TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Flugsystemdynamik

Contributions to Model-Based Safety Assessment

Alexander Wille

Vollständiger Abdruck der von der Fakultät für Maschinenwesen der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender: Prof. Dr.-Ing. Oskar Haidn

Prüfer der Dissertation:
1. Prof. Dr.-Ing. Florian Holzapfel
2. Prof. Dr. Julien Provost

Die Dissertation wurde am 14.01.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Maschinenwesen am 07.08.2019 angenommen.

Contents

List of Figures **v**

List of Tables **vii**

Acronyms **ix**

1 Introduction **1**
1.1 Motivation 2
1.2 State of the art 2
1.3 Objective 5
1.4 Contributions 5
1.4.1 Realistic physical failure simulation 5
1.4.2 Curbing combinatorial explosion 5
1.4.3 Omnidirectional fault propagation and transformation 6
1.4.4 Conceptual overview 6
1.5 Outline 9

2 Context and Background **11**
2.1 Mathematical preliminaries 11
2.1.1 Modern probability theory 11
2.1.2 Boolean reasoning 13
2.1.3 Constraint Satisfaction Problems 17
2.2 Concepts of model-based system engineering 17
2.2.1 Terms and definitions 17
2.2.2 Safety assessment and hybrid systems 18
2.2.3 Hybrid systems in model-based engineering 18
2.2.4 Formal verification 20
2.2.5 Hybrid systems verification problem 23
2.3 Safety assessment 23
2.3.1 Legal framework and standards 24
2.3.2 Safety assessment process 25
2.3.3 Fault tree solution theory 26

3 Operational Semantics for Hybrid Failure Behavior **33**
3.1 Formal definition of a hybrid system for failure simulation 34
3.1.1 Continuous activity specification 34

3.1.2 Structural aspects . . . 34
3.1.3 Discrete switching logic specification . . . 36
3.1.4 Initial state, exceptions and inputs . . . 37
3.1.5 Intermittent faults . . . 38
3.1.6 Modeling considerations . . . 38
3.1.7 Transitions and fault simulation schedules . . . 39
3.1.8 Requirements towards behavioral failure modeling languages . . . 39
3.2 Implementation aspects . . . 41
3.3 Case study: Simulating a hybrid flight control system's failure behavior 42

4 Completeness of Fault Trees 45
4.1 Relation to the safety assessment process . . . 46
4.2 Functions of static failure logic . . . 46
4.3 Pessimism and fault tree completeness . . . 47
4.4 Properties of the proposed method . . . 48
4.5 Description of the proposed method . . . 55
4.6 Example . . . 58
4.7 Case study: Validation of the ARP 4761 wheel brake system example . . . 58

5 Fault Tree Generation 65
5.1 Capturing safety design intent . . . 65
5.1.1 Concept . . . 66
5.1.2 Formal definitions . . . 67
5.2 Fault tree template generation and logic supplementation . . . 69
5.2.1 Structural classification of fault trees . . . 69
5.2.2 Fault tree templates and structure-neutral modification . . . 71
5.3 Constraint Satisfaction Problem formulation and solution aspects . . . 76
5.3.1 Constraint Satisfaction Problem assembly . . . 76
5.3.2 Contradiction-free, efficient problem formulation . . . 77
5.3.3 Binary Decision Diagrams and fault tree refinement . . . 78
5.4 Implementation aspects . . . 78
5.5 Case study: Fault tree generation for a hybrid flight control system . . . 81

6 Conclusion 85
6.1 Hybrid failure simulation . . . 85
6.2 Fault tree pessimism . . . 86
6.3 Fault tree generation . . . 88
6.4 Review of the research objective and future research directions . . . 88

A Software Architecture of Custom Components I
A.1 Hybrid Failure Simulation . . . I
A.2 Fault Tree Pessimism . . . II
A.3 Fault Tree Generation . . . V

B HSim Modeling Guide **IX**
B.1 Hybrid log and setup-blocks IX
B.2 Hybrid failure behavior specification IX

C Java Environment Configuration **XI**

List of Figures

1.1 Concepts for the application of fault tree pessimism 7

2.1 Example for a piecewise-smooth hybrid trace 19
2.2 The quintuple of a hybrid system producing a run 20
2.3 BDD example, encoding $x_1 + x_2 \cdot x_3$ 28
2.4 Binary Decision Diagram (BDD) reduction 29
2.5 BDD path cube 30

3.1 hybridFCS model structure 42
3.2 Simulation results for hinge movement 44

4.1 Partitioning strategies 49
4.2 Probability-weighted BDD 51
4.3 Probability truncation 52
4.4 Concepts of fault tree pessimism 53
4.5 Probability-guided BDD evaluation of a BDD 56
4.6 Probability-guided expansion of a BDD 57
4.7 Reduced, probability-weighted BDD for $ab + a'c$ 58
4.8 Effect of probability truncation 62
4.9 Effect of added events 63

5.1 Example for a simple, flat fault tree 69
5.2 Example for a simple, fault propagation depth-ordered fault tree 70
5.3 Example for a simple, model hierarchy-ordered fault tree 71
5.4 Example for a model hierarchy fault tree template 72
5.5 Example for a pessimistic fault tree template 72
5.6 Example for the steps of supplementing Minimal Cut Sets (MCSs) 74
5.7 Automatically generated fault tree for control surface hardover 84

A.1 Class diagram for the implementation of Chapter 3 I
A.2 Activity diagram for the implementation of Chapter 3 III
A.3 Class diagram for the implementation of Chapter 4 IV
A.4 Class diagram for the implementation of Chapter 5 VI
A.5 Activity diagram for the implementation of Chapter 5 VIII

List of Tables

2.1 Overview on the notation of hybrid systems 21
2.2 Fault tree gates and their equivalent Boolean expressions 27

3.1 hybridFCS blocks and failure behaviors 43

4.1 Fault Tree Pessimism Example: Part I — Tracing 59
4.2 Fault Tree Pessimism Example: Part II — Expansion 60
4.3 Case study results 61

5.1 Notation of safety design intent annotation 68
5.2 Example for fault tree refinement 79
5.3 Case study — Tag propagation and transformation rules 83

Acronyms

AFCS Automatic Flight Control System

AMC Acceptable Means of Compliance

API Application programming interface

ASA Aircraft Safety Assessment

BCF Blake Canonical Form

BDD Binary Decision Diagram

CCA Common Cause Analysis

CDF Cumulative Distribution Function

CMA Common Modes Analysis

CSP Constraint Satisfaction Problem

CTL Computational Tree Logic

DAE Differential-Algebraic Equation System

DNF Disjunctive Normal Form

EASA European Aviation Safety Agency

FAA Federal Aviation Administration

FC Failure Condition

FCS Flight Control System

FDAL Function Development Assurance Level

FHA Functional Hazard Analysis

FMEA Failure Modes and Effects Analysis

FMES Failure Modes and Effects Summary

FPTN Failure Propagation and Transformation Notation

FSM Finite State Machine

FTA Fault Tree Analysis

GUI Graphical User Interface

HiP-HOPS Hierarchically Performed Hazard Origin & Propagation Studies

IDAL Item Development Assurance Level

IDE Integrated Development Environment

MCS Minimal Cut Set

PASA Preliminary Aircraft Safety Assessment

PDF Probability Density Function

POS Product of Sums

PSSA Preliminary System Safety Assessment

RTCA Radio-Technical Commission for Aeronautics

SAE Society of Automotive Engineers

SAML Safety Analysis and Modeling Language

SDK Software Development Kit

SOP Sum of Products

SSA System Safety Assessment

UAV Unmanned Aerial Vehicle

UML Unified Modelling Language

Chapter 1

Introduction

The dependency of modern aircraft on complex systems is critical to their safe operation. The responsibility to ensure that the design of these systems satisfies in effect the level of safety expected by passengers and other commercial users rests with aircraft design, production, maintenance and operation organizations. Among other control mechanisms, it is enforced by public aviation safety agencies through obligatory design certification. During aircraft and systems design, collecting evidence for conformance to a cascade of functional and technical requirements associated with the safe operation is — in some parts — repetitive and thus suitable for automation.

The utilization of software support for the purely bureaucratic portion of safety assessment has become commonplace in the aircraft industry and is not discussed in this work. The decision whether a system's plausible malfunction is sufficiently improbable as determined by the gravity of the malfunction's effects is at the heart of safety assessment. This decision is made through a series of (usually iterative) steps, formalized under the term *fault tree analysis*:

1. Ensure that the malfunction (called *failure condition* in this context) does not ensue from normal operation of the system

2. Identify which component faults or external effects can cause the malfunction under consideration, alone or in combination

3. Determine the probability that these causes occur, e.g. through reliability studies

4. Calculate the probability of the system malfunction as a function of the probabilities of its causes

The challenge lies in identifying plausible causes for a failure condition in the first two steps. Because normal operation by definition only consists of a number of well-known scenarios, the second step is more difficult: The combinatorial diversity of each plausible event interacting with each other set of events within and without the system makes bottom-up analysis intractable, so heuristics on system behavior need to be employed to narrow the search space of critical scenarios. In order to apply them, system behavior must be described formally.

For different reasons, executable mathematical models of system behavior have become popular for system design. Extending them for the purpose of automating cause-consequence analysis as a part of safety assessment promises synergy between the two fields, and is the context of this work and its contributions.

1.1 Motivation

Cause-consequence analysis in complex technical systems is challenging, not only because it commonly requires the combined effort of multiple experts in the domains of the components of the system, but also because it requires strong abstraction of system behavior. In the theoretical essence, knowledge about system behavior is translated into logical conditions for the occurrence of the failure condition under consideration. In practice, this is also an organizational interface between experts on safety assessment and experts on system architecture and design. Supporting these experts in their work would help them design even safer aircraft and systems, save them time for the portions of their work that cannot be automated, and eventually contributes to faster, safer innovation in the aerospace industry.

1.2 State of the art

At the heart of model-based safety assessment in the question which properties models need to have in order to be useful in practice, for both system designers and safety engineers. Joshi and Heimdahl [1] derive a list of requirements towards a behavioral fault modeling language, or, in the terms of this work, operational semantics for failure simulation in general. They also highlight a problem of great practical relevance when trying to model failure behavior with dataflow languages: Faults can alter the direction in which cause propagates to consequence throughout the modeled system architecture. Taking this into account in directional modeling languages requires the addition of feedback loops, is error-prone, tedious and leads to models with low intelligibility and maintainability. This, in turn, is prohibitive in iterative safety engineering and system design of safety-critical systems.

Differential-Algebraic Equation System (DAE)-based modeling languages, also referred to as physical or equation-based modeling languages, do not require explicit fault propagation path modeling. Mosterman and Biswas [2] propose a DAE-based modeling language, modeling framework and simulator called HyBrSim. Albeit fulfilling all the above requirements, there are two issues preventing it from practical application in model-based safety assessment:

- HyBrSim allows specifying transition conditions for hybrid mode change on the continuous system state *after* discrete reconfiguration, which requires calculating continuous system state after discrete reconfiguration at each time step, for each such transition, and thus renders the simulation of large models computationally prohibitively expensive.

- HyBrSim does not offer the powerful user interface, integration, and extensibility of commercial modeling environments, which are pivotal in their economically efficient use in industrial practice.

In the Modelica-community, Schallert [3] has presented a framework for model-based safety assessment called DMP[1]. As a behavioral fault modeling language, it has several drawbacks:

- Only one fault per model node is allowed, and only one model node per real system component is allowed, which limits the number of faults per real system component to one.

- Fault behavior is limited to parameter changes of the underlying DAE. Replacing equations related to the failed component is not allowed.

Schellhorn et al. [4] contributed a formal definition of safety assessment through fault trees which explains how deterministic cause-consequence analysis is the prerequisite of the probabilistic calculation of top-event probability. This has narrowed down the problem of model-based safety assessment to cause-consequence analysis in the underlying model, by clarifying its interface to top-event probability calculation. It also justifies rigorous alignment of model-based safety assessment to the well-defined safety assessment process in the aviation industry.

During the writing of this thesis, MathWorks released MATLAB 2017b (see [5]) which allows the user to model and simulate hybrid behavior with the Simscape package. This implementation does not suffer from the drawbacks of HyBrSim. Yet, it does not fulfill the requirements towards a behavioral fault modeling language: Nominal and failure behavior are not semantically distinguished, because hybrid modes do not carry such an annotation. Consequently, there is no aspect separation in models between nominal and failure behavior, which in turn promotes modeling errors. Furthermore, hybrid modes and transitions are defined via textual definition of transition conditions only. In the rare case of complex failure behavior transition logic, this is another risk for modeling errors because it may lack clarity.

Hybrid systems with DAE-models as their continuous portion have benign properties for modeling and simulation purposes in the context of behavioral failure modeling. But the expressiveness of hybrid systems comes at the cost of intractability of their verification even for simple properties and slightly more complex models [6]. This fundamental contribution of outlining where tractability and decidability end has made it clear that abstractions are indispensable for verifying hybrid systems with DAEs and more general requirements.

With their modeling language AltaRica, Arnold et al. [7] have limited dataflow-based languages exactly to what is tractable in formal verification. AltaRica is limited to language features which produce formally verifiable models, and has been successfully integrated into commercial products such as Safety Designer by Dassault Systèmes (a recast of Cecilia OCAS; now unavailable), and, in its newest version 3.0, SIMfia by Apsys [8].

[1]Only the acronym is given in the paper.

Qualitative abstractions of failure behavior have proven flexible and fruitful in practical application, even though the creation of the abstraction is manual for general DAE-based modeling languages. Qualitative Deviation Models [9] abstract system behavior to discrete models through a concept inspired by state space model linearization in control systems. At the time of writing of this work, the QSafe research group with participation of the Institute of Flight System Dynamics at the TU München investigated methods for automatically abstracting more dynamically expressive modeling languages into qualitative deviation abstractions.

Hierarchically Performed Hazard Origin & Propagation Studies (HiP-HOPS) is an evolution of Failure Propagation and Transformation Notation (FPTN) and abstracts nominal and failure behavior as a directional propagation and transformation of tags for faulty behavior throughout the topology of a behavioral model.

Uniting the strengths of the above approaches, smartIflow by Hönig [10] allows Finite State Machines (FSMs) for component behavior specification, and employs formal verification against safety requirements in Computational Tree Logic (CTL) for cause-consequence analysis. This work nevertheless concentrates on quantitative models due to their popularity in control systems design.

Quantitative abstractions are more problem-specific than their qualitative counterparts, but once found, allow tradeoffs between the computational effort of verification and the amount of pessimism they impose on the verification problem. The continuization of a specific class of hybrid systems has been demonstrated by Althoff [11] to allow pessimistic verification of systems whose transitions only involve linear state reconfiguration functions, or such that can be replaced by a linear hull of sufficient precision for the requirement under verification. Transitions from nominal to failure behavior, in general, do not have that property.

Testing, in the sense of systematically simulating the model and comparing results to requirements, can also provide valuable insights into system behavior, even though simulation results cannot be generalized to statements on system safety. Fainekos et al. [12] have presented S-TaLiRo for test-based verification of control systems. It proves the value of testing, regardless of whether formal verification is available. Because it only supports dataflow-based modeling languages, it depends on dedicated failure simulation models.

Given that no complete set of abstractions for a sufficiently general class of hybrid systems (with DAE-based continuous models and sufficiently general transitions) for formal verification exists, a reduction of the search space for eligible component faults in cause-consequence analysis is of interest. Because, in the worst case, each fault alone can trigger the failure condition under analysis, importance metrics from the probabilistic domain of fault tree analysis allow focusing analysis attention on faults with great probability contribution. Kuo and Zho [13] give an overview of available importance metrics. While some cover scenarios where multiple faults are active, none are directly linked to a fault tree generation or validation process that would allow it to guide the analysis.

1.3 Objective

The objective of this doctorate has been to advance model-based safety assessment in two ways:

1. Physical simulation of nominal and failure behavior based on an efficient modeling process that produces maintainable models
2. Formalization of the problem of fault tree generation and validation in a way that allows a divide-and-conquer-approach to formal verification of safety properties of complex, safety-critical systems

1.4 Contributions

The contributions of this work are in three related areas of model-based safety assessment, hybrid failure simulation, fault tree validation and fault tree generation from qualitative fault propagation and transformation annotation. Separately, they are of value to the field of model-based safety assessment, but their synergy is essential for their effectiveness in reference to the previously stated objectives of this work. This section first explains their separate value and, in the end, explains their common application in fault tree generation and validation.

1.4.1 Realistic physical failure simulation

The presented operational semantics for hybrid failure simulation is based on that of Mosterman and Biswas [2]. Instead of limiting continuous behavior to energy bond graph models, the presented framework allows the more general model class of DAEs. In contrast to their semantics, it does not allow post-conditions. Post-conditions for transition allow correctly capturing transition times in fixed time step simulation. Its drawback is that it renders the simulation of models in discrete states where many hybrid components are in modes with post-conditions prohibitively expensive. By employing a variable time step solver for the continuous portion of hybrid simulation, post-conditions are obsolete in the presented semantics. The presented operational semantics for hybrid simulation has been implemented in the MathWorks tool chain [5], and allows seamless integration of nominal and failure behavior specification. It thus is the first semantics and implementation that suffices all requirements for a behavioral failure modeling language and modeling/simulation framework as defined by Joshi and Heimdahl [1].

1.4.2 Curbing combinatorial explosion

The method for fault tree validation proposed in this work complements robustness metrics-driven testing concepts as implemented in S-TaLiRo by Fainekos et al. [12] by using the probabilistic portion of fault tree analysis for determining when the deterministic portion of cause-consequence analysis can be stopped. It also directly applies

to fault tree validation in review processes by guiding the analysis through the aspects of safety design encoded by the given fault tree in descending order of their contribution to top-level event probability. The underlying concept of fault tree pessimism, which states that the most pessimistic fault tree possible for any failure is that in which each component fault individually triggers the top event, is developed into a process of iterative fault tree refinement during fault tree generation. The method simplifies automation of fault tree generation by breaking down the necessary verification work into individually treatable tasks. It employs probability truncation for curbing combinatorial explosion already during fault tree generation. Additionally, it produces standard-conformant, human-readable fault trees when combined with structural models of the system under analysis. In these properties, it is superior to current concepts for fault tree generation, which fail to produce standard-conformant fault trees and suffer from combinatorial explosion.

1.4.3 Omnidirectional fault propagation and transformation

The last contribution is a method for fault tree generation that analyzes the model structure of DAE-based (omnidirectional) and dataflow-based models of nominal and failure behavior. Additional modeling semantics for fault propagation and transformation are introduced and exploited in automated fault tree generation. They require designers to directly capture safety design intent (such as failure propagation barriers or redundancies) in their model. This effort is repaid by using that information in automated fault tree generation.

1.4.4 Conceptual overview

In Figure 1.1, two applications of Chapter 4 are illustrated: The center lane with the right lane describes fault tree validation, and the center lane with the left lane describes fault tree generation.

In order to produce a complete fault tree (during generation) or assess completeness (during validation), both methods rely on a step where the fault tree and 'reality' are compared. It is assumed that real system behavior is not computationally accessible for all failure conditions, so an adequate model of it called *reference behavior* is assumed to be available. *Evaluating* it means that 'true' system behavior under given circumstances is determined. In this work, it is used to determine if some other, simpler model (such as a fault tree) is adequate for its limited purpose. Safety assessment allows simpler models to predict that the system would fail when in reality it does not. In such a case,the simpler model is called *pessimistically wrong* and the circumstance in which the error is relevant is called the *false positive*. The contrary case, when the simpler model predicts the system *not* to fail when in reality it does, is called *optimistically wrong*, and *false negative*, respectively. In fault tree generation and validation, the simpler model is the fault tree.

In fault tree generation, the combinations of faults in which the failure condition under analysis is predicted to occur by the fault tree are of interest. Such combinations of faults are called *positive scenarios*. If the 'reality check' for a particular positive

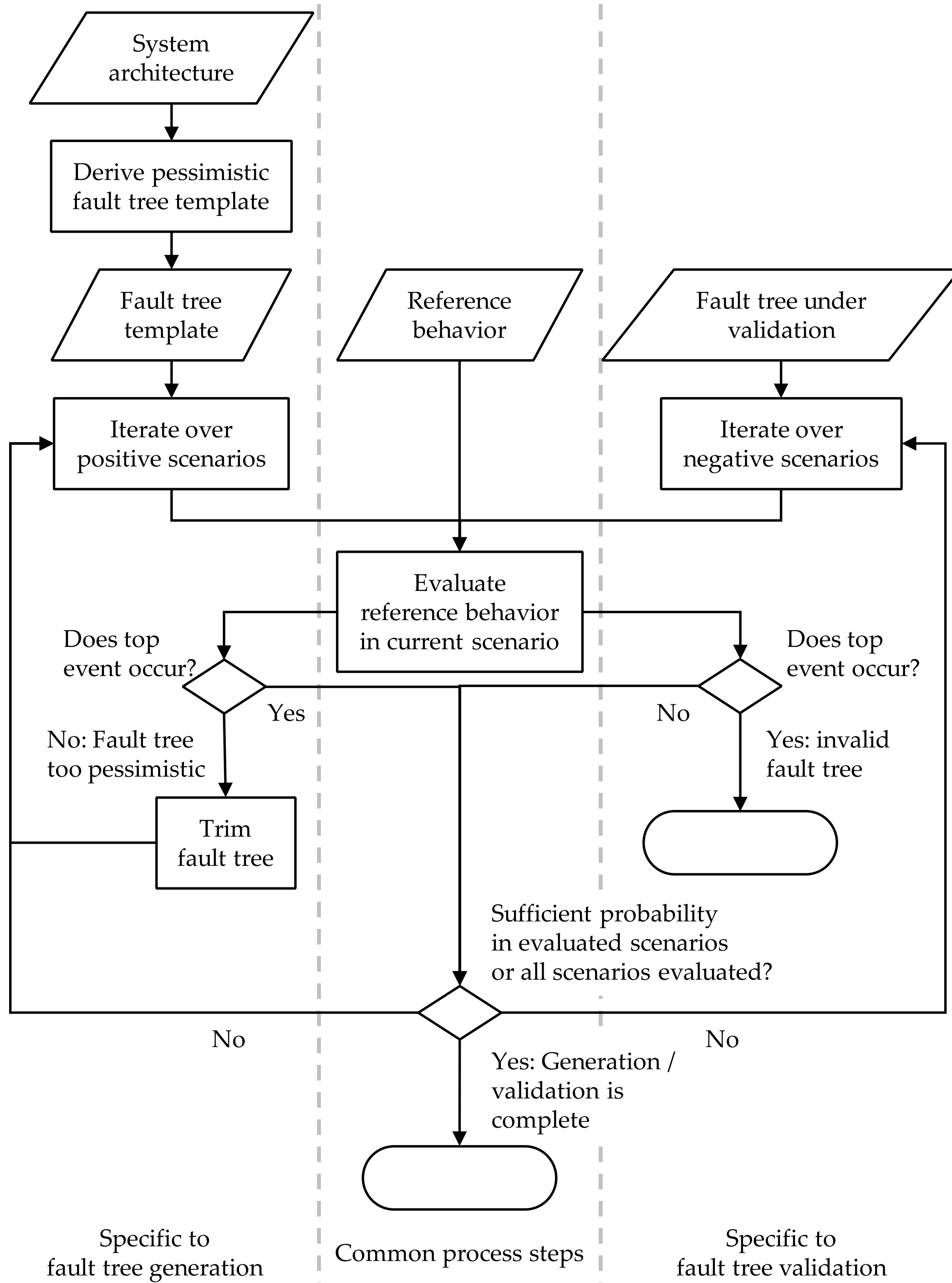

Figure 1.1: Concepts for the application of fault tree pessimism (see Chapter 4) for fault tree generation or validation

scenario of evaluating reference behavior for it yields that the fault tree's prediction is pessimistically wrong, the fault tree can be corrected or *trimmed* for it and make it less pessimistic. The loop of evaluation and, if possible, trimming is called *fault tree template refinement* and is continued until the fault tree's predicted probability for the top event to occur is below some threshold (e.g. the certification limit), or until all positive scenarios have been evaluated. This is illustrated in Figure 4.4 in Chapter 4.

In fault tree validation, *negative* scenarios, in which the fault tree predicts the failure condition *not* to occur, are of relevance. When a negative scenario is found to be a false negative, validation can be aborted and the result is that the fault tree is invalid, because it is optimistically wrong. When all negative scenarios have been evaluated or when the total probability of scenarios that have not yet been evaluated is below some threshold, the fault tree is valid.

Evaluating reference behavior is much more costly in manual labor or computational power than Boolean reasoning and probability calculations on the fault tree, so the core concept is to minimize the number of times that reference behavior needs to be evaluated, and make each evaluation as simple as possible. The simplicity of evaluations is promoted by how the space of the combinations of faults about which the fault tree reasons is partitioned. The number of evaluations is minimized through probability truncation of the iteration over the scenarios, i.e. iteration is aborted once sufficient probability has been accumulated.

In conventional safety assessment, the entirety of the processes of fault tree generation and validation is manual. In contrast to this, the processes illustrated in Figure 1.1 directly allows automation of all process steps except for the evaluation of reference behavior. The manual labor required for this step depends on which of the following types of behavioral description is employed for reference behavior:

1. Informal model: A set of documents that textually describe real system behavior for evaluation using engineering judgment only

2. Executable model: An executable model of system behavior that encompasses nominal and failure behavior (see Chapter 3)

3. Verifiable model: An abstraction of system behavior that is suitable for purely computational comparison to failure behavior as defined in the fault tree

Informal models require less manual labor without imposing greater manual effort for model preparation because no mathematical model of reference behavior is built. However, this renders automation impossible. Executable models allow simulating reference behavior, but require modeling effort to be spent beforehand. When simulation results can be evaluated in a way that they can be related to the Boolean logic of the fault tree, they can become validation rationale during fault tree validation. This variant combines the results of the Chapters 4 and 3. The third variant assumes that, through manual or automatic abstraction, reference behavior can be compared directly to what the fault tree expresses about a particular scenario. Chapter 5 introduces a manual abstraction method for this purpose.

1.5 Outline

The next chapter gives a brief introduction into the mathematical fields whose concepts are relevant for this work. The Chapters 3, 4 and 5 explain this work's contributions. In Chapter 3, an operational semantics for behavioral failure models is discussed, along with an implementation of a modeling framework and a simulator. Chapter 4 explains the algorithm for efficiently extracting failure scenarios with high probability contribution from fault trees. In Chapter 5, a concept for formally linking logical failure behavior as specified in fault trees and physical system behavior as specified in DAE-models is introduced, and the provided implementations of two applications of that concept in modeling and translation engines (from behavioral model to fault tree) are examined. The last chapter summarizes the results and proposes future research directions and next steps towards application in real aircraft programs.

Chapter 2

Context and Background

In this chapter, the mathematical preliminaries and legal context of this work are introduced briefly and with references to comprehensive sources on each subject. Its background in industrial practice is highlighted, as defined in standards and scientific contributions of outstanding significance. It serves as a dictionary for notations, terms, and definitions, and supplies the theoretical justifications for choices made in the main contributions of this work.

2.1 Mathematical preliminaries

This section explains the underlying mathematical concepts of this work. It assumes familiarity with the mathematical canon taught at TU München's mechanical engineering courses of study. Because this work lies at the interface of mechanical and software engineering, the relevant concepts of software engineering make up the major portion of the section.

2.1.1 Modern probability theory

Probability theory reasons about future events based on statistical data. This section introduces the concepts and notation of stochastic and statistic mathematics used in this work.

The fundamental element of probability theory is the stochastic *experiment*. It is defined by its repeatable procedure that results in a measurable *outcome*, denoted ω. Generally, multiple outcomes are possible and the procedure can be described as a random choice among the set of possible outcomes, called *sample space*, denoted Ω. One repetition or performance of the experiment is called a *trial*. Subsets of the sample space Ω are called a *events*. The natural case, where an event reflects a single outcome, is called an *elementary* or atomic event. Note that elementary events are always pairwise disjoint.

By applying (ideally complete) statistical data, any event can be attributed a *probability* measure, describing the fraction of trials in which an event will be observed for an infinite number of trials, denoted P. This imposes bounds on the probability mea-

sure, 1 for the event occurring in every trial (Kolmogorov's second axiom) and 0 for occurring in no trial (Kolmogorov's first axiom). Thus, $\mathrm{P}(\Omega) = 1$ and $\mathrm{P}(\emptyset) = 0$.

Events can be combined with the set-theoretic operators *union* $A \cup B$, *intersection* $A \cap B$ and *complement* $\overline{A}$. Useful for shorter notations is the *relative complement* which is equivalent to intersection with the complement $A \backslash B = A \cap \overline{B}$. Their probability is defined via the countable additivity property of measures: When the events $A_1, A_2, \dots$ are pairwise disjoint, e.g. $(\forall i \neq j)\ A_i \cap A_j = \emptyset$, the probability of their union is the sum of their probabilities.

$$\mathrm{P}\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} \mathrm{P}(A_i) \tag{2.1}$$

With eqn. 2.1, the probabilities of unions of events that are not disjoint can be calculated as well, as long as their intersections is known, through disjoint partition. Because of this, the probability of the intersection of events is used to classify their relationship. For disjoint or *mutually exclusive* events, which cannot occur together in one trial, the probability of their intersection is zero.

$$\mathrm{P}\left(\bigcap_{i=1}^{n} A_i\right) = 0 \tag{2.2}$$

When the probability of the intersection is the product of the individual probabilities, the events are called (mutually[1]) *independent*.

$$\mathrm{P}\left(\bigcap_{i=1}^{n} A_i\right) = \prod_{i=1}^{n} A_i \tag{2.3}$$

Computing the union of mutually independent events is more difficult than simply adding event probability. Such an approach would measure their intersection multiple times. To compensate for that, higher-order terms need to be corrected in an alternating pattern called the *inclusion-exclusion principle*:

$$\begin{aligned}\mathrm{P}\left(\bigcup_{i=1}^{n} A_i\right) = & \textstyle\sum_{1 \leq i \leq n} \mathrm{P}(A_i) \\ & - \textstyle\sum_{1 \leq i \leq j \leq n} \mathrm{P}(A_i) \cdot \mathrm{P}(A_j) \\ & + \textstyle\sum_{1 \leq i \leq j \leq k \leq n} \mathrm{P}(A_i) \cdot \mathrm{P}(A_j) \cdot \mathrm{P}(A_k) \\ & - \dots \\ & + (-1)^{n-1} \cdot \textstyle\prod_{1 \leq i \leq n} \mathrm{P}(A_i)\end{aligned} \tag{2.4}$$

The above algorithm has exponential complexity. However, ignoring higher order terms may produce an insignificant error when only low event probabilities are involved, which is why this is called the *rare event approximation*:

$$\mathrm{P}\left(\bigcup_{i=1}^{n} A_i\right) \leq \sum_{1 \leq i \leq n} \mathrm{P}(A_i) \tag{2.5}$$

[1] Note that events can be pairwise (or of higher order) independent without being mutually independent.

