Validation and Analysis of Formal Methods using an Airbag Control Unit

Source: Robert Bosch GmbH

Frank Werner <fwerner@cs.uni-sb.de>

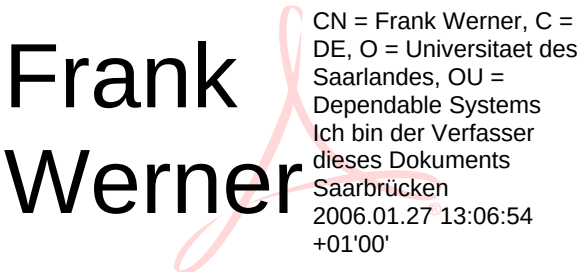
Diploma Thesis

Prof. Dr. Ing. Holger Hermanns
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung 6.2 – Informatik
Universität des Saarlandes, Saarbrücken, 2006

This Diploma Thesis is submitted to group of *Dependable Systems and Software* for the *Validation and Analysis of Formal Methods using an Airbag Control Unit* at Universität des Saarlandes, winter semester 2005/2006. The Airbag Controller and all related material at hand is designed and developed by *Robert Bosch GmbH*.

Being of sound of body and mind I hereby declare that the on-hand work was penned singly and no others but the stated sources and means were used.

Saarbrücken, January 23, 2006



Frank Werner

Acknowledgements

I want to thank Prof. Dr.-Ing. Holger Hermanns for proposing such an interesting and challenging mission with many various paths to be traversed, many things to learn and to develop on my own, and for constantly broadening my horizon during meetings.
Much of this success is owed to Marko Auerswald, my supervisor from the research group *CR/AEA* at *Robert Bosch GmbH*, for providing material, giving the right directions during this work as well as for help in finding proper information, and especially for patience in reviewing drafts.
In the end I would like to express my gratitude to my family, friends, and all those who gave me the possibility to complete this diploma thesis.

Contents

Chapter 0

Introduction

Many cars are nowadays equipped with a considerable quantity of safety features that protect their passengers in case of accidents and thus save them from lethal injuries. One of the most prominent to name in this area is the airbag and its control unit. Due to time criticality, the electronic control unit (ECU) has to decide within milliseconds whether the car hit a wall or just passed a bump on the road. Thus dependable safety systems are an important matter with respect to passengers life.
Dependability in this context can be understood as an integrative concept, that encompasses the attributes *availability* - the readiness to provide a correct service, *reliability* in the sense of to continue providing the correct service, and *safety* also known as the absence of catastrophic consequences on the user and the environment. Moreover dependable systems must fulfill requirements like *integrity* - understood as the absence of improper system state alterations, *confidentiality* or the absence of unauthorized disclosure of information, and *maintainability* or the ability to undergo repairs and modifications [LAR01].
All these requirements impose a challenge on the developing engineers as they need to be incorporated and fulfilled by the system. Especially the concept of maintainability is critical since every change within a component needs to conform to the proper function of the whole system.

The Airbag System

Modern belt tensioning systems - in particular in combination with an airbag - guarantee its passengers an excellent collision protection. One decisive element for the quality and effectiveness of such safety critical systems is the optimized interaction of its single components. This means that the perfect interaction of the chassis, the seat pattern, the tensioner characteristics, and finally the airbag determine the optimal system configuration and hereby ensuring maximal occupant protection.
Depending on the severity of the accident, the pyrotechnical belt tensioner deploys, driven by sensors after passing a defined threshold of procrastination. For example, a vehicle colliding with a solid barrier at a speed of 50 km/h, is causing the belt tensioner to deploy within 20 milliseconds, prestressing the belt with 1000 N [Bra01].
For the airbag deployment the situation is more complex. Different occupant kinematics - whether the occupant is belted or not, and the drivers weight - require different airbag system configurations, which are properly adjusted on the simultaneous interaction of various systems in use. A sensor clas-

sifies the severity of the accident by using a delay-time-course of the input signal. Once a sensor has triggered deployment, a fuze incorporated in a pyrotechnical container is activated and the bag inflates (deploys) within 25 milliseconds.

The restraint system FIRST (For Intelligent Restraint System Technologies) [Hub03] manufactured by Robert Bosch GmbH consists of sensors and actuators, distinguishable into two classes. One type is gathering data via sensors from exterior upon which they detect environment changes like front-, side-, and rear collisions, and rollovers. For this purpose crash sensors like pre-crash and up-front-crash sensor deliver sensor values to the central airbag control that has an integrated overroll sensing mechanism. The peripheral collision sensing is handled using side-airbag sensors (PAS).
The second type of sensors is dedicated to occupant sensing which builds the basis for computing the system configuration. In addition, an occupant classification sensor measuring the drivers weight, and the Out-of-Position-Sensor (OOP), to determine the optimal deployment configuration. For the co-driver's seat a mechanism is sensing mounted children-safety-seats, in which case the airbag deployment for the co-drivers airbag is switched off.

Preliminary Modeling Languages

Languages of frequent use are *MoDeST* and *Uppaal*. *MoDeST* [DHKK01] (*MOdeling and DEscription language for Stochastic Timed systems*) can be viewed as an overarching notation for a wide spectrum of models, ranging from labeled transition systems, to timed automaton, as well as prominent stochastic processes like Markov chains and decision processes. It contains features such as simple and structured data types that can be used to define subtypes (like ranges) of existing types, *C*-like structures, and also completely new types. In general *MoDeST* supports most C style expressions, arrays and structures.
In the current *MoTor* [BHKK03] (*MOdest TOol enviRonment*) implementation, which aims to provide means to analyze and evaluate *MoDeST* specifications, stochastic means to account for Normal, Uniform, and Exponential distribution are present. Exception handling, usually used to signal errors via `try` and `catch` in a *Java*-like style, provides a mean to handle exceptional events. Moreover constructs for non-deterministic branching (`alt`), random branching (`palt`), generic loop construct (`do`), and time are admissible. In *MoDeST* it is possible for each process to have its own notion of time provided by a `clock`, a variable like entity that changes its value linear and continuously over time.
A guard (`when`) is a boolean condition describing when an action is allowed to be executed. A deadline (`urgent`) in terms of a boolean condition described when an action must fired the latest. Multiple threads of control or parallelism can be introduced with the `par`-statement that enables action synchronization between processes. For running simulations, transitions are fired as soon as they become enabled (maximum progress).

Uppaal [LPY97] - which is accompanying models throughout this thesis - is an integrated tool environment for modeling, simulation, and verification of real-time systems. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or/and shared variables. *Uppaal*s main focus is on checking invariant- and reachability properties by exploring the state-space of a system, i.e. reach-

ability analysis in terms of symbolic states represented by constraints.
The model-checker is designed to check for invariants and reachability properties, in particular whether certain combinations of control-nodes and constraints on clocks and integer variables are reachable from an initial position or not. The query language of *Uppaal*, used to check specified properties is a subset of CTL [ACD90](*Computation Tree Logic*) .
The simulator allows the user to examine in an interactive and graphical fashion the dynamic behavior of a system. In contrast to the model-checker which explores the whole reachable state of a system, the simulator explores only a particular execution trace i.e. a sequence of states of the system.
The basis of the *Uppaal* model is the notion of timed automata developed by Alur and Dill as an extension of the classical finite-state automata with clock variables. An *Uppaal* model consists of a collection of timed automata extended with integer and clock variables. The edges of the automata can be optionally decorated with three types of labels: a *guard*, expressing a condition on the values of clocks and integer variables that must be satisfied in order for the edge to be taken; a *synchronization action* which is performed when the edge is taken, and finally a number of *clock resets* and *assignments* to integer variables. In addition, control nodes may be decorated with so-called invariants, which are conditions expressing constraints on the clock values in order for control to remain in a particular node.
Guards express conditions on the values of clocks, and integer variables that must be satisfied for the edge to be taken. Formally, guards are conjunctions of timing and data constraints.
In *Uppaal* automata may communicate either via global integer variables or using communication channels. As in CCS [Mil89] communication on channels occur as two-process synchronization. To identify the action that processes can perform when synchronizing with each other, the notation `a!` and `a?` is used to denote complementary actions of sending and receiving on channel `a`. In addition the use of keyword `broadcast` can be used for modeling a broadcast channel with one sender `a!` and multiple receivers `a?`.

About this Thesis

This thesis explores the possibilities of formal modeling and analysis of the airbag control unit, and has been carried out in close collaboration with *Robert Bosch GmbH, CR/AEA* in Frankfurt. The on-hand work strives for an assessment of the dependability of the airbag control unit manufactured by *Bosch*. Due to non-disclosure agreements, some parts of chapter 2 and 3 could not be displayed within the scope of this thesis.

A general overview of synchronization concepts will be given in chapter 1 where aspects of symmetric and asymmetric communication are pointed out that will be used later on in the modeling stage. The focus will be to develop a formal concept, out of which any possible combination of sender/receiver communication can be derived. In addition it is investigated, how to express certain concepts which are not natural in *MoDeST* or *Uppaal*. This gives for the remainder of this thesis an overview of synchronization, how it is understood in modeling languages like *Uppaal*, and *MoDeST*, and which aspects can be easily expressed.
Chapter 2 is a feasibility analysis on verifying properties by simulation with *MoDeST*. For this purpose a control unit of the airbag controller is analyzed using a critical situation that could lead to undesirable behavior. This very abstract model is already used in the scope of an *AMETIST* project

conducted by using *Uppaal*.
In this context an extended model of the ECU is modeled in chapter 3 to analyze the ECU using an exhaustive setting in order to assess the occurrence of race conditions. As before the main focus is on confirming results obtaining by the use of *Uppaal*. Since *MoDeST* is used for modeling, we find ourselves in the language class of *Stochastic Timed Automata* (STAs) which enables us to perform deeper studies with respect to probabilistic and stochastic means.
Chapter 4 will be devoted to rebuild the *Detailed System Modeling and Verification* from chapter 3 in *MatLab Simulink*. Although the *Stateflow* component of *Simulink* is deterministic and no stochastic and probabilistic means exists, we can use randomized sensor values from *Simulink* to obtain a distribution of the airbag deployment. Beyond that it is possible to predict according to the signal course, the deployment of the firing stage.

Analysis techniques for simulating the behavior of *Continuous Time Markov Chains* by *MoDeST* are covered in chapter 5 where the simulation outcome is compared with analytical solution approaches like *matrix exponentiation*. This insights are in turn used to assess the on-demand safety features of the airbag controller. In addition numbers like *Mean Time To Failure* (MTTF) and other probabilistic figures are computed by simulation in *MoDeST*.
The way fault trees can be computed is treated in chapter 6. Here a part of the airbag fault tree is modeled in *MoDeST* and outcomes are compared with results gained by *Fault Tree +* [Fau05]. Also preparing ground for *Fault Tree Generation*, this chapter gives the feasibility of simulating fault trees and especially to group component failures into logical units.
Automated failure analysis by using *Fault Tree Generation* is treated in chapter 7. Different possible points of failure like *noise*, *delay*, etc. are conceptually introduced and later on applied in simulation. Based on these simulation traces a static fault tree is extracted afterwards, reflecting the *static failure behavior* of the system. In addition, a failure translation for the preprocessor *GEMA* is defined.
Categorizing components within a circuit according to their importance is investigated in the *Importance Analysis* (chapter 8) by using *structural*, *marginal*, *Barlow-Proschan*, and *Fussell-Vesely* importance measures. These importances give different analysis techniques at hand to survey the impact of failing components on the system behavior. The *Fussell-Vesely* measure is later on applied to an example circuit to show how it can be obtained by simulation in *MoDeST*.

Chapter 9 is more visionary. It deals with the requirements to be fulfilled by a single source modeling formalism. Out of the single source model at hand simulation, verification, static-, and dynamic failure analysis can be derived. Special emphasis deserves the model of the water tank since it reveals best the advantage of dynamic over static failure analysis. Dynamic failures are computed according to a STA chain, reflecting the processes state space under the violating conditions.

Different extensions for *GEMA* - the *MoDeST* preprocessor - are illustrated in the appendix A, that were used in models. Due to the macro preprocessor, modeling in *MoDeST* is eased by introducing concepts, that can be used for *buffer generation*, *for-* and *while* loops, *invariants*, *clock arrays*, and *rates*.
Significant model sources and simulation results can be found in the corresponding appendix at the end of this thesis for the sake of completeness.

Chapter 1

Synchronization Concepts

Synchronization is coordination with respect to time and an important concept in the fields of computer science and electronic engineering. Synchronous operations take place in a fixed time relationship to other operations or events. In general, two notions of synchronization are used which are (1) the symmetric concept, were all participants are of the same kind and as thus treated equal, and (2) the asymmetric synchronization, where one differentiates between senders, and receivers.

This chapter investigates different principles of synchronization and tries to put the concepts as used in *Uppaal* and *MoDeST* into the right frame. Especially with respect to the electronic control unit of the airbag controller, different strategies are used to guarantee a synchronous behavior within a system. Since the concept of *broadcasting*, naturally useable in *Uppaal* is not natural in *MoDeST*, the *MoDeST* models of the airbag control unit have to be adopted for broadcasting.
Later on during this chapter, the *MoDeST* semantic is investigated with respect to symmetric communication, and common concepts are exemplified using workarounds, for instance to adopt for broadcasting channels in *MoDeST*.

1.1 Overview

Many different synchronization strategies are in place to fit specific needs, e.g. when dealing with a number of N devices and stressing that some of them are synchronizing at one instance of time, we refer to this as *some-of-N*. Furthermore, when having a sending- and a receiving side in terms of an asymmetric communication scenario we can refer to *one-of-M-to-at-least-n-of-N* to state, that one sender out of M is synchronizing (sending) with at least n out of N total receivers. This mechanism also referred to as *multi-casting*, is used to deliver information to multiple destinations simultaneously, using the most efficient allocation strategy possible. But what have all the concept mentioned so far in common, and how many participants are involved in a *some-of-M*, or *at-least-3* synchronization?
Mainly driven by the diversity of synchronization concepts, we try to formulate a general concept which comprises all known communication strategies. For example when using a synchronous scenario with 5 devices in total ($\Omega = 5$), and constrain that *at least 3* synchronize, we can rephrase this

as *at-least-3* synchronization. The total number of solutions in this case is 3, namely that either 3, 4, or 5 devices interact. The condition e.g. *"at-least 3"* is called predicate over the solution set for the remainder of this chapter.

1.1.1 General Concept

For investigating the general concept of synchronization, we use a formal approach to define a concept which has capability of providing a derivation for any elementary communication scheme. Ω is identified as the set containing all devices present to synchronize on some common action. We define the set $X_{\mathbf{P}(\cdot)} \sqsubseteq \Omega$ as the solution set containing all participants which are synchronizing assuming that predicate $\mathbf{P}(\cdot)$ holds. The "empty" synchronization where nothing takes place is explicitly forbidden since it has a rather philosophical character.
A trivial solution for the predicate $\mathbf{P}$ to become always true is $\top$ that contains all numbers of possible solutions. This puts basically no constraints on the subset relation and is just mentioned for completeness.

$$\begin{aligned} X_\top &:= \{i \in \mathbb{N} \mid 1 \leq i \leq |\Omega|\} \\ &= \{1, \ldots, \omega\} \qquad \text{, with } \omega = |\Omega| \end{aligned}$$

For the nontrivial case we write $\mathbf{P}(\sim m)$ where $\sim \in \{=, \leq, \geq, \neq\}$, $m \in \mathbb{N} \setminus \{0\}$, and demand for the set of the number of participants X, that $\mathbf{P}(\sim m)$ holds. Let $w_j \in \Omega$ be the name of device i and $A_i \subseteq \Omega$ set of size i of devices with

$$A_i := \dot{\bigcup_j} w_j.$$

It holds, that $|A_i| = i$ and

$$\bigcup_{1 \leq i \leq |\Omega|} A_i = \Omega.$$

$$X_\top \supseteq X_{\mathbf{P}(\sim m)} := \{i \in \mathbb{N} \mid i \sim m \quad \text{and} \quad 1 \leq i \leq |\Omega|\}$$

Example

Consider a scenario as mentioned above consisting of $\Omega = \{a, b, c, d, e\}$ devices and the predicate $\mathbf{P}(\geq 3)$ on set X ($X_{P(\geq 3)}$). According to the predicate, we demand each solution of the resulting set of participants to consist of at least 3 participants. The resulting set $X_{P(\geq 3)}$ contains - as defined by the nontrivial case from above - elements $\{3, 4, 5\}$, and as such there exist three possible solutions (derivations) for this scenario.

1.1.2 Symmetric Communication

The concept referred to as symmetric communication is similar to the one being used in *MoDeST*, since all parties are equal and one does not differentiate between sender and receiver. A derivation tree is depicted in the following where $\top$ denotes true, meaning that no constrains are requested on the subset relation. Ω denominates the set of all devices and m a natural number that adds the desired constraint. We abbreviate as already denoted above $\omega = |\Omega|$ and obtain the unconstrained solutions set $X_\top = \{1, \ldots, \omega\}$ which contains all solutions for synchronizing device combinations.

The derivation tree for the *symmetric synchronization* is depicted in figure 1.1. Solid lines indicate the different possible derivations. On the root the solution set X is shown on which we either put no constrains such that we end up with solution set $X_\top$, or we restrict the set by the use of a predicate $P(\cdot)$ over the number participants. When taking x as the cardinality of set X and a natural number m, the relation $x \sim m$ is used to condition the size of the solution set in one of the four possible ways.
By using $x \leq m$, we want the number of participants that result from the predicate to be exactly m. Furthermore, there exists one single deduction from the tree when using predicate $\mathbf{P}(x = m)$ which is the solution with exactly m synchronization partners. Using the notation $x \geq m$, the derivation contains all combinations with at lest m participants, analog to the $x \leq m$ case. Finally predicate $\mathbf{P}(x \neq m)$ yields all deductions that $X_\top$ supplies except that the number of participants can not be equal to m.
Dashed lines finally provide the resulting sets that follow out the derivation tree. Out of this concept all one-sort symmetric synchronizations can be derived.

Notice that two derivations become spare, which are

$$m = \omega \quad \Longrightarrow \quad X_\top = \{1, \ldots, m\} = X_{P(\leq m)}$$

because the relation condition is trivially true and

$$m = \omega \quad \Longrightarrow \quad X_{P(\geq m)} = \{\omega\} = X_{P(=m)}$$

since ω is the maximal number of synchronizing participants.

1.1.3 Two Sort Synchronization

Uppaal has a perception of a *pair concept* in which a sending and a receiving side establish a communication, and hence the symmetric model from above needs to be extended appropriately. In a *two sort* synchronization we differentiate between senders, and receivers, and formulate in general terms that some number $s \in W$ out of $|X|$ possible senders synchronize with some receivers $r \in Y$ out of $|Z|$ total receivers. Deviating from the model before, two groups of participants are identified as senders and receivers. This brings up the following shorthand, consisting out of a sending and a receiving side separated by "to". For $m, n \in \mathbb{N}$ we obtain

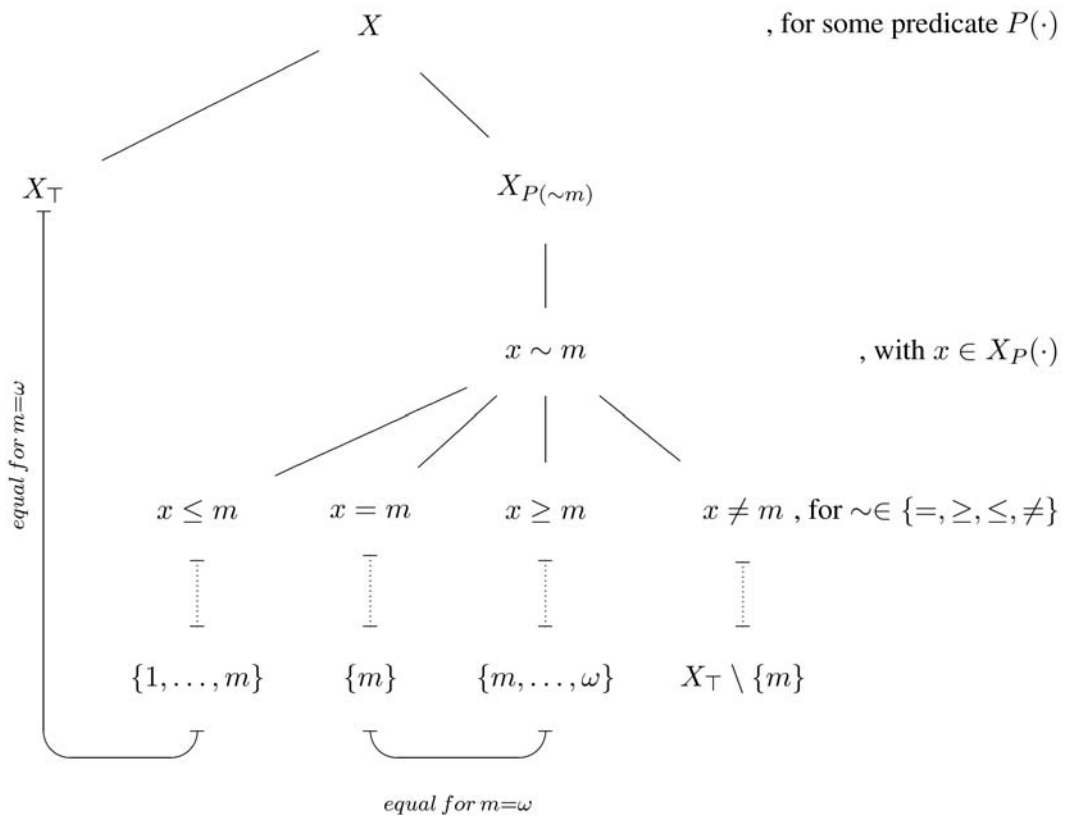


Figure 1.1: Derivation tree in which all actions need to synchronize instantly. Leaves denote the set of possible participants.

$$W_{P(\sim m)} \sqsubseteq X \qquad \text{to} \qquad Y_{P(\sim n)} \sqsubseteq Z$$

The derivation tree of asymmetric communication is thus more complex and shown in figure 1.2 with $w = |W|$ and respectively $y = |Y|$.

As for the symmetric version, we start on the root of the sending and respectively receiving side. For convenience the trivial case ($X_\top$) is omitted since is has less practical use. By using the constraining set with predicate $\mathbf{P}(\sim m)$ where $\sim$ is either $=$, $\leq$, or $\geq$, we derive the solution sets for the sending- and the receiving side. Note that the predicate $\mathbf{P}(\neq m)$ is avoided since is has no practical use. As before in the symmetric communication some of the derivations become redundant, that is for the sending side

$$m = \omega \quad \Longrightarrow \quad W_{P(\geq m)} = \{|X|\} = W_{P(=m)}$$

and the receiving side

$$m = \omega \quad \Longrightarrow \quad Y_{P(\geq m)} = \{|Z|\} = Y_{P(=m)}.$$

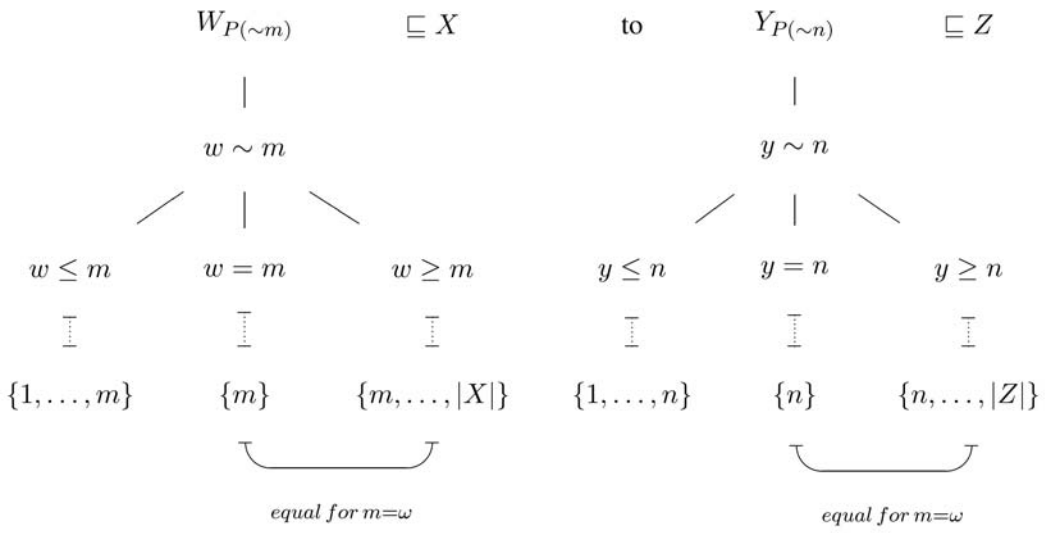


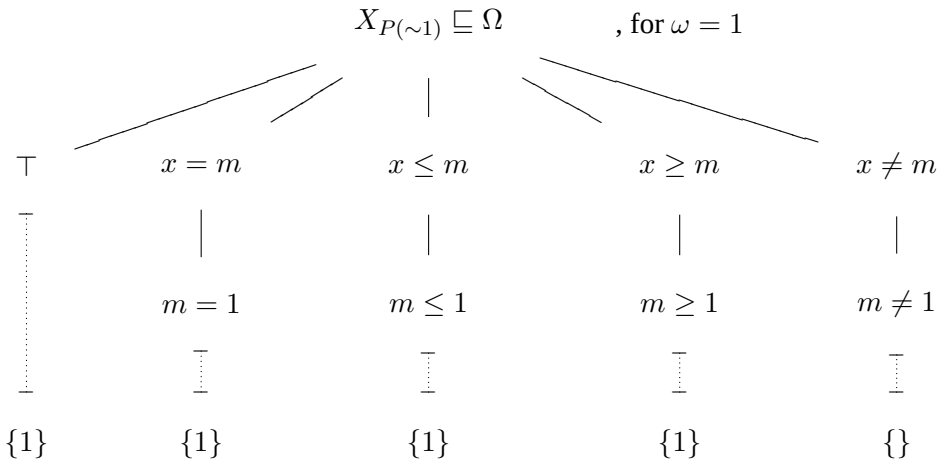
Figure 1.2: Derivation tree of the asymmetric synchronization consist of a sending and a receiving side. Dashed lines illustrate the derivation of the sets of possible participants.

1.1.4 Derivations

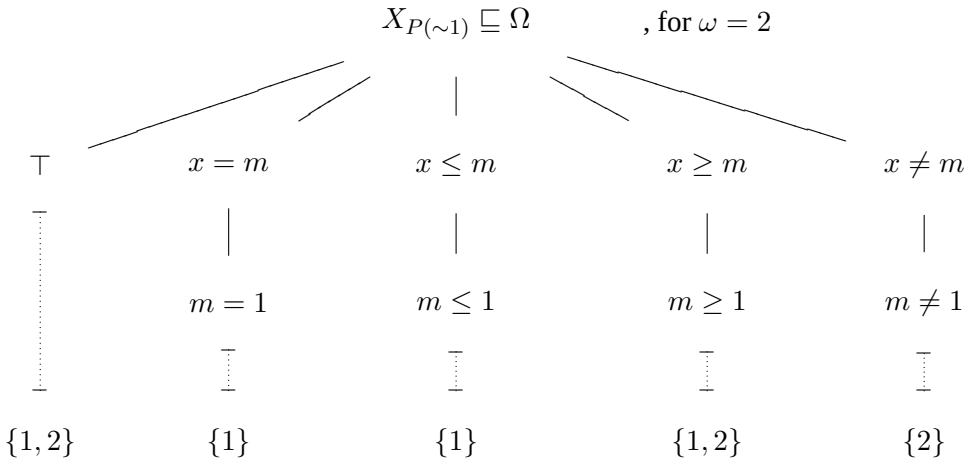
By having set the theoretic framework we are in the favorable position of building derivation trees that reflect the behavior of any sort of communication thinkable. Since interest is devoted to languages already in use we refrain from becoming too general and restrict on already known concepts as being used in *Uppaal* and *MoDeST*. Numbers in braces (participants) are connected to the tree using a dashed line, and denote the possible participating devices for this combination, that results from the chosen path.

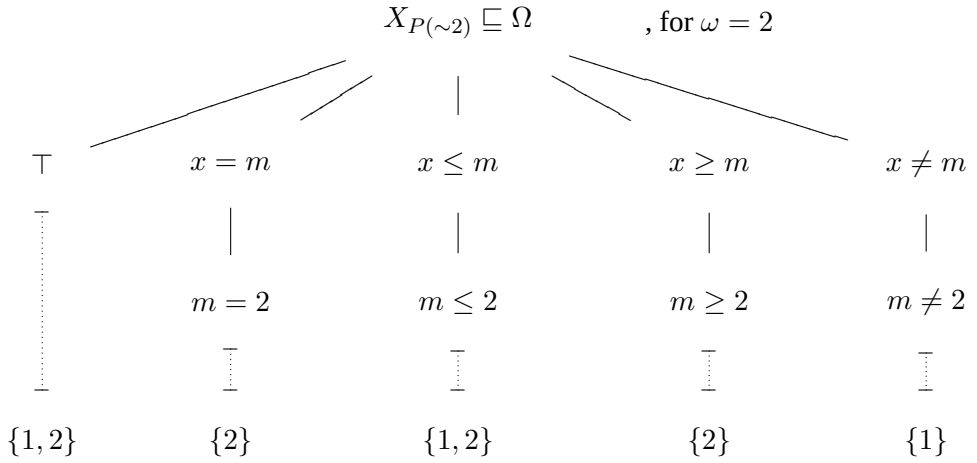
General Symmetric Communication

When having a set Ω consisting of one participant ($\omega = 1$) the following symmetric communication scheme is thinkable.



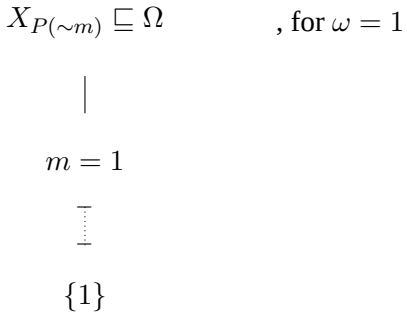
Extending Ω to contain two participants $(\omega = 2)$ we obtain two derivations trees. In the first tree by using predicate $\mathbf{P}(\sim 1)$ and the second tree by the use of $\mathbf{P}(\sim 2)$.

