Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Simulation, inference and verification for state-based advanced reliability models of complex cyber-physical systems

Scientific Students' Association Report

Author:

Simon József Nagy

Advisor:

Kristóf Marussy

2014

Contents

Abstract

Safety-critical cyber-physical systems, such as embedded control systems in the automotive, aeronautic and nuclear domains, must satisfy numerous extra-functional requirements. In addition to functional requirements, extra-functional properties of interest are safety, reliability and availability criteria. Standards, such as ISO 26262 for automotive, EIC 61513 for nuclear and IEC 61508 for general electronic systems require addressing these issues during system design. Commonly, the certification process uses top-down deductive analysis of the architecture and behaviors of the system to verify safety properties.

The increasingly complex behaviors of the system, including fault avoidance and other safety mechanisms, pose significant challenges in these analyses. Not only classical hardware redundancy mechanisms are employed, but also fault-tolerant sensor fusion algorithms and adaptive reconfiguration. We need expressive modeling approaches to precisely describe the possible system behavior and faults, as well as tool support for calculating reliability metrics.

Fault trees and dynamic fault trees are widely used to describe fault propagation in systems. However, they are limited to components without internal state. State-based formalisms, such as Markov chains and stochastic Petri nets are also frequently employed, but they provide a much lower level of abstraction than other systems engineering languages. The AADL Error-Model Annex provides state-based modeling by communicating automata, which integrates with the AADL architecture modeling languages, and it is more suited to the systems engineering process.

Typical computation tools for reliability models, such as the PRISM model checker or GreatSPN [6], are limited to a small set of fault probability distributions and are burdened by the state space explosion problem in analyzing large models. Statistical approaches, such as UPPAAL-SMC [7] alleviate some of these issues, but their sampling procedures may lead to large errors when computing with conditional probabilities and low probability events.

In this paper, we propose a safety and reliability modeling approach based the Gamma Statechart Composition Framework, which is a systems engineering toolchain based on communicating statecharts supporting code and test generation and formal analysis. Our approach extends the methodology proposed in the AADL Error-Model Annex by the advanced modeling features of statecharts, such as hierarchical states (faults), as well as local variables and actions (for modeling fusion algorithms). We perform deductive analysis on the models, as well as integrate the reliability models with the Pyro deep probabilistic programming environment for simulation and statistical inference based on state-of-the-art conditional sampling algorithms. This allows handling arbitrary fault distributions and can cope with low probability and conditional events.

We demonstrate the usefulness of our approach on a case study from the automotive domain. After modeling the component errors, error propagation and fault-avoidance

mechanisms, we examine the reliability of the system and perform conditional analysis with the help of deep probabilistic programming, as well as Pareto analysis to find critical components and situations. Hence statechart modeling techniques common in systems engineering, classical deductive safety analysis and state-of-the-art statistical techniques can be combined to ensure system safety.

Chapter 1

Introduction

In safety-critical cyber-physical systems, such as aerospace, nuclear, automotive, and railway domain, the protection of the human lives is the primary objective. Therefore modern critical systems contain large number of safety mechanisms. Adaptive reconfiguration strategies provide safe operation even in case of multi-point failures as the remaining components redistribute the safety critical functions. Moreover, modern systems contain advanced sensor fusion algorithms which drastically decrease the uncertainty of incoming the sensor data with the help of Bayesian statistics and neural networks.

At design-time, besides functional requirements the engineers of the system have to analyse several extra-functional, safety, reliability, and availability (dependability) requirements. These requirements are defined by the industrial standards such as ISO 26262 for automotive, EIC 61513 for nuclear IEC 62279 for railway and IEC 61508 for general electronic system.

But traditional analysis techniques have difficulties in case of modern adaptive systems since these analysis techniques are often unable to tackle the complexity raised by reconfiguration/adaptation. Since fault trees and dynamic fault trees are unable to capture the behaviour in various dynamic configurations of the system, consequently fault trees are unable to model reconfiguration strategies. Furthermore, fault trees have difficulties with representing complex sensor fusion algorithms for various reasons: sensor fusion algorithms are usually not memoryless, but even if the sensor fusion algorithm is memoryless, the logical connection between the failures of the data sources and the output of the algorithm is described with the high expressive power of programming languages and it does not have a predefined state space.

Beside fault trees, other formalisms such as Markovian models and stochastic Petri nets are also applied in the field of qualitative analysis. Petri nets provide a high-level formalism and Markovian methods are used to analyse the underlying Markov chain models. Markov chains are a low-level formalism that can not be directly used in the engineering practices. Moreover, the state-space explosion problem is inherent in Petri net based models, because the state-space of these model grows exponentially with the size of the system-under-analysis. Therefore the verification of these models is difficult even though the supported set of distributions is limited in case of Markov-chains and Petri-nets. Petri nets and Markov chains have a static structure that makes the representation of the different modes of operation difficult.

In addition several state machine-based modeling method was emerged such as UPPAAL-SMC, which uses communicating stochastic timed automata. These solutions are usually too complex to be analyzed with exact solvers. Consequently, simulation-based analysis

approaches are also widely used, and they support many kinds of modeling methods. On the other hand, these methods are really inefficient and wasteful if the analysis targets also small probability events. Furthermore, it is a challenging task to determine the accuracy of these methods, which makes difficult to verify the results of the simulations.

In this paper, we introduce a new safety analysis approach for complex cyber-physical systems. Our approach exploits high-level design languages to describe the system under analysis. We provide various languages to describe the different aspects of the systems and we support the engineers to use the language they are familiar with, because our models are based on high-level formalisms and standards such as statechart and SysML.

In addition, we created a multi-level top-down modelling technique, which supports the deductive safety analysis as well as systems engineering. Furthermore various transformations and mappings are also given to bridge the gap between engineering and analysis models. Thus the engineering models can be directly translated to analysis model. As a result, we can analyse directly the system model immediately even in the concept phase of the development. Additionally, due to the high-level modeling languages, our safety analysis approach provides an easy-to-use end-to-end solution, and do not require any special expertis or competence.

Therefore with the help of our approach we can analyse easily many ideas and system variants to choose the best ones. Moreover, we can verify the compliance of the safety requirements with an additional Markovian analysis technique according to the industrial standard ISO 26262 safety analysis requirements.

We can use one model for number of different analysis algorithms to support the hardware and software design in every phase of the development. Thus our approach support includes several analysis algorithms, with which we can identify the dominant contributors of the system failure revealing which hardware components are related mostly to the system failures. Beside that, we can discover the hidden weak-points of the safety concept and the safety mechanisms. Since we can examine in details the system behavior focusing on the cause-causation relation between the errors and the safety goal violations determining the system level effect of a component failure. Moreover, all these analysis techniques can be executed separately on the components. Therefore we can examine one-by-one the safety mechanisms of the system such as sensor fusion algorithms.

In Chapter 2 the necessary background of the new approach is presented together with the state-of-the-art safety analysis techniques. In Chapter 3 the overview of the novel approach and analysis algorithms are presented, together with a case-study from the automotive industry. Furthermore we present our deductive modeling technique in Chapter 4, with the modeling of the case-study. Additionally, in Chapter 5 we present our analysis algorithms and in Chapter 6 we show how we can apply these methods to the case-study model with run-time measurements. Finally, in Chapter 7 we summarise the results of the measurements and present the future possibilities of our approach.

Chapter 2

Background

In this chapter, we present the theoretical background of our analysis method. In the first section, we define the main concepts of safety and safety analysis, including the concepts of malfunctioning behavior.

After that, we introduce the state of the art fault models. We show their formalism their advantages and their disadvantages. Then we present the state of the art analysis techniques. Finally, we present the industrial standards related to the safety-critical systems.

2.1 Qualitative metrics of dependability

In this section, we present all the concepts and definitions that are related to safety, reliability, and availability. These definitions are based on the various standards and papers, such as the standard ISO26262 [1] and [5] .

2.1.1 Definitions related to the correct operation

Reliability represents the continuity of the correct service, which can be measured by the average frequency of the service interruptions. In a reliable system the frequency of service disruption is under a tolerable limit.

Availability represents the readiness of the correct service. Availability means that the capability of the system to provide all safety critical function in all specified circumstances, defined by the customer.

Safety means the absence of unreasonable risk and any other factor that can result catastrophic consequences, which is unacceptable for the moral concepts of the society.

Dependability represents the capability of the system to avoid that the service failures occur more frequently than it is acceptable.

2.1.2 Definitions related to the malfunction of the system

Error represent difference between the expected and the actual behavior of the system. Every error is originated from a fault. We call every state of the system, which does not meet the expectations, erroneous.

Fault is the erroneous behavior of a part of the system. This condition can lead to bigger problems in the system.

Failure mode represents the way that the system or a part of the system fails. Every part of the system can have several different failure modes. Furthermore different failure modes have different effect on the availability and on the reliability of the system.

Failure represents the situation when the system or a part of the system stop any function as it is required. There is system failure, if any safety critical function is incorrect or interrupted.

Failure rate is a mathematical measure which specify the distribution of the failure occurrence over time. Failure rate represents the average number of expected faults in a given period of time. If the failure rate is constant over time, then the failure can be described by the *exponential* distribution [21]. In this case, we denote the failure rate with "λ".

Random hardware failure represents if we are unable to predict the time when the system or a part of the system fails. We model the uncertainty of these failures with a random variable and its stochastic distributions. Most random hardware failure can be modeled with an exponential random variable accurately.

Permanent fault is a fault, which will not disappear without external intervention. The system remain faulty until an external repair. Most hardware faults are permanent.

Operating time is the sum of the periods when the system was operational.

First-time-to-failure (FTTF) or just time-to-failure (TTF) is the time until the first interruption of any safety criticall funtions. FTTF can be represented with a random variable with a given distribution. The average value of this random variable is the mean-time-to-failure (MTTF).

Phase is the stage of the safety lifecycle during the development of the system. For example in the automotive industry there are four phases of the development (according to ISO26262):

- concept,
- product development at the system level,
- product development at the software level, and

- production and operation.

Cascading failure The failure of a part of system can cause the failure of other part of the system. We call all this type of event cascading failure.

Common cause failure Sometimes a event in the system cause that two part of the system fails at once. We call all this type of event common cause failure.

Component is every non system-level element of the system that are clearly identifiable.

2.1.3 Definitions related to the avoidance of the failure

Safety mechanism Modern critical systems contain several advanced solution to increase the availability and reliability. We call these solutions safety mechanisms. They are always implemented in the hardware or the software of components. Complex components contain several different hardware and software safety mechanisms. Safety mechanisms can either bring the system into a safe state or alert the user.

Safety measure is every mechanism or function of the system which can either:

- avoid or control systematic failures,
- detect or control hardware failures,
- mitigate the effects of failure.

All safety mechanism is a safety measure.

Safety concept is the collection of all specification of the functional safety requirements.

Safety goal is a top-level requirement of the system. It usually specifies the correct operation of the safety critical functions.

System is a set of components, which is connected to at least a sensor, a actuator and a controller for the actuator. A system can be a component of another system.

Functional safety requirement of a system is the specification of all the implementation-independent safety related behavior. These requirements can contain information about the attributes of the safety measures and the applied technologies in the system.

Hardware architectural metrics is a measure which specifies the effectiveness of the system from a given safety perspective.

Malfunctioning behaviour is every type of operation which endanger any functionality of the system or a component.

Redundancy is the coexistence of components which have the same function in the system. The main purpose of the redundant component is to provide a specific function even if there is a hardware failure in the system. For example if there are two light bulb in a room, there will be light, even if one bulb is faulty.

Safety goal violation is the state of the system when any requirement of the safety goals is not satisfied.

Safe state is an operation mode of the system, where despite a fault in the system, the effect of the fault is reduced and the probability of a safety goal violation is reasonably small.

Error propagation When a cascading failure in the system causes the malfunction of other component, we speak about error propagation. The propagation of an error can be stopped by a safety mechanism or cause a safety goal violation. There are some cases when a fault can not cause safety goal violation at all. In this case, we speak about a safe fault.

Latent fault A fault is regarded a latent fault if it is undetected by the safety mechanisms.

Adaptation Every system, which is capable of reconfiguring its behaviour, is called an adaptive system. Thus, if some faults occur in an adaptive system and the system detects it, then the system executes a reconfiguration strategy to minimise the probability of a system failure. All adaptations need at least one safety mechanism to be executed.

FIT Failure in time (FIT) is the measure of the average number of failure occurring in 10^9 operating hours. FIT is a probabilistic measure if the system contains at least one component with stochastic failure model.

Reconfiguration If there are some fault in the system, in case of reconfiguration, the components of the system redistribute the safety critical funtions among the remaining, operational components.

Adaptive system is every system, which is capable of reconfiguring itself according to environmental/external inputs.

Cyber-physical system

Sensor fusion creates new information from data collected from multiple sources in order to filter out the uncertainty of the single data sources. Modern cyber-physical systems contain several redundant sensors, that collect data about the same phenomenon, and use sensor fusion to provide general and also safety-critical functionalities.

SysML Systems Modeling Language (SysML) [19] is a widely used modeling language in the field of systems engineering. SysML is an extension of UML [23] and is compatible with IEEE-Std-1471-2000 (IEEE Recommended Practice for Architectural Description of Software Intensive Systems) [28]. Additionally, the use of SysML is usually required in field of critical systems.

2.1.4 Standards

In the last 50 years, several standards were developed in various domains to guarantee safety in every circumstance. These standards specify requirements about the design process from the concepts to the production. Every step of the design process must conform to the standards (including also the safety analysis, which is also defined by the standards). Therefore these standards also define requirements about how to analyze the system. Most standards require the application of an inductive and also a deductive safety analysis method. The latter should be a bottom-up technique, which examines every part of the system and investigates which combination of the faults can lead to safety goal violation.

The most popular deductive analysis method is the fault mode and effect analysis (FMEA) and the fault mode and effect diagnostic analysis (FMEDA). The FMEA examines every hardware part of the system one-by-one and calculates the failure rate of every failure mode of the hardware parts. Furthermore, FMEA investigates whether a failure mode can cause safety goal violation or not. The FMEDA extends the FMEA with the analysis of the diagnostic coverage. The diagnostic coverage defines the probability that the given failure mode is detected and mitigated by one or more safety mechanisms. In the case of some industrial standards, the coverage can not be higher than 99% unless the engineer applies some particular analysis method on the diagnostics.

In contrast, inductive methods do a top-down analysis of the system. These methods examine every possible way the system can become faulty. Then these methods search for every possible source of the system failure. An inductive method examines every component of the system recursively to find the source of the system-level failure events. This process goes on until we reach the atomic sources of the failures. The most popular top-down analysis method is the fault tree analysis (FTA), which models the system with logic gates and failure events. Thus, this method traces back the inductive analysis to the analysis of a logical network. Additionally, several extensions were introduced, such as dynamic fault trees (DFT) and hybrid fault trees (HFT), that improve the functionality of the original FT.

One of the essential objectives of the safety analysis is repeatability and verifiability. All the steps of the analysis must be reviewed and oversaw by both the engineers and external experts. Moreover, all the used algorithm must be verified, or its output has to be verified manually. Furthermore, the safety analysis usually does not include software faults because the software failures are always systematic. The safety analysis focuses on random hardware failures and their effects on system behavior.

One of the most modern industrial standards is the ISO26262 standard, which defines the functional safety requirements of automotive electronics. ISO26262 is the adaptation of the IEC 61508, which is the general standard of the critical electrical and electronic systems (E/E). ISO26262 is intended for every automobile which is manufactured in series production and weights at most 3,500 kg. It specifies four levels of product development:

- Concept phase,

- System-level development,
- Hardware-level development and
- Software-level development.

Moreover, ISO26262 specifies supporting processes for development like documentation and safety analysis. In addition, safety analysis should be applied at every level of development. Furthermore, ISO 26262 also requires the application of a top-down and a bottom-up analysis method.

2.1.5 Basic error models

In order to describe the behavior of random hardware faults, random variables are used. These random variables have their stochastic distribution, which specifies the occurrence time of the different failure events.

Engineers use several different distributions to model the occurrence of random hardware failures. The most widespread distributions are the normal, the exponential, and the Weibull distributions.

Normal distribution We use the normal distribution when the time of the failure is known with a given uncertainty. The normal distribution has two parameters, namely, the mean (denoted by μ) and the standard deviation (denoted by σ). The mean specifies when the system will fail on average, and the standard deviation defines, on average, how much TTF can differ from the mean.

The normal distribution starts in minus infinity, and thus in practice, the truncated normal distribution might be used instead of normal distribution. The truncated normal distribution is a normal distribution of which tail is cut off. The truncated normal distribution always starts in zero, but its shape is the same as a normal distribution (with the same μ and σ parameters) in the positive region.

Exponential distribution The exponential distribution is the simplest of all distributions. It represents an event in a system, which has absolutely no memory. It means that aging has no effect on the system. If the system is alive after a given time, then it does not influence the expected remaining time. The only parameter of the exponential distribution is the rate parameter (denoted by λ), which defines the average number of fault in a given period. In addition, the mean time to failure in the case of the exponential distribution is the reciprocal of the rate parameter.

Weibull distribution In contrast to the exponential distribution, the Weibull distribution is applied when the aging has a major effect on the occurrence of the failure. The Weibull distribution has two parameters[1] namely, scale (denoted by λ) and shape (denoted by k) parameter. The shape parameter defines how aging affects the system:

- If $k < 1$, then the aging increases the probability of survival. In this case, the failure rate decreases over time. These types distributions represent a group of systems where the weak ones become faulty fast, and the strong ones live long.

[1]The three-parameter Weibull distribution also exists, but it is not used in this paper.

- If $k = 1$, then the aging does not affect the system. In this case the distribution is exponential.
- If $k > 1$, then the aging has a negative effect on the system, and the failure rate increases over time. For example, the mechanical component has this behavior because the cyclic stress causes fatigue. Additionally, the field-effect transistors in power electronics have the same behavior [43].

2.2 Modeling error propagation and fault avoidance

In this section we introduce the state-of-the art safety modeling methods. These methods introduce some mathematical modeling language, which describe the connection between the random hardware faults and the operational mode of the system.

2.2.1 Fault tree and dynamic fault tree

Fault tree (FT) is the most commonly used safety modeling method. FT is used for automotive as well as aerospace and nuclear systems. FT is a top-down deductive analysis method, which searches all possible source of a system failure. FT specifies the logical connection between the random hardware faults and the system failure. In FT the logical connection is defined by logic gates. The types of logic gate can be AND, OR, XOR, NOT, NOR, NAND and VOTE. We can create if we decompose step by step the failures to the combination of smaller failures.

Đynamic fault tree (DFT) is an extension of the fault tree. Despite FT the DFT can contain time requirements. As a result in the DFT beside the combination of the faults also the sequence of the occurrence determine the state of the system. As a result, the DFT contain several new types of logic gate, which output is "*true*" if and only if the desired fault combination occur in the desired order. These new types are the PAND, the SEQ, the SPARE and the FDEP. For example the SEQ gate is the sequence enforcing gate, the output of which is "*true*" if and only if the inputs of the SEQ gate become "*true*" in a given order.