A sample space may be discrete or continuous. Continuous sample spaces have a infinite number of outcomes. The assignment of probabilities to their elementary events is not useful, as the probability for a random experiment to result in precisely one outcome in a dense sample space is generally[2] infinitely small. Instead the density of probability is defined as a function that maps from the continuous sample space to positive reals called *Probability Density Function (PDF)* and denoted f.

$$f(\omega) := \Omega \to \mathbb{R}_0^+ \tag{2.6}$$

The probability of non-elementary events can be obtained by integrating over the desired subset of the sample space. The integral from one bound of the sample space to a specific elementary outcome is called the *Cumulative Distribution Function (CDF)*. For a continuous sample space with the bounds $\{\omega_{\min}, \omega_{\max}\}$, it is denoted F.

$$F(\omega) := \Omega \to \mathbb{R}_0^+ \leq 1 = \int_{\omega_{\min}}^{\omega \leq \omega_{\max}} f(z)dz \tag{2.7}$$

Only such functions are valid PDFs which obey $\int_\Omega f(\omega)d\omega = 1$, Kolmogorov's second axiom. Because elementary events are by definition disjoint, eqn. 2.1 applies and can be used to compute the probability of arbitrary events $\subseteq \Omega$.

In safety assessment and reliability analysis, the stochastic experiment is the procedure of, beginning with a nominal system under study, measuring the *survival time* $\tau \in \Omega, \Omega = \mathbb{R}_0^+$ for which it operates before it fails in a specified way. The PDF in such an example gives the probability density of failure over operation time. The probability to have failed after a given time specified by the CDF. Its complementary event, having reached a specified survival time, is denoted R.

$$R(\tau) := \Omega \to \mathbb{R}_0^+ \leq 1 = 1 - F(\tau) \tag{2.8}$$

Of great practical relevance is the *failure rate*, denoted λ which specifies the conditional probability to fail at a specified time given the system under study has not failed before.

$$\lambda(t) = P(\tau = t | \tau \geq t) = \frac{f(t)}{R(t)} \tag{2.9}$$

This assumes a system that never returns from failed to nominal. For cases where maintenance or self-repair is to be taken into account, availability (in normal mode) depends not only on failure rate. It also depends on the time required for restoration to the nominal state. However, such aspects are irrelevant for Fault Tree Analysis (FTA): It enforces the pessimistic assumption that no maintenance or self-repair is conducted.

2.1.2 Boolean reasoning

Boolean reasoning is a useful formalization when manipulating fault trees. This work adopts the notation of [14], which is briefly summarized hereafter.

[2] With the Dirac delta distribution, a continuous sample space can be created in which all probability is lumped on finitely many outcomes, but such hybrid discrete-continuous distributions are not taken into account in this work.

Boolean reasoning is based on Boolean algebras. For the purpose of this thesis, only binary Boolean algebras are relevant. They are algebras that can be described with a quintuple $(\mathbb{B}, +, \cdot, 0, 1)$ and satisfy the following properties:

1. $\mathbb{B} = \{0, 1\}$ is a set, and $+$ and $\cdot$ are binary operations on $\mathbb{B}$.
2. Both $+$ and $\cdot$ commutate and both $+$ distributes over $\cdot$ and $\cdot$ over $+$.
3. 0 is the neutral operand for $+$ and 1 is the neutral operand for $\cdot$.
4. $'$ is the unary complement operator. There is a complementary element a' to every element in $a \in \mathbb{B}$, such that $a + a' = 1$ and $a \cdot a' = 0$.

Other binary Boolean algebras could rename the elements of the quintuple, e.g. many programming languages call the 0-value `false`, 1 `true`, $+$ `OR`, $\cdot$ `AND` and $\mathbb{B}$ `bool`. Without limitation to generality, this work uses only the former proposed quintuple.

This work introduces an additional rule for the sake of clarity: $\cdot$ is given priority in evaluation before $+$, which makes brackets around $\cdot$-terms obsolete. Finally, writing two variables without an operator is a shorthand notation for the $\cdot$ operator: $ab = a \cdot b$.

A *formula* is anything one can write down. A *well-formed*, *Boolean* formula is a non-empty composition of the operators $+$, $\cdot$ and $'$ with variables and the constant values 0 and 1, that — when all variables are replaced by a constant value — can be evaluated to either 1 or 0. They are denoted with capital letters and are followed by a list of variables in round brackets. An example is $F(x_1, x_2, x_3) \coloneqq x_1 + x_2 \cdot x_3$. This work considers only well-formed, Boolean formulas. A formula containing a relational operator $(=, \neq, \leq)$, is called a *predicate*. It yields a scalar result in $\mathbb{B}$. An example is $x_1 + x_2 \cdot x_3 = 1$. The greek letter μ is reserved for predicates in this work. Predicates form the bridge between formal logic and Boolean reasoning, allowing true/false-statements to be expressed in a Boolean algebra[3].

Assigning variables of formulas a specific value, i.e. replacing each occurrence of one or more variable with either 0 or 1, can be denoted in two ways:

1. By a Boolean product term that contains the variables to be replaced by 1, and the complement of the variables to be replaced by 0
2. If an order of variables is defined, by a Boolean product term that has 1 in the position of each variable to be replaced with 1, and 0 respectively — or the additional symbol d for variables not to be replaced at all

As an example, consider any Boolean formula with three variables a, b, c. Replacing all occurrences of a with 1, b with 0 and keeping c as it is, with the order of variables a, b, c, is denoted as follows in the two notations:

1. ab'
2. $10d$

[3]See [14] for a more formal differentiation between the predicate calculus of formal logics and Boolean reasoning.

In this work, the first notation is used because of its clarity. The software components employed for Binary Decision Diagram (BDD) calculation in Chapter 4 use the second notation. With an assignment $\mathbf{x} \in \mathbb{B}^n$, the cardinality of an assignment is the number of variables that is to be replaced, and is denoted $|\mathbf{x}| \leq n$.

An assignment's cardinality, denoted $|x|$ for an assignment denoted x, is the number of variables that are to be replaced by a constant value. In the first notation of the previous paragraph, this is equal to the number of symbols of the product term. In the second notation, it is the total number of variables minus the number of 'don't care'-assigned variables. As an example for both notations, consider $|ac| = 2 = |0d1|$.

Some forms of formulas have special designations because of their importance in various algorithms. Most notably, when it is a disjunction of conjunctions, it is said to be in Sum of Products (SOP) form. Similarly, when it is a conjunction of disjunctions, it is said to be in Product of Sums (POS) form. While the SOP form makes it easy to identify if a given assignment to the Boolean formula would simplify it to 1, the POS form does the same for the result 0.

Formulas are only considered equivalent when they contain identical sequences of symbols. *Functions* are more diverse: They can be expressed with any formula, as long as it evaluates to the same value for all possible assignments. This also means that two functions are considered equivalent when they evaluate to the same result for every possible assignment. Predicates can be interpreted as single-valued functions of the expression of the predicate simplifying to 0 or 1, thus predicates are equivalent to scalar functions. Just as formulas, we denote them with capital letters. In functions, not all symbols contribute to the result. Through absorption, some of them may be omitted. Because of this, formulas describing the same function can be of vastly differing length.

An important relation between Boolean functions is *inclusion*, denoted by the symbol $\leq$. A function P is included in a function Q when Q returns 1 when P does, and P returns 0 when Q does. This can be written concisely as

$$P(\mathbf{x}) \leq Q(\mathbf{x}) := (\forall \mathbf{x} \in \mathbb{B}^n)\; P(\mathbf{x}) \cdot Q'(\mathbf{x}) = 0 \tag{2.10}$$

The inclusion property can also be applied to assignments: When an assignment $\mathbf{y} \in \mathbb{B}^n$ includes an assignment $\mathbf{x} \in \mathbb{B}^n$, the latter can not be 1 for assignments where the former evaluates to 0:

$$\mathbf{x} \leq \mathbf{y} := xy' = 0 | \mathbf{x}, \mathbf{y} \in \mathbb{B}^n \tag{2.11}$$

This can be employed to define the *monotonicity* property.

$$F \,\text{is monotone} \Leftrightarrow \mathbf{x} \leq \mathbf{y} \implies F(\mathbf{x}) \leq F(\mathbf{y}) | \mathbf{x}, \mathbf{y} \in \mathbb{B}^n \tag{2.12}$$

A most influential theorem in Boolean algebra is the *expansion* theorem, also called *Shannon decomposition* in information science.

$$\begin{aligned} F(\mathbf{x}) = \;& x_i' \cdot F(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_{n-1}, x_n) \\ & + x_i \cdot F(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_{n-1}, x_n) \end{aligned} \tag{2.13}$$

Every Boolean function can be replaced by a disjunction: Each of the terms is itself a conjunction of one of the variables x_i and the function itself, but the variable in question replaced by a constant value. The term with the variable replaced with 0 in the function, contains the complement of the variable x_i'.

Before algorithmic symbolic manipulation took the place of manual calculation of Boolean functions, another identity was employed in the same applications where Boole's expansion theorem is employed today: The *consensus* theorem. For the sake of clarity, $\cdot$ is omitted in the following equation.

$$ab + a'c + bc = ab + a'c \tag{2.14}$$

The term bc is called the consensus or the resolvent of the others. Note that the right hand side is the expanded form of the function on the left hand side on the variable:

$$\begin{aligned} F(a,b,c) &= ab + a'c + bc \\ &= a \cdot (1 \cdot b + 0 \cdot c + bc) \quad + a' \cdot (0 \cdot b + 1 \cdot c + bc) \\ &= a \cdot (b + bc) \quad + a' \cdot (c + bc) \\ &= ab \quad + a'c \qquad \blacksquare \end{aligned} \tag{2.15}$$

An assignment which causes a function to evaluate to 1 is called an *implicant* of the function. The assignment does not need to cover all variables: The number of variables that are assigned a value is called the *order* of the implicant. An implicant not included in any other one (of lower order) is called a *prime implicant*, denoted π. The set of all prime implicants of a function is denoted PI, thus $\mathrm{PI} := \{\pi \,|\, F(\pi) = 1\}$ and $\mathbf{x} \notin \mathrm{PI} \Rightarrow F(\mathbf{x}) = 0$. For Boolean formulas without complemented variables ($'.$), the following special terms are used:

- Assignments consisting only of 1-assignments for which the expression simplifies to 1 are called *cut sets*, and *Minimal Cut Sets (MCSs)* when they are not included in any other of lower order.
- Assignments consisting only of 0-assignments for which the expression simplifies to 0 are called *path sets*, and *minimal path sets* when they are not included in any other of lower order.

The notation of implicants can happen in two ways: One is by giving a Boolean formula that has the given implicant as a prime implicant, e.g. $a \cdot b$ or ab, for stating that both a and b need to be 1 for some function to return 1, or ab' for stating that a needs to be 1 and b 0[4]. Another is by introducing a third value d (as in 'don't care') specifying that the variable with the same index is not part of an implicant and giving its corresponding explicit partial assignment. In this second way, 0, 1 or d needs to be given for each variable, e.g. for a Boolean function with three variables, the prime implicant ab' could also be identified by its corresponding partial assignment $10d$. In this work, we use the first notation unless specified otherwise.

Any function can be written as a disjunction of prime implicants. When each one is given exactly once, this form is called the Blake Canonical Form (BCF). Under the assumption that the variables of the function can be ordered, and that this order is extended to all possible combinations of variables, the variables in each prime implicant and the prime implicants in the disjunction can be ordered accordingly, which yields

[4] An implicant assigning all variables in this way in this notation is called a *minterm*, but that definition is not used in this work.

a canonic expression. Even though there may be shorter forms of a function, the BCF has the advantage that evaluating an assignment can be done by testing inclusion with each prime implicant, and that the ordering of variables and prime implicants makes finding relevant implicants fast. Another application is in forming partitions of the input domain, which are particularly useful in probability theory.

2.1.3 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is a finite set of variables and a finite set of *constraints* defined as equalities and inequalities. When each variable is assigned a value satisfying each constraint, it is called a *solution* for the CSP. In this work, their generality is employed as a means for linking models of physical system behavior and their logical abstraction in fault trees. The basics of defining and solving CSPs are discussed in [15]. A concept that is pivotal in this context is constraint reification: A given constraint needs to be fulfilled by a solution only when a specific Boolean variable is set to 1, and is ignored otherwise.

2.2 Concepts of model-based system engineering

This section outlines the dependencies between system design, verification and the core topic of this thesis, model-based safety assessment. It also introduces the concepts and notation of hybrid systems and describes the inherent difficulties of their formal verification.

2.2.1 Terms and definitions

The purpose of engineering, in general, is creating entities (e.g. machines, software or microorganisms) that exhibit a *desired* behavior, by the composition of entities that, each on its own and in interaction, exhibit their *natural* behavior. It is the carefully engineered composition that gives the created entity usefulness. The composition process is called *system design*. The composite entities are referred to as *components*, the created entity as *system*. The abstract description of desired and component behavior is called a *behavioral model*. Other kinds of models are not taken into account in this work, such as technical drawings or models that group information about components without describing their behavior.

The level of detail of the model needs to be sufficiently fine to capture all relevant aspects, but no finer: Engineering effort increases not just with desired behavior being less similar to natural behavior (e.g. automobile vs. aircraft), but also with the (temporal or spatial) scales of detail required to interact harmoniously to produce desired system behavior (e.g. water mill vs. fusion reactor or mining using pick-axes vs. explosives). *Hybrid systems* are models that allow introducing a varying level of detail over time within a component's behavior description, depending on and affecting its interaction with other components.

Producing proof that composite component behavior produces desired system behavior is called *verification*. Deviations from desired system behavior are referred to as *failures* and can have various causes: Engineering errors, environmental influence or changes in component behavior. Deviations in component behavior are called *faults* and can cause failures. When failures can harm humans, verification before the introduction of the system into service ensures their safety and, in aerospace engineering, is called *safety assessment*.

2.2.2 Safety assessment and hybrid systems

Traditionally, safety assessment is carried out primarily by applying engineering judgment: Qualitative justification for the system to be sufficiently safe is derived from the engineer's experience. This is due to the organizational interaction between design and verification: The former traditionally builds models for nominal system behavior, e.g. in the absence of component faults. The latter cannot ignore the influence of plausible component faults. Extending models with failure behavior without separating the concerns of design and verification is an undue increase of model complexity [16]. With no solution to this issue, model-based safety assessment is not helpful: The additional time spent on model extension and by designers on continuing their activities on the more complex, extended model negate the benefit of abstract and automatable identification of the possible causes of failures. A safety engineer in practice simply spends more time with manual safety assessment, likewise increasing the quality of safety assessment results. A second problem hinders model-based safety assessment: The level of detail required to capture all significant behavioral traits of component faults can be higher than that for nominal behavior. This requires models with a varying level of detail among components, so that the component under fault can be described in adequate detail without exchanging the behavioral descriptions of all other components. Both of these two issues are addressed by hybrid systems. For that reason, they are the centerpiece for the realization of model-based safety assessment.

2.2.3 Hybrid systems in model-based engineering

Hybrid systems achieve model flexibility by separating behavior into time intervals of continuous state evolution, joined by discrete reconfiguration steps that can make the state jump to a new value and alter continuous evolution afterward. They differ mainly by the allowed space of functions for defining continuous behavior and the rules of discrete reconfiguration.

This section will introduce a notation to explain the fundamental concepts of hybrid systems. For the sake of clear comparability with related works, we use a notation similar to the one of Alur et al. [17].

It builds upon the definition of hybrid *traces*. A hybrid trace τ is a set of constituent functions τ_i defined over a (finite or infinite) partition of time defined as a sequence of intervals $I_0 I_1 I_2 \ldots$, where I_i is the input domain of τ_i. By convention, they are naturally ordered.

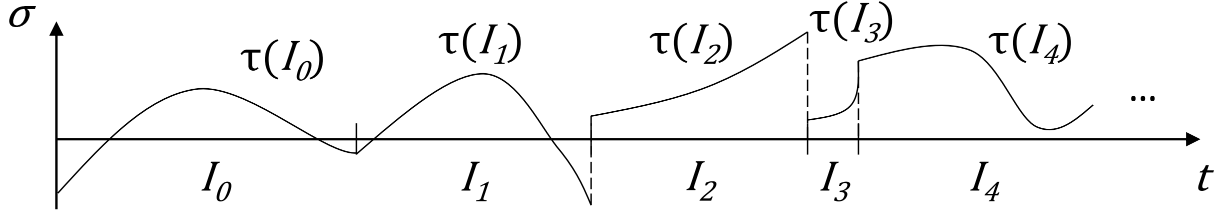

Figure 2.1: Example for a piecewise-smooth hybrid trace

When comparing hybrid systems to initial value problems, the function space of all possible traces is the solution space of the initial value problem.

Hybrid *runs* ϕ are traces conforming to a set of rules which characterize the system under study. In the terms of initial value problems, runs are the equivalent of solutions. The declaration of these rules can be denoted as a quintuple $(\Sigma_D, Q, \mu_1, \mu_2, \mu_3)$, again following Alur et al. [17].

1. The output of constituent functions $\tau_i(t)$ is a vector of continuous variables called a *data state*. Σ_D denotes the domain of data states, e.g. the co-domain of the constituent functions, and is invariant over the intervals of time I_i.

2. Q is a finite set of *locations*. The variable $\ell_i \in Q$ holds the *active* location[5] during a given interval I_i. During an interval, the active location does not change. See the next points for a definition of 'active'.

3. μ_1 assigns each ℓ a set of *activities*. An activity is a C^∞-function, defines continuous system behavior at the given location and does not depend on global time [18].

4. μ_2 assigns each ℓ a set of *exceptions*. The active location must change before the continuous system state $\tau_i(t)$ reaches any exception.[6] The complement $\Sigma_D \backslash \mu_2(\ell)$ is called the *invariant* of (the system at) ℓ.

5. μ_3 assigns locations and states a *successor* tuple. A state $\sigma' \in \Sigma_D$ at location $\ell' \in Q$ is the successor of a state $\sigma \in \Sigma_D$ at location $\ell \in Q$ if and only if $(\ell', \sigma') = \mu_3(\ell, \sigma)$.

In the terms of initial value problems, $(\Sigma_D, Q, \mu_1, \mu_2, \mu_3)$ takes the place of the differential equation (with the set of its variables).

Initial conditions can be expressed by specifying system state and location at the beginning of the first interval $\sigma_0 \in \Sigma_D$ and $\ell_0 \in Q$, or as a predicate μ_0 on the continuous

[5] Note that traces do not contain the evolution of the active location over time, only that of the continuous states.

[6] μ_2 allows imposition of additional limitations on Σ_D. In safety assessment, exceptions would be states in which analysis is irrelevant, e.g. corresponding to different failure conditions than the one under analysis. Defining them is optional, because they can already be taken into account when defining Σ_D. But separating exclusions for different reasons into different semantic parts of the model definition can encourage model reuse.

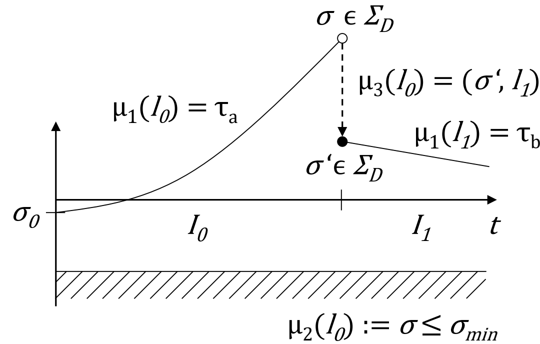

Figure 2.2: The quintuple of a hybrid system producing a run: The first trace starts at σ_0, and the activity selector μ_1 selects the activity for the first location ℓ_0 that produces the trace piece τ_a. After some evolution, the successor selector μ_3 enforces a transition to σ', where the activity selector μ_1 selects an activity that produces τ_b. The exception defined by the exception selector μ_2 is identical for both locations and never triggered. All constituent functions τ range over the state space Σ_D.

state and discrete location at $t = 0$. While the former is useful for *simulation*, where exactly one run is to be identified by specifying the hybrid system and an initial state, the latter is more useful for *verification*, where all possible runs that pass a given acceptance condition are determined.

For verification purposes, runs also conforming to μ_0 are compared to *acceptance conditions* μ_4. Acceptance conditions are predicates on runs in an appropriate temporal logic.

Table 2.1 summarizes the notation of hybrid systems used in later sections of this work.

Simulating a hybrid system is equivalent to finding a run $\phi \in \mathcal{S}$ that conforms to a given initial condition μ_0.

Verifying a hybrid system is equivalent to determining whether any runs exist that fulfill a given acceptance condition. In safety assessment, the acceptance condition would be the failure condition[7].

2.2.4 Formal verification

Formal verification is the process of determining whether a mathematical model (of a system) has a given property, called the *requirement*. It can be done by a number of methods, of which not all are relevant for this work:

1. formally documented engineering judgment — not relevant because, by definition, it cannot be automated

2. testing — not relevant because it cannot prove the absence of errors when the system has an infinite number of states (e.g. due to a real-valued system state)

[7] One may speculate that the creators of the term 'failure condition' were aware of this coincidence in terminology between automata theory and safety assessment.

Table 2.1: Overview on the notation of hybrid systems

Term	Type	Symbol	Definition
Time	Scalar value	t	$\in I$
Interval indices	Set of natural numbers	T	$\subseteq \mathbb{N}_0$
Interval of time	Set of real numbers	I_i	$\subseteq R^+, i \in T$, simply connected, $\bigcup_i^T I_i = R^+$, $\bigcup_{i,j,i \neq j}^{T \times T} I_i \cap I_j = \emptyset$
Domain of continuous state	Vector of sets of real numbers	Σ_D	$\subseteq \mathbb{R}^{n \in \mathbb{N}}$
Constituent function/-s of a trace	Vector of functions	$\tau_i(t)$	$: I_i \to \Sigma_D$, $\tau_i \in (C^\infty)^n$
Trace (evolution of continuous state)	Set of vectors of functions	τ	$= \bigcup_i^T \tau_i$
Domain of traces	Set of sets of functions	$\mathcal{S}$	$= \Sigma_D \times \mathbb{R}^+$, piece-wise smooth
Domain of discrete locations	Finite set of natural numbers	Q	$\subset \mathbb{N}$
Path (evolution of discrete location)	Discrete function	ℓ_i	$: T \to Q$
Activities assignment	Function	μ_1	$: Q \to (C^\infty)^{m \in \mathbb{N}, m \geq n}$
Exceptions assignment	Predicate (on location, state)	μ_2	$: Q, \Sigma_D \to \{0,1\}$
Transition relations	Vector function (on location, state)	μ_3	$: Q, \Sigma_D \to Q, \Sigma_D$
Run	Set of vectors of functions	ϕ	$= \tau$, $\mu_2(\ell_i, \tau_i(t)) = 1 \, \forall \, i \in T, t \in I_i$ $\tau_i = \mu_1(\ell_i) \, \forall \, i \in T$, $(\ell_j, \tau_j(t)) = \mu_3(\ell_i, \tau_i(t)) \, \forall \, i, j \in T, j = i+1$, $t = \max(I_i) = \min(I_j)$
Initial condition (explicit form)	Tuple	$\ell_0, \tau_0(0)$	$\in Q, \in \Sigma_D$
Initial condition (predicate form)	Predicate	μ_0	$: Q, \Sigma_D \to \{0,1\}$
Acceptance condition[1]	Predicate	μ_4	$: \mathcal{S} \to \{0,1\}$

3. theorem proving — not relevant because of its strong dependency of the employed techniques on properties of the mathematical model under analysis

4. model checking

In model checking, the property to be assessed is given in the form of a set of states, a sequence of states, or combinations of both. Furthermore, an initial condition to the system state is given. The states the system can reach and/or their sequence is compared to the property. Determining the reachable states and their sequence is carried out iteratively: Initial states are reached by definition, and the model (e.g. the specification of the system under analysis) is used to determine reachable states over time.

For systems with a finite, tractable number of states that reach a steady state[8] or infinite loops of states, the set of reachable states can be determined through *exhaustive state space exploration*: At every iteration, immediately reachable states are added to the set of all reachable states until no new states can be reached. Without loss of generality, computational effort and problem size can be balanced by imposing the Markov-property on the model. However, applying this concept to physical systems is difficult: In order to model a finite, tractable number of states, abstraction from real-valued physical variables is required. For aerospace systems, this abstraction can be as difficult as manual safety assessment. Additionally, abstractions often are specific for a certain failure.

Nevertheless, if a valid abstraction is found and formal verification is carried out, its results do not suffer from analysis defects that may be introduced in purely manual engineering judgment. A number of tools exist that specialize in the application of model checking for safety assessment:

1. The Safety Analysis and Modeling Language (SAML) [19] is an extension of the PRISM language [20] and also the name of a supporting toolchain. Depending on the type of properties to verify, the model is translated into input languages for an appropriate model checker. This makes SAML the most versatile base for model checking in the context of safety assessment.

2. AltaRica [7] is a modeling language that has been commercially distributed and scientifically studied in great detail, e.g. in [21, 22]. It also inspired a number of methods that are more powerful (see next paragraph) or efficient in practical application ([23, 24]) or that target design optimization rather than safety assessment ([25]).

Alur [6] has developed techniques from symbolic model checking for application in hybrid systems verification as an alternative to model abstraction. Instead of storing reachable states as explicit sets of states and modifying them as the search for further reachable states is conducted, *symbolic reachability analysis* stores symbolic expressions that could be used to generate the explicit set of reachable states, even if its cardinality

[8]This work focuses on safety requirements, which typically (but not generally) express that the system shall not enter a set of states.

is infinite, in the form of predicates[9]. After starting with an initial state (or multiple allowed initial states) specified as a predicate, model behavior is applied to the predicate iteratively, until the state space stops expanding or a decision is made on whether the model suffices the property or not. This method itself imposes requirements on models and properties under analysis:

1. Model behavior needs to be specified as operations on predicates over states

2. Properties need to be specified as predicates over predicates over states

For efficient computation, simple predicates with linear boundaries are chosen for both the initial state and the property. By over- or under-approximation of the exact predicates, proving a property to be satisfied or not can be carried out more efficiently. Such approaches have been pursued e.g. by Althoff et al. [11] and Girard et al. [26].

2.2.5 Hybrid systems verification problem

In publications that discuss the decision problem of hybrid system verification from a logical perspective, the term *decidability* is used in place of tractability, with the same meaning: When there is no method for determining whether the state space will intersect the property's subspace or when all methods are at least of exponential complexity, the combination of model and property is called *undecidable*. For the sake of comparability of problems, operators in the property other than equality and inequality comparison of continuous state variables are, by convention, encoded in the model. Any expression in a predicate that can be expressed as a smooth function can be replaced by an additional, 'dependent' continuous variable. The expression is added to the model as a location-invariant activity, i.e. it stays the same at all locations. Predicates that do not present the required smoothness are out of the scope of this work.

Hybrid systems by definition impose smoothness on activities, i.e. $\mu_1(\ell) \in C^\infty \,\forall\, \ell \in Q$. Transitions are not limited besides being deterministic.[27] have proven that this makes verification generally undecidable, and given a few simple models for which it is decidable.[11] has shown how abstraction of special cases of more complex models into decidable models is possible. In summary, the intuitive assumption holds that the limitations on decidability, e.g. as for stability or boundedness proofs in control system theory, are not lifted by additionally allowing discrete changes to otherwise smooth system state evolution.

2.3 Safety assessment

Collecting evidence that an aircraft or system complies with functional safety requirements of its governing certification specification is structured by a process framework

[9] The difference between symbolic reachability analysis and theorem proving is that theorem proving reasons about traces (sequences of states), whereas symbolic reachability analysis does not consider the information of the sequence of states — only that they were visited at all.

titled *safety assessment*[10], which is specified in [28]. This section introduces the legal obligations and relevant standards for safety assessment, as well as the mathematical concepts of its core analysis technique.

2.3.1 Legal framework and standards

Federal law in Germany mandates that every manned aircraft, commercial pilot, airfield, air navigation services provider etc. is subject to certification (unless an exception applies, e.g. for military manned aircraft) [29, 30]. Certification is relayed to the European level and, there, allocated to the European Aviation Safety Agency (EASA) [31]. This is similar to the practice in the United States, where the Federal Aviation Administration (FAA) fulfills the same function under a similar mandate.

The annexes of [31] and related European Commission regulations define a binding framework for technical requirements. Detailed technical requirements that need to be fulfilled to achieve certification are set forth — in concord with that precept — in non-binding agency rules. 'Non-binding' in this context means that these rules are binding to the certification applicant, but not to the certification authority: They may be altered by the certification authority for a specific certification application as a purely executive act. This way novel technologies and technical solutions unforeseen by the technical certification requirements are supported on a case-by-case basis, without a need for intervention by the legislative power. However, any technology or technical solution must be proven to provide at a safety level at least equivalent to that provided by conventional certification.

For aircraft and their onboard equipment, technical requirements are expressed in certification specification documents. For normal, utility, aerobatic, commuter and large aircraft, § 1309 in [32] [11] and [33] addresses functional safety, i.e. the aircraft or system responding to plausible environmental conditions, input and hardware failure in a way that poses no unacceptable risk to aircrew, occupants and third parties in other aircraft or on the ground. Regulation under FAA-sovereignty has the same structure, and — for § 1309 — identical technical content.

Proving compliance with these technical requirements is structured by AMCs. They are specified by the certification authorities and, in turn, refer to industry standards for best practice and defining the state of the art. Differing approaches to proving compliance with certification specification are theoretically acceptable, but substantiation that such an individual solution is at least of equivalent significance as agency-defined AMCs is required to obtain certification credit.

Specifically for very elaborate AMCs such as [34] for the aforementioned functional safety paragraph 1309, the effort required for such a comparison in significance for

[10] This definition is weaker than the one given in Section 2.2.1. Engineering judgment cannot provide proof but only evidence. On the other hand, it may have greater significance because of not being limited to the model's domain and detail.

[11] During the final phase of writing, citeCS23 has received a major update and paragraph numbers have been reissued. In the new numbering schema, the core content of the former § 1309 is now expressed in § 2510. Because US Federal Aviation Regulations (FAR) and CS25-requirements still use the "old" numbering schema, the text of this work is left unchanged. Acceptable Means of Compliance (AMCs) are unchanged.

largely different approaches would be prohibitively high for single certification applicants. This forces methodological and process-related advances in such areas to either progress 'organically', or rely on effort pooling with competitors in standardization bodies, such as Radio-Technical Commission for Aeronautics (RTCA) or Society of Automotive Engineers (SAE).

The aforementioned AMC [34] by the FAA[12] is of particular relevance to this work: It addresses proving compliance with [32], § 1309 and refers to [35] for design and [28] for safety assessment. Therefore, model-based means of automation of safety assessment need to adhere to their precepts on process and method in order to be eligible for transfer into industrial practice in the short and medium term.

2.3.2 Safety assessment process

The safety assessment process is organized by the reports that need to be handed in for certification. Its first deliverable is the collection of the feared top-level events, which are called *Failure Conditions (FCs)*, classification of their severity and derivation of high-level safety requirements in the *Functional Hazard Analysis (FHA)*. In the following deliverable, called *Preliminary Aircraft Safety Assessment (PASA)* on aircraft-level or *Preliminary System Safety Assessment (PSSA)* on system level, a search for root causes for these events is conducted, usually in the form of one *Fault Tree Analysis (FTA)* per FC. Once aircraft or system design is frozen, the qualitative assumptions made during FTA are validated and their sufficiency to prove compliance to higher-level safety requirements is reviewed in the *Aircraft Safety Assessment (ASA)* or *System Safety Assessment (SSA)*. Additionally, the independence of identified root causes is verified during *Common Cause Analysis (CCA)*, which greatly simplifies probability computations, strengthens the significance of FTA results and covers additional top-level events, or such events that are intrinsically difficult to handle with the hierarchical, deductive approach of FHA, PSSA, and SSA. The assessment of independence of root causes happens in the Common Modes Analysis (CMA) and covers physical and functional separation.

