of this appraoch to complex systems using MARTE is hence laden with substantial risks.

In [17] Jansen shows a stochastic extension of Statecharts. With this extension only the behavior of the system is analyzed and the structure is omitted completely. The work presented in [25] introduces a new UML profile for the annotation of software dependability properties. After transforming these models into an intermediate format, they are converted into Timed Petri Nets [26]. Like in the approach discussed above, a drawback is that the conversion has to be done manually. The second disadvantage of this approach is that it only focuses on the structural aspects of the system. Similarly to the MARTE approach, there is high number of stereotypes and often redundant information attached to the model. In [12], the architecture dependability analysis framework Arcade is presented. A limitation of this approach is that Arcade cannot be easily integrated with industrial design processes, which are often based on UML and SysML

2 Foundations

In this section we give a short introduction to the UML [3], the basis of the *QuantUM* [22] approach. Another technique described in this section is probabilistic model checking [19], used for the analysis of the ASR model.

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

2.1 UML / SysML

The Unified Modeling Language is a specification language which is a de-facto standard in modeling structure, behavior and architecture of software systems. It was standardized by the *Object Management Group*, a no-profit consortium of companies from the computer industry, founded by IBM, Apple, SUN and other well known companies.

There are many tools supporting the UML for systems and software development. For instance, IBM Rational Rhapsody[3], Sparxsystems Enterprise Architect[4], or ArgoUML[5] are tools supporting software design using the UML. For a detailed description of the language see the UML standard specification given in [3]. Since UML was mainly developed for software engineering, it was necessary to extend UML to systems engineering in order to make it usable in this domain. System engineering is concerned with software and hardware design. The *International Council of Systems Engineering* (INCOSE) added specific modeling to the UML that reflect needs from the system engineering domain. The resulting language specification is called *SysML* [1]. In this report we will use SysML to model the ASR system.

The main categories of SysML are structure and behavior, the same as in UML. In the structural category there are diagrams like block definition diagrams and parametric diagrams, while in the behavioral category Statecharts and activity diagrams are found. Overall, SysML comprises 11 diagrams types, some of which can also be found in the UML specification.

2.2 Probabilistic Model Checking

Probabilistic model checking (PMC) is a technique for automated analysis of safety-critical systems. The term *PMC* was introduced by Kwiatkowska et al. [19] in 2001, while the methodology was invented by Aziz et al. in [8].

Model checking, a property or specification S is automatically checked against a model M, which means that it is determined whether S holds of M. In most cases the model is defined as a transition system (TS) [9]. The states represent

[3] `http://www.ibm.com/software/awdtools/rhapsody/`
[4] `http://www.sparxsystems.de/uml/`
[5] `http://argouml.tigris.org/`

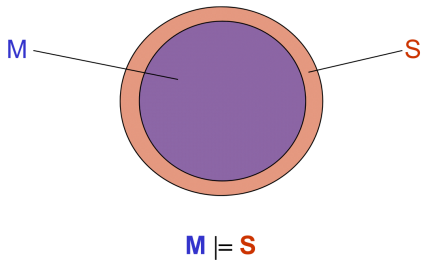


Figure 1: An abstract representation of the model M and the specification S [24]

the observable values of the program variables and the program control at a given point of time while the transitions represent system functionality carried out while transiting from one state to another. Transitions can be guarded so that state changes in the system can only happen if the guard condition is satisfied.

In probabilistic model checking the transitions of the TS are labeled with quantitative information related to the probabilistic behavior of the system. This information can be given in terms of probabilities or rates at which the system can proceed. The QuantUM tool that we use for our analysis uses the probabilistic model checker *PRISM* [15] as a verification back-end. Like many other model checking tools, *PRISM* requires models to be represented in its own input language. *PRISM* supports Discrete-Time-Markov chains (DTMC), Continuous-Time Markov chains (CTMC) and Markov Decision Processes (MDP) as models [15]. CTMCs are used in the QuantUM approach to check probabilities on models. *PRISM* is a symbolic model checker, which means that it uses data structures based on binary decision diagrams (BDD) and multi-terminal binary decision diagrams to represent the model. The use of BDDs helps to compress the whole model in terms of memory efficiency since they allow to perform transformations directly on the compressed data without the need to decompress them first [4].

On the specification side, *PRISM* supports a number of logics, such as PCTL, CSL, LTL and PCTL* [15]. In the context of the *QuantUM* approach we use the CSL (continuous stochastic logic) [8, 10] to specify properties. CSL is an extension of CTL, which adds two new probabilistic operators. The first operator refers to the behavior of the system in a specific set of states, in partic-

ular how long a system is in one state. The other operator constrains execution paths representing the transient behavior of the system. The resulting specifications can, for instance, express the property that nothing bad happens along some such path.

3 The QuantUM-Approach

In the software developing industry, UML has become a widely used modeling technique used to document system analysis and system design. But there are only few tools directly supporting the development of safety critical software systems, even though there exist statutory requirements for most of those systems. The recently proposed *QuantUM* tool [22] offers the ability to augment UML and SysML models with annotations so that both the normal and the stochastic failure behavior can be represented in such a model. *QuantUM* possesses a code generation engine that translates UML and SysML models directly to PRISM code which can then be directly checked by the PRISM model checker. The QuantUM process consists of the following steps:

1. Modeling of the model in an arbitrary UML modeling tool. In the project that we report on the IBM Rational Rhapsody [6] tool was used.
2. Add the QuantUM stereotypes to the models.
3. Export the whole project into an XMI file, a standard representation of UML diagrams in XML format [2].
4. Import the generated XMI file into the QuantUM tool. The PRISM code and properties are automatically generated in the course of this process.
5. Choose the properties which shall be checked. The options are the generation of a fault tree using DiPro [7], or to simulate the state space exploration in DiPro.

In the modeling step it is necessary to incorporate the faulty behavior of the model directly into Statecharts so that the desired functionality of the system as well as its failure behavior are represented in the same UML model. In addition

[6] `http://www.ibm.com/software/awdtools/rhapsody/`

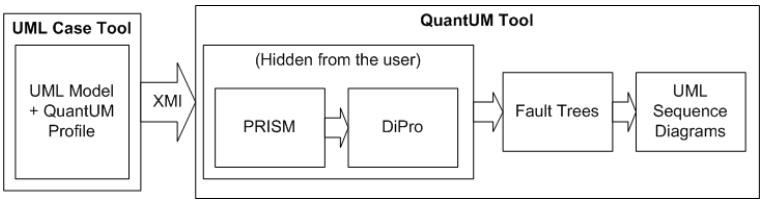


Figure 2: The QuantUM-tool chain [22].

to the modeling of the false behavior the developer needs to tag the states and transitions in the UML model as well as selected other UML elements with the provided *QuantUM* stereotypes. This has to be done so that the QuantUM-tool knows which Statecharts to convert into PRISM-code. This indicates, that it is not necessary to tag all elements of the UML model with stereotypes. Less important parts (e.g., displays) which are not critical for safety can be modeled in the same UML diagram, but are not checked in the model checking process. The tags described here are also used when specifying the ASR-model. For a detailed description of how the conversion is done, especially how the different UML diagrams are combined and translated into PRISM, we refer to [22].

The extension of QuantUM to SysML was accomplished by extending the QuantUM profile with new SysML stereotypes. The underlying structure of the tool was not changed, but only extended to handle the additional stereotypes [21]. In the next section the modeling of the ASR system including the use of the newly added stereotypes is discussed.