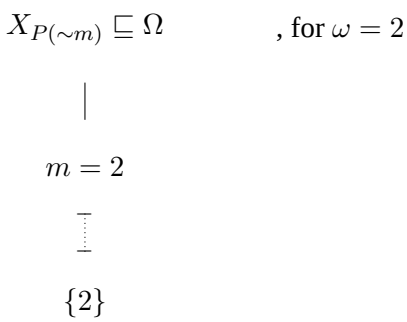




Special Case: *MoDeST*

Due to the fact that *MoDeST* is using a special notion of synchronization on actions, it deserves special emphasis. Since *MoDeST* implements symmetric communication in an "all-or-none"-fashion ($m = \omega$) it is a distinct instantiation from the general symmetric communication concepts. Therefore the only valid derivation is $X_{P(=\omega)}$. As before solid lines indicate the different derivations and dashed lines the derivation of the final solution set.
The following trees show derivations for different ω's, ranging from 1 up to 3.





$$X_{P(\sim m)} \sqsubseteq \Omega \qquad \text{, for } \omega = 3$$

$$\Big|$$

$$m = 3$$

$$\vdots$$

$$\{3\}$$

Special Case: *Uppaal* **Pair**

The *Uppaal* pair synchronization is asymmetric allowing one sender to communicate with one receiver. Since the presence of senders and receivers is required to establish a synchronization, there exists only one valid branch.

$$W_{P(\sim m)} \sqsubseteq X \qquad to \qquad Y_{P(\sim n)} \sqsubseteq Z \qquad \text{, for } |X| \geq 1, |Z| \geq 1$$

$$\Big| \qquad\qquad\qquad\qquad \Big|$$

$$m = 1 \qquad\qquad\qquad\qquad n = 1$$

$$\vdots \qquad\qquad\qquad\qquad \vdots$$

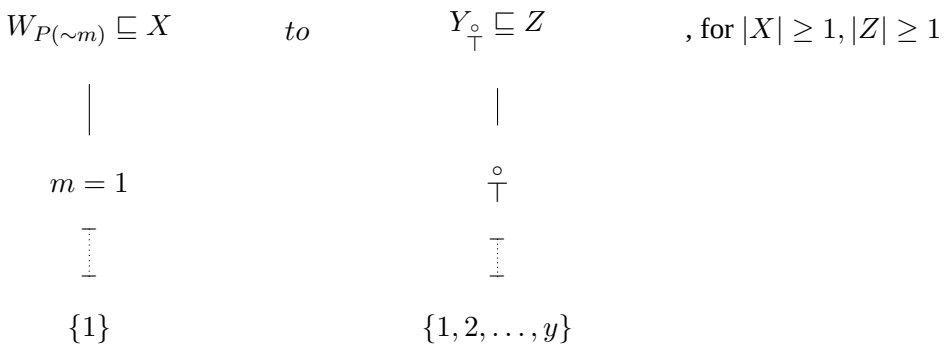
$$\{1\} \qquad\qquad\qquad\qquad \{1\}$$

Special Case: *Uppaal* **Broadcast**

The *Uppaal* broadcast is a specialization of the general asymmetric variant. It allows only 1 sender to send to arbitrary many "ready-to-receive" receivers. In addition to the predicate defined above we write $\overset{\circ}{\top}$ as the property that fulfills all receivers which can synchronize and are ready-to-receive.

$$\overset{\circ}{\top} ::= \quad P(Y) \iff \forall y \in Y : \text{can synchronize}$$

$$\wedge \qquad \forall z \in Z \setminus Y : \text{can not synchronize}$$



1.2 Semantics of *MoDeST*

Having a method at hand that subsumes all synchronization concepts we shortly outline the semantics of *MoDeST* [DHKK01] and *Uppaal* to see where the differences in synchronization behavior stem from. The only construct in *MoDeST* that enables synchronization of processes $P_1, \ldots, P_k$ is the parallel composition PAR {$::P_1 \ldots ::P_k$ }. Operator $||_B$ denotes parallel composition of processes with B$\subseteq$Sync[1], the set of common actions that is acquired by the union of the alphabets.

PAR is defined by

$$\mathsf{PAR}\{:: P_1 \ldots :: P_k\} := (\ldots((P_1||_{B_1}P_2)||_{B_2}P_3)\ldots P_{k-1})||_{B_{k-1}}P_k$$

This implements the classical *blocking-sender broadcasting synchronization* or so called *symmetric synchronization*, by which no explicit sender is identified. Synchronizing occurs whenever two or more processes that run in parallel have a common action on which they must then synchronize. Hence only actions out of the common synchronizing set are worth considering.

Let in the following P and Q be two concurrent processes, a$\in$ B an action out of the shared synchronization set B. Set B consists out of all actions names that appear in process P and Q minus τ-actions. Expressing the operational semantics in inference rules the following two rules are obtained. The first rule states in the premise that process P can do an a-step to P‘ (written $P \xrightarrow{a} P'$) and similar for Q. Since a is in the synchronization alphabet ($a \in$ B) both processes can jointly change their states via action a to P‘ and Q‘ which is stated in the conclusion of the inference rule.
In case that a$\notin$B no communication occurs and none of the processes can change its state. This could for example happen if action a is as well in the alphabet of a third process, which is not ready for an a-step. The inference rules shown below denote synchronization of two parties. Only small changes are required to adopt for more participants.

[1]Note that for simplicity we only restrict to one type of action, rather than differentiating between *impatient* and *patient actions* as in [DHKK01].

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q' \quad (a \in \mathsf{B})}{P||_B Q \xrightarrow{a} P'||_B Q'} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q' \quad (a \notin \mathsf{B})}{P||_B Q \xrightarrow{a} P||_B Q}$$

As shown above no sender or receiver is identified in the semantics of *MoDeST* and processes P and Q are treated equal. On the opposite when dealing with asymmetric concurrency (*Uppaal pair concept*) sending and receiving processes are identified as follows:

$$\frac{P \xrightarrow{a!} P' \qquad Q \xrightarrow{a?} Q'}{P||Q \rightarrow P'||Q'}$$

Here process P is explicitly identified as sender and process Q as receiver. So whenever P can communicate via a! to Q where the latter is listening, both processes can move to the next state.

Extending *MoDeST* to account for the non-blocking sender broadcasting, a sender must have to be identified within the system, which is then never blocked and always able to execute its action (*input-enableness*). Therefore modelling the broadcast behavior in *MoDeST* is related to putting some extra work in modelling. By adding an appropriate extension of the *GEMA* preprocessor a considerable amount of work could be removed.

1.3 Broadcasting in *Uppaal* and *MoDeST*

Since *MoDeST* does not support broadcasting by nature, the investigation of the broadcasting ability seems eligible. Beyond that the analysis shows to what extent modeling broadcasting in *MoDeST* is possible and which of the concepts seen earlier can be expressed in *Uppaal*.

1.3.1 Broadcasting in *Uppaal*

Uppaal has apart from the pairwise synchronization (regular channel `chan`) an other mean of expressing synchronization of processes, namely broadcasting.
Binary synchronization is defined in *Uppaal* by `chan c` in which a sender `c!` will synchronize over a channel `c` with a receiver `c?`. A synchronization pair is chosen non-deterministically if several combinations are enabled.
Broadcast channels (`broadcast channel`) [BDL02] are defined in *Uppaal* as non-blocking synchronization, where one sender `c!` can synchronize with an arbitrary number of receiver `c?`. Receiver that can synchronize in the current state must do so. If there is no receiver, the sender can sill execute the `c!` action since broadcast sending is never blocking.
Since no such concept exists in *MoDeST*, the question rises whether a work-a-round for broadcast channels exists, meaning that such a channel can be easily modeled?

1.3.2 Broadcasting using a Channel

Intuitively one can think of broadcasting as a system consisting of senders, receivers, and a common channel. So whenever a sender has to transmit, the message is passed on to the channel which in turn tries to distribute it among "ready-to-receive" receivers in a multi-casting fashion. The channel has thus to maintain a list of currently sending processes and ready receivers and decide upon this knowledge which parties will be included in the communication. Synchronization is done using sender-side actions that are used to synchronize with sending processes and similar for the receiving side. In other words modelling a channel in *MoDeST* two groups of actions are used, one for the sending and one for the receiving side.
The problem that arises now when trying to model in *MoDeST* is, that it lacks constructs that support atomicity of actions. Atomicity is only possible in variable statements but no construct is given to execute more than one action at an instance of time. This is rather a design issue of the *MoDeST* language, because by synchronization of more than one action atomically at a time, processes might get stuck.

1.3.3 "One-to-One" in *MoDeST*

The One-to-One synchronization exists in two variants, where the first is standard *binary synchronization* and the 2^{nd} is *non-blocking broadcasting* where the sender is not blocked. Both types can be modelled easily in *Uppaal*. *MoDeST* actions are by nature blocking synchronization which means, that a workaround has to be applied for modelling the broadcasting feature called *input enableness*. Input enableness states that in order to overcome the blocking behavior τ-actions are introduces that do not change the function but guarantee instead, that every action can be executed at any time. A τ-transition does not affect the behavior of its containing process and in consequence not the process's "state".

"One-to-Many" in *MoDeST*

In *MoDeST* no broadcast primitive exists, so the system is modelled explicitly using an approach where the sender has the status of all receivers and decides upon that which receivers take part in the synchronization. It is desirable to have broadcasting ability atomic, but although *MoDeST* features an atomic primitive for variables to ensure atomicity on a sequence of statements, this cannot be used in our situation, as only non blocking statements are allowed in atomic enclosings.
A model that has the characteristics of a broadcast channel can be modeled in *MoDeST*, although being circuitous, it is still possible to do. The atomicity requirement stresses that no sending process may start a new communication before each listening receiver has obtained the old message. In other words, we forbid the occurrence of two consecutive sends while capable receivers did not finish to receive the first broadcasting sequence. This is modeled by allowing the sender to check which of the participating receivers are ready, before a broadcast is initiated.
The following model consists out of 4 parties, a `sender` and three `receivers`. The sender sends data with a frequency, that is uniformly distributed between 0 and 3. The receivers pick their waiting

times out of different uniform distributions such that they are not ready to receive all at the same time. Variables `r1, r2, r3, s` count the number of packets being received by a receiver or sent by the sender. The simulation results and *MoDeST* model of the One-to-One synchronization can be found in appendix B.1.

"One-of-Many-to-One" in *MoDeST*

In the following, a model is given that contains three sending processes put in parallel with one receiver. The actions for sending or receiving are independently obtained from a Uniform distribution. Appendix B.2 contains simulation results and a summary of the synchronizing events.

"One-of-Many-to-Many" in *MoDeST*

The concept of "one-of-many-to-many" is exercised in *MoDeST* using two senders and two receivers which send/receive determined by independent Uniform distributions. Simulations are executed for 100 time units. The sources and simulation results are captured in appendix B.3.

“At-least-m“ in *MoDeST*

This communication concept is rather seldom used since the user has to know how many parties are ready to participate in the communication. Nevertheless, *Uppaal* does not have any means to account for this behavior by nature and so doesn‘t *MoDeST*, since it would require counting the number of participants. Nevertheless it is feasible but awkward to put extra guards and constraints into the model that require a specified number of participants to be ready before the communication is accomplished. The question popping up here, what should be done if less than M parties are ready, is left open and depends on the situation.

1.3.4 "N-to-N" in *Uppaal*

Having a system in which all processes have to synchronize at an instance of time [Figure 1.3] can be modelled in *Uppaal* by choosing one process to be the *leader* which does pairwise synchronizations steps with all other processes involved. In addition committed states are added, that guarantee that no time is passing, and no other communications takes place in between two consecutive synchronization steps.
An alternative implementation is to model the leader process additionally, issuing broadcasting to all N participants. Out of the N participants perspective this will look like N-to-N. This is awkward when modelling big systems and at the same time not a proper concept of multi-way synchronization since actions are executed sequentially rather that all in one shot.

In *MoDeST* this principle of N-to-N is rather natural since all actions that have a common action synchronize unless actions are hidden via the **hide**-operator or relabeled.

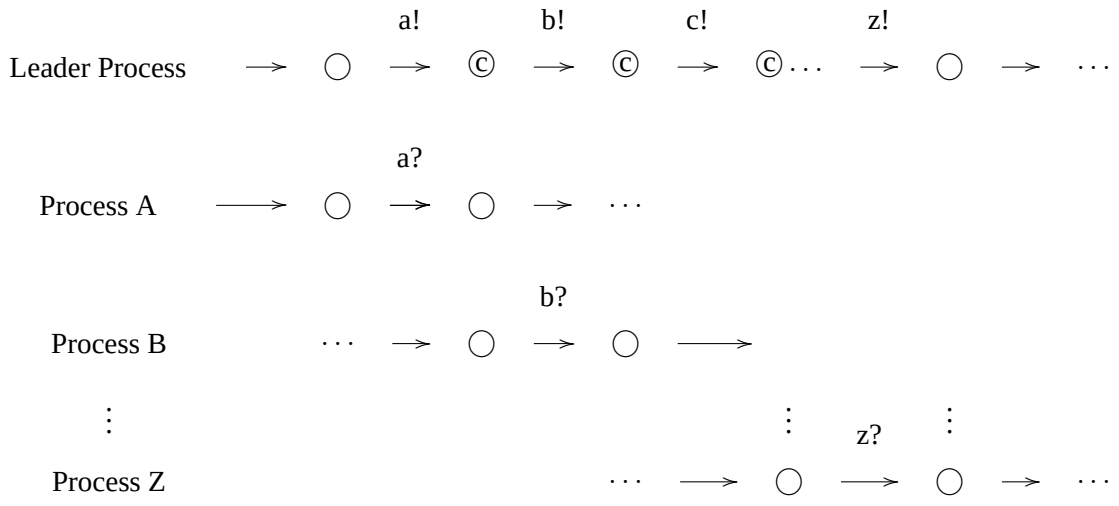


Figure 1.3: *Uppaal* N-to-N synchronization obtained by adding a so called leader process.

1.4 Conclusion

Summing up many different communication concept were studied with respect to their implementability and usability. We introduced a formal representation that allows to derive symmetric and asymmetric synchronization of any kind. Models of frequent use are the classical *one-to-one* synchronization, or *one-of-many-to-one* which can be used for instance for modelling a network with many clients and one central server. The pure multi-casting (e.g. *one-of-many-to-M*) is seldom used since it requires the knowledge of how many receivers are ready to receive.
Table 1.1 depicts some of the concepts with their implement ability in *MoDeST* and *Uppaal*. Symbol $\surd$ is used to stress that the concept is native in the language. ($\surd$) denotes that it is possible by putting some effort in the modelling as shown in the examples above. Concepts labelled with [1] are exemplified and can be found on the pages of the appendix of this chapter [App. B].
Where broadcasting exists as an in-house concept in *Uppaal*, it requires some extra work to adopt for that ability in *MoDeST*. In contrary, for a non-broadcasting synchronization with more than 2 devices, *MoDeST* is the language of first choice.

Sync Type	Formal Definition	|X|	|Z|	*Uppaal*	*MoDeST*
one-to-one (binary sync)	$W_{P(=1)} \sqsubseteq X$ - to - $Y_{P(=1)} \sqsubseteq Z$	1	1	$\surd$	$\surd$
one-of-many-to-one	$W_{P(=1)} \sqsubseteq X$ - to - $Y_{P(=1)} \sqsubseteq Z$	> 1	1	$\surd$	$(\surd)^1$
one-of-many-to-many	$W_{P(=1)} \sqsubseteq X$ - to - $Y_{P(\geq 2)} \sqsubseteq Z$	> 1	> 1	$\surd$	$(\surd)^1$
one-of-many-to-M	$W_{P(=1)} \sqsubseteq X$ - to - $Y_{P(=M)} \sqsubseteq Z$	> 1	> 1	$(\surd)$	$(\surd)$
one-of-many-to-at-least M	$W_{P(=1)} \sqsubseteq X$ - to - $Y_{P(\geq M)} \sqsubseteq Z$	> 1	> 1	$(\surd)$	$(\surd)$
some-of-all-to-1	$W_{P(\geq 1)} \sqsubseteq X$ - to - $Y_{P(=1)} \sqsubseteq Z$	> 1	1	$(\surd)$	$(\surd)$
some-of-all-to-some	$W_{P(\geq 1)} \sqsubseteq X$ - to - $Y_{P(\geq 1)} \sqsubseteq Z$	> 1	> 1	$(\surd)$	$(\surd)$
some-of-all-to-M	$W_{P(\geq 1)} \sqsubseteq X$ - to - $Y_{P(=M)} \sqsubseteq Z$	> 1	> 1	$(\surd)$	$(\surd)$
M-of-all-to-1	$W_{P(=M)} \sqsubseteq X$ - to - $Y_{P(=1)} \sqsubseteq Z$	> 1	1	$(\surd)$	$(\surd)$
M-of-all-to-some	$W_{P(=M)} \sqsubseteq X$ - to - $Y_{P(\geq 1)} \sqsubseteq Z$	> 1	> 1	$(\surd)$	$(\surd)$
at-least-M-to-some	$W_{P(\geq M)} \sqsubseteq X$ - to - $Y_{P(\geq 1)} \sqsubseteq Z$	> 1	> 1	$(\surd)$	$(\surd)$
at-least-M-of-all-to-1	$W_{P(\geq M)} \sqsubseteq X$ - to - $Y_{P(=1)} \sqsubseteq Z$	> 1	1	$(\surd)$	$(\surd)$
all-of-N	$W_{P(=\|N\|)} \sqsubseteq X$	N		$(\surd)^1$	$\surd$

Table 1.1: Overview of synchronization concepts and whether they are easy to model in *Uppaal* or *MoDeST* or not.

Chapter 2

Scenario Analysis

In this chapter a critical situation of the supplement restraint system (SRS) of the electronic control unit (ECU), formerly tested in *Uppaal* [Aue05] is modeled in *MoDeST* [DHKK01]. Intended to fortify present analysis of the *SRS ECU Behavior Model*, this approach is giving insights of how to do model checking via simulation. In particular, since concepts like broadcasting (cf. chapter 1) are not natural in *MoDeST*, different ways are investigated to adopt for a broadcasting-like behavior. By using probabilistic branching and random variables within the new model we expect to fortify the present analysis results gained in recent studies where *Uppaal* was used.

2.1 Model structure

The model is build in analogy to the *Uppaal* model at hand with the following processes. For reasons of secrecy we only display the model of the Observer Process.

- environment model (*Event_1*, *Event_2*, *Event_3*)
- model of the micro controller that has control for a font bag (*FrontBag*) and a belt tensioner (*BeltTensioner*)
- model of the external approver (*Approver*)

The following figure [2.1] gives an abridgement of processes used. Rectangles with a solid line represent single automaton, while dashed line boxes indicate a set of automata, and synchronization channels are shown as named edges.

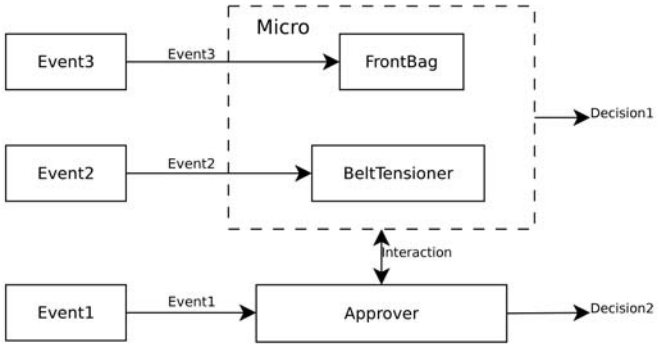


Figure 2.1: Device setting of the electronic control unit for the scenario analysis.

2.1.1 Environment Model

The environment model describes a simulation scenario of special interest by controlling the model behavior. Variable **waiting** is introduced to delay the execution of two consecutive events and measure the impact on the correct behavior. It will be increased over simulation runs. The waiting time is defined as the time between two consecutive *Event1*s.
The environment is modeled as the sequential occurrence of events *Event2* and *Event3*, each immediately preceded by an *Event1* event with no time passing in between.

2.1.2 Micro controllers

The simplified model of the micro controller consists of two automata, one for two-stage front bag and the other for two-stage belt tensioner deployment. The two stage front bag is modeled to deploy its 1st stage upon event *Event3* with a disposal of the 2nd stage 200ms later. The two stage belt tensioner is modeled similar where the 1st stage deploys upon event *Event2* and the second stage 60ms later.
The *Uppaal* Model of the ECU uses four channels (**Interaction**, **Event1**, **Event2**, **Event3**), declared as broadcasting. In the *MoDeST* model the latter three are declared as binary synchronization, since a interaction between the Environment and FrontBag/BeltTensioner is desirable. Thus Front Bag or Belt Tensioner have to synchronize always on an *Event* action.
The channel **Interaction** is devoted to be of special interest since it is modeled in *Uppaal* using a multiple-to-one fashion, meaning the two processes *Front Bag* and *Belt Tensioner* should not being blocked when sending *Interaction*. An adequate design in *MoDeST* is thus adding actions *Interaction_fb_joined*, *Interaction_fb_alone*, *Interaction_bt_joined*, and *Interaction_bt_alone* indicating, that the approver synchronizes on *Interaction* with a corresponding process or not . By using the variable **ready_approver** of type integer the approver indicates whether it is ready to do the transition. By this approach, process *BeltTensioner* and *FrontBag* have no capability to synchronize

between each other, but only with the external approver via actions *Interaction_bt_joined*, and *Interaction_fb_joined*. If the approver is not ready to synchronize, the processes *belt tensioner* and *front bag* are not blocked and can execute the corresponding *alone*-action (*Interaction_bt_alone*, *Interaction_fb_alone*).

2.1.3 External Approver

The design of the *Approver* process is using states, that are represented via a global vector (variable **state**) of type integer. A state change is consecutively expressed by a change of this vector, e.g. by taking transition from state A to B the state variable changes from **0** to **1**. Possible values for the **state** variable are ranging from 0 to 3 to represent one of possible four states (A, B, C, D).

2.2 Properties and Variables of Interest

Some variables in the model are defined to merely reflect state changes. In the *approver* process actions **A**, **B**, **C**, and **D** reflect the actual approver state, since the value of the concurrent variable **state** can not be inferred during the simulation run. After simulation they appear in the simulation trace and can be used for analyzing the system behavior. Especially when debugging a model, defining such variables is beneficial because they have no impact on process synchronization, but help to detect errors in the model using the trace path.

```
action A, B, C, D;  //Approver States
action FirstStage, FirstStage_fb, FirstStage_bt;
action SecondStage, SecondStage_fb, SecondStage_bt;
action SetEnabled, SetDisabled;  //trace purpose
```

By having some idea about the times spend in a certain location one can easily check the plausibility of the whole model. This motivated the following timing variables that measure times, or time intervals spent in certain states.

```
//Abort_X measures the sojourn time in a state X at abort
float Abort_B, Abort_C, Abort_D;
float EnableTimeSpan;   //the time for which  enable == 1
```

Finally one is interested in the number of violations, as being declared in property 1 on the following page that occur during a run. To be able to trace the occurrence of violation, action *violated* is defined that appears in the trace whenever a violation actually occurs.

```
//Counter for violation to test against property 1
```

```
int violation=0;
action violated;
```

The **waiting** time is defined external and increased during the simulation as being the measure of interest. This is accomplished via the definition of

```
extern const float waiting;
```

2.2.1 Observer Process

By having a *MoDeST* model of the electronic control unit at hand we now strive to verify the results obtained by the *Uppaal* study by means of simulation. This is obtained by definition of an observer process that guards the desired condition and increments a *violation* counter whenever property 1 is violated.
The property of primary interest is that whenever the *Front Bag* is at stage one or two, or the *Belt Tensioner* is at stage one or two respectively, **enable** should be true; or stated vice versa, the vital system condition is violated whenever property 1 holds. Our particular interest is devoted to the behavior within a critical interval that has been gained by previous "human" analysis and has been testified with *Uppaal*. We restrict simulation on this interval plus some overhead, to fortify previous model checking approaches.

Property 1

$$\begin{array}{ll} \mathbf{E}\Diamond & (FrontBag.FirstStage \vee FrontBag.SecondStage \vee \\ & BeltTensioner.FirstStage \vee BeltTensioner.SecondStage) \\ \wedge & \neg enable \end{array}$$

The observer process guards the expression stated at property 1. Besides the violation counter the trace path [Fig. 2.2] gives an additional mean of detecting violations.

```
process Observer(){
    do{
        ::alt{
            ::when (enable == 0)
                alt{
                    ::FirstStage_bt      {= violation+=1 =}
                    ::SecondStage_bt     {= violation+=1 =}
                    ::FirstStage_fb      {= violation+=1 =}
                    ::SecondStage_fb     {= violation+=1 =}
                }; violated
            ::when (enable == 1)    tau
        }
    }
}
```

2.3 Simulation

The simulation is carried out using the reward variables set in *Möbius* [San05] as named below measuring the *Instant Of Time* value at 500 time units. *Möbius* is a discrete event simulation runtime environment used to simulate *MoDeST* models. The values for **enabled** are measured using the *Time Averaged Interval* of length 500.
The values for variable **waiting** are successively incremented starting from 80 up to 296 time units.

2.3.1 Reward Variables

Variables of primary interest is **violation** that counts incrementally the number of violations over a simulation run. In addition four variables (**InA**, **InB**,**InC**,**InD**) are defined to capture the number of times a state is entered.
Möbius supports the use of four types of *reward variables* that measure the variable assignment for which they are defined. The type of a reward variable determines the point or interval in time when the reward function is evaluated.

- **Instant of Time:** The reward function is evaluated at the specified point in time.
- **Interval of Time:** Returns the weighted sum of all of the values of the reward function, where each value is weighted by the amount of time the value is in existence within the defined interval.
- **Time Averaged Interval of Time:** The variable returns the interval of time result, divided by the length of time for the interval.
- **Steady State:** The reward function is evaluated after the system reaches a steady state. The steady state simulation algorithm used is referred to in literature as *batch means*. This approach assumes that there is an initial transient period that must pass before the system reaches its steady state behavior. Once the system is in steady state, the algorithm evaluates the reward function multiple times to gather the observations and to compute the statistics. This technique is appropriate when enough time occurs between the samples to permit the assumption that the samples are independent of each other.

2.3.2 Trace Path

A clipping showing the firing rules for a simulation trace is given in figure 2.2. The trace path gives a second method at hand to see how the simulation is carried out. All actions that synchronize are followed by a *Sync*. Actions which are executed solely are labeled without *Sync*.

```
–> A
–> Event_1Sync
–> Event_2Sync
–> FirstStage_btSync
–> SetEnabled
–> B
–> Interaction_bt_joinedSync
–> C
–> SecondStage_btSync
–> Event_1Sync                                                    10
–> Event_3Sync
–> FirstStage_fbSync
–> Interaction_fb_alone
–> SetDisabled
–> A
–> SecondStage_fbSync
–> tau
–> violated
```

Figure 2.2: One batch of a trace path is depicted for a waiting time of 140. Only the firing rules are shown for clarity.

2.4 Observation Results

The following results are obtained by simulating 1000 times where repeating lines are dropped out of the table. Reward variable **ViolationCount** changed over the critical interval to a value above zero as predicted by previous model checking. This fortifies the *Uppaal* study and proves the feasibility of expressing *Uppaal* models in *MoDeST*.
Table 2.1 holds the number of times a location is reached. This means for instance, that in a run with **WaitingTime** of 257, the approver is twice in state *A*, and once in states *B*, *C*, and *D*.

2.5 Conclusion

One can conclude that property 1 on page 22 is satisfied over the critical interval. On the border region the property is violated because the violation count is above 0. This observation is coherent with the findings obtained by present analysis approaches, showing that *MoDeST* has capabilities of expressing *Uppaal* models.

Using the simulation approach we end up with two means to show the behavior of the model. On the one hand the simulation trace can be considered to check which process synchronized using which action. Besides this reward variables allow to extract certain values of the reward function and thus obtain a second way to investigate the model.
It is worth to mention that simulation has its assets and drawbacks, i.e. simulating a model for one

WaitingTime	InA	InB	InC	InD
80	2	1	1	0
⋮	⋮	⋮	⋮	⋮
87	2	1	1	0
88	2	1	1	0,344
89	2	1	1	0,499
⋮	⋮	⋮	⋮	⋮
256	2	1	1	0,499
257	2	1	1	1
⋮	⋮	⋮	⋮	⋮
287	2	1	1	1
288	2	1,514	1,514	0,486
289	2	2	2	0
⋮	⋮	⋮	⋮	⋮
296	2	2	2	0

Table 2.1: Simulation results revealing the time spent in each states.

million times and analyzing for an undesirable condition does not refute the fact, that these conditions do not exist at all. It rather stresses that during one million runs these situation has not occurred in the system and consequently has not been seen by an observer.

Chapter 3

Detailed System Modeling and Verification

The correct operation of the SRS ECU [Aue05] depends on one channel having current information about the other channel. A *race conditions* (race hazard) could occur if one channel changes its state while another channel produces its outputs based on outdated information about the first channel. Thus two simultaneous processes might falsely process inputs upon which they generate different data. This behavior was analyzed in previous studies using *Uppaal* and situations in which such race conditions were examined.
This chapter is a feasibility analysis of present *Uppaal* models that aims to check how *Uppaal* models can be converted into *MoDeST*. In particular probabilistic branching and stochastic information are added to the models. By the use of an observer process the former *Uppaal* properties like “a deadlock free model” etc. are verified.

3.1 Model Structure

The model of the SRS ECU is structured as figure 3.1 shows. The environment is producing sensor values that are read by the micro controller and approver. By interaction between this two components they send their commands to the firing stage which then triggers the airbag deployment. The dashed interaction line represents several actions upon which approver and micro controller synchronize.

Some of the processes used in *MoDeST*[DHKK01] could be easily derived from *AMETIST Uppaal* report [Aue05], others needed special treatment to adopt for broadcasting behavior, and to guarantee deadlock free behavior. As shown in chapter 1 we account for broadcasting behavior with two involved processes by using binary synchronization. In case that more than two participants are present, the behavior is incorporated in several actions that imitate broadcasting, i.e. every sender has several actions to synchronize on. In case that no receiver is present it does a unique action, on which no other process can synchronize. If ready-to-synchronize processes are present in the model, an action is chosen such that interaction is enabled on this. Due to existing non-disclosure agreements the mod-

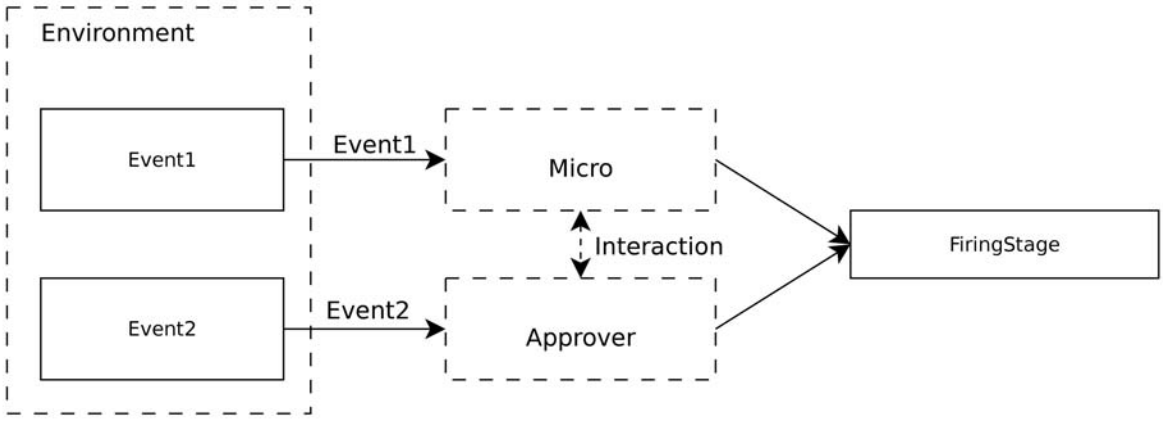


Figure 3.1: Device setting of the electronic control unit for the detailed system modeling and verification.

els that were used for simulation can not be displayed within this thesis.
In many processes τ-actions are introduced over which no synchronization is possible. This is done using *input enableness* to guarantee a deadlock free simulation. Input enableness states that in any state any action is enabled. In *MoDeST* τ (`tau`) represents an internal step, in particular τ is a local actions which is not attainable for synchronization. If taking a τ transition the respective process does not change its state, since τ actions have no effect on the behavior of its process.

3.1.1 Observer Processes

The following properties are defined to be of special interest:

- The model is free from deadlocks.

  Property 0 $\mathbf{A}\Box \quad \neg deadlock$

- The model is able to fire.

  Property 1 $\mathbf{E}\Diamond \quad FiringStage.Fire$

- If race conditions occur, the micro controller will not be able to make the correct assumption about the state of the firing stage. The result will be that although the micro controller wants to fire (Micro.Firing) and the external approver enables firing (Approver.Enable), the firing stage will not fire.