During fault tree generation and before design freeze, the probabilities or failure rates of basic events are estimated pessimistically on the basis of experience and available data on similar aircraft or systems. This is called probability or failure rate *budgeting*. Once specific components have been selected, their budget is replaced with substantiated evidence specific to the employed component designs. They are retrieved from reliability handbooks, in-service experience and reliability tests in *Failure Modes and Effects Analysis (FMEA)*.

A detailed description of FTA is given in [36, 37]. Two steps of the method are defined: Generation and solution. During fault tree generation, the search for root causes for the top event is conducted in a deductive manner by system and safety engineers. The key benefit of FTA is that the deductive argument leading to the root causes is captured along with them and presented in a graphical notation, enabling validation of the conducting experts' reasoning. During fault tree solution, Minimal Cut Sets (MCSs), top-event probability and other safety-relevant metrics are calculated

[12]Note that some AMCs are accepted by both EASA and FAA.

from the generated fault tree and the probabilities of the identified root causes (see also Section 2.3.3).

With a skilled fault tree generation team, the resulting arguments take a divide-and-conquer form that relies on *functional decomposition* of the top-level behavior. Decomposition principles require independence of its composites, e.g. each component function contributing on one and only one level in the decomposition. This is seldom true for root causes in complex aircraft systems for two reasons: First, resource functions commonly supply many levels of functional decomposition, e.g. electric energy being supplied to both a flight control computer and its connected sensors or electric actuators drives complexity of failure behavior prediction in case of faults in the electric energy supply. Secondly, functional integration creates implicit contributors, e.g. when a display is used both for displaying aircraft status and warnings, erroneous presentations of status may mask warnings about other, more critical failures, leading to inadequate flight crew actions for their mitigation.

Both resource sharing (e.g. in integrated modular avionics or all-electric aircraft), as well as functional integration (e.g. thrust vectoring integrating aircraft stabilization with propulsion control, or autopilot integration into flight control systems), have a potential to significantly increase efficiency in existing aircraft concepts or enable entirely new ones. Examples for the latter are small Unmanned Aerial Vehicles (UAVs), where both concepts are key to reaching adequate payload or range for some applications. By contradicting strict functional decomposition, fault tree generation and thereby safety assessment feels the same increase in complexity as system design. This is not a drawback of FTA specifically. All deductive reasoning cannot circumvent its subject's complexity, but only its size, through a divide-and-conquer approach.

The intrinsic difficulty of fault tree generation of complex systems and the growing urgency of the practical problem motivate this work to curb the growth in effort of safety assessment.

2.3.3 Fault tree solution theory

Once a fault tree has been generated, it can measure susceptibility to its top event both qualitatively and quantitatively. Methods for this purpose are described in [28], but they assume that the fault tree has been converted into a Boolean formula and its prime implicants have been determined. Obtaining the prime implicants of the Boolean function encoded in a given fault tree is called *fault tree solution*.

Before tools for Boolean expression manipulation can be employed, a fault tree needs to be converted into an equivalent Boolean expression F. Its underlying function returns 1 when the top event[13] occurs and 0 otherwise. It depends on the occurrence of the basic events of the fault tree. This gives functions encoding fault trees the signature $F \coloneqq \mathbb{B}^n \to \mathbb{B}$. Each basic event's occurrence is encoded as the value taken by a Boolean variable. This defines the input vector $\mathbf{x}$. The logic implemented by the gates

[13] According to [37], the root node is displayed at the top of a fault tree, and its child nodes recursively below it. This is different in some German standards, which display fault trees left-to-right (instead of top-to-bottom). This work follows the internationally common top-to-bottom orientation in illustrations and verbal descriptions.

Table 2.2: Fault tree gates and their equivalent Boolean expressions

Fault Tree Gate	Equivalent Boolean Expression
AND-Gate	$x_1 \cdot x_2$
Priority AND-Gate	$\mathrm{ROF} \cdot x_1 \cdot x_2$
Inhibit	$\mathrm{ROF} \cdot x_1$
OR-Gate	$x_1 + x_2$
Exclusive OR-Gate	$(x_1 + x_2) \cdot (x_1' + x_2')$
Combination Gate	With 2-out-of-3: $x_1x_2 + x_2x_3 + x_3x_1$

of the fault tree is translated into the operators $+$, $\cdot$ and $'$. See Table 2.2 for how specific gates are translated into formulas. Gates requiring events to happen in a specific order are handled with an ROF event (as in 'required order factor'). They occur only when the related basic events occur in the specified order. Their associated probability is the fraction of the number of permutations of event order allowed over the total number of permutations possible.

Fault trees created in conformance with [28] code their logic through gates that can be expressed with the Boolean operators $+$ and $\cdot$, with one exception: The XOR-gate[14]. It relies on the complement operator $'$ for excluding the intersection/-s of its subordinate events. A fault tree containing one or more XOR-gates is called non-coherent[15]. Its corresponding Boolean function is non-monotonic. It can be approximated by an OR-gate, which introduces only a second order error on probability computations when operand probabilities are low. Fault tree solution of non-coherent fault trees requires substantially greater effort as it cannot rely on monotonicity of the underlying Boolean function. An implicant of a coherent fault tree is called a *cut set* in [28], and the equivalent of a prime implicant is called a *Minimal Cut Set (MCS)* (see Section 2.1.2).

Once a fault tree has been translated into a Boolean formula, it can be brought into BCF, e.g. its prime implicants can be retrieved, using methods based on the following principles (see Brown [14], Section 2.1.2):

1. *Exhaustion of implicants*: Testing all possible assignments and identifying prime implicants through absorption
2. *Iterated consensus*: Repeated application of the consensus theorem and subsequent absorption, which requires the formula to be in SOP form
3. *Iterated expansion*: Repeated application of Boole's expansion theorem

As the number of possible assignments grows exponentially over the number of variables, exhaustion of implicants has prohibitive computational complexity for solving large fault trees. Methods based on iterated consensus are only slightly more efficient: Bringing a Boolean formula into Sum of Products (SOP) form is computationally less

[14] Exclusive OR-Gates are not mentioned in [28], but in the documents referred to for detailed descriptions of the FTA method [36, 37].

[15] The origin of this term becomes apparent when examining the Venn diagram of the XOR-operator on sets.

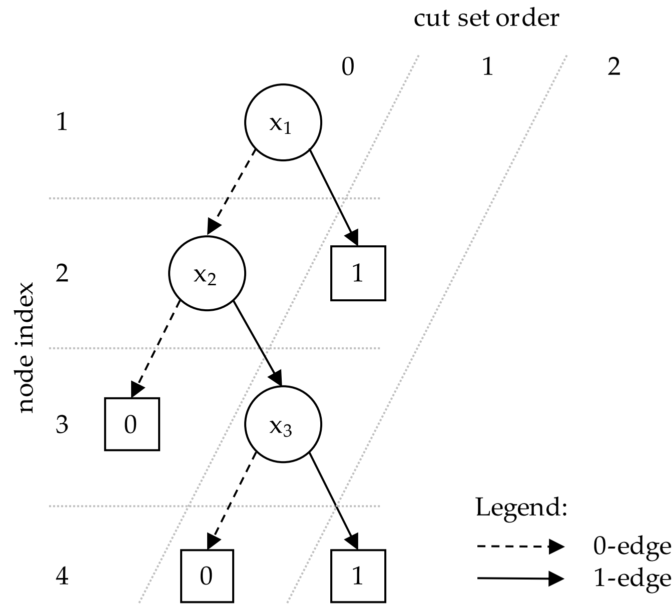

Figure 2.3: BDD example, encoding $x_1 + x_2 \cdot x_3$

costly than exhaustion of implicants, but once SOP form is reached, the actual method only begins with reaching Blake Canonical Form (BCF) from SOP form. The third class of methods does not require such conditioning and is employed in today's most efficient fault tree solvers ([38, 39, 40]). It yields a disjoint partition of the solution space in SOP form, which is particularly efficient in probability calculations.

An instructive visualization of Boole's expansion theorem and the inspiration that has led to highly-efficient data structures today's fault tree solvers are based on are *Binary Decision Diagrams (BDDs)*. They are tree graphs, consisting of *vertices*, also called *nodes*, and acyclic, directed *edges*. A node can be either a *variable node*, commonly denoted by a circle, or *terminal node*, commonly denoted by a square. Exactly one variable node has no inbound edge. It is called the root node. Other variable nodes have one inbound and two outbound edges each. Terminal nodes have one inbound and no outbound edges.

A BDD can be interpreted as a visual form of iterative application of Boole's expansion theorem to a Boolean formula. This way, the characteristic function of a fault tree can be encoded in the computationally efficient data structure of a BDD. Each node represents the expansion of the Boolean function resulting from its inbound edge for the variable denoted in it. As an example, consider Figure 2.3. Applying Boolean expansion on x_1 on the characteristic function $F = x_1 + x_2 \cdot x_3$ of a fault tree yields $F = x_1 \cdot (F(x_1 = 1, x_2, x_3)) + x_1' \cdot (x_1 = 0, x_2, x_3) = x_1 \cdot 1 + x_1' \cdot (x_2 \cdot x_3)$. The terms in the brackets are the sub-trees behind the edges of the root node that correspond to the preceding variable's value. Thus, the solid 1-*edge*, which corresponds to x_1, leads to 1, and the dotted 0-*edge*, which corresponds to x_1', leads a the BDD obtained from further applying Boolean expansion to $x_2 \cdot x_3$. The transformation from a Boolean function F to a BDD is denoted $\mathrm{bdd}(F)$. BDDs also have the benefit of being *canonical* for a given order of variables during expansion: When variables are expanded in that same order,

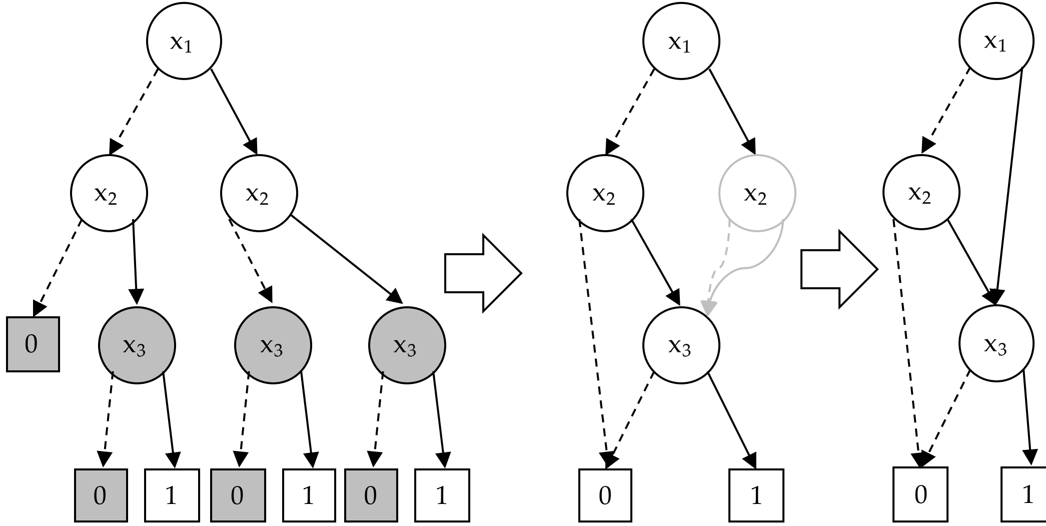

Figure 2.4: BDD reduction: Identical subtree merging and obsolete node removal of the BDD encoding $x_1 \cdot x_3 + x_2 \cdot x_3$

the resulting BDD of every Boolean function with the same truth table will be the same.

For more complex BDDs, the benefits of BDD-based data structures become apparent. When two sub-trees of a BDD are identical, e.g. two intermediate functions obtained through expanding for different values are identical, the subtree only needs to be created once. All edges to any identical subtree can point to the one such subtree. This operation is called identical subtree merging. The second operation that reduces both the size of a BDD and the effort required for their construction is obsolete node removal: When the outcome of expanding a specific node is identical for both edges, the node can be skipped by the edge leading to it. Both operations can also be applied to the leaves of the BDD, to further conserve storage space. They are illustrated in Figure 2.4. This particular example also illustrates how variable ordering is crucial to BDD size: The ordering (x_3, x_1, x_2) would have produced a smaller BDD. Unfortunately, finding the optimal variable ordering is of exponential computational complexity and thus only allows to tweak algorithm computational effort versus storage space. Note that reduced BDDs are no longer trees, because no node of a tree may have more than one parent node, but the benefits of lower memory consumption and intermediate result re-usage greatly outweigh the greater costs for some elementary operations on general undirected graphs.

All set operations can be applied to BDDs as they are applied to Boolean functions. Special in the context of this work is the *relative complement*, denoted $\setminus$ or *nand*. For two BDDs $\mathrm{bdd}(F_1)$ and $\mathrm{bdd}(F_2)$, the relative complement is defined as:

$$\mathrm{bdd}(F_1) \setminus \mathrm{bdd}(F_2) := \mathrm{bdd}(F_1 \cdot F_2') \tag{2.16}$$

This operation is can be used to test for Boolean inclusion: If $F_1 \leq F_2$, the result is a BDD consisting only of the 0-leaf. It is special because Boolean inclusion can be argued to test for cause-consequence relationships.

Reasoning on the logic displayed in a BDD can be formalized in *paths* between nodes. Such paths are sets containing nodes and edges in between them. Because no

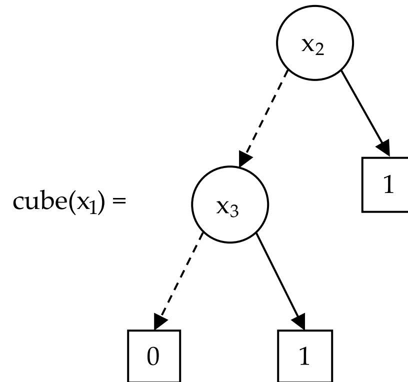

Figure 2.5: BDD path cube: In a BDD with the variables (x_1, x_2, x_3) (in that order), $\mathrm{cube}(x_1) = x_2 + x_3)$. Also, $\mathrm{cube}(x_1) = \mathrm{cube}(x'_1)$.

pair of nodes is directly connected by more than one edge after obsolete node removal, paths can be specified by giving a complete list of traversed edges only. E.g. the 0-node of Figure 2.4 is connected to the root node via $x'_1x'_2$ and by $x_1x'_3$. This notation correctly suggests that such paths can be interpreted as Boolean formulas or BDDs, which yields a closed theory on Boolean formula manipulation solely expressed in BDDs and operations on them. Nodes are partially ordered by the number of inbound edges that need to be traversed to reach the root node, called the *node index*. The number of 1-edges on the path from a 1-leaf to the root node is called its *(cut set) order* (see Figure 2.3).

The *cube* of a path is a BDD where all variables not present in the path (i.e. all variables with higher node index) are ORed. This is illustrated in Figure 2.5. The cube of a path that reaches the highest node index is 0, because there are no variables left below it. Additionally, the cube of the empty path is all variables ORed.

Once a BDD for a given Boolean formula has been constructed, a disjoint partition of the subspace of its input space that maps to 1 can be directly read from the BDD: It is specified by the set of paths from the root to the 1-leaf. E.g. in Figure 2.4, these paths are $x'_1x_2x_3$ and x_1x_3. With a disjoint partition, the fault tree's top event probability can be obtained without the inclusion-exclusion principle, when each path's edges are replaced by the probabilities of the related basic events of the fault tree for 1-edges, or their complement for 0-edges, respectively.

Prime implicants can also be obtained by examining paths from the root to 1-leaves. In contrast to the subspaces for disjoint partition, not all edges covered by the path are necessary. This can be illustrated with Figure 2.3: For the path $x'_1x_2x_3$ the edge x'_1 can be omitted, because x_1 is an implicant as well. This problem can be addressed by incremental computation of prime implicants from 1-leaves upwards towards the root node. During this traverse, implicants of both sub-trees of a node are not expanded; Only such implicants occurring in only one subtree are expanded by adding either the variable (for the 1-edge) or its complement (for the 0-edge) to list of implicants of currently handled tree.

The incremental pattern for finding prime implicants ensures that the retrieved implicants are prime, but cannot ensure that all of them are *essential*, e.g. that they are not covered by disjunctions of other prime implicants. Even though obtaining the set

of essential prime implicants would make further processing even easier, as it exploits symmetries in the underlying Boolean formula, their complete calculation is of exponential complexity. Because of this, their practical use is limited, despite their potential for simplifying further analysis. For coherent fault trees, prime implicants (e.g. MCS) are always essential, which makes the differentiation irrelevant for safety assessment as per [28].

Chapter 3

Operational Semantics for Hybrid Failure Behavior

Hybrid systems, as introduced in Section 2.2.3, provide a means for abstracting physical behavior at smaller temporal (or spatial) scales than what the model generally covers. This is useful in both nominal and failure behavior: All behavioral models have their 'normal' timescale, at which most of the effects play out. The effects that happen over longer timescales are usually modeled by setting specific initial conditions that stay constant over course of the simulation, or by model variants for different such circumstances. Faster timescales could theoretically be modeled by increasing the overall model granularity so that all effects are modeled precisely enough for the fastest timescale effect, but the required modeling and simulation effort only pays off when and where the fastest timescale effect happens. In nominal behavior, this may not be the case during the vast majority of simulation time and most components of the model. As an example, consider physical contact or collision in mechanical simulation, which only requires high temporal resolution and modeling granularity during brief moments, while most of the time, the simulated bodies move freely at slower timescales and can be described by much simpler physical laws than during contact. Hybrid simulation allows the separation of sub-models of different modeling granularity (and thus simulation timescales) over simulation time, triggers granularity changes through rules on the simulation's state and ensures sub-model compatibility.

In safety assessment, where failure behavior is the subject of analysis, this is of great benefit. The *process of failing* often happens at very fast temporal scale compared to nominal behavior, such as the contact closure when short-circuiting electric circuits, or pipe rupture in hydraulic systems. When the failure will happen may not be known a priori but depend on the simulation's state, and the process of failing often takes very little of the simulation's total time. Even without timescale variation, hybrid simulation is useful in failure simulation: Failure behavior, i.e. *being failed*, often involves different physical effects or completely replaces those of nominal behavior, e.g. when a stuck clutch ceases its dampening mechanical behavior regardless of the applied contact pressure and becomes a stiff connection instead. Both aspects can be described elegantly in hybrid systems. Additionally, they allow separating nominal and failure behavior in an aspect-oriented way that contributes to model maintainability.

The proposed operational semantics for hybrid failure behavior is the first such semantics that is ready for practical application. This is supported by the fulfillment of the requirements for such application that have been derived by Joshi and Heimdahl [1].

3.1 Formal definition of a hybrid system for failure simulation

This section details the definition for hybrid systems given in Section 2.2.3 and provides an interpretation for its specifics that naturally applies to behavioral system modeling for aerospace systems under consideration of both nominal and failure behavior.

3.1.1 Continuous activity specification

Hybrid systems as defined by Alur et al. [41, 18] require that activities $\mu_1(i)$ are smooth functions. We relax this requirement to be compatible with MATLAB's [5] behavioral modeling environments Simulink and SimScape: There, traces are computed numerically discretized in line with IEEE 754 [42] and possible lack of smoothness is compensated by a variable time step solver. SimScape allows arbitrary specification of both explicit and implicit algebraic and ordinary differential equations (with time derivatives) to define system state evolution for the continuous portion of the hybrid system. This work does not require verifiability of hybrid models of the presented operational semantics, so the modeler can choose freely among the features for continuous behavior specification.

There is a fundamental difference between Simulink and SimScape: While Simulink models specify *directional* signal propagation and transformation, SimScape models specify state variables and their interrelation through (omnidirectional) Differential-Algebraic Equation Systems (DAEs). Thus SimScape models do not limit the direction of flow of energy, mass or information. This is essential for describing failure behavior in the same model as nominal behavior: Because failure behavior does not necessarily obey the propagation direction of nominal behavior in physical systems, direction-neutral models save modeling effort and prevent subtle modeling flaws that inhibit comprehensive failure modeling in directional models (see also Joshi and Heimdahl [1]).

3.1.2 Structural aspects

By extending Simulink and SimScape, three structural aspects of their modeling languages present themselves for employment in this method.

- *Component-orientedness*: All elements of the hybrid system are organized in components. Continuous state variables are the energy, mass or information storages of each physical component of the system (at the desired level of detail). This is sufficient for comprehensive physical behavior specification as per Karnopp et

al. [43]. Additionally, auxiliary variables can be introduced which depend on the continuous state variables and are carried along for convenience in initial state, exception, transition or acceptance condition definition.

- *Hierarchicality*: Components can be grouped into *subsystems*. Subsystems can be grouped into subsystems as well, which allows for multiple layers of hierarchy and a divide-and-conquer-philosophy in system structure and behavior specification.[1]

- *Aspect-orientedness*: Nominal and failure behavior specification of each component are mutually exclusive: Only one behavior specification for each component can be active at any time (see also Subsection 3.1.3). All of them are defined in a similar way, so users understanding one domain of component behavior can explore and contribute to behavior definition of the other.

- *Object-orientedness*: Components can be instantiated from libraries of components into models or new libraries, allowing parametrization but forbidding fundamental behavioral changes. Components can also be derived from ancestor components, allowing parametrization and extension of behavior, e.g. with failure behavior or with more detailed nominal behavior.[2]

From component-orientedness, the elements of hybrid systems in the notation introduced in Section 2.2.3 are annotated with an index for the corresponding component, and the entire model results from joining component-specific sub-models:

- The finite set of component indices $B \subset \mathbb{N}$ and the specific component index b, which identifies the component addressed by the sub-model
- The component-specific domain of data states $\Sigma_{D,b}$
- The set of modes of the component Q_b with the active mode $\ell_{i,b}$ during interval I_i, see Subsection 3.1.3
- The activity assignments to each mode $\mu_{1,b} : Q_b \to f_{\ell_b}$, where f_{ℓ_b} is a DAE describing component behavior in mode $\ell_b \in Q_b$, see also Subsection 3.1.1
- The exceptions $\mu_{2,b} : Q_b, \Sigma_{D,b} \to \{0, 1\}$, see also Subsection 3.1.4
- The transition relations $\mu_{3,b} : Q_b, \Sigma_{D,b} \to Q_b, \Sigma_{D,b}$, see Subsection 3.1.3
- The explicit initial state specification $(\sigma_{0,b} \in \Sigma_{D,b}, \ell_{0,b} \in Q_b)$, see also Subsection 3.1.4

In SimScape, model assembly from the component sub-models happens automatically.

[1] Unfortunately, only structure and behavior specification profits from hierarchicality. Simulation and analysis of system behavior require full, flat models because DAEs have no such notion of hierarchical abstraction.

[2] Simulink allows no deeper ancestry than one ancestor, so changes to one component can only affect components *directly* linked through ancestry. SimScape does not have this limitation.

3.1.3 Discrete switching logic specification

Transition relations $\mu_{3,b}$ are specified for each component as a Finite State Machine (FSM). Due to the component-oriented notation, a component-specific location ℓ_b is introduced, called the component's *mode*. Modes for one component are mutually exclusive: A component can only either operate normally or be failed in exactly one way. The modes of the component are represented by the states of the FSM.

A subset of the system's continuous state is the input of the FSM: Only such elements of the continuous state of the system need to be handed over to the FSM which directly affect the component. This subset can be determined from a comparison of the component's activity to Bond Graph components in Lagrangian mechanics. In SimScape, the "through"- and "across"-sensors determine the equivalent of local "flow" and "effort" in Bond Graphs [43]. When components of linear, macroscopic physics are used, relevant states can only be the flow through the component and the effort dissipated or produced over the component. Components taking higher order effects into account may be treated similarly, but the final choice which elements of the component's state are relevant rests with the modeling engineer. In component-oriented hybrid system notation, it is proposed to select all data state variables of the relevant component as input to the FSM, regardless of whether all of them are used in transition conditions.

The transitions of the FSM represent changes in behavior from nominal to failure behavior. Their execution is controlled by *transition conditions*[3] specified with each transition, which are predicates on the input of the FSM.

Transitions in the proposed hybrid systems model are deterministic: For any given mode and continuous state input, at most one transition can be defined. If only inequalities on continuous system state are used as predicates, this ensures that for each mode, the continuous state space can be thought of as being partitioned into a subspace for which no transition is defined to happen, and one or more sub-spaces for each transition departing from that mode.

Additionally, when the transition condition is fulfilled, the transition is executed at the same simulation time step. In safety assessment, deterministic variance in aging effects, manufacturing imperfections, and random fault occurrence are handled by probabilistic failure models. This can be taken into account by defining such faults as modes with no transition condition based on continuous system state, but instead based on a discrete fault trigger that can be scheduled by the modeler. Modes reachable via both one or more transition based on continuous state and via a fault trigger are allowed as well, in order to cover faults that can be triggered by both probabilistic and deterministic effects.

Because the physical effects of a fault do not depend on how it was caused, fault tree models should use modes as basic events, not transitions. In the safety assessment process, this correlates to the Failure Modes and Effects Summary (FMES) process step at the interface between component reliability analysis and fault tree analysis, see also Section 3.1.6.

The output of the FSM is also specified with its transitions, making it a Mealy ma-

[3]Transition conditions are also called *guards* in publications with a focus on discrete behavior.

chine. They report the target mode and explicit functions for re-configuring the continuous state. The following cases can occur:

- *Buffer state alteration*: When mode changes abstract component behavior in a much smaller time scale than normal system behavior, the state of the affected continuous system states is altered so that they instantly contain the value after the 'fast' behavior.

- *Buffer merging or splitting* or *splitting*: When mode changes reconfigure the physical behavior of a component in such a way that the number of independent variables of the model changes, the energy, mass or information stored in the system needs to be redistributed. In such cases, the continuous state reconfiguration expression specifies the rules of redistribution.

Mosterman and Biswas [44, 2] require continuous state re-configurations to honor conservation of energy and mass. This work proposes to drop these restrictions in favor of allowing energy and mass dissipation. This is justified by the fact that physical models of real systems usually do not cover all domains of physics, and conservation generally applies only when all forms of mass and energy are taken into account. E.g. when an electric film capacitor is broken by overloading, not all electric energy it had stored at the point of avalanche breakdown is released as electric energy. Instead, the breakdown process chemically re-configures the dielectric material to become a high-resistance conductor in a violent reaction. Energy conservation in the electric domain would over-estimate the electric energy released into the circuit, as all energy in reality released as heat, spent on the chemical reaction etc. would have to be modeled in the electric domain. However, it is often undesirable to model all physical domains that are involved in each failure mode because model fidelity without some of them is still sufficiently high for the model's purpose. Dropping the requirement of energy conservation allows for simplified modeling of such phenomena.

The author argues that only dissipation should be allowed. Although it is theoretically feasible that physical domains outside the model's scope contribute enough energy (or mass) to over-compensate energy (or mass) dissipation from the modeled domains, this extreme case warrants deeper study and model extension. The modeling engineer should be motivated to include the physical domain that dominates fault behavior from the energy and mass conservation perspective, because the underlying physical phenomenon likely drives the safety design of the corresponding system portion.

3.1.4 Initial state, exceptions and inputs

Initial states $\sigma_{0,b}$ are always specified explicitly, in the form of initial assignments to the continuous state variables and a choice of an initial mode for each component. By default, nominal behavior is the initial mode for each component. This ensures that, when system engineers with no knowledge of the annotated failure behavior execute simulations, they are presented results for the expected, nominal behavior specification of each component — unless their simulation triggered any transition of a component

into a mode representing failure behavior. Thus, the hybrid simulation model becomes a digital prototype of the real system with the capability to "break".

Exceptions μ_2 can be specified using Assertion blocks from the 'Model Verification' Simulink library.[4]. They can be used for ensuring that modeling assumptions on the range of continuous states are met, e.g. limiting speeds to sub-sonic gas dynamics or electric peak power consumption of an electro-mechanic assembly to the linear operating region of the power supply.

The concepts of input-output-models of directional behavioral specification such as Simulink models can be translated into the semantics of non-directional, hybrid systems. For hybrid systems, outputs are measurements of continuous state and location. As with system behavior, continuous and discrete inputs are handled separately. Continuous inputs can be entered by adding dedicated system input components, which constrain specific continuous states to explicit trajectories. Discrete inputs are mode changes to specific components at pre-defined points in simulation time, i.e. dynamic injection of component faults.

3.1.5 Intermittent faults

Safety-critical embedded systems can be capable to recover from faults through continuous, built-in tests and automated recovery mechanisms such as self-resetting or -rebooting. This leads to certain faults only being active for a very short period until the built-in test detects it, followed by a period of unavailability during the execution of the recovery mechanism and, subsequently, fault-free operation. In safety assessment, such faults are called *repairable* faults. Such faults can be modelled by mode transitions that depend on continuous state and (relative or cyclical) simulation time or absolute simulation time (i.e. fault schedules, see 3.1.7).

3.1.6 Modeling considerations

Through the modeling semantics presented in this work, operational restrictions and probabilistic faults of components translate directly to the FSM specification for hybrid failure simulation. Component modes and transitions carry meaning and, in a fully model-based safety assessment process, are artifacts of safety assessment. As a consequence, changing FSM states representing faults is only allowed when changes are motivated in both points of view: FMES for safety assessment *and* finite state machine optimization for efficient hybrid behavior simulation. An intermediate FMES step between hybrid failure simulation and fault tree analysis is required to decouple FSM states representing behavioral modes in the model and basic events in the tree.

This is justified by transitions between hybrid modes being used both for abstracting microscopic nominal behavior and for abstracting the process of a component of failing.

For nominal behavior, there is no reason why nominal behavior of a component should not be represented by multiple states of the FSM. Consider the bouncing ball example that is often employed in illustrating hybrid systems: Mechanical contact with

[4] It is conceptually irrelevant whether exceptions are defined in component sub-models or globally.

the floor is no 'fault' by any definition, but a discrete physical property in macroscopic physics. It could be modeled as a nominal operation mode during contact, or as a transition that simply reverses the direction of bouncing ball's impulse.

But treating nominal and failure behavior differently in this respect seems arbitrary: By allowing multiple nominal FSM states for the nominal behavioral mode, but only one FSM state for each fault, the direct coupling between basic events in fault trees and hybrid modes in the behavioral model would impose undue limitations on the modeler.

Therefore, the role of FMES in traditional safety assessment needs to be expanded. In traditional safety assessment, it groups physically distinct faults with identical effects on component behavior [28] and faults on sub-components. Physically distinct faults with identical effects need to be identified before hybrid mode definition, so that the hybrid mode is modeled only once. Faults on sub-components equally need to treated as a group and modeled as a single hybrid mode as well. Both these modeling steps relate to model granularity choices that are relevant for nominal behavior modeling as well. Thus, it is suggested to carry them out as a common task for design and safety engineers.

3.1.7 Transitions and fault simulation schedules

Transition conditions based on continuous system state allow modelers to capture fault propagation as a physical effect on the modeled components. As such, it is irrelevant whether it was caused by external events (e.g. lightning strike) or fault propagation (e.g. short circuit in other system portions). This has several beneficial effects:

- Fault propagation can be simulated as realistic as the continuous model portion is modeled.
- Fault propagation is context-independent because it is specified only on components, not models.
- The interaction of external events and internal faults can be simulated without additional modeling effort.