4 Modeling of the ASR

The Airport Surveillance Radar (ASR) is a system for monitoring the airspace in the vicinity of an airport. It is used by air traffic controllers who guide aircrafts according to their flight plan. The ASR consists of three main components. The first component is the Primary Surveillance Radar (PSR) which uses radar signals that are reflected from the body of an object, which in the normal case is an aircraft that is to be tracked, in order to locate the object. The second component is the Secondary Surveillance Radar (SSR) which communicates with the transponder of the aircraft in order to obtain, amongst others, location information. The internal structure of the radar consists of two channels. Each

channel processes the data provided by the PSR and the SSR. The model we are presenting in this report represent a one- and two-channel variant of the system architecture of the ASR. The model also describes how the components of the ASR process the data. The time horizon that the model considers consists of three cycles of the radar antenna.

The modeling of the ASR is described at three stages.

- At the first stage the main functionality is described and some SysML-Diagrams are presented to show the interaction of the different components in the *ASR*.
- At the second stage a very simple *brute-force*-approach is used to extend the simple model to a parallel one in order to obtain a higher availability of the whole system. As shall be argued later, this modeling strategy is not memory efficient.
- At the third and last stage an optimization of the parallel model is introduced. We use a newly introduced stereotype derived from the `QUMSpare` stereotype of QuantUM. This stereotype can represent the situation where a component has one or more spares which take over the functionality of the component when a failure occurs.

4.1 Stage 1 and Functionality

At the first stage we model the ASR system as consisting of only one channel. A channel consists of 5 components:

1. Receiver (not shown on the diagram, see below)
2. Signal processor
3. Parameter extractor
4. Sensor Tracker
5. Validation Unit

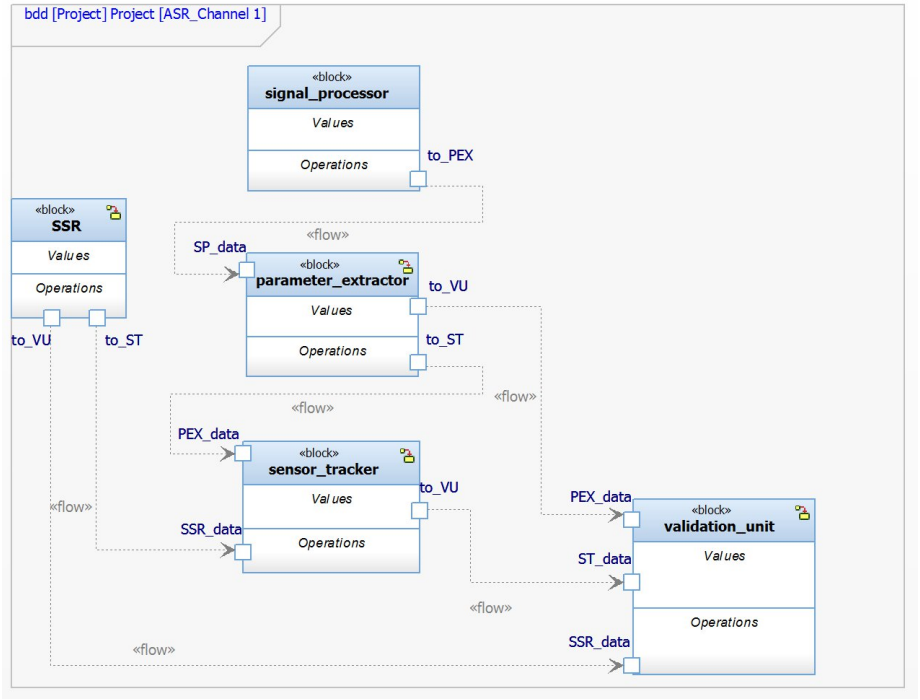


Figure 3: An overview block-definition-diagram of one channel and the SSR

There is one additional component, the secondary surveillance radar (SSR), which is used for the digital communication with the transponders of the aircrafts. The receiver collects the reflected radar signals from the environment. It forwards them to the signal processor, which is an analog-digital converter for the signals. The digitalized signal is analyzed inside the parameter extractor in order to determine where plots have to be initialized. This is accomplished by discriminating the reflections of aircrafts from reflections of the surrounding terrain (houses, trees, etc.) by means of what is referred to as a clutter map. After identifying the plots the data is sent to the sensor tracker (ST). The sensor tracker combines the plots from the primary radar and the secondary radar and tries to discern possible tracks of aircrafts. If the ST finds a track it is added to the list of tracks that the system maintains. A track is observed until the aircraft leaves the range of the radar signal, or a failure happens. In the last step the validation unit checks whether the outcome of the sensor tracker is plausible or not. After performing these steps the calculated data is presented

on the graphical display of the ASR system.

In Figure 3 the overview block-definition-diagram [1] of one ASR channel is shown. The signal processor is omitted from the analysis described in this report since we are focusing on the digital part of the system. A Statechart containing the normal behavior of the component is defined for all other components in the block-definition-diagram. This is indicated by the yellow boxes in the upper right corner in each component in Figure 3. In addition to the components shown in the window there is an environment component called `FlightObject`. It simulates the behavior of the environment, i.e., the aircrafts. This modeling step is necessary since model checkers require closed models that represent both the behavior of the system to be analyzed as well as the behavior of the environment in which it operates. In other words, the external functionality has to be defined within the model. We now explain the functionality of each component of the *ASR*-System in detail.

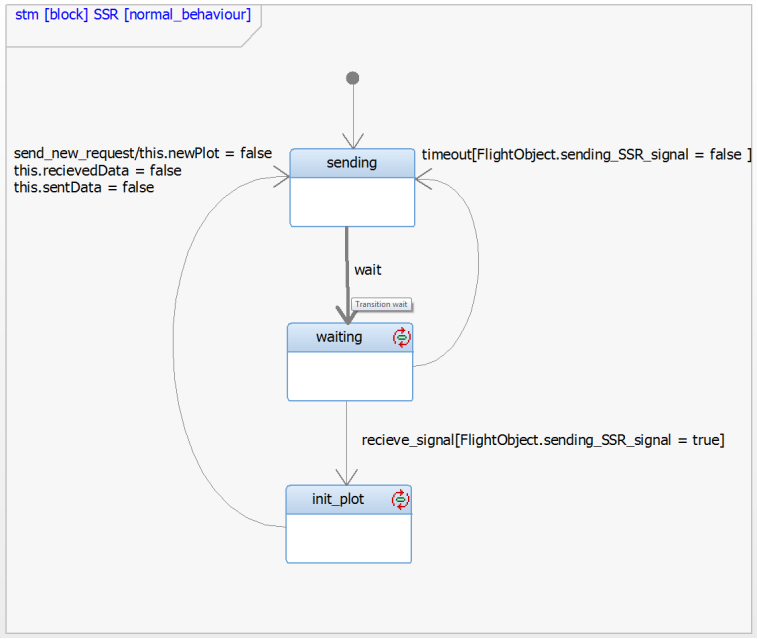


Figure 4: The Statechart of the SSR

Secondary Surveillance Radar. The first component is the *SSR*. The state chart is given in Figure 4. This component contains 3 states: `sending`, `waiting`

and `init_plot`. Initially the Statechart is in the `sending` state where the SSR sends a request to the environment. After sending the message the SSR transitions into the `waiting` state and waits for an answer from the transponder of an aircraft. If after a certain timeout period no answer is sent, the messages will be resent. Otherwise a message is returned from the aircraft and the SSR initializes a plot (`init_plot`). The new plot is then sent to the *sensor tracker* and the *validation unit*. The next cycle of the SSR starts with the sending of a new message to the environment.