The greatest advantage of the FT is its simplicity, one can easily understand a FT without any particular knowledge. On the other hand, the greatest disadvantage of the FT is the limited expressiveness. For instance, the sensor fusion algorithms can be described only with complex logical connections, which require enormous amount of logic gates, for example an extended voting need 25 traditional logic gate.

2.2.2 Stochastic Petri nets

Stochastic Petri (SPN) [33] net is a modeling language especially good for distributed systems. Every SPN consists of places and transitions. Moreover, directed arcs lead between places via transitions. The transitions define the possible state changes and every transition delays the change randomly. Furthermore random variables define these delays. The advantage of the SPN that it can directly translated to a Markov-process for analysis. On the other hand, the SPN can not model neither the reconfiguration and the operational states of the system. Moreover, the SPN is a low level modeling language consequently, modeling of large and complex systems is a cumbersome, time-consuming process.

2.2.3 AADL Error-Model Annex

The Architecture Analysis and Design Language (AADL) is a tool set for model-based engineering, which support the analysis of both the software and the hardware of the system. The AADL Error-Model Annex is a modeling language, which specifies the architecture and the error propagation in the system. The AADL Error models can be easily visualised and several graphical editors exist for AADL. The advantages of the AADL that we can use a single model for multiple analysis, AADL has a precise but expressive syntax and allows the life-cycle tracking in the analysis. As a result, it is used in the military domain by the DARPA.

2.3 Statechart modeling

Statecharts are the advanced state-based models in systems engineering. Statecharts provide a hierarchical, high-level representation for reactive systems. Statecharts consume events from their inputs, and as a reaction, they provide output events. Statecharts can represent complex logic and data-dependent behavior. Statecharts provide various constructs to support the modeling: states and complex states are used to define concurrent and hierarchical behavior, events, transitions, and actions are used to change the state of the system and produce a reaction, variables and operations are used to store and manipulate complex states

In Figure 2.1, we can see a simple statechart model from the case study. It has a complex state with two orthogonal regions. *Main* is the initial state of the statechart and it has two orthogonal regions namely, *Status* and a *Output*.

When entering the *main* state, the variables are set to a specific value The state transition from *Normal* to *Warning* state is triggered by the events *S1HW.det*, *S2HW.det* and *S3HW.det* events. It means that if any of these three event are raised then the transition will be activated.

Moreover in the *Status region* at every *S1HW.det*, *S2HW.det* and *S3HW.det* events we decrease the variables *on_sensors* and if we reach zero we go to state *Error*.

As it is easy to see on the example, state machines are a convenient formalism to model the behavior of reactive systems, which process events from the environment and react to them in accordance with their internal states. Statecharts [24] are a popular extension to the state machine formalism that features complex constructions to support high-level design.

2.3.1 Elements of statecharts

The primary feature of statecharts is that every (composite) *state* may specify its own state machines using one or more *regions*. A region may contain several states, out of which a single one can be active at a time. If a state contains multiple regions, they are called orthogonal. When a composite state is entered or exited, each region of the composite state is entered or exited at the same time, respectively. Also, states may have *entry* and *exit actions* that are executed each time a particular state is entered or exited. Statecharts support the concept of memory by means of *variables* and *history states*. If a region is entered through a history state, the last active state of the region will be entered. The definition of complex actions is supported with composite transitions. Choices of

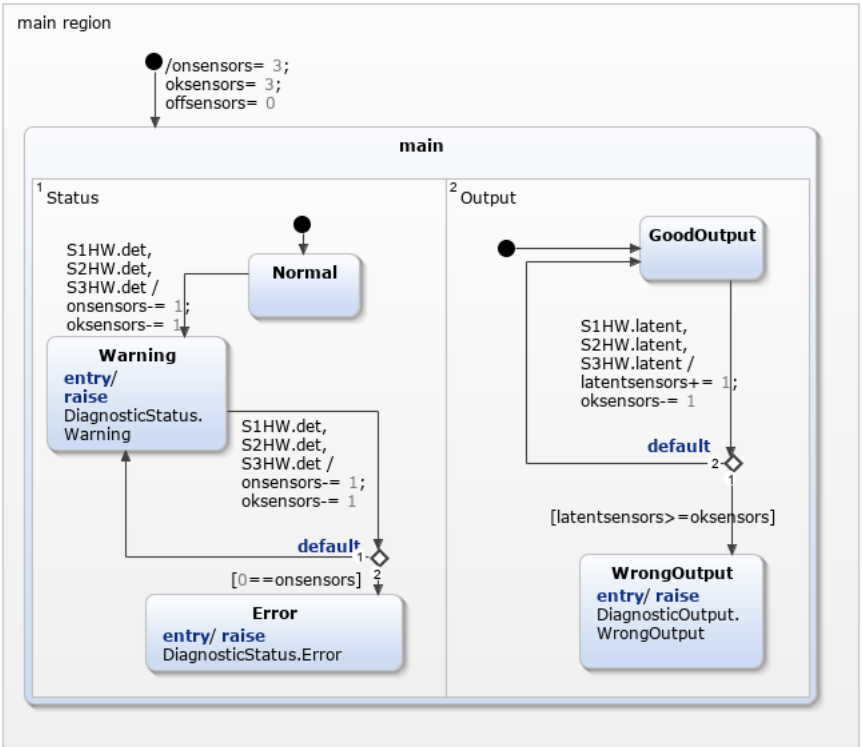


Figure 2.1. Example of a statechart

actions with respect to certain conditions can be defined using *choice transitions*, whereas activities in orthogonal regions can be synchronized using *fork* and *join transitions*.

2.3.2 Gamma Statechart Composition Framework

The model-based design of state-based systems can be supported with modeling tools. The Gamma Statechart Composition Framework[2] [35] is an open-source, integrated modeling toolset aiming to support the semantically sound composition of heterogeneous statechart components. The framework purposefully reuses statechart models of third-party tools and their code generators, e.g., YAKINDU Statechart Tools[3], for separate components. As a core element, the framework provides the Gamma Composition Language, which supports the interconnection of components hierarchically based on precise semantics. Moreover, Gamma provides automated code generators for created models as well as test case generators for the analysis of interactions between components. Gamma also supports system-level formal verification and validation (V&V) functionalities, that is, the system model can be exhaustively analyzed with respect to formal req uirements, by mapping statechart and composition models into formal automata of the UPPAAL[4] [7] model checker.

[2] http://gamma.inf.mit.bme.hu/
[3] https://www.itemis.com/en/yakindu/state-machine/
[4] http://www.uppaal.org/

2.4 Analysis approaches

In this section we present the analysis appoaches of the safety models (introduced in the previos section).

2.4.1 Markov chains solution

In the recent years several Markov-chain-based solution appeared like PRISM [30] and Storm [18]. They use symbolic Markovian model languages like Jani [11] or PRISM[5]. In this language, several rules and variables define the state and the transition probabilities of the Markov model.

The PRISM and the Storm model checker can analyze several different types of Markovian models. PRISM supports discrete- and continuous-time Markov-chains, discrete- and continuous-time Markov decision processes and stochastic Games. Storm can analyze discrete- and continuous-time Markov-chains, discrete- and continuous-time Markov decision processes and Markov automates.

On the other hand, the symbolic Markov-chain solvers have several disadvantages. Similarly to the FT-s, the state-space of the symbolic Markov-chains grows exponentially with the size of the model. Furthermore, the creation of a symbolic Markov-chain is a time-consuming task because the PRISM and Jani are low-level modeling languages. Furthermore, because of the state-space explosion, solving a continuous-time Markov-chains and Markov decision process are exponentially hard tasks. As a result, the solvers use many sophisticated algorithms, thus the certification and the verification of these algorithms are extremely hard.

2.4.2 Simulation, statistical model checking

Simulation-based analysis methods are widespread in the field of engineering. Because of the simplicity of the simulation in the field of railway [38], automotive [22] and aerospace [40] industry. They estimate the probabilities of the possible outcomes of a random event with the relative frequency. These methods run a large number of simulations and examine how many times the specified event occurred. Because of the ergodic theorem [34] these methods can approximate the distribution of the occurrence time of a given event. Thus these methods can estimate the probability density function with the histograms of the simulation data.

Several modeling tools provide a simulation-based statistical analysis method. The UPPAAL-SMC [17] can analyze stochastic timed automata (STA), which is a state machine based model type. ISOGraph [2] also provides Monte-Carlo analysis methods for FTA.

These methods can easily analyze events that occur with relatively significant probability, yet the traditional simulation-based methods are ineffective for analyzing low probability events because traditional methods can not model conditional models and conditional probabilities.

[5]The PRISM model checker, and the PRISM modeling language has the same name. The PRISM model checker is created to analyze the PRISM models.

2.4.3 Fault tree analysis

The fault tree analysis is the quantitative evaluation of the FT in order to investigate whether the safety requirements are satisfied or not. There were several algorithms developed to calculate MTTF, failure probability, and average failure rate from the FT-s.

Manual evaluation If all events in the FT are independent and the output of every gate and every event is connected to at most one gate, then the FT can quickly be evaluated. Thus we can easily calculate the MTTF and the failure probability. On the other hand, these constraints limit the too much the expressiveness of the FT-s. Consequently, this method used only for small and simple systems.

Markov-chain approach For analysis, the FT can be translated to a continuous-time Markov chain, which can be easily analyzed with existing Markov-chain solvers. The advantage of this method that it traces back the FTA to an existing solution. On the other hand, this FTA method inherits the disadvantages of the Markov-chain analysis. As a result, this method works only if the events have exponential or psi distribution, and the state-space of the translated Markov-chain grows exponentially with the number of events in the FT. Moreover, the DFT can be translated to a continuous-time Markov decision process, and this fact makes the analysis of the DFT (DFTA) much harder because the MDP solvers are much slower than the MC solvers [18].

Simulation Simualtion based appoaches are used for FTA and DFTA [14, 32, 44, 41] . These methods analyse the system with a huge number of virtual measurements. These measurement take place in the a simulation time. The simulation-based analysis methods estimates the probability of an event by counting the times when an event occurs in the simulation time and divide it with the number of simulations. However these methods can simulate the output of the gates but traditional simulation-based FTA methods are unable to calculate conditional models. This makes difficult to support the design process with this method.

2.5 Probabilistic programming

To deal with *uncertainty* in computer-controlled systems, a new programming paradigm, called probabilistic programming was introduced, which facilitates the development and analysis of complex models with both deterministic and stochastic properties. In the last ten years, a large number of tools were developed to support this paradigm, e.g., Stan [13] Edward [42], Anglican [4], PyMC [39] and Pyro [8].

In comparison to other tools and approaches, their greatest advantage is the *inference algorithm*, which can analyze advanced conditional models [9] and calculate posterior distributions. These type of models consists of a prior and a posterior model the prior model represents the overall probabilistic system behavior, which describes in general the output of the system.

In contrast, the posterior model observes some of the output of the prior model. In other words the posterior model assumes that some of the outputs of the piory model have a specified value. For instance, an average hardware component have two kind of probabilistic output the failure mode and the failure time. If we want to analyse the

failure time, when the system fails with a specified output. Then we have to create a posterior model where we assume that the hardware component fails with the specified failure mode.

We can create a posterior model with the *observe* function. This function denotes in the stochastic model if a random variable is conditioned to a specified value.

We can analyze these types of models only with simulation-based techniques since the complexity of these models is much higher than it could be analyzed explicitly.

Most inference algorithms use a gradient-based Monte-Carlo method, such as the Hamiltonian Monte-Carlo [15] and the No-U-Turn algorithm [26].

Recently, a new approach called deep probabilistic programming [8] emerged that uses neural-network acceleration in addition to the Monte-Carlo algorithm to make the computation fast and efficient for huge models. With the stochastic variational inference algorithm (SVI) [27] they can fit directly a probabilistic model to the posterior distributions, even if the prior model contains parameters.

Stochastic variational inference The stochastic variational inference (SVI) is one of the most commonly used inference algorithms in deep probabilistic algorithm. SVI can analyze models that include conditional properties and parameters. This algorithm traces back the analysis problem to a optimisation task, which can be executed by machine learning algorithms.

Besides the stochastic model, this algorithm needs an optimiser and a *guide* model. The *guide* is a parametric model which defines the type of the sought posterior model. Thereafter, the optimiser tries to find the proper parameters for the *guide* model. As a result, the guide with the parameters from the optimiser resemble the sought model.

2.5.1 Pyro

Pyro is one of the most advanced deep probabilistic programming languages today developed by UBER AI Labs. Pyro combines the two greatest branches of machine learning namely, the deep learning and the Bayesian statistics.

Pyro is written in Python, it can be accessed with a simple and universal syntax that can be used to define even complex models such as stochastic control structures. A Pyro model (called *Pyro stochastic function*) can contain any kind of Python source code. In addition, Pyro includes all the state-of-the-art inference algorithms, such as No-U-Turn Sampler [26], Hamiltonian Monte-Carlo [15] and Sequential Monte-Carlo [20].

Pyro can execute the calculations fast with the GPU-accelerated tensor computation libraries. Pyro is built on the top of the PyTorch open-source machine learning framework. We can utilize the opportunities of PyTorch in two ways. Firstly, we can include neural networks in our stochastic models to model complex reconfiguration strategies and sensor fusion. Moreover, with the help of PyTorch we can replace the guide function with ell-trained a neural network. The latter is especially useful, because creating a proper guide function for complex models is a challenging task.

Example of a probabilistic program Finally we show a simple example about the usage of the deep probabilistic tool Pyro to solve a simple problem in field of safety analysis. Our example is based on the official Pyro tutorials available on *pyro.ai*.

We have to analyse an infrared approximation sensor. its output is 1 if something is in front of the sensor. This sensor sometimes does not provide good output. We know that the probability of a good measurement depend on strongly the measurement configuration. As a result this probability can be everywhere between 0 and 1. We executed 10 measurements and in two measurements the sensor did not perceived the object. From these result we have to analyse where the probability moves.

First of all, we created a the priory model. In this model we sampled the probability of a good measurement from an Uniform distribution. We call this probability *success_probability*. Thereafter, in the model we sampled the 10 measurement from a Bernoulli distribution. In the model 1 means a good and 0 means a bad measurement. We put an *observe* each measurement according to the results of the real measurements. The result of the measurements are in the *data*.

We know that the posterior distribution of the *success_probability* can be approximated with normal distribution. Thus we created a *guide* function which samples the *success_probability* from a Normal distribution. The parameters of this distribution is calculated with the SVI. We created an *Adam* [29] optimiser which found the guide function which is the closest to the real posterior distribution.

```
#measurement data: 0 means bad, 1 means good measurement
data =    [1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,0.0,0.0]

def model(data):
    # success probability  equally  can be anywhere between 0 and 1
    low = torch.tensor(0.0)
    high = torch.tensor(1.0)
    # sample sampling
    f = pyro.sample("success_probability", dist.Uniform(low, high))
    # we observe each measurement
    for i in range(len(data)):
        # in Pyro we sample and observe at once
        pyro.sample("obs_{}".format(i), dist.Bernoulli(f), obs=data[i])

def guide(data):
    #we create the generic parameters for the  posterior  model
    mu = pyro.param("mu", torch.tensor(0.5),
                            constraint=constraints.positive)
    sigma = pyro.param("sigma", torch.tensor(0.01),
                           constraint=constraints.positive)
    #sample the "success_probability"
    pyro.sample("success_probability", dist.Normal(mu, sigma))

# setup the optimizer
adam_params = {"lr": 0.0005, "betas": (0.90, 0.999)}
optimizer = Adam(adam_params)

# setup the SVI inference algorithm
svi = SVI(model, guide, optimizer, loss=Trace_ELBO())

# execute optimisation algorithm
for step in range(n_steps):
    svi.step(data)
    if step % 100 == 0:
        print('.', end='')

# get the learned parameters of the  conditional  model
mu = pyro.param("mu").item()
```

```
sigma = pyro.param("sigma").item()
```

Listing 2.1. example of a probabilistic program

2.6 Related work

In this section, we present the state-of-the-art analysis tools.

2.6.1 PRISM

The PRISM model checker [30] is one of the most popular symbolic Markovian model checker. PRISM uses an exact solver, which can analyse both discrete and continuous-time Markov-chains, discrete and continuous-time Markov processes [21] and stochastic games. These Markovian models can be defined in the PRISM language (the model chacker and the modeling language have the same name), which can describe even complex behaviors. Moreover PRISM can analyse wide range of properties including conditional ones. Additionally, PRISM provide an easy-to-use graphical user interface. On the other hand, the use of PRISM need special expertise in the field of probability theory. Additionally, the translation of a system model into a symbolic Markov-chain is a long and time consuming process, because the PRISM language supports only a low-level modeling.

2.6.2 ISOGraph

ISOGraph Reliability Workbench [2] support all the traditional analysis techniques such as FTA. ISOGraph provides an easy-to-use graphical user interface for editing and analysing models. In addition ISOGraph is certified by both SGS-TÜV Saar[6] and SAP [7]. Because of these certifications ISOGraph is widely used in the automotive industry. On the other hand, the application of ISOGraph need special competences in the field of safety engineering.

2.6.3 UPPAAL-SMC

The UPPAAL is a high-level analysis tool for networks of communicating timed automata. With the UPPAAL we can analyse interconnected state-machines but UPPAAL supports only deterministic models. UPPAAL-SMC (statistic model checker) is an extension of the UPPAAL. The UPPAAL-SMC provides also a graphical statechart editor. On the other hand, the UPPAAL-SMC does not support the analysis of conditional models and the output of the analysis can not be certified.

2.6.4 MDAA Workbench

MDAA Workbench [25] is a composite modeling and analysis tool. With MDAA, we can define state machine-based behavior models and error propagation models according to the AADL Error Annex. Then it can translate the system model into a stochastic analysis network, which can be analyzed with MOBIUS [16]. On the other hand, MDAA supports

[6] https://www.sgs-tuev-saar.com/en/about-us.html
[7] https://www.sap.com/index.html

only the Poisson, and Bernoulli distributions, and MDAA does not support statecharts and conditional analysis properties.