3.1.8 Requirements towards behavioral failure modeling languages

Joshi and Heimdahl [1] have stated requirements towards behavioral failure modeling languages, which ensure their applicability in industrial practice. This subsection explains how the presented operational semantics for hybrid failure simulation fulfills them.

1. *"**Component Fault Behavior**: The notation must enable the engineer to specify component fault behaviors for both internal faults and vulnerabilities to external faults."*
   Internal and vulnerabilities to external faults are specified as hybrid modes.

2. *"**Explicit Associations**: Since the system fault model is defined separate from the nominal model, the notation must enable specifying explicit associations between the relevant fault behaviors and the nominal components."*
   Hybrid modes (representing faults or nominal behavior) are defined on model components. One nominal behavior mode is the default mode.

3. *"**Multiple Associations**: Since a component can fail in more ways than one, the notation must enable associations of more than one fault behavior to a particular component."*
   Each model component has at least one hybrid mode representing nominal behavior. It can have an arbitrary number of faults.

4. *"**Conflict Resolution**: A conflict may occur between the multiple fault behaviors (multiple internal fault behaviors or vulnerabilities) associated with a single component. These conflicts must be resolved by defining some form of priorities or user-defined strategies."*
   Model components can only have one active mode. Transition conditions are required to be disjoint, thus conflicts are prevented at modeling time.

5. *"**Nominal Component Types**: For flexible associations, a notion of component types must be supported. The user can specify component types to group together nominal components that have similar nominal or fault behaviors for the purpose of easy associations."*
   Model components derive from library components that are parametrized at insertion into the model. The information from which library component a model component derives is persisted.

6. *"**Trigger and Persistence/Duration**: The language shall support the trigger and persistence specification for both internal and external faults. It shall also support the specification of conditional fault activation, where the trigger and the persistence will be controlled by the condition."*
   Manually fault triggers can be scheduled. Fault transition conditions can be defined on continuous system state, so the duration an automatically triggered fault is active depends on continuous simulation.

7. *"**Error Propagation Rules**: For identifying and activating the external faults, in addition to specifying the vulnerability behaviors, the notation shall also support specification of error propagation rules [. . .]."*
   Because transition conditions can be defined on the continuous system state, faults propagate through the system as any behavior does, which makes explicit fault propagation rules obsolete.

8. *"**Fault Model Hierarchies**: For more flexibility, the engineer must be able to successively specialize fault behavior definitions as the design of the system and fault model progresses."*
   Model components' links to their library counterparts can be disabled or broken, enabling the possibility to refine fault behavior specification without affecting library components.

3.2 Implementation aspects

In the course of this work, a prototypical implementation for a hybrid modeling and simulation engine for the presented operational semantics has been developed under the name *HSim*. It consists of the following components:

- Base modeling engine for physical simulation (DAE-based)
- Hybrid component modeling framework
- Simulation engine for discrete transition detection and execution

A MATLAB-based [5] software stack was employed with custom components extending it as needed:

- Simulink and Simscape as a base modeling engine
- Stateflow and Simscape as a hybrid component modeling framework
- MATLAB and custom Simulink-components for discrete transition detection and execution
- A simple Graphical User Interface (GUI)

The custom components of HSim are grouped by their layer name in the Model-View-Controller software architecture pattern. *CoreController* contains the transition detection and execution, *CoreView* the Graphical User Interface (GUI) and *CoreModel* contains shared data model classes for all components of the package and resources for hybrid component modeling. *CoreLibrary* supplies a set of hybrid components packaged into a Simulink library, as a proof of concept that the library block parametrization principle of Simulink can be applied to hybrid components as well. A set of sample models is also provided with this work in the *TestModels*-folder[5].

Subsection 3.1.3 explains that transition conditions need to be disjoint, so that only one transition is set to be carried out for each *individual* component mode and continuous state. The implemented hybrid component modeling framework does not test for this and cannot warn the user during model design. During simulation, however, a warning is issued when such a situation occurs. Even if *individual* components do not trigger multiple transitions at the same time step, *multiple* components may do so. As an example, consider a similar redundancy (e.g. for reliability improvement). A failure that propagates to both similar components at the same time step may trigger hybrid mode transitions for both of them. Their transition relations $\mu_{3,b}$ may set states in- or outside of the respective component. There is no general way to test for compatibility of the transition relations, so for the sake of greater experimental freedom, the transition detection engine issues only a warning. Thus, when multiple transitions for identical time steps are triggered, only warnings are issued, regardless of whether they originated from an individual component, multiple components or a mixture of both[6]

[5] Samples for the other chapters of this work are given in the same folder.

[6] E.g. when two transitions are triggered for one component, and another transition for another component, all at the same time step.

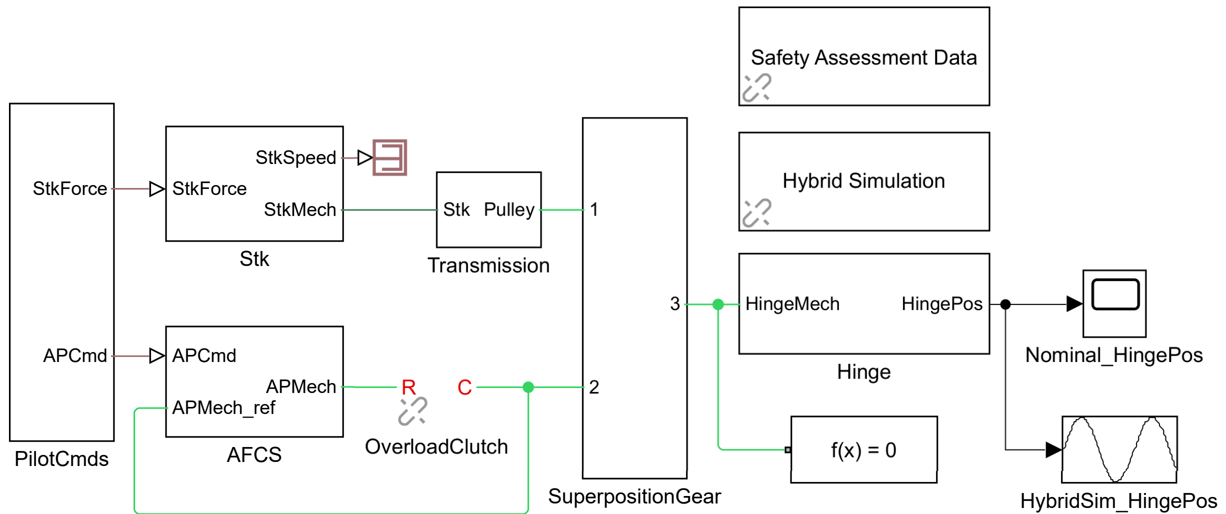

Figure 3.1: hybridFCS model structure

The same Subsection 3.1.3 also recommends avoiding energy or mass input into the system over transitions. No warning is issued if this is not followed, neither at modeling nor at simulation time.

All source code written in the course of this work has been commented extensively.

3.3 Case study: Simulating a hybrid flight control system's failure behavior

Hybrid, electro-mechanical Flight Control System (FCS) have a mechanical portion for direct pilot commands and an Automatic Flight Control System (AFCS) for auto-flight and flight control functionality with an electro-mechanical actuator. The outputs of both portions are superimposed at the attached flight control surfaces. Safety assessment of such a system typically concentrates on the effects of faults of the autopilot, which is the only non-conventional portion of the system. As a measure of failure propagation limitation against erroneous autopilot output, an overload clutch disconnects the autopilot's actuator from the mechanical portion. This architecture has been modeled by Lauffs [45], and has been simplified for showcasing hybrid simulation of failures on a practically relevant example. In Chapter 5, the same model is used for a case study on fault tree generation.

The model created for this purpose has been named 'hybridFCS'. Figure 3.1 shows a block diagram of its top-level model structure[7]. Custom behavior blocks used for simulating failure behavior of the AFCS, such as loss and erroneous output, are provided in a block library called 'hybridFCSlib'.

Hybrid failure behavior of blocks in the model are given in Table 3.1. In order to be

[7]The 'Safety Assessment Data'-block is relevant only for tests of the implementation of Chapter 4, which use the same model.

Table 3.1: hybridFCS blocks and failure behaviors

Block	Behavior	Description
AFCS	Loss	No output (always 0)
AFCS	Erroneous	Random output in steps of 1 second
OverloadClutch	StuckOpen	No contribution of the AFCS to the hinge
OverloadClutch	StuckClosed	Full contribution regardless of torque/speed
Hinge	JammedUp/Down	Fixed hinge position at mechanical limit

able to showcase failure propagation, the hinge's failure behaviors JammedUp/Down are triggered when the hinge crashes into its mechanical limits with a force beyond a given threshold.

In nominal behavior, the hinge motion displayed in Figure 3.2 as a solid line is simulated. It shows manual pilot commands in the interval $[1\,\mathrm{s}, 6\,\mathrm{s}]$ superimposed with autopilot output in the interval $[3\,\mathrm{s}, 8\,\mathrm{s}]$. No hybrid mode change is reported in nominal simulation.

Simulating an erroneous AFCS triggers the overload clutch to open, protecting the hinge from jamming. This is shown in Figure 3.2 as a dashed line. In this experiment, the erroneous AFCS behavior starts at 1 s and (randomly) has no effect until 3 s. Then, it starts to move downward but the overload clutch disconnects the actuator before higher speeds are reached. Hinge movement hits its rotational limits, but soft enough for it not to jam. Only the mode change for the fault of the AFCS is reported during simulation.

When both the AFCS shows erroneous behavior and the overload clutch is stuck closed, the failure can propagate to the hinge. Figure 3.2 shows such a scenario as a dotted line, where an erroneous AFCS command causes the hinge to jam in the upper position. The loss of torque limitation from the autopilot from the overload clutch leads to faster hinge movement with sharper speed changes. A (random) downward burst around 6 s jams the hinge in its maximum downward position[8]. The simulation engine reports the two scheduled faults as well as the triggered jamming of the hinge.

In practice, the safety engineer would execute a series of randomized experiments with active faults. Thus simulation performance is a relevant measure of the maturity of the implementation. On a consumer-grade computer, the hybrid simulation overhead results in a five-fold increase in computational effort over standard Simulink simulation (49 s vs. 9 s).

The simulation experiment makes the driver of safety design, the dependency of AFCS torque output and maximum overload clutch torque transmission from stick force and maximum torque at the hinge's limits, apparent. The implementation's modeling principles promote the co-design of nominal behavior-related and safety-related functionality while keeping the aspect-specific results of this process separate.

[8]The downward movement beyond movement limits around 9 s is due to a simplification in the model of the AFCS.

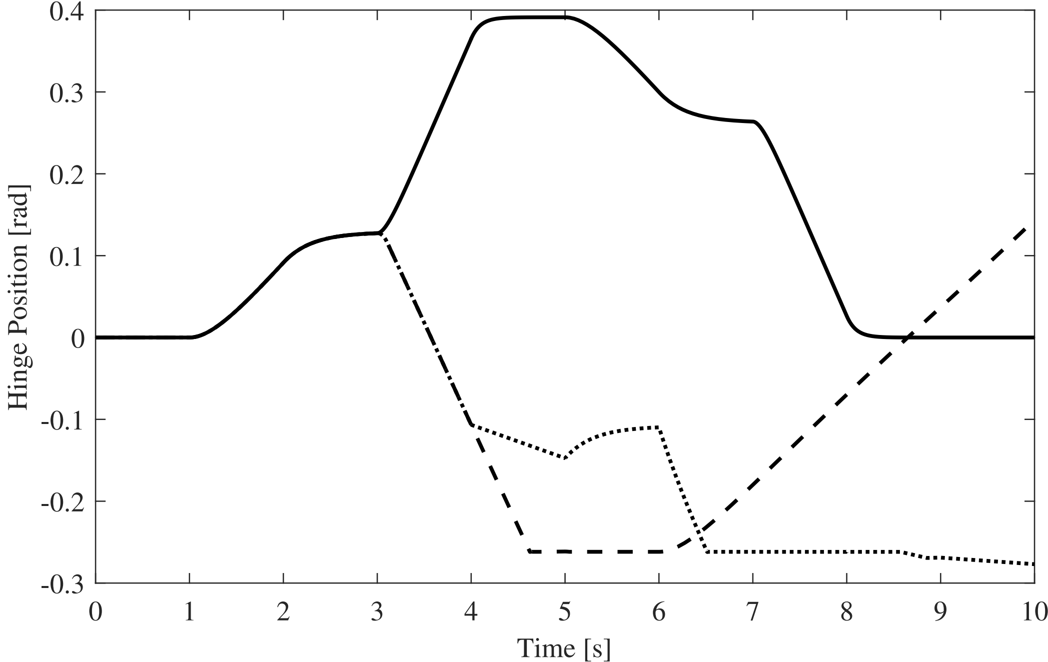

Figure 3.2: Simulation results for hinge movement: nominal behavior (solid line), erroneous AFCS behavior (dashed line) and erroneous AFCS with the overload clutch stuck closed (dotted line)

Chapter 4

Completeness of Fault Trees

The second contribution of this thesis consists of a formalization of the completeness of a fault tree as a predicate on Boolean functions abstracting failure behavior, and a method for its efficient exploitation in fault tree generation and validation. It has been published separately (see [46]).

Both generation and validation of fault trees are time-consuming and error-prone, due to the same property of a fault tree that makes it expressive yet intuitive: Only events that contribute to the top event are displayed. In fault tree generation, this requires the author to keep in mind all implicit exclusions of non-contributing events he has made, along with their rationale, in order to ensure consistency. Especially in the analysis of highly severe top events, in which safety features of system architecture commonly are diverse (as per dissimilarity requirements) and numerous (as per the single fault robustness requirement for catastrophic Failure Conditions (FCs)), this adds to the inherent challenge of the task. When validating fault tree completeness, the same implicit exclusions and their corresponding rationale would need to be retraced from the finished fault tree and its accompanying documentation. A complete fault tree must not make any implicit exclusions without justified rationale — and a simple fault tree would exclude as much probability as required to undercut the certification limit and any single fault robustness requirement with as little distinct rationale as possible.

In this section, this problem is formalized, the proposed solution is first characterized and then described, and its usefulness is showcased with the well-known Wheel Brake System example from ARP 4761 [28].

The formalization of completeness is similar to that of Schellhorn et al. [4] but allows for a computationally more efficient solution algorithm. It uses the concepts of efficient manipulation of Boolean formulae employed for fault tree solution based on Binary Decision Diagrams (BDDs) proposed by Bryant [47], applied and proven by Coudert and Madre [38] and improved by Rauzy and Jung et al. [48, 40]. The proposed algorithm allows highly efficient iteration of the leaves of a probability-weighted Binary Decision Diagram (BDD) in the order of their probability. It is employed for fault tree validation with a minimal effort given the fault tree and the order of variables in the BDD.

The original contribution of this thesis lies in

- its mathematically strict formalization of fault tree completeness, and
- its algorithm for iterating over disjoint sub-spaces of the solution space in order of their associated probability.

The algorithm essentially applies fault tree completeness in order to enable efficient fault tree generation and validation. These results have already been published by the author (see [46]).

4.1 Relation to the safety assessment process

As per ARP 4761 [28], fault trees are used to break down high-level safety requirements from aircraft- to system- and from system- to item-level[1] in a deductive manner. ARP 4754A [35] requires that safety requirements are validated.

For the validation of high- and low-level safety requirements, a catalog of correctness and completeness criteria is specified in ARP 4754 [35]. Among the completeness criteria, the first is that it should be "apparent from the traceability and supporting rationale that the requirement/-s will satisfy the parent requirement." Thus determining whether the union of all analysis errors (for completeness i.e. omitted critical combinations of events) violates higher level safety requirements is the formal goal of validation activities. Conversely, validating the fault tree is finished once large enough a portion of the fault tree has been validated so that the remaining part, even if it is completely wrong in an optimistic way, cannot jeopardize fulfillment of the higher-level safety requirement.

For fault tree generation, achieving apparency is the second concern that results from the above completeness criteria. A fault tree that needs extensive supporting rationale is thus to be regarded as inferior to one that confines itself to few but important items of rationale, and pessimistically simplifies the rest. Identifying which safety features of a system make up such a minimal set would ideally be the core of fault tree generation. These safety features are the same rationale that would be checked first during validation. Fault tree validation could be considered the dual problem of fault tree generation.

4.2 Functions of static failure logic

Boolean logic makes automated reasoning simple, but require that all variable domains are confined to $\{0, 1\}$, and that time is abstracted away. Boolean functions encoding failure behavior are thus called functions of *static failure logic* in this work. The Boolean function encoded by a fault tree is called its *characteristic function*, so static failure logic are a natural form for discussing abstraction in the context of Fault Tree Analysis (FTA).

Before implementation of the system under study is completed, system failure behavior is modeled either descriptively (e.g. SysML [49]) or in analytic, executable models (e.g. see Chapter 3 of this work)[2]. They necessarily have common properties:

[1] For particularly complex systems, e.g. flight controls and engines, an additional subsystem-level may be introduced to facilitate inter-organizational communication.

1. They reflect inherent system complexity.
2. Their model class is more expressive than Boolean functions $\mathbb{B}^n \to \mathbb{B}$.
3. They are supposed to describe system failure behavior adequately, just as well as nominal system behavior.

Because of this, their abstraction into a Boolean function is intrinsically difficult. The rest of this thesis assumes that analytic, executable models are not accessible to formal verification and costly to evaluate in comparison to the fault tree's characteristic function.

4.3 Pessimism and fault tree completeness

In qualitative FTA, only single fault robustness requirements can be handled, while probability requirements necessitate quantitative FTA. For each of these two requirement types, a formalization is given in this section. The last class of safety requirements, Function Development Assurance Level (FDAL) and Item Development Assurance Level (IDAL) assignments, are not covered by this work.

Single fault robustness requirements can be formalized as predicates on Minimal Cut Set (MCS) cardinality, or — in the terminology of boolean reasoning — prime implicant cardinality. Bringing together

- the characteristic function of the fault tree under consideration $F(\mathbf{x}) := \mathbb{B}^m \to \mathbb{B}$,
- the set of F's prime implicants $\text{PI} \subset \mathbb{B}^n, \text{PI} := \{\pi \mid F(\pi) = 1\}$,
- a prime implicant π's cardinality $|\pi|$,

the predicate of single fault robustness μ_{SFR} can be written as

$$\mu_{SFR} := (\forall \pi)\, |\pi| > 1 \tag{4.1}$$

Probability limit requirements formalization is straightforward: With the probability obtained through fault tree solution $P(F(\mathbf{x}) = 1)$ and the probability limit P_{cert}, a probability limit requirement μ_{Prob} can be written as

$$\mu_{Prob} := P\left(\bigcup_{\text{PI}} \pi\right) \leq P_{cert} \tag{4.2}$$

In conventional safety assessment, higher-level safety requirements are fulfilled when the aforementioned requirements in eqns. 4.2 and 4.1 derived from it are fulfilled. However, both types of requirements assume that the fault tree identifies all relevant contributing item faults and environmental or operational conditions $\mathbf{x} \in \mathbb{B}^m$. If the domain of the characteristic function covers all such contributors and it fulfills the above completeness criteria, the fault tree is complete. Note that this allows for the fault tree to indicate the top event to occur when, in 'reality'[3], it does not. Infor-

[2]Descriptive models focus on system structure; Behavior is only one property among others. Analytic models make future system behavior "playable" and not necessarily describe logical or physical system structure. Discussion of the underlying design paradigms is out of the scope of this work.

[3]'Reality' as formalized as reference behavior, see section 1.4.4 of this work.

mally, this defines *fault tree pessimism*: When the fault tree predicts the system to fail, it may or may not fail in reality — the fault tree is a pessimistic approximation of failure behavior.

The formal definition of fault tree completeness is based on the comparison of reference and fault tree-encoded failure behavior. A fault tree's characteristic function is readily defined as a Boolean function. For the reference behavior, a Boolean function $G(\mathbf{y})$ is defined. It is not pessimistic: Only when a combination of item faults, environmental events (e.g. lightning strike) and operational conditions (e.g. wet or icy runway) causes the real system to fail, its static failure logic $G(\mathbf{y})$ returns 1. It could be used for fault tree verification, but because it is assumed to be impossible to abstract and costly to evaluate, minimizing the need for its evaluation is a relevant contribution to bridging the abstraction gap between reference behavior and static failure logic encoded in fault trees.

The input domain of $G(\mathbf{y})$ in general is not the input domain of the fault tree's characteristic function $F(\mathbf{x})$, but a superset of it: $\mathbf{y} \in \mathbb{B}^n \supseteq \mathbb{B}^m \ni \mathbf{x}$: It may take into account item faults, environmental events or operational conditions the fault tree ignores. In order to reason about such cases, the difference between the two domains is defined to be $\mathbf{d} \coloneqq \mathbf{y} \backslash \mathbf{x}$, and a compatible version of F is defined as $\hat{F}$ that 'does not care' about the variables $\mathbf{d}$.

With $G(\mathbf{y})$ and $\hat{F}(\mathbf{y})$ defined formally, it is clear that a fault tree can be incomplete for two distinct reasons:

- The input domain of F is too small: It can fail to take relevant basic events into account, i.e. $\mathbf{x}$ misses events that are part of a MCS of $G(\mathbf{y})$.
- The logic of F is optimistically wrong: It can take all relevant basic events into account, but not return 1 for an assignment corresponding to a case in which the real system fails.

In both cases, the effect is a false negative approximation in the fault tree: It does not predict failure when, in reality, the system fails.

Finally, fault tree completeness can be expressed as a predicate of Boolean inclusion of true system behavior in static failure logic in the fault tree's characteristic function:

$$\mu_{Completeness} \coloneqq (\forall \mathbf{y})\, G(\mathbf{y}) \leq \hat{F}(\mathbf{y}) \tag{4.3}$$

The next section describes the properties of the proposed method for exploiting eqn. 4.3, and the one after describes how the proposed method works.

4.4 Properties of the proposed method

The most naïve algorithm for evaluating eqn. 4.3 would be to evaluate every possible assignment $\mathbf{y} \in \mathbb{B}^n$. This is impractical due to combinatorial explosion. However, the search for assignments violating eqn. 4.3 can be reduced in such a way that combinatorial explosion is no longer an issue. This corresponds to the results that curb combinatorial explosion in fault tree solution by Rauzy and Dutuit [48] and Jung et

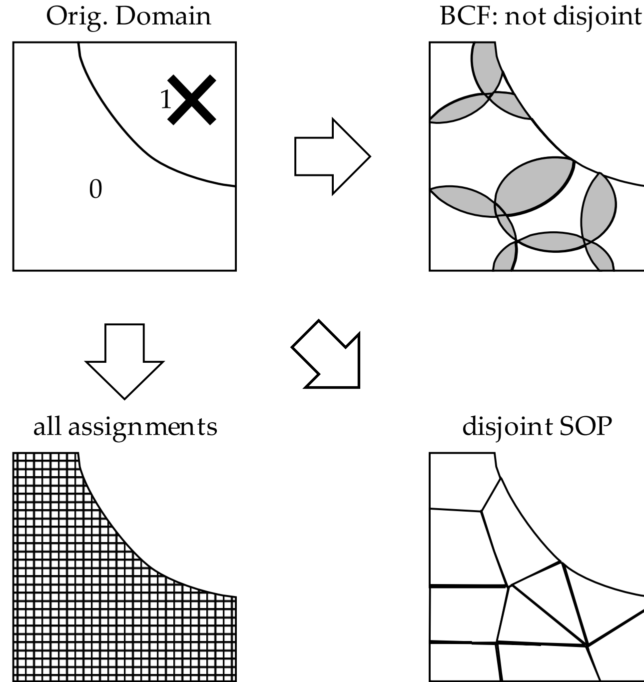

Figure 4.1: Partitioning strategies: Two-dimensional illustration of the high-dimensional input domain, mapping to $\mathbb{B}$, partitioned using BCFs (non-disjoint), explicit assignments to all variables (many elements) or disjoint SOP terms

al. [40]. It exceeds similar efforts by Schellhorn et al. [4] by not only regarding fault trees as graphical forms of Boolean functions, but also taking their probabilistic nature and their role in the safety assessment process into account through probability truncation of the search for violating assignments.

Combinatorial explosion is not the only concern when evaluating eqn. 4.3. Due to the effort required to evaluate G, i.e. to abstract reference behavior into a Boolean function of static failure logic, minimization of the number of evaluations of G is paramount. The following subsections explain the measures applied to reduce the number of required evaluations.

Evaluating only negative scenarios

Only assignments $\mathbf{y} \in \mathbb{B}^n$ for which $\hat{F}(\mathbf{y}) = 0$ can render eqn. 4.3 false. In safety assessment, this is equivalent to stating that only scenarios where the top event is not expressed to occur by the fault tree can render it incomplete. We call such assignments a *negative scenario* and denote the set of all negative scenarios $Y_0 \subseteq \mathbb{B}^n$:

$$Y_0 = \left\{ \mathbf{y} \mid \hat{F}(\mathbf{y}) = 0 \right\} \tag{4.4}$$

In Figure 4.1, this is illustrated: The portion of the input domain mapping to 1 is not taken into account in further processing.

Disjoint Sum of Products (SOP)-formulas for denoting negative scenarios

Solution strategies for Boolean equations (for finding $G(\mathbf{y}) = 0$) depend on the desired description format of the result. Before proposing a solution strategy, the desired description format is thus characterized.

For computational efficiency, a format that describes the solution space (or here, its complement) by as few, disjoint sub-spaces as possible would be most beneficial. This is due to the fact that each sub-space $\mathbf{y}_0 \in Y_0$ will, in practical application, be associated with a justification or proof that, for all assignments in that sub-space, $\hat{F}(\mathbf{y}) = 0$. Such pieces of justification or proof are called *rationale* in the context of fault tree validation. Few pieces of powerful rationale[4] are preferable over many weak pieces because they yield the same result — a validated fault tree — with less effort. Disjointness ensures that no piece of rationale overlaps with that of another sub-space, so they can be validated separately and cannot contradict each other. This is illustrated in Figure 4.1.

The two common formats for describing the solution space of a Boolean equation do not have both of these properties:

- Explicitly listing every assignment to all variables that satisfies the equation ensures sub-space disjointness but yields the maximum number of sub-spaces without repetition.
- The Blake Canonical Form (BCF) yields the minimum number of sub-spaces but they are not disjoint.

Thus neither of them is suitable for the purpose of this work. A suitable description format would be some sort of SOP-formula. They can be evaluated quickly by considering terms only until the first term evaluates to 1. Fortunately, paths from a BDD's root node to its 1 (or 0) node yield disjoint product terms describing sub-spaces of the solution space (or its complement) that can be arranged trivially in a SOP-formula by ORing them, as described in the following section. BDD reduction ensures that the resulting number of sub-spaces is also comparably small.

Boole's expansion theorem and BDDs for obtaining disjoint negative scenarios

Applying Boole's expansion theorem to $\hat{F}$ for each of its variables $y_i \in \mathbf{y}$ yields its BDD (see [47]), whose paths are disjoint product terms by construction. Each path from the root node to a 0-node describes a negative scenario. This can be visualized in BDDs as described in section 2.3.3. Each path from the root node to a terminal node correlates to an assignment that, when evaluating the underlying Boolean function for it, yields the result given by the path's terminal node. Graphically, the assignment can be read from the path by ANDing each node assignment on the path. The number of negative scenarios is thus the number of paths from the root node to the 0-node, and depends on the variable ordering.

Because such a path can be formalized as a BDD itself, BDDs provide a closed arithmetic for the purpose of this work. For an example, re-consider Figure 2.3: The paths

[4] *Powerful* in that the rationale is applicable to a sub-space of large probability.

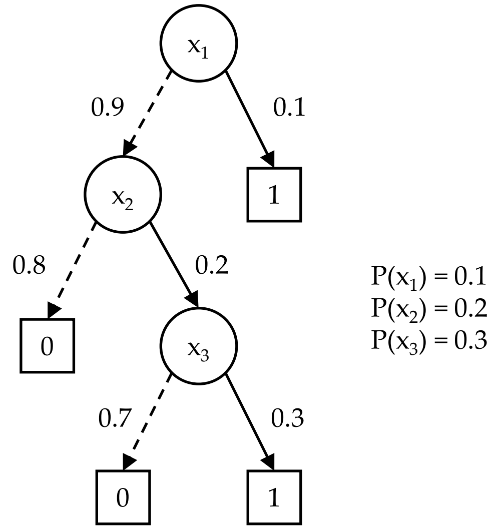

Figure 4.2: Probability-weighted BDD with basic event probability associated with 1 and complementary basic event probability associated with 0 edges.

to its 0 nodes fully describe the complement of its solution space, and can be 'read' from the BDD as $x'_1x'_2 + x'_1x_2x'_3$.

The availability and size of the BDD of $\hat{F}$ is, in practice, not a problem. BDD size problems can be curbed easily: Even though the maximum number of nodes of a BDD encoding a Boolean function with n variables is 2^n and depends on variable ordering, the maximum is seldom reached. Even when a size near the maximum is obtained from natural variable order, reordering variables in such a way that variables are ordered by their corresponding basic event's distance to the top event (and randomly where distance is equal) mitigates the problem (see [39])[5]. Availability is also not problematic: The characteristic function's BDD is used for fault tree solution anyway, and the BDD of $\hat{F}$ is the same as that of F. Thus, there is no need to take BDD creation into consideration for determining the computational efficiency of the presented algorithm.

Using edge weights to convey stochastic information into the BDD

By weighting the edges[6] with basic event probability (for the 1 edge) or its complement (for the 0 edge, respectively), the stochastic information of the corresponding fault tree is captured directly in the BDD, as illustrated in Figure 4.2. This allows obtaining the probability P of a scenario $\mathbf{y}$ — if all its basic events are independent — by multiplying the probabilities associated with the edges of its corresponding path in the BDD:

$$P(\mathbf{y}) = \prod_{i=1}^{i=n} P(y_i \in \mathbf{y}) \tag{4.5}$$

[5] Rauzy gives no proof for the fitness of this heuristic, and the selected BDD computing component in this work simply reorders variables randomly when storage space grows above a certain threshold.

[6] Weighted BDDs were originally proposed by Ossowski and Baier [50].

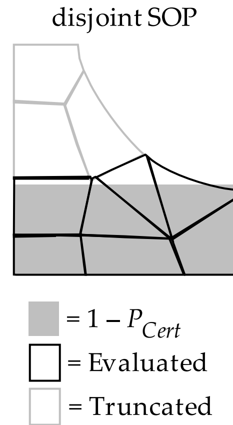

Figure 4.3: Probability truncation: Two-dimensional illustration of high-dimensional input domain, mapping to 0, partitioned using disjoint SOP terms. Once enough scenarios have been evaluated for the complement of the certification limit to be passed, evaluation can stop.

The underlying assumption of eqn. 4.5 that all basic events in a path are independent is pessimistic: Common Cause Analysis (CCA) ensures that they are not related in a stronger way than independence, i.e. they can be independent or even antagonistic to the point of being disjoint. Examples of less strongly related basic events than independent are faults of the same item (disjoint), basic events describing flight phases (disjoint), or specific redundancy concepts. This topic is not discussed further in this work and proposed for future investigation in the last chapter.