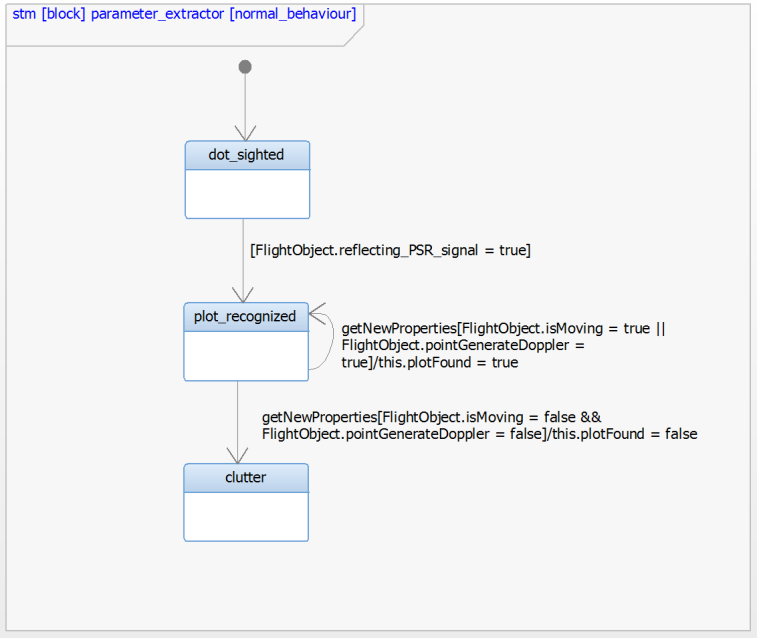


Figure 5: The Statechart of the parameter extractor

Parameter Extractor. This component processes the already digitalized signal reflections from the PSR. It contains three states: `dot_sighted`, `plot_recognized` and `clutter`. In the beginning, when a dot is sighted the parameter extractor is in the `dot_sighted`-state. When there is a flight object which can reflect the radar signal then it transitions into the `plot_recognized`-state. The parameter extractor remains in this state as long as the Flight object is moving or generates a DOPPLER-effect (Helicopter with rotating rotors). If this case is not fulfilled the plot is categorized as clutter and is saved in the clutter-map, so

that it is not displayed in the next cycle.

Sensor Tracker. The Statechart of the sensor tracker can be found in the appendix (Figure 12). In this component the digital and the analog plots are combined into tracks representing the flight paths of the tracked aircrafts. The sensor tracker consists of 4 states: `first_plot`, `plotDetected`, `track` and `coastedTrack`. The states `first_plot`, `plotDetected` and `coastedTrack` contain additional Statecharts where the track labeling is handled. Since all of these sub-Statecharts have more or less the same functionality only one is given in this report (Figure 13 in the appendix). This Figure shows the sub-Statechart of the `track`-state.

In the beginning, the sensor tracker is in the `first_plot`-state where an initial labeling is done. There are three possible labels for a track, namely *combined*, *SSR* and *PSR only*. When the system is in the *SSR only* state then there is only a SSR-signal attached to the plot. The *PSR only* is defined accordingly and the `combined` label says that both signals are found. After labeling the first plot with one of the possible labels the sensor tracker is transitioning into the `plotDetected`-state where two possible successor decisions can be made. If there is only a PSR-plot found the sensor tracker goes back to the first plot and increments a counter. If this has happened more than three times, a new track tagged *PSR only* is initialized. The other possibility is that a *combined* or a *SSR only* plot is found. In this case a new track is initialized immediately, since it is very unlikely that there is a SSR signal without an aircraft.

When a new track is found, new plots which are matching this track are added after each cycle of the system. If there is a cycle where no new plot is found the track changes its labeling in a bad manner. For example a labeling can change from *combined* to *PSR only* or *SSR only*. “Bad” in this case means that information about the track is lost in comparison to the last cycle. Then the track will be coasted. Coasting a track means that the possible position of the aircraft is extrapolated from recent data. The system then tries to recover the track to its original labeling through three more cycles. If this is not possible the track is closed, which means that the aircraft is no longer appearing on the screen. This is one of the worst scenarios which can occur in this system: The aircraft is still up in the air but does not appear on the screen. When there is a new plot found within three cycles, then the track is recovered to its state

before the coasting.

Another possibility is that the track changes its labeling in a positive way, for instance when the track was labeled with *PSR only* and in the next cycle the label is changed to *combined*. Of course this is not a bad behavior and thus the track is continued under the new label.

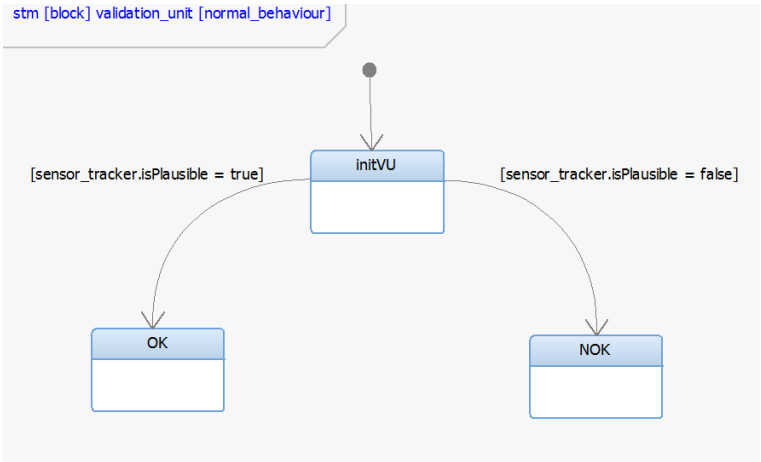


Figure 6: The Statechart of the validation unit

Validation Unit. The last component in the channel is the validation unit which checks whether the output of the sensor tracker is plausible or not. Its behavior Statechart is therefore very simple. It contains a basic decision between the two states `OK` or `NOK`. The attribute which holds the value of the plausibility is set in the failure pattern of the sensor tracker. The meaning of the failure pattern is explained in the following paragraph.

Failure Patterns. Failure patterns are used to determine whether a component has a failure or not. Since we later want to analyze the correct as well as the failure behavior of the system, both behaviors need to be modeled. In the model the failure patterns are attached to the components in the form of additional Statecharts. An example failure pattern Statechart for the SSR is shown in Figure 7.

The SSR contains four possible failure modes:

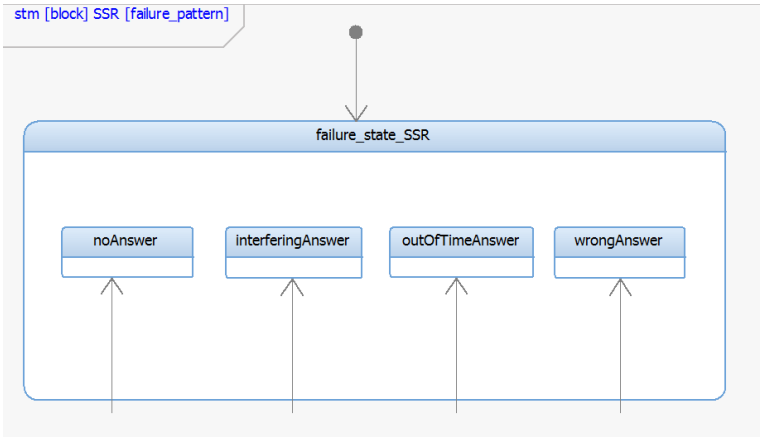


Figure 7: The Statechart for the failure pattern of the SSR

1. **noAnswer.** This failure can occur when there is an aircraft but it is not responding to any requests of the SSR.

2. **interferingAnswer.** In this case two or more aircrafts are simultaneously sending a response to the SSR so that the messages are interleaved. This is not handled in the implementation of the SSR protocol, because it is very unlikely that such an interference is lasting for more than one cycle.

3. **outOfTimeAnswer.** The message which is sent to the aircraft was not answered within one cycle.

4. **wrongAnswer.** The message was returned with the wrong parameters.

There are failure patterns similar to those of the SSR defined for every component in the ASR. These are shown in the appendix in Figures 14, 15 and 16.

4.1.1 Preparing the Model for *QuantUM*

In this section we illustrate the preparation of the model for the verification with the *QuantUM* tool.