Chapter 3

Overview

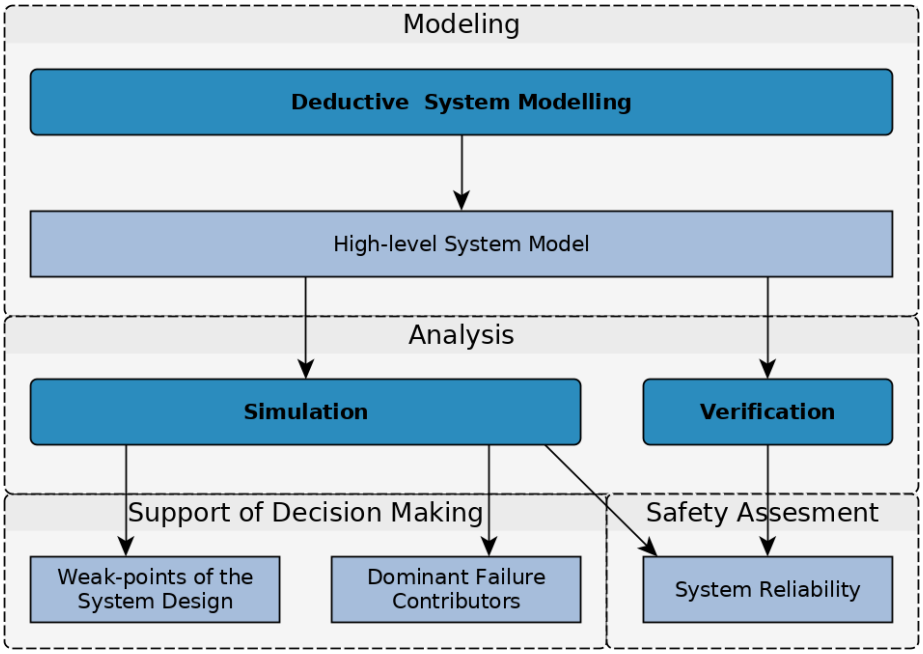


Figure 3.1. Structure of our approach

This chapter presents our integrated, top-down, quantitative analysis method for safety assessment. In contrast to traditional safety analysis techniques, our method is capable of verifying reconfiguration strategies and sensor fusion algorithms. In addition to the safety assessment, our analysis approach supports the decision making by identifying dominant failure contributors to the system failure and the hidden weaknesses of the safety concept. Moreover, our analysis technique can also be applied in the concept phase of the development when the available information is insufficient for traditional methods.

We created a composite analysis approach that consists of an expressive multi-level modeling language, a simulation-based analysis technique, and a state exploration-based verification method. The structure of our approach is depicted in Figure 3.1. For the modeling, we used high-level formalisms and standards, which are widely used in systems engineering to fill the gap between engineering and analysis models. Therefore we used a statechart-based adaptation model beside an AADL-like system configuration, which describes the communication and the error propagation between components with the help of a SysML

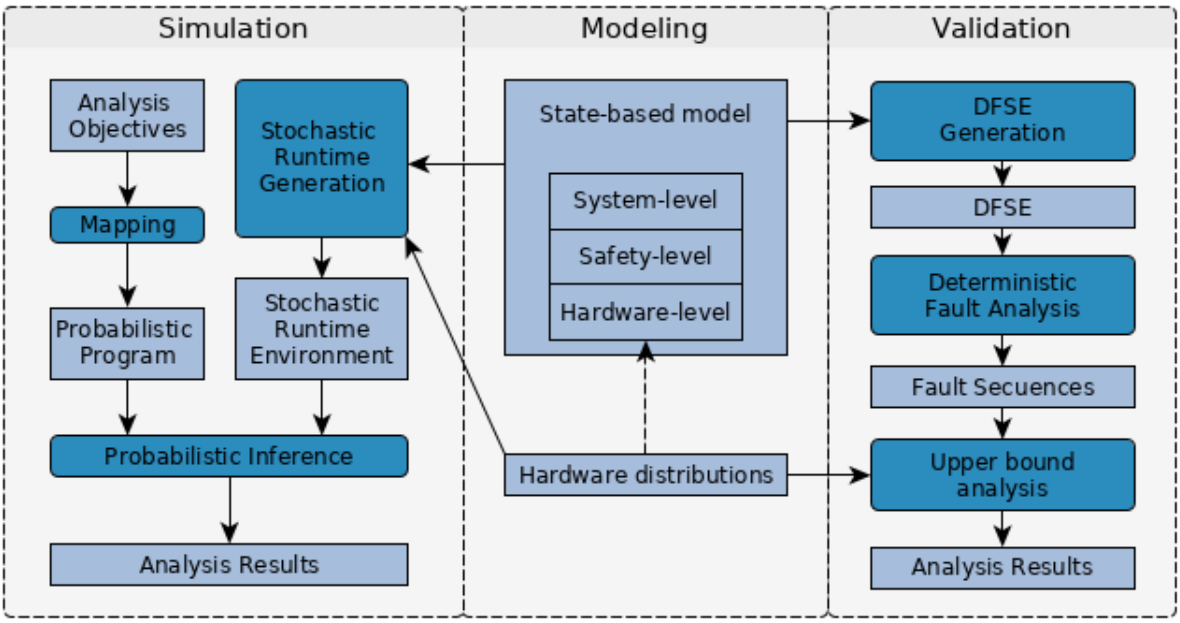


Figure 3.2. Detailed structure of artifacts and methods used during the safety assessment approach

like modeling-language. Thus the application of our analysis technique does not require any special competence or expertise (beyond traditional systems engineering knowledge).

As it can be seen in Figure 3.1, a novel top-down analysis technique is introduced, which can systematically identify all possible source of a system failure and the error propagation. Thus, we decompose step-by-step the system to interconnected component behavior models on system-, safety- and hardware-level. The hardware-level model defines the failure model of the hardware components. The safety-level model describes the behavior of the safety mechanisms, and the system-level model specifies the operating states of the system and how the lower-level models influence its state transitions.

Moreover, with the simulation-based analysis, we can analyze the system model with deep probabilistic programming. This simulation method reveals the failure rate of the system and the significant failure contributors. Moreover, as we can see in Figure 3.1, the simulation also supports the system design, with simulation, we can analyze the system behavior in details revealing both the hidden weaknesses in the safety concept and also the dominant contributors of a system failure.

Finally, as can be seen in Figure 3.1, we validate the failure probabilities with an upper estimate provided by the Finite Fault Sequence Analysis (FFSA). FFSA explores the state-space of the deterministic system models. Then with the help of stochastic system models, FFSA calculates an upper estimate of the probabilities of the failure states. However, the accuracy of the FFSA is limited, but the results and the partial results can be verified manually.

3.0.1 Activities of the analysis

Modeling During the top-down analysis, we examine every part of the system, searching for all possible sources of failures, which can result in the interruption on any safety-critical-

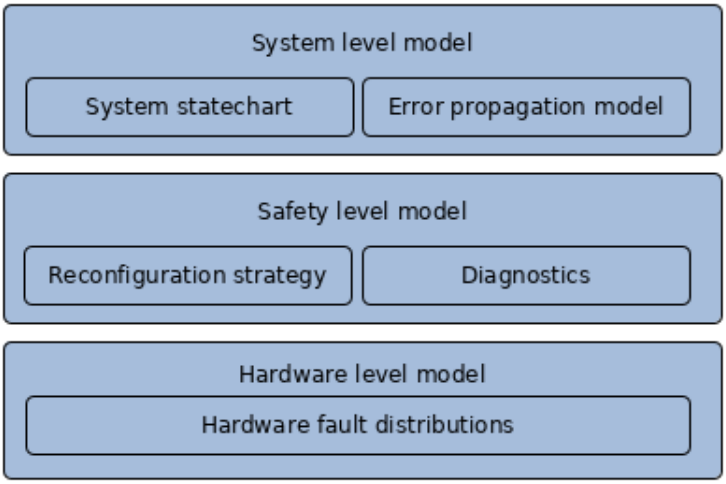


Figure 3.3. Structure of the system model

function (SCF). Thus we have to decompose the system step-by-step consistently with the development subphases to obtain its behavior. As one can see in Figure 3.3 it results three (system - safety - hardware) level model . At every step, we examine a lower level of the system than before.

On the top, there is the system level, which defines both all SCFs, what components can influence a SCF. Thus this level includes also the system configuration. Below is the safety level, which contains the safety function models of all critical components, their reconfiguration strategies, and their failure model. These strategies make decisions at every perceived hardware failure, consequently, includes also the diagnostic functions. They describe what kind of hardware failures can be detected.

Finally, the basis of the system-model is the hardware level, which contains all possible operational- and failure-modes of the hardware components and the stochastic transitions between them. The AADL error annex allows stochastic behavior everywhere, and this type of modeling makes the analysis complicated. For this, we separated them from the statechart-based models. As a result, the hardware model consists of a failure mode statechart and a failure-to-transition map, which specify the connection between the failure distributions and the state transitions of the statechart-based hardware model. The distributions are defined with *Pyro stochastic functions* in the analysis environment.

Analysis We modeled all deterministic behavior with the Gamma framework with interconnected statecharts, since Gamma provides all the required modeling and analysis functionality with a clear and easy-to-use graphical interface. We extended Gamma with the stochastic distributions, which allows us to do a detailed analysis of the system model with probabilistic simulation, as it can be seen in Figure 3.2.

Thus we translate the deterministic model and the fault distribution into the input of a *Stochastic runtime*. The Gamma framework can directly generate Java source-code from the statechart-based models. Then these codes are placed in the *Probabiistic Runtime Environment* so the system can be analyzed with a *probabilistic program*. Besides simulation, for conditional analysis, we can run several inference algorithms to get a full picture of the behavior of the system, including all availability and reliability measures.

Thus we developed several analysis techniques, which support the design process in every phase of the development. In the concept phase, we can quickly validate the new ideas because of the flexibility of the high-level modeling languages. In this phase, it is advised to use a simplified component model. Later, in the software and development phase, we can replace simplified models with the actual component models. Moreover, we can analyze the components separately with the help of the modular analysis. Finally, to improve the system, we can identify the failure contributors with the Pareto analysis and with the conditional analysis.

In addition, we also created an upper bound verification technique, which results in a conservative approximation (upper bound for the failure rates and probabilities) of the simulation results. The most significant problem with the probabilistic programming methods is verifiability and accuracy. However, the output of the probabilistic algorithms converges to the accurate value with the number of simulations, the speed of the convergence is unknown. Additionally, probabilistic programming tools can contain bugs and hidden software errors, which can corrupt the analysis results. Therefore we created a verification method, which can approximate the failure probabilities and rates with a given uncertainty. As a result, we can both estimate the speed of convergence and verify the results of the simulation

This verification method is based on the fact that the failure rates of the hardware components are really small (usually less than 200 FIT) [36]. Therefore the probability that more than 4 or 5 hardware components fail during the lifetime is minimal. As a result, if we collect each at most 5 long sequence[1] of hardware faults which causes safety goal violation and we sum their probability we will get the FIT numbers for all possible type of safety goal violation.[2] .

3.1 Steps of the analysis process

The process of our analysis method consists of five steps, as can be seen in Figure 3.4. This process contains redundancy at every step and is supported by high-level automatism to maximize the reliability of the results and minimize the chance that a design error remains undetected.

In the first step, we have to apply our top-down modeling technique. Thus we have to decompose the system in 3 steps. Firstly, we model the system-level behavior and error propagation. Then we model the safety mechanisms, including both the reconfiguration strategies and the sensor fusion algorithms. Finally, we create a failure model of hardware components.

In the second step, we verify the created system model. Our solution use high-level modeling languages such as statecharts. As a result, both the software and the hardware engineers can verify the correctness of the model.

In the third step, we calculate the required failure probabilities and failure rates with simulation. We use the *probabilistic* programming paradigm to make the computation efficient and accurate. Firstly, we generate from the system model a *Stochastic Runtime*. It generates random hardware faults and calculates when and how the system becomes faulty. After that, we map the objectives of the analysis into a probabilistic program (introduced

[1] The maximum length of the fault sequences is dependent on the system configuration, the lifetime, the number of components and their failure rates.

[2] During this calculation, we assume that the 6th fault will cause safety goal violation.

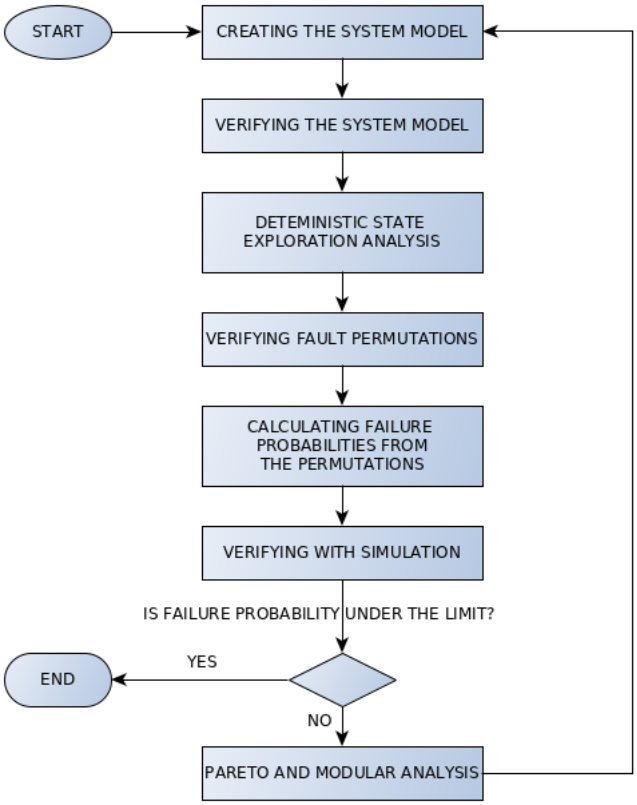


Figure 3.4. Flowchart of the analysis process

in Section 2.5), which contains the properties and the conditions of the analysis. Finally, we use the *Probabilistic Inference* algorithm to calculate the failure probabilities and rate.

In the fourth step, we verify the result of the simulation with the help of finite fault sequence analysis. In this method, we explore the state space of the deterministic model. First, we do deterministic state exploration in the deterministic system model. Thus we generate all significant permutation of faults. After that, we generate the Deterministic Fault Sequence Evaluator (DFSE) with the help of the *Gamma* framework. The DFSE determines what fault permutations causes safety goal violation. After that, we calculate the probability that a fault permutation occurs during the mission time, for all fault permutation, and we sum these probabilities. If the resulted failure probabilities are under the required limits, the analysis ends.

In the fifth step, if any of the failure probabilities are above the limits, we identify the hidden weaknesses of the safety concept and the dominant failure contributors. We can investigate the effects of hardware components with *Pareto analysis*, and examine the effects of the safety mechanisms with the modular analysis. These analyses use probabilistic simulation, as well as the reliability calculations in the sixth step. All these analysis methods are located in the *Probabilistic Runtime Environment*.

3.2 ISO 26262 safety analysis requirements compatibility

The safety analysis of critical components has to be approved by several industry-specific standards. Therefore we created our analysis method following the ISO 26262-2018 stan-

dard as it is one of the most modern and relevant standards. Thus in this chapter, we present the connection between our method and the ISO 26262. In Table 3.1, we show the connection between our analysis method and the safety analysis requirements of the ISO 26262 standard.

3.3 Case study

We applied our top-down safety analysis method on a real-life example from the automotive industry, on an electronic control unit (*ECU*) of an electronic power-assisted steering (*EPAS*) [12]. Due to its critical role in the car, it has several safety functions including adaptive reconfiguration and high redundancy in the hardware. Besides we have to calculate all probabilistic measures required by the ISO26262 standard.

The EPAS has to provide a safety critical function, the steering actuation, which has to meet several availability and reliability requirements specified by the ISO26262 and the customer. The availability requirements specify the probability of a failure in the $50000h$ long mission time must be lower than $2 \cdot 10^{-6}$.

The simplified structural model of a typical, widely-used *EPAS ECU*, can be found in Figure 3.5a. It has two microcontrollers (uC), that are completely separated. The uC-s have three-three redundant sensors, each of which can be modeled with an operational mode and with two failure modes namely, *Shutdown*, and *Drift*. In shutdown mode, the sensor stops working, which can be detected every time. Beside that, the *drift* mode represents a latent failure mode: in this state the sensor seems to be working correctly, yet it provides a erroneous output. Therefore, the detection of this failure mode requires redundant sensors and a voting mechanism. With the help of the sensors, the uC-s provide the steering-actuation functionality (a safety-critical function) that must operate during the mission-time of the car continuously.

Figure 3.5b[3] depicts the simplified statechart describing the behavior of a uC. The initial state is *Normal operation*. The model goes to state *AssistLoss* if the uC fails or all sensors fail (go to state *Shutdown*). In contrast, the uC goes to state *LatentError* if at least two sensors have latent error as the bad sensors will vote down the good one. Therefore, the uC will use wrong input data in the control loop causing the EPAS (the states of which are modeled in an evaluation statechart) to go to state *Uncontrolled self-steering*. The EPAS system can fail in two ways:

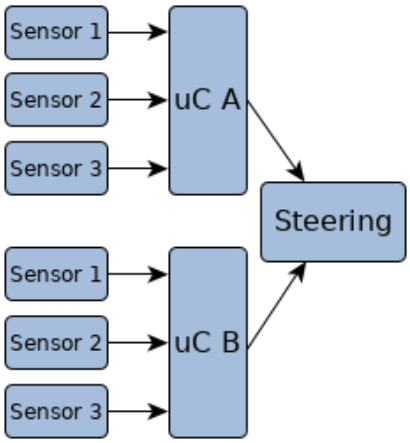
- If both uCs go to state *AssistLoss*, the steering assistance will also stop operating, resulting in a troublesome steering experience. This state of the *EPAS* is called *Loss of assist* (LoA).
- If a uC goes to state *LatentError*, the whole EPAS will go to state *Uncontrolled self-steering* (SS). This situation is extremely dangerous and must be prevented by any means, as the driver loses control over the car.

[3]Variables in the statechart:

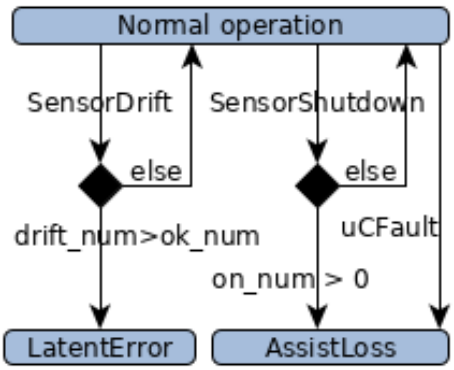
- drift num : number of drifted sensors
- ok num : number of normal sensors
- on num : number of sensors that seem to be working

N.o.	Connection	Section
8.3.1	Our analysis technique supports the development in the full life-cycle. With the help of the modular analysis we can analysis the components separately. Thus we can try out new ideas even the system model is incomplete.	5.1.6
8.3.2	We can include any kind of external hardware behavior in the system model, because of the flexibility of Pyro	4.4
8.4.2	With our analysis methods we can easily determine, whether the requirements are complied or not. Moreover with FFTA we can verify the results.	5.1.4, 5.2
8.4.3	If the safety requirements are not complied, the Pareto analysis technique and conditional analysis identifies the dominant failure contributors. Moreover with modular analysis diagnostic we support the design process to increase the detection.	5.1.6, 5.1.5, 5.1.7
8.4.4	The use of our analysis method do not require any special competences, thus the safety analysis can be an integrated part of the development.	4.1, 5.1.6
8.4.5	The newly identified hazards can be easily implemented in the system model, because of the modular system model and the easy-to-use graphical interface provided by the Gamma.	4.1
8.4.6	The component models in the system model can be analysed individually in the development subphases. Because statechart-based component models can reviewed by directly the engineers and the engineers by themselves are able to create component models without any training.	5.1.6 and 4.1
8.4.7	We can determining additional test cases because the FFSA can list those fault combination which are the most dangerous.	5.2
8.4.10	Our qualitative analysis algorithms can calculate the all of the most common hardware architectural metrics such as failure rates and probabilities. Furthermore, with conditional analysis and FFSA we can investigate the relation between the hardware faults and the safety goal violations. With modular and Pareto analysis we can discover the weaknesses of the system design and safety concepts.	5.2, 5.1.6, 5.1.7

Table 3.1. Comparison of our analysis approach and the ISO 26262 requirements



(a) Structure of the EPAS



(b) Behavior statechart of the EPAS

Figure 3.5. EPAS case study

Chapter 4

Modelling methodology

Traditional modeling techniques like the use FTA for adaptive cyber-physical systems might be long and inefficient [10]. Because FTA can not model degradation modes and the error propagation is complex because of the reconfiguration. Since these methods model the system behavior on a low level, consequently, the complex behavior results and unclean models.