Note that all outgoing edges starting at the same index and of the same kind (1 or 0-edges) have the same weight. This is also true for edges spanning multiple indices, as obtained by BDDs reduction. This needs to be taken into account during implementation to avoid associating memory storage costs of one edge weight per edge. A prudent implementation needs to store only two edge weights per index.

Determining fault tree completeness only to the degree of certainty required by the corresponding quantitative safety requirement

With stochastic information from the fault tree available to the analysis of fault tree completeness, another pessimistic simplification is introduced: It is not required to evaluate G for the *entire* sub-space Y_0 of negative scenarios. The exact probability of all negative scenarios of $P(Y_0)$ is not relevant for fault tree completeness. It only needs to be less than the complement of the certification limit P_{cert}. Once enough scenarios have been evaluated for passing that threshold, the process can be stopped because the fault tree is proven to be *sufficiently* complete. The sub-space of Y_0 evaluated so far is denoted $Y_{0,\partial}$ This can be formalized as a predicate:

$$\mu_{TracingComplete} :\!= P\left(\bigcup_{Y_{0,\partial}} \mathbf{y}\right) \geq 1 - P_{cert} \tag{4.6}$$

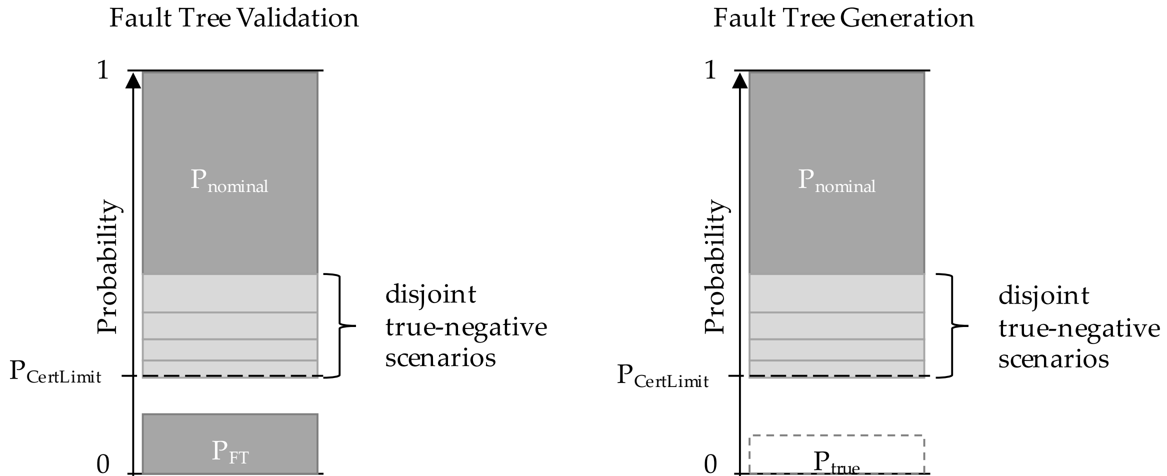

Figure 4.4: Concepts of fault tree pessimism: Preferring high-probability negative scenarios allows verification with minimal effort. Similarly in fault tree generation, excluding high-probability negative scenarios first allows generating a complete fault tree of minimal size for the given FC and system under consideration.

The process of stopping evaluation when this predicate returns 1 is called *probability truncation*. Figure 4.3 illustrates the concept.

Evaluating negative scenarios in order of their probability

Probability truncation in the context of fault tree verification is illustrated on the left of Figure 4.4. Negative scenarios are evaluated one after another. Nominal behavior, when no faults occur, can be skipped: It can be assumed to be the negative scenario of the highest probability, because failure probability usually is much lower than .5. Beyond that, the number of required evaluations of G to prove completeness is minimal when the individual probability of the evaluated negative scenarios is maximal. In other words, few negative scenarios of high probability for reaching the certification limit are preferable over many negative scenarios of low probability.

In BDDs, tracing paths of high probability is achieved by keeping a sorted list of the weights of all untraced edges during tracing, and following edge by edge in descending order of their probability (see 4.5). Thus the BDD under consideration is only traced as far as required to evaluate the negative scenarios that have the highest probability. Especially for large BDDs, this is beneficial for the overall method's computational efficiency.

Two factors influence the number of negative scenarios required for proving completeness: The order in which they are evaluated (preferably large scenarios first) and — because BDDs are used to construct the negative scenarios — BDD size and thus the ordering of the variables in the underlying BDD. In Figure 4.3, this can be graphically interpreted as different ways of partitioning the solution space that have the same properties (disjoint SOP formulas). This means that, with n variables, there are $n!$ possible orders of variables, and thus $n!$ ways to partition the solution space. Finding the weighted BDD that yields the most efficient partitioning for the purpose of this work is

an extension of the problem of finding the smallest BDD for a given Boolean function, and has consequently not been examined further.

Expanding to full assignments before evaluation

Negative scenarios obtained by tracing paths in a BDD by construction omit variables that have no effect on the corresponding Boolean function's outcome. We call assignments that replace *every* variable with a value *full assignments*, denoted $\hat{\mathbf{y}}_0$. Complementarily, we call those that replace only a part of the set of variables *partial*. Because full assignments vary widely in probability (the most probable being the nominal case) and can also be traced in order of probability, there are situations where applying Boolean expansion after tracing to the relevant negative scenarios is useful.

In the next step of fault tree validation, the result of the proposed method would be used in formal validation or simulation to gather a piece of rationale for each evaluated assignment to be a *true* negative. If the actual technique of gathering pieces of rationale has to consider a definitive state of each variable, it is efficient to continue Boolean expansion on the negative scenario with the omitted variables: Few full assignments gained from a negative scenario with unassigned variables usually contain the vast majority of its probability, and Boolean expansion is computationally cheaper than gathering many pieces rationale.

In order to gain a full assignment from a partial assignment represented by a path $\mathbf{y}_0$, a variable to be expanded must first be selected. The variable not in the path already, whose probability is farthest from 0.5, leads to the highest probability full assignment, so it is selected for the next step[7]: With the selected variable denoted x_e, the original partial path is replaced with two paths $\mathbf{y}_0 \cdot x_e$ and $\mathbf{y}_0 \cdot x_e'$. The result can be a path for full or a longer partial assignment. This is repeated on the partial assignment with the highest probability in the set of all partial assignments until enough probability has been gathered in full assignments so that the probability of their union is equal to or greater than the complement of probability limit of certification. This can be formalized as a predicate:

$$\mu_{ExpansionComplete} \coloneqq P\left(\bigcup_{Y_0,\partial} \hat{\mathbf{y}}\right) \geq 1 - P_{cert} \tag{4.7}$$

Note the similarity to the probability truncation predicate in eqn. 4.6. Here, only full assignments are taken into account.

When G is not evaluated but reference behavior is analyzed manually or through formal verification (when applicable to the system under study), it is unnecessary to provide different justification for two assignments when they are found to lie in the same minimal path set. This fact is used by Schellhorn et al. [4] in their approach that is based on formal verification, but as it does not take probability truncation into account, it is not only limited by the preconditions to formal verifiability but also by combinatorial explosion in large fault trees.

[7]When $P(x_e) < 0.5$, $P(\mathbf{y}_0 \cdot x_e')$ is greater; When $P(x_e) > 0.5$, $P(\mathbf{y}_0 \cdot x_e)$ is greater. If all basic events' probabilities are < 0.5, the variable representing the basic event with the highest probability can be selected.

4.5 Description of the proposed method

The proposed method can be separated into two consequent parts: Evaluation of the BDD or *tracing*, and — if full assignments are preferred, e.g. for simulation as per Chapter 3 — expansion of negative scenarios obtained by tracing. Both parts store intermediate data on three lists of (partial) paths:

- The *tracing backlog* stores partial paths that are possible next edges to evaluate for the algorithm. I.e. it stores "where the algorithm could trace next".
- The *results backlog* contains paths that have not yet been added to the results list because it has not yet been ensured that its path of the highest probability has a higher probability than any path the algorithm can still find. I.e. it stores "what paths the algorithm has seen but not yet returned as a result".
- The *results list* contains the BDD's (probability-truncated list of) paths in order of their probability. I.e. it stores "the result of the algorithm".

Both the entries tracing and in the results backlog are stored in descending order of (partial) path probability. The entries in the results list are by construction ordered in this way by the presented algorithm.

Tracing is illustrated in Figure 4.5. It begins with the root node of the reduced BDD and initializes by storing its edges in the tracing backlog.

Then, the 'tracing' loop begins. Each time, the partial path from the tracing backlog with the highest probability is evaluated and removed from the tracing backlog. As during initialization, if the 0-node is reached, it is added to the results backlog. If the 1-node is reached, the path is discarded. If no terminal node is reached, the partial path is added to the tracing backlog. Before the tracing loop continues with the next edge, the inner 'checking' loop is executed.

During the checking loop, the paths in the results backlog are compared to the partial path in the tracing backlog with the highest probability in descending order of probability: If the 'best' path's probability is higher than the partial path's, it is moved to the result list. If the result list's probability is higher than that of the complementary certification limit on probability, tracing is complete. The checking loop is quit when a path is of lower probability than the partial path of with the highest probability.

Expanding the paths obtained by tracing works similarly to tracing and is illustrated in Figure 4.6. The tracing backlog is filled with the negative scenarios obtained by tracing. As in tracing, the expansion loop works on the most probable path in the tracing backlog. In contrast to tracing, it begins with expanding the current path on the variable not in the path that has its probability farthest from 0.5. For moving the expanded paths to the results backlog, they need to represent full assignments, i.e. they have to assign a value to every variable. If they do not, they are added to the tracing backlog. In every other respect, the expansion process is the same as the tracing process.

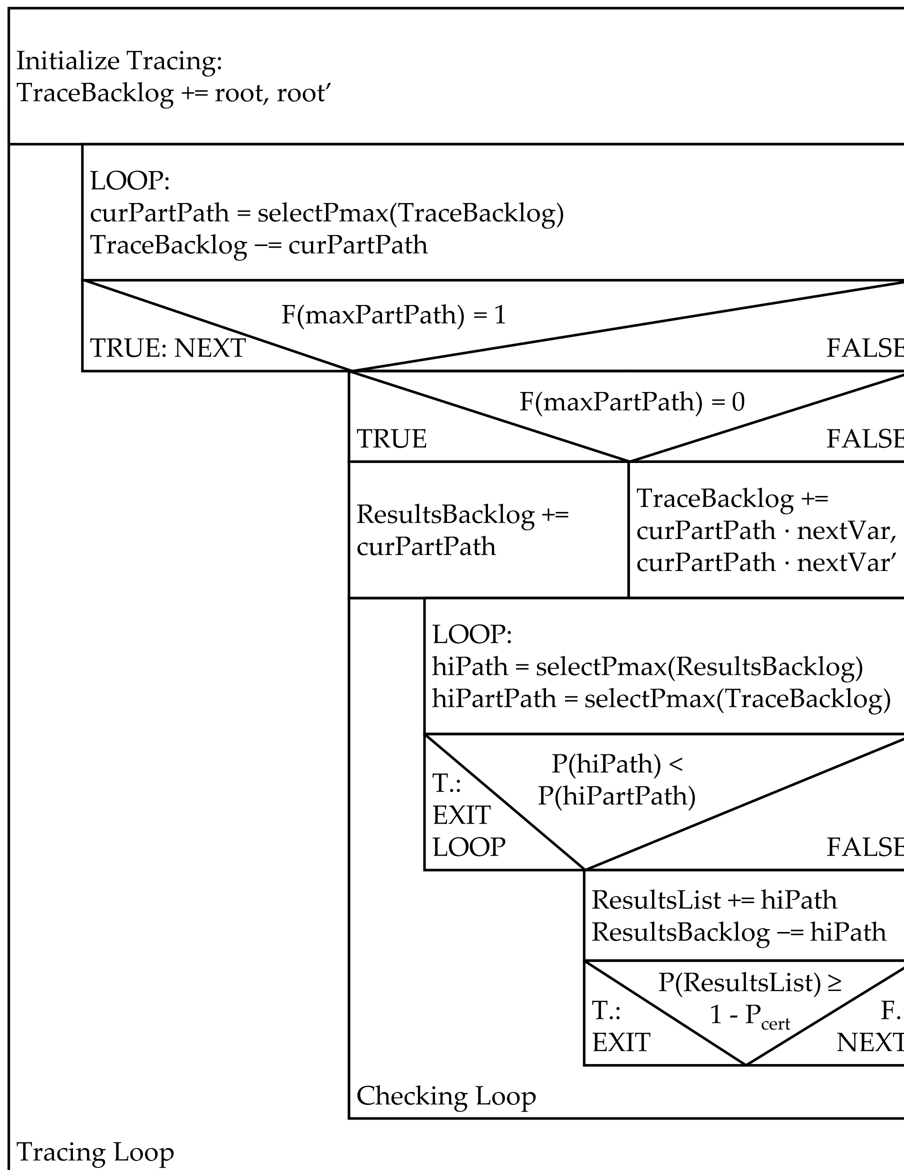

Figure 4.5: Nassi-Shneiderman diagram for the probability-guided evaluation process of a BDD

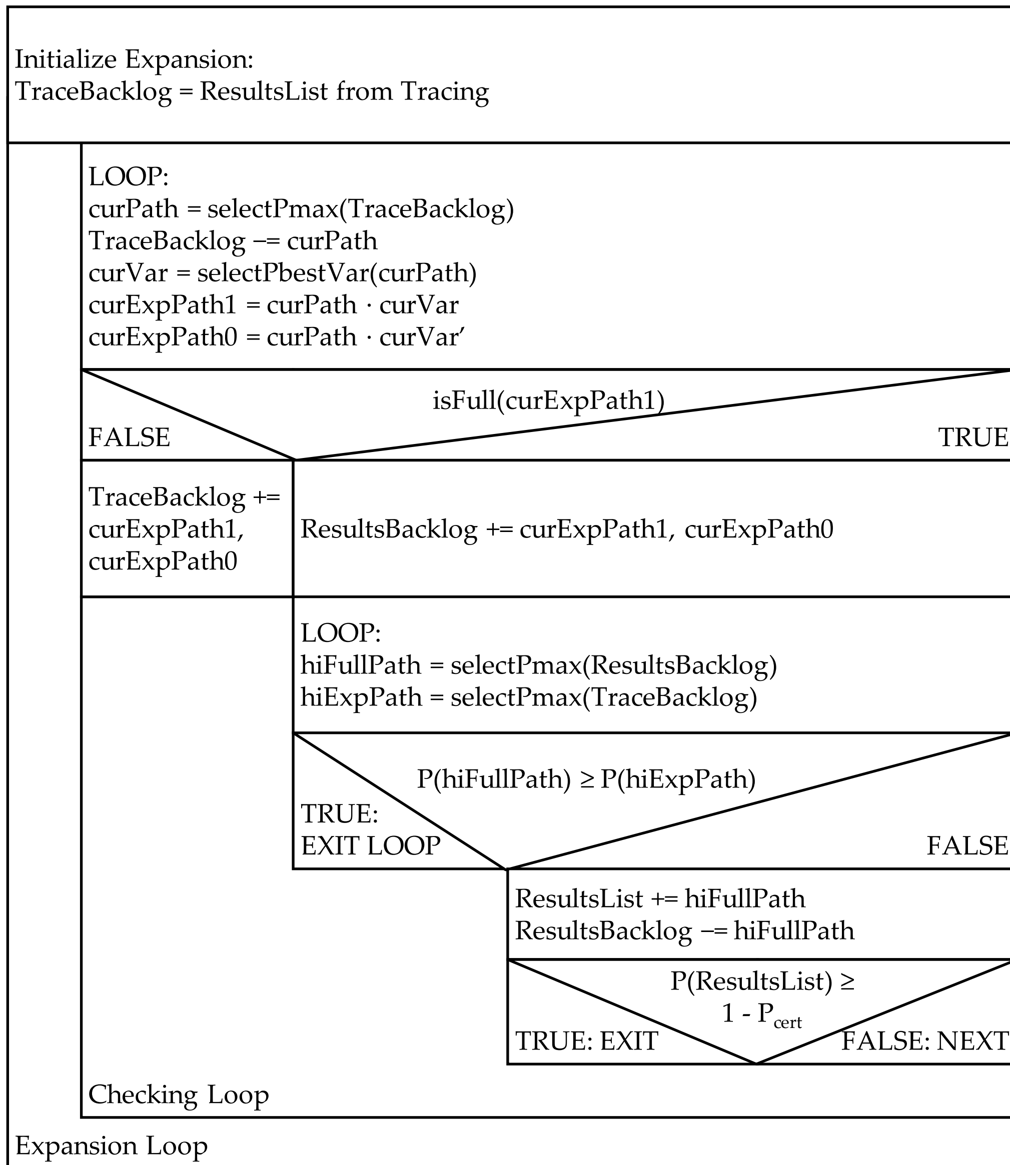

Figure 4.6: Nassi-Shneiderman diagram for the probability-guided path expansion process of the paths of a BDD

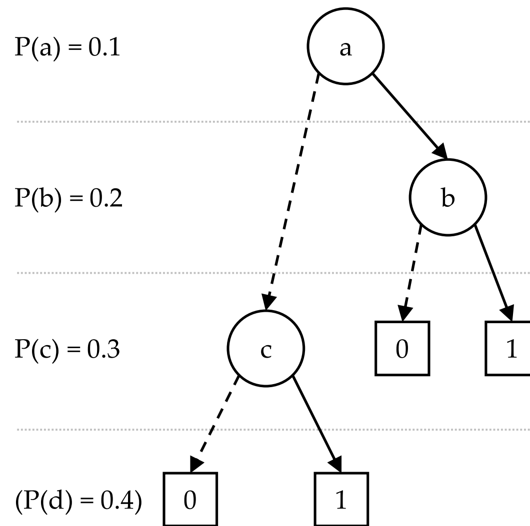

Figure 4.7: Reduced, probability-weighted BDD for $ab + a'c$

4.6 Example

As an example, consider a fault tree with the characteristic function $F(a, b, c) = ab + a'c$. In reference behavior, another basic event d is defined (but not captured in the fault tree). Thus the extended characteristic function is $\hat{F}(a, b, c, d) = ab + a'c$. The probabilities of the basic events are $P(a) = 0.1$, $P(b) = 0.2$, $P(c) = 0.3$, and $P(d) = 0.4$. Figure 4.7 shows the associated reduced, probability-weighted BDD.

A certification limit of $P_{cert} = 0.4$ is assumed. The probability of the top event to occur is $P(ab + a'c) = 0.29$. The proposed method is applied to find the minimal number of full assignments that need to be verified in order to validate the fault tree, in order of their probability, and using the given variable order a, b, c, d.

Table 4.1 displays the state of the tracing and results backlogs and executed actions per step during tracing. Table 4.2 gives the same information for the expansion part of the process.

4.7 Case study: Validation of the ARP 4761 wheel brake system example

In order to practically substantiate the claims of effectiveness of the presented algorithm made earlier in this chapter, it has been implemented. It has been applied to two examples for fault tree analysis given in ARP 4761 [28]. Results of measurements of its performance characteristics are discussed in this section, while the method's fitness for introduction into industrial practice is discussed in the last chapter of this work.

The implementation of the algorithm is written in Java [51]. It uses the JavaBDD package [52] with the pure Java BDD solver[8]. It extends the BDD iterator provided by

[8]The recommended, native C BDD solver BuDDy [53] was not used because of a bug with non-coherent, symmetric BDDs: Calculating all implicants of $x_1x_2' + x_2x_3' + x_3x_1'$ throws an exception.

Table 4.1: Fault Tree Pessimism Example: Part I — Tracing

Step	Action	Tracing/Results Backlog
Initialization	-/-	Tracing: $a'\,(P = 0.9)$ $a\,(P = 0.1)$ Results: $\emptyset$
1. Tracing	Partial path to trace: a' Subtree: $F(a') = c$ Next variable in subtree: c	Tracing: $a'c'\,(P = 0.63)$ $a'c\,(P = 0.27)$ $a\,(P = 0.1)$ Results: $\emptyset$
1.1 Checking	Results backlog is empty	
2. Tracing	Partial path to trace: $a'c'$ Subtree: $F(a'c') = 0$ $\rightarrow$ Path found	Tracing: $a'c\,(P = 0.27)$ $a\,(P = 0.1)$ Results: $a'c'\,(P = 0.63)$
2.1 Checking	$P(a'c') = 0.63 \geq P(a'c) = 0.27$ $\rightarrow$ $a'c'$ is the next result $P(a'c') = 0.63 \geq 1 - P_{cert} = 0.6$ $\rightarrow$ Tracing is finished!	Tracing: $a'c\,(P = 0.27)$ $a\,(P = 0.1)$ Results: $\emptyset$

Table 4.2: Fault Tree Pessimism Example: Part II — Expansion

Step	Action	Tracing/Results Backlog
Initialization	-/-	Tracing: $a'c'\,(P = 0.63)$ Results: $\emptyset$
1. Expansion	Path to expand: $a'c'$ Variable to expand: b $a'bc'$ is not a full assignment	Tracing: $a'b'c'\,(P = 0.504)$ $a'bc'\,(P = 0.126)$ Results: $\emptyset$
1.1 Checking	Results backlog is empty	
2. Expansion	Path to expand: $a'b'c'$ Variable to expand: d $\rightarrow$ Results found	Tracing: $a'bc'\,(P = 0.126)$ Results: $a'b'c'd'\,(P = 0.3024)$ $a'b'c'd\,(P = 0.2016)$
2.1 Checking	$P(a'b'c'd') \geq P(a'bc')$ $\rightarrow$ $a'b'c'd'$ is the next result $P(a'b'c'd') = 0.3024 < 1 - P_{cert}$ $\rightarrow$ Continue	Tracing: $a'bc'\,(P = 0.126)$ Results: $a'b'c'd\,(P = 0.2016)$
2.2 Checking	$P(a'b'c'd) \geq P(a'bc')$ $\rightarrow$ $a'b'c'd$ is the next result $P(a'b'c'd') + P(a'b'c'd) =$ $0.504 < 1 - P_{cert}$ $\rightarrow$ Continue	Tracing: $a'bc'\,(P = 0.126)$ Results: $\emptyset$
3. Expansion	Path to expand: $a'bc'$ Variable to expand: d $\rightarrow$ Results found	Tracing: $\emptyset$ Results: $a'bc'd'\,(P = 0.0756)$ $a'bc'd\,(P = 0.0504)$
3.1 Checking	Tracing backlog is empty $\rightarrow$ $a'bc'd'$ is the next result $P(a'b'c'd') + P(a'b'c'd) +$ $P(a'bc'd') = 0.5796$ $< 1 - P_{cert}$ $\rightarrow$ Continue	Tracing: $\emptyset$ Results: $a'bc'd\,(P = 0.0504)$
3.2 Checking	Tracing backlog is empty $\rightarrow$ $a'bc'd$ is the next result Both backlogs are empty $\rightarrow$ Expansion is finished!	Tracing: $\emptyset$ Results: $\emptyset$

Table 4.3: Case study results

Metric	Value for BSCUINADV	Value for LOSSALLWB
Total no. of nodes	33	29
No. of basic events	17	14
No. of house events	4	2
Proba. margin / top-event proba.	3.06	0.56
No. of neg. scenarios (no trunc.)	6160	1155
Trunc. no. of neg. scenarios	63	28
After doubling no. of basic events	363	58
Computation time for all scenarios	190 ms	145 ms
Computation time with truncation	ca. 8 ms	¡ 2 ms

JavaBDD to accept edge weights[9], representing the probabilities (and complementary probabilities) of each basic event. It encompasses all optimizations proposed in the previous section. Export of iteration results to MATLAB [5] for analysis and figures production has been carried out using JMatIO [54].

[28] gives a series of examples for fault tree analysis. Two of them have been chosen for benchmarking: The largest one, *BSCU Commands Braking in Absence of Brake Input and Causes Inadvertent Braking*, abbreviated BSCUINADV, and a second one, *Loss of All Wheel Braking*, abbreviated LOSSALLWB.The former is the largest tree by node count, and the latter has the smallest probability margin of the examples given in the standard.

The size of both fault trees makes them too large for evaluation of every possible assignment, but still small enough so that the list of assignments to evaluate can be generated in a reasonable amount of time. The size of real fault trees can, nevertheless, reach far beyond theirs.

But even for those two trees, the number of negative scenarios is in the order of thousands of scenarios[10]. Applying probability truncation on a probability-sorted iteration of negative scenario reduced the number of scenarios required to fit remaining scenarios into the probability margin to dozens of scenarios. The extreme effect of probability truncation on the number of iterated scenarios is displayed in Figure 4.8. As predicted, negative scenarios of high probability are few. Even for very low probability margins, the number of scenarios drops sharply with increasing probability margin. Similarly, calculation times drop to a small fraction, although not as sharply as scenario count. Only for the larger of the two fault trees can the calculation time be measured readily in the Java Virtual Machine for Windows. Measurements of the smaller one seem to be dominated by the computational effort for the execution overhead.

Considering only the basic events used by the fault tree, however, is an unjustified simplification. To determine the effect of adding basic events that should not trigger

[9]It enforces the restriction of only two different edge weights per index, thus it is not a general implementation of edge-weighted BDDs.

[10]The exact number depends on variable ordering.

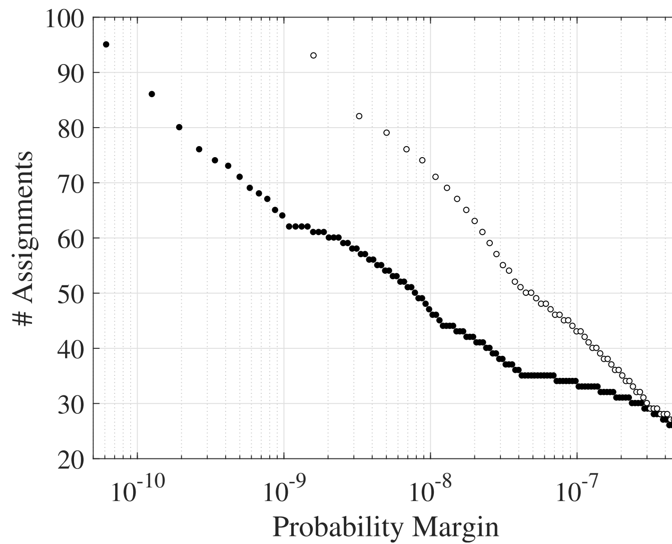

Figure 4.8: Effect of probability truncation: Minimal probability margins result in the truncation of the vast majority of negative scenarios; • for BSCUINADV (6160 full negative assignments in total), ◦ for LOSSALLWB (1155 negative assignments in total)

the top event, such events at random probabilities have been added. Their probability has been distributed lognormally around 10^-6 and censored to fall between 10^{-12} and 1 per flight hour. The experiment of adding such random basic events has been repeated 80 times for each number of basic events added, to ensure statistical robustness. The number of required negative scenarios was averaged over the set of outcomes for each number of events added. Figure 4.9 shows the results. While the number of possible, full assignments grows exponentially with base 2, the number of scenarios required for assessing fault tree completeness grows much slower. In the BSCUINADV-example, doubling the number of basic events introduces not $63 \cdot (2^{17} - 1)$ but only ca. 300 new negative scenarios that need to be considered. Combinatorial explosion is effectively curbed.

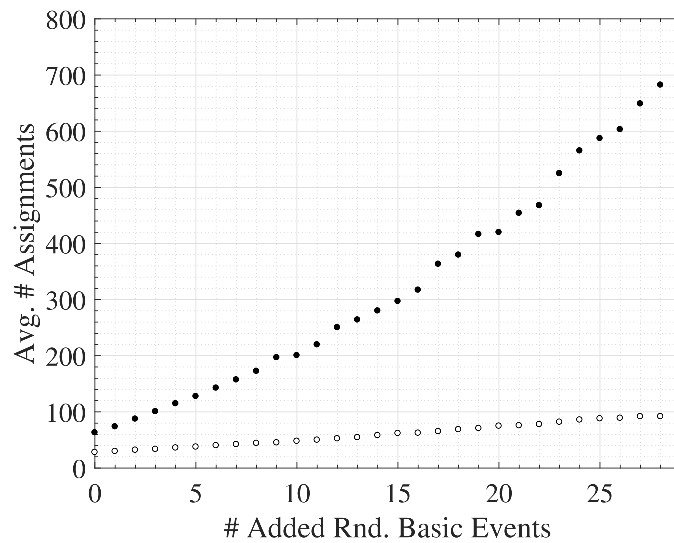

Figure 4.9: Effect of added events: Additional unconsidered basic events do not lead to combinatorial explosion, as both series are far below $2^{|\hat{y}_0|}$; $\bullet$ for BSCUINADV, $\circ$ for LOSSALLWB

Chapter 5

Fault Tree Generation

In system architecture design, fault tree generation and the design of the system under study are refined in an iterative process: The fault trees for all Failure Conditions (FCs) of the system for the current state of its architecture is generated, safety design intent is derived and the system architecture is refined, beginning the next iteration. This process continues until the system architecture fulfills top-level safety requirements. For the applicable standards in aerospace engineering, see [35, 28]. This section proposes a method for

1. covering system architecture by building logic-neutral fault tree templates from component-based, hierarchical, Differential-Algebraic Equation System (DAE)-based models,

2. covering safety design intent by manual annotation to data flow graph or DAE-based models, and

3. deriving static failure logic from safety design intent and applying it to fault tree templates, thereby transforming templates into fault trees.

The result is a fault tree that reflects system architecture and safety design intent in reference to the underlying failure condition as annotated in the model.

This method for model-based, automated fault tree generation is the first to produce a fault tree that suffices the requirements towards fault tree structure stated in ARP 4761 [28]. The annotation of behavioral model from which the logic of the fault tree is derived is a generalization of the work of Fenelon and McDermid [55, 56] and Papadopoulos [57] to omnidirectional models such as DAE-based models.

5.1 Capturing safety design intent

In Chapter 3, operational semantics for hybrid failure simulation are introduced. They are capable of describing failure behavior, in the same level of detail as nominal behavior, as DAE-based, component-oriented models. During system architecture design, the desired behavior of the system under component fault(-s) is described in a more abstract way:

- fail-safe or fail-passive designs prevent specific safety-relevant repercussions[1] without preserving related nominal behavior

- fail-ops designs prevent specific safety-relevant repercussions and at the same time preserve nominal behavior

Fenelon and McDermid [55] have shown that both concepts, in a component-oriented approach, relate to limiting failure propagation and transforming severe failures to negligible ones. *Safety design intent* is the definition of the desired, component-level behavior abstracted to rules for fault propagation and transformation. This section describes how safety design intent can be annotated into DAE-based models and how the logic to be encoded in a fault tree can be derived from it. This bypasses the hybrid systems verification problem (see Section 2.2.5) by manual annotation.

5.1.1 Concept

Assuming the designed functionality of the system under study is safe, there needs to be some reason or cause for it to become unsafe. These causes may be outside of the system, when environmental conditions (in the broadest sense) exceed what the system was designed for, or inside of it, when components of the system fail. Component faults themselves cannot be safety-critical, as only harm to humans, the environment of the system or the property of third parties is by definition a relevant repercussion in safety assessment. Safety design targets the interrelation between cause (component fault) and effect (repercussion): It requires the behavioral change to the system (i.e. the failure) to be transformed or blocked from propagating to the system portion where the actual harm is done. Shielding portions of a system from faults that occurred elsewhere is achieved through components that only allow omnidirectional flow of energy, mass or information, such as fuses, clutches, diodes, relief valves etc. Failures can be transformed intentionally by introducing redundancy or monitoring components that passivate system portions upon detecting their failure.