The first step is to identify the components in the model which are safety-critical. We omit the signal processor since it is merely an analog-digital con-

verter and hence considered irrelevant for the safety proof. All other components are crucial, since we are interested in the digital part of the system. All crucial components are tagged with the `QSyMComponent` tag so that QuantUM can convert them into individual modules for the model checking process. For this tag the user needs to set at least one parameter, namely the normal behavior of this module. Figure 8 depicts part of the QuanUM specification of the parameter extractor. Notice that the specification includes a normal behavior and a failure pattern. The two parameters set in the specification refer to the Statecharts defined as illustrated in the previous section. The other tags of the `QSyMComponent` are optional and filled in automatically during the parsing step in *QuantUM*.

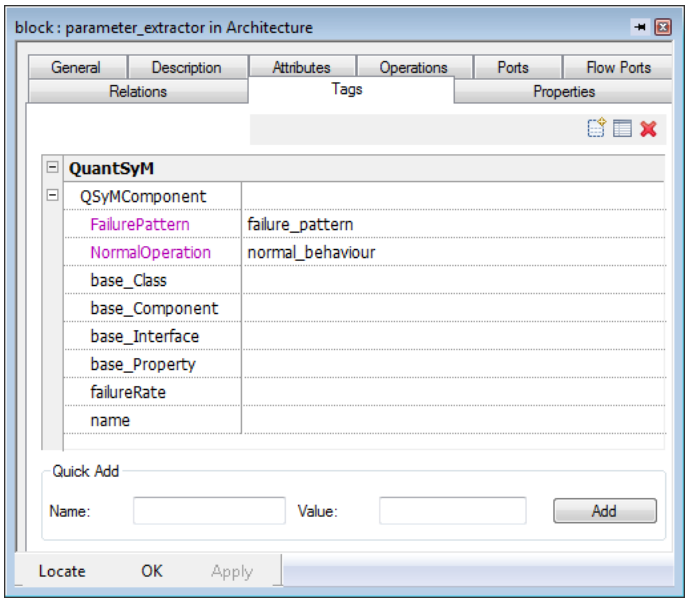


Figure 8: The stereotype `QSyMComponent` for individual modules

These are the basic steps necessary to obtain an output model from *QuantUM* that can be analyzed with the PRISM model checker. There is another stereotype called *QSyMFailureTransition* available to add rates to the transitions in the failure pattern so that it is possible to calculate probabilities on the paths in the model. There is only one parameter in this stereotype, which is the rate of this transition being fired. For this survey we use 10^{-6} as a typical value.

Another stereotype being used is the `QSyMAttributeRange`. This is added to the attributes using integers as domain. Since the size of the domain of integer variables in programs finite, but very large, we restrict the domain of the integer variables in the SysML model to a small range. This helps to contain the state space size of the *PRISM* model.

There is one other stereotype used in this model which is explained in Section 4.3. The other stereotypes defined in the *QuantUM* profile are not used in this model. It becomes obvious that instrumenting the SysML design model so that it meets the *QuantUM* requirements for verification requires only little modifications of the model. The attachment of the stereotypes to an existing model takes only a fraction of the time that is needed for the remainder of the system design and modeling process.

4.1.2 Computing Probabilities for Safety Requirements

After exporting the one channel model to an XMI-file it can be opened in *QuantUM*. In *QuantUM* the user has two options: computing the probability of a given hazard or error, or generating the fault tree for a given hazard or error. The experiments below were performed on a PC with an Intel i7 3.6 GHz processor and 24 GB of RAM. We are interested in the probability of an SSR failure within one hour. Another failure specification that we have analyzed is "the probability to obtain a coasted track within one hour".

The first possibility is to compute the probability of a given property directly. With this computation the user directly obtains the probability of an error or hazard. The one channel model described in this Chapter consists of approximately 1.2 million states and 7.5 million transitions. The direct computation of the SSR-failure took 3.53 seconds with a memory consumption of 31 MB and results in a probability of $1 \cdot 10^{-6}$. For the failure of getting a coasted track we have computed a probability of $8.939 \cdot 10^{-10}$ within 5.69 seconds and a memory consumption of 31MB. The direct computation only yields the probability for an error to happen, but it does not provide any insights in how or why the hazard or error happened. In order to gain insight in how the hazards happens, *QuantUM* can compute the fault tree [27] for the hazard we are interested in.

4.1.3 Automatic Fault Tree Generation

The type of properties that we analyze are time bounded or time unbounded probabilistic reachability properties. In the commonly used topological classification scheme of safety and liveness properties these qualify as safety properties. In order to automatically generate fault trees for those safety properties we use the approach proposed in [18]. The fault tree is computed from a full set of counterexamples, that is all bad execution traces, and all corresponding good execution traces. Consequently, the fault tree generation algorithm presumes that the full state space of the model has been explored. In case of the 1-channel ASR model the full state-space exploration takes a considerable amount of time. We aborted the state space exploration after 12 hours and generated an incomplete fault tree. Note that the resulting fault tree is incomplete with respect to the probabilities and the displayed branches. The probability of an incomplete fault tree is lower than the actual probability and it might be the case that not all combinations of basic events that lead to the error are contained in the fault tree. The incomplete fault tree for the event "coasted track within one hour" is depicted in Figure 9. The figure shows that a coasted track is detected whenever the status of a track is "PSR_only" or "SSR_only". As discussed above, since the fault tree is incomplete the probabilities are underestimations of the actual failure probabilities and there might be other combination of basic events that can lead to a coasted track.

In order to obtain an estimate how much time this exploration would take we have performed a series of tests. In these tests we are using the fact that the probability collected on the paths increases when new paths leading into an error state are found. The collected probability then converges to the maximum probability of the failure to happen. The verification results from the first direct computation of a property can hence be taken as an upper bound for the Fault Tree generation. We have started the fault tree generation and measured the collected probability at defined time intervals so that we could estimate the time-consumption for the complete state space exploration process. In Table 1 the probabilities at predefined time intervals are shown for the property that the SSR fails within one hour. Since we are using estimated rates in the transitions the values displayed are only approximate values that do not represent the actual values of the real ASR system. In the calculation of the probabilistic counterexamples using *DiPro* the calculation is implemented in an on-the-fly