In this chapter, we present the modeling technique of our top-down modeling technique, which can deal with the complexity of the modern adaptive cyber-physical systems. Firstly, we define the main concepts and goals, and then we show the steps of our modeling method on the case-study. Finally, we present our modeling technique on the case-study.

In our analysis method the analysis of the software is excluded, because the software failures are always systematic. Because hardware-level is only this part of the system model which has stochastic behavior, and in safety-critical systems the software is always completely deterministic. (For example in critical systems the use of dynamic memory handling and garbage collection is prohibited.) Thus we consider the software reliable since the errors can not come from them. The errors can only go through these components (for example, a motor control function always provides appropriate output until the incoming measurements are valid). The exclusion of software failures is also required by several standards like ISO 26262.

Consequently, we include only those parts of the software which are directly related to the hardware fault. We call software components the *safety mechanisms* like diagnostics and the degradation controllers. Furthermore the safety mechanisms can also be implemented only in the hardware without any software because the safety mechanisms represent the system level adaptivity of the hardware components. As a result, these mechanisms are always located in the hardware components, which means that if the containing hardware component is faulty, then the safety mechanism also stops.

The malfunctions and errors are always originated from the random hardware faults, never from the safety components. Thus we have to create a system configuration model inspired by the AADL Error annex. The configuration determines both the communication and the error propagation between the safety functions and the hardware components. We used SysML-like statecharts based to model the configuration and the safety mechanisms, as the statecharts provides a clear, easy-to-use high-level modeling formalism that can define even complex behaviors.

We use the Gamma framework to model both the safety mechanisms, hardware states and the error propagation, since Gamma is compatible with the traditional modeling methods

and it makes the model clean and modular. We can define in Gamma complex component behavior with statecharts and connect them with channels. These channels specify both the error propagation and the communication between the components. Moreover we extended the functionality of Gamma with stochastic distribution to model the uncertainty of random hardware failure.

4.1 Steps of the modelling

In this section we show how to execute our top-down modeling method step-by-step until we get all atomic failure source which can be modeled by a stochastic distribution. The steps of the modeling are depicted in Figure 4.1 with a simple example model.

The first step is to determine the a system level model, which contains the evaluation statechart and the configuration of the system-under-analysis. The evaluation statechart describes behavior of the system from a top level perspective. Therefore this statechart contains the operational states and the failure states of the system. When we create statechart we have to distinguish every state of the system where the quality of the SCF are different.

Moreover the state transitions are influenced by the random hardware faults and the fault propagation. As a result the top-level model contains a fault propagation map. This map connects with channels all subcomponent of the system, which can effect the SCF. These component can be hardware models as well as safety mechanisms.

Modern critical systems include several safety measures including adaptation and reconfiguration strategies. They increase the reliability and the availability of the SCFs by the parallel execution of reliability optimisation strategies at every significant hardware component. Thus we have to identify them during the safety level modeling and formalise these strategies with statecharts . [1] These controllers are supported by diagnostics and sensor fusion algorithms, which help the controllers by signaling hardware faults. Their statechart-based model determines how the controller observes the states of the hardware components.

Finally the last step of our analysis is the identification of the state-based failure model of the relevant hardware components. The documentation of the hardware components provides all necessary information. Because in case of critical systems the datasheet and the safety manual must include both all failure modes and the failure rate of the hardware component. Furthermore we do not model the recovery of the components since the critical electronics do not contain any self-repairing mechanisms. We know that in some special circumstances it is possible that a previously failed component suddenly works again, but its probability is extremely low and about this event no data are available.

4.2 System level

The system-level model is obtained from the high-level functional safety requirements defined by the customers and the industrial standards. These requirements specify all safety-critical functions and define the continuity and the readiness of the SCF-s. Moreover, during the analysis, we have to decompose it for the hardware components.

[1] If any safety component can not be formalised with statecharts it can be replaced by its source code.

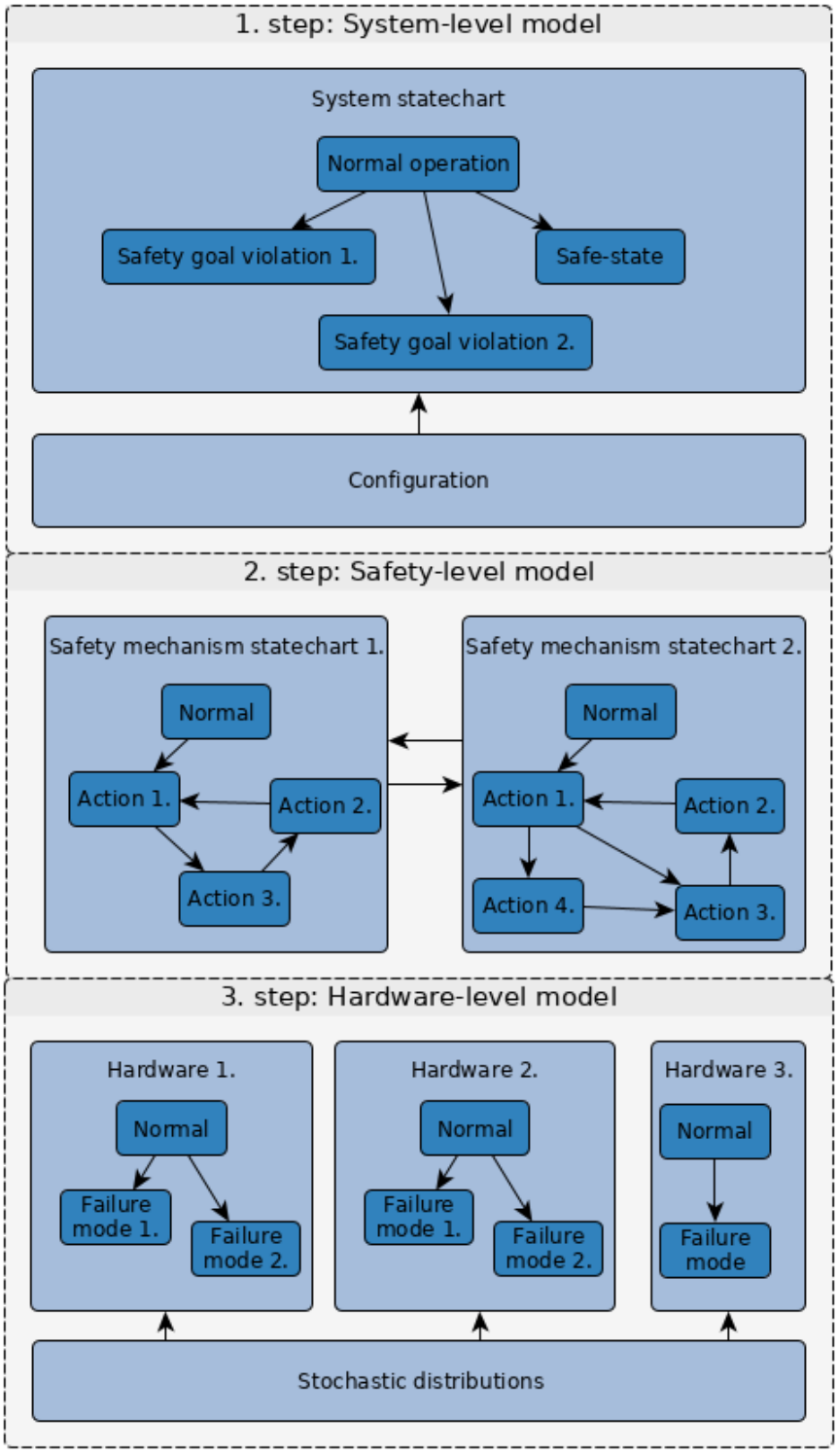


Figure 4.1. The steps of modeling on a simple example

Therefore we divided the system-level model into an evaluation model and a system configuration. The evaluation model has to contain every different state and action which can influence any SCF. Hence the output of this model is a port containing all possible failure modes and safe states of the system. Furthermore, this system evaluation model is formalized with a statechart and determines the reliability of the system by observing the state of all critical components providing SCF.

All components providing SCF are capable of influencing directly or indirectly the be the state transitions of the evaluation statechart. Thus the top-level model also includes the system configuration model, which defines the structure of the components, the communication, and the fault propagation between them. This model uses channels to connect the elements of the lower levels with directed event-based interfaces.

4.3 Safety mechanisms

The hardware components contain several builtin safety functions, including self-diagnostic and reconfiguration strategies. We call these functions *safety mechanisms*. If they perceive their fault, they can bring the component into a safe state to prevent any safety goal violation. In the case of adaptive systems, this safety action causes a reconfiguration in the system when other redundant components take the role of the failed ones. Therefore the system remains operational even when faulty.

These safety mechanisms specify the high-level, safety-related functionality of the components, and they are always located in a hardware component. Consequently, these mechanisms are dependent on the operation of the containing hardware. Thus if it is faulty, then its all safety function stops immediately. There are two main types of these safety mechanisms, namely, the safety controller and the diagnostics. The safety controllers specify reconfiguration strategies, and the diagnostics specify how the system perceives the hardware faults.

We model the safety mechanisms with the statechart language introduced in the AADL Error Model Annex since it provides a variety of tools to describe complex decision making. Additionally, the statechart s of the safety mechanisms consists of two orthogonal regions, namely, the control and decision and the operation region. The former defines how the component will react to the different hardware faults and situations. Furthermore, the control models are also able to communicate with each other to coordinate resource management and redistribute the SCFs.

The latter region defines the correctness of the operation. Thus the operation region contains the failure model of the safety mechanism. Thus this failure model should include some operational and some failure modes of the safety mechanism. The state transitions of the failure model are determined by the hardware models and by the safety mechanisms.

4.3.1 Diagnostics

The diagnostics are a significant part of the safety mechanisms because all industrial standards e.g. ISO26262 require them. Diagnostic functions can detect the errors of the hardware components in the system and alarm the safety controllers. In many cases, the detection of the erroneous operation is a challenging task. As a result, diagnostics usually use sensor fusion algorithms to detect the malfunction of redundant hardware components.

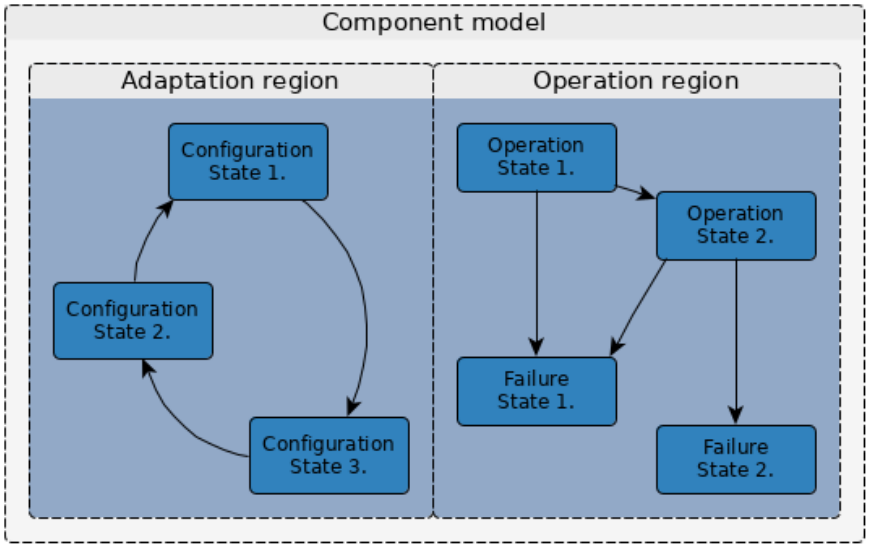


Figure 4.2. Adaptation and operation of the components

Several industrial standards like ISO26262 require the use of coverage for the diagnostics. Thus if a component fails, which is observed by a diagnostic function, it will detect the failure with a given *coverage* probability. Additionally, the standards require an upper limit for the coverage: in most cases, the coverage can not be higher than 99%. As a result, we must model the software components with stochastic behavior. Therefore the use of diagnostic coverage is contradictory to the modeling concept that all software models have to be deterministic.

To resolve this contradiction, we introduce the concept of a virtual component, which is responsible for the malfunction of the diagnostics. Thus we move all the stochastic behavior of the safety mechanism into the hardware-level model. The virtual component has two states *detected* and *undetected* and when the system is created it goes to state *detected* with the coverage probability and it goes to state *undetected* with one minus the coverage probability. Thus if the virtual component in the state *detected*, the diagnostic function operates normally. In contrast, if the virtual component is in state *undetected*, then some functions of the diagnostics are disabled at all. Consequently, the diagnostic function can not signal the failures. Beside the virtual component, we present another method, the *coverage transformation* to model coverage in Section 4.4. This method is more simple than the virtual components but the *coverage transformation* can not be applied in all circumstances.

We can see a simple example in Figure 4.3. We can see that the failure of the virtual component triggers (via a channel) a state transition in the *operation*. This transition bring the system into the *Coverage failure state* where the diagnostic is disabled. One can see that the virtual failure component has probabilities instead of failure distributions, because this coverage failure transition is either occur at the beginning of the simulation (with *1-coverage* probability) or this failure do not occur at all (with *coverage* probability).

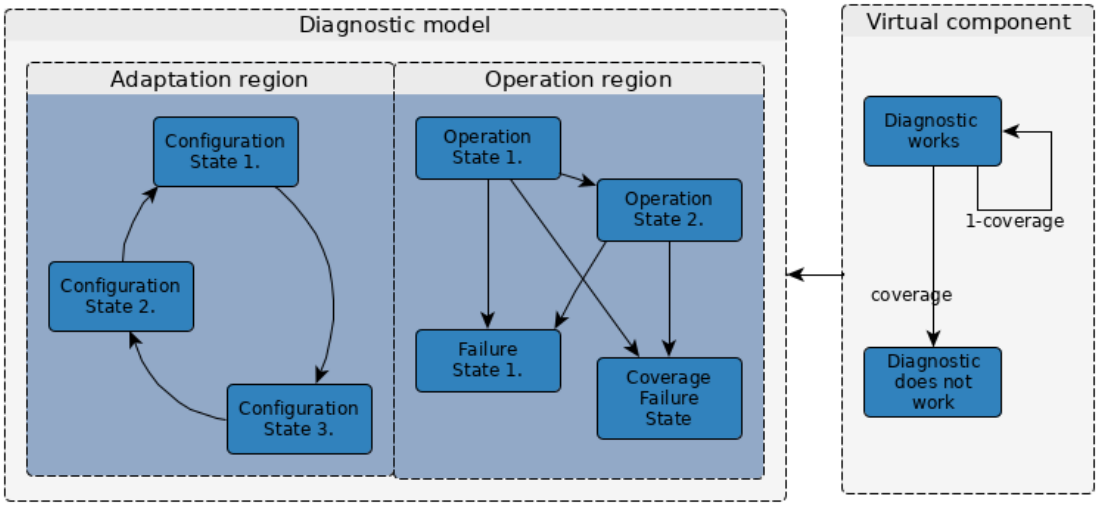


Figure 4.3. Virtual component on a simple example

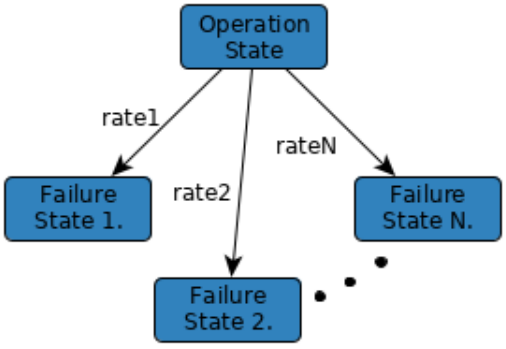


Figure 4.4. Example statechart of a state based hardware component

4.4 Hardware-level model

Our approach introduces a fault-modeling technique based on high-level state-based i.e., statechart models. Statecharts provide an expressive language to capture even complex error propagation scenarios of systems.

To create a state-based failure model for the hardware components, we have to identify the operational, the failure states, and also the transitions between them. We can get this information from the supplier, or the standards [36, 1] specify the operational and failure modes. As a result, we can create statechart from this data by creating a failure state for every failure mode, but complex high-functionality hardware components contain several different failure modes.

We can simplify the statechart by grouping the failure modes based on their effects and their observability. Therefore we should merge those states which have the same effect on the system and are observed equally by all diagnostic. (If no diagnostics can observe some failure states, then these states will be equally observed by each diagnostic.)

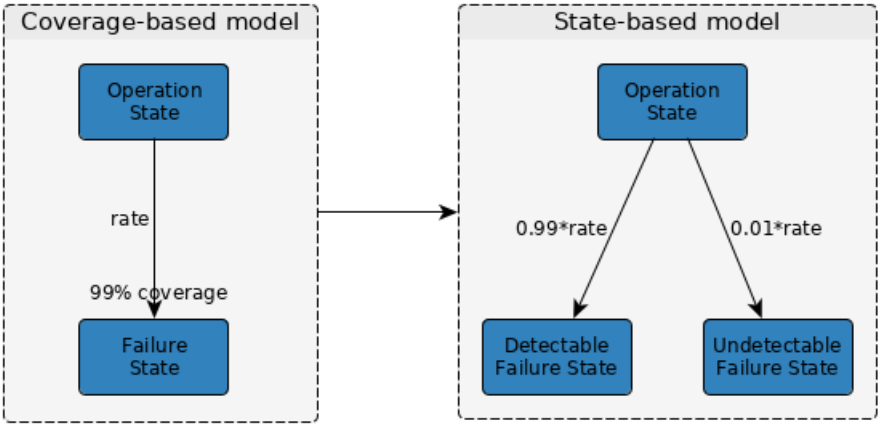


Figure 4.5. Dividing a failure state into two according to the diagnostic coverage

4.4.1 Coverage transformation

At critical electronic systems the software is always deterministic and only the hardware malfunctions can have stochastic behavior, since all software fault are sistematic. In certain cases the software has errors which occur apparently at random times even tough this software malfunction is caused by a spacial input combination from either its hardware or its environment.

As a result, we introduced the virtual component for diagnostics in the previous section. This component can be modeled by a virtual hardware element which sometimes goes to a failure state causing the malfunction of the software. Therefore in the analysis, we can put every probabilistic behavior in the hardware model as it makes the modeling fast and clean. Yet the use of the virtual component can sometimes overly increase the size of the hardware-level model.