  Property 2 $\mathbf{E}\Diamond \quad (Micro.Firing \wedge Approver.Enable \Longrightarrow \neg FiringStage.Firing)$

```
/*
        Property1: E<> FiringStage_RV1_MV1.Fire

        Property2: E<> (Micro.Firing AND Approver.Enable
                        -> NOT FiringStage.Firing)


*/

process Observer_Prop1()                                                    10
{
        when(LocationFiringStage == 3)
                Property1Satisfied {= Property1+=1 =}
}


process Observer_Prop2()
{
        when (enable == 1 && LocationMicroFiring==1 && LocationFiringStage!=2)
                Property2Violated {= Property2+=1 =}                        20
}
```

Figure 3.2: Sources of the two observer processes.

The observer process [Fig. 3.2] handles validation of property 1 on the preceding page and property 2 on the page before. Property 1 on the preceding page is proven by using process Observer_Prop1() which checks if it is possible for process FiringStage to eventually reach location Fire. When doing so action Property1Satisfied is issued and the property counter is incremented. Violation of Property 2 on the previous page is watched by Observer_Prop2().

3.2 Simulation and Observation

Simulation is done over the variables as mentioned in table 3.1. The variable results were calculated using *instant of time* after one million time units, i.e. that the reward function is evaluated at this specific point in time, returning the corresponding variable value. Variables Property1 and Property2 [Tab. 3.1] reflects the number of times the properties are satisfied or respectively violated.
1 000 runs of the model are simulated, each for one million time units and the model did not lock up. This testifies that no deadlock behavior is observable up to the maximum time of simulation which also hints, that property 0 is not violated. This is a rather vague argumentation but since the simulation model used was checked for deadlock behavior in *Uppaal*, it is a sanity check. If a deadlock would exist, the simulation would only do τ-steps up from this point, because τ steps can be executed at any point during a simulation without any constraints. But since this is not the case, the model can be considered to be deadlock free.

```
13625.91292472  Event_1_0
13804.08185945  Event_2_1
14000.00000000  fg_interruptSync
14000.00000000  start_foregroundSync
14000.00000000  read_sensor_valueSync
14001.00000000  read_status_algoDecisionSync
14001.00000000  AlgoDecisionUnl
14002.00000000  read_status_approver_bSync
14003.00000000  algo_decisionSync
14003.00000000  AlgoDecisionUnl
14004.00000000  trigger_firingSync
14004.00000000  finished_foregroundSync
14004.00000000  break
14004.00000000  MicroFiringFiring
14004.00000000  send_fireSync
14004.00000000  MicroFiringInitial
14004.00000000  Property1Satisfied
14004.00000000  FiringStageFire
14004.00000000  FiringStageUnl
14500.00000000  fg_interruptSync
14500.00000000  start_foregroundSync
14500.00000000  read_sensor_valueSync
14501.00000000  read_status_algoDecisionSync
14501.00000000  AlgoDecisionUnl
14502.00000000  read_status_approver_bSync
14503.00000000  algo_decisionSync
14503.00000000  send_lockSync
14503.00000000  AlgoDecisionEn
14503.00000000  FiringStageEn
14504.00000000  trigger_firingSync
14504.00000000  finished_foregroundSync
14504.00000000  break
14504.00000000  MicroFiringInitial
```

Figure 3.3: Part of a simulation trace where the μC is firing and Property1 holds.

	Mean Value	Confidence Interval +/-
Property1	1	0
Property2	0	0

Table 3.1: Proof of properties that were obtained by the observer process via simulation.

Results obtained by simulation are the same as shown by the present *Uppaal* study. By simulation we can prove that property 1 holds on all experiments and conclude that the airbag controller is able to fire at least once within a simulation run.
A violation of property 2 is never seen by process Observer_Prop2(), meaning that in case the micro controller wants to fire and firing is enabled by the external approver, the firing stage will fire.

3.3 Conclusion

This detailed system modeling and verification approach shows that it is feasible to obtain the previous acquired results by simulation in *MoDeST*. In particular we pinpointed the benefits and drawbacks of verifying properties by simulation. As a downside it can be stated that simulation does never cover the complete state space of some model, and as such it can be dangerous to purely base the results on verification by simulation.
Moreover by the simulation approach as being states in this chapter, simulation trace and reward variables can be used to analyze the model of concern.

Chapter 4

Simulink Stateflow

Stateflow is a graphical design and development tool for control and supervisory logic used in conjunction with *MatLab Simulink* [Mat05] which allows to visualize models and simulate complex reactive systems. Far more the system's behavior can be verified at any design stage. As *Stateflow* is part of the *MatLab Simulink* package models a variety of components from libraries can be incorporated into models and used as external stimuli for *Stateflow* diagrams.

Stateflow is using finite state machine theory, flow diagram notations, and state-transition diagrams all in the same *Stateflow* diagram. Flow diagram notation creates decision-making logic such as for loops and if-then-else constructs without the use of states.

This chapter gives an introduction to *Stateflow*, providing the notation used, and illustrating how models are interpreted by the *Stateflow* semantics. Moreover it strives for a feasibility analysis for modeling the *Detailed System Modeling and Verification* approach from chapter 3 in *Stateflow*. Special interest is put on some statistical figures like a firing distribution of the airbag controller when using random *Simulink* signals as input.

4.0.1 *Stateflow* and *Simulink*

The collection of all *Stateflow* blocks in the *Simulink* model is a machine (cf. [Mat05]). When *Simulink* is used with *Stateflow* for simulation, *Stateflow* generates an *S-function* (*MEX-file*) for each *Stateflow* machine to support model simulation. This generated code is a simulation target and is called the *sfun* target within *Stateflow*.

4.0.2 Finite State Machine Representations

A finite state machine is a representation of an event-driven (reactive) system. In an event-driven system, transitions are taken from one state (mode) to another prescribed state, provided that the condition defining the change is true.

A *finite state machine* (*FSM*) or *finite automaton* is a model of behavior composed of states, transitions and actions. A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment. The way transition labels can be composed and state actions can be executed is illustrated in figure 4.1.

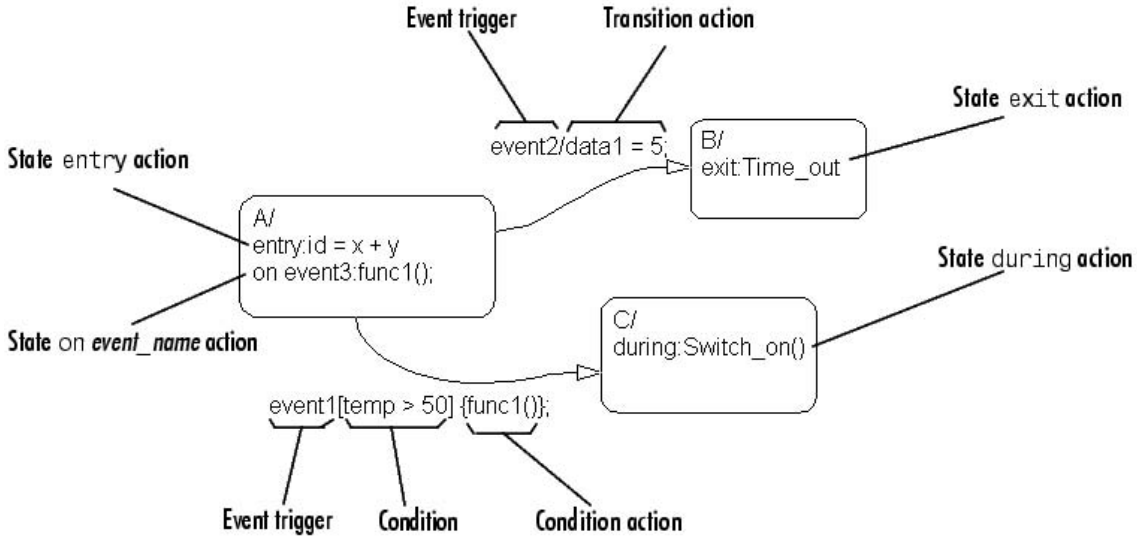


Figure 4.1: Overview of some basic *Stateflow* notations (cf. [Mat05]).

4.1 *Stateflow* Notation

4.1.1 *Stateflow* Diagram Objects

This part describes most of the graphical and non graphical objects in a *Stateflow* diagram along with the concepts that relate them. The following sample *Stateflow* diagram [Fig 4.2] displays a summary of the key graphical objects of a *Stateflow* diagram. Objects and examples described in following refer to this diagram.

States

A state describes a mode of an event-driven system. The activity or inactivity of the states dynamically changes based on events and conditions. Every state has a parent, where as in a *Stateflow* diagram consisting of a single state, that state's parent is the *Stateflow* diagram itself, also called the *Stateflow*

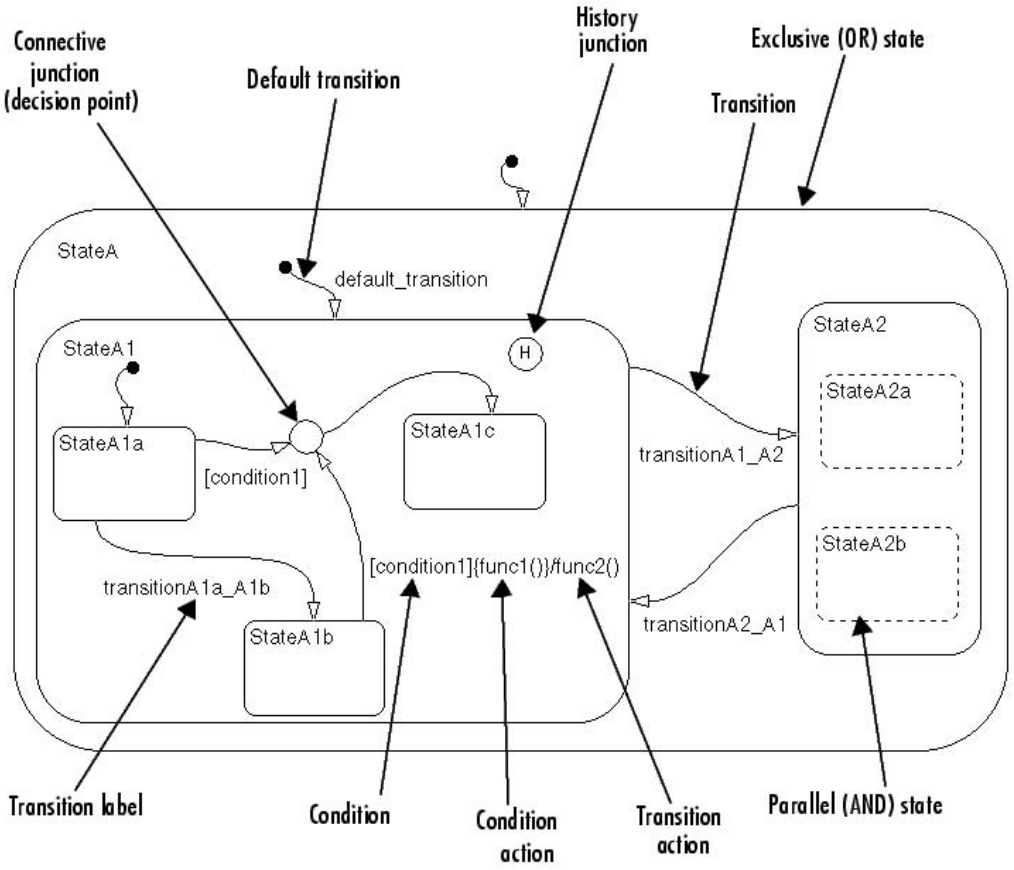


Figure 4.2: Summary *Stateflow* diagram revealing some of the key features like parallel composition, history junction, connective junction, et cetera (cf. [Mat05]).

diagram root. States can be placed within other higher-level states. In figure 4.2, **StateA1** is a child of **StateA**.

The decomposition of a state defines the way states are incorporated in the next level of containment. *Stateflow* provides two types of states, namely *exclusive* (OR), and *parallel* (AND) states. Exclusive (OR) states are used to describe modes that are mutually exclusive. A chart or state that contains exclusive (OR) states is said to have exclusive decomposition. A chart or state with parallel states has two or more states that can be active at the same time. A chart or state that contains parallel (AND) states is said to have parallel decomposition. Parallel (AND) states are displayed as dashed rectangles. The activity of each parallel state is essentially independent of other states. In the figure 4.2, **StateA2** has parallel (AND) state decomposition. Its states, **StateA2a** and **StateA2b**, are parallel (AND) states.

States can have **entry**, **during**, **exit**, and **on event** actions [Fig. 4.3]. The action language defines the types of actions you can specify and their associated notations. An action can be a function call, the broadcast of an event, the assignment of a value to a variable, and so on.

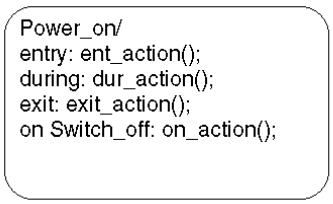


Figure 4.3: Possible events and function calls within a state (cf. [Mat05]).

Stateflow supports both *Mealy* and *Moore* finite state machine models. In the Mealy model, actions are associated with transitions, whereas in the Moore model they are associated with states.

Transitions

A transition is a graphical object that, in most cases, links one object to another. One end of a transition is attached to a source object and the other end to a destination object. The source is where the transition begins and the destination is where the transition ends. A transition label describes the circumstances under which the system moves from one state to another. It is always the occurrence of some event that causes a transition to take place. In figure 4.2 on the preceding page, the transition from **StateA1** to **StateA2** is labeled with the event **transitionA1_A2** that triggers the transition.

Default Transitions

Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy. For example, in figure 4.2, the default transition to **StateA1** resolves the ambiguity that exists with regard to whether **StateA1** or **StateA2** should be active when state A becomes active. In this case, when **StateA** is active, by default **StateA1** is also active.

Note that *history junctions* override default transition paths in super-states with exclusive (OR) decomposition and in parallel (AND) states, a default transition must always be present to indicate which of its exclusive (OR) states is active when the parallel state becomes active.

Events

Events drive the *Stateflow* diagram execution but are non graphical objects and are thus not represented directly in a *Stateflow* chart. All events that affect the *Stateflow* diagram must be defined. The occurrence of an event causes the status of the states in the *Stateflow* diagram to be evaluated. The broadcast of an event can trigger a transition to occur or can trigger an action to be executed. Events

are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Data

Data objects are used to store numerical values for reference in the *Stateflow* diagram. They are non graphical objects and are therefore not represented directly in a *Stateflow* chart. Data objects have a property called scope that defines the availability of the object for states.

Conditions

A condition is a Boolean expression specifying that a transition occurs, given that the specified expression is **true**. In the component summary *Stateflow* diagram [Fig. 4.2], **condition1** represents a Boolean expression that must be true for the transition to occur.

History Junction

A history junction records the most recently active state of a chart or super-state. If a super-state with exclusive (OR) decomposition has a history junction, the destination sub-state is defined to be the sub-state that was most recently visited. A history junction applies to the level of the hierarchy in which it appears. The history junction overrides any default transitions. In the summary *Stateflow* diagram [Fig. 4.2], the history junction in **StateA1** indicates that when a transition to **StateA1** occurs, the sub-state that becomes active (**StateA1a**, **StateA1b**, or **StateA1c**) is based on which of those sub-states was most recently active.

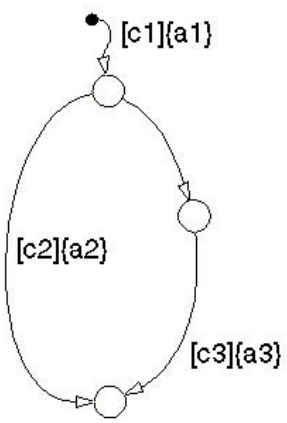
Actions

Actions take place as part of *Stateflow* diagram execution. The action can be executed either as part of a transition from one state to another or based on the activity status of a state. Transitions ending in a state can have condition actions and transition actions. In the summary *Stateflow* diagram [Fig. 4.2] the transition from StateA1b to the connective junction has condition action func1() and transition action func2().

Connective Junctions

Connective junctions are decision points in the system and of particular interest. They provide alternative ways to represent desired system behavior. In figure 4.2 on page 33, the connective junction is

used as a decision point for two *transition segments* that complete at **StateA1c**.
Transitions connected to junctions are called transition segments. Transitions, apart from default transitions, must go state to state. However, once transition segments taken completely to the next state, the accumulation of these transition segments taken forms a complete new transition. Example in figure 4.4 shows how connective junctions are used to represent the flow of an *if-else* structure accompanied by pseudo code.




```
if [c1]{
      a1
      if [c2]{
         a2
      }else if [c3]{
        a3
      }
}
```

Figure 4.4: Example of a nested if-else condition and its counterpart in *Simulink* (cf. [Mat05]).

4.2 *Stateflow* Semantics

The semantics of *Simulink*/*Stateflow* is rule based in a sense that transition-taking is deterministic. This can be seen by having a look at the semantics where e.g. connective junctions completely avoid the use of non-determinism by considering the angular position of the transition source. Integrating *Stateflow* models into *Simulink*, more expressiveness is gained. Following below, rule- and execution orders are provided that constitute the *Stateflow* semantics.

4.2.1 Event Execution

Execution of events occurs in two levels and only in response to the execution of a *Stateflow* chart. When starting the simulation, first the chart is updated, awakened for execution and second it is responding to events until all events are processed. Since *Stateflow* is single threaded, actions that take place on an event are atomic to that events, i.e. all activity caused by the event in the chart is completed before returning to activities that were taking place prior to reception of the event.
When an event occurs, it is processed from the top of the *Stateflow* diagram down through the hierarchy. At each level in the hierarchy any during and on event actions for the active state are executed, and completed, and a check for the existence of a transition among children of the state is conduced.

4.2.2 Chart Execution

The execution of charts is triggered by events originating from *Simulink*. A chart is inactive when it is first triggered by an event from the *Simulink* model and has no active states within. After the chart is executed and its initial trigger events from the *Simulink* model are completely processed, it remains active and goes to sleep. A sleeping chart has active states within it, but no events to process.

Active Charts

After a chart has been triggered the first time by the *Simulink* model, it is an *active chart*. When it receives another event from *Simulink*, it executes again as an active chart. If the chart has no states, each execution is equivalent to initializing a chart, otherwise the active children are executed. Parallel states are executed in the same order that they are entered.

Inactive Charts

When a chart is inactive and first triggered by an external event from *Simulink*, it first executes its set of default flow graphs. If this does not cause an entry into a state and the chart has parallel decomposition, then each parallel state is entered. If executing the default flow paths does not cause state entry, a state inconsistency error occurs.

Flow Graphs

The flow graph group is executed in the order of group priority until a valid transition is found. The default transitions group is executed first, followed by the inner transitions group and then the outer transitions group. Each flow graph group is executed with the following procedure:

1. Order the group's transition segments for the active state. An active state can have several possible outgoing transitions, which are ordered before checking them for a valid transition.

2. Select the next transition segment in the set of ordered transitions.

3. Test the transition segment for validity. If the segment is invalid, go to step 2.

4. If the destination of the transition segment is a state, do the following:

   (a) No more transition segments are tested and a transition path is formed by including the transition segment from each preceding junction back to the starting transition.

   (b) States that are the immediate children of the parent of the transition path are exited.

   (c) The transition action for the final transition segment of the full transition path is executed.

(d) The destination state is entered.

5. If the destination is a junction with no outgoing transition segments, do testing without any states being exited or entered.
6. If the destination is a junction with outgoing transition segments, repeat step 1 for the set of outgoing segments from the junction.
7. After testing all outgoing transition segments at a junction, back up the incoming transition segment that brought you to the junction and continue at step 2, starting with the next transition segment after the backup segment. The execution of the set of flow graphs is done when all starting transitions have been tested.

4.2.3 Transition Execution

Transitions play a large role in defining the animation or execution of a system. If the chart has exclusive (OR) states, its execution begins with the default transitions that points to the first active state in the chart. Any actions associated with the sources or destinations are related to the transition that joins them. The type of the source and destination is equally important to define the semantics.

Flow Graph Types

Before transitions are executed for an active state or for a chart, they are grouped by the following types:

- *Default flow graphs* are all default transition segments that start with the same parent.
- *Inner flow graphs* are all transition segments that originate on a state and reside entirely within that state.
- *Outer flow graphs* are all transition segments that originate on the respective state but reside at least partially outside that state.

Each set of flow graphs includes other transition segments connected to a qualifying transition segment through junctions and transitions.

Ordering Single Source Transitions

Transitions from a single source are ordered for testing according to the following three sorting guidelines, which appear in order of their precedence (first step is highest priority):

1. Transitions whose end points are attached to higher hierarchical levels (*Endpoint Hierarchy*) are placed first in testing order.

2. Transitions are ordered for testing according to the types of action language present in their *labels*. The order of precedence is
   (a) Labels with events and conditions
   (b) Labels with events
   (c) Labels with conditions
   (d) No Labels
3. Transitions are ordered for testing based on the *angular position of the transition source* on the surface of the originating object. Multiple outgoing transitions from states that are of equivalent label, source, and end point the hierarchy priority are evaluated in a clockwise progression starting at the upper left corner of the source state.

4.2.4 State Execution

States are either active or inactive. The following subsection describes the stages of state execution that take place between becoming active and becoming inactive.

Entering a State

A state is entered (becomes active) in one of the following ways:

- Its boundaries are crossed by an incoming executed transition.
- Its boundary terminates the arrow end of an incoming transition.
- It is the parallel state child of an activated state.

When specified, the state performs its entry action when it becomes active. The state is marked active before its entry action is executed and completed. The execution steps for entering a state are as follows:

1. If the parent of the state is not active, perform steps 1 through 4 for the parent first.
2. If this is a parallel state, it is checked that all sibling parallel states with a higher execution order are active. If not, perform all entry steps for these states first in the appropriate order of entry.
3. Mark the state active.
4. Perform any entry actions.
5. Enter children, if needed:
   (a) Execute the default flow paths for the state unless it contains a history junction.

(b) If the state contains a history junction and there is an active child of this state at some point after the most recent chart initialization, perform the entry actions for that child.

(c) If this state has children that are parallel states (parallel decomposition), perform entry steps 1 to 5 for each state according to its entry order.

6. If this is a parallel state, perform all entry actions that exist for the sibling state next in entry order.

7. If the transition path parent is not the same as the parent of the current state, perform entry steps 6 and 7 for the immediate parent of this state.

8. The chart goes to sleep.

Executing an Active State

Active states that receive an event that does not result in an exit from that state execute a during action to completion if a during action is specified for that state. An on event action executes to completion when the event specified occurs and that state is active. An active state executes its during and on event actions before processing any of its children's valid transitions. during and on event actions are processed based on their order of appearance in the state label. The execution steps for executing a state that receives an event while it is active are as follows:

1. The set of outer flow graphs is executed. If this causes a state transition, the execution of the state is stopped.

2. during actions and valid on event actions are performed.

3. The set of inner flow graphs is executed. If this does not cause a state transition, the active children are executed, starting at step 1. Parallel states are executed in the same order that they are entered.

Exiting an Active State

A state is exited and accordingly becomes inactive in one of the following ways:

- Its boundary is the origin of an outgoing executed transition.
- Its boundary is crossed by an outgoing executed transition.
- It is a parallel state child of an activated state.

The state is marked inactive after the exit action has executed and completed. The execution steps for exiting a state are as follows:

1. If it is a parallel state, and one of its sibling states was entered before, exit the siblings starting with the last-entered and progressing in reverse order to the first-entered.

2. If there are any active children, perform the exit steps on these states in the reverse order they were entered.

3. Perform any exit actions.

4. Mark the state as inactive.

4.3 Airbag Controller by Means of *Stateflow*

After having studied the theory of *Stateflow* it seems eligible to obtain a comprehension how the previous model from chapter 3 can be transformed into *Stateflow* successfully. Where attention was devoted to the verification of certain properties by simulation, we now focus on real-time simulation and thus exhaust the spectrum of available validation methods. Moreover, by capturing the times of deployment we will be able to derive some statistical measure reflecting the airbag deployment distribution.
Simulink has apart from basic circuit components a variety of means to address higher mathematical functionals that will be partly used.

4.3.1 *Simulink* Models

Environment

The environment signals [Fig. 4.5] that represent sensor values as seen by micro controller and approver are generated by a uniform random number generator from the *Simulink* library. Rand_Env_A and Rand_Env_B generate values out of interval [-0.4; 2.4] and [-0.4; 1.4] to assure that we obtain by the post processing round-function integers between [0; 2] and [0; 1]. Gates sample_A and sample_B (Zero-Order-Hold) sample the delivered data with a fixed frequency of 2 time units. In addition to the number generation it is possible to use user-defined signal-values from *MatLab workspace* that will be used later on.

ECU Controller

The main *Simulink* template [Fig. 4.6] holds all constants as being used in the *MoDeST* simulation. Besides this, a clock is contained that gives the model a notion of global time. A scope (Scope) allows to investigate the course of each signal par and post simulation.

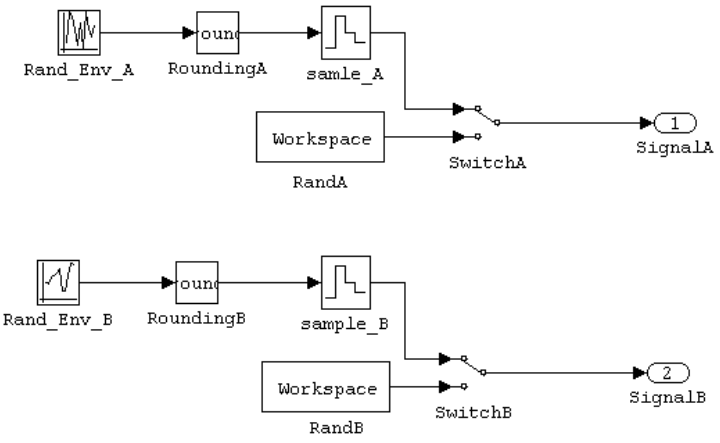


Figure 4.5: *Simulink* model of the environment that generates sensor values

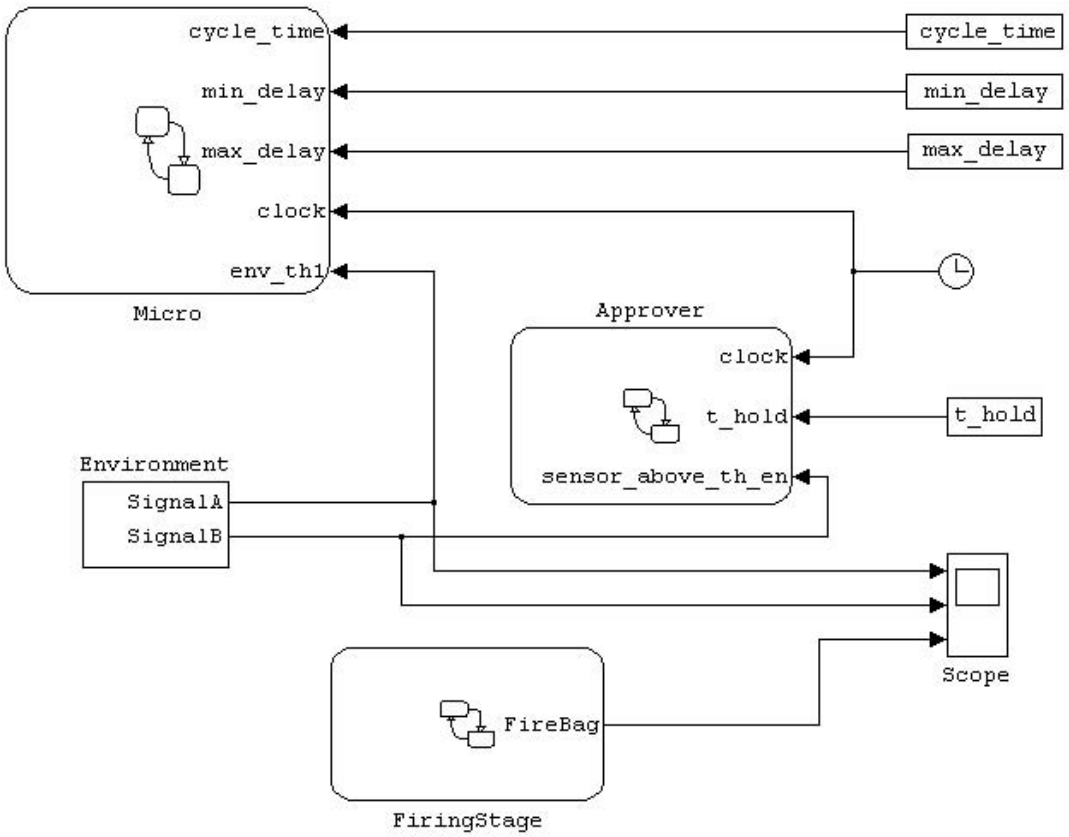


Figure 4.6: Summary *Simulink* model of the electronic control unit with *Stateflow* components FiringStage, Approver, and Micro.

4.3.2 *Stateflow* Models

Stateflow models are based on models from the previous *Uppaal* study [Aue05]. Since no option to define local clocks for each component and no option for clock reset is given, one has to incorporate time constraints of former clocks into transition guards. All timing constraints and functions from former local clocks have be incorporated into new guards. Due to existing non-disclosure agreements, we restrict on giving only the simulation results, rather than displaying the *Stateflow*models of the electronic control unit.

4.3.3 Simulation

Figure 4.7 illustrates the scope during one simulation run, where the first two lines represent random sensor values. In the last column the signal FireBag is shown that is triggered by FiringStage.
The input signals as being produced by the environment stem from the *Simulink* random generator. As a consequence, when repeating the experiments always identical outputs are obtained unless the seed is changed. This guarantees a better trace-ability of models for repeating runs.

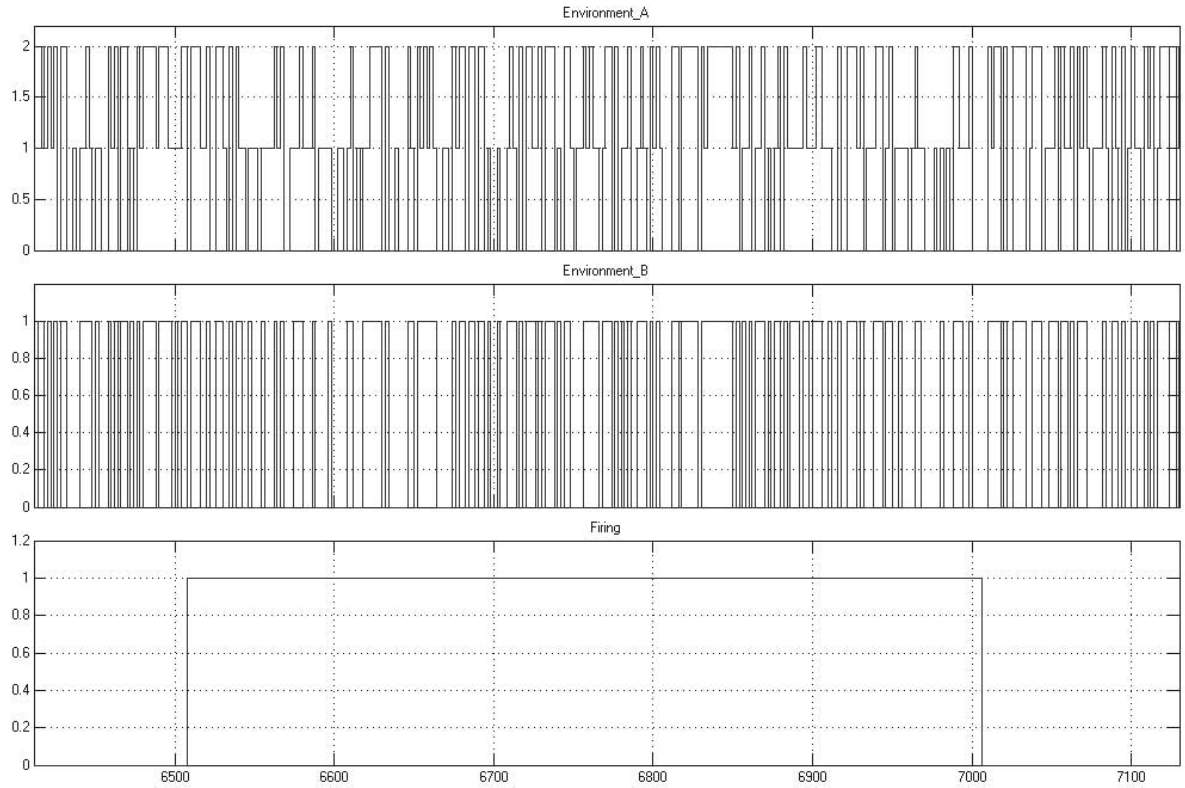


Figure 4.7: Simulation output showing signal values of environment A, B, and in the third row the deployment of the firing stage.