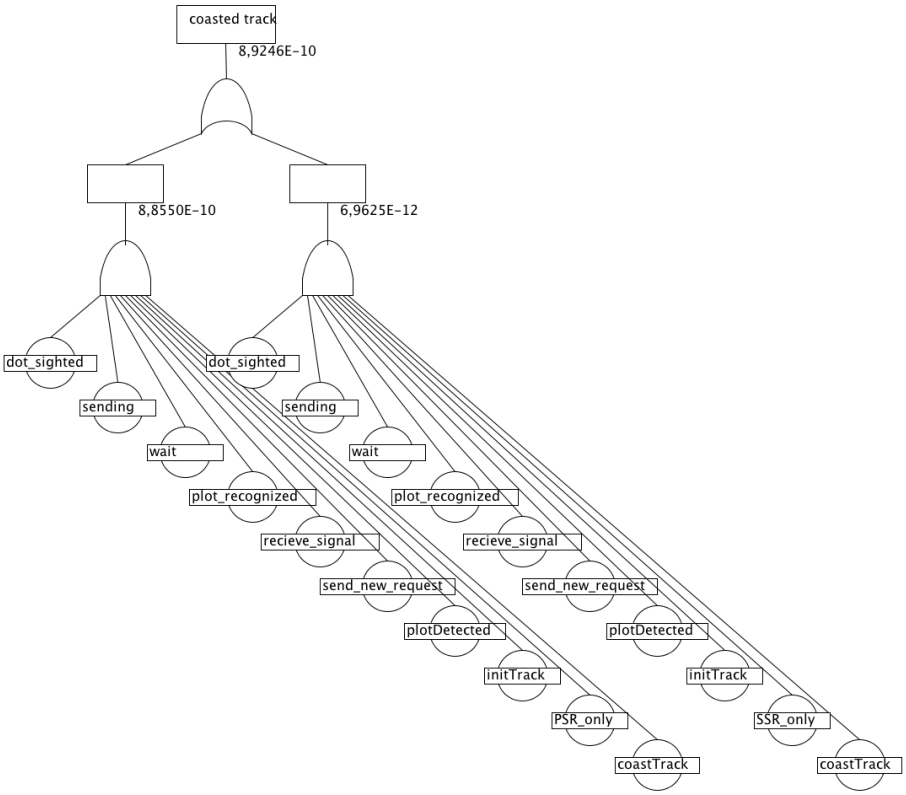
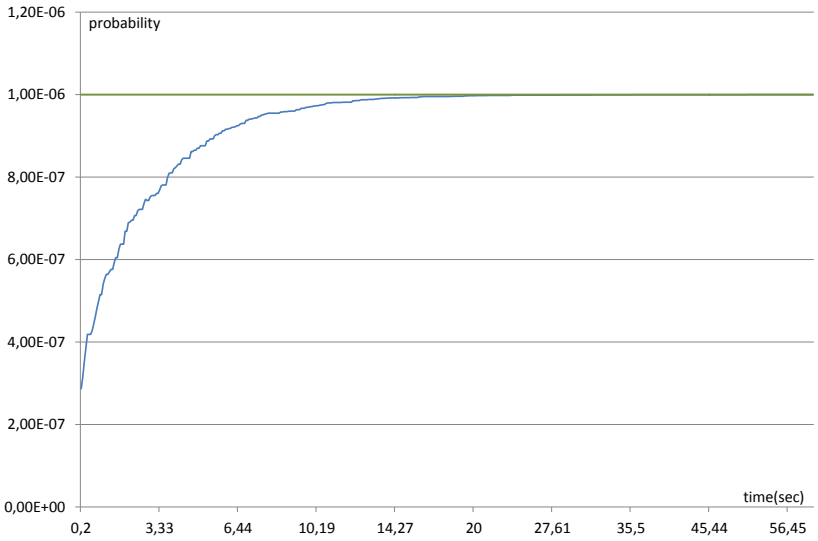


Figure 9: Fault tree for event "coasted track within one hour"

manner. This is important because otherwise the whole state space has to be stored in the memory. With the on-the-fly implementation the generation of the counterexample can be done on a normal office computer. The values in Table 1 show that even after the short time period of 1 minute the collected probability is already slightly below the target probability.

4.2 Stage 2 - A Naive Approach

We now consider the two channel variant of the ASR system. It consists of two concurrently executing copies of the one channel system. The challenge is to add the second channel to the existing one channel model without experiencing an exponential blow up of the state space. This blow up is called state space



prob. (2 chan.)$\cdot 10^{-7}$	4.9832	8.6989	9.6930	9.9981	10
prob. (1 chan.)$\cdot 10^{-7}$	5.1506	8.8746	9.7239	9.9985	10
time (sec)	1	5	20	60	MAX

Table 1: Collected probabilities for the probability of an "SSR-failure within one hour"

explosion and it is a well known phenomenon in concurrent systems analysis.

The first naive approach that we chose was to concurrently instantiate a second copy the first channel and to adjust the constraints of each component so that they propagate their results to both channels. With this approach the number of components rose up to 9. With this extension the size of the model rose up to appr. 3.4 billion states and appr. 17 billion transitions. This very large number of states cannot be handled by state-of-the-art model checkers on current hardware and thus has to be reduced.

4.3 Stage 3 - Using `QSyMSpare`

After trying the naive approach, we changed an existing *QuantUM* stereotype to address the state explosion issue. The `QSyMSpare` stereotype from the original QuantUM specification can be added to a component to indicate that the component has one or more spare-parts that can take over processing when it experiences a failure. In the ASR system the spare parts of a component are identical copies of this component that run in parallel. In the QuantUM model we represent the switching from a component to its spare copy by incrementing a counter. The switching can only occur if the counter is less than the amount of spares available which is defined in the `QSyMSpare` stereotype. In particular, the `QSyMSpare` stereotype was changed so that it now is an extension to the *QSyMComponent* stereotype. Figure 10 shows two additional attributes of this stereotype, the `activationRate` of the spare components and the `numberOfSpares` indicating how many spares there are.

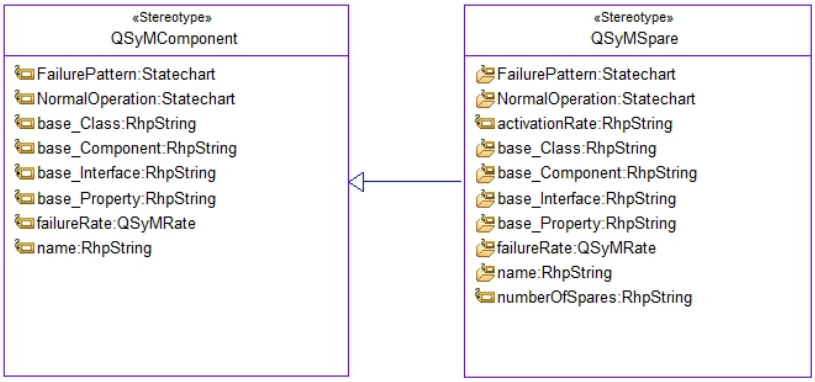


Figure 10: The generalization of the QSyMComponent

This new stereotype was used to replace the `QSymComponent` stereotype in the parameter extractor, sensor tracker and validation unit of the component. The `SSR` has no spare modules and thus did not receive a `QSyMSpare` tag in our ASR model. The number of spares of each component was set to 1 so that after a failure of one component its spare can be activated. Since there were no changes made in the implementation of this stereotype, we refer to [22] for a detailed explanation. The activation rate of the spares in our model was set to 1.

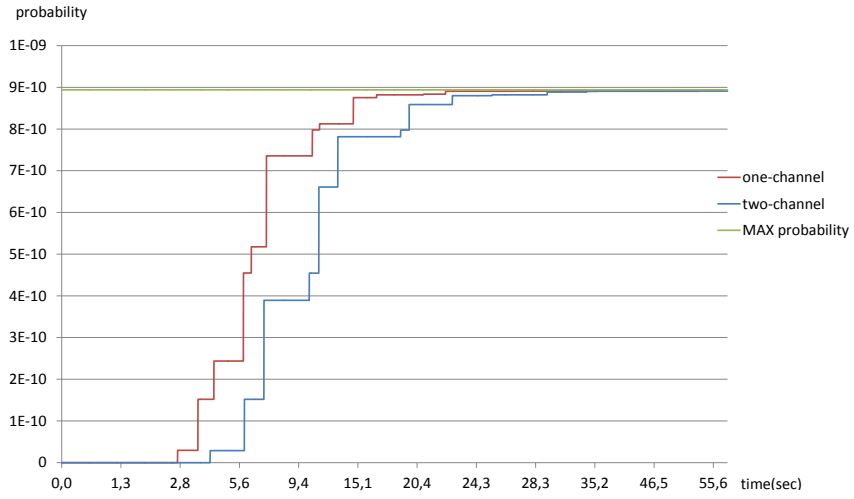


Figure 11: The collected probabilities for the probability to "obtain a coasted track within one hour".

The number of states in the ASR model with the `QSyMSpare` tags attached was appr. 46.4 million and it has appr. 326 million transitions. This number of states and transitions is analyzable with state-of-the-art hardware. The computation of the SSR failure in the two-channel model took 307.3 seconds with a memory consumption of 1.5 GB. The computed probability was $1 \cdot 10^{-6}$. For the coasted track property we have computed a probability of $8.939 \cdot 10^{-10}$. The computation was finished after 509.5 seconds and a memory consumption of 1.5GB.