Therefore we introduce the *coverage transformation* technique which put the diagnostic coverage into the state space of the hardware statechart. It means, if a hardware component have a failure state which is detected with 99% we will divide this state into two states. As it can be seen, one is completely detectable by the diagnostics and the other one is completely undetectable for the diagnostics.

This transformation greatly effects the stochastic model of the state transitions. If the distribution of the failure state is exponential with a given rate parameter, then the new failure states have exponential distribution too with the 99% and a 1% of the rate. Obviously it can not be used for some special sensor fusion algorithm then the method of virtual component has to be applied.

4.5 Stochastic behavior model

In dependability analysis it is not enough to model the functions and services of the system, but we also have to model extra-functional aspects such as failure occurrence distributions and environmental effects. Various distributions are used to model the random hardware faults. Our modelling approach supports the application of arbitrary distributions, including the definition of joint distributions too.

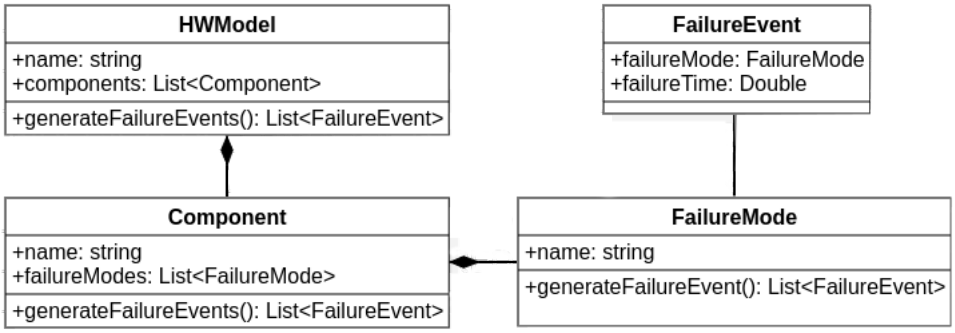


Figure 4.6. Class diagram of the failure/environmental distributions

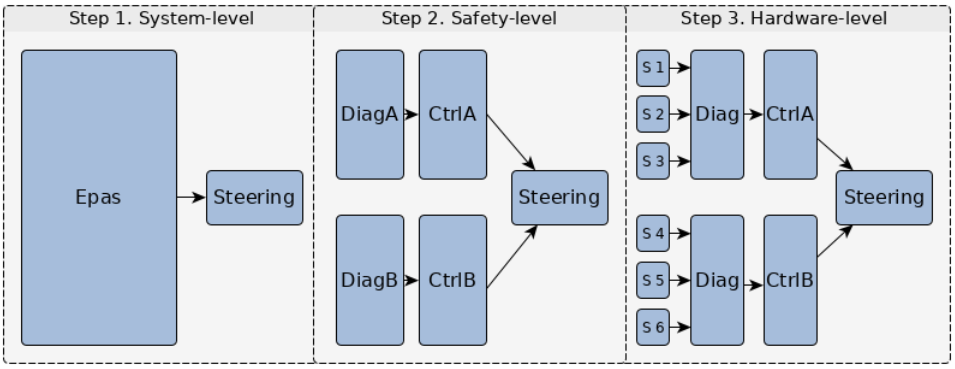


Figure 4.7. Levels of the top-down analysis

They can be the results of some previous analysis or measurements of the hardware components. For example in manufacturer provides the rate parameter with a variance of uncertainty of the exponential distribution of a specific failure mode.

These distributions are mapped to the state transitions of the state statecharts of the hardware elements as these stochastic models define when the hardware component should go into a failure state. If a component has several different failure modes, according to its statechart-based model, always the first failure mode will dominate, and the component will stay there forever.

4.6 Modeling the case-study

In this section we present how to apply the aforementioned modeling technique to the EPAS model in the case-study. Therefore we executed our modeling method step-by-step as it can be seen in Figure 4.7.

First of all, we have to create a system level model including both the (*loss of assist* and *self-steering*) failure states and the operational states. These states are effected by the several safety mechanism in the uCs of the EPAS. Consequently we have to create a controller level representation which defines the adaptation strategy namely, how the uC's software reacts to the perceived (uC and sensor) hardware failures. Additionally, this layer

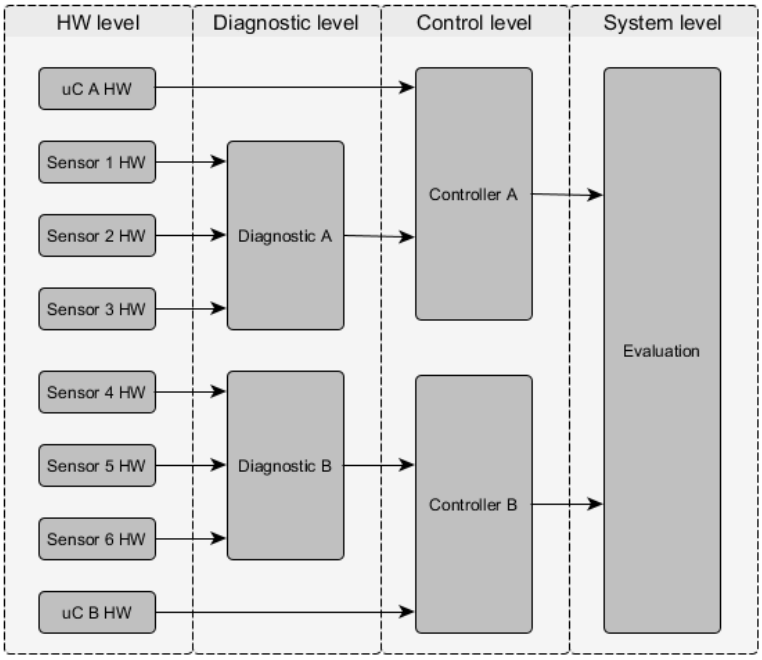


Figure 4.8. Layered structure of the EPAS model

has to specify every effect of the uC's controller on the steering actuation namely, when a uC stops working and when a uC provides erroneous steering actuation.

The controller makes their decisions based on the sensor diagnostics, which defines the voting mechanism connected the torque sensors. The model of these sensors and the uC take place in the hardware layer which define the operational and the failure state of every hardware components and the stochastic behavior of the faults.

The detailed structure of the resulted analyzis is depicted in figure 4.8. One can see that the sensor faults can be detected by the diagnostics consequently, some one latent sensor failure can be canceled at each side. In contrast the failure of the uC effect directly the controllers, as its permanent damage stops all software funtions in the uC. Therefore in the system level evaluation model the states of the two uCs determine the state of the *EPAS*.

4.6.1 System level

The evaluation statechart of the case-study EPAS is depicted in figure 4.9. One can see that the state of EPAS is effected by the controllers of the uC A and B as they providing steering assist which is the SCF of the system. Thus the evaluation statechart have two operational states, *Normal* and *Warning*. In the former state there is no perceived fault in the EPAS. In contrast the system goes into warning state if any controller notify the user about a detected a failure. Besides the EPAS has also two failure sates, *Self-steering* and *Loss of assist*. In these states the EPAS can not provide the steering actuation (which is a SCF) correctly. Consequently, when the system goes into a failure state a safety goal violation event is generated immediately to the environment. Thereafter no other event will occur as these events end the operation of the system.

The configuration of the system is depicted in figure 4.8. It contains the hardware failure model of the sensors and the uC. The sensors are connected to the diagnostics which

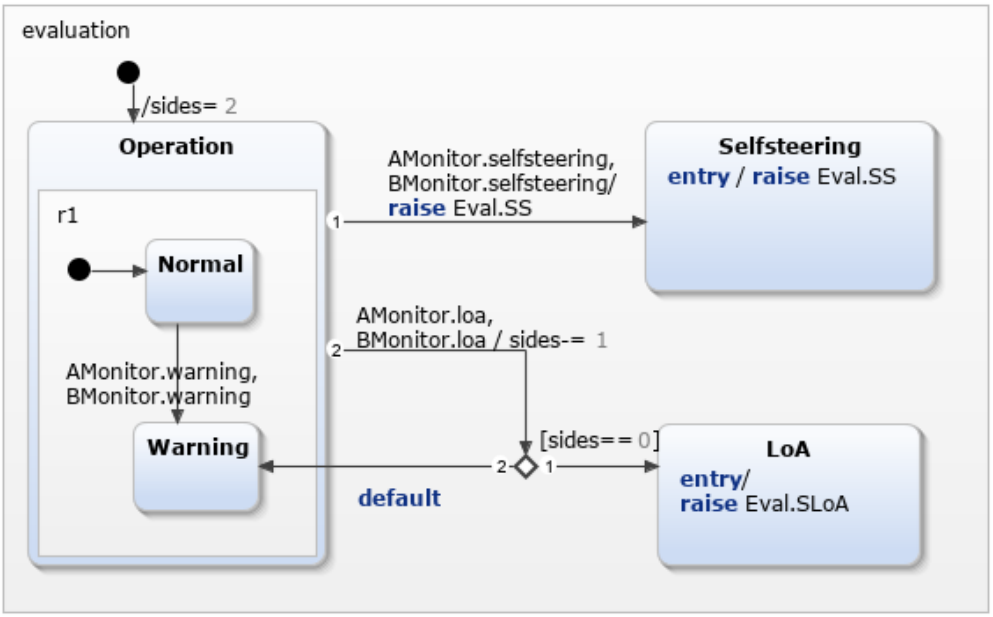


Figure 4.9. Evaluation system level statechart of the EPAS

provide a warning channel about the detected faults and an output correctness channel to a controller.

4.6.2 Safety functions

The EPAS has two type of safety function included in both uCs, a torque sensor diagnostics and a motor controller. The diagnostics (depicted in figure 4.10) are able to cancel a latent failure of the sensors with a voting mechanism. It provides the following functionalities:

- it reads in the data from three sensors,
- it sends warning if a sensor stops working,
- it choose one sensor data with the voting mechanism and it sends towards to the motor control.

One can see that the diagnostics contain an adaptation and an operation region. The adaptation region can notify other with a *Warning* event other safety functions if one of the sensors stops working and it alarm the functions with an *Error* event if all sensor stopped. The operation region define that the voting mechanism provides erroneous output more sensor has latent failure (it seem to be working but it provides bad output) than the good sensors. This controller have outgoing interface, a status and an output. The status interface provides information about the quality of the outgoing sensor data and the output interface provides information about the correctness of the outgoing data.

The motor controller use the output of the diagnostics to provide steering actuation with a closed control loop. It behavior is depicted in figure 4.11. One can see that it also this statechart has an adaptation and an operation region. The adaptation controller send a warning signal to the car if the diagnostic sends *Warning* status event. If the diagnostics sends *Error* status event the controller turn the uC off since without sensor data it can not provide actuation anymore. Additionally, the motor control stops working if the uC

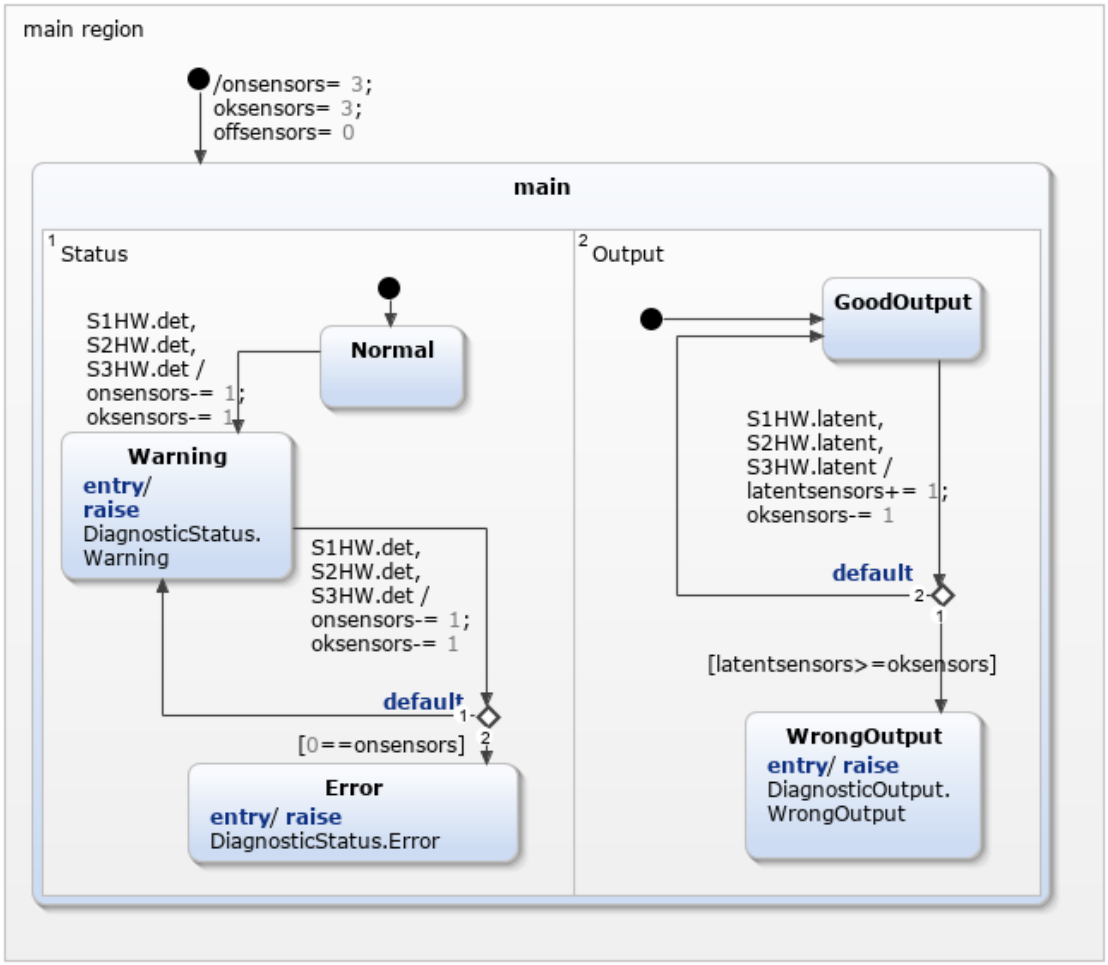


Figure 4.10. Statechart of the torque sensor diagnostics modeled by Gamma

is faulty and the control provides erroneous steering and has latent error if the diagnostic provides wrong output without *Error* status event.

4.6.3 Hardware level

Since most component failure models have an initial operational state and some failure modes and after any failure event was raised, no other event will occur in the failure model. For example the uCs in the case-study have only two states an operational and a failure one namely, *On* and *Off* as can be seen in Figure 4.13. In state *On* the uC controller operates normally, in contrast in state *Off* the uC stops working completely due to some permanent damage. Obviously the uC contains a builtin watchdog canceling the effects of software malfunctions but the uC recovers from this faults in milliseconds with 100% success ratio. As a result it should not be included in the top-down analyzis.

Furthermore the torque sensors of the *EPAS* have an additional failure state besides state *Off* namely the *LatentFailure*. In the former state the sensors stops providing output consequently it can be detected every time. In contrast in the latter failure state the sensor seems to be working, but its provides erroneous output. Its detection is much more difficult than revealing the *Off* failure state. To discover the *LatentFailure* of a

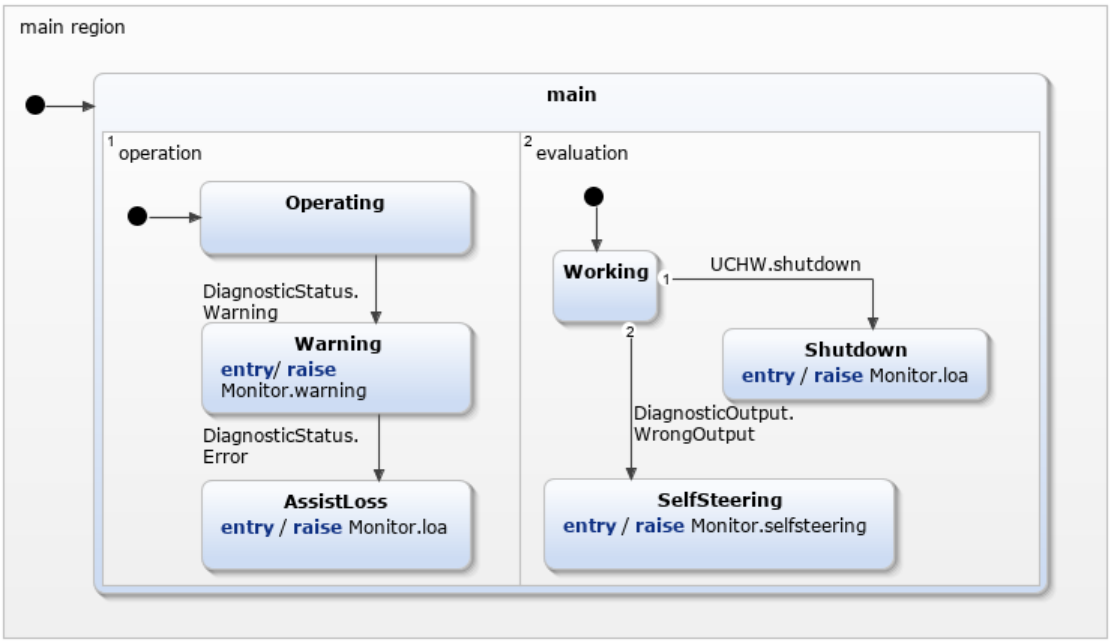


Figure 4.11. Statechart of the motor controller modeled by Gamma

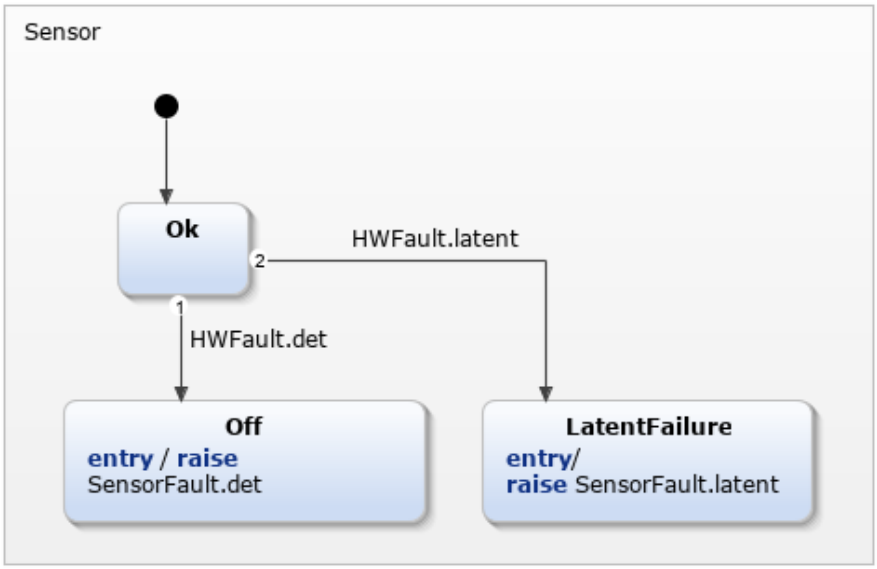


Figure 4.12. Statechart based failure model of the sensor modeled by Gamma

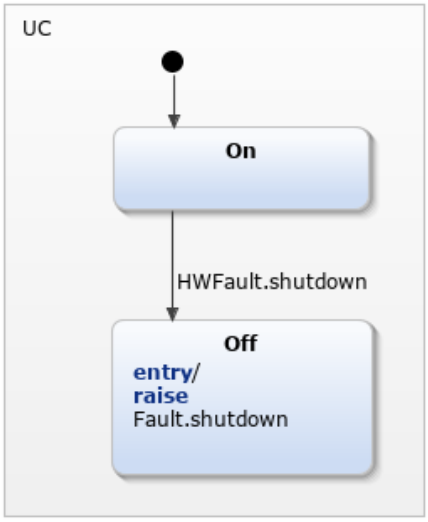


Figure 4.13. Statechart based failure model of the uC sensor modeled by Gamma

Failure mode	Statechart of transition	from state	to state
uC.off	uC	On	Off
sensor.off	Sensor	Ok	Off
sensor.latent	Sensor	Ok	LatentFailure

Table 4.1. Connection between the distributions and the state transitions in the case-study

torque sensor we need several (in case of the EPAS three) redundant sensor and a voting mechanism. It can provide appropriate output until the number of good sensors is greater than the number of bad sensors.