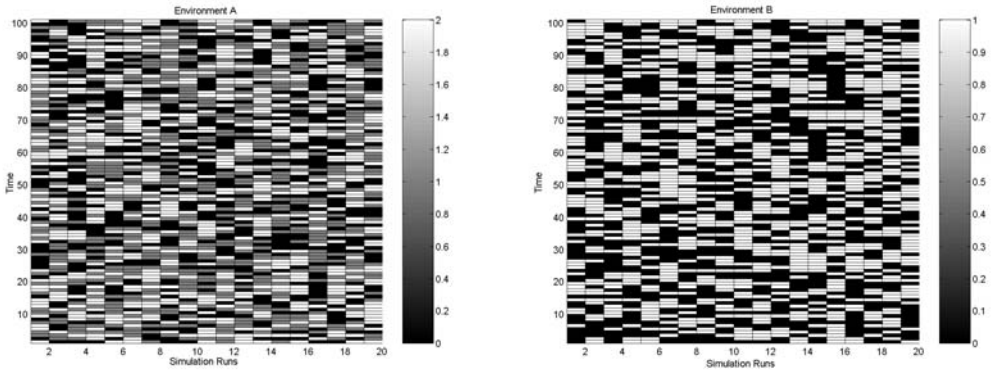


Figure 4.8: Sensor values as being produced by the environment for 20 simulation runs. EnvironmentA is a three-valued variable where EnvironmentB can only reach two values.

4.3.4 Statistical Analysis

Since our interest is also devoted to some statistical measures, the macro (*M-file*) shown in appendix C will guide the simulation and do the initialization of arrays. Results of 2.000 simulations are thus obtained that can be used to do the statistical analysis. To ensure real random sensor values for multiple simulations we pass on using the *random number generator* from the *Simulink* Library since using the same seed the exactly same environment signals are received.
Motivated by this observation, two two-dimensional arrays (RandA, RandB) are instantiated by real random values that are renewed every simulation run [Fig. 4.8]. Values from this arrays are sampled with a frequency of one random sample every 100 time units. This correlated to 100-times scaling on the X axis and in between the sensor signals retain their values. After each simulation, values from scope (structure ScopeData) are copied into data structure Results to retain them for further surveys.

The random variables in turn deliver the input to the airbag controller that decides upon some algorithm whether to deploy the airbag or not. The figure below depicts the firing distribution of the firing stage for 2.000 simulation runs. The Y-axis denotes the probability of deployment at a specific point in time. Note that the time axis (X-axis) is labeled with simulation time scaled by factor 100.

4.4 Conclusion

The *Stateflow* simulation of the ECU shows up a new approach of how the airbag control unit can be analyzed based on outputs generated by the environment. Valuable information can be gathered about a model by visualizing the correlation of sensor values generated by the environment, and the deployment of the firing stage. Moreover this chapter explains how to integrate *Stateflow* diagrams into the *Simulink* environment of *MatLab*.
The resulting distribution [Fig. 4.9] gained in the statistical section has a bell-like shape (normal distribution) with mean around 35. The gaps in the plot - where no data is available - originate from the fact that scope values are sampled with a sampling time of 500 time units (equal the cycle time of the

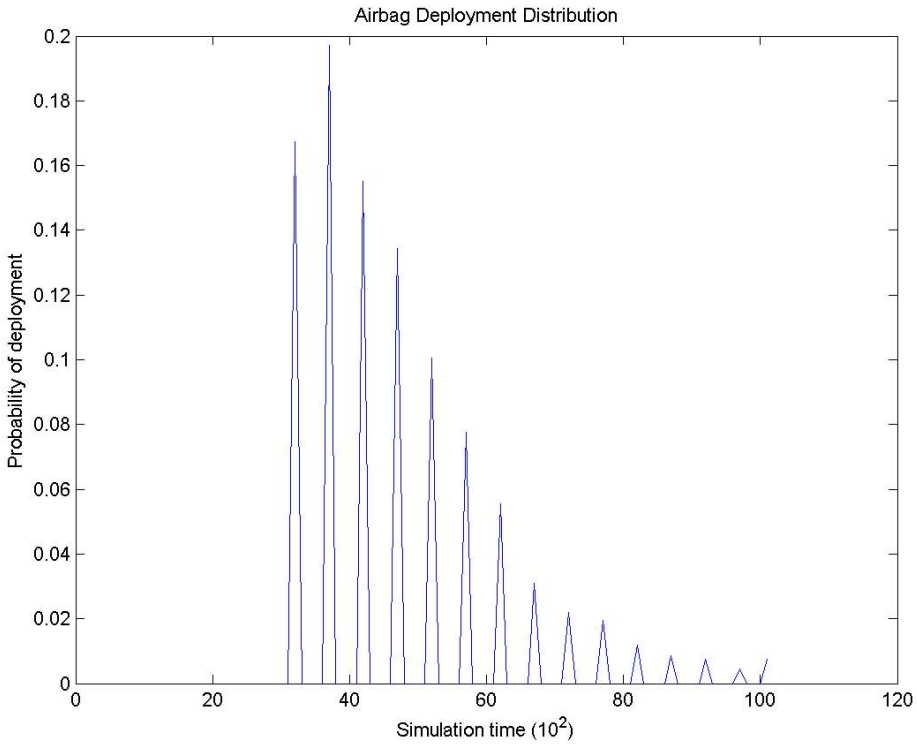


Figure 4.9: Distribution of airbag firing, acquired by running 2.000 simulations.

micro interrupt) to reduce the scope data. Notice that the earliest point of deployment is at 3.000 μs because only after that time the mechanism contained in the controller has reached the state where it allows firing.
Concluding, the *Stateflow* modeling gives new insights into the airbag controllers behavior. The visualization of results delivers a new aspect from a totally different point of view, still having in mind that the statistical analysis is rather a feasibility analysis than intended to bring up new insights into the topic.

Chapter 5

Markov Chain Analysis

In probability theory, a *Continuous-Time Markov Chain (CTMC)* is a stochastic process that enjoys the Markov property and takes values from elements of a discrete set called the state space. The Markov property states, that at any times $s > t > 0$, the conditional probability distribution of the process at time s given the whole history of the process up to and including time t, depends only on the state of the process at time t. In effect, the state of the process at time s is conditionally independent of the history of the process before time t, given the state of the process at time t. Using CTMCs one can efficiently compute failure behavior, given the fact that failure rates are known, by using steady state simulation.
This chapter covers on-demand failure analysis of the airbag control unit by analyzing the steady state behavior with focus on the safety integrity requirements. On demand in this context means that we focus on critical situations if the airbag fails to work properly at times of an airbag relevant crash, where failures besides the unintended deployment are more or less accepted if being repaired in time.

5.1 Assumptions

For determining the safety integrity requirements of the airbag control unit the following assumptions are made. The *overall operation time* of a single ECU is $T_0 = 9.000 hours$, which correlates to a usage of 15 years with an average duty of $600 \frac{hours}{year}$. The total number of ECUs to be produced is $N_0 = 30.000.000$ pieces. An airbag relevant crash occurs exponentially distributed with a crash rate of

$$\lambda_E = 4.0 \cdot 10^{-5} h^{-1}.$$

Once a failure of the airbag system is indicated, the driver is expected to visit the garage on average after 20 hours, which correlated to an exponentially distributed repair rate of

$$\mu_{repair} = 5.0 \cdot 10^{-2} \quad h^{-1}.$$

Variable Name	Value	Description
λ_E	4.0E-5	event (crash) rate
λ_{FI}	1.0E-9	rate of indicated failure
λ_{FNI}	1.0E-12	rate of not indicated failure
λ_I	1.0E-7	rate of the indication to fail
μ	0.05	repair rate
P_{TFD}	1.0E-9	probability of temporary faults upon process demand
λ_{TFE}	4,0E-14	rate of temporary failure on demand

Table 5.1: Failure and repair rates that were used in the Markov Chain analysis.

5.2 Modeling On-Demand System Failures

The considered Markov chain [Fig. 5.1] consists of five states with rates as shown in the table 5.1 below. Each state has transitions, labeled with rates. Transition from state OK to the fault state (F) is taken upon occurrence of an indicated fault, where as state FNI is entered on incident of a *failure not indicated*. The indication (warning lamp) permanently fails with a rate of λ_I in which case state IF (indication failure) is reached. Temporary faults are added using probability P_{TFD} (probability of temporary faults upon demand) which directly lead from state OK to state X, where the system fails to respond to external events (crash).

The rate of the λ_{TFD} transitions, that represent the rate of *temporary failure on processing demand* are obtained by directly multiplying the probability P_{TFD} by the event rate λ_E:

$$\lambda_{TFE} = P_{TFD} \cdot \lambda_E$$

5.3 Variables of Interest

Special interest is dedicated to the variables that follow below and in particular how precise they can be obtained by simulation.

- MTTF (Mean Time To Failure)
- $P(X)|_{t=9.000h}$
  Probability that one airbag fails after $t = 9.000$ hours of operation.
- EX_F
  Expected number of failing ECUs assuming N_0 are in operation.
- $P_{\geq 1}$
  Probability that at least one of the N_0 ECUs will fail during an assumed operation time of T_0.

5.4 Simulation and Results

Simulation of the *MoDeST* model as shown in appendix D.1 is carried out using variables TimeBeforeX ($P(X)|_{t=9.000h}$) to account for *mean time to failure* and PofXat9000 (read: probability of occurrence X at time 9000) which measure the probability that one airbag fails after 9.000 hours of duty. The variables were captured at *Instant of time* at 10^{15} time units to assure that the majority of simulation runs reached a stable failing state. For the following table [Tab. 5.2] 60 million batches are computed at a level of confidence of 95%. Lines labeled by (*) did not reach the confidence Interval during simulation.

Variable	Mean Value	Confidence +/-
MTTF	1,0107824732E09	9,9995125920E06
$P(X)\|_{t=9.000h}$	1,6666666667E-08	3,2666666667E-08(*)

Table 5.2: Figures MTTF and $P(X)|_{t=9.000h}$ received via simulation.

The probability of at least one ECU failing during the operation time of T_0 is calculated by

$$P_{\geq 1} = 1 - (1 - P(X)|_{t=9.000h})^{N_0}. \tag{5.1}$$

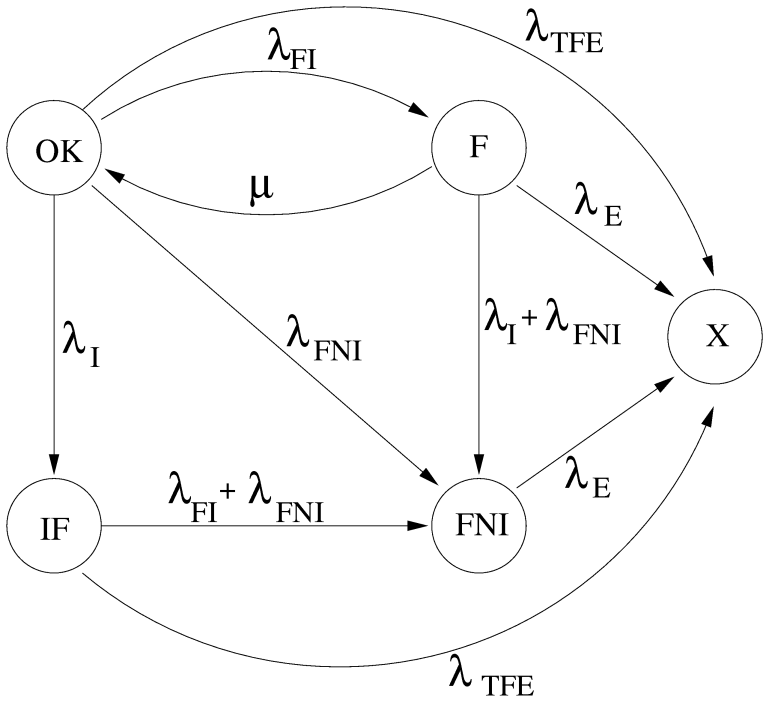


Figure 5.1: Model of the Markov Chain (CTMC) that is used for the on-demand safety analysis.

The expected number of failing ECUs is computed using

$$EX_F = P(X)|_{t=9.000h} \cdot N_0. \tag{5.2}$$

Since the result for $P(X)|_{t=9.000h}$ did not converge to a significant level, outcomes for EX_F and $P_{\geq 1}$ are expressed for mean, and values on the border of the confidence interval. Mean and values at the limit of the confidence interval that can be calculated using equation 5.1 and equation 5.2 are shown in the table below.

	Lower	Mean	Upper
EX_F	0,0100	0,5000	0,9900
$P_{\geq 1}$	1,00%	39,35%	62,84%

Table 5.3: Results for EX_F and $P_{\geq 1}$ for mean and border values.

In addition to the variables of interest, table 5.4 contains the total time spend in each of the states in addition to the failure measure. Reward variable measurement is *Instant of time* at $1 \cdot 10^{15}$ time units. This proofs that on average most of the time is spend in state IF, where the failure indication mechanism fails to work.

5.5 Analytical Approach

To verify the simulation results we use, the analytical approach by matrix exponentiation as being denoted in [Neu81]. A distribution $F(\cdot)$ on $[0, \infty)$ is a continuous *Phase-Type* distribution iff the distribution of the time until absorption to state $n+1$ in a CTMC of the type described above where n is finite. The pair $(\boldsymbol{\alpha}, \mathbf{T})$ is called the representation of $F(\cdot)$.
By translating the Markov model into a transition matrix $\mathbf{T}$, the probability of reaching state X can be calculated. Suppose a CTMC with $n+1$ states and a generator matrix $\mathbf{Q}$ given by

$$\mathbf{Q} = \begin{bmatrix} \mathbf{T} & T^0 \\ \mathbf{0} & 0 \end{bmatrix},$$

Variable	Mean Value	Confidence +/-
tOK	9,9990828441E06	2,5299634979E03
tF	1,9930258776E-01	7,1527508848E-04
tIF	9,9890425697E08	2,5273683107E05
tFNI	2,4996094361E04	6,3254611763E00

Table 5.4: Results obtained by simulation for the time spend in each of the states before entering state X over a simulation of 10^{15} time units.

where $\mathbf{T}$ is an $n \times n$ non-singular matrix satisfying, for all $1 \leq i \leq n$ that $T_{ij} \geq 0$ for $i \neq j$ and $T_{ij} \leq 0$ for $i = j$. Furthermore it holds that $\mathbf{Te} + T^0 = \mathbf{0}$ with $\mathbf{e}$ being a column-vector of size n whose elements are all 1. We assume that the states $1, \ldots, n$ are all transient, so that absorption into the state $n+1$ from any initial state is certain. The initial probability vector of the CTMC is given by $[\boldsymbol{\alpha}, \alpha_{n+1}]$, with $\boldsymbol{\alpha}\mathbf{e} + \alpha_{n+1} = 1$.

Let $\mathbf{Q}$ denote the matrix that is used to define a continuous Phase-Type distribution (PH) as the underlying CTMC. The Cumulative Density Function (CDF) of a PH with representation $(\boldsymbol{\alpha}, \mathbf{T})$ as F_{PH} is given by

$$\begin{aligned} F_{PH}(x) &= P(t \leq x) \qquad , \text{ for } x \geq 0 \\ &= 1 - \boldsymbol{\alpha} e^{\mathbf{T}x} \mathbf{e} \\ &= 1 - \boldsymbol{\alpha} \left(\sum_{n=0}^{\infty} \frac{\mathbf{T}^n \cdot x^n}{n!} \right) \mathbf{e} \end{aligned}$$

In the model of the on-demand system failures we use $\mathbf{Q}$ as generator matrix where $\sum$ denotes the sum of all row entries.

$$\begin{aligned} \mathbf{Q} &= \begin{bmatrix} \mathbf{T} & T^0 \\ \mathbf{0} & 0 \end{bmatrix} \\ &= \begin{pmatrix} -\sum & \lambda_{FI} & \lambda_I & \lambda_{FNI} & \lambda_{TFE} \\ \mu & -\sum & 0 & \lambda_I + \lambda_{FNI} & \lambda_E \\ 0 & 0 & -\sum & \lambda_{FI} + \lambda_{FNI} & \lambda_{TFE} \\ 0 & 0 & 0 & -\sum & \lambda_E \\ 0 & 0 & 0 & 0 & -\sum \end{pmatrix} \\ &= \begin{pmatrix} -\sum & \lambda_{FI} & \lambda_I & \lambda_{FNI} & \lambda_{TFE} \\ \mu & -\sum & 0 & \lambda_I + \lambda_{FNI} & \lambda_E \\ 0 & 0 & -\sum & \lambda_{FI} + \lambda_{FNI} & \lambda_{TFE} \\ 0 & 0 & 0 & -\sum & \lambda_E \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{aligned}$$

We identify

$$\mathbf{T} = \begin{pmatrix} -\sum & \lambda_{FI} & \lambda_I & \lambda_{FNI} \\ \mu & -\sum & 0 & \lambda_I + \lambda_{FNI} \\ 0 & 0 & -\sum & \lambda_{FI} + \lambda_{FNI} \\ 0 & 0 & 0 & -\sum \end{pmatrix}$$

and

$$T^0 = \begin{pmatrix} \lambda_{TFE} \\ \lambda_E \\ \lambda_{TFE} \\ \lambda_E \end{pmatrix}.$$

The initial probability mass is completely contained in α_1, since we want to investigate a sound device upon start, and check its failure probability over time. So the resulting vector representing the initial state probability is

$$\boldsymbol{\alpha} = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}.$$

By putting in the corresponding rates into matrix $\mathbf{T}$, the time of absorption of the the finite Markov process at hand can be computed [App. D.2], and accordingly the probability of reaching state X.

$$\mathbf{T} = \begin{pmatrix} -.00000010100104 & .000000001 & .0000001 & .000000000001 \\ .05 & -.050040100001 & 0 & .000000100001 \\ 0 & 0 & -.00000000100104 & .000000001001 \\ 0 & 0 & 0 & -.00004 \end{pmatrix}$$

By using PH with representation $(\mathbf{T}, \boldsymbol{\alpha})$ the distribution is generated via matrix exponentiation. Figure 5.2 reveals a distribution for reaching state X. The exact value at 9.000 hours of operation is

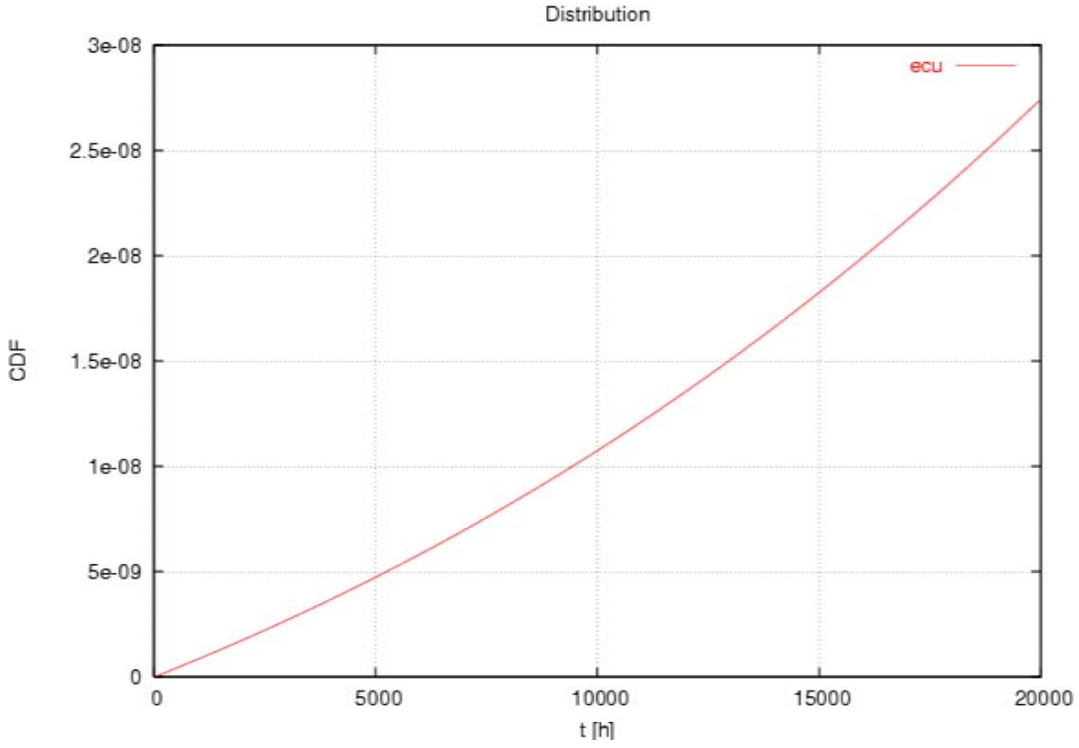


Figure 5.2: Phase type distribution with representation $(\boldsymbol{\alpha}, \mathbf{T})$ showing the relation between the time on the abscissa and and the resulting probability of reaching state X.

$p = 9.425010E - 09$. Using equations 5.1 and 5.2 on page 49 we compute the failures based on the analytical approach as shown at table 5.5. This proves, that assuming 30 million ECUs to be build less that one will fail over an time of duty of 9.000 hours under terms of normal usage.

Variable	simulation (mean)	matrix exponentiation
$P(X)\|_{t=9.000h}$	1.66666E-08	9.425010E-09
EX_F	0,5	0,282750
$P_{\geq 1}$	39,3469%	24,6292 %

Table 5.5: Comparison of the results obtained by simulation and matrix exponentiation.

5.6 Conclusion

Simulation showed that for the selected rates the on-demand safety failures are within acceptable ranges considering the fact, that less then 1 (0,5 piece) ECUs fail out of 30 million produced devices within time T_0. It is worth noticing that the confidence interval of 95% is not achieved within 60 million simulations for $P(X)|_{t=9.000h}$ and hence no accurate forecast can be given.
Considering that a long time on average is spend in state IF where the indication fails, the security could be substantially improved by modifying the indication procedure. Mean time to failure is with $1,011 \cdot 10^9$ hours of operation far beyond the necessary requirements of 9 000 hours.
Rare event simulation could be used to achieve an even better simulation outcomes. The number of failing devices obtained by *matrix exponentiation* approved simulation results to be close to the actual failures and revealed a nice way to compute the failure distribution, thus obtaining an illustrative overview.

Chapter 6

Fault Tree Analysis

Fault Tree Analysis (FTA)[WGRH81] is a technique to determine reliability and safety of complex systems developed 1962 by *Bell Telephone Laboratories* for the U.S. Air Force *Minuteman* system. Fault tree analysis is one of many symbolic "analytical logic techniques" found in operation research and in system reliability by working in the “failure space” and looking at system failure combinations.

This chapter is a feasibility check of *Fault Tree Analysis* by simulation and its implementation in *MoDeST*. Of interest is the way fault trees can be modeled. Special emphasize is devoted to grouping of components in a natural way, i.e. that all points of failure that can occur within a device are modeled as such. This is a quested and desirable feature because the interrelation and dependability is completely lost in conventional FTA models. Towards the end, simulation results are compared to results obtained by commercial fault tree tools, like *Fault Tree +* [Fau05].

6.1 Representation of Events

Fault trees are built using gates and events (blocks). The two most commonly used gates in a fault tree are AND (⍝) and OR (⍝) gates. Other gates to account for *majority votes* or *priority* are neglected since they are not used in the model at hand.

As an example, consider two events comprising a *top event (TE)*. If occurrence of either event causes the top event, then these events are connected using an OR gate. Alternatively, if both events need to occur to cause the top event to occur, they are connected by an AND gate.
For better understanding, imagine a simple fault tree consisting of seven components [Fig. 6.1], whereas BE1 to BE4 are *base events (BE)*. In case of the occurrence of a Top Event, the error is propagated from the base events by Gate2 and Gate3 such that either base event BE1 or BE2 and BE3 or BE4 contribute to the system failure and final top event occurrence.

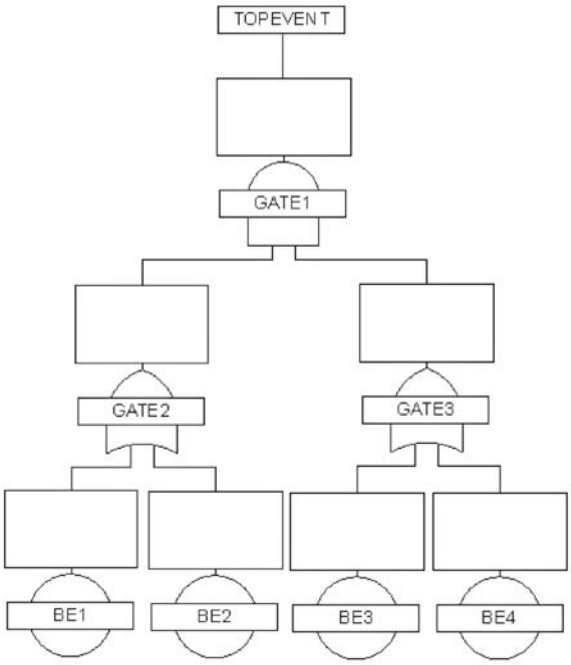


Figure 6.1: Fault tree example with four bottom events which are connected to the top event using AND- (Gate1) and OR-gates (Gate2, Gate3).

6.1.1 Events

Gates in a fault tree are logic symbols that interconnect contributory events and conditions. The most prominent to name are *fixed rate* and *fixed probability* events. An event (or a condition) block in a fault tree can have a probability of occurrence or a distribution function, where probabilistic events are time insensitive in contrary to distribution functions.
Fixed probability events specify unavailabilities which do not vary over time, modeling failure probabilities per demand. Opposing to that, fixed rate events are used to represent events whose failures are immediately revealed to the system. The underlying failure distribution can be exponential, with a constant failure rate, but also other distributions like Erlang, Weibull, or others are possible.
An approach to model these events in *MoDeST* is depicted in figure 6.2 for the exponential distribution and figure 6.3 for the fixed probability.

6.1.2 Probabilities of mixed Events

Before starting with the fault tree it seems desirable to have some notion of how probabilities of events is propagated when using gates. As such consider a fixed rate event that is exponentially distributed with mean $1/\lambda$ and a fixed probability event with probability p. X and Y are in the following random variables that define the resulting probabilities P_{exp} and P_{prop}

```
int EventBE1;

process BE1(){

  clock t;
  float ExpBE1;

  {= ExpBE1=Exponential(rateBE1) =};
  when (t==ExpBE1) {= EventBE1=1 =}

}
```

Figure 6.2: Exponential event BE1 that occurs with rate $r = rateBE1$.

```
Int EventBE2;

process BE2(){

  tau;
  //#FTA [ 1E-2: EventBE2 ]
  palt{
    :99: tau
     :1: {= EventBE2=1 =}
  }

}
```

Figure 6.3: *MoDeST* code of a probabilistic base event with probability $Q = 1.0E - 2$. Notice that this *MoDeST* construct is generated by *GEMA* on input `#FTA [ 1E-2: EventBE2]`

```
int EventGate1=0;

process Gate1(){

  when (EventBE1==1 && EventBE2==1)
    {= EventGate1=1 =}

}
```

Figure 6.4: *MoDeST* implementation of an AND gate. Notice that the guard can be extended easily to fit arbitrary many sub events.

$$P_{exp}[X \leq t] = 1 - e^{-\lambda t}$$

$$P_{prob}[Y] = \begin{cases} p & \text{, if Y} = \ddagger \\ 1 - p & \text{, if Y} = \dagger \end{cases}$$

where † and ‡ are mutually exclusive events. The resulting probabilities by interrelating P_{exp} and P_{prop} with AND, and respectively OR are

AND gate

$$\begin{aligned} P[X \leq t \wedge \ddagger] &= P[X \leq t] \cdot P[\ddagger] \\ &= \left(1 - e^{-\lambda t}\right) \cdot p \end{aligned}$$

OR gate

$$\begin{aligned} P[X \leq t \vee \ddagger] &= P[X \leq t] + P[\ddagger] - P[X \leq t \wedge \ddagger] \\ &= \left(1 - e^{-\lambda t}\right) + p - \left(1 - e^{-\lambda t}\right) \cdot p \\ &= (p-1) \cdot e^{-\lambda t} + 1 \end{aligned}$$

To obtain a clear and descriptive perception about the gates denoted above, the resulting distributions are plotted in figure 6.5.

6.2 Simulation and Results

A sub tree of the airbag control unit modeled is shown in appendix E that accounts for general hardware failures of the μC leading to incorrect processing. This branch has probabilistic (fixed probability) and exponential (fixed rate) events.

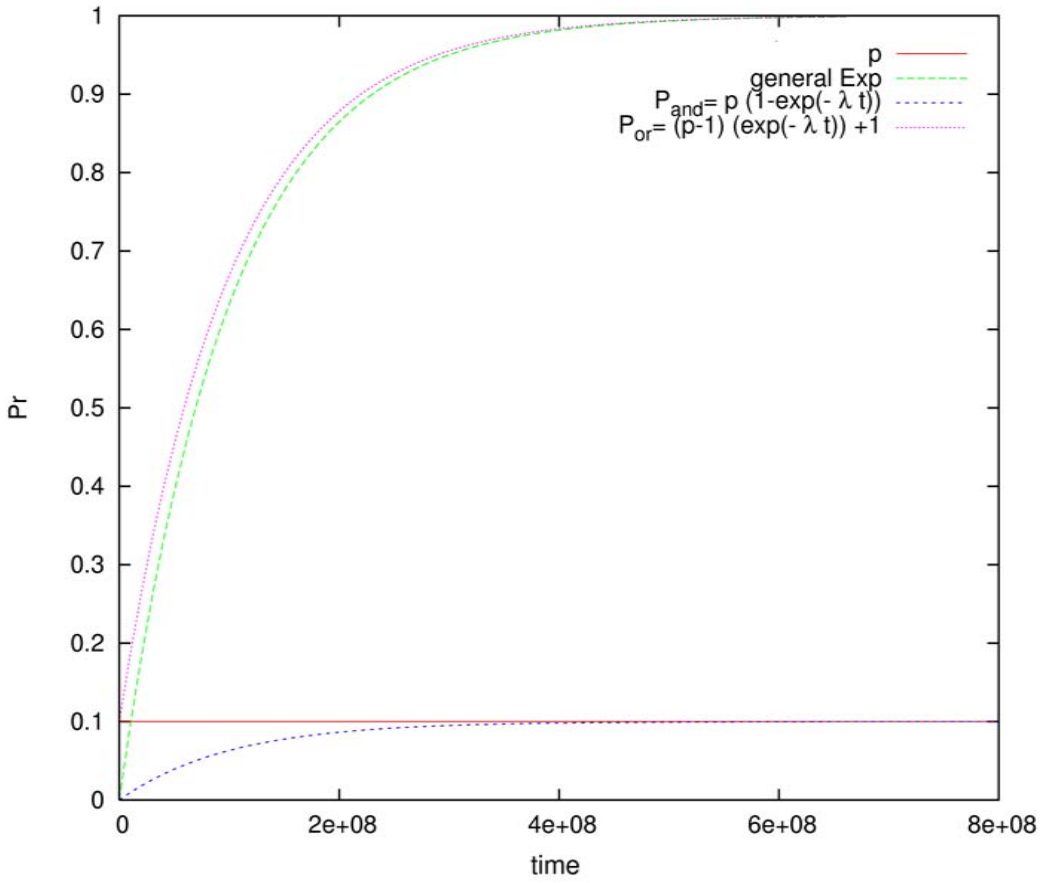


Figure 6.5: Figure showing the distribution of a general exponential function generalExp (green) with parameters λ=1.0E-8 and probability function P_{prop} (red) with p=10%. The resulting distribution using an AND-gate is denoted as Pand (blue) where as the or gate distribution is labeled by Por (purple). The constant probability P is time insensitive and hence parallel to the axis of abscissae.