Interpretation of the results.

– The verification results show that the probability of each property for the one-channel and the two channel architecture model are the same. For the SSR-failure it is obvious that the probabilities are the same since the SSR occurs only once in the ASR system. As a consequence the asymptotic approximation curves for performing an FTA for an SSR-failure in the one channel and the two channel model in Table 1 coincide.

- The property for obtaining a coasted track, shown in Figure 11, results in different curves for the one channel and the two channel models. Every time a new failure path is found during the state space exploration, the probability of this path is added to the previously found path probabilities. This leads to the step-like jumps in the curve.
- The FTA for the one channel model is faster than for the two channel model because less paths have to be checked.
- From Figure 11 we can derive the observation that although the computation for the one channel model reaches a higher probability faster than the computation for the two channel model the asymptotic probability for both models is identical. This phenomenon can be explained as follows:
  - Both models converge to the same probability of $8.939 \cdot 10^{-10}$. This can be traced to the fact that we analyze the probability of getting a coasted track without any prior component failing. The case where a component in the channel fails is automatically detected and the spare component is activated. In other words, it is important for the ASR system that the probability of the system to fail is not dependent on the number of channels working.
  - In the asymptotic approximation curve for the coasted track property in Figure 11 we can see this availability property since the target probability does not depend on whether there are one or two channels working.

We have shown that it is possible to apply the quantitative analysis using QuantUM to both the one channel and the two channel models. A potential threat to the scalability of the analysis lies in the combinatorial state space explosion due to concurrent system components, as can be seen in the case of the two channel model. However, as we illustrate by introducing the `QSyMSpare` stereotype, a judicious choice of the way in which these components are represented in the model can address this threat. A further scalability issue is the excessive runtime needs of the fault tree analysis. We will further address this issue in Section 5.

The experimental results also reveal to what extent the one channel and the two channel models behave identically or differently with respect to certain

properties. This shows that the QuantUM based analysis that we propose can be used to perform architecture variant analysis. This observation is particularly interesting because valid observations can be derived in spite of the fact that no real failure probability values are used.

5 Scalability

We succeeded in computing probabilities for the safety requirements within a few minutes for both the one channel and two channel model. However, the application of the automatic fault tree generation was hampered by the need for a complete state space exploration. Apart from the fact that the full state space has to be explored, the fault tree generation is also slowed down because the probability for each bad trace has to be computed.

If we are only interested in the cause of the error, but not in the probabilities, we can use functional model checking techniques, such as implemented in the explicit state model checker SPIN [16], to perform the fault tree generation. First experiments, not presented in this report, show that we are able to perform a functional model checking of the two-channel model within a few minutes. We are planing to integrate the causality checking algorithms with the on-the-fly algorithms used for functional explicit state model checking. Subsequently we will use the results from the causality checking done with functional model checker to speed up the probabilistic model checking process. The key idea of this approach is that the failure paths will be determined using functional causality checking, and afterwards the probability for the different failure paths will be computed.

A second approach to use functional model checking in order to speed up the probabilistic model checking is to use the functional counterexample to derive heuristics for the probabilistic counterexample computation. Experiments reported in [6] indicate that heuristics are an effective mean to speed up the probabilistic counterexample computation.

6 Conclusions and Future Work

In this report we have presented a case study illustrating the use of the *QuantUM* approach in an industrial system setting. The functionality of an ASR system was modeled using a SysML model. We illustrated how the *QuantUM* profile is used to annotate the model. Three different modeling stages were presented, a one-channel model to explain the process in the system and a two-channel system which represents the actual system functionality.

Furthermore, we showed that it is possible to analyze the ASR model with state-of-the-art analysis tools running on current standard hardware. However, we discussed that it is necessary to use strategies when synthesizing the analysis model which contain the state space explosion, in particular when analyzing concurrent systems. We also realized that a complete fault tree analysis based on the functional system model is not yet possible. Addressing this problem is the subject of future research.

The QuantUM tool currently only supports the analysis of Continuous Time Markov Chains [15]. This special type of Markov Chain does not support models that encompass concurrent behavior leading to non-deterministic state transitions. We therefore work on incorporating the analysis of Markov Decision Processes [14] into the analysis offered by QuantUM. Furthermore, we plan to implement the scalability optimizations proposed in Sec. 5 in QuantUM.

References

[1] Systems modelling language, specification 1.2, Jun. 2010. URL `http://www.sysml.org/specs/`.

[2] Xml metadata interchange, specification 2.4.1, 2010. URL `http://www.uml.org/`.

[3] Unified modelling language, specification 2.4.1, August 2011. URL `http://www.omg.org/spec/XMI/`.

[4] S. B. Akers. Binary decision diagrams. IEEE Trans. Comput., 27:509–516, June 1978. ISSN 0018-9340. URL http://dx.doi.org/10.1109/TC.1978.1675141.

[5] L. Alawneh, M. Debbabi, Y. Jarraya, A. Soeanu, and F. Hassayne. A unified approach for verification and validation of systems and software engineering models. In Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, pages 409–418. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2546-6. URL http://dl.acm.org/citation.cfm?id=1126179.1126225.

[6] H. Aljazzar, M. Kuntz, F. Leitner-Fischer, and S. Leue. 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, pages 25–32. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-972-5. URL http://doi.acm.org/10.1145/1808877.1808883.

[7] H. Aljazzar, F. Leitner-Fischer, S. Leue, and D. Simeonov. Dipro - a tool for probabilistic counterexample generation. In Model Checking Software, volume 6823 of Lecture Notes in Computer Science, pages 183–187. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-22305-1. URL http://dx.doi.org/10.1007/978-3-642-22306-8_13. 10.1007/978-3-642-22306-8_13.

[8] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time markov chains. In Computer Aided Verification, volume 1102 of Lecture Notes in Computer Science, pages 269–276. Springer Berlin / Heidelberg, 1996. ISBN 978-3-540-61474-6. URL http://dx.doi.org/10.1007/3-540-61474-5_75. 10.1007/3-540-61474-5_75.

[9] C. Baier and J.-P. Katoen. Principles of Model Checking. The MIT Press, 2008.

[10] C. Baier, J.-P. Katoen, and H. Hermanns. Approximative symbolic model checking of continuous-time markov chains. In CONCUR'99 Concurrency Theory, volume 1664 of Lecture Notes in Computer Science, pages 781–781. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-66425-3. URL http://dx.doi.org/10.1007/3-540-48320-9_12. 10.1007/3-540-48320-9_12.

[11] S. Bernardi, J. Merseguer, and D. Petriu. A dependability profile within marte. Software and Systems Modeling, 10:313–336, 2011. ISSN 1619-1366. URL http://dx.doi.org/10.1007/s10270-009-0128-1. 10.1007/s10270-009-0128-1.

[12] H. Boudali, P. Crouzen, B. R. H. M. Haverkort, G. W. M. Kuntz, and M. I. A. Stoelinga. Architectural dependability evaluation with arcade. In Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Anchorage, USA, pages 512–521. IEEE Computer Society, Los Alamitos, 2008.

[13] M. Debbabi, F. Hassaïne, Y. Jarraya, A. Soeanu, and L. Alawneh. Verification and Validation in Systems Engineering Assessing UML/SysML Design Model. Springer Berlin / Heidelberg, Nov. 18, 2010. ISBN 978-3-6421-5227-6. 153-166 pp.

[14] A. Feinberg, Eugene A.; Shwartz. Handbook of Markov Decision Processes. Springer US, 2002. ISBN ISBN 978-0-7923-7459-6.