Owing to the huge number of components and their various failure modes it is necessary to use a hierarchic modeling structure. The class-diagram of the stochastic behavior model can be seen in Figure 4.6. The *HWModel* is the top-level class that contains several *Component* object. They contains several *FailureMode* objects, which define their stochastic behaviors over a lifetime of the system. Due to their complexity the failure distribution model has to be defined in the PRE.

In the case-study the *HWModel* contains all together eight components, two uCs and six sensors. The uCs have only one failuremode[2], *off* which occurs with Weibull distribution with scale parameter 0.1 $10^9 h$ and with shape parameter 1.5. Besides the sensors have two failure modes, *off* and *latent*. They have both exponential distributions with rate in order 10 and 1 FIT. These failure modes trigger their associated state transition in the state based fault model (defined in Section 4.4). The association table (Table 4.1) shows the connection between the failure mode and the state transition in the state based fault model.

[2]The failure models of the case study were greatly simplified because of clarity.

Chapter 5

Safety analysis

In this chapter, we introduce our analysis algorithms. In the first section, we present our simulation approach with deep probabilistic programming. These algorithms can verify the safety requirements as well as support the design process in every development phase. After that, in Section 5.2, we present a Markovian verification approach that checks the compilation of the safety requirements.

5.1 Analysis with deep probabilistic programming

Traditional analysis techniques have only limited capabilities to support the design process, even though at design time, safety engineers have to provide immediate feedback to system designers. We created a simulation-based analysis technique that can identify the primary sources of the system failure in addition to the calculation of the failure probabilities. Our technique is based on *deep probabilistic programming*. Thus our approach supports the decision making even in the concept phase of the development when the available information about the system is insufficient for traditional analysis techniques. Since, with the help of our probabilistic analysis, we can validate the new design ideas and safety concepts directly from the system model. Since our analysis method does not require any additional manual system-to-analysis processing as our solution process, the system models automatically.

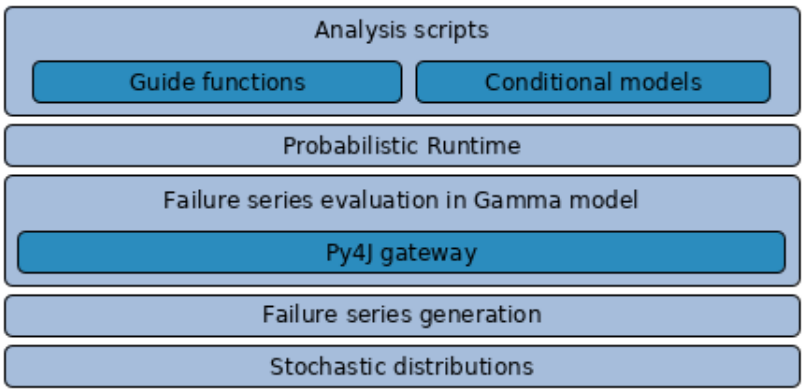


Figure 5.1. Structure of the *Probabilistic Runtime Environment*

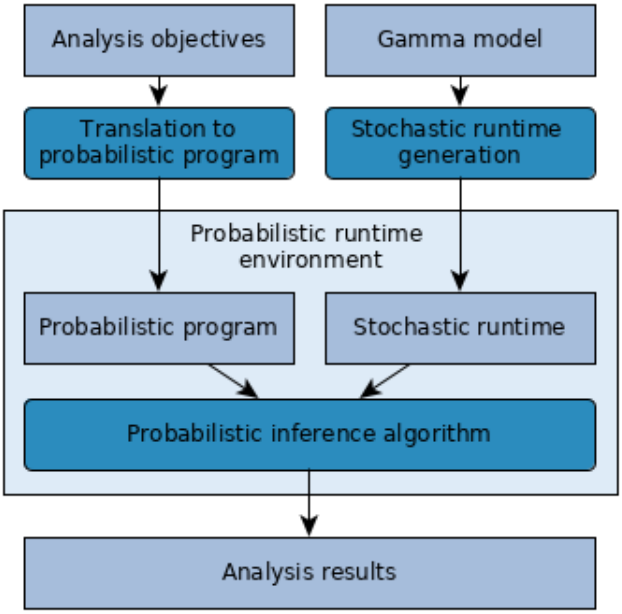


Figure 5.2. Analysis with deep probabilistic programming

During the analysis, we have to investigate several different aspects of the behavior of the system. Thus we created the *Probabilistic Runtime Environment* (PRE), which provides a unified, integrated platform for stochastic analysis. The *PRE* has a layered architecture as can be seen in Figure 5.1. The bottom layer process the failure distributions of the hardware components. This layer generates random hardware failures and puts it into a *fault series*, which can be evaluated by the deterministic Gamma model. On top of these layers is the *Probabilistic Runtime*, which is the complete stochastic model of the system. The interface of the *Stochastic Runtime* is a simulation function. It can be called from the *analysis scipts*, which are simple *probabilistic programs*. These scripts contain all the objectives and the boundary conditions of the system analysis.

The activities of the probabilistic analysis are depicted in Figure 5.2. The first step of the analysis is the *stochastic runtime generation* when we generate a Pyro stochastic function (introduced in Section 2.5) from the Gamma models. After that, we translate the objectives of the analysis into a *probabilistic program*. It contains both the conditions and the examined properties of the analysis. Both the probabilistic program and the *Stochastic Runtime* is located in the *PRE* from where we can run the inference algorithms of the *Pyro* which compute the analysis results directly.

We implemented this environment in Python to increase compatibility as the most popular probabilistic programming tools e.g. Stan [13], Pyro [8] and PyMC3 [39] have Python interfaces. Out of them, we built our implementation of the *PRE* on Pyro as it includes all state-of-the-art inference algorithms, and Pyro also provides access to cutting edge technologies like deep probabilistic programming and SVI algorithms. Additionally, it is based on the artificial intelligence framework PyTorch. Thus we can analyze the system even it uses neural networks and machine learning. Moreover, Pyro can use GPU acceleration for the analysis in the *PRE* due to the PyTorch framework.

5.1.1 Translation into probabilistic program

All deterministic descriptor is modeled in the Gamma framework out of which we can generate Java implementation automatically. Consequently, we have to put them into a Python-Java gateway module with a Py4J bridge [3], which can access the Java objects for the python analysis modules. This gateway also simplifies the use of the generated Java classes and provides an easy-to-use interface for the *PRE*. With this interface, the *PRE* can easily calculate the system-level result of the environmental events.

The analysis objectives have to be translated into a probabilistic program manually according to a simple schema presented in Section 5.1.4, 5.1.5, 5.1.6 and 5.1.7. Even though if a temporal logic expression [31] contains the objectives, then this expression can be easily mapped into a probabilistic program.

5.1.2 Fault series generation

Our simulation approach is based on continuous-time, asynchronous simulation. In order to make it transparent and straightforward, we simulate every probabilistic event in the system at once. Thus as it can be seen in Algorithm 5, we sample each distribution of each failure mode of each component, which results in the occurrence time of the failure modes. Thereafter, from the resulted samples, we generate (hardware) fault events with the *distribution-to-transition map* and collect the events into the fault set. Then we arrange the fault events in chronological order to get a fault series. Hence these events can be processed by the deterministic Gamma model.

```
faults ← ∅
for all component ∈ system do
  for all failure_mode ∈ component do
    fault ← sample(failure_mode.distribution)
    faults ← fault ∪ faults
  end for
end for
return  faults =0
```

Algorithm 1: Pseudo code of fault series generation

5.1.3 Fault series evaluation

The *PRE* can evaluate the result of a component failure series with a *while* cycle as it can be seen in Algorithm 1. This algorithm sends one-by-one each event in the fault series into the Python-Java gateway, queries the actual state of the deterministic model, and if we reach a failure state, the cycle stops, and we return with the failure state. All of these steps are packed into the *Probabilistic Runtime* function. This function is available for the analysis scripts with the *simulate* function from the *PRE*.

5.1.4 First-time-to-failure analysis

The lifetime prediction is an essential quantitative analysis for any critical system. For this calculation, we have to call several times the *simulate* function from the *PRE* and create a histogram from the simulated failure times, which will be an approximation of the lifetime distribution.

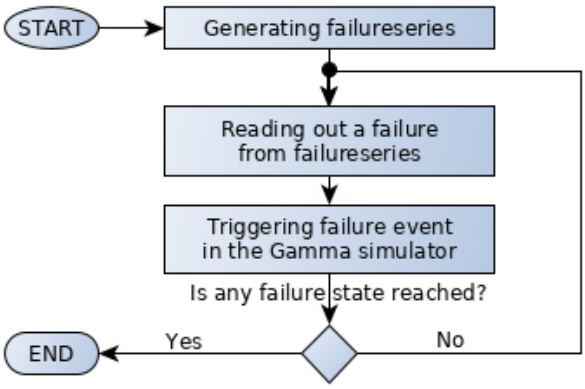


Figure 5.3. Evaluating the fault series during simulation

Owing to Pyro's builtin optimizer algorithms, we are also able to fit a distribution directly to our model. Thus our first task is to create a parametric *guide* function about the distribution we expect for the lifetime. In most cases, it will be a two-parameter Weibull distribution, yet the type of the expected distribution has to be defined manually by the user. Then, we put the *simulate* function and the *guide* into a SVI optimization algorithm, which will find the optimal Weibull shape and scale parameters revealing the fitted lifetime distribution. This method helps us to understand the behavior of the system as the shape parameter tells us how the failure rate of the system changes over time.

Moreover, this method is advantageous if we want to calculate small probabilities when traditional simulation methods are wasteful and inefficient. The most common case is when we want to calculate the probability that the system fails during the mission time. For example, the ISO26262 standard specifies failure probabilities under 10^6. As a result, the traditional analysis methods need several millions of simulations to provide results with acceptable accuracy.

For example, Figure 5.4 shows the probability density function of a simple Weibull distribution. We depicted with green color the probability that the system fails before the mission-time ends and with red color the probability that the system becomes faulty after the end of the mission-time. In contrast if we use the SVI algorithm and we can fit an appropriate distribution to the system model. As a result with the fitted distribution we can explicitly calculate the failure probabilities without any unnecessary computation.

5.1.5 Conditional analysis

During safety assessment, we have to examine complex questions, which include many aspects of the system at once, e.g., what will be the lifetime distribution if the system goes wrong via a specified way. Traditional statistical methods are usually really inefficient and slow in such cases as these types of failure combinations occur in a low percentage of the simulations. The solution to this problem is an inference algorithm introduced in section 2.5. To apply these inference algorithms we first have to create a *conditional model* from the *simulate* function in the *PRE*, which includes *observe* function for each conditioned random variable. The *conditional model* is a key part of the *probabilistic program* as it defines the boundary conditions and the objectives of the analysis. After we created the probabilistic program, we either put it into a Pyro inference algorithm and run

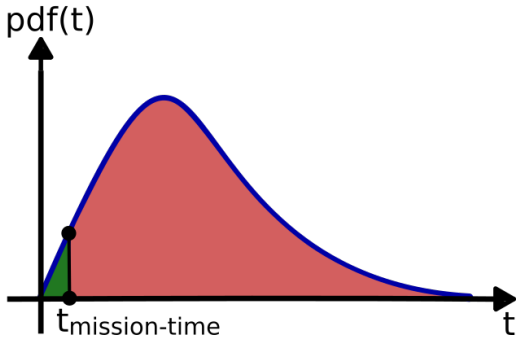


Figure 5.4. Example probability density function of a failure time distribution

Monte Carlo simulations, or we use the Pyro's model fitting (SVI) algorithm (introduced in the previous subsection) with an appropriate *guide* function.

Conditional analysis can answer several essential questions in design time. Firstly conditional analysis can be used for failure mode analysis. If in the *conditional model*, we put an *observe* statement to a specified failure mode, we can calculate the distribution of the failure time, assuming that the system fails with the specified way.

For example, if we want to know the distribution of the time when the system fails with a specific failure mode "*shutdown*," then we have to apply conditional analysis. We call this type of analysis, conditional time-to-failure analysis. Algorithm 2 shows the pseudo-code of this conditional analysis. Thus we create a conditional model from the *simulate* function in the PRE, where we *observe* the *endstate* of the system. In this step, we create a conditional model, where the endstate of the system is assumed to be *shutdown*. After that, we put this conditional model, *condmodel* into an inference algorithm, and run it. Then the inference algorithm will return the approximated conditional distribution (*conddist*) of the *failure time*.

```
procedure CONDMODEL
    endstate,failuretime=simulate();
    observe(endstate,shutdown");
    return failuretime;
end procedure
conddist=runinference(condmodel);
return conddist;
```

Algorithm 2: Conditional lifetime analysis

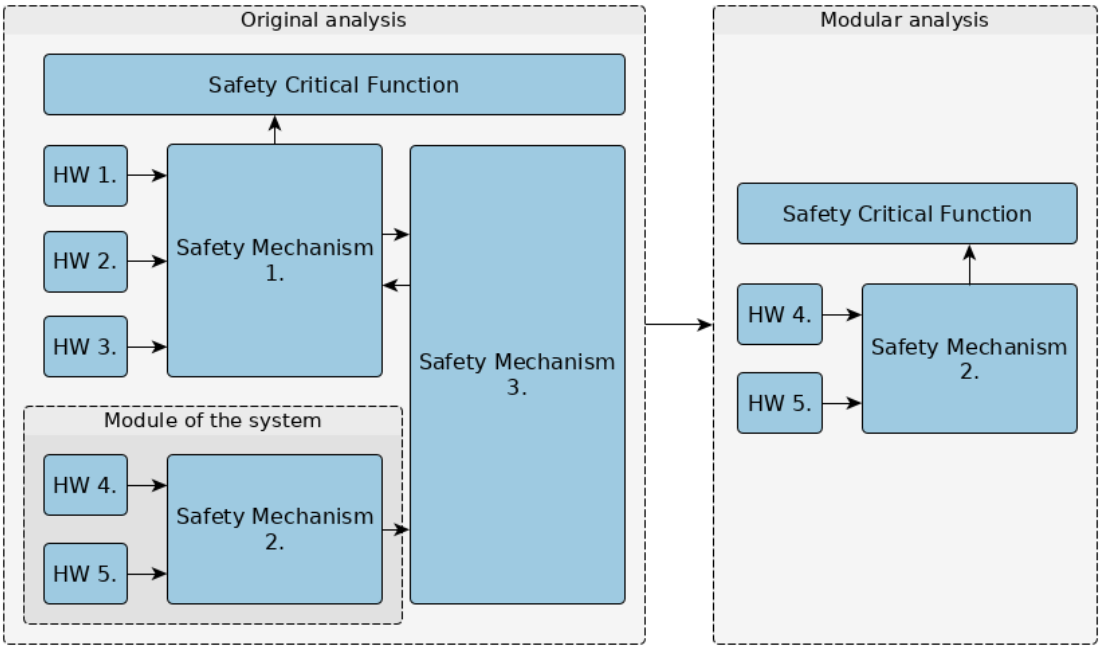


Figure 5.5. Example of the modular analysis

The conditional analysis also can be used for component sensitivity calculations. The main objective of this method is the investigation of how the lifetime of the system changes if a given component fails. Thus we have to create a *conditional model* where we assume (*observe*) that a given component fails during the mission time.

5.1.6 Modular analysis

Design time, it is essential to be able to analyze the system as well as the subcomponents of the system. Additionally, to increase reliability and availability, engineers have to know the behavior of the subcomponents. Moreover, ISO 26262 also requires this type of analysis. Therefore we have created the *modular* analysis algorithm, which can analyze the safety mechanism of the system individually. This method is especially useful if the system contains several complex safety-mechanisms.

We can analyze any component of the system separately from the other parts of the system with the help of the Gamma framework. An example of the modular analysis can be found in Figure 5.5. We can see that in this example, the function provided to Safety Mechanism 3. by Safety Mechanism 2. is the new safety-critical function in the modular analysis. As a result, we can apply to the new modular model all other analysis algorithm.

Thus we can execute on the module model the time-to-failure analysis or the conditional time-to-failure analysis. If the way that the subcomponent fails does not matter, we only have to execute the lifetime analysis as it defined in section 6.2.1.

If the failure mode of the component influence the whole system, in addition to lifetime prediction, we also have to execute conditional analysis introduced in section 5.1.5. Thus we create a *conditional model* for each failure mode and calculate the probability density function for all *conditional models*. In addition, we have to calculate the probabilities of the failure modes.

For instance, if on the example model the channel from Safety *Mechanism 2.* to *Safety Mechanism 3.* contains only one event, a failure event, then we only need the time-to-failure analysis. In contrast if this channel contains several failure events (*FailureMode1*,*FailureMode2*,...), then we have to execute conditional analysis on every event.

5.1.7 Pareto analysis

Besides ISO26262 industrial standards requires that an analysis must support the design process by discovering the most significant sources of the system failure. Nevertheless, traditional analysis techniques can not provide direct information about which hardware component contributes mostly to the failure of the system.

Therefore we have created the Pareto analysis, which calculates which component contributes mostly to the failure of the system. It collects all the dominant failure contributors with a simple qualitative algorithm (Algorithm 3). Thus we create a counter for each component initialized to zero. Then we run a vast number of simulations that collect the failure events, occurred until the failure of the system. After that, we increment each counter if its component failed during the simulation. Otherwise, if the failure occurred after the system became faulty, we decrement each counter if its component failed. After the simulations, the more critical components will have a higher counter than the irrelevant ones.

```
HWcomp1_cntr ← 0
HWcomp2_cntr ← 0
HWcomp3_cntr ← 0
...
for from 0 to 10000 do
  simulate()
  if HWcomp1 failed during the simulation then
    HWcomp1_cntr ← HWcomp1_cntr + 1
  else
    HWcomp1_cntr ← HWcomp1_cntr - 1
  end if
  if HWcomp2 failed during the simulation then
    HWcomp2_cntr ← HWcomp2_cntr + 1
  else
    HWcomp2_cntr ← HWcomp2_cntr - 1
  end if
  if HWcomp3 failed during the simulation then
    HWcomp3_cntr ← HWcomp3_cntr + 1
  else
    HWcomp3_cntr ← HWcomp3_cntr - 1
  end if
  ...
end for
return HWcomp1_cntr, HWcomp2_cntr, HWcomp3_cntr,... =0
```

Algorithm 3: Pseudo code of the Pareto analysis

5.2 Finite fault sequence analysis

The application of traditional Markov-chain-based analysis methods like DFTA and symbolic Markov-chains is difficult in the case of adaptive cyber-physical systems. Because the Markov-chain of the system model become huge because of the vast number of hardware components and also complex because of the sophisticated safety mechanisms. Due to this complexity also the verification of the results and the methodology might be difficult.