6.2.1 *Fault Tree +*

The fault tree of concern is evaluated [Fig. 6.6] on first hand using *Fault Tree +* version 11, available as unlicensed demo version [Fau05]. The evaluation time used to compute the time insensitive exponential function is one time unit, i.e. the exponential distributed base events are evaluated at time one. The predicted probability of top event occurrence is $1.9702E-10$. Beyond measuring the system failure at 1 time unit, values to justify breakdowns over an airbag's life-cycle at 7 500, 9 000, and 12 000 are used. Results are shown in table 6.1.

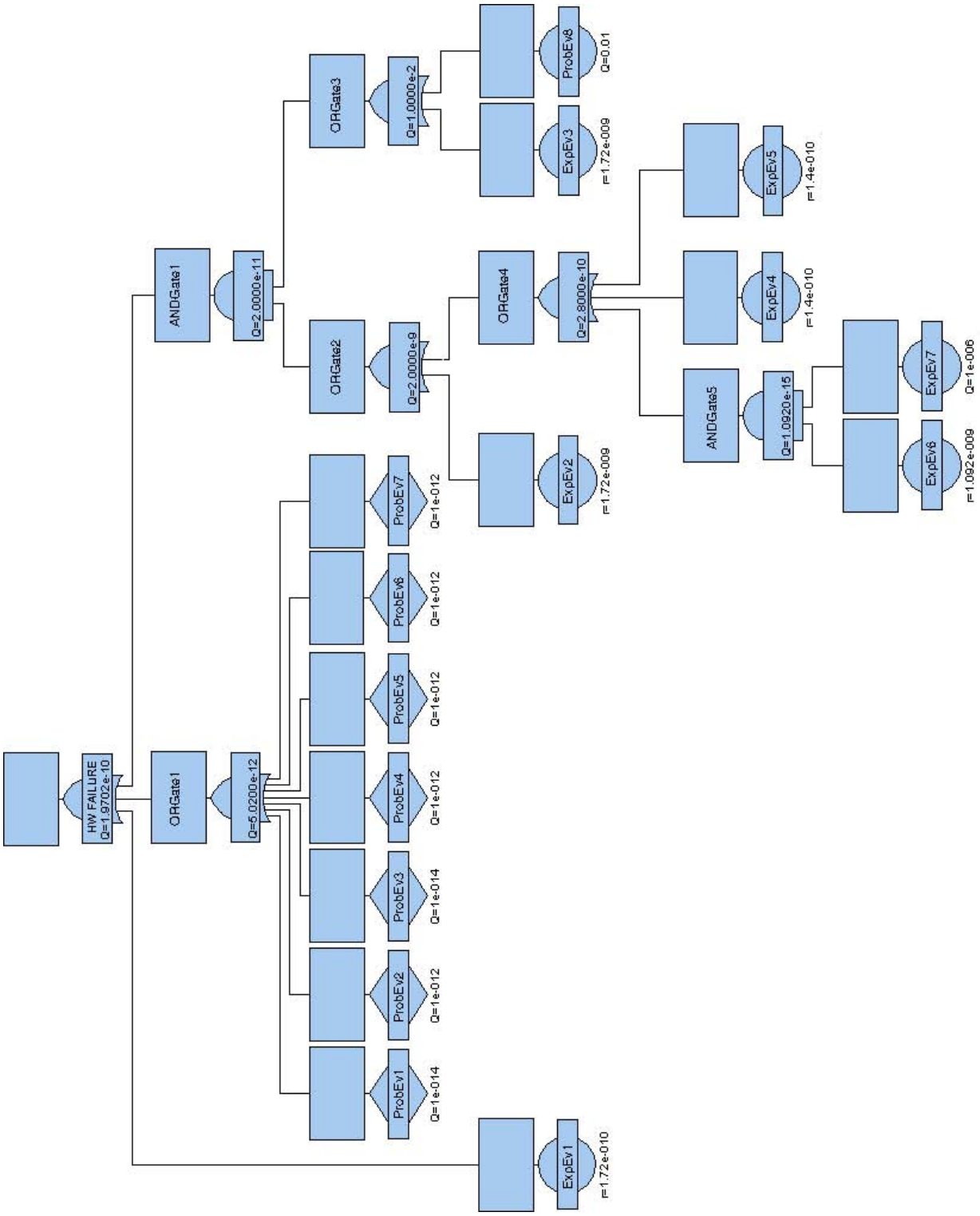


Figure 6.6: Clipping of the fault tree as used in *Fault Tree +*. The resulting TOP EVENT probability by evaluation over one time period is $1.9702E-10$.

time	FT+ Results	*MoDeST* Results	Confidence +/-	Batches	Process Time
1	1.970200E-10	5.67350E-09[1]	3.93153E-09	1 410M	103.66h
7 500	1.440195E-06	1.41326E-06	1.23754E-07	355M	23.74h
9 000	1.728278E-06	1.73495E-06	1.50725E-07	293M	21.36h
12 000	2.304490E-06	2.23000E-06	1.93542E-07	228M	16.54h

Table 6.1: Comparison of analytical and simulation results for different time values.

6.2.2 *MoDeST*

Simulation

The fault tree used for FTA analysis via *Fault Tree +* is identical to the one used for simulation to have posterior comparable results. To ease the work load a *GEMA* keyword (#FTA) that can be used for adding probabilistic errors into the model.
The simulation is on the first hand done over reward variable **Top-Event** measuring the top events seen at *Instant of Time* at one time unit. The confidence level is at 95% with a relative confidence interval of 0.1. Due to the very low rates top events show up very rarely. As results failed to converge when evaluating the exponential function at time one, they converge when increasing times. For more expressive values, time units are adopted to satisfy a *general airbag's life-cycle* under light (7 500h), normal (9 000h) and heavy (12 000h) usage (operation hours in parenthesis).

In addition, variable **TimeTE** is added that counts the mean time until the top event occurs, which is using the same reward variable than experiments before. By this, one can check to what extent probabilistic events contribute to the overall failure.

Results

Simulation and *Fault Tree +* results are captured in the table 6.1. For instant-of-time at one time unit, simulation results do not converge even when simulating over the maximal batch size. Attempts to rescale times did fail to reach confidence too, since *MoDeST* does discrete event simulation which is indifferent towards the time scale used.
For higher system lifetimes, the results of simulation correspond to the analytical results measured by *Fault Tree +*.

Table 6.2 summarized the time until the occurrence of a top event. Since the numbers are relatively small one can conclude that the number of on-demand errors outperforms the exponential failures, that occur over time. Consequently we can stress, that under normal usage (9 000 h) the investigated part is is failing on average at 7.8E-3 hours (28 seconds).

[1] results did not converge

time	TimeTE	Confidence +/-
7 500	5.4485575908E-03	5.4456679875E-04
9 000	7.8003129775E-03	7.7978655632E-04
12 000	1.3119907569E-02	1.3116989964E-03

Table 6.2: Simulation results for time passed until the occurrence of a top event.

6.3 Conclusion

The analysis provided here shows how fault trees can be modeled in *MoDeST*, where especially grouping of components into logical units can be done. This approach is motivated by the fact, that devices often have more than one point of failure, e.g. 1st the failure of incorrect voltage supply and 2nd the probability that incorrect voltage is not detected in the system, that should be modeled in the same fault tree branch.
By defining a preprocessor keyword much of the modeling activity is removed and system models are more structured and as such better readable. As a draw back it can be stated that when having small system lifetimes, the times required for simulation are extremely high. Like in the example at hand for one time unit of operation approximately 5 billion batches would be needed to reach a confidence at the desired level. This is a general downside of simulation in contrary to analytical computation.

Chapter 7

Fault Tree Generation

Economical development and quality of life depend to an increasing extent on a more and more complex technical infrastructure. This includes all means of technology, transportation, industry processes, and energy production. Such systems often carry a significant lethal potential while the prevailing tolerance in society is decreasing. With the entering of these models in the industrial process, a high demand on adequate and possibly automated analysis techniques come along, leading to a diversity of tools for different classes of models [Har87].

Fault tree generation establishes such an attempt to automate failure analysis of complex systems by deriving fault trees out of the behavior model via simulation. This chapter's focus is on the automated generation of fault trees and the way minimal cut sets can be gained via simulation in *MoDeST*.

7.1 Preliminary Concepts

7.1.1 Binary Decision Diagrams (BDDs)

A *Binary Decision Diagram (BDD)* is a directed acyclic graph representation of a Boolean function, where each non-terminating vertex is labeled by a variable. It has in addition two directed edges, one is a zero variable assignment and the other an one-edge. A BDD is derived by reducing a binary decision tree, which represents the recursive execution of *Shannon's* decomposition [Bry87]. In contrast to binary decision trees, a BDD's children node can have multiple parent nodes. By using BDDs the cut-sets can be computed efficiently.

7.1.2 Minimal Cut Sets (MCSs)

Among various fault tree analysis methods *minimal cut sets* (MCSs) are playing a central role in the assessment of fault trees. Minimal cut sets define the "failure modes" of the top event and are usually

obtained via fault tree evaluation [WGRH81]. A minimal cut set is the smallest combination of component failures which, if they all occur, will cause the top event to occur.
MCSs are the basis for qualitative analysis representing the minimal set of component failure that cause the failure of the whole system. They can also be used in the quantitative analysis which determines the probability of the top event occurrence for a given fault tree [TD03].

7.2 Fault Tree Generation

The computation effort to determine the qualitative and quantitative analysis of FT can exponentially grow by the number of *base events (BE)*. To lower computation costs, the FT can before computation be divided into modules according to the *divide and conquer* principle. One often differentiates between *stochastic independent sub trees* (*SIST*), logical combinations of components, and predefined modules [Buc00]. A *SIST* is sub tree whose nodes have no edges to vertices outside this sub tree and which consists solely of stochastic elements that can be evaluated independently from probabilistic base events.
For doing the quantitative analysis, a module can be analyzed independently from the rest of the tree and afterward being treated like a new BE with the computed failure behavior. For the qualitative FTA evaluation, modularization can reduce the complexity drastically. Sub tree modules often reflect parts in real systems and hence the failure behavior or the impact of modules on prime implicants leads to a better system understanding.

The model which will be used to investigate fault tree generation is the simple model of the Airbag Controller from the Scenario Analysis [Ch. 2]. Focus on the first hand is on adding probabilistic error events, and in the 2^{nd} run solely stochastic components that contribute to the SIST are added. Fortunately, since exponentially distributed errors are independent from the rest, they can be simulated disjoint. Up to here no assumption about how to continue after the occurrence of an error is made, since it is unclear how sensitive the model will react on failures. So the simulation is simply stopped and further investigations are kept as a open quest.

7.2.1 Representation of Probabilistic Errors

Data for the fault tree analysis can be gained by artificially inserting errors into the simulation model (*MoDeST* sources). As noted in [BHB+04] we differentiate the error behavior by the following preprocessor commands:

Actions

Actions can be manipulated by delay(a,p,t,clk) and stuck(a,p). This causes on the one hand to delay action a with probability p for t time units or generates a infinite delay for process a with probability

p. clk refers to any clock present in the model. In addition, one could think of randdelay to adopt for a random action delay in the future but for grasping the principles of fault tree generation the two error notions for actions should suffice. The preprocessor keywords delay(a,p,t,clk) and stuck(a,p) are explained in detail in appendix A.

Variables

For imitating error behavior of float variables we reserve noise(x,p,n) and rand(x,p,n) to account for noisy and random values. Noise is defined as a random fluctuation of magnitude n around a predefined value of variable x that occurs with probability p. With the keyword rand a random value of magnitude n is invoked and assigned to variable x with probability p. Note that no restriction to purely positive values is made, although this might be desirable for some models. Especially negative values can cause some models to deadlock, if not being considered during design phase.

In addition to the artificial errors inserted into the model, errors have to be labeled with an unique naming in order to relate the error definition in terms of "when does it occur?" to the corresponding error event "which component?" and the top event "what happens next?". This is achieved by defining exceptions to the corresponding error. All thrown exceptions are caught and lead to an immediate simulation abort.

7.2.2 Representation of Exponential Distributed Errors

When dealing with exponential distributed errors the preprocessor commands from before have to be modified by replacing the probability p by a rate λ and develop a mean to measure times within each process.

Actions

For simplicity we stick close to the naming for probabilistic errors and define delayexp(a,t,lambda) and stuckexp(a,lambda) as the corresponding exponential versions of the former used definitions. From the semantic side delayexp and stuckexp are similar with the only difference being that the simulation in the first case can be continued after time t passed.

Variables

Insertion of variable errors is done by noiseexp(x,lambda,n) and randexp(x,lambda,n) to account for noisy and random values. Due to the fact that the new defined processes need to be incorporated and run in parallel with other processes, a *GEMA* translation is not possible in one shot.

7.2.3 Representation of Nominal Faulty Behavior

Another way to start the fault tree generation is by adding nominal errors into the simulation. That means, that components contain explicit description of nominal and faulty behavior [MMT00] possibly stored in a database. This behavior is in turn described using a set of predefined constraints, relating the local variables of a component to possible failures. For instance a valve model may have three behavior modes, namely “ok”, “stuck open”, and “stuck closed”, representing the nominal behavior of two different failure modes.

7.3 Simulation of the Failure Model

Four errors are inserted into the model according to table 7.1 that represent probabilistic failures. No reward variables are needed since the resulting simulation traces of 10.000 batches is used for identifying the occurrence of actions before error. A clipping of the simulation trace is illustrated in figure 7.2.

	Error Type	Prob	Values	Delay
enable	RAND	10	[0,1]	
Event1	DELAY	10		5
Event2	STUCK	4		
Event3	DELAY	5		19

Table 7.1: Probabilistic failures which are inserted into the behavior *MoDeST* model.

The stochastic error side (SIST) of the model is represented by two exponentially distributed errors [Tab. 7.2] for the delay of in process FrontBag and BeltTensioner. Both failures are of type noise with $\lambda = 1E-4$ for BeltTensioner and $\lambda = 5E-5$ for FrontBag. By this, a random delay of magnitude 30 for BeltTensioner and 100 for FrontBag is induced. The absolute value of noise is chosen to be half of the predefined delay.
Since the stochastic failure model is independent from probabilistic one, the failure simulation of both models can simply be merged. Stochastic errors are added into the SIST (*Stochastic Independent Sub Tree*) for the fault tree model. The complete *MoDeST* sources can be found in appendix [G].

	λ	noiseless delay	noise magnitude
BeltTensioner	1E-4	60	30
FrontBag	5E-5	200	100

Table 7.2: Exponential point of failures with corresponding rates and parameters used for simulating the *MoDeST* model.

7.4 Results

The following preliminary failure diagram [Fig. 7.1] is generated out of the simulation trace, where $\overline{event}$ denotes a failure of action or variable $event$. It is extracted from the simulation trace and reveals that for failures enable_error and Event2_error two different runs are possible. For clarity only the branches that will contribute to the minimal cut set are considered.

The failure diagram gained by simulation is now manually inserted into *Fault Tree +* for doing the top event analysis. Error events enable_error and Event2_error are expressed in two weighted Boolean formulas according to the simulation likelihood. By doing the fault tree analysis in *Fault Tree +* the top event occurrence is 26.78%. The exponential failures are directly inserted into the SIST branch of the fault tree with their respective failure rates, since they are not categorized as on-demand failures.

7.5 Conclusion

As shown by this chapter, fault tree generation is feasible by simulation, although it is not appropriate since the state space is not exhaustively explored for all states fulfilling some condition. For complex systems with small exponential rates it is still feasible by increasing the number of batches. Although there can still be some states or traces not considered, this would hardly contribute to the probability of the top event occurrence since in that case the probability would be very small.

It seems desirable to investigate and especially automate the process of fault tree generation as described in this chapter. In particular with respect to more complex models, an automated generation is essential. For the FT generation most of the conversion from the simulation trace into the preliminary failure diagram and from there into a fault trees is done by hand although this could be done using some appropriate parsing mechanisms.
Towards the complexity of a model, a more realistic model is desirable in which a mechanism is provided to recover from induced errors. By this it becomes feasible to observe the model under errors and identify bottle-necks within the model.

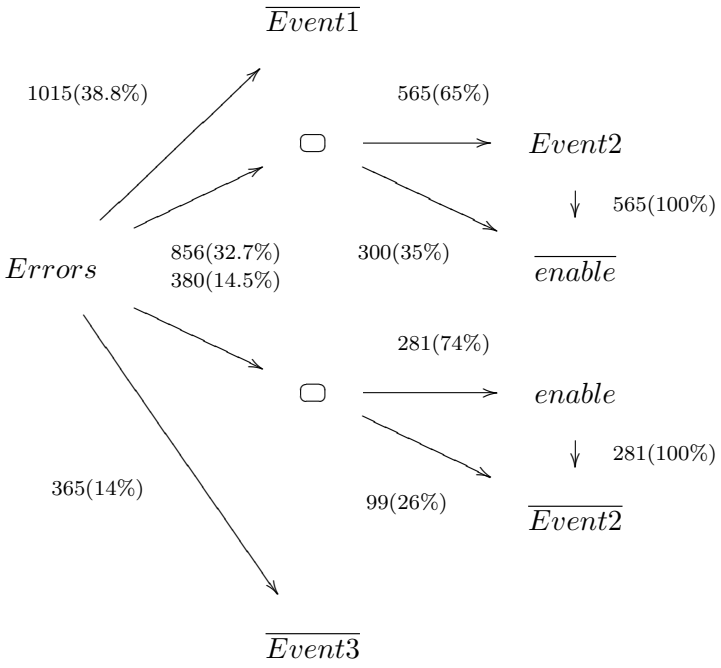


Figure 7.1: Preliminary failure diagram showing the occurrence of failures with corresponding frequencies (probabilities in parenthesis) that contribute to the minimal cut set.

```
###Simulation Errors for Event2###
  4.53525912    Event1_error

###Simulation Errors for Event2###
  0.00000000    Event1Sync
  0.00000000    SetEnabled
  0.00000000    Event2_error

###Simulation Errors for Event3###
  0.00000000    Event1Sync
  0.00000000    SetEnabled
  0.00000000    Event2Sync
  87.00000000   Event1Sync
  87.19817358   Event3_error

###Simulation Errors for enable###
  0.00000000    Event1Sync
  0.00000000    enable_error
```

Figure 7.2: Clipping of simulation runs showing the pre- error execution of actions.

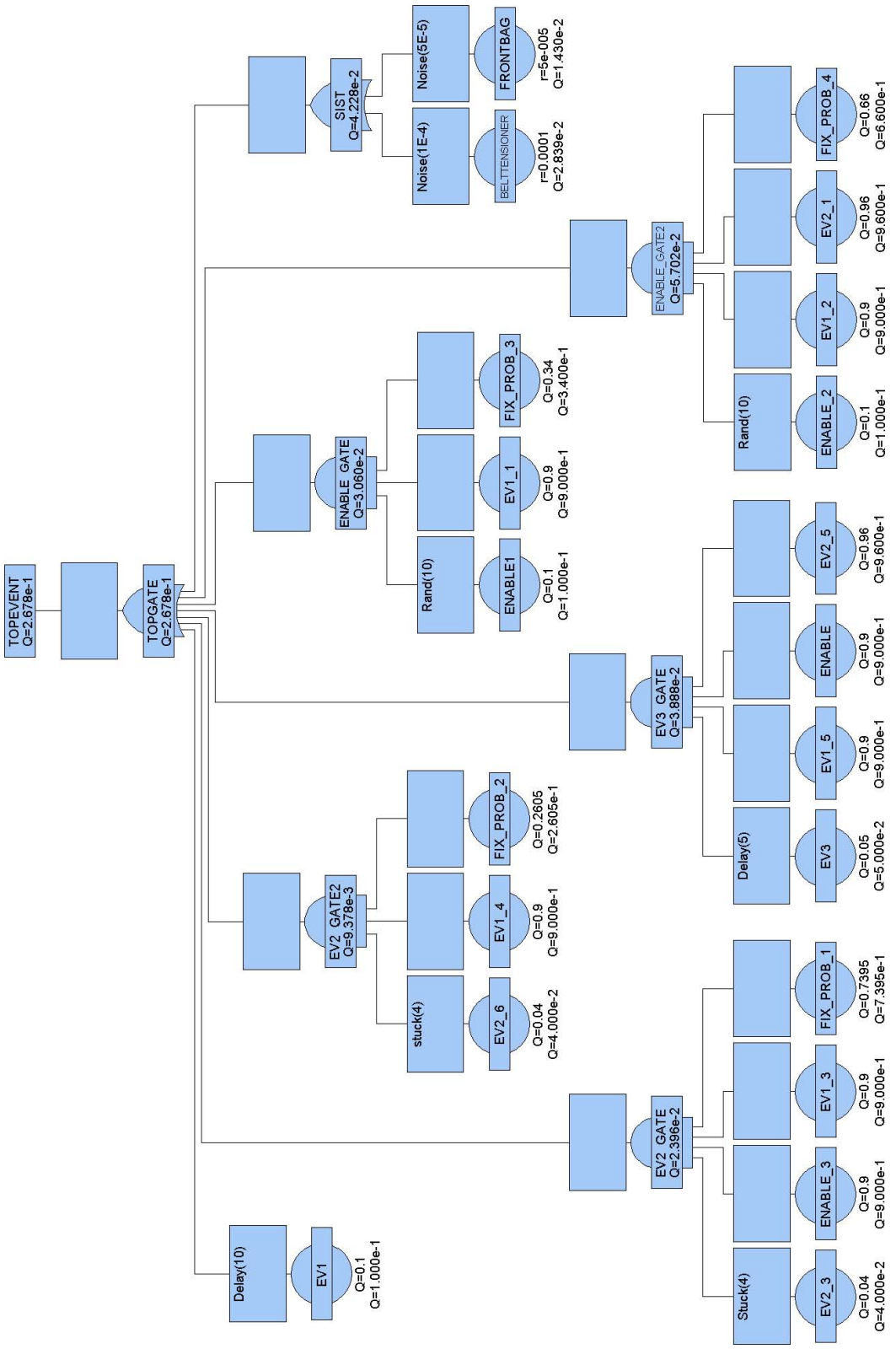


Figure 7.3: Generated fault tree with four artificial inserted errors with the SIST on the right.

Chapter 8

Importance Analysis

Reliability characteristics of a system are determined by the reliability of its units. Due to the structure of such systems some units are more important than others with respect to failure sensitivity. The importance of a single unit is determined by its parameters, that is e.g. the failure rate λ, repair rate μ, and its interrelation with the system.

Especially in complex modern systems, it is essential to identify the unit, for which improvements have the biggest impact on the overall system reliability (system optimization). In case the system break-down is triggered by the failing unit, the unit is required to get immediate repair to bring the system back into a sound state. When trying to optimize the maintenance, the most important unit should be repaired first, which brings the system again state "up" with the highest probability[Moc05].

In this chapter different approaches towards safe and failure critical systems are investigated with special emphasis on the gate level. As such different important measures are defined that can be used to classify vital components according to their importance figures.

8.1 Structural Importance

We consider a system representation in terms of a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$ consisting of n components. To every unit i of this vector a state can be assigned, namely $x_i = 1$ for failing or $x_i = 0$ for faultless. A system that is represented by units x_i for $i = 1, ..., n$ can only assume states 0 or 1. The *system function* $\Phi()$ is defined as the the current system state, expressed by state vector $\vec{x}$ and evaluates given a system vector either to 0 in case of a sound system or to 1 in case of an erroneous one. Since n components are in the system, there are 2^n different state vectors of a system (combinations of zeros and ones).
A system is failing if unit i fails ($\Phi(\vec{x}, x_i = 1) = 1$), and stays faultless if component i is sound ($\Phi(\vec{x}, x_i = 0) = 0$). In case that the state of one particular unit i is fixed, the number of possible outcomes reduces to 2^{n-1}.

Gates, vital for the system fulfill *critical vectors* condition, since a failure of one of these components causes the whole system to fail. The critical vector condition [Eq. 8.1] states that if the difference of system functions Φ of 1^{st} a system where component i fails and 2^{nd} component i is proper, is calculated to be 1, component i is a critical and vital gate.

$$\Phi(\vec{x}, x_i = 1) - \Phi(\vec{x}, x_i = 0) = 1 \tag{8.1}$$

The structural importance is defined as quotient of the number of all critical vectors of unit i ($n_{\Phi(i)}$) and the total number of all possible vectors (2^{n-1}). The *relative* importance of unit i for the proper function of the system is hence

$$I_{\Phi(i)} = \frac{n_{\Phi(i)}}{2^{n-1}}$$

Example

Consider the circuit of figure 8.1. The Boolean function that suffices this figure is as follows, where $\overline{x}_i$ denotes a failure in unit x_i and $\overline{y}$ failure of the system. It stresses that either component x_1 fails ($\overline{x_1}$), or x_2 and x_3 ($\overline{x_2} \cdot \overline{x_3}$).

$$\overline{y} = \Phi(\vec{x}) = \overline{x_1} + \overline{x_2}\,\overline{x_3} - \overline{x_1}\,\overline{x_2}\,\overline{x_3} \tag{8.2}$$

For the two possible states vectors for unit x_1 being fixed, we obtain

$$\begin{aligned} 1 + \overline{x_2}\,\overline{x_3} - \overline{x_2}\,\overline{x_3} &= \Phi(\vec{x}, x_1 = 1) \qquad \text{, if } \overline{x_1} = 1 \\ \overline{x_2}\,\overline{x_3} &= \Phi(\vec{x}, x_1 = 0) \qquad \text{, if } \overline{x_1} = 0 \end{aligned}$$

Applying the *critical vector* condition 8.1 we end up with

$$\Phi(\vec{x}, x_1 = 1) - \Phi(\vec{x}, x_1 = 0) = 1 + \overline{x_2}\,\overline{x_3} - \overline{x_2}\,\overline{x_3} - \overline{x_2}\,\overline{x_3} = 1 - \overline{x_2 x_3} \stackrel{!}{=} 1 \tag{8.3}$$

By probing the correct assignment for x_2 and x_3 we find 3 possible solutions that boil down equation (8.3) from above and hence $n_{\Phi(1)} = 3$. The structural importance regarding unit x_1 is therefore

$$I_{\Phi(1)} = \frac{n_{\Phi(1)}}{2^{n-1}} = \frac{3}{4}$$

Proceeding in the same way for unit x_2 and x_3, we obtain

$$I_{\Phi(2)} = I_{\Phi(3)} = \frac{1}{4}$$

and conclude that unit x_1 has the highest structural importance and is hence the most significant unit within the circuit.

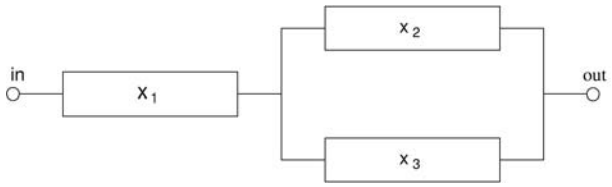


Figure 8.1: Example circuit with three components.

8.2 Marginal Importance

The *Birnbaum* or marginal unavailability importance measure $I_m(i)$ for an event represents the sensitivity of system unavailability with respect to changes in the events unavailability. It is defined as the probability that the system reaches a state, in which the proper operation of unit i becomes critical. For calculating the marginal importance the system function $\overline{y}$ is used as before where each component failure is now represented by the components probability to fail. The new function obtained is called probability function $F(\vec{q})$, and the importance measure is received by partial derivation

$$I_m(i) = \frac{\partial F(\vec{q})}{\partial q_i}.$$

Example

Considering the example from above [Fig. 8.1] where a failing component $\overline{q_i}$ fails according to a exponential distribution $q_i = 1 - e^{-\lambda_i t}$. Starting with the system failure probability function

$$F(\vec{q}) = \overline{q_1} + \overline{q_2}\,\overline{q_3} - \overline{q_1}\,\overline{q_2}\,\overline{q_3}$$

we end up with the marginal importance measure by taking partial derivatives of each component.

$$\begin{aligned} I_m(1) &= \frac{\partial F(\vec{q})}{\partial q_1} = 1 - q_2 q_3 \\ I_m(2) &= \frac{\partial F(\vec{q})}{\partial q_2} = q_3 - q_1 q_3 \\ I_m(3) &= \frac{\partial F(\vec{q})}{\partial q_3} = q_2 - q_1 q_2 \end{aligned}$$

The resulting marginal importance measure can be computed by plugging in the appropriate probabilities. The greater the importance measure of unit i, the more important it is.

8.3 Barlow-Proschan Importance

The *Barlow-Proschan* importance measure is used for analyzing systems, whose units could possibly fail sequentially. The general analysis yields an expression for the expected failure frequency of base event i during mission time $[0, t_M]$. The failure probability q_i of a system unit *i* is the expected value $E(\Phi(\vec{x}))$, that *i* fails (reaches state 1). The probability that the system is in state 1 (failed) depends on the individual q_is by

$$F(\vec{q}) = F(q_1, q_2, ..., q_n) := Pr(\Phi(x_1, x_2, ..., x_n) = 1) = E(\Phi(x_1, x_2, ..., x_n))$$

Every component *i*'s probability to fail is described by an exponential distribution function as stated below with λ_i being the failure rate of component i.

$$q_i(t) = 1 - e^{-\lambda_i t}$$

The probability, that the system is reaching a state at time t_M in which unit i fails is

$$\frac{\partial(F(\vec{q}\ (t_M)))}{\partial q_i(t_M)}.$$

The probability that unit i fails within time $[t_M, d \cdot t_M]$ for $d \geq 1$ is

$$d \cdot q_i(t_M).$$

We obtain the probability, that unit i causes a failure of the system within time $[t_M, d \cdot t_M]$ with $t_M \leq t$

$$\frac{\partial(F(\vec{q}\ (t_M)))}{\partial q_i(t_M)} \cdot d \cdot q_i(t_M).$$

By integrating over time t_M the probability that unit i causes a system failure within $[0, t]$ is

$$\int_0^t \frac{\partial(F(\vec{q}\ (t_M)))}{\partial q_i(t_M)} \cdot d \cdot q_i(t_M) \quad dt_M.$$

The failure probability for a system containing n units within time [0, t] is hence

$$F(\vec{q}) = \sum_{i=1}^{n} \int_0^t \frac{\partial(F(\vec{q}\ (t_M)))}{\partial q_i(t_M)} \cdot d \cdot q_i(t_M) \quad dt_M$$

Since we are interested in the failure probability of one particular unit, the *Barlow-Proschan Importance* of unit i is the quotient of the probability for unit i failing within time t over the overall probability of the system failing.

$$I_i^{BP} = \frac{\int_0^t \frac{\partial(F(\vec{q}(t_M)))}{\partial q_i(t_M)} \cdot d \cdot q_i(t_M) \quad dt_M}{F(\vec{q})}$$

With a failure density

$$f_i(t) = \frac{\partial q_i(t)}{\partial t}$$

we end up with the *Barlow-Proschan* importance for unit i

$$I_i^{BP} = \frac{\int_0^t \frac{\partial(F(\vec{q}(t_M)))}{\partial q_i(t_M)} \cdot f_i(t_M) \quad dt_M}{F(\vec{q})}.$$

Apart from this it holds that

$$\sum_{i=1}^{n} I_i^{BP} = 1$$

which trivially follows of the derivation of the failure model.