[15] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In Tools and Algorithms for the Construction and Analysis of Systems, volume 3920 of Lecture Notes in Computer Science, pages 441–444. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-33056-1. URL http://dx.doi.org/10.1007/11691372_29. 10.1007/11691372_29.

[16] G. J. Holzmann. The SPIN Model Checker: Primer and Reference Manual. Addision–Wesley, 2003. ISBN 0-321-22862-6.

[17] D. N. Jansen. Extensions of statecharts : with probability, time, and stochastic timing. PhD thesis, University of Twente, Enschede, October 2003. URL http://doc.utwente.nl/58230/.

[18] M. Kuntz, F. Leitner-Fischer, and S. Leue. From probabilistic counterexamples via causality to fault trees. In Computer Safety, Reliability, and Security, volume 6894 of Lecture Notes in Computer Science, pages 71–84. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-24269-4. URL http://dx.doi.org/10.1007/978-3-642-24270-0_6. 10.1007/978-3-642-24270-0_6.

[19] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic model checking. In On probabilistic model checking. Springer, N.Y., 2001.

[20] J. C. Laprie. Dependability: Basic Concepts and Terminology: in English, French, German, Italian and Japanese (Dependable Computing and Fault-Tolerant Systems). Springer, 1991. ISBN 3211822968.

[21] D. Lehle. Quantitative safety analysis of sysml models, Nov. 12, 2011.

[22] F. Leitner-Fischer. Quantitative safety analysis of uml models. Master's thesis, Universität Konstanz, 2010. URL http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-125206.

[23] F. Leitner-Fischer and S. Leue. The quantum approach in the context of the iso standard 26262 for automotive systems. Technical report, Universität Konstanz, 2011. URL http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-131130.

[24] S. Leue. Model checking of software and systems, lecture, summer term,2011.

[25] I. Majzik, A. Pataricza, and A. Bondavalli. Architecting dependable systems. In Architecting dependable systems, chapter Stochastic dependability analysis of system architecture based on UML models, pages 219–244. Springer-Verlag, Berlin, Heidelberg, 2003. ISBN 3-540-40727-8. URL http://dl.acm.org/citation.cfm?id=1768179.1768192.

[26] M. Marsan and G. Chiola. On petri nets with deterministic and exponentially distributed firing times. In Advances in Petri Nets 1987, volume 266 of Lecture Notes in Computer Science, pages 132–145. Springer Berlin / Heidelberg, 1987. ISBN 978-3-540-18086-9. URL http://dx.doi.org/10.1007/3-540-18086-9_23. 10.1007/3-540-18086-9_23.

[27] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault tree handbook, 2002. URL http://handle.dtic.mil/100.2/ADA354973.

7 Abbreviations and Explanations

ASR Airport Surveillance Radar. This is the whole radar system.

SSR Secondary Surveillance Radar. This is the digital radar system which communicates with the transponders of the aircrafts.

PSR Primary Surveillance Radar. The classical radar system where the rays are emitted from the antenna and then are received by the receiver.

Clutter-Map A map where the environmental reflections (houses, trees, etc.) from the *PSR*, are saved

SP Signal processor. The analog-digital converter, which converts the high frequent radar signals from the *PSR* to digital values.

ST Sensor tracker. It tracks the aircraft's ways through the range of the radar

PEX Parameter extractor. Here the digitalized data from the are seperated into aircrafts and environemt.

VU Validation unit. It checks whether the outcome of the is plausible or not.

XMI XML Metadata Interchange. It is a standard defined, like the UML specification, by the Object Management Group (OMG) and is used to save and exchange UML-diagrams and metadata between different programs.