In this section, we introduce a new analysis technique that can calculate a conservative approximation of the failure mode probabilities and the failure rates. This algorithm evaluates the system model created by the modeling technique introduced in chapter 4. The concept of the algorithm is that the stochastic failure model of the hardware component and the deterministic system behavior are separated. We model the hardware components with several independent Markov-chains and the deterministic system behavior modeled by Gamma. As a result, we can also separately analyze the deterministic and the stochastic behavior of the system. Thus we separate the huge but structure part of the system from the complex but small part of the system. We decompose the analysis to a continuous-time Markov-chain analysis and a statechart verification. Because we use Markov-chains for the calculations, we have to approximate every failure distribution in the hardware model with an exponential distribution.

First of all, we do a finite step state exploration in the deterministic model to collect all possible fault sequences (with a given maximum length) that can result in safety goal violation. Because an only small number of hardware faults are expected during the lifetime of the system as the hardware components have a low failure rate. As a result, the probabilities of the long sequences are smaller with orders of magnitude. Thus we assume that a given number of hardware faults always causes safety goal violation. The efficiency of the state exploration can be increased with the following algorithm presented in [37].

Thereafter, we calculate the probabilities of each fault sequence that happens during the lifetime of the system. Then we sum the probabilities of each fault sequence that results in a safety goal violation in order to obtain the probability of the safety goal violation. The output of the analysis, besides the probability of safety goal violation, is a list. Its elements consist of failure sequence, the resulting end-state of the system, and the probability of the fault sequence. This list and the failure probability can be verified under ISO26262 by overseeing the elements of the list can be reviewed one-by-one.

5.2.1 The algorithm

In this section, we present step-by-step the execution of the FFSA. The flowchart of this algorithm is depicted in Figure 5.6.

In the first step, we generate from the deterministic models the *deterministic fault sequence evaluator* (DFSE) with the help of the Gamma framework. DFSE is a function which gets as parameter a series of faults called *fault_sequence* and investigates whether these faults brings the system into a failure state or not. Algorithm 4 shows how the DFSE calculate the state of the system after the faults in *fault_sequence*. The algorithm in a while cycle:

1. picks the faults from the *fault_sequence* one-by-one,
2. searches with the *fault_event_map* (presented in section 4.4) the state transition of the Gamma model which is triggered by the fault,

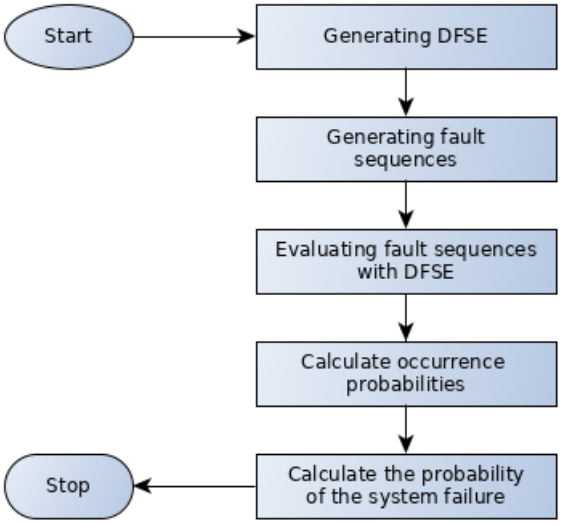


Figure 5.6. Flowchart of the FFSA

3. raise the event in the Gamma model,
4. inspects state of the system.

The cycle stops if a failure state is reached or we examined all elements of the *fault_squence*. In the latter case the endstate of the system is *"OK"*.

Result: state
state = "OK" ;
while *fault_sequence.has_next() AND state != "OK"* **do**
    fault = fault_sequence.next()[1];
    fault_event= fault_to_event_map.get(fault)[2];
    Gamma_model.raise_event(fault_event)[3];
    state = Gamma_model.get_state()[4];
end

Algorithm 4: Algorithm of the DFSE function

In the second step we generate every possible maximum $L \in N$ long fault sequences with the algorithm 5. Firstly, we generate every given L long permutation of the set of the hardware components. Then for each element of the permutation, we generate the Cartesian product of the sets of the component failure modes. After that, we calculate the union[1] of all these products, which gives the set of the L long fault sequences.

Thereafter in the third step, we investigate with the *DFSE* whether these fault sequences cause safety goal violation or not. Additionally, if the sequence caused safety goal violation, the *DFSE* also calculates which fault brought the system from an operational state into a failure state. We denote the index of this fault in the sequence with $K_{seq} \in N$.

In the fourth step we compute the occurrence probability of each fault sequence with the formula 5.2. In this formula $\lambda_{Ci}|i \in [1; L]$ defines the overall failure rate of ith component in the fault sequence and $\lambda_{Mi}|i \in [1; L]$ denotes the actual failure mode as the ith component

[1]Because the Cartesian products are distinct, we do not have to sort out the elements which are in two or more set. As a result, the computation of the union is fast.

```
failure_sequences=empty_set()
permutation_set = generate_permutations(length=L,set=components)
for all permutation in permutation_set do
  product_set=generate_cartesian_product(permutation)
  for all product in product_set do
    failure_sequences = failure_sequences ∪ product
  end for
end for
=0
```

Algorithm 5: Generation of the fault series

become faulty in the sequence. Furthermore λ_{sys} denotes the overall system failure rate which is the sum of the components failure rates as it was defined in equation 5.1.

$$\lambda_{sys} \leq \sum_{i=1}^{N} \lambda_{Ci} \tag{5.1}$$

$$P(seq) = \frac{\lambda_{M1}}{\lambda_{sys}} \cdot \frac{\lambda_{M2}}{\lambda_{sys} - \lambda_{C1}} \cdot \ldots \cdot \frac{\lambda_{MK}}{\lambda_{sys} - \lambda_{C1} - \lambda_{C2} - \ldots - \lambda_{C(M-1)}} \tag{5.2}$$

This probability gives information about when this sequence happens. Thus if we want to investigate whether the fault sequence occurs in the lifetime, we have to multiply the sequence probabilities with $P(t < T_{mission})$ the lifetime probability. We show in equation 5.3 overestimate of the probability that a L long sequence occurs during the lifetime of the system.

This probability gives no information about when this sequence happens. Thus if we want to investigate whether the fault sequence occurs in the mission time, we have to multiply the sequence probabilities with the probability that the sequence occurs in the mission time. We call this probability lifetime probability and we denote it with $P(seq; t \leq T_{mission})$. We show in equation 5.3 overestimate of the probability that a L long sequence occurs during the lifetime of the system. $P(t < T_{mission})$ can be calculated, where $Poisson(\lambda_{sys} \cdot T_{mission})$ is a $Poisson$ random variable with parameter $\lambda_{sys} \cdot T_{mission}$. Since in this step, we estimate the number of faults in the system with a Poisson process [21] with parameter λ_{sys}. As a result, this solution works only for exponential distributions.

$$P(seq; t < T_{mission}) \leq P(K_{seq} \leq Poisson(\lambda_{sys} \cdot T_{mission})) \tag{5.3}$$

In the last, fifth step we sum the occurrence probabilities of all sequence that cause the same type of safety goal violation. Thus to we get the upper-estimate of the probabilities of the safety goal violations we have to add to the previously calculated sums the correction probability. Because during the analysis we assume that the $(L+1)$th hardware fault always causes safety goal violation. The occurrence probability can be calculated with equation 5.4. In this formula we sum the probabilities of all possible at most N fault combination occur during the mission time.

$$P_{corr} = \sum_{i=1}^{N} \sum_{Ci \subset C} \left(\prod_{c \in Ci} \right) 1 - e^{(\lambda_c * T_{mission})} \tag{5.4}$$

L is the most critical parameter of the algorithm since this number defines how accurately we analyze the system behavior. If L is too small, the estimated lifetime probability will be unrealistically high, and the system will not pass the analysis even if the failure probability is under the limit. In contrast, if L is too big, the analysis will take too much time since the number of possible fault sequences grows exponentially with L.

No one in the automotive industry requires the analysis of more than two independent hardware faults. As a result, $L = 3$ is the right choice in most industrial cases.

Chapter 6

Evaluation

Therefore in this chapter, we present and answer one-by-one the most critical research questions about our analysis approach. Firstly, we examined the design parameters of our algorithm. Secondly, we examined the usability of our approach, we applied and evaluated our algorithms on the case-study (presented in Section 3.3). Finally, we examine the run time of our algorithm. We created several benchmark models to measure the scaling of our algorithm.

6.1 Design parameters

In this section, we present how we determined most of the most significant parameters of our algorithms based on the analysis of the case study. We focused on the parameters which directly affect the running time of our algorithms.

6.1.1 Simulation number

The TTF analysis has only one parameter, the simulation number. This parameter specifies both the resolution of the histogram and the accuracy of the results.

Setup To analyze the simulation number, we have chosen the TTF analysis of the case study model. We have run the TTF algorithm three times with exponentially increasing simulation numbers. Then we have chosen the smallest simulation number that provided acceptable accuracy for a histogram with 100 bins.

Results Firstly, we run 1000 simulations, but as can be seen in Figure 6.1, the resolution of the histogram was too low because we wanted at least 100 bins in the timescale. Then we run 10,000 simulations, which provided the required resolution. Finally, we set 100,000 the simulation number, and this parameter provided the same accuracy at the histogram as the 10,000 simulations. As a result, we run 10,000 simulations also for conditional TTF analysis.

Discussion We know that all of our analysis algorithms are strongly related to the operating time of the system. Therefore the simulation number is always 10 000 in every

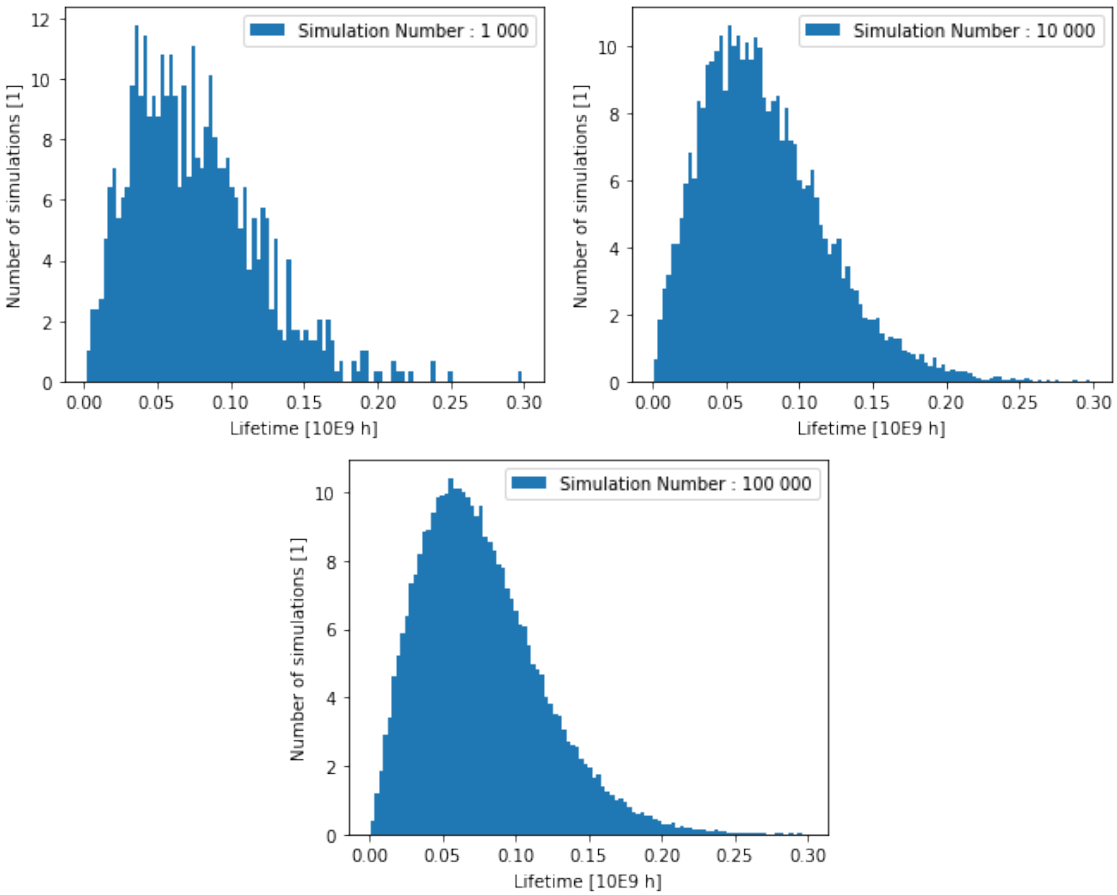


Figure 6.1. Comparison of different simulation numbers in case of TTF analysis of the case study

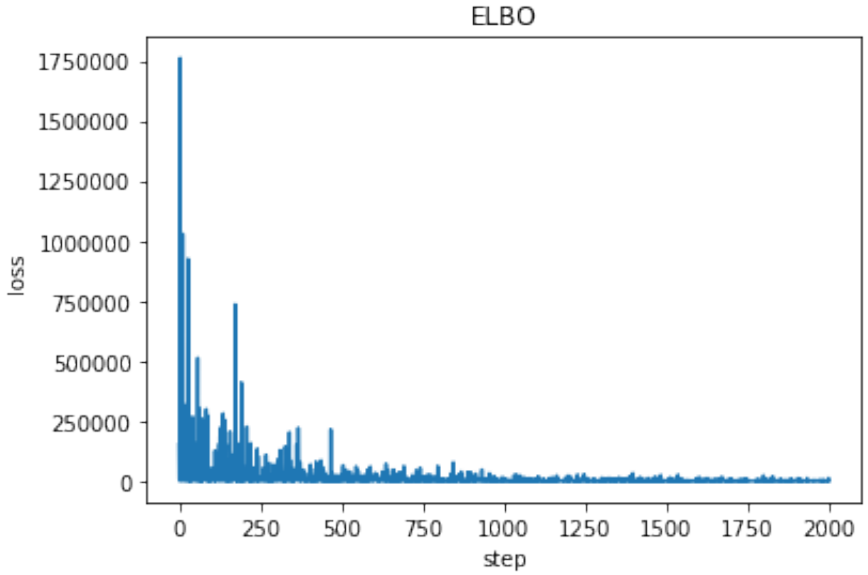


Figure 6.2. Diagram of the loss as a function of the step number

measurement. We know that these parameters can be different for a different model, but with this methodology, we can always easily calculated them.

6.1.2 SVI step number

If we use the SVI algorithm, one of the most critical parameters is the number of steps of the optimizer algorithm because this number determines both the run time and the accuracy of the algorithm.

Setup In order to determine this parameter, we saved a loss in the SVI algorithm. This loss is the difference between the model and the actual state of the guide. During every measurement, when we used SVI, we were continuously monitoring the difference between the fitted model and the real model.

Results As we can see, the model guide difference in Figure 6.2, the loss reaches a minimum after 1000 steps and stays there. Therefore when we used the SVI algorithm, we always used 2000 steps.

Discussion During the measurements, if we used the SVI algorithm, then we used always the same optimizer and the same (Weibull-based) guide function. Thus for all measurements with SVI, we used 2000 steps, and the 2000 steps were always enough.

6.2 Analysis of the case-study model

This section presents the application of the aforementioned analysis algorithms and evaluates their results for the case-study (introduced in Section 3.3). Section 6.2.1 introduces the probability distribution of the lifetime of the system, whereas Section 6.2.2 presents conditional lifetime predictions regarding the system.

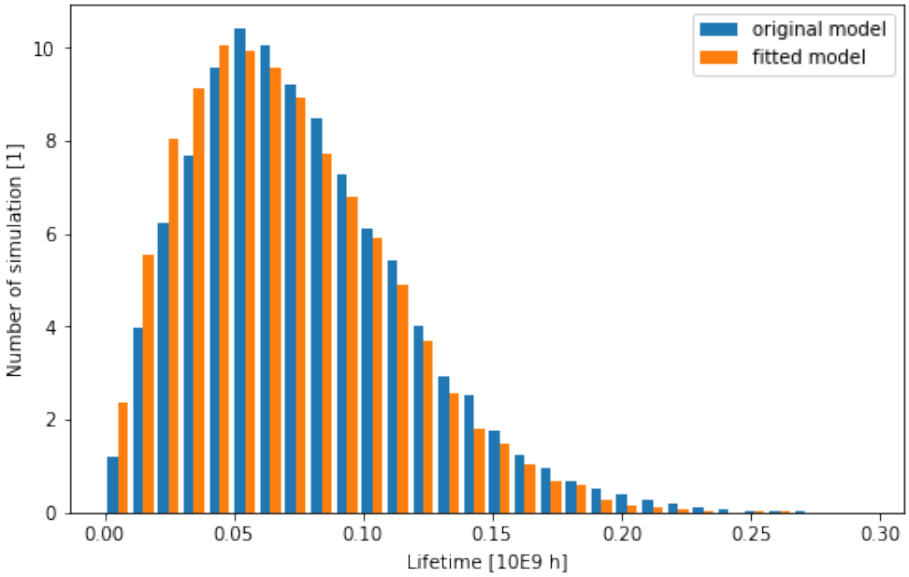


Figure 6.3. Comparison of the simulated and fitted lifetime distribution

6.2.1 Time-to-failure analysis

According to ISO26262, the main objective of every safety analysis technique is to decide whether the system complies with the safety requirements or not. Thus in the case of the EPAS, we have to calculate when and how the system fails. In this section, first, we present how to use our simulation-based analysis technique to calculate the probability density function of the time when the system fails. Then we show how to determine the satisfaction of the requirements. Finally, we verify the result with the *finite fault sequence* analysis method.

Setup We have run 10,000 independent simulation in the *PRE*, and visualised the results in Figure 6.3 with the blue histogram. As can be seen, the results resemble Weibull distribution characteristics. Thus, using the Pyro optimizers and SVI, we can easily fit a Weibull model to it. We have run 2000 steps with the SVI for the fitting.

Results The comparison of the real and the fitted Weibull model is depicted in Figure 6.3, where the fitted Weibull model is the orange chart. One can see that the resulting model is a good approximation for the scale and the shape parameters, which help us to understand how the reliability of the system changes over time. The scale parameter is 0.0794, which estimates the mean-time-to-failure. Furthermore, the shape parameter is 1.824, which means that the system is aging as the failure rate of the system is increasing as the time is passing.