For directional behavioral models, Failure Propagation and Transformation Notation (FPTN) [55, 56] and, more powerful, Hierarchically Performed Hazard Origin & Propagation Studies (HiP-HOPS) [57] have proposed methods for specifying safety design intent. However, it is in the nature of a failure not to necessarily propagate in the direction of energy, mass or information flow during nominal system behavior, but in any way allowed by the physical principles of behavior of the real components of the system. The method proposed in this work builds upon FPTN and HiP-HOPS and can be interpreted as a generalization of HiP-HOPS to DAE-based models. This allows for simpler, more reusable, extended models of nominal and failure behavior that include safety design intent.

Conceptually, the proposed method propagates and transforms qualitative failure *tags* across directional graphs of component-oriented, hierarchical model topology. The

[1] In practice, prevention of repercussions usually is probabilistic, not definitive, e.g. an aircraft's tail-hook for emergency landing may fail to drop or catch the arresting wires, thus its prevention of runway overshoot is probabilistic.

nodes of these graphs represent component interfaces, called *ports*. Their edges represent both component-internal failure propagation and transformation as well as inter-component failure propagation. So conceptually, failures only propagate between components — Transformation only happens in the intra-component edges. The graphs are directional but can encode omnidirectional edges by having two edges between two nodes, one in each direction. In this work, they are called *topological graphs*. The topological graphs are built from directional, dataflow-based models in Simulink and omnidirectional, DAE-based models in Simscape, or mixtures of the two. The formal definition this transformation is given in Subsection 5.1.2.

Three tags are defined:

1. *Neutral behavior* — **N**: No function is provided by this node. This is the default tag and conceptually represents the loss of the function in safety assessment.

2. *Nominal (specification-conformant) behavior* — **C**: The function is provided in conformance to the desired behavior of this node. This represents nominal behavior.

3. *Terminating behavior* — **T**: Erroneous / inadvertent execution of behavioral specification is provided by this node.

Nodes can have unconstrained tag assignment or be assigned a fixed tag. Tag propagation on inter-component edges follows an ordinal relation, called *tag precedence*:

$$\mathbb{T} = \{\mathbf{T}, \mathbf{C}, \mathbf{N}\} \tag{5.1}$$

$$\mathbf{T} > \mathbf{C} > \mathbf{N} \tag{5.2}$$

The destination of an inter-component edge cannot have a lesser tag than its origin, unless the destination has a fixed tag assignment. The relation between intra-component tag assignments follows a custom rule for each component. The nominal behavior and the faults of a component each are encoded as a rule for tag assignment.

A tag assignment *solution* for each node in a given graph is one that satisfies all constraints from the inter-component propagation rule, intra-component assignment rules and fixed tag assignments.

FCs are translated into predicates on solutions. When a solution fulfills the predicate, the FC occurs. By applying the intra-component assignment rules corresponding to the active faults of a given scenario (see Section 1.4.4), positive scenarios can be evaluated with reference behavior modelled according to this concept.

5.1.2 Formal definitions

Model topology of Simulink and Simscape models enforces identity of state on connected ports, and component behavior specifies transformation, or more generally, interrelation between variables. Hierarchicality is captured by implicitly requiring identity for the purpose of simulation between each subsystem's lower-level, 'internal' ports and its higher-level, 'external' ports. Table 5.1 defines annotation features for such elements of Simulink and Simscape models for safety design intent.

Table 5.1: Notation of safety design intent annotation

Feature	Symbol & Definition
Adjacency matrix of model with n ports	$\mathbf{A} \in \mathbb{B}^n$
Assigned tag of port p in block b	$\sigma_{b,p} \in \mathbb{T}$
Allowed tag assignments of block b with k ports	$\mu_{1,b} : \mathbb{T}^k \to \mathbb{B}$
Allowed tag assignments for propagation from port p in block b to port q in block c	$\mu_{3,b,p,c,q} = \begin{cases} \sigma_{b,p} = \sigma_{c,q} & \text{if} \mathbf{A}_{p,q} = 1 \\ \text{unconstrained} & \text{otherwise} \end{cases}$
Failure Condition f triggered by m tag assignments	$\mu_{4,f} : \mathbb{T}^m \to \mathbb{B}$

The transformation from flat, global models to topology graphs is formalized in the adjacency matrix: When there is a connection from port p to another port q in the model, then there also is an edge from node p to node q in the graph, and the entry in row p in column q in the adjacency matrix $\mathbf{A}_{p,p}$ is 1 — otherwise, it is 0. Port type (inbound, outbound, omnidirectional) and thus connection direction can be ignored here, thus the adjacency matrix is symmetric. Per convention, the diagonal is 0^n. Because tag transformation across intra-component edges is constrained by custom rules anyway, the adjacency matrix entries there are irrelevant for the solution. For simplification in later Constraint Satisfaction Problem (CSP) creation, these edges are assigned 0.

In order to relieve the user from having to enter tag transformation rules $\mu_{1,b}$ for nominal behavior, a default tag transformation rule for intra-component edges is defined: Each inbound and each omnidirectional port inside the component are connected to each outbound and each omnidirectional port by the following tag assignment rule: Tag assignment to each outbound or omnidirectional port is the highest tag (as per eqn. 5.1) of all connected inbound or omnidirectional ports.

Tag assignment can be fixed, i.e. the assigned variable has a value chosen by the modeler, irrespectively of the assignment of tags to connected ports. This has two applications: Usually, logical signal sources such as sensors in nominal behavior have a fixed assignment of **C**, which is then propagated to other components. Conversely in the failure case, fixed assignments of **N** or **T** have to be made.

In this abstraction of failure behavior, propagation, and transformation, FCs are formalized as predicates μ_4 on tag assignments on a subset of the ports in the model, i.e. in Table 5.1, $n >= m$.

Note that safety design intent is static: No time domain is involved. This is a strong abstraction of real system behavior, yet less abstract than that in FPTN [55, 56] and HiP-HOPS [57], because it does not impose static directionality on the exchange of tags.

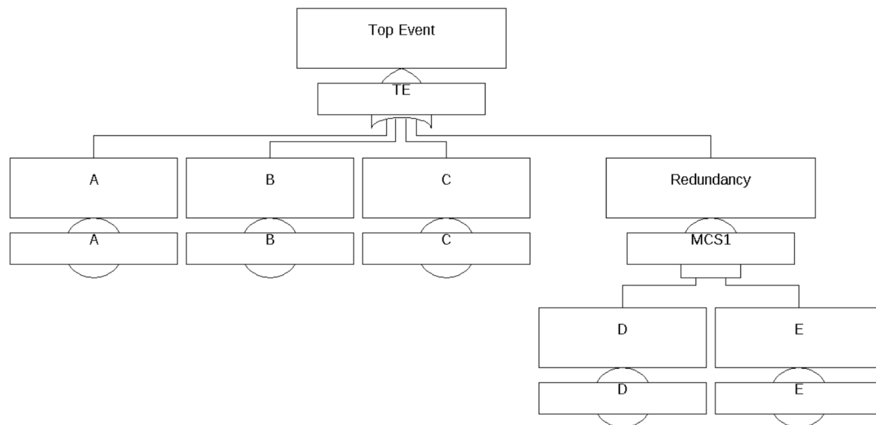

Figure 5.1: Example for a simple, flat fault tree

5.2 Fault tree template generation and logic supplementation

Safety design intent contains the logic of the fault tree to be created, while model hierarchy encodes fault tree structure. These two aspects can be handled with a certain degree of independence because fault trees can express equivalent logic in different ways, just as Boolean expressions can encode the same function in different ways. The mathematical concepts this idea is founded upon is introduced in this section, and it is applied for refining (or 'trimming') fault tree templates in a fault tree generation process according to Section 1.4.4.

5.2.1 Structural classification of fault trees

For automated fault tree generation, structural properties of fault trees are of great interest. They can be used to describe what makes an automatically generated fault tree intelligible and useful[2] and help to structure the process of fault tree generation that is the subject of automation in this chapter.

In the context of this work, three structural types of fault trees are relevant:

- *Flat* fault trees have an OR-gate as their root node, and arrange all Minimal Cut Sets (MCSs) directly below it (see Figure 5.1). First order MCSs are arranged directly below the root node, while higher-order MCSs are each grouped by an AND-gate directly below the root node. This corresponds to the Disjunctive Normal Form (DNF) of symbolic Boolean expressions. Flat fault trees carry no information on model hierarchy or fault propagation.

- *Deep* fault trees contain additional gates in comparison to flat fault trees. They can carry additional information (beyond the MCSs) on the system under study.

[2] In the aviation industry, the standard and guidance material for safety assessment [28, 37] require specific types of fault tree structure, introduced later in this section as the 'hierarchically-ordered' type.

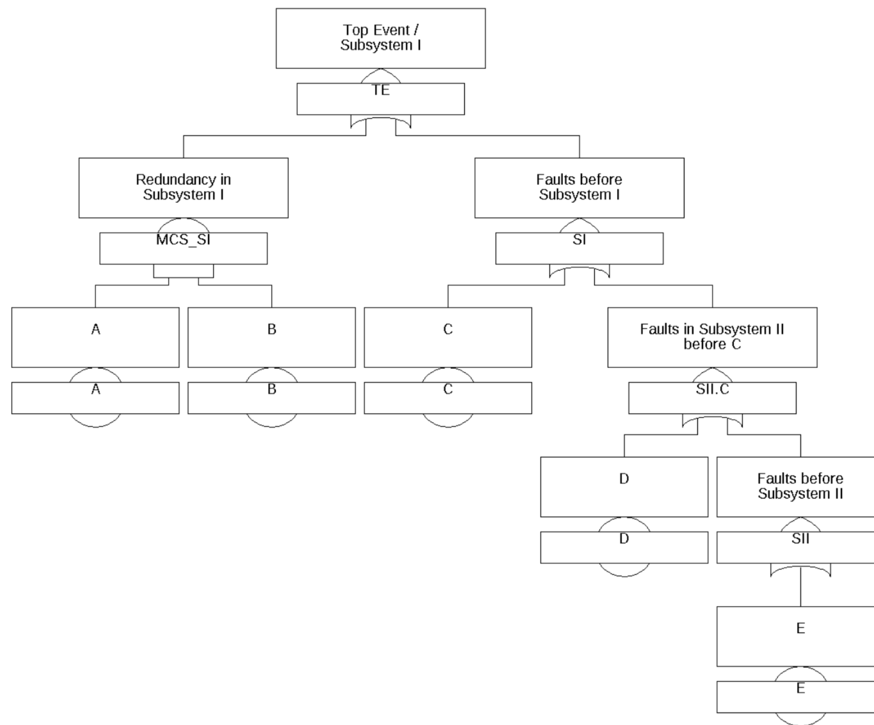

Figure 5.2: Example for a simple, fault propagation depth-ordered fault tree, assuming the subsystems' order in the functional chain is III → II → I, and the faults in subsystem II are on sequential components

Flatness and deepness are mutually exclusive. Sub-trees of deep fault trees can be further classified as follows:

- *Fault propagation depth-ordered* fault trees encode failure propagation in their additional gates (see Figure 5.2). The root node represents the component where the FC manifests. A sequence of OR-gates stretches from the root node, where each level indicates that the faults directly below it are one component farther away from the component where the FC manifests. Maintaining this structure can be difficult for elaborate safety architectures, where the components providing a redundancy may be distributed across different parts of the system. Furthermore, this approach makes re-using sub-trees between FCs in fault tree generation difficult because fault propagation paths can differ from FC to FC. Lastly, this also means that divide-and-conquer concepts depending on model hierarchicality do not apply directly to such trees. HiP-HOPS creates fault trees of this type, even though the method takes model hierarchicality into account during model annotation.
- *Model hierarchy-ordered* fault trees have a scaffolding of OR-gates which portray the composite hierarchy of the model (see Figure 5.3). For hierarchical DAE-based and dataflow-based models such as Simulink and Simscape models, this is the same hierarchy as that of system architecture. Faults are arranged below their governing system portion, and MCS spanning multi-

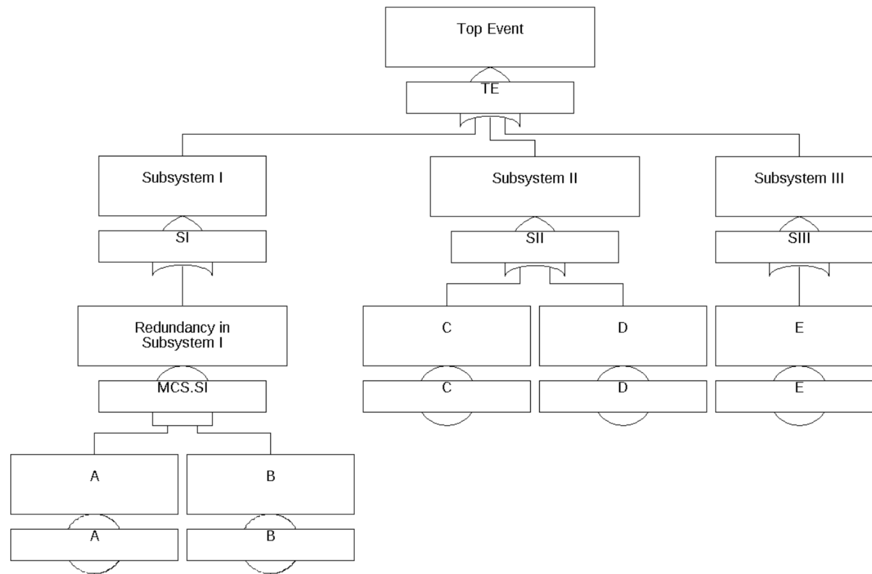

Figure 5.3: Example for a simple, model hierarchy-ordered fault tree

ple such portions are arranged directly below their deepest common ancestor. Minimal cut set can be read easily from such a fault tree, and the fitness for reuse of a subtree only depends on the failure behavior context of its top node. The divide-and-conquer principle for fault tree analysis enabled by such a structure is especially helpful when multiple design and safety engineers collaborate in one fault tree, because responsibility for system portions maps to responsibility for sub-trees.

Manually-built fault trees tend to be a mixture of both forms of deep fault trees. The deductive nature of fault tree analysis applies more gracefully to model hierarchy-ordered fault trees. Most importantly, the standard and guidance material for fault tree generation in the aviation industry (see [28] and [37]), where fault tree analysis has been routinely and successfully applied for safety assessment of large, complex systems for decades, requires system hierarchy-ordered fault trees. This work assumes that model hierarchy is identical to system hierarchy, which makes the fault trees produced by the presented method standard-conformant in the aviation industry.

5.2.2 Fault tree templates and structure-neutral modification

Fault tree templates encode model structure that should be conserved in the final fault tree. Thus, all information required for fault tree generation other than (FC-specific) system behavior is already captured in the template, and the following steps of the analysis then code abstractions of system behavior into the template. This section describes fault tree templates for encoding model structure, methods for encoding any logic into them without changing the part of the fault tree that describes model structure. This section does not cover how to determine the logic that is to be encoded, but instead leave that topic for later sections of this chapter.

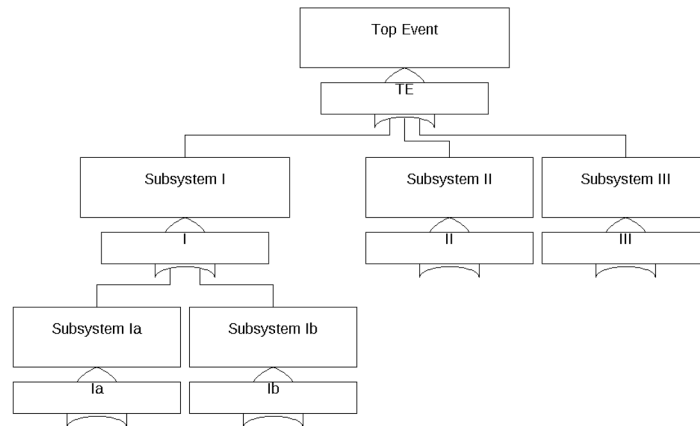

Figure 5.4: Example for a model hierarchy fault tree template

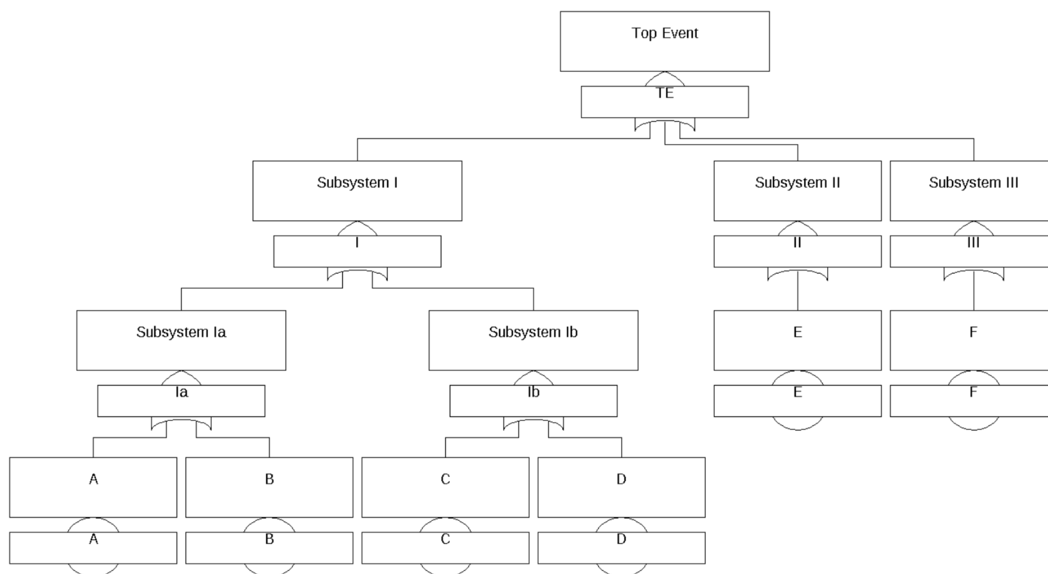

Figure 5.5: Example for a pessimistic fault tree template

The scaffolding of OR-gates of model hierarchy-ordered fault trees corresponding to model components is the same for all fault trees of that model. Thus it can be created once, and then reused in the fault tree generation for each FC. It is called the *model hierarchy fault tree template* and an example is given in Figure 5.4. Also shared by all fault trees of a given model is the domain of faults of each component and relevant external conditions. The most pessimistic fault tree possible for that model is the one in which each of these faults triggers the top event individually, i.e. there are no AND-gates in the tree and all faults are ORed. When each component's faults are ORed under the component's OR-gate in the model hierarchy fault tree template, the tree is called the pessimistic fault tree template for that model. An example is given in Figure 5.5.

The former template type contains no basic events and, thus, no logic. Positive scenarios need to be inserted into it. The latter specifies that all but nominal behavior

triggers the top event. Negative scenarios need to be removed from its logic. The following subsections describe the methods for their refinement into a FC-specific fault tree in detail. Specifically, section 5.2.2 gives a method for adding individual MCS to a fault tree template in a way that there is a formal relationship between the MCS and its position in the tree, which yields canonical fault trees. Section 5.2.2 transfers the concept for fault tree validation from Chapter 4 to fault tree generation, which corresponds to the left side of Figure 1.1 and uses the former method only for fault tree modification.

Supplementing MCSs for model hierarchy fault tree templates

This method requires that MCSs have been created from a set of positive scenarios which are either true positives or false positives (pessimistic approximation).

MCSs are encoded as AND-gates with each of its faults as children, and each MCS is inserted into the tree by adding the AND-gate below the deepest common ancestor OR-gate.

As an example, consider the fault tree template in Figure 5.4, and assume that the characteristic function of the fault tree that is to be created is $F = AC + BD$, then the steps of this method would be to first add AC and then BD. Both MCS have to be added below the gate for Subsystem I, because the deepest common ancestor of A and C as well as B and D is Subsystem I. This is illustrated in Figure 5.6, where the intermediate fault tree with the MCS AC and the final fault tree with both MCSs are shown.

Before all of the MCSs have been inserted into the fault tree, it is not ensured to be pessimistic, because an MCS containing a true positive may still be missing. Therefore, only the final fault tree built with this method is valid.

Redundancy integration for pessimistic fault tree templates

Negative scenarios, i.e. conjunctions of basic events that do not trigger the top event of the fault tree, are iteratively removed from the logic of the fault tree. This method does not require MCSs to be determined before beginning modification of the fault tree, and intermediate results are pessimistic approximations. Binary Decision Diagrams (BDDs) can be employed for the required logical computations and, based on section 5.2.2, the characteristic function represented by the fault tree and by the BDD can be kept equivalent at each iteration.

Each MCS in a fault tree corresponds to a path to the 1-node in the BDD. When a particular 1-node is found to be a false positive, i.e. the corresponding MCS is revealed not to cause the top event, it can be removed from the BDD and the fault tree. At the same time, there may be higher-order cut sets that include the tested MCS which do cause the top event. Whenever evaluation cannot rule that out, the higher-order cut sets are added to the fault tree at this step. The process of removing low-order MCSs corresponding to false-positives and adding higher-order MCSs that include their original low-order MCS is called 'fault tree trimming' (see also Figure 1.1).

Formally, fault tree trimming of a scenario with the path $\mathbf{y}$ is carried out by applying the following operations on the BDD and the fault tree:

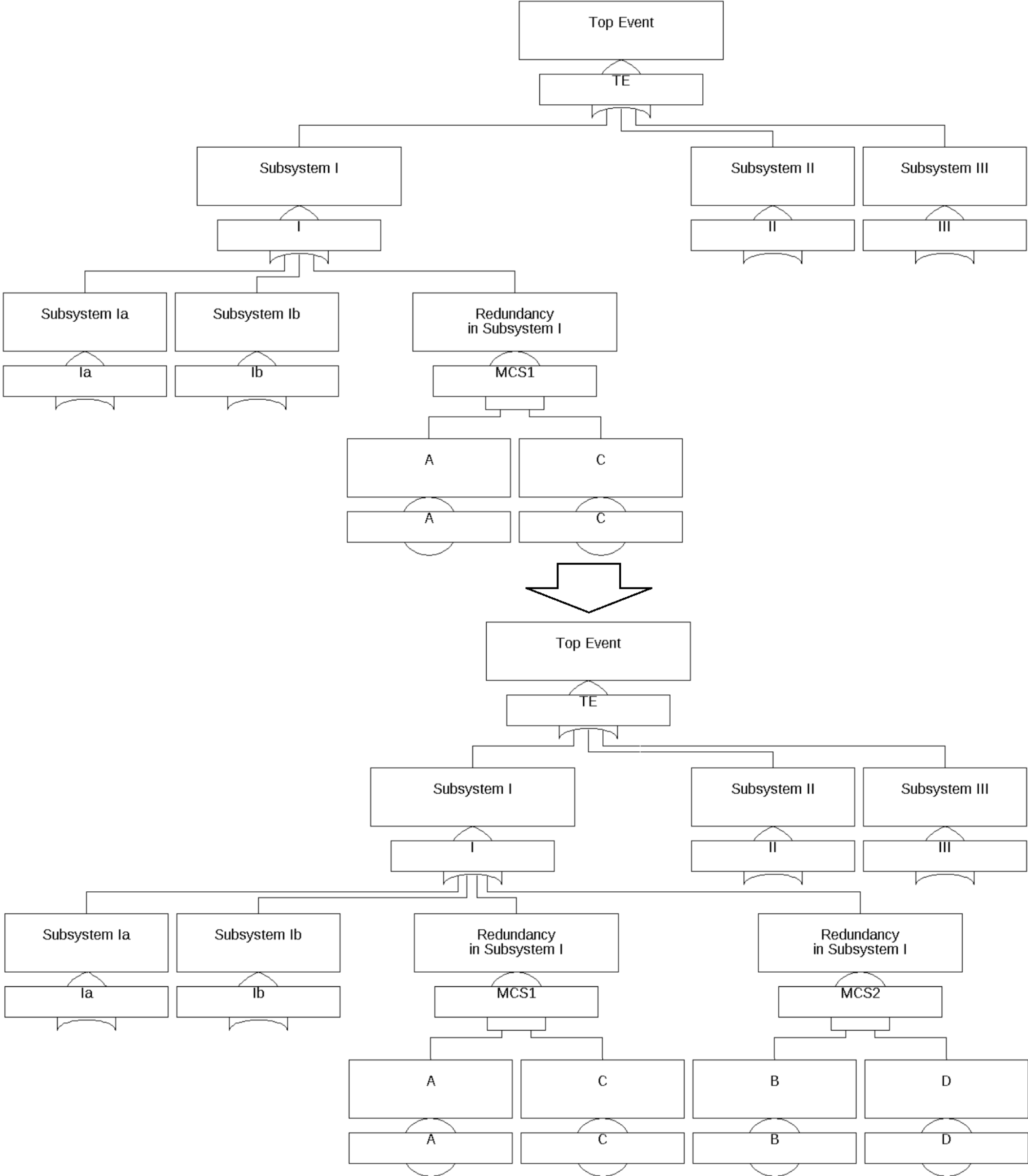

Figure 5.6: Example for the steps of supplementing MCSs

1. Evaluate the path $\mathbf{y}$ to the 1 node in the BDD against reference behavior.
   - Positive evaluation result: The path truly triggers the top event, skip all following steps because no changes to the fault tree or the BDD are required.
   - Negative evaluation result: The path does not trigger the top event, and adding more basic events to it will not change that. Replace the final edge of the path with an edge to the 0-node. In BDD operations, this is achieved by computing the relative complement of the BDD and the path.
   - Indeterminate result: The path does not trigger the top event, but some of the paths in its sub-tree do and some do not. Replace the final edge of the path with its cube, because it is the pessimistic approximation of the true logic of its sub-tree[3].
2. Remove the MCSs whose corresponding paths to the 1-node no longer exist from the fault tree.
3. Add a new MCSs to the fault tree for each new path to the 1-node in the BDD.

Steps 2 and 3 build on a formal relation between 1-nodes of the BDD and the AND-gates and basic events of the fault tree. Because of the recursive nature of the method, both steps have to use the same relation. Both BDDs and fault trees are directed, acyclic graphs, so nodes are unambiguously identified by their path to the root node. The paths to 1-nodes in the BDD are MCSs[4]. For the relation between MCSs and fault tree nodes, additional definitions are needed. We define that each basic event has a *natural path*, which is the path of its corresponding component in the model hierarchy fault tree template. The set intersection of the natural paths of each basic event in a MCS describes the node representing their deepest common ancestor component in model hierarchy. For first-order MCS, the basic event's natural path is its representation in the fault tree. For higher-order MCSs, and AND-gate is added between all basic events' deepest common ancestor's node, which always is an OR-gate in the model hierarchy fault tree template, and each basic event. This is the same relation between an MCS' basic events and their representation in the fault tree as the one imposed generatively in section 5.2.2.

During the intermediate iterations of fault tree trimming, BDD identical subtree merging keeps the size of the BDD within reasonable bounds. The relation between paths to the 1-node and MCSs can be used to keep coherent fault trees small as well: Absorption can be carried out in the BDD by applying the method for discovering prime implicants in BDDs proposed by Coudert and Madre [38] or refined versions of their method. Nevertheless, in practice it is advisable to do logical calculations on the BDD only and, once the final set of MCSs has been determined, carry out the steps for fault tree modification only on the final BDD. It is also advisable to limit cut set order

[3]This is only correct for coherent fault trees. For non-coherent fault trees, replace the 1-node at $\mathbf{y}$ with a subtree of a single level with both nodes 1, or with 0 when there is no higher node index. In this case, the fault tree is only updated whenever a positive or negative evaluation result occurs.

[4]In coherent fault trees, the 0-edges can be omitted from each MCS' path, because they correspond to complemented basic events.

by replacing paths that would exceed the limit with 0. Similarly to the method for fault tree validation in Chapter 4, probability truncation can be applied here as well: Once the probability of the probability-weighted BDD falls below the certification limit, the intermediate fault tree is a valid fault tree from the perspective of fault tree pessimism. The final fault tree's MCSs will be included in the MCSs of the intermediate one, and the probability of the intermediate fault tree will be greater than or equal to that of the final fault tree.

5.3 Constraint Satisfaction Problem formulation and solution aspects

The previous section covered how static failure logic can be encoded into fault tree templates as introduced in Subsection 5.2 to create fault trees. This section explains how Constraint Satisfaction Problem (CSP) solution theory can be employed in the evaluation step in Figure 1.1, for determining whether a combination of active basic events causes the FC under analysis, given the annotated safety design intent.

5.3.1 Constraint Satisfaction Problem assembly

The formalization of safety design intent annotation from Subsection 5.1.2 lends itself to being encoded as a CSP, with each variable representing a tag assignment of a port $\sigma_{b,p} \in \{\mathbb{T} = (\mathbf{N}, \mathbf{C}, \mathbf{T})$. The 'input' to the CSP is the choice of active intra-component tag transformation rules representing nominal behavior and faults for each block, and its 'output' is whether the FC under analysis is met.

The CSP is built from model topology and tag transformation rules for each of its blocks. The following steps are carried out during CSP assembly:

1. Model topology is copied to allow simplifications without affecting the original model.

2. The copy of model topology is flattened, i.e. hierarchical ordering of blocks in subsystems is reduced to a flat network of connected ports.

3. Ports of blocks that do not specify any behavior are removed (e.g. signal terminators, Simscape solver configuration blocks).

4. The adjacency matrix of inter-component edges $\mathbf{A}$ is translated into pairwise equality constraints on connected ports.

5. At least one intra-component tag transformation rule μ_3 per block is applied as a table constraint on the variables representing the block's ports.

6. A boolean reification variable is created for the FC(s) and associated with its triggering tag assignment μ_4.

7. For performance, a boolean reification variable is created for each tag transformation rule and reified with its fulfillment. A constraint that requires at least one of these variables to be 1 for each block is created, so that at least one tag transformation rule is fulfilled. When a set of tag transformation rules for block behavior is to be evaluated, the corresponding reification variables for these rules are constrained to be 1 with equality constraints. Only failure behavior is enforced this way, so that failure propagation is not hindered artificially. See also section 5.3.2.

5.3.2 Contradiction-free, efficient problem formulation

For practical model sizes, it is imperative that the created CSP covers all possible combinations of active basic events. If a CSP were to be created for a single combination, combinatorial explosion would strike: The complete computational effort for creating an appropriate CSP for the current combination of active basic events would need to be spent at each iteration of fault tree refinement. Instead, the alternative component behavior rules for each block $\mu_{1,b}$ are all encoded into one CSP, but only the desired component behavior rule for the current combination of active basic events is required to be fulfilled by the solution by reification. The inactive behavior rules are 'switched off'. As an additional safeguard, the prototypical implementation provided with this work contains an additional constraint on the switching variables that enforces at least one of them[5] to be active[6].

In practical models, not all variables have only one valid tag assignment due to information loss over components. This issue is specific to omnidirectional tag propagation or transformation models, because cause and consequence is determined by signal flow. In such methods, such as FPTN and HiP-HOPS, the cause of one fault can overrule another component's nominal behavior by being 'downstream' in signal flow. Omnidirectional models do not have the semantical element of signal flow direction, so the ambiguity of tag assignments irrelevant to the evaluation result of the FC needs to be resolved in some way. As an example, consider the overload clutch and the superposition gear in the case study. In nominal behavior, assignments of **N** to ports between these components would have no effect on the tag assignment at the hinge. The most plausible tag assignment however is **C** to both these ports. Failure, such as loss of function (**N**) or erroneous function (**T**), needs a reason to occur. This is built into the solution process by choosing the solution for which the least assignments of **N** and **T** need to be made. However, this is debatable, because modelling errors resulting in under-constrained portions of the model do not become apparent in the final tag assignment, which makes debugging modelling errors more difficult.