Example (stationary case)

Let $F(\vec{q})$ be the failure probability function from the example above [Fig. 8.1] with an exponentially distributed failure probability $q_i(t)$ as follows

$$F(\vec{q}) = \overline{q_1} + \overline{q_2}\,\overline{q_3} - \overline{q_1}\,\overline{q_2}\,\overline{q_3}$$

$$q_i(t) = 1 - e^{-\lambda_i \cdot t}$$

It follows

$$
\begin{aligned}
I_1^{BP} &= \frac{\int_0^t \frac{\partial(F(\vec{q}\,(u)))}{\partial q_i(u)} \cdot f_i(u)\, du}{\overline{q_1} + \overline{q_2}\,\overline{q_3} - \overline{q_1}\,\overline{q_2}\,\overline{q_3}} \\
&= \frac{\int_0^t (1 - q_2 q_3) \cdot f_1(u) \quad du}{F(\vec{q})} \\
&= \frac{\int_0^t \left(1 - (1 - e^{-\lambda_2 u}) \cdot (1 - e^{-\lambda_3 u})\right) \cdot \lambda_1 e^{-\lambda_1 u}\, du}{F(\vec{q})} \\
&= \frac{\lambda_1 \int_0^t (e^{-\lambda_2 u} + e^{-\lambda_3 u} - e^{-(\lambda_2+\lambda_3)u}) \cdot e^{-\lambda_1 \cdot u} \quad du}{F(\vec{q})} \\
&= \frac{\lambda_1 \int_0^t (e^{-(\lambda_1+\lambda_2)u} + e^{-(\lambda_1+\lambda_3)u} - e^{-(\lambda_1+\lambda_2+\lambda_3)u}) \cdot du}{F(\vec{q})} \\
&= \frac{\lambda_1 \cdot \left[-\frac{1}{(\lambda_1+\lambda_2)} e^{-(\lambda_1+\lambda_2)u} - \frac{1}{(\lambda_1+\lambda_3)} e^{-(\lambda_1+\lambda_3)u} + \frac{1}{(\lambda_1+\lambda_2+\lambda_3)} e^{-(\lambda_1+\lambda_2+\lambda_3)u} \right]_0^t}{F(\vec{q})} \\
&= \frac{\lambda_1 \cdot \left(\frac{1-e^{-(\lambda_1+\lambda_2)t}}{\lambda_1+\lambda_2} + \frac{1-e^{-(\lambda_1+\lambda_3)t}}{\lambda_1+\lambda_3} - \frac{1-e^{-(\lambda_1+\lambda_2+\lambda_3)t}}{\lambda_1+\lambda_2+\lambda_3} \right)}{F(\vec{q})} \\
&\overset{Eq8.4}{=} \frac{\lambda_1 \left(\frac{1}{\lambda_1+\lambda_2} + \frac{1}{\lambda_1+\lambda_3} - \frac{1}{\lambda_1+\lambda_2+\lambda_3} \right)}{F(\vec{q})} \\
&\overset{F(\vec{q})=1}{=} \lambda_1 \left(\frac{1}{\lambda_1+\lambda_2} + \frac{1}{\lambda_1+\lambda_3} - \frac{1}{\lambda_1+\lambda_2+\lambda_3} \right)
\end{aligned}
$$

For simplifications reasons we only consider the stationary limit ($\lim_{t\to\infty}$) were the system will eventually end up in a failure state. Thus we obtain for $t \to \infty$ the system will end in a failure, and $F(\vec{q}) = 1$, and hence

$$\lim_{t\to\infty} e^{-\lambda t} = 0. \tag{8.4}$$

This reduces the *Barlow-Proschan importance* for gate 1 to

$$I_1^{BP} = \frac{\lambda_1}{\lambda_1+\lambda_3} + \frac{\lambda_1}{\lambda_1+\lambda_2} - \frac{\lambda_1}{\lambda_1+\lambda_2+\lambda_3}$$

For gate 2 and 3 the following Barlow-Proschan importances are obtained:

$$I_2^{BP} = \frac{\lambda_2}{\lambda_1 + \lambda_2} - \frac{\lambda_2}{\lambda_1 + \lambda_2 + \lambda_3}$$

$$I_3^{BP} = \frac{\lambda_3}{\lambda_1 + \lambda_3} - \frac{\lambda_3}{\lambda_1 + \lambda_2 + \lambda_3}$$

To check for correctness the sum of all BP importances have to equal one.

$$\begin{aligned} \sum_{i=1}^{n} I_i^{BP} \stackrel{!}{=} 1 &= I_1^{BP} + I_2^{BP} + I_3^{BP} \\ &= \frac{\lambda_1 + \lambda_2}{\lambda_1 + \lambda_2} + \frac{\lambda_1 + \lambda_3}{\lambda_1 + \lambda_3} - \frac{\lambda_1 + \lambda_2 + \lambda_3}{\lambda_1 + \lambda_2 + \lambda_3} \\ &= 1 + 1 - 1 \end{aligned}$$

We consider the unit i to be most important according to the *Barlow-Proschan Importance* if I_i^{BP} yields the highest value.

8.4 Fussell-Vesely Importance

The Fussell-Vesely standard importance measure for gates - also known as diagnostic importance - indicates an event's or event group's contribution to the gate unavailability. Considering the state of all system units right after a system failure one can conclude, that all units of at least one minimal cut set failed. By repairing at least one unit of the *minimal cut set*, the system becomes operational again. *Minimal cut sets (MCS)* are all unique combination of component failures that can cause system failure. Specifically, a cut set is said to be a minimal cut set if, when removing any basic event from the set, the remaining events collectively are no longer a cut set.
We say that unit *i* has a big diagnostic impact on the system, if at least one of the minimal cut sets that contain *i* contribute with a high probability to a system failure. The diagnostic importance measure is computed by

$$I_d(i) = \frac{Pr(\text{failure of a minimal cut set, that contains unit i})}{Pr(\text{system failure})} = \frac{Pr(\Phi_i(\vec{x}) = 1)}{Pr(\Phi(\vec{x}) = 1)}$$

where the system function of all minimal cut-sets that contain i is identified as

$$\Phi_i(\vec{x}) = \bigvee_{j=1}^{n} \left[\bigwedge_{c \in C_{ij}} \overline{x}_c \right].$$

n is the number of minimal cut-sets, that contain i and C_{ij} is the j-th minimal cut, that contains *i*. It holds that $Pr(\Phi_i(\vec{x})) \approx \sum_{j=1}^{n} Pr(C_{ij})$. As before it is said, that unit i has the highest diagnostic importance if $I_d(i)$ yields the highest value.

Example

Consider the example from figure 8.1 on page 70 were we identify two minimal cut sets as $\{\overline{x}_1\}, \{\overline{x}_2, \overline{x}_3\}$.

$$Pr(\Phi(\vec{x}) = 1) = Pr(\overline{x}_1 \vee \overline{x}_2 \wedge \overline{x}_3) = q_1 + q_2 q_3 - q_1 q_2 q_3$$

$$I_1^D = \frac{F_1(\vec{q})}{F(\vec{q})} = \frac{q_1}{q_1 + q_2 q_3 - q_1 q_2 q_3}$$

$$I_2^D = \frac{F_2(\vec{q})}{F(\vec{q})} = \frac{q_2 q_3}{q_1 + q_2 q_3 - q_1 q_2 q_3}$$

The most important unit follows out of the values for q_i and since the second MCS contains two components, both have the same diagnostic importance of I_2^D.

8.4.1 Modeling in *MoDeST*

The diagnostic importance is the only one that can be obtained by simulation in *MoDeST* [App. F]. Figure 8.2 depicts the example considered for simulation. Notice that we focus on OR-gates since AND-gates have no impact on the *Fussell-Vesely* measure. For this a circuit having 4 components is chosen.

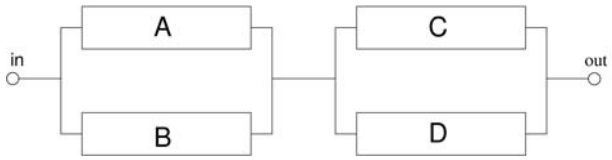


Figure 8.2: Circuit that is examined using the Fussell-Vesely importance by simulation.

Variable	value		confidence
ImpA	1.0030050000E-02	+/-	1.9530736850E-05
ImpB	5.0170930000E-02	+/-	4.2786314378E-05
ImpC	5.9013840000E-02	+/-	4.6187518417E-05
ImpD	3.5686000000E-04	+/-	3.7019240478E-06

Table 8.1: Component failures simulated by *MoDeST* which are used to compute the Fussell-Vesely Importance.

	Simulation ($Q_{system}(q_i = 0)$)	simulation $I_d(i)$	FT+ analytical $I_d(i)$
TE	5.92E-002		
A	1.00E-002	0.83056	0.883
B	5.02E-002	0.15244	0.167
C	5.90E-002	0.00305	0.00596
D	3.57E-004	0.99397	0.994

Table 8.2: Comparison between the simulated importance results and the analytical numbers obtained by *Fault Tree +*.

Simulation

By simulating 100 million batches results are obtained as shown in table 8.1. The data for the simulated importances can be calculated by

$$I_d(i) = \frac{Q_{system} - Q_{system}(q_i = 0)}{Q_{system}}.$$

A comparison of figures obtained by simulation and using the analytical methods is given in table 8.2. The fault tree that corresponds to the example circuit is depicted in figure 8.3 where the top event is computed by *Fault Tree +*.

Conclusion

Concluding, the analytical importance measures can be reached even closer by increasing the number of computed batches. The results reveal that importance analysis is possible by using simulation. It remains an open question whether this is the best approach and for obtaining the Fussell-Vesely importance measure.

8.5 Conclusion

In the end it remains to an open question which importance is most significant. The choice of the appropriate importance measure as stated here always depends on the scenario. It can be seen that for

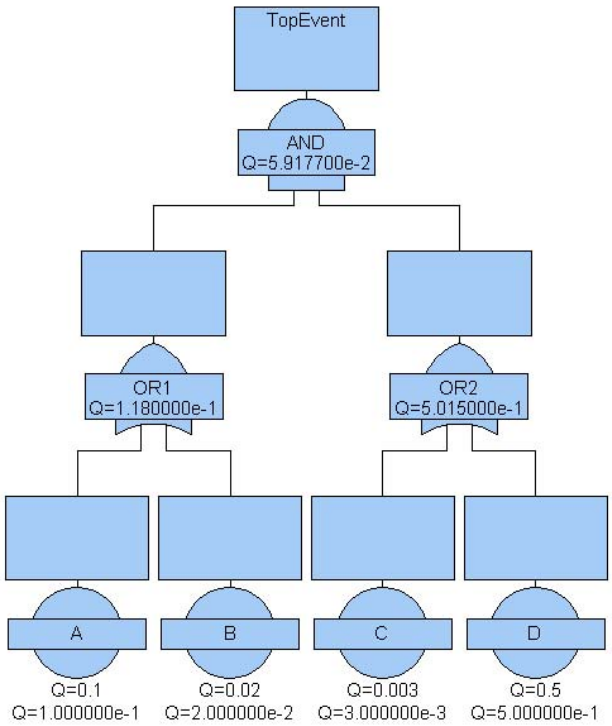


Figure 8.3: FTA for doing Fussell-Vesely Importance analysis, derived from the example circuit.

basic circuits where only the structure matters and neither time nor failure probabilities, the structural importance measure fits best - as the naming suggests.
When using unit failures over time, the rating received by *Birnbaum* or *Barlow-Proschan* are preferable to use. Investigating a system by minimal cut sets (MCS) is done via the *Fussell-Vesely* importance where the system is considered up unless one of the components out of a minimal cut set is failing.

Chapter 9

Single Source Vision

Due to the complexity of modern electronic products and their design cycles there is an ambition in industries for a unified concept (*single-source*) that can be used throughout all development phases of a product. During such cycles, model, source code, documents, and plans (*artifacts*) are devised. This attempt is also known as *agile modeling* [Amb02] thrives for a unified single-source containing a behavior model in terms of source code, specifications, requirements, and failure description. Although it can not always be achieved it is nonetheless desirable to record each piece of information only once, as far as possible.

This chapter is an attempt to test the feasibility of a single-source model with special emphasis of how failures can be represented in such a model. Moreover by providing sufficient failure information, and incorporating them in an appropriate fashion into the model representation, static failure analysis via *Fault Tree +* can be achieved by one derivation of the single source model. In addition, since the behavior model is available, a state space can be computed for each process to check for dynamic failures.

9.1 Overview

Using a single-source model has three strong pro arguments that are, it reduces the maintenance burden because the more representations are kept in the model, the greater the effort to adopt for changes and keeping track of versions. Second, the trace ability is flattened since every redundant piece of information hampers the debugging process unless it is clear where the information stems from. And last, an increase in consistency is achieved because the chance of having outdated artifacts is dramatically reduced.

Ideal is a single source of information containing all artifacts as stated in figure 9.1 that can be used for deriving simulation, and verification. Beyond that it should be eligible to infer stochastic “views” from the single-source model that can be used for Fault Tree Analysis (FTA) and Markov Chain Analysis

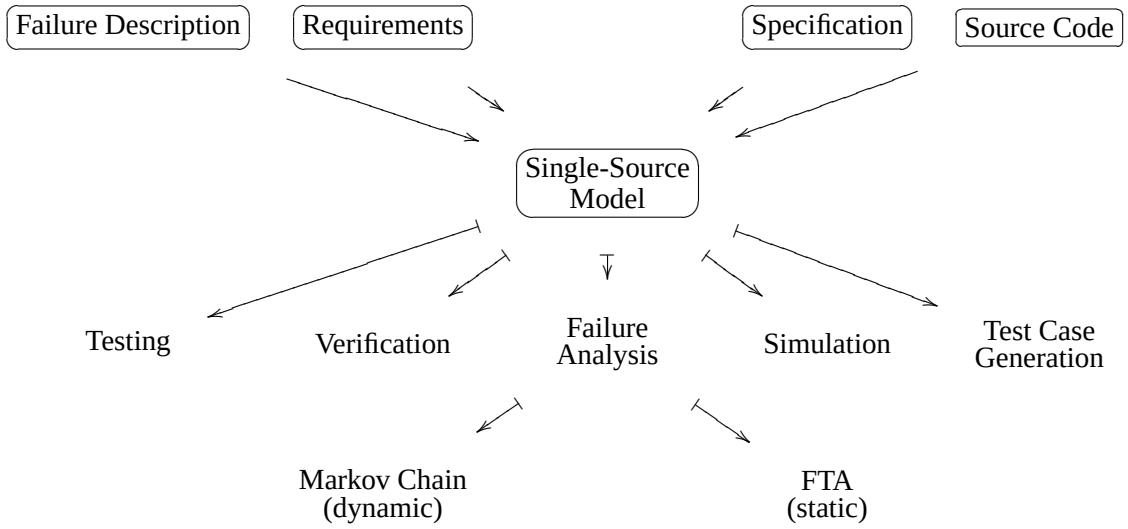


Figure 9.1: Illustration of a Single-Source Modeling attempt, showing all desired derivations. Available information is distributed in sources Specification, Failure Description, Requirements, and Source Code and combined for the appropriate representation. Possible derivation to obtain are Testing, Verification, Failure Analysis, Simulation, and Test Case Generation.

(MCA). We do not forbid the Single-Source-Model to have more than one format for data storage since information should be stored in the most appropriate format that offers the best representation.

9.2 Choice of the overarching Language

To accomplish the traditional single-source vision, a common way of recording information is needed. Since every representation has its pros and cons we strive to use the most appropriate representation. *MoDeST* has full notion of expressing STAs (Stochastic Timed Automata) and using a subset of *MoDeST* one can express DTMCs, CTMCs, and IMCs, needed for the Markov analysis. Besides this different stochastic concepts like Exponential-, Uniform-, Normal distribution, and probabilistic branching are present to account for fault-tree analysis.
Since the success of a single-source approach is determined by the expressiveness of its underlying language, this approach is promising a valuable basis for further proceedings. For the sake of simplicity, the behavior model will be written in *MoDeST* plus some extra information inserted by leading preprocessor keywords (#). In particular by using *GEMA* - the *MoDeST* preprocessor - information can be easily extracted and repeating pattern can be efficiently reduced to a minimum workload. By using an approach as shown in figure 9.2, a failure description and a behavior model is added to the single source model. The derivations as shown here will be derived in the following.

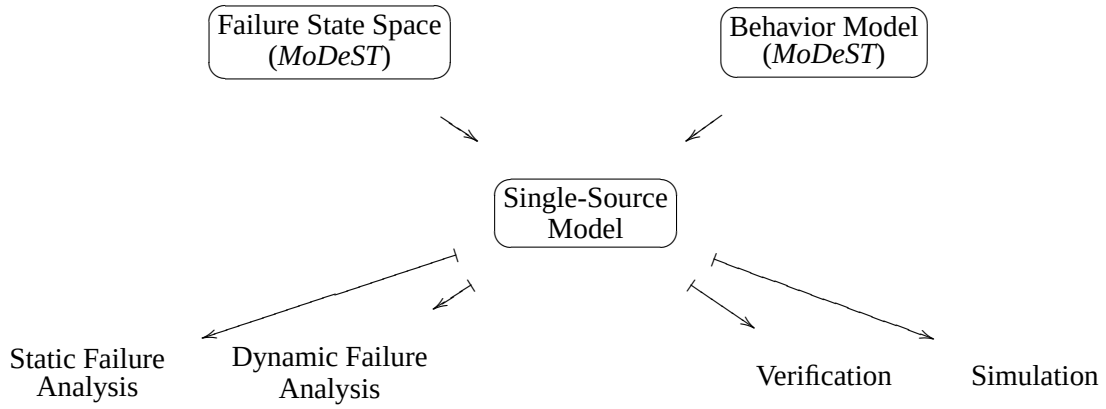


Figure 9.2: The way we want to investigate and distribute information using a single-source approach doing simulation, verification, and deriving failure analysis.

9.3 Example of a Water Cycle

The example that we want to use for the feasibility analysis is a simple scenario [Fig. 9.3] consisting of a water pump, pumping water into a water tank, a sensor that controls the pump, and finally a consumer that is attached to the tank and consumes water at some rate. The sensor signals control the pump and shut it off once a certain pressure (MAX) is reached. In case that the water pressure in the tank is passing some threshold MIN, the sensor commands the pump to power up again. Pressure sensor and water pump need power from outside power supply (VCC) for proper functioning. Possible points of failures of the components modeled are the *SensorFailure*, *PumpFailure*, and *TankBurst*, and the external lack of power (*missingVCC*), of which are all repairable, but the water tank.
The setting of the components with their respective failures is illustrated in the *MoDeST* code [App. H.1]. Once a device fails, it is eventually repaired at some rate μ. The water tank fails or bursts at rate λ, if the pressure becomes critical above the maximal threshold MAX.

9.3.1 *MoDeST* Behavior Model

The *MoDeST* model is given in appendix [H.1] with processes *Sensor()*, *Tank()*, *Pump()*, *Power()*, and *Consumer()*. The failures that each process has are declared if present within the process declaration as follows:

#FailureName(failure rate, repair rate)

In addition, input (#in), and output (#out) ports are declared within each process to help relate the process to the environment which gives later on a point for automating. Integers *max* and *min* indicate the maximal pressure where the sensor should power off the pump and the minimal pressure where the pump is reactivated again.

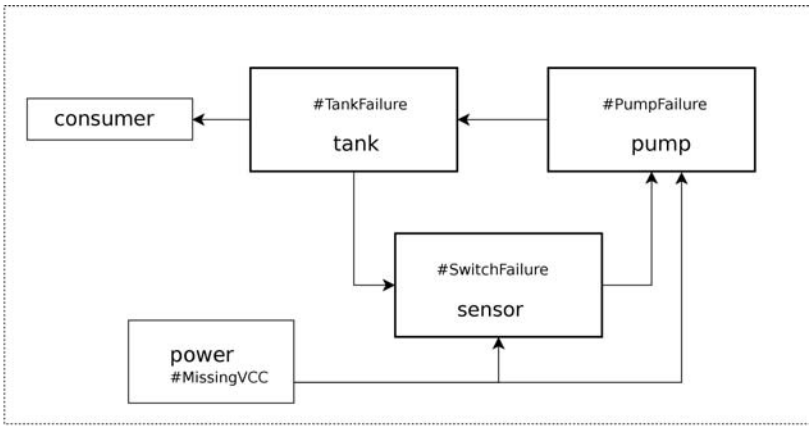


Figure 9.3: The above figure depicts an overview of components including their signals and point of failures. Failures are indicated by leading #'s

9.4 Failure Analysis

9.4.1 Static FTA

In static failure analysis one is interested in the break down of a system according to failure rates of its components. The decision about which base events contribute to the top event solely depends on the modeling engineer who has to bring experience and a well shaped background about the modeled device to build a sound model. In our simple example the static fault tree is obtained by extracting the failures that each process owns and simply relate them using gates as described in the model to a top event. The interrelation of the top event needs to be specified in the model like

$$\text{process(A)} \vee \text{process(B)} \rightarrow \text{TopEvent}.$$

In the static FTA it is considered to be critical if the pump fails while the tank can still provide water. Applying conventional failure analysis to our *MoDeST* model, we end up with a fault tree having 4 base events - like shown in the behavior model - that are related to the top event by an or-gate. The top event is classified here by either a pump-, tank-, sensor-, or the power supply break down.

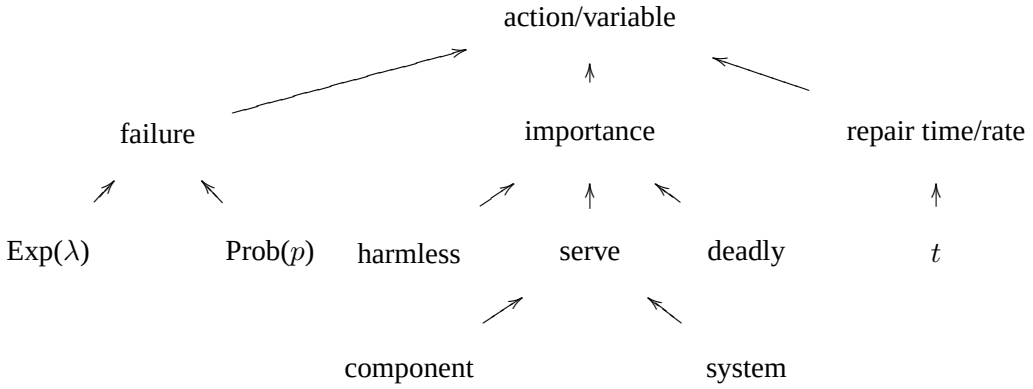
The disadvantage when doing the static fault tree analysis is that the water tank example has a dynamic behavior, and as such it is inappropriate of representing errors, and doing the dependability analysis. Imagine a pump being switched on while the sensor is failing. This scenario is not harmful for the system if the sensor is repaired before the pressure exceeds the MAX threshold and the water tank consequentially bursts. In contrary, in the dynamic system a critical error is classified by a burst of the tank or 2^{nd} a consumer lacking water supply. To adopt for this dynamic error specification, a stochastic timed automata chain representation is to be favored, rather than the static fault tree analysis.

9.4.2 Dynamic Failure Analysis

The above mentioned motivates a classification as follows below. A failing component which is not used at point of its failure (harmless error) has no impact on the system behavior as long as it will be repaired before being demanded. If it fails to do so, the failure endangers the proper function of the component and is classified thus as *serve*. It influences in consequence other components and leads eventually to a deadly not repairable system.
Component failures can be categorized as follows:

1. harmless
   The failure doesn't have any impact on the overall system behavior at occurrence and is repairable, e.g. lack of power while the pump is switched off.
2. serve
   This has effect on the proper function of the containing device, e.g. sensor failure while pump is switched on. Although the possibility for repairing exists other components might already be influenced by the catatonic behavior.
3. deadly
   A system is classified to be deadly if no possibility of repairing or recovery exists, e.g. the burst of the water tank.

To have useful specifications and error descriptions each action has to hold the following information:



9.5 STA Chain Representation

In the preceding we add the failure description into the single source model. The classification is not explicitly contained in the behavior model until now and has to be inserted manually and independently for each process. The chain representation is obtained by exploring the state space of each

process and relating variables and guards on the appropriate transitions.
Imagine the example from above with the water tank, the sensor, and the pump, with each component having a different importance for the system's proper function. In case of a failing pump the failure has no direct impact on the system if the water tank is filled and able to provide water, where as a burst of the water tank has deadly implications for the system.

The behavior is stated in the following model [Fig. 9.4] by using a STA Chain, since it has in addition to exponential transitions synchronous and asynchronous actions like Interactive Markov Chains (IMC) [Her02]. To handle additional probabilities and condition checking the expressiveness of Stochastic Timed Automata is required in which case we find ourselves back in the language of *MoDeST*. Thus the STA chain is implemented in *MoDeST*, already in use for expressing the behavior model.

GEMA

To ease the work load, a *GEMA* extension [App. A.11] is provided that translates rates prefixed by #LAMBDA or #MU into *MoDeST* equivalent. In the remainder, rates with leading #LAMBDA will be consider as failure rates while #MU denote a component specific repair rate.

9.5.1 STA Error Chain in *MoDeST*

The final *MoDeST* model representing the STA-Markov chain is illustrated in appendix [H.2] and can be directly used to simulate the errors within the dynamic system of the water cycle. Rates *LAMBDAvcc*, *MUvcc*, etc and actions *TankFull*, *TankBurst*, and *NoWater* need to be defined appropriate as *Möbius* reward variables to gain valuable results.

9.6 Conclusion

Beyond the methods for failure analysis as described above, further extensions are possible to derive. As such it is feasible to run model checking and verification via a translation into *Uppaal* that is currently implemented by *Christoph Keppner*. This provides in turn the last open derivation of our single-source model at figure [9.2].
This chapter shows how to build a modular single source model by using sparse information. In addition we are able to obtain a static failure representation that can be extracted out of the behavior model. The dynamic failure representation was done manually although it is thinkable of automating the state generation process and derive the STA-Chains out of the behavior model.
Having in mind the drawback of simulation as denoted in the chapters before, no guarantee is given for a exhaustive state exploration.

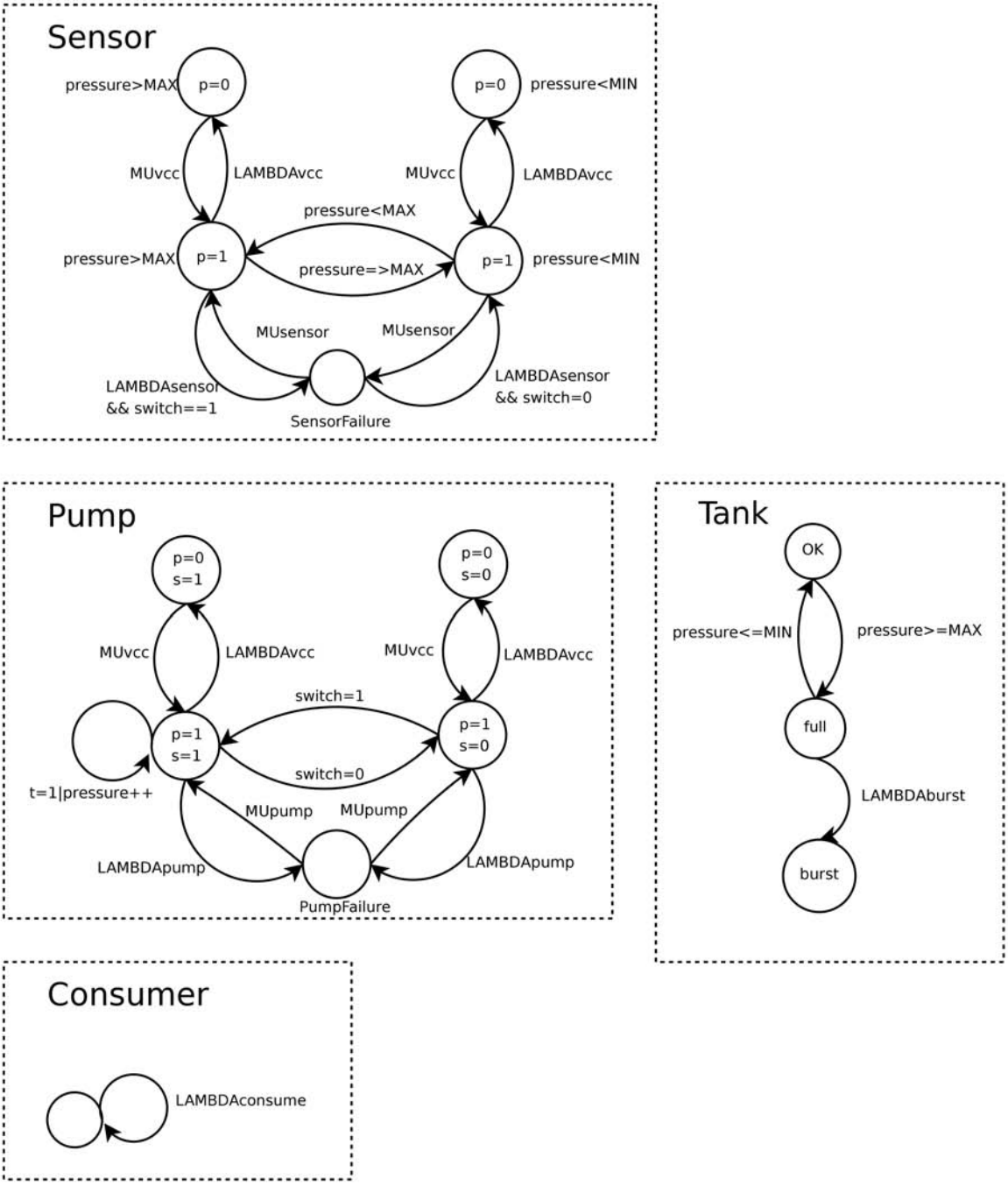


Figure 9.4: STA chain where edges can be labeled with probabilities, condition checks, and actions. Node labels are optional and not required for simulation.

Chapter 10

Conclusion

Summing up this work, the principles of synchronization concepts are investigated in chapter 1, and a formalism is defined out of which any symmetric or two-pair synchronization can be derived. By exemplifying communication concepts and check their feasibility in *Uppaal* and *MoDeST*, the topic is bridged from a formal definition towards informal nomenclature. In addition, the implementability of the most useful schemes like *one-of-many-to-many*, *one-of-many-to-one*, broadcasting , etc are studied using *Uppaal* and *MoDeST*.