Appendix

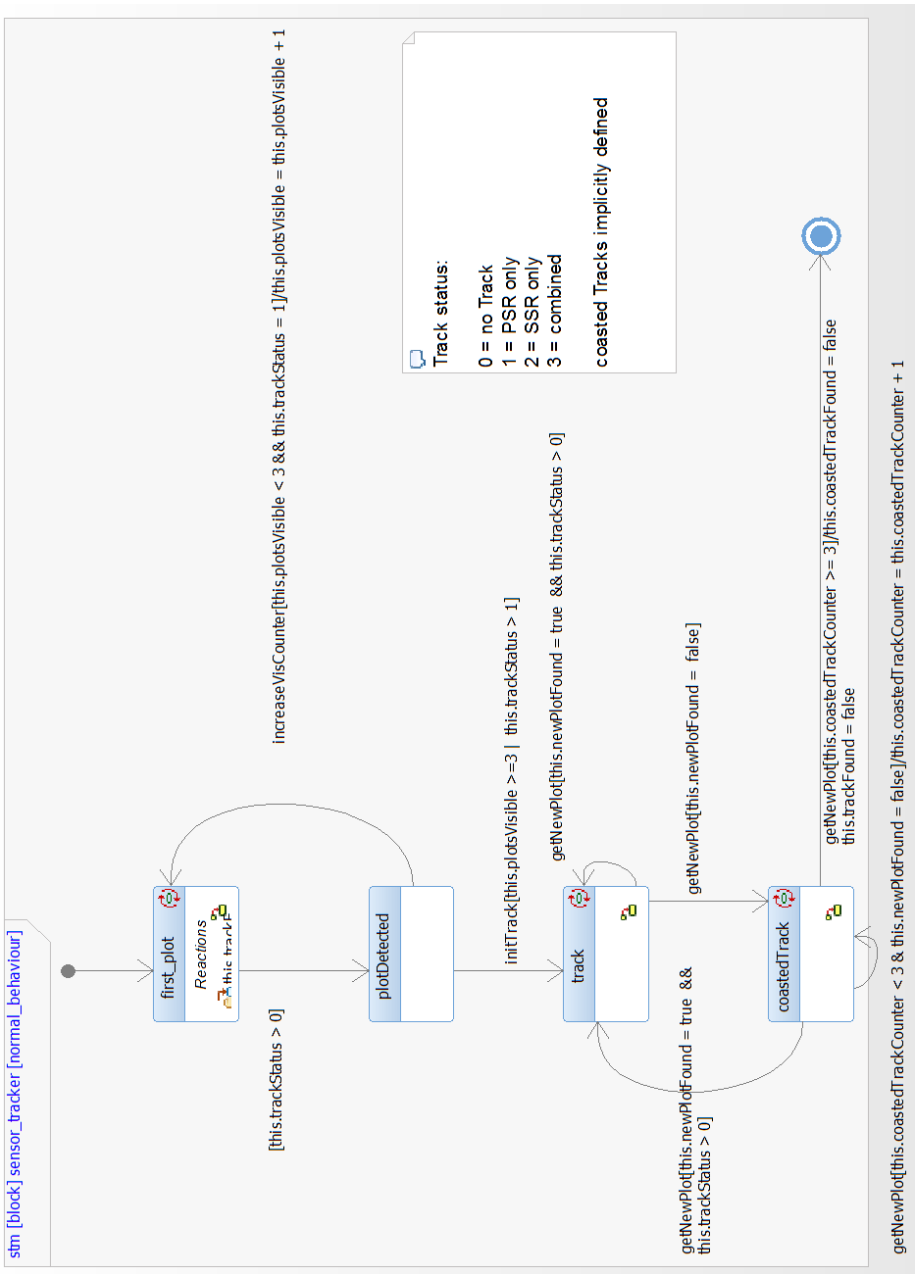


Figure 12: The state-chart of the sensor tracker

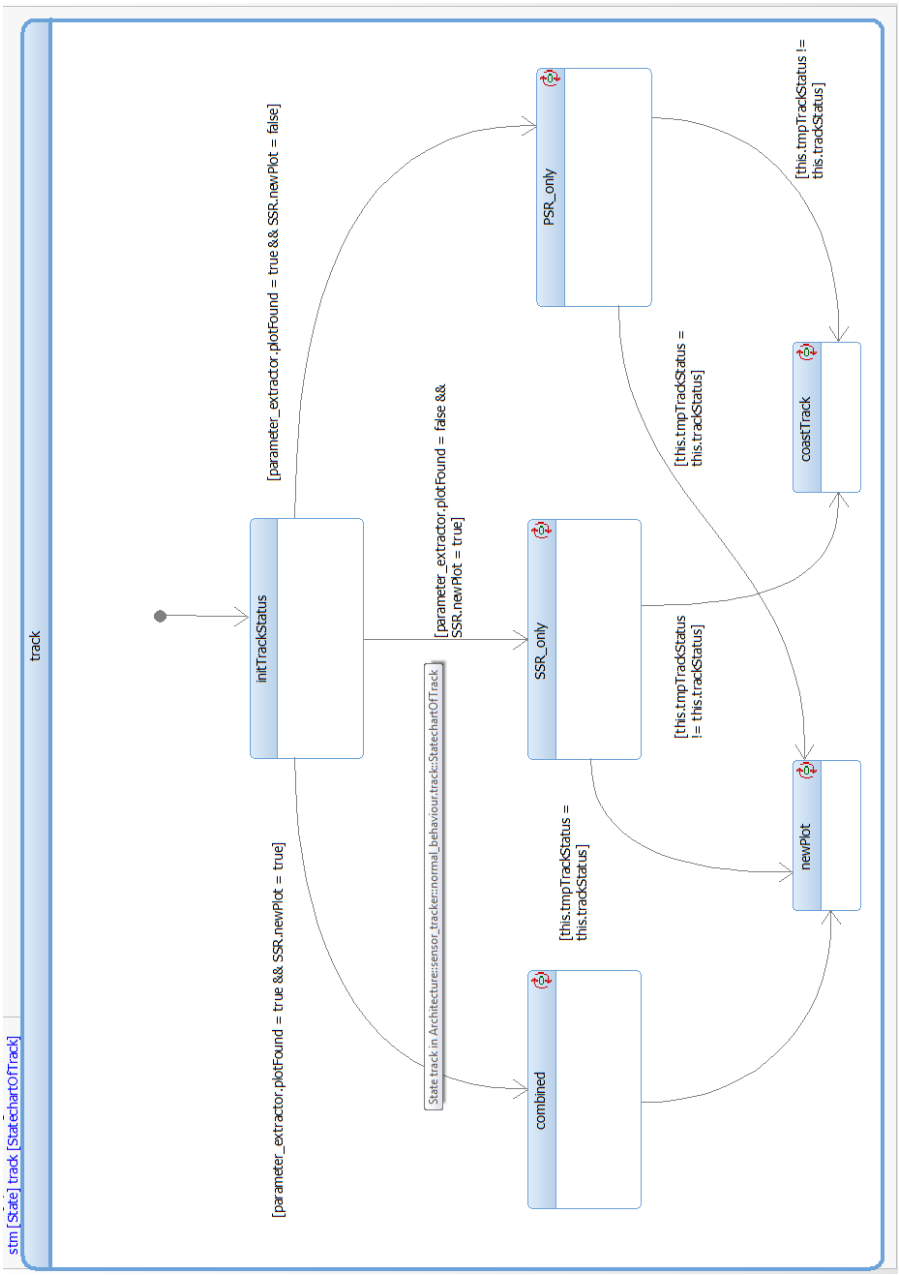


Figure 13: the sub-state-chart of the state track from the sensor tracker

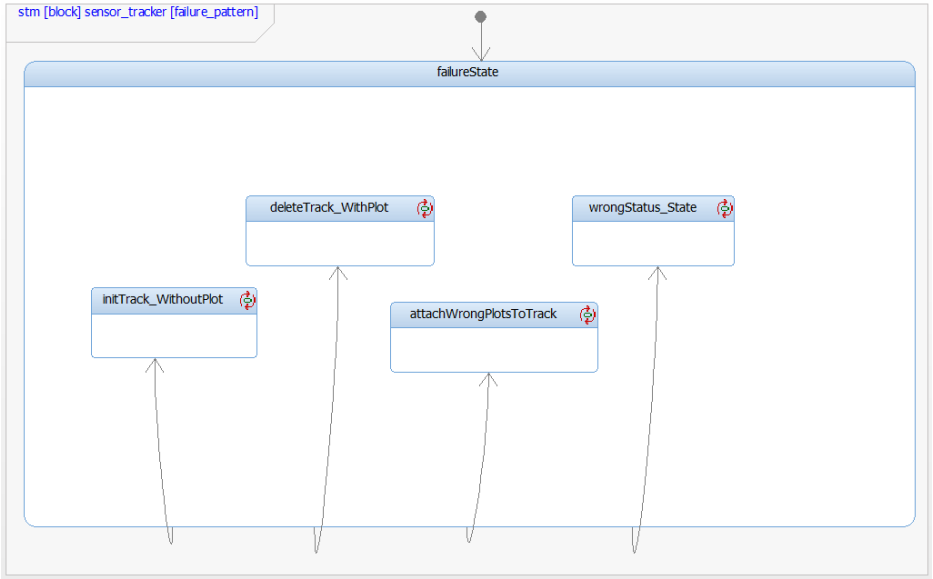


Figure 14: The state-chart for the failure pattern of the sensor tracker

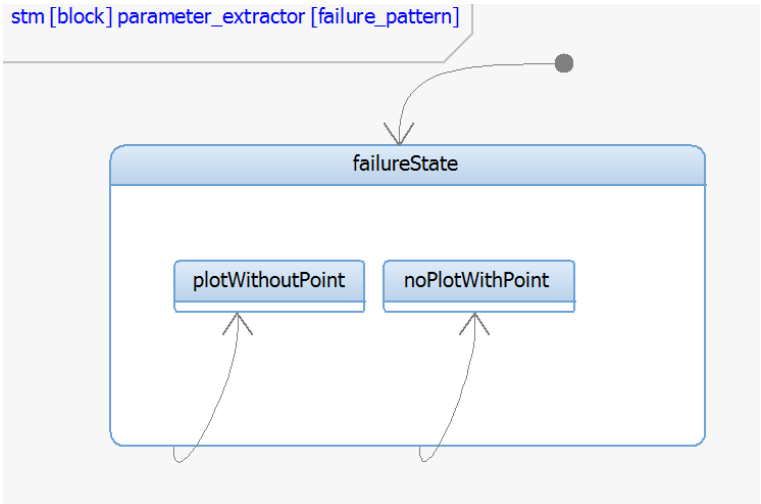


Figure 15: The state-chart for the failure pattern of the parameter extractor

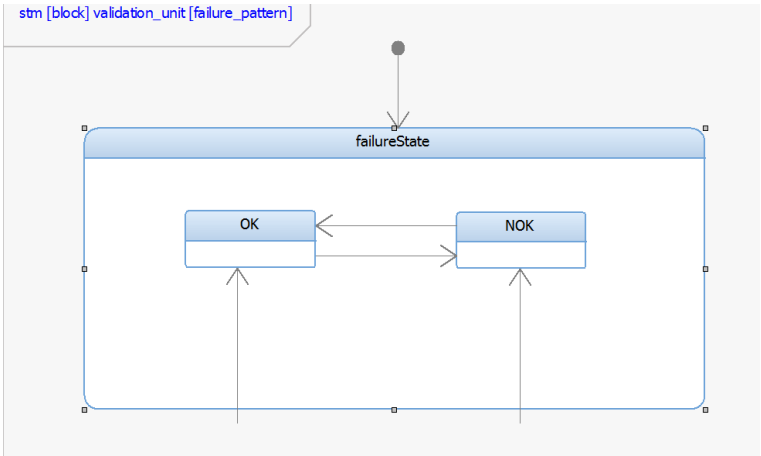


Figure 16: The state-chart for the failure pattern of the svalidation unit