[5]More than one behavior can be active because physically different faults may correlate to the same port assignments.

[6]The same technique can be applied to the analysis of multiple FCs using one CSP, but because FCs usually are few and their combinations are not subject to analysis, this has little impact on computational effort.

5.3.3 Binary Decision Diagrams and fault tree refinement

The process of fault tree refinement begins with the pessimistic fault tree template and its corresponding BDD. Positive scenario after positive scenario is evaluated as a CSP. If a positive scenario is revealed to be a false positive, it is trimmed from the BDD. If the scenario is indeterminate, the scenario's node in the BDD is expanded. Trimming a scenario from a fault tree and its corresponding BDD is described in section 5.2.2.

As an example, the refinement steps for producing the fault tree in the case study in section 5.5 are as follows. With the basic events 'AFCS erroneous' abbreviated as A, 'overload clutch stuck closed' B and 'hinge jammed' C, ordered A, B, C, and the common assumption that a coherent fault tree is to be built, Table 5.2 shows the executed steps. To illustrate indeterminate evaluation results, a Venn-diagram shows assignments where the top event is triggered in reference behavior hatched, and where the current BDD evaluates to 1 in gray. As proposed in section 5.2.2, the fault tree update steps are only executed once the BDD is final. Once all paths have been evaluated, the MCSs can be read or 'collected' from the BDD and the fault tree template modification steps can be executed in bulk.

5.4 Implementation aspects

The proposed method has been implemented a pipeline of open source and commercial tools:

- *Modeling environment*: Simscape and Simulink with custom extension: The extension was implemented in MATLAB [5] under the name `SafetyLink` and uses various components from the Java [51] package `org.willea.ftms` [7,8,9], which has been created by the author of this thesis as well.

- *Topological Simulink/Simscape model analysis*: `TopologicalAnalysis`: AC custom package written in MATLAB[10]

[7] In order to use Java 8 components in MATLAB 2017a, which natively only supports Java 7, the path to the Java Runtime Environment needs to be specified in the `MATLAB_JAVA` environment variable. This has side effects: The desktop configuration cannot be persisted due to a deprecated version of Saxon being used by MATLAB (see also [58]) and license re-validation ceases to work due to an exploit of a Java 7 security leak by MATLAB that has been fixed in Java 8 (see also [59], problem 5). These issues have been reported to Dr. Kluge by The MathWorks at the European MATLAB Advisory Board 2016 and the MATLAB Service Team in an improvement suggestion. Migration to a newer Java version than 7 (i.e. 8 or 9) is planned for a near MATLAB release [60].

[8] In order to debug Java components called from MATLAB, MATLAB needs to be started with the `-jdb` flag [61] and the MATLAB session needs to be shared, i.e. allow an external debugger to browse symbols, memory content and debug output, as described in [62].

[9] For using a package such is the one provided with this work, the JAR-archive needs to be added to the Java class path. See [63] for a step-by-step manual.

[10] There is a known issue with the `TopologicalAnalysis` package: It skips blocks that are not connected to any other blocks. The `ExploreBlock` function follows connections instead of model hierarchy. It is arguable whether disconnected blocks are relevant for behavioral analysis at all, thus this issue was left open.

Table 5.2: Example for fault tree refinement

Path	Action	BDD	Venn-diagram
—	Initialization: cube(∅)		
A	Evaluate: Indeterminate. Replace with cube(A).		
$A'B$	Evaluate: Indeterminate. Replace with cube($A'B$).		
All remaining paths	Evaluate: Positive. No change.	—	—
—	Collect MCSs: AB, C		—

- *Fault tree generation*: `FaultTreeFactory`, a custom component written in Java [51] in `org.willea.ftms`
- *CSP-engine*: Choco [64]
- *BDD-engine*: JavaBDD [52]
- *Communication from Java components to MATLAB*: matlabcontrol[11] [65] as wrapped by MatConsoleCtl [66]
- *Fault tree renderer*: `FaultTreeView`, a custom component written in Java [51] in `org.willea.ftms.gui`

This pipeline is maximizes synergy with the two implementations for the other methods of this work (see Chapters 3 and 4).

The modeling environment consists of a set of Graphical User Interfaces (GUIs) for specifying user input for the proposed automation. Access to them is provided through the mask of the Safety Assessment Data' block in the `SafetyLink` Simulink block library. It stores the wrapper methods in MATLAB for calling the custom Java-based GUI which also manage persistence of the information with the Simulink or Simscape model or block libraries[12]. Therefore, the Simulink library link of 'Safety Assessment Data' block needs to be disabled, so model-specific settings can be stored[13]. The Java GUI components are all located in `org.willea.ftms.gui` and each serve a specific purpose:

- *Failure conditions* can be specified in the MATLAB-component `guiFailureConditions.m` in the `SafetyLink` package, which calls and the Java-component `FailureConditionBrowser` for providing its frontend.
- *Library* and *model block safety design intent* can be specified in a GUI provided by `guiBlockBehaviors.m` in MATLAB. It calls the Java-component `BehaviorManagement` for drawing its frontend and passing events back to it. The GUI does not support the specification of tag transformation rules. Transformation rules need to be specified textually, as in the case study in section 5.5.

As of now, the graphical modeling environment only supports tag propagation specification. The fault tree generation engine, however, can operate in two modes[14]:

[11] matlabcontrol uses the unofficial MATLAB JMI interface. The MATLAB API for Java would be the choice supported by MathWorks, but requires greater effort in GUI development.

[12] The chosen storage concept is debatable. The implementation centralizes safety assessment data storage in one block for the entire model. If the model changes, references break. It has been chosen because it simplifies experimentation with the method proposed in this chapter, but for productive use, a more localized storage concept would be preferable. The design principle of Simulink would be to use Model Verification blocks from the Simulink Design Optimizer library for failure conditions, and store block behavior locally with each block.

[13] A warning is automatically issued when this recommendation is violated.

[14] I.e. the fault tree generation engine is functionality not yet supported by the graphical modeling environment.

- *Pure tag propagation-based cut set search*, without the dependency to choco and supported by the GUI, in the class `PropagationCutSetTester` and its dependencies

- *CSP-based cut set search*, limited to a scripting interface, in the class `Constraints-CutSetTester` and its dependencies, implementing the full algorithm described in Section 5.3.

An example for the scripting interface of the CSP-based cut set search engine is provided in `org.willea.ftms.ConstraintTest.m` in the additional `tests` package.

Because tag propagation-based cut set search does not allow specification of fault transformations, it is purely experimental and not recommended for practical use. It is based on finding connections between target ports and tag sources. If any **T**-port is connected to the target port, the final tag assignment can directly be assumed to be **T**. Otherwise, search for a connected **C**-port is conducted, and, if found, **C** is assigned. If neither are connected, **N** is assigned. Without the added complexity of tag transformation, this algorithm is especially useful for understanding the interaction between the classes involved in fault tree generation, and how topology graphs are handled programmatically.

All source code written in the course of this work was commented extensively.

5.5 Case study: Fault tree generation for a hybrid flight control system

In order to showcase fault tree generation in a practically relevant example, the model of Chapter 3 by Lauffs [45] is being re-used in this chapter's context. Refer to Figure 3.1 in in Chapter 3 for model structure. In this case study, a fault tree for control surface hardover (usually classified as catastrophic according to [34]) is generated from safety design intent specification and model structure. It can be formalized pessimistically as the tag **T** being propagated to the hinge. Two possible sets of root causes are of interest: Erroneous Automatic Flight Control System (AFCS) output alone and in combination with the overload clutch being stuck closed.

While tag propagation rules are straightforward for most components in the model, the following components have specific behavior:

- The overload clutch acts as a bi-directional filter for load spikes. In terms of safety design intent, it decouples the assignments across its ports: All combinations are plausible. Minimization of the final tag assignment ensures that excessively pessimistic assignments are discarded.

- When stuck closed, the overload clutch propagates tags without filtering or transformation.

- The superposition gear's carrier receives the highest tag assignment from its sun and planet gears[15].

[15]In the model given by Lauffs [45], the AFCS portion is self-locking, so the port connected to the

- The hinge can jam. In terms of safety design intent, it becomes a source of **T**.
- The FC is observed at the Hinge's position scope.

Because no tag transformation is performed, even the simplistic tag propagation semantics would suffice for determining possible tag assignments at the hinge. For the sake of showcasing the full tag propagation and transformation semantics, this simplification is not exploited in this case study. The full tag propagation and transformation rules are given in Table 5.3.

All behavior below the model's top level is abstracted away by top level component specification. Not all behaviors modelled in the case study of Chapter 3 are taken into account in this case study, because only faults that can cause the selected FC are modelled in this case study. Additionally, the following of components are simplified for the sake of clarity:

- the input 'APMech_ref' of the 'AFCS' block at which the angle deviation is measured is ignored because it does not take part in failure propagation
- the output 'StkSpeed' of the 'Stk'-block is ignored because its signal is discarded anyway
- the block 'PilotCmds' is ignored — instead, the blocks 'Stk' and 'AFCS' are modelled with fixed tag assignments, which is equivalent to them being modelled as tag propagation sources

Figure 5.7 shows the fault tree produced automatically from the specified safety design intent. As expected, the overload clutch in nominal behavior provides a fail-safe against erroneous AFCS output.

overload clutch never receives **T** from the superposition gear, but this is irrelevant in this case study and has not been modelled for the sake of clarity.

Table 5.3: Case study — Tag propagation and transformation rules of a hybrid flight control system; Behavior tables are behaviors × ports and ports are in the order of low to high port indices and left to right, e.g. L1 L2 R1 R2.

Block	Behavior	Tag assignments
AFCS	nominal	$\begin{bmatrix} \mathbf{C} & \mathbf{N} & \mathbf{C} \\ \mathbf{C} & \mathbf{C} & \mathbf{C} \\ \mathbf{C} & \mathbf{T} & \mathbf{C} \end{bmatrix}$
AFCS	erroneous	$\begin{bmatrix} \mathbf{T} & \mathbf{N} & \mathbf{T} \\ \mathbf{T} & \mathbf{C} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}$
Hinge	nominal	$\begin{bmatrix} \mathbf{N} & \mathbf{N} \\ \mathbf{C} & \mathbf{C} \\ \mathbf{T} & \mathbf{T} \end{bmatrix}$
Hinge	jammed	$\begin{bmatrix} \mathbf{T} & \mathbf{T} \end{bmatrix}$
Overload clutch	nominal	$\begin{bmatrix} \mathbf{N} & \mathbf{N} \\ \mathbf{N} & \mathbf{C} \\ \mathbf{N} & \mathbf{T} \\ \mathbf{C} & \mathbf{N} \\ \mathbf{C} & \mathbf{C} \\ \mathbf{C} & \mathbf{T} \\ \mathbf{T} & \mathbf{N} \\ \mathbf{T} & \mathbf{C} \\ \mathbf{T} & \mathbf{T} \end{bmatrix}$
Overload clutch	stuck closed	$\begin{bmatrix} \mathbf{N} & \mathbf{N} \\ \mathbf{C} & \mathbf{C} \\ \mathbf{T} & \mathbf{T} \end{bmatrix}$
Stk	nominal	$\begin{bmatrix} \mathbf{C} & \mathbf{C} \end{bmatrix}$
Transmission	nominal	$\begin{bmatrix} \mathbf{N} & \mathbf{N} \\ \mathbf{C} & \mathbf{C} \\ \mathbf{T} & \mathbf{T} \end{bmatrix}$

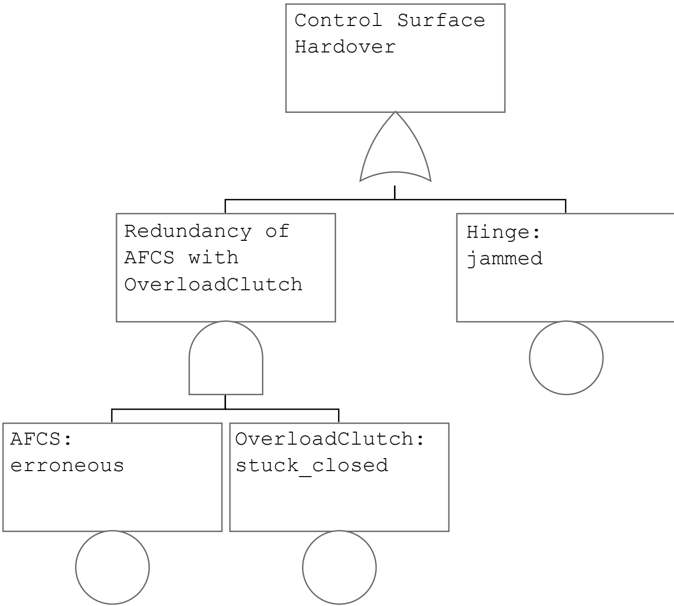

Figure 5.7: Automatically generated fault tree for control surface hardover[16]

Chapter 6

Conclusion

This chapter summarizes the obtained results for each contribution chapter and towards the objective of the thesis as a whole. It also gives directions for continuing towards open issues of the objective, and research opportunities stemming from the obtained results.

6.1 Hybrid failure simulation

The operational semantics for hybrid failure simulation presented in this work fulfills the requirements towards behavioral failure modelling languages as stated by Joshi and Heimdahl [1]. Conceptually, it should be ready for industrial application because it is capable of producing maintainable models in an efficient way. This claim should be evaluated in future research, proving that fulfillment of the above requirements yields the expected results in practical application. Additionally, the currently prototypical implementation could be extended, tested independently of the author of this work, and its performance optimized. Because the underlying technology stack is proprietary, a hand-over to its owner MathWorks could be prepared to stimulate development of the commercial software in a direction that is compatible with behavioral failure modelling and simulation. Such a hand-over would ideally contain the following items:

- Overview on theoretical background on hybrid simulation
- Requirements towards behavioral failure modelling languages
- Software design of the implementation
- Source code and known issues of the implementation
- Sample/demonstration models and tests

The performance of the implementation for hybrid failure simulation presented in Chapter 3 mainly depends on the computational costs for discrete transitions and the ensuing reconfiguration of continuous system state. Execution is slow due to the

overhead of the interfaces to Simulink and Simscape that are used in the process of executing transitions.

After the completion of the presented implementation, a deeper interface for implementation became available [67]. This motivates revisiting the topic and determining whether an equivalent implementation of the same operational semantics is possible on these new interfaces, and whether the reduced overhead of the new interface manifests in faster execution. Furthermore, it is evidence for the practical relevance of hybrid simulation not only for failure simulation, but also for nominal behavior simulation.

The omission of post-conditions for hybrid mode transitions makes variable time-step solvers the preferable choice for simulation. Fixed time-step solvers would, in the worst case, detect a mode change at the end of one time step in the calculation of the next time step. This effect can be limited by setting short time-steps. In variable time-step solvers, this is not necessary: They are capable of resolving the time of discrete mode changes and are thus more robust against problems that arise from similar time step scale and physical effect time scale (or even larger time steps). However, this issue is not specific to hybrid models and affects all models made for simulation. In practice, fixed time-step solvers are preferred in the context of embedded systems because they make fulfillment of real-time requirements easier. For safety assessment, this becomes relevant when software components are part of the system under analysis. In such cases, the fixed time-step width of the embedded software component and the variable time-step simulation of the hybrid system portion need to be made compatible[1]. Two solutions can be envisioned: Either, the fixed time-step intervals could be added as mandatory time steps to variable time step choice and the real-time embedded component outputs are only allowed to change at those time-steps, or hybrid simulation needs to be executed in sufficiently short, fixed time-steps. Exploring these options might enable the usage of the proposed method in the safety assessment of real-time embedded systems.

6.2 Fault tree pessimism

A computational method for iterating over the leaves of binary decision diagrams representing fault trees, called scenarios, is given, with the following properties:

1. Selective: Only 0-leaves (negative scenarios) are returned[2].

2. Disjoint: Intersections of scenarios are empty, i.e. no scenario or portion thereof is returned twice.

3. Ordered by probability: Scenarios are returned in descending order of their probability.

[1] I thank Prof. Dr. Julien Provost for pointing that out to me.

[2] An iterator with the same other properties but returning 1-leaves for application in fault tree generation is achieved by operating on the complementary Binary Decision Diagram (BDD).

4. Truncatable: At a given probability, e.g. the complement of the certification limit, iteration can be cancelled, so no more scenarios than operationally necessary are returned.

If can also be configured to return full assignments instead of leaves, i.e. expanding leaves until a value for each variable of the BDD is given. This set of properties enables two applications: Guiding fault tree validation or generation, as explained in Section 1.4.4.

Its application to fault tree validation is ready for introduction into industrial application. Therefore, the next step would be to present it to design organizations and aviation safety authorities for evaluation and support its application through software integration into existing frameworks for fault tree analysis.

The presented method enables a simplification in the safety assessment process. In the traditional method, probability limits for certification are set for each Failure Condition (FC). Conceptually, this is counter-intuitive: The passenger of a commercial aircraft would expect probability limits to be set against him being harmed. Probability limits against FCs are an incentive for design organizations to define their FCs at a fine level of granularity, so that many, low-probability FCs pass certification requirements on probability. In practice, this is prohibited by limiting the number of FCs in an Aircraft Safety Assessment (ASA) to about 100. Historically, this was a necessary workaround for not being able to do fault tree analysis and validation on large fault trees. With the advent of efficient fault tree analysis through BDDs and the proposed method for validation, that compromise becomes obsolete. Instead of treating individual fault trees, *fault tree forests* of ORed fault trees could be analyzed. Fault tree forest evaluation would make the somewhat arbitrary process of FC definition irrelevant for the safety proof. Highly-integrated systems such as modern flight control and guidance systems would benefit the most from such an approach: They tend to have fewer but more probable FCs because they have fewer system boundaries than conventional flight control system architectures. In the light of the current trends towards greater automation and integration, fault tree forests could aid in rapid innovation without compromising safety if they were accepted by certification authorities as a (superior) alternative to probability evaluation of individual fault trees.

The application of the presented method to fault tree generation is of value when applying formal verification to fault tree generation. It allows dividing a probabilistic safety property corresponding to the certification limit to top event probability into a set of deterministic safety properties that address disjoint combinations of active and inactive fault. In formal verification, this would make it possible to spend computational effort in proportion to the associated probability of the properties under verification. If a method for formal verification of safety properties of general hybrid systems was available but computationally costly, this would help to keep required computational effort within manageable bounds.

6.3 Fault tree generation

A useful abstractions of quantitative failure behavior into unidirectional fault propagation and transformation has been proposed. It allows directly capturing safety architecture design intent by specifying barriers to fault propagation and redundancies and produce standard-conformant fault trees for the aviation industry. These features are specified declaratively — They are not derived from quantitative behavior specification. The generation of the fault propagation and transformation abstraction is the only manual element in this framework for model-based safety assessment.

The method has similarities with Hierarchically Performed Hazard Origin & Propagation Studies (HiP-HOPS) [57], but does not depend on the annotated model being directional, i.e. being specified in a dataflow-based modelling language. Because of the practical relevance of Differential-Algebraic Equation System (DAE)-based modelling languages, this difference is essential to widespread practical application. The underlying semantics of the abstraction are transformed into a Constraint Satisfaction Problem (CSP), which in turn makes specification less intuitive: Transformation of faults in directed models is specified as a local input-output-model, while the undirected approach indicates that transformations always act in both directions.

The method shares the same limitations as any model-based method for fault tree generation: Only specification in the model can be taken into account during fault tree generation. While this covers most of the work of conventional fault tree generation, physical separation that only becomes relevant in case of failure cannot be part of a behavioral model based on nominal behavior. As an example, consider rechargeable batteries with malfunctioning chargers which cause them to overheat and explode, which may in turn damage and disable fire containment equipment. Because nominal behavior models would not explicitly contain measures of physical separation of failing equipment and the equipment for containing the failure, such interaction would not be taken into account in fault tree generation. In safety assessment as per ARP 4761 [28], Common Modes Analysis (CMA) reveals such issues. When fault tree generation is automated with the help of behavioral models, the importance of analyses of the non-behavioral aspects of the real system in CMA increases.

6.4 Review of the research objective and future research directions

With the contribution from Chapter 3, the first objective of enabling physical simulation of nominal and failure behavior has been fulfilled, and the requirements towards maintainable models and an efficient modelling process stated by Joshi and Heimdahl (see [1]) are met.

The second objective is fulfilled through the contribution of Chapter 4. The consequent problem of efficiently defining reference behavior in a way that can be evaluated automatically is solved by the contribution of Chapter 5.

A definitive follow-up research question is finding a suitable formalization schema of safety requirements into a formally verifiable format. It could be established from

practical experience with the proposed operational semantics for failure behavior simulation and the proposed method for fault tree generation. Comprehensive application of model-based system development — with models encompassing the complete environment of the system and capable of simulating 'missions' (or use cases of system application, in a broader sense than aircraft development) in nominal and emergency circumstances — makes a breakthrough on this research question within reach. The proposed operational semantics for hybrid (failure) simulation provides a missing piece in modeling and simulation technology for this goal.

A solution to fully automated fault tree generation without safety design intent as an intermediate layer of abstraction may be the concept of over-approximation of state space evolutions into linear convex hulls (see Althoff et al. [11]) or similar verification-friendly abstractions on model-level. There is a synergy between this approach and the contribution of Chapter 4: Over-approximation strategies require a computationally cheap method for allocating level of detail of the approximation to 'interesting' subspaces of the solution space under analysis. Chapter 4 offers the associated probability of the scenario under analysis, which would be ideal in safety assessment.

The method proposed in Chapter 5 invites skepticism as to whether newly created effort for manual annotation is lower than the effort previously spent on the now automated process steps. Additional case studies could assess the method's practical feasibility.

Hybrid simulation as proposed in Chapter 3 can be used in conjunction with the method for fault tree generation proposed in Chapter 5 for verification of the effectiveness of the implemented design to fulfill safety design intent through testing. Which aspects of safety design are probabilistically most relevant can be derived using the contribution in Chapter 4. While the method proposed in Chapter 5 addresses Preliminary System Safety Assessment (PSSA), such a workflow for verifying fault trees obtained automatically would support System Safety Assessment (SSA) in a purely model-based workflow. However, this would not relieve the safety engineer from manually specifying safety design intent.

Whether closing the link between DAE-based component behavior definition as proposed in Chapter 3 and safety design intent specification on component level as proposed in Chapter 5 can be automated, is the eminent research question that arises from this work. It would clear the last manual process step in model-based safety assessment besides physical modelling.

There may be a component-oriented solution to the problem of automatic formulation of safety design intent in a practically relevant subclass of hybrid systems. It is motivated by the abstraction of Lagrangian mechanics in bond graphs (see Karnopp et al. [43]). DAE-based systems of classical mechanics interpret the macroscopic, physical processes as the exchange of two states, *flow* and *effort*, and thereby localize causes and effects when defining the order of computation of each component's DAEs. When eligible causes could be shown to be separated from where the effect under analysis would manifest, this may allow fully automated detection of the necessity of barriers to fault propagation or fault transformation. The issue in choosing between fault propagation barriers and fault transformation is that the reasons for which the safety engineer makes his decision usually are not in the behavioral model.

Bibliography

[1] A. Joshi and M. P. Heimdahl, "Behavioral fault modeling for model-based safety analysis," in *10th International Symposium on High-Assurance Systems Engineering*, pp. 199–208, 2007.

[2] P. J. Mosterman, "Hybrsim, a modelling and simulation environment for hybrid bond graphs," *Journal of Systems and Control Engineering*, vol. 216, no. 1, pp. 35–46, 2002.

[3] C. Schallert, "Automated safety analysis by minimal path set detection for multi-domain object-oriented models," in *11th International Modelica Conference* (P. Fritzson and H. Elmqvist, eds.), pp. 565–575, 2015.

[4] G. Schellhorn, A. Thums, W. Reif, *et al.*, "Formal fault tree semantics," in *6th World Conference on Integrated Design & Process Technology, Proceedings of*, (Pasadena, CA), Society for Design and Process Science, 2002.

[5] "Product documentation." http://www.mathworks.com/help/index.html, 2017. Accessed: 2017-11-21.

[6] R. Alur, "Formal verification of hybrid systems," in *International Conference on Embedded Software*, pp. 273–278, 2011.

[7] A. Arnold, G. Point, A. Griffault, and A. Rauzy, "The AltaRica Formalism for describing concurrent systems," *Fundamenta Informaticae*, vol. 40, no. 2, pp. 109–124, 1999.

[8] APSYS, "Simfia." https://www.apsys-airbus.com/en/digital-software-en/#SIMFIA. Accessed: 2017-11-29.

[9] P. Struss and A. Fraracci, "Automated model-based FMEA of a Braking System," in *8th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes, Proceedings of* (Astorga Zaragoza, Carlos Manuel and A. Molina, eds.), (Laxenburg, Austria), International Federation of Automatic Control, 2012.

[10] P. Hönig, R. Lunde, and F. Holzapfel, "Model based safety analysis with smartiflow," *Information*, vol. 8, no. 1, 2017.

[11] M. Althoff, A. Rajhans, B. H. Krogh, S. Yaldiz, X. Li, and L. Pileggi, "Formal verification of phase-locked loops using reachability analysis and continuization," in

Proceedings of the International Conference on Computer-Aided Design, pp. 659–666, 2011.

[12] G. E. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel, "Verification of automotive control applications using S-TaLiRo," in *American Control Conference*, pp. 3567–3572, 2012.

[13] W. Kuo and X. Zhu, "Some recent advances on importance measures in reliability," *IEEE Transactions on Reliability*, vol. 61, no. 2, pp. 344–360, 2012.

[14] F. M. Brown, *Boolean Reasoning: The Logic of Boolean Equations*. New York: Springer Science & Business Media, 2012.

[15] S. J. Russell and P. Norvig, *Artificial Intelligence: A modern approach*. Prentice Hall Series in Artificial Intelligence, Upper Saddle River, NJ: Pearson, 3 ed., 2009.

[16] A. Joshi, *Behavioral Fault Modeling and Model Composition for Model-based Safety Analysis*. Dissertation, University of Minnesota, Minneapolis, MN, 2008.

[17] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," *Lecture Notes in Computer Science*, vol. 736, pp. 209–229, 1993.

[18] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[19] M. Güdemann and F. Ortmeier, "A framework for qualitative and quantitative formal model-based safety analysis," in *12th International Symposium on High-Assurance Systems Engineering*, pp. 132–141, 2010.

[20] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *Computer aided verification* (G. Gopalakrishnan and S. Qadeer, eds.), Theoretical Computer Science and General Issues, (Berlin), pp. 585–591, Springer, 2011.

[21] M. Bozzano, A. Villafiorita, O. Åkerlund, *et al.*, "ESACS: An integrated methodology for design and safety analsyis of complex systems," in *European Safety & Reliability International Conference* (S. Anderson, M. Felici, and L. Bev, eds.), vol. 2003, (Berlin), pp. 237–245, Springer, 2003.

[22] T. Peikenkamp, A. Cavallo, L. Valacca, E. Böde, M. Pretzer, and E. M. Hahn, "Towards a unified model-based safety assessment," in *Computer Safety, Reliability and Security* (J. Górski, ed.), vol. 4166 of *Lecture Notes in Computer Science*, (Berlin), pp. 275–288, Springer, 2006.

[23] P. Hönig and R. Lunde, "A new modeling approach for automated safety analysis based on information flows," in *25th International Workshop on Principles of Diagnosis*, 2014.

[24] O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, A. Griffault, *et al.*, "ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects," *Proc. ERTS*, vol. 2006, pp. 1–11, 2006.

[25] M. Adachi, Y. Papadopoulos, S. Sharvia, D. Parker, and T. Tohdo, "An approach to optimization of fault tolerant architectures using HiP-HOPS," *Software: Practice and Experience*, vol. 41, no. 11, pp. 1303–1327, 2011.

[26] A. Girard, C. Le Guernic, and O. Maler, "Efficient computation of reachable sets of linear time-invariant systems with inputs," in *Hybrid Systems: Computation and Control: 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006. Proceedings* (J. P. Hespanha and A. Tiwari, eds.), pp. 257–271, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

[27] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?," *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 94–124, 1998.

[28] Society of Automobile Engineers, "Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment." http://standards.sae.org/arp4761/, 1996-12-01. SAE ARP 4761.

[29] Bundesrepublik Deutschland, "Luftverkehrs-Ordnung: LuftVO," 1963-08-10.

[30] Bundesrepublik Deutschland, "Luftverkehrs-Zulassungs-Ordnung: LuftVZO," 1964-06-19.

[31] European Parliament and Council, "Common rules in the field of civil aviation and establishing a european aviation safety agency: Ec 216/2008," 2008.

[32] European Aviation Safety Agency, "Certification specification for normal, utility, aerobatic and commuter aeroplanes: CS-23." https://www.easa.europa.eu/document-library/certification-specifications/cs-23-amendment-5, 2017-03-29. CS-23 Amd. 5.

[33] European Aviation Safety Agency, "Certification specification for large aeroplanes: CS-25." https://www.easa.europa.eu/document-library/certification-specifications/cs-25-amendment-20, 2017-08-24. CS-25 Amd. 20.

[34] Federal Aviation Administration, "System safety analysis and assessment for part 23 airplanes." http://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentID/1019681, 2011-11-17. AC-23.1309 1E.

[35] Society of Automobile Engineers, "Guidelines for development of civil aircraft and systems." http://standards.sae.org/arp4761/, 2010-12-21. SAE ARP 4754 Ed. A.

[36] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook." https://www.nrc.gov/reading-rm/doc-collections/nuregs/staff/sr0492/, 1981. NUREG-0492.

[37] W. E. Vesely, J. Dugan, J. Fragola, J. Minarick, III, and J. Railsback, "Fault tree handbook with aerospace applications." https://elibrary.gsfc.nasa.gov/_assets/doclibBidder/tech_docs/25.%20NASA_Fault_Tree_Handbook_with_Aerospace_Applications%20-%20Copy.pdf, 2002-08.

[38] O. Coudert and J. C. Madre, "Implicit and incremental computation of primes and essential primes of boolean functions," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 36–39, IEEE, 1992.

[39] A. Rauzy, "New algorithms for fault trees analysis," *Reliability Engineering & System Safety*, vol. 40, no. 3, pp. 203–211, 1993.

[40] W. S. Jung, S. H. Han, and J. Ha, "A fast BDD algorithm for large coherent fault trees analysis," *Reliability Engineering & System Safety*, vol. 83, no. 3, pp. 369–374, 2004.

[41] R. Alur, C. Courcoubetis, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, 1995.

[42] IEEE, "Floating-point arithmetic." http://ieeexplore.ieee.org/document/4610935/, 2008-08-29. IEEE 754:2008.

[43] D. Karnopp, R. Rosenberg, and A. S. Perelson, "System dynamics: A unified approach," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. SMC-6, no. 10, p. 724, 1976.

[44] P. J. Mosterman and G. Biswas, "A theory of discontinuities in physical system models," *Journal of the Franklin Institute*, vol. 335, no. 3, pp. 401–439, 1998.