Simulations in *MoDeST* and *Stateflow*

Conclusions which can be drawn out of chapter 2 are that simulation results testify the insights attained by the *AMETIST* project, and state that it is possible to achieve the results from analysis in *Uppaal* by simulation in *MoDeST*. It turned out that simulation has its assets and drawbacks, i.e. simulating a model for one million times and analyzing for undesirable conditions does not refute nor endorse the same situation in focus. It rather accentuates, that during one million runs these situation has not occurred in the system and consequently has not been seen by an observer. Where chapter 2's focus is directed on a special scenario with focus on the approver, and the feasibility of expressing this scenario in *MoDeST*, a model with increased complexity is examined in chapter 3, where interaction and communication of components with respect to race conditions is examined.
This detailed system modeling and verification approach shows that it is feasible to obtain the previous acquired results by simulation in *MoDeST*. Especially, since timing and stochastic components are important to build a realistic model, the approach in *MoDeST* gives complementing results.
In chapter 4 the world of the stochastic timed automata is left [Fig. 10.1] when entering the world of *SimulinkStateflow*, and new focus is put on analyzing the airbag deployment times using statistical measures. By running 2 000 simulations a normal distribution with mean around $3\,500\mu s$ is obtained where the earliest point of deployment is at 3 $000\mu s$ since the *MicroForeground* process needs some cycles until firing is enabled.

Failure Analysis

The second part of this thesis highlights the failure analysis for the airbag control unit. The analysis of the on-demand safety features is discussed in chapter 5 by use of CTMCs. Modeling in *MoDeST* reveals how Markov analysis can be received via simulation. In addition figures are computed to account for *mean time to failure (MTTF)*, etc. It turned out that using rates of one time unit, no confident results are reached during simulation, unless increasing the simulation time up to 7 000 time units. The simulation results are confirmed by the analytical approach using matrix exponentiation were a failure distribution (Phase type distribution) of the controller is gained.
The Fault Tree Analysis in chapter 6 concludes that grouping of components into logical units can be an useful advantage during all cycles of a product. This benefit is exemplified by simulation in *MoDeST*. Although when dealing with small rates and system lifetimes, the number of experiments needed for expressive results is very high, they are nevertheless closely approached.
An other approach for failure analysis is the *automated fault tree generation* illustrated by simulation in chapter 7. Although the state space is not exhaustively search for all candidates the resulting fault tree is still useful. It contains the static error behavior of the system and can easily be obtained in particular for huge and complex systems. States not seen during simulation in the trace need no attention because they hardly contribute to the probability of the top event occurrence due to the relatively small probability. In lieu of simulating stochastic and probabilistic errors in the same model, a separation can bring some advantage. In particular since it suffices to simulate the on-demand errors for one time unit, results can be gained faster. Moreover by adding the stochastic components directly into the fault tree (SIST) their results become more accurate.
Concluding the importance analysis (chapter 8), several formalisms are discussed. Using simulation, the *Fussell-Vesely* importance measure obtained by *Fault Tree +* are approximated closely. The remaining three importances like *Structural-*, *Marginal-*, and *Barlow-Proschan Importances* can not be simulated since a perception of how component are connected among each other is required.
For this reason some effort to investigate and especially to automate the process of fault tree generation is desirable. With respect to more complex models an automated generation is essential since most of the conversion from the simulation trace into preliminary fault trees, and from there into *Fault Tree +* is done by hand. The more, towards the complexity of a model, one could think about more realistic models with failure recognition and recovery from errors induced by noise using some self-healing mechanisms. By this it becomes feasible to observe the model under errors and identify bottle-necks within the model.

Unified Concept of a Single Source

So far the behavior and failure aspects of models are investigated that are spread over many different representations. In chapter 9 a Single Source Modeling approach is introduced that unifies static-, and dynamic failure analysis, verification, and simulation in one model. More than that, it shows how to build a modular single source model by using sparse information. Static failure representation is covered by extracting failures out of the behavior model. Dynamic failure representation has to be done manually although it is thinkable of automating the state generation process and derive the STA-Markov Chains out of the behavior model. No transformation exists up to now which converts STA processes into GSMPs, or CTMCs that could in turn be used for a analytical analysis using e.g.

Acronyms	Model Name	Citation	Used in tool
DFA	Deterministic Finite Automata	Huffman (1954)	*Simulink Stateflow*
SMP	Semi-Markov Process	Levy (1954)	
SA	State Automata		
DTMC	Discrete Time Markov Chain	Markov (1913)	
CTMC	Continuous Time Markov Chain	Kolmogorov (1938)	Markov Analysis
GSMP	Generalized Semi Markov Process	Glynn (1983)	
TA	Timed Automata	Alur (1990)	*Uppaal*
STA	Stochastic Timed Automata	D'Argenio (1997)	*MoDeST*

Table 10.1: Languages with acronyms, year of discovery, and tools that incorporate them.

phase type distributions. Hence the only way left is to simulate, still having in mind the drawback of simulation. For the verification aspect a translation into *Uppaal* already exists.

The success of a single source model also depends on *GEMA*, the *MoDeST* preprocessor. Some useful extensions are provided in appendix A that ease the work load when modeling and facilitate many of the single source derivations.

Taxonomy

This thesis follows different paths by traversing distinct aspects, depending on the respective task. Starting at the world of *DFA (Deterministic Finite Automata)* as used by *Simulink Stateflow* up to the region of *Stochastic Timed Automata (STAs)* many language classes are crossed in between by adding non-determinism, stochastic, time, and probability. Table [10.1] gives some compendium about the most the taxonomy candidates with the tools they are used in, and citations.

The following synopsis [Fig. 10.1] depicts the parts of the "world" with respect to the expressiveness of automata by giving a tractable and descriptive approach. The hierarchy in use will combine dimensions like probability, time, stochastic, and the presence or absence of action & delay nondeterminism that will span the space in an orthogonal fashion. Although there is a close relation between continuous probability and full stochastic in place, they are treated as distinct dimensions in the taxonomy. Cubes span the probability-stochastic-timed space.
Time axis is labeled with *none*, *discrete*, *continuous*, and *hybrid* to account for the different language capabilities. Hybrid in this context reflects a non-linear scale of time. In the probability dimension we only consider systems that have *none*, *discrete*, or a *continuous* notion of probability. The stochastic aspects used are no-stochastic, only exponential distributions stated by *exp* where especially the Markov property holds, or *full* stochastic ability where all distributions are admissible.
Curved arrows indicate the fourth dimension considered as the absence of non-determinism, the presence of action non-determinism, and finally the presence of action-& delay non-determinism that allow traversing the deterministic cube by passing the space bottom up. For clarity, *action non-determinism* is related to state changes ("pick the successor state non-deterministically"), where as

delay non-determinism is related to timing ("take transition within some time interval"). Note at this point that the presence of *delay nondeterminism* always requires *action nondeterminism* to be in place.

Deriving Languages

By forgetting about time, probability, the stochastic aspects and nondeterminism we end up with the simplest class of automata, the **DFA** (Deterministic Finite Automata) that have capability of recognizing regular expressions. A representative from this class is for instance *Stateflow* from chapter 4. From this outcome there are two classes reachable by just spending on one dimension. By adding discrete probability the language recognized by **DTMCs** (Discrete Time Markov Chains) is entered, and by adding action nondeterminism the class of state-transition diagrams known as **LTSs** (Labeled Transition Systems) is obtained. The DTMCs counterpart by appending action nondeterminism is the **PTS** (Probabilistic Transition System).
In our view the DTMC - used in Chapter 5 for the Markov analysis - has no direct "neighbor" that can be directly reached because **CTMCs** (Continuous Time Markov Chains) require exponential stochastic and continuous timing. Note here, that the nomenclature for the discrete time Markov chain does not refer to the memoryless property of the exponential function but rather to the Markov property, that the present state gives any information of the future behavior of the process, and knowledge of the history of the process does not add any new information.
From the class of CTMCs by allowing all stochastic distributions the language able of expressing **GSMPs** (Generalized Semi Markov Processes) is entered. Alternatively equipping the CTMC with action nondeterminism we end up at **CTMDPs** (Continuous Time Markov Decision Processes).
Since it is awkward to find an appropriate path that traverses the remaining languages of our world, the class of automata is wrapped up starting from the already noted deterministic finite automata. All automata besides the **DFA** have action- & delay nondeterminism since they have a notion of time. For example *Uppaal* (chapter 1) represented by **TA** (Timed Automata) is one candidate out of the elementary language class. Starting from here, the language of **PTA** (Probabilistic Timed Automata) is entered by adding discrete probability. **HA** (Hybrid Automata) in which each clock variable has a bounded time drift can be derived out of TA by adding hybrid time. Moving from the **TA** one step in probability dimension and reaching the discrete probability, the language of **PTA** (Probabilistic Timed Automata) is entered.
STA (stochastic timed automata) are equipped with full stochastic behavior, continuous probabilities, continuous time, and delay-&action non-determinism. This is the language of *MoDeST* and justifies the use of *MoDeST* as the base language of a Single Source Vision (chapter 9). The only extension left is to combine hybrid time with **STA** and enter **SHA** (Stochastic Hybrid Automata), the topmost language having ability of expressing all the others stated here.

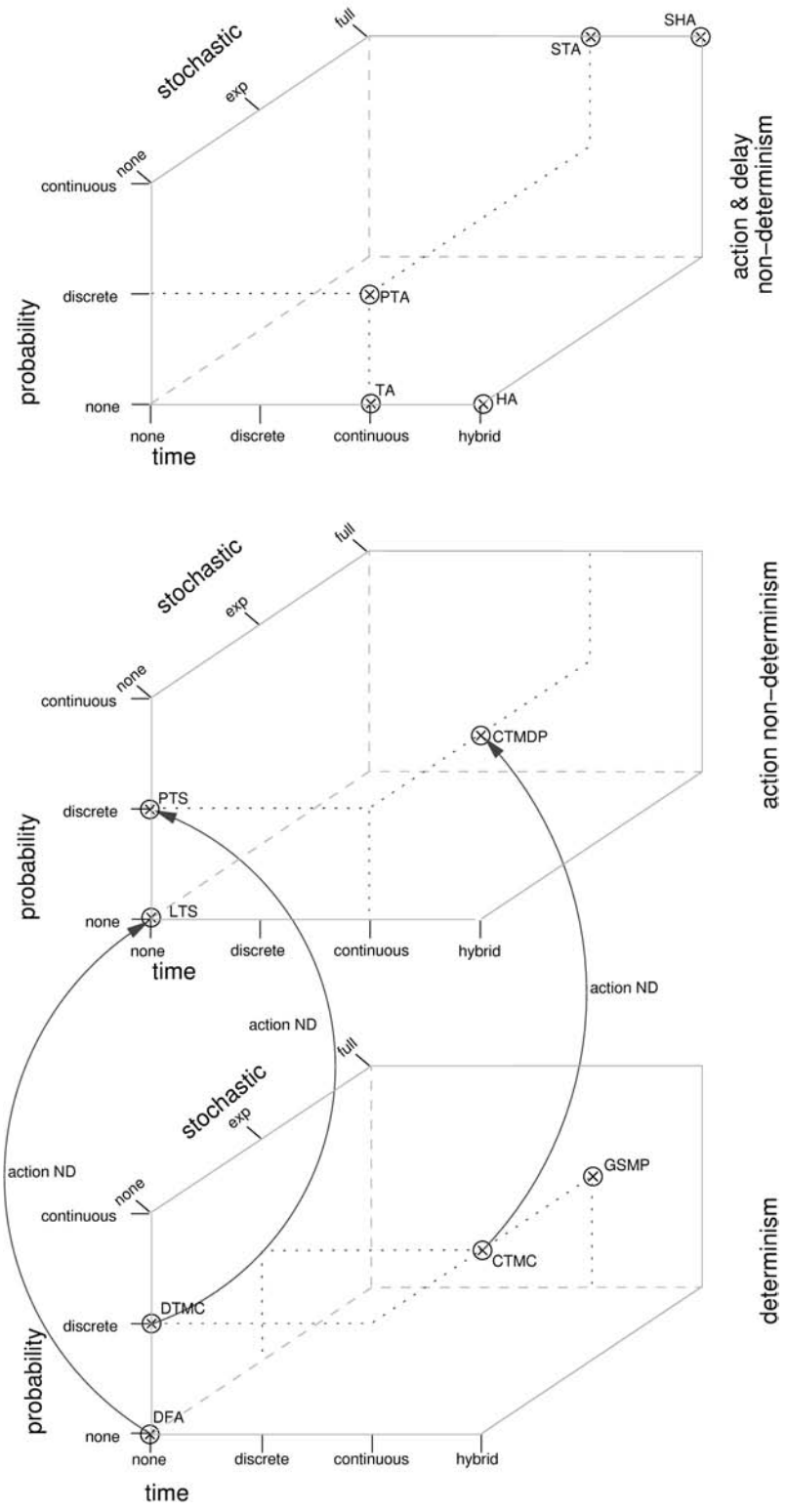


Figure 10.1: Four dimensional taxonomy of language classes that is spanned up by probability, stochastic, time, and determinism/nondeterminism. The lower cube spans the deterministic world that is successively extended by action nondeterminism and in top most cube with action- & delay non-determinism.

Appendix A

GEMA - a *MoDeST* Preprocessor

Introduction

GEMA [Gra03] is a general purpose text processing utility based on the concept of pattern matching. It can be used to do the sorts of things that are done by Unix utilities such as *cpp*, *grep*, *sed*, *awk*, or *strings*. It is also suitable as macro processor, but much more general than *cpp* or *m4* because it does not impose any particular syntax for what a macro call should look like. *GEMA* can deal with patterns that span multiple lines and nested constructs. It is also capable of using multiple sets of rules to be used in different contexts.
Based on *GEMA* a variety of preprocessor commands is introduced that ease the work with *MoDeST* in many ways.

A.1 #for

Enumerates each occurrence of <Variable> in <BODY> by integers starting at <n> up to <m>:

Syntax

```
#for [ <Variable> :  <n>..<m> ]{ <BODY>}
```

Grammar

```
\#for\W\[\W<I>\W\:\W<D>..<D>\W\]\W\{<Body>=\
@set{counter; @add{@sub{$3;$2};1}}\
@bind{c;$2}\
@repeat{@var{counter}; @{@subst{$1=$c;$4}} @incr{c}}\
```

```
@unbind{c}
```

A.2 #define

Can be used to define variables which are only accessible in the preprocessor. Further rules are needed to do e.g. constant substitution throughout the text.

Syntax

```
#define <Identifier> <Value>
```

Grammar

```
\#define\s<I>\s<N>=@bind{$1;$2}\/\/variable \ $1\ \ @var{$1}\n
```

A.3 #while

While construct as defined in the *MoDeST* standard with the following semantics:

$$\text{while (b)\{ P \}} \stackrel{def}{=} \text{do \{::when(b) P ::else break \}}$$

Syntax

```
#while (<b>) { <P> }
```

Grammar

```
\#while\W\(<T>\)\W\{<Body>\}=\
     do\{\n\t\
     \:\:when\($1\)\ $2\n\t\
     \:\:when\(\!\($1\)\) break\n\
     \}
```

A.4 #invariant

Invariant are defined in the *MoDeST* standard, where <b> is a guard and *invariant* an exception which is not used in the rest of the *MoDeST* specification. Hence every invariant has to have a unique naming which is obtained by `Invariant_<n>` with `n` being enumerated over the document.

$$\text{invariant (b) P} \stackrel{def}{=} \text{alt \{ ::when(b) P ::when } (\neg \text{ b}) \text{ urgent (ff) throw(}\textit{invariant}\text{) \}}$$

Syntax

```
#invariant ( <b> ) <P>
```

Grammar

```
@bind{invariantcount;0}

\#invariant\W\(\W<t>\W\)\W<I>=\
     alt\{\n\t\
     \:\:when\($1\)\ $2\n\t\
     \:\:urgent\(\!\($1\)\)\ when\(false\)\
     throw\(Invariant_$2\_${invariantcount}\)\n\
     @incr{invariantcount}\
     \}
```

A.5 #do and #alt

By prefixing `do` constructs with #, the preprocessor collects all guards found in `when` clause. The negated conjunctive normal form of all guards found is the new guard for the else case.
The same can be applied by using the `#alt` statement.

Syntax

```
#do { ::when <guard1> <body1> ...  ::else <ELSEBODY> }
```

Example

```
#do{
```

```
  ::when(t!=2) tau
  ::when((t<=10)||(x==1)) tau {= Dummy=1 =}
  ::when(((t==1)&&(p==2))||((f<=2)&&(t==3))) tau
  ::else tau {= 2e2e2 =}
}

result:

do {
::when(t!=2) tau
::when((t<=10)||(x==1)) tau {= Dummy=1 =}
::when(((t==1)&&(p==2))||((f<=2)&&(t==3))) tau
::When(!(t!=2) && !((t<=10)||(x==1))
               && !(((t==1)&&(p==2))||((f<=2)&&(t==3)))
               &&  true) tau {= 2e2e2 =}
}
```

Grammar

```
\#do\W\{\W\N<MyWhenRec>=do\ \{$1@set{ElseGuard;}\n
\#alt\W\{\W\N<MyWhenRec>=alt\ \{$1@set{ElseGuard;}\n

MyWhenRec:<MyWhen><MyWhenRec>=$1\

MyWhen:\N\W\:\:<MyWhenConstr>\N=$1

MyWhenConstr:\Wwhen\W\(<T>\)\ <T>\N=:\:\Wwhen\($1\)\ $2\
     @append{ElseGuard;\!\(}\
     @append{ElseGuard;$1}\
     @append{ElseGuard;\) && }

MyWhenConstr:\Welse\W<T>=\:\:\WWhen\(\
     @append{ElseGuard; true}\
     @var{ElseGuard})\ $1\
     @set{ElseGuard;}\n
```

A.6 Probabilistic Events in Fault Trees

In particular useful for modelling fault trees' probabilistic events. Out of the probability provided in scientific format, a `palt` expression is generated that can directly be used in fault tree modelling.

Syntax

```
#fta [ <INT>E-<INT> :  <BODY> ]
```

Example

```
#fta [5E-2 : r2]

result:

palt{   //5E-2
        :95: tau
        :5: tau {= r2=1 =}
}
```

Grammar

```
\#fta\W[\W<D>E-<D>\W\:\W<A>\W]=\
     palt\{\t\/\/$1E-$2\n\t\:\
     @shell{echo -n ‘echo ’10\^@{$2} - @{$1}’ | bc -l -q‘}\
     \:\ tau\n\t\:$1\:\ tau\ \{\=\ $3\=1\ \=\}\n\}
```

A.7 Clock Arrays

The preprocessor has knowledge of two commands for clock arrays. The first generates an array of clocks of name <Identifier> and size <int>. The second can be used to reset the array of clocks to a predefined value, preferably 0. A weak point is that in order to reset all clocks, the user has to specify exactly the size of the array.

Syntax

```
clock <Identifier>[<int>]

#{= <Identifier> [<int>] = <INT> =}
```

Example

```
clock t2[8];
```

```
#{= t2[8]=0 =}

results:

clock t2_7;
clock t2_6;
clock t2_5;
clock t2_4;
clock t2_3;
clock t2_2;
clock t2_1;
clock t2_0;


{= t2_7=0, t2_6=0, t2_5=0, t2_4=0, t2_3=0, t2_2=0, t2_1=0, t2_0=0 =}
```

Grammar

```
\Wclock\W<I>[\W<D>\W]\;=\
@bind{counter;@sub{$2;1}}\
@repeat{@add{@var{counter};1};
        \ clock\ $1\_@var{counter}\;\n@decr{counter}}\
@unbind{counter}


\#\{\=\W<I>\W\[\W<D>\W\]\W\=\W<N>\W\=\}=\
@bind{counter;@sub{$2;2}}\
\{\=\ $1\_@sub{$2;1}\=$3\
@repeat{@add{@var{counter};1};
        \,\ $1\_@var{counter}\=$3@decr{counter}}\ \=\}\n\
@unbind{counter}
```

A.8 Forward Declaration Fix

The *forward declaration* rule consists of three parts. Before the preprocessing is started, a variable called `ProcessList` is initialized. For each processes definition found in the model, a variable with the respective process name is set to 1. Whenever a process is called, the name of process is used to check whether is has been already declared. If not, the process is added to the `ProcessList` which is later saved in a file (`ForwardProcessCalls.txt`) where is can be used for further processing. To fully adopt to this feature, the motor codes needs some modification.

Grammar

```
@bind{ProcessList;}

process\ <I>\W\(\)=$0\t@bind{$1;1}

<I>\(\)=$0@repeat{@cmps{$1;@var{$1};0;1;0};\
@write{ForwardProcessCalls.txt;process $0\;\n}}
```

A.9 Buffer Generation

The buffer generator is invoked, providing an argument specifying the desired buffer size. The interfaces to access the data are `DataIn` and `DataOut`, referring to data that should be inserted into the buffer, or data which just popped out of the buffer. This can as well be understood as two pointers that point to the head element that just dropped off the buffer, and the pointer to an elements that one wants to insert into the buffer.
Data insertion occurs by calling process `InBuffer()` and respectively `OutBuffer()` for taking an element out of the buffer.
In case of buffer overflows, action `BufferOverflow` is given, whereas in case of an empty buffer `BufferUnderrun` is issued. These actions can in turn be used to trigger other events.

Syntax

```
#MakeBuffer[<int>]
```

Example

```
#MakeBuffer[3]

------------------------------

//Begin of Buffer
action Element1, Element2, Element3;
action BufferOverflow;
action BufferUnderrun;
float DataIn;
float DataOut;

typedef struct{
        int i;          //number of elements
```

```
        //3-place Buffer
        float B_0;
        float B_1;
        float B_2;

} Buffer3;

Buffer3 mybuf;

process InBuffer(){

  alt{
     ::when(mybuf.i==0) Element1 {= mybuf.i+=1, mybuf.B_0=DataIn =}
     ::when(mybuf.i==1) Element2 {= mybuf.i+=1, mybuf.B_1=DataIn =}
     ::when(mybuf.i==2) Element3 {= mybuf.i+=1, mybuf.B_2=DataIn =}
     ::when(mybuf.i==3) BufferOverflow
        }
}

process OutBuffer(){

  alt{
     ::when(mybuf.i==0) BufferUnderrun
     ::when(mybuf.i==1) Element1 {= mybuf.i-=1, DataOut=mybuf.B_0 =}
     ::when(mybuf.i==2) Element2 {= mybuf.i-=1, DataOut=mybuf.B_0,
          mybuf.B_0=mybuf.B_1 =}
     ::when(mybuf.i==3) Element3 {= mybuf.i-=1, DataOut=mybuf.B_0,
          mybuf.B_0=mybuf.B_1, mybuf.B_1=mybuf.B_2 =}

        }
}
```

Grammar

```
\#MakeBuffer\W\[<D>\]=@bind{n;1}\/\/Begin\ of\ Buffer\naction\ \
Element@var{n}@incr{n}@repeat{@sub{$1;1};\,\ Element@var{n}\
@incr{n}}\;\naction\ BufferOverflow\;\naction\ BufferUnderrun\;\n\
float\ DataIn\;\nfloat\ DataOut\;\n\n\

@bind{n;0}typedef\ struct\{\n\tint\ i\;\t\t\/\/number\ \
of\ elements\n\t\t\n\t/\/$1\-place\ Buffer\n\

@repeat{$1;\tfloat\ B\_@var{n}@incr{n}\;\n}\n\}\ Buffer$1\;\n\n\
```

```
Buffer$1\ mybuf\;\n\n\

@bind{n;0}process\ InBuffer\(\)\{\n\n\talt\{\n@repeat{$1;\
\t\t\:\:when\(mybuf\.i\=\=@var{n}\)\ Element@add{@var{n};1}\
\ \{\=\ mybuf\.i\+\=1\,\ mybuf\.B\_@var{n}\=DataIn\ \=\}\n\
@incr{n}}\t\t\:\:when\(mybuf\.i\=\=@var{n}\)\ BufferOverflow\n\t\}\
\n\}\n\n\

@bind{n;1}@bind{mystr;DataOut=mybuf.B_0}process\ OutBuffer\(\)\
\{\n\n\talt\{\n\t\t\:\:when\(mybuf\.i\=\=0\)\ BufferUnderrun\n\
@repeat{$1;\t\t\:\:when\(mybuf\.i\=\=@var{n}\)\ Element\
@var{n}\ \{\=\ mybuf\.i\-\=1\,\ @var{mystr}\ \=\}\n@incr{n}\
@append{mystr;, mybuf.B_@sub{@var{n};2}=mybuf.B_@sub{@var{n};1}}}\
\n\t\}\n\}\n\n
```

A.10 Stack Generation

Interfaces provided for measuring overflows and respectively under runs are `StackOverflow` and `StackUnderrun`, defined as actions as in the Buffer-Scheme. Data is put on the stack by invoking processes `InStack()` and popped from the stack by process `OutStack()`. All else is equal to the buffer generation.

Syntax

```
#MakeStack[<int>]
```

Example

```
#MakeStack[3]

-------------------------------------

//Begin of Stack
action Element1, Element2, Element3;
action StackOverflow;
action StackUnderrun;
float DataIn;
float DataOut;

typedef struct{
        int i;          //number of elements
```

```
        //3-place Stack
        float S_0;
        float S_1;
        float S_2;

} Stack3;

Stack3 mystk;

process InStack(){

  alt{
    ::when(mystk.i==0) Element1 {= mystk.i+=1, mystk.S_0=DataIn =}
    ::when(mystk.i==1) Element2 {= mystk.i+=1, mystk.S_1=DataIn =}
    ::when(mystk.i==2) Element3 {= mystk.i+=1, mystk.S_2=DataIn =}
    ::when(mystk.i==3) StackOverflow
  }
}

process OutStack(){

  alt{
    ::when(mystk.i==0) StackUnderrun
    ::when(mystk.i==1) Element1 {= mystk.i-=1, DataOut=mystk.S_0 =}
    ::when(mystk.i==2) Element2 {= mystk.i-=1, DataOut=mystk.S_1 =}
    ::when(mystk.i==3) Element3 {= mystk.i-=1, DataOut=mystk.S_2 =}
  }
}
```

Grammar

```
\#MakeStack\W\[<D>\]=@bind{n;1}
     \/\/Begin\ of\ Stack\naction\ Element\
     @var{n}@incr{n}@repeat{@sub{$1;1};\,\ \
     Element@var{n}@incr{n}}\;\n\
     action\ StackOverflow\;\n
     action\ StackUnderrun\;\n
     float\ DataIn\;\n\
     float\ DataOut\;\n\n\

@bind{n;0}typedef\ struct\{\n
     \tint\ i\;\t\t\/\/number\ of\ elements\n\
     \t\t\n\t/\/$1\-place\ Stack\n@repeat{$1;\tfloat\ S\_@var{n}\
     @incr{n}\;\n}\n\}\
```

```
Stack$1\;\n\n\ Stack$1\ mystk\;\n\n\

@bind{n;0}process\ InStack\(\)\{\n\n\talt\{\n\
@repeat{$1;\t\t\:\:when\(mystk\.i\=\=@var{n}\)\ Element@add{\
@var{n};1}\ \{\=\ mystk\.i\+\=1\,\ mystk\.S\_@var{n}\=DataIn\
\ \=\}\n\
@incr{n}}\t\t\:\:when\(mystk\.i\=\=@var{n}\)\
\ StackOverflow\n\t\}\ \n\}\n\n\

@bind{n;1}process\ OutStack\(\)\{\n\n\talt\{\n\
\t\t\:\:when\(mystk\.i\=\=0\)\ StackUnderrun\n\
@repeat{$1;\t\t\:\:when\(mystk\.i\=\=@var{n}\)\ Element\
@var{n}\ \{\=\ mystk\.i\-\=1\,\ DataOut\=\
mystk\.S\_@sub{@var{n};1}\ \=\}\n\
@incr{n}}\n\t\}\n\}\n\n
```

A.11 Rate Conversion

A *MoDeST* model can be delayed according to an exponential rate using the prefix `#LAMBDA` or `#MU` whether it is a failure or repair rate. The preprocessor command (e.g. `#LAMBDAmyrate`) is using a defined constant `myrate` to impose a exponential delay with rate `myrate` at the appropriate position in the model. Thus a clear structure is obtained which make the model easy readable and enables beyond that conversion into other languages.
In the example below the execution between action a and b is delayed by rate `ECU2fail`. Clock and integer definition at the beginning of the corresponding process need manual investigation. To fully adopt for the *GEMA* grammar, some minor quests are still open, because clock and variable definitions are only admissible right after the process definition. These are not currently untented and kept as a challenge for further preprocessor investigations.

Syntax

```
float ECU2fail;

\#LAMBDAECU2fail
```

Example

```
float ECU2fail=1E-3;

a; \#LAMBDAECU2fail; b
```

```
-----------------------------------

clock LAMBDAclk_1;
float myrateTIMER;

a; {= myrateTIMER=Exponential(LAMBDAECU2fail), LAMBDAclk_1=0 =};
when(LAMBDAclk_1 == myrateTIMER) tau; b
```

Grammar

```
\#LAMBDA<I>=@incr{lambdaclkcounter}\
        clock\ LAMBDA_clk${lambdaclkcounter}\;\n\
        \tfloat\ LAMBDA$1TIMER\;\n\n\
        \{\=\ $1TIMER\=Exponential(LAMBDA$1)\,\
        \  LAMBDA_clk${lambdaclkcounter}=0\ \=\}\;\n\
        \twhen(LAMBDA_clk${lambdaclkcounter}\ === $1TIMER)\ tau\n

\#MU<I>=@incr{lambdaclkcounter}\
        clock\ MU_clk${lambdaclkcounter}\;\n\
        \tfloat\ MU$1TIMER\;\n\n\
        \{\=\ $1TIMER\=Exponential(MU$1)\,\
        \ MU_clk${lambdaclkcounter}=0\ \=\}\;\n\
        \twhen(MU_clk${lambdaclkcounter}\ === $1TIMER)\ tau\n
```

A.12 Failure Generation: Errors for Actions

A.12.1 Probabilistic Errors

Ways to insert erroneous actions into the model are given via `stuck (action a, p)` and `delay (action a, p, t, clk)` . The first way of adding failures is to stop on the execution of action `a` with probability `p`. Afterwards the process gets stuck. Another way is to use keyword `delay` as defined above to induce a lag of time `t` time units with probability `p`. `clk` refers to some clock which is not modified by the preprocessor but gives the error a notion of time without defining a new clock variable.

Syntax

```
stuck(action a, p)  delay(action a, p, t, clk)
```

Example

```
delay (action a, p, t,clk) ::=
      float a_timer;
      exception a_error;
      {= a_timer=t+clk =};
      palt{
         :p: when(c==t) urgent(true) throw a_error
         :1-p: a
      }


stuck (action a, p) ::=
      exception a_error;
      palt{
         :p: throw a_error;
         :1-p: a
      }
```

Grammar

```
action\W<T>\W{\#\W<ActionErrorType>\W\#}\;=action\ $1\;\ \/\/$2

ActionErrorType:STUCK,<D>=Stuck\ $1\%
ActionErrorType:DELAY,<D>,\[<D>,<D>\]=$1\ \
     \%\ Delay\ within\ \[$2,$3\]\ time\ units

DELAY\(\W<T>\W,\W<D>\W,\W\[\W<D>\W,\W<D>\W\]\)\;=
     \/\/----$1\ DELAY\ BEGIN-----\n\n
     clock\ $1\_clk\;\nfloat\ $1\_timer\;\n
     \{\=\ $1\_timer\=Uniform\($3\,$4\)\ \=\}\;\n
     palt\{\n\t\:$2\:\ when\($1\_timer\=\=$1\_clk\)\ $1\n
     \t\:@shell{echo -n ‘echo ’100-$2’ | bc -l -q‘}\:\ $1\n\}\;
     \/\/----$1\ DELAY\ END-----\n

STUCK\(<T>\W,\W<D>\W\)\;=\n
     \/\/----$1\ STUCK\ BEGIN-----\n
     action\ $1\_stuck\;\npalt\{\n
     \t\:$2\:\ $1\_stuck\n
     \t\:@shell{echo -n ‘echo ’100-$2’ | bc -l -q‘}\:\ $1\n
     \}\;\/\/----$1\ STUCK\ END-----\n
```

A.12.2 Exponential Errors

When dealing with exponential distributed errors the preprocessor commands from before have to be modified by replacing the probability p by a rate λ. This involves some minor changes within the preprocessor grammar since the component is suppose to fail now according to a exponential distribution.
For simplicity we stick close to the naming for probabilistic errors and define `delayexp (a, t, lambda)` and `stuckexp(a, lambda)` as the corresponding exponential versions of the former used keywords.