Moreover, with the help of the Weibull parameters, we can calculate the probability that the failure occurs in the $50000h$ mission-time. This probability is $1.446 \cdot 10^{-6}$.

We verified these results with the FFSA. We examined every five long failure series and calculated the upper estimate of a safety goal violation, which is $1.583 \cdot 10^{-6}$. (The FFSA supports only the exponential distribution. Therefore we approximated the Weibull distribution with an exponential distribution with rate parameter.)

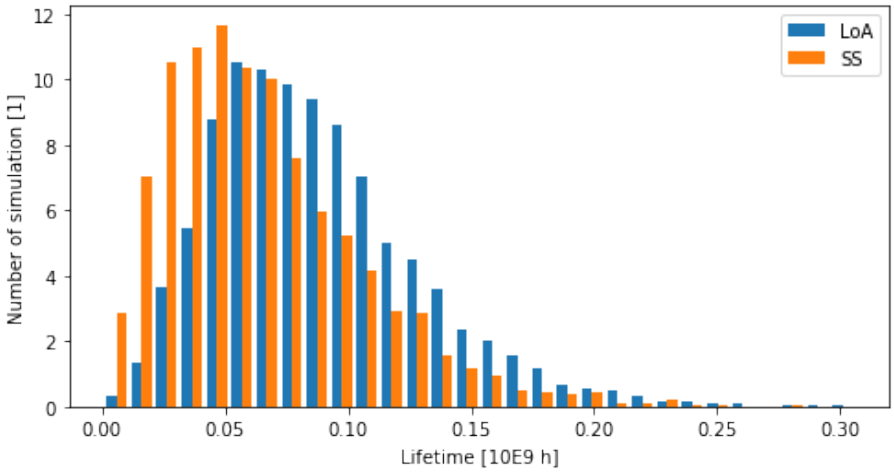


Figure 6.4. Comparison of the lifetime distribution of the *self-steering* (SS) and the *loss of assist* (LoA) failure modes based-on *Interest sampling*

Discussion We can see that the result of the FFSA is very similar to the result of the simulation, and both probabilities are lower than the required limit (defined in Section 3.3). Consequently, with our analysis approach, the safety requirements of the EPAS are satisfied.

6.2.2 Conditional lifetime prediction

The ISO26262 standard requires a detailed analysis of the failure modes. Therefore we have to analyze the conditional behavior the *self-steering* and the *loss of assist* failure modes from the perspective of the time of occurrence.

Setup Consequently, we created two conditional models in the *PRE* that give us the posterior distribution of the lifetime, assuming that the failure mode is *loss of assist* or *self-steering*. We used a simple inference algorithm since the vast number of random variables in the stochastic model makes the computation very slow for each algorithm that uses gradient estimation, e.g., Hamiltonian Monte-Carlo. Therefore we have chosen *importance sampling* algorithm, and we ran this algorithm ten-thousand times. Furthermore, we also applied the SVI algorithm with 2000 steps and with Weibull shaped guide function.

Results The comparison of the two resulted posterior lifetime distributions are depicted in Figure 6.4 and 6.5. We can see that in both diagrams that the *self-steering* occurs much earlier than *loss of assist*, but both failure modes have Weibull shaped failure time distributions.

Discussion The results meet the expectations owing to the following rule: if a sensor has a latent failure and any other sensor has any problem, the system will activate *self-steering* immediately. Even though the *loss of assist* failure mode occurs, when both uC-s have several shutdown type faults due to the high redundancy in the hardware. Thus *loss of assist* takes a much longer time than *self-steering*.

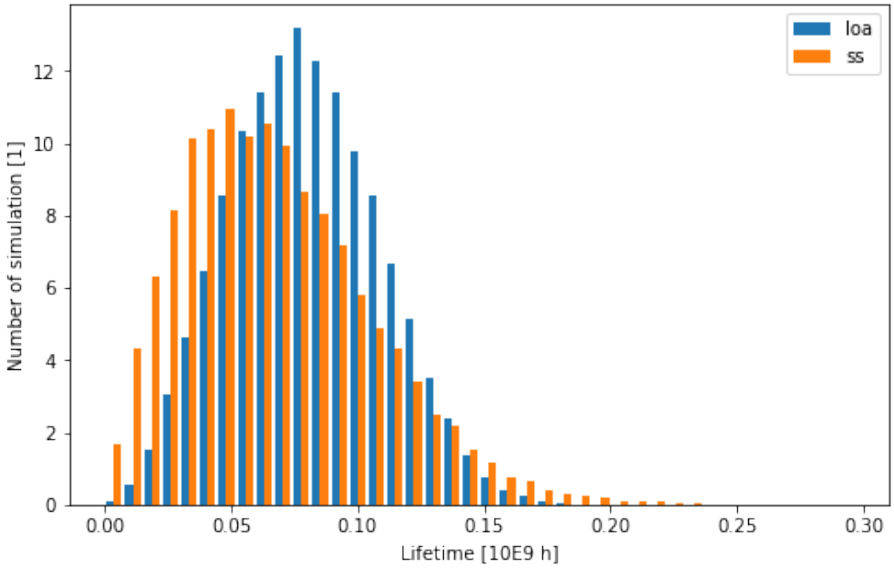


Figure 6.5. Comparison of the lifetime distribution of the *self-steering* (SS) and the *loss of assist* (LoA) failure modes based-on *SVI*

Hardware component	uC1	uC2	S1	S2	S3	S4	S5	S6
SS and LoA	1610	1684	1056	1120	1156	1150	1056	1044
only LoA	4228	3972	2052	1932	2096	1788	1984	1764
only SS	-738	-806	222	258	158	210	314	286

Table 6.1. Results of the Pareto analysis with 5000 cycle

6.2.3 Pareto analysis

In order to improve the availability of the EPAS, we have to examine what components are the most significant contributors to the system failure and what component is related to the system failures. Additionally, the ISO26262 standard requires that if the system complies the requirements, the safety analysis should help the design process to improve the safety of the system. Therefore we applied the Pareto analysis to the EPAS to reveal those failures of what hardware components in the system are correlated with the system failure.

This method gives no information about the cause and effect relationship, but it helps the designers of the system to find out which components should be in focus. As a result, after the Pareto analysis, another analysis method should be executed to reveal the exact effect of the components in focus.

Setup We executed the Pareto analysis for both a general system failure (when the failure mode of the system does not matter) and one-by-one for the failure modes of the EPAS (*Uncontrolled self-steering* (SS) and *Loss of assist*) (LoA).

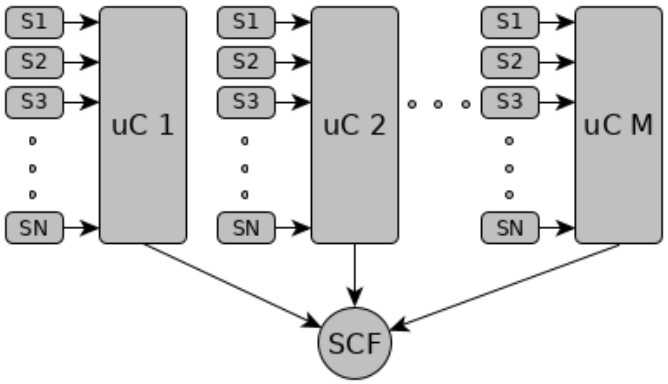


Figure 6.6. Additional models for benchmarking

Results The correlation ration of this analysis can be found in Table 6.1. The results of the Pareto analysis show that the EPAS failure is correlated closely with the failures of the uC-s even though the sensor failures are correlated with system failure.

Moreover, if we examine the failure modes separately one-by-one, then we can see that the *Loss of assist* failure mode is closely correlated with uC failures and the *Self-steering* is correlated closely with sensor failures. In contrast, the *Self-steering* is correlated negatively with uC failures since the failure of the uC turns off its motor control function immediately. As a result, after that, no sensor failure can cause *latent error* in the motor control of this uC.

Discussion The results meet the expectations, since only the latent sensor failures can cause *Uncontrolled self-steering* and the failure of the uC prevents the *Uncontrolled self-steering* in that uC. Moreover, a uC failure increases the chance of the *Loss of assist* significantly.

6.3 Run time measurements

In order to validate our analysis method, we have done several run time measurements. We examined the time-to-failure analysis as well as the conditional time-to-failure analysis and Pareto analysis. We ran all analysis scripts 5 times and calculated the median and checked the consistency of the results.

6.3.1 Setup

We tested our tool on several different benchmark models. For this, we used the EPAS, defined in Section 3.3, as a template of the benchmark models. We created three new versions of this model. Each new model version got some additional sensors and some new uC-s, as depicted in Figure 6.6. Similarly to the original EPAS model, each uC has a voting mechanics for their sensors. Furthermore, the system goes to state *Loss of assist* if no uC is operational and goes to state *Uncontrolled self-steering* if at least one of them uses bad sensor-data for the steering control.

For the measurements, we used an average PC configuration:

uC num.	2	2	4	4
Sensor num.	6	12	24	48
FFSA	0.477	1.150	5.189	36.856
TTF analysis	20.828	33.348	49.556	89.657
Cond TTF analysis	82.903	122.616	201.310	449.114
Pareto analysis	0.883	30.885	89.976	87.053

Table 6.2. Running time of the benchmark model simulations in seconds

- Ubuntu 18.04.3 LTS
- AMD Athlon X4 860K CPU
- 8 GB RAM
- GeForce GTX 1050Ti videocard
- Pyro-0.4.1
- PyTorch-0.4.1

6.3.2 Results

The results of the run time measurements are depicted in Table 6.2. We can see that all analysis scripts run successfully within 10 minutes.

6.3.3 Discussion

The results show that the speed of the analysis algorithms depends on the number of components (how many random variables are in the system). We can see that the run-time of the simulation-based analysis algorithms increases linearly with the size of the model. In contrast, the run time of the FFSA grows exponentially with the size of the model.

Moreover, the size of the state-space of the smallest model is all-together $3^6 \cdot 2^2 \cdot 4^2 \cdot 3^2 \cdot 3^2 \cdot 3^2 \cdot 3 > 1.02 \cdot 10^8$, because there are six sensors in the system which have 3-3 states. In addition, there are two uC-s with two states. Each of the two diagnostics have (3+1) states for steering actuation and three states in the operation region, and the controllers have three states both for steering and in the operation region. Finally, the evaluation statechart has also three states.

Moreover the size of the biggest model is all-together $3^48 \cdot 2^4 \cdot 13^2 \cdot 3^2 \cdot 3^2 \cdot 3^2 \cdot 3 > 4 \cdot 10^{68}$. Therefore we showed that our analysis algorithms could work reliably, even if the size of the model is too big to be analyzed by an exact solver.

Finally, the benchmark models have no unique attribute, which can make the analysis faster. Consequently, we can assume that our analysis algorithms are not significantly slower for any other similar-sized model.

Chapter 7

Conclusion and future work

In this work, we introduced a novel modelling, simulation and verification approach for the top-down reliability analysis of complex stafety-critical systems.

- The modelling approach proposed in Chapter 4, which relies on a SysML-like statechart modelling and composition languages *Gamma*, leverages tools familiar for systems engineers. Statecharts, which allow the use of numerous language extensions compared to basic state machines, are expressive enough to capture the high hardware of the hardware and complex reconfiguration strategies employed by the software components. The methodology conforms to recommendations in the ISO 26262 standards, making it especially amenable for use in the automotive domain.
- Chapter 5 presented the *Probabilistic Runtime Environment*, which can be automatically created from the system model using code generators. Leveraging the deep probabilistic programming framework *Pyro*, simulations can be executed in this environment rapidly and with high accuracy. The probabilistic programming paradigm allows analysts to quickly define new and advanced measures of interest and types of simulations. We also provides a collection of standard analyses, such as lifetime estimation and Pareto analysis.
- In addition Chapter 5 also presented *Finite Fault Sequence Analysis* based on the system models to verify simulation results. For qualitative verification, the analysis finds sequences of faults that lead to failure. Quntitatively, the analysis finds an upper estimate of the probabilities of the fault states in a manner that can be reviewed and verifier manually for certification purposes.
- Throughout this work, a cased study called the *Electronic Power-Assisted Steering* (EPAS) subsystem, inspired by an industrial project, was used to illustrate modelling and analysis concepts. The case study was first introduces in Chapter 3. In Chapter 6 we evaluated the reliability of this system, as well as measured the scalability of our implementations of the analysis algorithms. The experimental results show that our approach can model and analyze the reliability of a complex cyber-phyisical system, and it scales well even for large models.

In the future we plan to extend our methodology by incorporating the simulation of software components based on artificial intelligence, which nowadays play and ever increasing role in critical systems like self-driving cars. The integration between Bayesian learning approaches, neural networks and deep probabilistic programming provides an interesting avenue for end-to-end reliability prediction.

Moreover, we would also like to study the optimization of system parameters and the synthesis of system-level adaptation strategies based on deep probabilistic inference. The inference algorithm could output optimal design parameters (such as required component reliabilities and diagnostic coverage) at design time, or even suggest optimal reconfigurations of system architecture at runtime by real-time inference in response to diagnostic observations by monitoring.

Acknowledgements

We thank Dr. Péter Györke and ThyssenKrupp Presta Kft. for the summer internship.

Bibliography

[1] Iso26262; automotive industrial standard. https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en, . Accessed: 2018.

[2] Isograph reliability workbench. https://www.isograph.com/software/reliability-workbench/, . Accessed: 2019.

[3] Py4j python-java gateway. https://www.py4j.org/. Accessed: 2009.

[4] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

[5] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[6] Soheib Baarir, Marco Beccuti, Davide Cerotti, Massimiliano De Pierro, Susanna Donatelli, and Giuliana Franceschinis. The greatspn tool: recent enhancements. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):4–9, 2009.

[7] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. 2006.

[8] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[9] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[10] Hichem Boudali, Pepijn Crouzen, and Marielle Stoelinga. Dynamic fault tree analysis using input/output interactive markov chains. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 708–717. IEEE, 2007.

[11] Carlos E Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. Jani: quantitative model and tool interaction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–168. Springer, 2017.

[12] Anthony W Burton. Innovation drivers for electric power-assisted steering. *IEEE control systems Magazine*, 23(6):30–39, 2003.

[13] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell.

Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[14] Marko Čepin and Borut Mavko. A dynamic fault tree. *Reliability Engineering & System Safety*, 75(1):83–91, 2002.

[15] Tianqi Chen, Emily Fox, and Carlos Guestrin. Stochastic gradient hamiltonian monte carlo. In *International conference on machine learning*, pages 1683–1691, 2014.

[16] Tod Courtney, Shravan Gaonkar, Ken Keefe, Eric WD Rozier, and William H Sanders. Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 353–358. IEEE, 2009.

[17] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[18] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*, pages 592–600. Springer, 2017.

[19] Dov Dori and Edward F Crawley. *Model-based systems engineering with OPM and SysML*. Springer, 2016.

[20] Arnaud Doucet, Simon Godsill, and Christophe Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and computing*, 10(3):197–208, 2000.

[21] Richard Durrett and R Durrett. *Essentials of stochastic processes*, volume 1. Springer, 1999.

[22] Predrag Filipovikj, Nesredin Mahmud, Raluca Marinescu, Cristina Seceleanu, Oscar Ljungkrantz, and Henrik Lönn. Simulink to uppaal statistical model checker: Analyzing automotive industrial systems. In *International Symposium on Formal Methods*, pages 748–756. Springer, 2016.

[23] Sarita Gulia and Tanupriya Choudhury. An efficient automated design to generate uml diagram from natural language specifications. In *2016 6th International Conference-Cloud System and Big Data Engineering (Confluence)*, pages 641–648. IEEE, 2016.

[24] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987. ISSN 0167-6423. DOI: 10.1016/0167-6423(87)90035-9. URL http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[25] Myron Hecht, Alexander Lam, and Chris Vogl. A tool set for integrated software and hardware dependability analysis using the architecture analysis and design language (aadl) and error model annex. In *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 361–366. IEEE, 2011.

[26] Matthew D Hoffman and Andrew Gelman. The no-u-turn sampler: adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research*, 15 (1):1593–1623, 2014.

[27] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

[28] Ademir AC Júnior, Sanjay Misra, and Michel S Soares. A systematic mapping study on software architectures description based on iso/iec/ieee 42010: 2011. In *International Conference on Computational Science and Its Applications*, pages 17–30. Springer, 2019.

[29] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[30] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[31] Martin Leucker and César Sánchez. Regular linear temporal logic. In *International Colloquium on Theoretical Aspects of Computing*, pages 291–305. Springer, 2007.

[32] Gabriele Manno, Ferdinando Chiacchio, Lucio Compagno, Diego D'Urso, and Natalia Trapani. Matcarlore: An integrated ft and monte carlo simulink tool for the reliability assessment of dynamic fault tree. *Expert Systems with Applications*, 39(12):10334–10342, 2012.

[33] M. Ajmone Marsan. Stochastic petri nets: An elementary introduction. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1989*, pages 1–29, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. ISBN 978-3-540-46998-8.

[34] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. *Advances in applied probability*, 18(3): 747–771, 1986.

[35] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma Statechart Composition Framework. In *40th International Conference on Software Engineering (ICSE 2018)*, Gothenburg, Sweden, 2018. ACM, ACM.

[36] US (US) Department of Defense. *Military Handbook. Reliability Prediction of Electronic Equipment*. US Department of Defense, 1986.

[37] Enno Ruijters and Mariëlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer science review*, 15:29–62, 2015.

[38] Enno Ruijters and Mariëlle Stoelinga. Better railway engineering through statistical model checking. In *International Symposium on Leveraging Applications of Formal Methods*, pages 151–165. Springer, 2016.

[39] J. Salvatier, T. V. Wieckiâ#128#139, and C. Fonnesbeck. PyMC3: Python probabilistic programming framework. Astrophysics Source Code Library, October 2016.

[40] Lijun Shan, Yuying Wang, Ning Fu, Xingshe Zhou, Lei Zhao, Lijng Wan, Lei Qiao, and Jianxin Chen. Formal verification of lunar rover control software using uppaal. In *International Symposium on Formal Methods*, pages 718–732. Springer, 2014.

[41] Masoud Taheriyoun and Saber Moradinejad. Reliability analysis of a wastewater treatment plant using fault tree analysis and monte carlo simulation. *Environmental monitoring and assessment*, 187(1):4186, 2015.

[42] Dustin Tran, Alp Kucukelbir, Adji B Dieng, Maja Rudolph, Dawen Liang, and David M Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

[43] Huai Wang, Marco Liserre, Frede Blaabjerg, Peter de Place Rimmen, John B Jacobsen, Thorkild Kvisgaard, and Jørn Landkildehus. Transitioning to physics-of-failure as a reliability driver in power electronics. *IEEE Journal of Emerging and Selected Topics in Power Electronics*, 2(1):97–114, 2013.

[44] Peng Zhang and Ka Wing Chan. Reliability evaluation of phasor measurement unit using monte carlo dynamic fault tree method. *IEEE Transactions on Smart Grid*, 3 (3):1235–1243, 2012.