Example

```
delayexp (action a, t, lambda) ::=
      process a_ErrorProc(){
         clock clk;
         float timer;
         exception a_error;
         {= timer=Exponential(lambda) =};
         when(clk==timer) urgent(true) throw a_error {= clk=0 =};
         when(clk=t) a
      }

stuckexp (action a, lambda) ::=
      process a_ErrorProc(){
         exception a_error;
         clock clk;
         float timer;
         {= timer=Exponential(lambda) =};
         when(clk==timer) urgent(true) throw a_error
      }
```

Grammar

A.13 Failure Generation: Errors for Integers

A.13.1 Probabilistic Errors

An erroneous integer `x` is generated using `noise (float x, p, n)` or `rand(float x, p, n)`. `noise` induces a random fluctuation of magnitude `n` around a predefined value of variable `x`

that occurs with probability `p`. `rand` assigns a random value of magnitude `n` which occurs with probability `p` to variable `x`. Note that the possibility of obtaining a negative variable is possible is both cases.

Syntax

```
noise(float x, p, n)  rand(float x, p, n)
```

Example

```
{= noise(float x,p,n) =} ::=
      exception x_error;
      palt{
         :p/2: throw x_error {= x=x+Uniform(0,n) =}
         :p/2: throw x_error {= x=x-Uniform(0,n) =}
         :1-p: tau
      }


{= rand(float x,p,n) =} ::=  exception x_error;
      palt{
         :p: throw x_error {= x=Uniform(0,n) =}
         :1-p: tau
      }
```

Grammar

```
int\W<T>\=<D>\W{\#\W<IntErrorType>\W\#}\;=int\ $1=$2\ \/\/$3

IntErrorType:NOISE,<D>,\[<D>\]=Noise\ $1\%\ level\ $2
IntErrorType:RAND,<D>,\[<D>,<D>\]=Random\ $1\%\ with\ in\ \[$2,$3\]

RAND\(<T>\,<D>\)=$1\=@shell{echo -n ‘echo \$RANDOM | cut -b -$2‘}
NOISE\(<T>\,<D>\)=$1\=@shell{echo -n ‘echo \$RANDOM | cut -b -$2‘}
```

A.13.2 Exponential Errors

From the semantic side `delayexp` and `stuckexp` are similar with the difference that the simulation the first mentioned can possibly be continued after time t passed. Insertion of variable errors is done by `noise(x,lambda,n)` and `rand(x,lambda,n)` to account for noisy and random values.

Syntax

```
noiseexp(float x, lambda, n)  randexp(float x, lambda, n)
```

Example

```
noiseexp (float x,lambda,n) ::=
      process x_ErrorProc(){
         exception x_error;
         clock clk;
         float timer, noise;
         {= timer=Exponential(lambda) =};
         palt{
            :1: {= noise=x+Uniform(0,n) =};
            :1: {= noise=x-Uniform(0,n) =};
         }
         when(clk==timer) urgent(true) throw a_error {= x=noise =}
      }

randexp(float x,lambda,n) ::=
      process x_ErrorProc(){
         exception x_error;
         clock clk;
         float timer, noise;
         {= timer=Exponential(lambda), noise=Uniform(0,n) =};
         when(clk==timer) urgent(true) throw a_error {= x=noise =}
       }
```

Grammar

Appendix B

Synchronization Concepts

B.1 "One-to-Many" in *MoDeST*

	Mean Result	Confidence +/-
Sender1	66,492	0,3038
Receiver1	55,789	0,2145
Receiver2	39,808	0,1127
Receiver3	28,37	0,7852

Table B.1: Simulation results of One-to-Many synchronization.

```
//Counter for sending and received packets
int r1=0, r2=0, r3=0, s=0;

//Ready signals for the receivers
int ready1=0, ready2=0, ready3=0;


action sending_alone, sending_1_2_3;
action sending_1, sending_1_2, sending_1_3;
action sending_2, sending_2_3;                                          10
action sending_3;


process Sender(){
        clock t;
        float waiting;

        do {
                ::{= t=0, waiting=Uniform(0,3) =};
                alt{                                                    20
                ::when (ready1 == 0 && ready2 == 0 && ready3 == 0 && t==waiting)
                        urgent (true) sending_alone {= s+=1 =}
                ::when (ready1 == 0 && ready2 == 0 && ready3 == 1 && t==waiting)
                        urgent (true) sending_3 {= s+=1 =}
```

```
                ::when (ready1 == 0 && ready2 == 1 && ready3 == 0 && t==waiting)
                        urgent (true) sending_2 {= s+=1 =}
                ::when (ready1 == 0 && ready2 == 1 && ready3 == 1 && t==waiting)
                        urgent (true) sending_2_3 {= s+=1 =}
                ::when (ready1 == 1 && ready2 == 0 && ready3 == 0 && t==waiting)
                        urgent (true) sending_1 {= s+=1 =}
                ::when (ready1 == 1 && ready2 == 0 && ready3 == 1 && t==waiting)
                        urgent (true) sending_1_3 {= s+=1 =}
                ::when (ready1 == 1 && ready2 == 1 && ready3 == 0 && t==waiting)
                        urgent (true) sending_1_2 {= s+=1 =}
                ::when (ready1 == 1 && ready2 == 1 && ready3 == 1 && t==waiting)
                        urgent (true) sending_1_2_3 {= s+=1 =}
                }
        }
}

process Receiver1(){
        clock t;
        float waiting;

        do{
        ::{= t=0, waiting=Uniform(0,1) =};
                when (waiting==t) urgent (true) {= ready1=1 =} ; alt{
                ::sending_1         {= r1+=1, ready1=0 =}
                ::sending_1_2       {= r1+=1, ready1=0 =}
                ::sending_1_3       {= r1+=1, ready1=0 =}
                ::sending_1_2_3 {= r1+=1, ready1=0 =}
              }
        }
}


process Receiver2(){
        clock t;
        float waiting;

        do{
                ::{= t=0, waiting=Uniform(1,2) =};
                when (waiting==t) urgent (true) {= ready2=1 =}; alt{
                        :: sending_2        {= r2+=1, ready2=0 =}
            :: sending_1_2          {= r2+=1, ready2=0 =}
            :: sending_2_3          {= r2+=1, ready2=0 =}
                :: sending_1_2_3 {= r2+=1, ready2=0 =}
                   }
        }
}


process Receiver3(){
        clock t;
        float waiting;

        do{
                ::{= t=0, waiting=Uniform(2,3) =};
                when (waiting==t) urgent (true) {= ready3=1 =}; alt{
```

```
                    :: sending_3     {= r3+=1, ready3=0 =}
                    :: sending_1_3 {= r3+=1, ready3=0 =}
                    :: sending_2_3     {= r3+=1, ready3=0 =}
                    :: sending_1_2_3 {= r3+=1, ready3=0 =}
                }
        }
}

par{
        ::Sender()
        ::Receiver1()
        ::Receiver2()
        ::Receiver3()
}
```

B.2 "One-of-Many-to-One" in *MoDeST*

	Mean Result	Confidence +/-
Sender1	199,935	1,1586
Sender2	99,75	0,7743
Sender3	49,51	0,2864
Receiver1	98,845	0,7778

Table B.2: Simulation results of one-of-many-to-one synchronization.

```
int r1=0;
int s1=0, s2=0, s3=0;

action send_1, send_2, send_3;
action send_1_alone, send_2_alone, send_3_alone;

process Sender1(){
        clock t;
        float wait;

        do {
                ::{= t=0, wait=Uniform(0,1) =};
                        when (wait == t) alt{
                                                        ::send_1
                                                        ::send_1_alone
                                                };      {=   s1+=1 =}
        }
}

process Sender2(){
        clock t;
        float wait;
```

```
        do {
                ::{= t=0, wait=Uniform(0,2) =};
                        when (wait == t) alt{
                                                                        ::send_2
                                                                        ::send_2_alone
                                                                };      {=  s2+=1 =}
        }
}

process Sender3(){
        clock t;
        float wait;

        do {
                ::{= t=0, wait=Uniform(1,3) =};
                        when (wait == t) alt{
                                                                        ::send_3
                                                                        ::send_3_alone
                                                                };      {=  s3+=1 =}
        }
}

process Receiver1(){
        clock t;
        float wait;
        do{
                ::{= t=0, wait=Uniform(0,1) =};
                        when (t==wait) urgent (true) alt{
                                                                                ::send_1
                                                                                ::send_2
                                                                                ::send_3
                                        }; {=   r1+=1 =}
        }
}

par{
        ::Sender1()
        ::Sender2()
        ::Sender3()
        ::Receiver1()
}
```

B.3 "One-of-Many-to-Many" in *MoDeST*

```
int r1=0, r2=0, r3=0;
int s1=0, s2=0, s3=0;

action snd1_rcv1, snd1_rcv2, snd1_none, snd1_rcv12;
action snd2_rcv1, snd2_rcv2, snd2_none, snd2_rcv12;
```

	Mean Result	Confidence +/-
Send1	66,492	0,3038
Send2	55,786	0,2145
Receive1	39,808	0,1127
Receive2	28,37	0,0785

Table B.3: Simulation results of one-of-many-to-many synchronization concept.

```
process Sender1(){
clock t;
float wait;

do {
        ::{= t=0, wait=Uniform(0,1) =};
                when (wait == t) urgent(true)
                alt{
                        ::snd1_none {= s1+=1 =}
                        ::snd1_rcv1 {= s1+=1 =}
                        ::snd1_rcv2 {= s1+=1 =}
                        ::snd1_rcv12 {= s1+=1 =}
                }
}
}



process Sender2(){
clock t;
float wait;

do {
        ::{= t=0, wait=Uniform(0,2) =};
                when (wait == t) urgent(true)
                alt{
                        ::snd2_none {= s2+=1 =}
                        ::snd2_rcv1 {= s2+=1 =}
                        ::snd2_rcv2 {= s2+=1 =}
                        ::snd2_rcv12 {= s2+=1 =}
                }
}
}



process Receiver1(){
clock t;
float wait;

do{
        ::{= t=0, wait=Uniform(0,1) =};
                when (t==wait) urgent (true)
                        alt{
                                ::snd1_rcv1 {=    r1+=1 =}
                                ::snd2_rcv1 {=    r1+=1 =}
```

```
                                        ::snd1_rcv12 {=    r1+=1 =}
                                        ::snd2_rcv12 {=    r1+=1 =}
                                }
}
}
                                                                                    60
process Receiver2(){
clock t;
float wait;

do{
        ::{= t=0, wait=Uniform(0,2) =};
                when (t==wait) urgent (true)
                        alt{
                                        ::snd1_rcv2 {=    r2+=1 =}
                                        ::snd2_rcv2 {=    r2+=1 =}                  70
                                        ::snd1_rcv12 {=   r2+=1 =}
                                        ::snd2_rcv12 {=   r2+=1 =}
                                }
}
}

par{
        ::Sender1()
        ::Sender2()
        ::Receiver1()                                                               80
        ::Receiver2()
}
```

Appendix C

MatLab Simulink

M-file being used to execute the simulation and capture the desired values.

```
%Creating Data Structure with Mem Alloc
Result.times=[];
Result.EnvA=[];
Result.EnvB=[];
Result.Fire=[];
Result.dimensions=3;
randA.time=[];
randA.signals.dimensions=1;
randB.time=[];
randB.signals.dimensions=1;
distr=zeros(1,101)';

%Opening Simulink File
load_system('ecu2_v2');

for i=1:2000
    %Pre Simulation
    randA.signals.values=RANDOM('Uniform', -0.4, 2.4, 200, 1);
    randB.signals.values=RANDOM('Uniform', -0.4, 1.4, 200, 1);

    %Running Simulation
    cvsim ecu2_v2;

    %Post Simulation - Normal
    OldEnv=Result.EnvA;
    temp=ScopeData.signals(1).values;
    Result.EnvA=[OldEnv, temp];

    OldEnv=Result.EnvB;
    temp=ScopeData.signals(2).values;
    Result.EnvB=[OldEnv, temp];

    OldEnv=Result.Fire;
    temp=ScopeData.signals(3).values;
    Result.Fire=[OldEnv, temp];
end;
```

Appendix D

Markov Chain Analysis

D.1 *MoDeST* Code

```
// Markov Model complete MOdel!

//states
action sOK, sF, sX, sIF, sFNI;

extern const float LambdaFI;
extern const float LambdaI;
extern const float LambdaFNI;
extern const float PTFD;
extern const float Mu;
extern const float LambdaE;

float LambdaTFE;

//sojourn times according to edges clockwise
float t1, t2, t3, t4;

//Locations
int Location;
float TimeBeforeX;

//P(X)|9000h - Prop that one airbag fails after 9.000 hours
int PofXat9000;

int AOK, AF, AIF, AFNI, AX;
float TOK, TF, TIF, TFNI, TX;

process run(){

        clock t;
        clock runtime;

        {= LambdaTFE=4.0E-14 , Location=0, runtime=0 =};
        do{
```

```
                ::when(Location==0)        //OK
                        sOK {= AOK+=1,
                                t1=Exponential(LambdaTFE),
                                t2=Exponential(LambdaFI),
                                t3=Exponential(LambdaFNI),
                                t4=Exponential(LambdaI)=};
                        alt{
                                //LambdaTFE Transisions
                                ::when(t>t1) {= TOK+=t =}; {= Location=4, t=0 =}
                                ::when(t>t2) {= TOK+=t =}; {= Location=1, t=0 =}
                                ::when(t>t3) {= TOK+=t =}; {= Location=3, t=0 =}
                                ::when(t>t4) {= TOK+=t =}; {= Location=2, t=0 =}
                        }

                ::when(Location==1)        //F
                        sF {= AF+=1,
                        t1=Exponential(LambdaE),
                        t2=Exponential((LambdaI+LambdaFNI)),
                        t3=Exponential(Mu) =};
                        alt{
                                ::when(t>t1) {= TF+=t =}; {= Location=4, t=0 =}
                                ::when(t>t2) {= TF+=t =}; {= Location=3, t=0 =}
                                ::when(t>t3) {= TF+=t =}; {= Location=0, t=0 =}
                        }

                ::when(Location==2)        //IF
                        sIF {= AIF+=1,
                        t1=Exponential((LambdaFI+LambdaFNI)),
                        t2=Exponential(LambdaTFE) =};
                        alt{
                                ::when(t>t1) {= TIF+=t =}; {= Location=3, t=0=}
                                //LambdaTFE Transisions
                                ::when(t>t2) {= TIF+=t =}; {= Location=4, t=0 =}
                        }

                ::when(Location==3)        //FNI
                        sFNI {= AFNI+=1,
                        t1=Exponential(LambdaE) =};
                        alt{
                                ::when(t>t1) {= TFNI+=t =}; {= Location=4, t=0, AFNI-=1 =}
                        }

                ::when(Location==4)        //X
                        alt{
                                ::when(runtime<9000) {= PofXat9000=1 =}
                                ::when(runtime>9001) {= PofXat9000=0 =}
                        }; sX {= AX+=1, TX=t, TimeBeforeX=runtime =}; break
        }
}

run()
```

D.2 Analytical Results

The following table shows the numbers obtained by matrix exponentiation:

Time	Probability
7.450000e+03	7.500815e-09
8.000000e+03	8.167936e-09
8.500000e+03	8.789248e-09
9.000000e+03	9.425010e-09
9.500000e+03	1.007503e-08
1.000000e+04	1.074010e-08
1.050000e+04	1.141969e-08
1.100000e+04	1.211485e-08
1.150000e+04	1.282534e-08
1.200000e+04	1.355070e-08
1.250000e+04	1.429236e-08

Appendix E

Fault Tree Analysis

```
clock gt;
float TETime;
int TE;


float     ExpEv1=Exponential(1.72E−10),
          ExpEv2=Exponential(1.72E−9),
          ExpEv3=Exponential(1.72E−9);

float     ExpEv6=Exponential(1.092E−9),                                    10
          ExpEv4=Exponential(1.4E−10),
          ExpEv5=Exponential(1.4E−10);


int ORGate4=0;

process Proc_ORGate4(){

    int r1, r2, r3, r4;
    clock t;                                                               20

    par{
        ::when (t==ExpEv6)tau   {= r1=1=}
        ::tau; #FTA [ 1E−6: r2 ]
        ::when(t==ExpEv4) tau   {= r3=1=}
        ::when(t==ExpEv5) tau   {= r4=1=}
        ::when((r1==1 && r2==1) || r3==1 || r4==1)
            tau {= ORGate4=1 =}; break
    }
}                                                                          30


int ANDGate1=0;

Process Proc_ANDGate1(){

    int r2, r3, r4;
    clock t;
```

```
    par{
        ::Proc_ORGate4()
        ::when (t==ExpEv2) tau {= r2=1 =}
        ::when (t==ExpEv3) tau {= r3=1 =}
        ::#FTA[ 1E-2: r4 ]                    }
        ::when( (ORGate4==1 || r2==1) && (r3==1 || r4==1) )
            tau {= ANDGate1=1 =}; break
    }
}


int ORGate1=0;

process Proc_ORGate1(){

    int r1, r2, r3, r4, r5, r6, r7;

    par{

      palt{
          ::#FTA[ 1E-14 : r1]
          ::#FTA[ 1E-12 : r2]
          ::#FTA[ 1E-14 : r3]
          ::#FTA[ 1E-12 : r4]
          ::#FTA[ 1E-12 : r5]
          ::#FTA[ 1E-12 : r6]
          ::#FTA[ 1E-12 : r7]
          ::when(r1==1 || r2==1 || r3==1 || r4==1 || r5==1 || r6==1 || r7==1)
            tau {= ORGate1=1 =}; break
      }
}


process Proc_Main(){

    clock t;
    int r1;

    par{
        ::when(t==ExpEv1) tau   {= r1=1 =}
        ::Proc_ORGate1()
        ::Proc_ANDGate1()
        ::when(r1==1 || ORGate1==1 || ANDGate1==1)
            tau {= TE=1, TETime=gt =}; break
    }
}


//Run Simulation
Proc_Main()
```

Appendix F

Importance Analysis

```
int TopEvent=0;
int ImpA, ImpB, ImpC, ImpD;
int A=0, B=0, C=0, D=0;


process RunNoFail(){

        par{
                ::tau #FTA [ 1E-1: A]
                ::tau #FTA [ 2E-2: B]
                ::tau #FTA [ 3E-3: C]
                ::tau #FTA [ 5E-1: D]
        }

}


process GuardAll(){

        int LocalGate1, LocalGate2;

        par{
                ::when(A==1 || B==1) tau {= LocalGate1=1 =}
                ::when(C==1 || D==1) tau {= LocalGate2=1 =}
                ::when(LocalGate1==1 && LocalGate2==1) tau {= TopEvent=1 =}
        }
}


process GuardImpA(){

        int LocalGate1, LocalGate2;
        par{
                ::when(B==1) tau {= LocalGate1=1 =}
                ::when(C==1 || D==1) tau {= LocalGate2=1 =}
                ::when(LocalGate1==1 && LocalGate2==1) tau {= ImpA=1 =}
        }
}
```

```
process GuardImpB(){

        int LocalGate1, LocalGate2;
        par{
                ::when(A==1) tau {= LocalGate1=1 =}
                ::when(C==1 || D==1) tau {= LocalGate2=1 =}
                ::when(LocalGate1==1 && LocalGate2==1) tau {= ImpB=1 =}
        }
}

process GuardImpC(){

        int LocalGate1, LocalGate2;
        par{
                ::when(A==1 || B==1) tau {= LocalGate1=1 =}
                ::when(D==1) tau {= LocalGate2=1 =}
                ::when(LocalGate1==1 && LocalGate2==1) tau {= ImpC=1 =}
        }
}
process GuardImpD(){

        int LocalGate1, LocalGate2;
        par{
                ::when(A==1 || B==1) tau {= LocalGate1=1 =}
                ::when(C==1) tau {= LocalGate2=1 =}
                ::when(LocalGate1==1 && LocalGate2==1) tau {= ImpD=1 =}
        }
}

//Run Simulation
par{
        ::RunNoFail()
        ::GuardAll()
        ::GuardImpA()
        ::GuardImpB()
        ::GuardImpC()
        ::GuardImpD()
}
```

Appendix G

Fault Tree Generation

```
//Sync actions
action Event1, Event2, Event3;


//No Sync actions, help to discover errors in trace path
exception Event1_error, Event2_error, enable_error, Event3_error;

//variables for the broadcasting feature
int ready_approver=0;

//Errors of SIST
float FrontBagDelay=200;
float BeltTensionerDelay=60;
exception NoiseBeltTensioner_error;
exception NoiseFrontBag_error;


process NoiseBeltTensioner_Process(){

        clock clk;
        float timer, noise;

        {= timer=Exponential(1E-4) =};
        palt{
                :1: {= noise=BeltTensionerDelay+Uniform(0,30) =}
                :1: {= noise=BeltTensionerDelay-Uniform(0,30) =}
        };
        when(clk==timer) urgent(true) throw NoiseBeltTensioner_error; {= BeltTensionerDelay=noise =}
}

process NoiseFrontBag_Process(){

        clock clk;
        float timer, noise;

        {= timer=Exponential(5E-5) =};
        palt{
                :1: {= noise=FrontBagDelay+Uniform(0,100) =}
```

```
                :1: {= noise=FrontBagDelay−Uniform(0,100) =}
        };
        when(clk==timer) urgent(true) throw NoiseFrontBag_error; {= FrontBagDelay=noise =}
}

...

process Approver()          {

    ...
        //—-RAND(enable,10,1) BEGIN—-
        palt{
            :10: throw enable_error
            :90: tau
        };
        //—-RAND END—-
    ...

}

process Env()     {

    ...

        //—-DELAY(Event1,10,5,t1) BEGIN—-
        {= Event1_timer=Uniform(0,5)+t1 =};
        palt{
            :10: when(Event1_timer==t1) urgent(true) throw Event1_error
            :90: Event1
        };
        //—-Event1 DELAY END—-
    ...
        //—-STUCK(Event2,4) BEGIN—-
        palt{
             :4: throw Event2_error; when(false) Event2 {= time1=waiting =}
            :96: Event2 {= time1=waiting =}
        };
        //—-Event2 STUCK END—-
    ...
        //—-DELAY(Event3,5,19,t1,{= time2=t1 =}) BEGIN—-
        {= Event3_timer=Uniform(0,19)+t1 =};
        palt{
             :5: when(Event3_timer==t1) urgent(true) throw Event3_error;        Event3 {= time2=t1 =}
            :95: Event3 {= time2=t1 =}
        };
        //—-Event3 DELAY END—-

    ...
}

try{
        par{    ::NoiseBeltTensioner_Process()
                ::NoiseFrontBag_Process()
                ::FrontBag()
```

```
                ::BeltTensioner()
                ::Approver()
                ::Env()
                ::hide{violated} Observer()
        }                                                                    100
}
catch(Event1_error){
        break
}
catch(Event2_error){
        break
        }
catch(enable_error){
        break
        }                                                                    110
catch(Event3_error){
        break
}
catch(NoiseBeltTensioner_error){
        break
}
catch(NoiseFrontBag_error){
        break
        }
```

Appendix H

Single Source Vision

H.1 Behavior Model

```
/*
  MoDeST Behavior Model
*/
process Sensor()
#SwitchFailure(LambdaSwitch, MuSwitch)
#in: int pressure, bool power
#out: bool switch
{
  do{
      ::alt{
          ::when(power==1 && pressure<min) {= Switch=1 =}
          ::when(power==1 && pressure>max) {= Switch=0 =}
        }
  }
}

process Tank()
#TankFailure(LambdaTank, 0)
{
  do{
      ::water_in
      ::water_out    {= pressure-- =}
      ::when(pressure>max_pressure) ExplodePressure
   }
}

process Pump()
#PumpFailure(LambdaPump, MuPump)
#in: bool power, bool switch
#out: int pressure
{
  do{
      ::alt{
        ::when(switch==1 && power==1) water_in  {= pressure++ =}
```

```
        ::when(switch==0 && power==1) tau
      }
  }
}

process Power()
#PowerFailure(LambdaPower, MuPower)
#out: power
{
  clock t;
  float tfail, trepair;

  do{
    ::{= t=0, tfail=Exponential(LambdaPower), trepair=Exponential(MuPower) =};
      alt{
        ::when (t=repair && power==0) {= power=1 =}
        ::when (t=tfail && power==1) {= power=0 =}
      }
  }
}

process Consumer()
{
  clock t;
  float timer;
  do{
    ::{= t=0, timer=Exponential(LAMBDAconsumer) =};
        when(t=timer) urgent(true) water_out
  }
}

par{
    #MissingVCC (LambdaPower, MuPower)
    # process(Sensor) OR process(Tank) OR process(Pump) OR process(Power) -> TopEvent
    ::Consumer()
    ::Sensor()
    ::Tank()
    ::Pump()
    ::Power()
}
```

H.2 STA Markov Chain

Pseudo-CTMC equivalent expressed in terms of *MoDeST*

```
int MAX=100;
int MIN=10;
int pressure;


process Pump(){
```

```
  clock clk;
  int location;
                                                                    10
do{
 ::alt{
    ::when(location==0) //power=1, switch=1;
        alt{
            ::#LAMBDAvcc; {= location=1 =}
            ::when(switch==0) {= location=2 =}
            ::#LAMBDApump; {= location=4 =}
            ::when(clk==1) {= pressure++, clk=0 =}
        }
    ::when(location==1) #MUvcc; {= location=0 =}//power=1, switch=0   20
    ::when(location==2)//power=0, switch=1
        alt{
            ::#LAMBDAvcc; {= location=3 =}
            ::when(switch==1) {= location=0 =}
            ::#LAMBDApump; {= location=4 =}
        }
    ::when(location==3) #MUvcc; {= location=0 =} //power=0, switch=0
    ::when(location==4)   #MUpump; //PumpFailure
    alt{
      ::when(switch==1) {= location=0 =}                            30
      ::when(switch==0) {= location=1 =}
    }
 }
}

}

process Sensor(){

  int location;                                                     40

do{
  ::alt{
    ::when(location==0) //power=1, switch=1
        alt{
            ::#LAMBDAvcc; {= location=2 =}
            ::when(pressure>=MAX) {= location=1 =}
            ::#LAMBDAsensor {= location=4 =}
        }
    ::when(location==1) #MUvcc; {= location=0 =} //power=0, switch=1  50
    ::when(location==2) //power=1, switch=0
        alt{
            ::#LAMBDAvcc
            ::when(pressure<MIN) {= location=0 =}
            ::#LAMBDAsensor; {= location=4 =}
        }
    ::when(location==3) #MUvcc; {= location=2 =} //power=0, switch=0
    ::when(location==4) #MUsensor; //Sensor Failure
        alt{
          ::when(switch=0) {= location=2 =}                         60
          ::when(switch=1) {= location=0 =}
        }
  }
}
```

```
}

process Tank(){

  int location;

do{
  ::alt{
    ::when(location==0) tau; when(pressure==MAX) {= location=1 =}
    ::when(location==1) TankFull; alt{
                                    ::when(pressure<MAX) {= location=0 =}
                                    ::#LAMBDAburst {= location=2 =}
                                  }
    ::when(location==2) TankBurst;
  }
}

}


process Consumer(){

  do{
    ::#LAMBDAconsume;
      alt{
        ::when(pressure==0) NoWater
        ::else {= pressure-- =}
      }
  }
}
```

Bibliography

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. Technical report, 1990.

[Amb02] Scott W. Ambler. Single Source Information: An Agile Practice. Technical report, http://www.agilemodeling.com/, 2002.

[Aue05] Marko Auerswald. SRS ECU Behavior Modeling, IST Project AMETIST. Technical report, Bosch Industrial Case Study, March 2005.

[BDL02] G. Behrmann, A. David, and K.G. Larsen. A Tutorial on Uppaal. Technical report, Department of Computer Science, Aalborg University, Denmark, 2002.

[BHB+04] Matthias Bretschneider, Hans-Jürgen Holberg, Eckard Böde, Ingo Brückner, Thomas Peikenkamp, and Karriet Spenke. Model-based Safety Analysis of a Flap Control System. Technical report, ICOSE 2004 - 14th Annual International Symposium Proceedings, 2004.

[BHKK03] Henrik Bohnenkamp, Holger Hermanns, Jost-Pieter Katoen, and Ric Klaren. The MoDeST Modeling Tool and Its Implementation. Technical report, 2003.

[Bra01] *Kraftfahrzeugtechnik*, volume 2. Vieweg Handbuch, 2001.

[Bry87] R. E. Bryant. Graph based algorithms for Boolean function manipulation. Technical Report 35 (8), IEEE Transactions on Computer, 1987.

[Buc00] Kerstin Buchacker. Definition und Auswertung erweiterter Fehlerbäume für die Zuverlässigkeitsanalyse technischer Systeme. Technical report, Dissertation, Arbeitsberichte des Instituts für Informatik Universität Erlangen, Juli 2000. Band33, Nummer 3.

[DHKK01] P.R. D'Argenio, H. Hermanns, J.P. Katoen, and R. Klaren. *MoDeST - a modelling and description language for stochastic timed systems*. Springer, 2001. In Proc. PAPM-ProbmiV'01, LNCS 2165.

[Fau05] *FaultTree+*. Handbook and Demo-Licence at http://www.isograph-software.com/, 2005. Version 11.0.

[Gra03] David. N. Gray. *Gema - a general purpose macro processor*. http://gema.sourceforge.net, December 2003.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. Technical report, Science of Computer Programming 8, 1987.

[Her02] Holger Hermanns. *Interactive Markov Chains, and the quest for quantified quality*. PhD thesis, Lecture Notes in computer science, 2002. Springer, LNCS 2428.

[Hub03] Thomas Huber. Zuverlässigkeit und Ausfallsicherheit elektronischer Systeme im Automobil. Technical report, Robert Bosch GmbH, IIR-Konferenz, 2003. Safety First - For Intelligent Restraint System Technologies.

[LAR01] Jean-Claude Laprie, Algirdas Avizienis, and Brian Randell. Fundamental Concepts of Dependability. *Report N01145, LAAS-CNRS*, April 2001.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. *Uppaal in a Nutshell*. Int. Journal on Software Tools for Technology Transfer, October 1997.

[Mat05] The MathWorks, http://www.mathworks.com/. *Stateflow, Matlab Simulink Handbook*, 2005.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, 1989.

[MMT00] J. Mauss, V. May, and M. Tatar. Towards Model-based Engineering: Failure Analysis with MDS. Technical report, ECAI-2000 Workshop on Knowledge-Based Systems for Model-Based Engineering, August 2000.

[Moc05] Ralf Mock. Risiko und Sicherheit von Netzwerken. Technical report, Laboratorium für Sicherheitsanalytik, Juni 2005. lecture notes.

[Neu81] Marcel F. Neuts. *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*. The John Hopkins University Press, 1981.

[San05] William H. Sanders. Model-based environment for Validation of System Reliability, Availability, Security and Performance. Möbius Manual, 2005. http://www.mobius.uiuc.edu.

[TD03] Zhihua Tang and Joanne Bechta Dugan. Minimal Cut Set/Sequence Generation of Dynamic Fault Trees. Technical report, University of Virginia, Charlottesville, 2003.

[WGRH81] W.E.Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. Fault Tree Handbook. Technical report, Office of Nuclear Regulatory Research, January 1981. U.S. Nuclear Regulatory Commission.