[45] P. J. Lauffs, "Entwurf und Simulation einer Systemregelung zum Betrieb einer hybriden elektromechanischen Flugsteuerung," 2011.

[46] A. Wille and F. Holzapfel, "Exploiting pessimism for fault tree completeness," in *AIAA SciTech Forum*, (Dallas, TX), American Institute of Aeronautics and Astronautics, 2017.

[47] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[48] A. Rauzy and Y. Dutuit, "Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia," *Reliability Engineering & System Safety*, vol. 58, no. 2, pp. 127–144, 1997.

[49] Object Management Group, "Systems modelling language (SysML)." http://www.omg.org/spec/SysML/1.5/, 2017-05. Ed. 1.5, formal/17-05-01.

[50] J. Ossowski and C. Baier, "Symbolic reasoning with weighted and normalized decision diagrams," *Electronic Notes in Theoretical Computer Science*, vol. 151, no. 1, pp. 39–56, 2006.

[51] Oracle Corporation, "Java." http://www.java.com, 2014. Accessed: 2016.02.08.

[52] J. Whaley, "Javabdd." http://javabdd.sourceforge.net, 2005. Accessed: 2016-02-04.

[53] J. Lind-Nielsen, "Buddy." https://sourceforge.net/projects/buddy/, 2004. Accessed: 2016-02-04.

[54] W. Gradkowski, "Jmatio." https://github.com/gradusnikov/jmatio, 2014. Accessed: 2016-02-04.

[55] P. Fenelon and J. A. McDermid, "An integrated tool set for software safety analysis," *Journal of Systems and Software*, vol. 21, no. 3, pp. 279–290, 1993.

[56] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey, "Towards integrated safety analysis and design," *ACM SIGAPP Applied Computing Review*, vol. 2, no. 1, pp. 21–32, 1994.

[57] Y. Papadopoulos and J. A. McDermid, "Hierarchically performed hazard origin and propagation studies," in *Computer Safety Reliability and Security* (A. Pasquini, ed.), vol. 1698 of *Lecture Notes in Computer Science*, (Berlin), pp. 139–152, Springer, 1999.

[58] "Saxon 9.5 known issue." https://saxonica.plan.io/issues/1944. Accessed: 2017-08-16.

[59] "Matlab license validation issue." https://de.mathworks.com/matlabcentral/answers/97167-why-will-matlab-not-start-up-properly-on-my-windows-based-system. Accessed: 2017-08-16.

[60] "Matlab compiler support." https://de.mathworks.com/support/compilers.html. Accessed: 2017-08-16.

[61] "Debugging java components in the matlab java vm." https://de.mathworks.com/matlabcentral/answers/102080-how-do-i-debug-my-java-source-code-that-i-am-calling-from-the-matlab-environment. Accessed: 2017-08-16.

[62] "matlab.engine.shareengine: Convert running matlab session to shared session." https://www.mathworks.com/help/matlab/ref/matlab.engine.shareengine.html. Accessed: 2017-11-20.

[63] "Call java libraries: Static path." https://www.mathworks.com/help/matlab/matlab_external/static-path.html. Accessed: 2017-11-20.

[64] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco solver*, 2016. Accessed: 2018-01-04.

[65] J. Kaplan, "matlabcontrol: A wrapper for the matlab-to-java interface." https://code.google.com/archive/p/matlabcontrol/. Accessed: 2017-08-16.

[66] N. Twigg and G. Alankus, "Matconsolectl: A java api that allows calling matlab from java," 2016.

[67] The MathWorks, Inc., "Discrete events and mode charts: Model discrete changes in system behavior." https://www.mathworks.com/help/physmod/simscape/discrete-events-and-mode-charts.html. Accessed: 2017-12-01.

[68] Object Management Group, "Unified modelling language (UML)." http://www.omg.org/spec/UML/2.5/, 2015-05. Ed. 2.5, formal/15-03-01.

Appendix A

Software Architecture of Custom Components

This appendix contains Unified Modelling Language (UML) [68] class and activity diagrams as a detailed software architecture description of the custom components written in the course of this work.

A.1 Hybrid Failure Simulation

Hybrid simulation is written in MATLAB. It is mixed between object-oriented and procedural programming. Functions and scripts are greyed out in the class diagram in Figure A.1. Utility classes and functions (such as Graphical User Interface (GUI) components) and block libraries are not displayed for the sake of clarity.

CoreController contains the logic for executing hybrid simulation. Following the concept of MATLAB that classes are only introduced when object-oriented information storage is required, all members of the package are functions. CoreModel contains the classes for extending the information associated with models and blocks. Because navigating Stateflow diagrams is very slow its Application programming interface (API), lean wrapper classes (Mode and Transition) have been created. They are used for un-

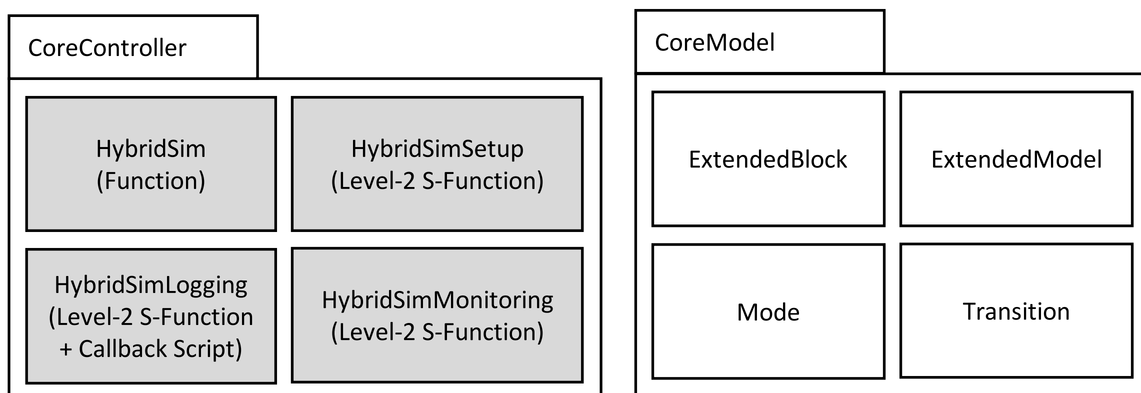

Figure A.1: Class diagram for the implementation of Chapter 3, hybrid failure simulation

coupling the Stateflow diagram's structure of each extended block from hybrid simulation during the simulation's initialization phase.

HybridSim is the function that controls hybrid simulation. Figure A.2 illustrates its interaction with the other components during simulation. The other three functions, HybridSimLogging, Monitoring, and Setup are Level-2 MATLAB S-Functions that define the behavior of the blocks in the `hsimctrl_lib` Simulink library (not displayed in Figure A.1) at simulation-time. HybridSimMonitoring-blocks listen for transitions in the Stateflow diagrams, which in turn specify the discrete portion of hybrid behavior. The HybridSimSetup-block collects all transitions reported by HybridSimMonitoring-blocks and executes transitions as specified in the target block's Stateflow diagram. Because each extended block has its own HybridSimMonitoring- and -Logging-block, transition detection, and continuous state logging are localized at simulation time. This ensures context-free hybrid behavior modeling and simulation. The single HybridSim-Setup-block per model contains data for failure scheduling (i.e. triggering transitions at specific times).

The GUI for scheduling forced transitions during simulation (from the CoreView-package) can be accessed through the block mask of the HybridSimSetup-block. There is also a callback that initializes hybrid simulation there[1]. During simulation, the Level-2 S-Function of the HybridSimMonitoring-block injects scheduled transitions.

The HybridSimLogging-block allows viewing the retrieved run through its callback function. The trace pieces of each simulation interval are concatenated[2].

A.2 Fault Tree Pessimism

The probability-sorted, probability-truncated iterator of 0-leaves of fault trees has been implemented in Java, along with a test harness for executing the provided case study. The components it shares with the implementation of the third contribution of this thesis are in the `org.willea.ftms` package. Its specific components are in the `org.willea.ftms.fta` package. Figure A.3 shows the class diagram of both packages. Components solely relevant for the GUI are omitted for the sake of clarity.

During execution, the test harness `FaultTreePessimismTest` retrieves a Fault Tree's BDD from the FaultTreeLibrary and hands it over to the ProbabilisticLeafIterator, along with the fault tree's basic event probabilities. The ProbabilisticLeafIterator is the core component for probability-sorted, probability-truncated iteration of 0-leaves[3]. Because

[1]Simulink's green 'run'-button triggers default Simulink simulation, so the presented implementation of hybrid simulation can be added to existing Simulink/Simscape models without interfering with their default simulation behavior. Only the hybrid simulation button from the HybridSimSetup-block mask triggers hybrid simulation.

[2]In the current implementation, location changes are not stored along with the continuous state evolution. This limits the visualization of the retrieved run: Ideally, one would want each continuous trace piece to be visually differentiated, e.g. by being color-coded, with a legend reporting the active modes of the component. Currently, jumps are visualized by a linear interpolation of nearly infinite gradient between σ and σ' and the location of each trace piece is only reported in the textual log during simulation.

[3]The implementation is agnostic to whether the supplied BDD encodes the characteristic function of a fault tree. Any BDD can be treated.

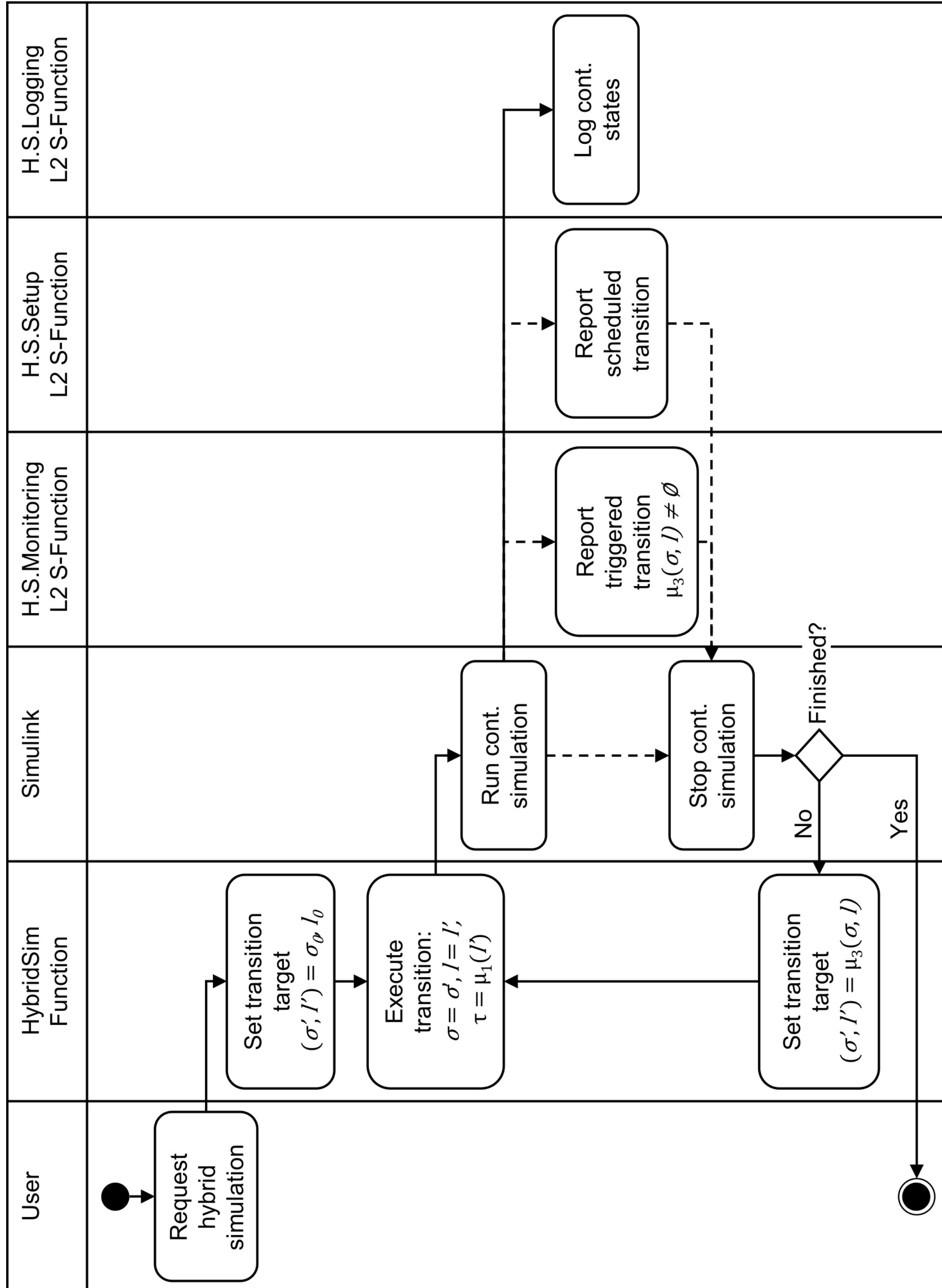

Figure A.2: Activity diagram for the implementation of Chapter 3, hybrid failure simulation

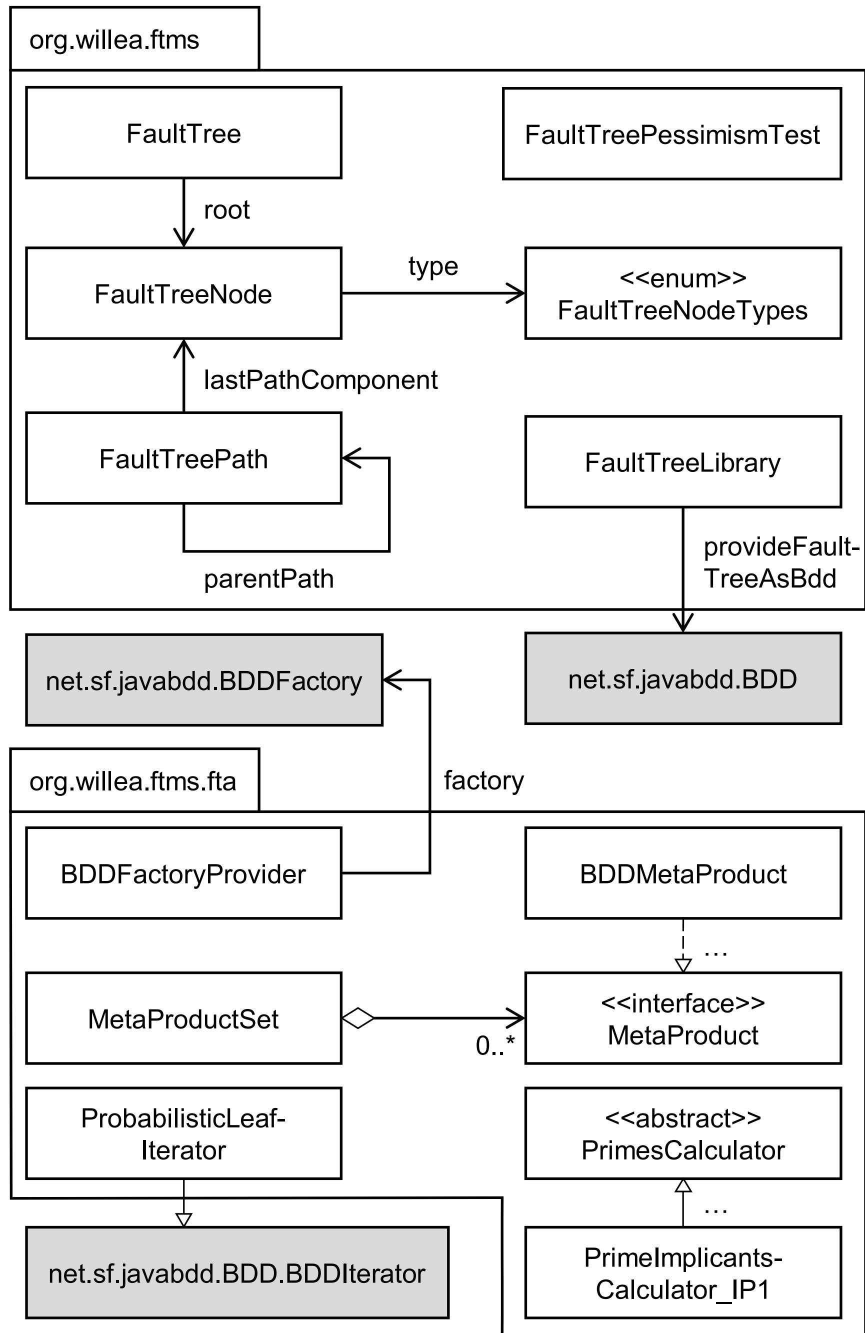

Figure A.3: Class diagram for the implementation of Chapter 4, the probability-sorted, probability-truncated iterator of 0-leaves of fault trees

it inherits from BDDIterator, it takes a BDD as input and returns the paths to its leaves as BDDs.

As a by-product of the presented implementation, the prime implicants calculation algorithms proposed by Coudert and Madre [38] have been implemented. Only one is shown in the diagram. The other two (IP2 and positive prime implicants calculation) have been abbreviated by the three dots next to the inheritance connector in the diagram. Likewise, alternate implementations of Minimal Cut Sets (MCSs) as BDDs in the form of meta-products have been created (with bit sets and with boolean arrays) and abbreviated in the diagram. They are of no relevance to the proposed method. During the development, the implementations of PrimesCalculator where used for debugging the probabilistic leaf iterator, and the implementations of MetaProduct for benchmarking various libraries for BDD storage and manipulation.

The BDDFactoryProvider is a workaround for a bug [4] in the chosen library for BDD storage and manipulation and maintains a singleton BDDFactory instance.

A.3 Fault Tree Generation

The implementation of fault tree generation based on model topology and fault propagation and transformation annotation has two main components: One for translating Simulink and Simscape model topology into a directional graph, and one for creating a fault tree template, inferring MCSs from model topology and annotation, and transforming the fault tree template according to the derived MCSs. MATLAB components are annotated with 'M:' in the class diagram illustrating this software architecture in Figure A.4.

The MATLAB TopologyAnalysis package not only contains a function for building a MATLAB digraph-object from model topology, but also to convert it into a format that can be marshaled to the Java portion of the implementation, or plot it for debugging purposes.

The architecture of the implementation is designed with adaptability to other modeling environments and extensibility with new algorithms in mind. The interface to the modeling environment is grouped in the 'org.willea.ftms.matlabinterface' package. This reduces the effort for adding support for an additional modeling environment, e.g. a Modelica-based one, because the portions of the implementation that would need to be instantiated and re-engineered are clearly identified.

Extensibility for new algorithms is even simpler: The FaultTreeFactory class has interfaces for exchanging all its model data classes and algorithms. Some are custom, such as TopologyDigraph or CutSetsFinder.CutSetTester, while others are Java-inherent default interfaces, such as the ones used for the block behavior applicator, the cut sets finder the default behavior provider or the fault tree beautifier.

[4] It can be argued that the performance of the chosen library relies on assumptions about the usage of BDDFactory instances. When two instances share the memory of a Java virtual machine, performance-boosting simplifications in garbage collection produce crashes and incorrect behavior. The performance benefit of the library in the context of this work outweighs the overhead for maintaining a singleton instance and the limitations from employing singletons are irrelevant here.

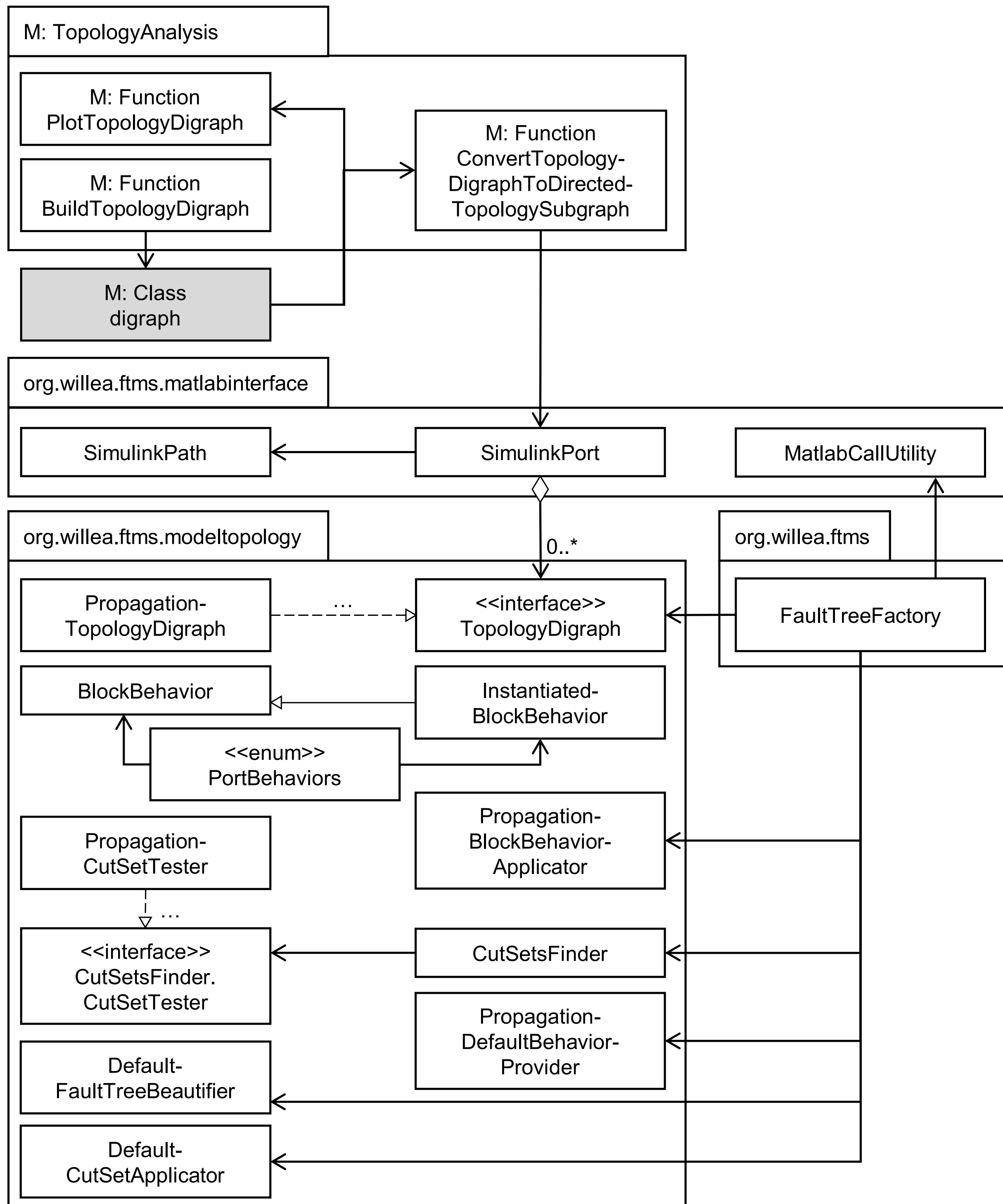

Figure A.4: Class diagram for the implementation of Chapter 5, fault tree generation from fault propagation and transformation annotation

The model data classes are BlockBehavior and InstantiatedBlockBehavior. They contain the relation between Simulink blocks and their ports and abstracted port behaviors. While BlockBehavior addresses library block behavior specification, InstantiatedBlockBehavior is the pendant for model-specific, modified block behavior.

The Javadoc of each of these classes explains their role in the fault tree generation process. An overview of the entire process is given in the activity diagram in Figure A.5.

Each fault tree is based upon a fault tree template, which is system architecture-specific, but component behavior-agnostic. Once the template is built, the topology graph is annotated with all possible component behaviors as specified for each block of the model. The FaultTreeFactory controls the process of determining MCSs, applying them to the fault tree and beautifying it. Each step of that process is executed by a dedicated class and their interaction is abstracted through interfaces. The CutSetFinder decides which cut set candidates to try in PropagationBlockBehaviorApplicator creates a topology graph with the selected behavior for each model block, and PropagationCutSetTester checks whether the resulting system behavior triggers the failure condition. Once all MCSs have been identified, they are applied to the fault tree template. At last, the fault tree beautifier prunes obsolete nodes.

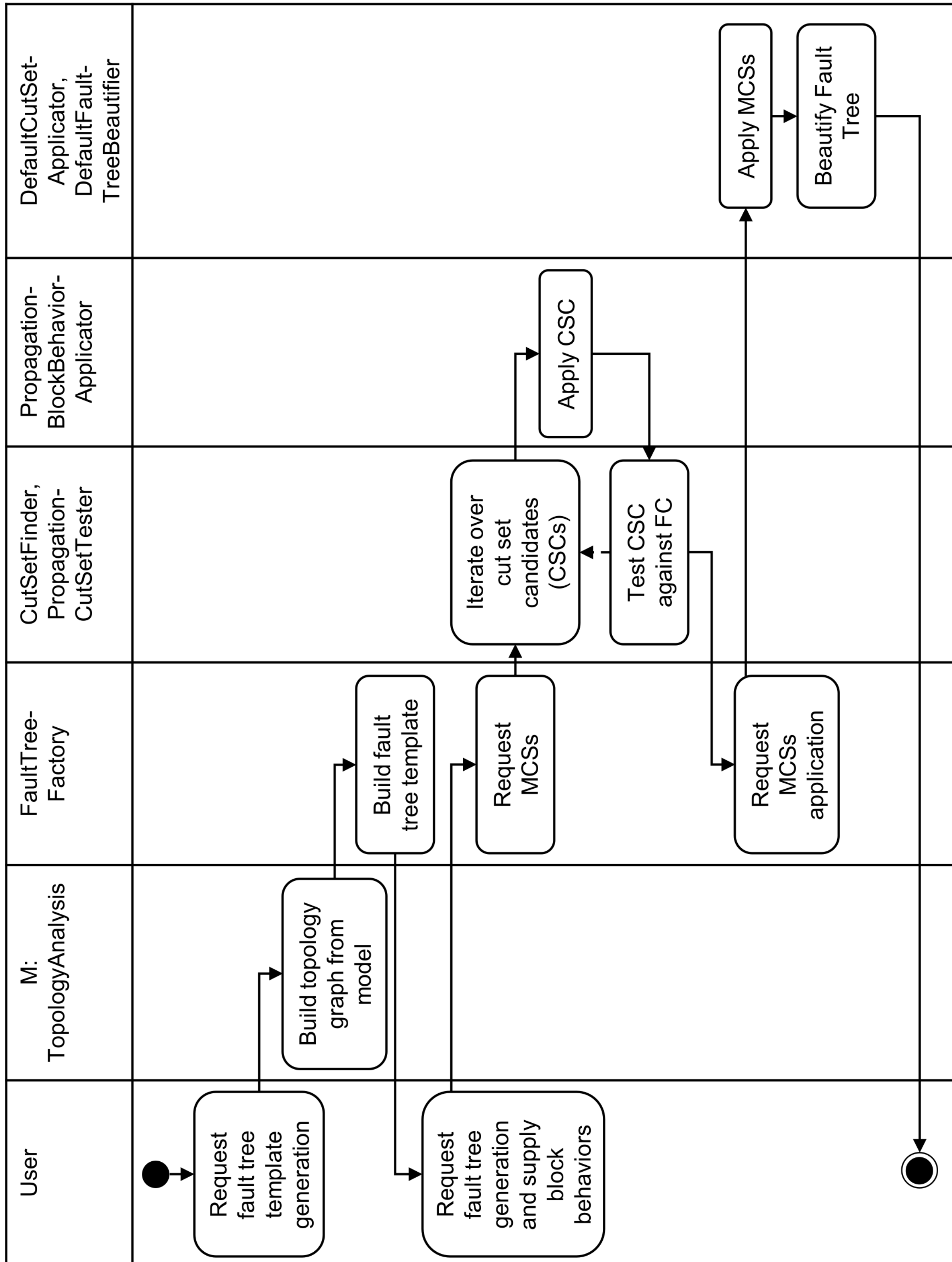

Figure A.5: Activity diagram for the implementation of Chapter 5, fault tree generation from fault propagation and transformation annotation

Appendix B

HSim Modeling Guide

This appendix describes how each element of the operational semantics for failure simulation defined in Chapter 3 are modeled in Simulink and Simscape, using the prototypical implementation *HSim* that provides switching logics definition, execution and monitoring. Its purpose is to help new users employ this prototype software for evaluation and document the design compromises made in building HSim in MATLAB because MATLAB components cannot be altered by the author.

B.1 Hybrid log and setup-blocks

Extended blocks are blocks with hybrid failure behavior semantics attached. During regular Simulink or Simscape simulation, hybrid failure behavior definition is ignored. Only during hybrid simulation is the definition for hybrid failure behavior taken into account. The simulation results of hybrid simulation are only provided in the *Hybrid Log*-blocks, which are the equivalent to Scope-blocks in hybrid simulation. The Hybrid Simulation Setup-blocks provide the functionality for defining custom failure schedules (corresponding to $\ell_{0,b}$ in hybrid failure simulation semantics) and starting hybrid simulation.

Because failure schedules are stored locally in the Hybrid Simulation Setup-block's data, its library link needs to be disabled.

B.2 Hybrid failure behavior specification

Each extended block is a subsystem in which one or more blocks defining its continuous behavior is embedded. The continuous behavior definition block inside it is called the *implementation* of the extended block and corresponds to $\mu_{1,b}$ in hybrid failure simulation semantics. During nominal behavior, the original block takes the role of the implementation by default. Additionally, a handle to the currently active implementation block is stored in a hidden mask parameter called 'hsimActiveMode', which corresponds to $\ell_{i,b}$[1], and whose value is never saved with the model (because it is only relevant during hybrid simulation runtime).

[1]'hsimActiveMode' may only be available at hybrid simulation runtime.

The physical states of each component are measured by sensors and fed into a Stateflow chart (i.e. a graphically-modeled Finite State Machine (FSM)). Each mode in that chart represents a mode of failure behavior. By convention, the default mode is nominal behavior. The transition conditions of each mode correspond to the transition relations in hybrid failure simulation semantics $\mu_{3,b}$. The implementation of the hybrid failure behavior simulator implicitly assumes that physical states are left unchanged over hybrid transitions, but the user can specify state change code as text in the transition action to override defaults.

Note that every mode needs to be reachable from the default mode. When a mode is modeled that cannot be activated by physical effects within the model's scope (e.g. melting clutch in a purely mechanical model), a transition from nominal to that mode with a transition condition that is always false needs to be added to incorporate it in the model anyway.

In Stateflow's object model, each mode is an element in the model hierarchy below the chart. For each such mode, a data object with the title 'Implementation' contains the path to its corresponding library block. Additionally, all parameters of the block are stored as data objects. When the implementation is changed, the parameters from the library block are applied with it, so they model-specific parameter values are conserved through the data object, and automatically re-applied to the new implementation after mode change[2]. The reference between the Stateflow object for the parameter and the property of the 'Implementation' block is created by its technical name. The technical names of parameters can be obtained by searching the list of object parameters of a block, with `get_param(gcb, 'ObjectParameters')`. The usage of these parameters by HSim is not recognized by Simulink, which therefore throws warnings during Simulation. These warnings (for the 'Implementation' and parameter data objects) can safely be ignored.

Because parameters are stored locally in each extended block, their library links need to be disabled.

For the Stateflow chart and the hybrid simulation engine to exchange information on the currently active mode, each chart has an output that reports the currently active mode to an HSim monitoring block with the title 'Execute Mode Changes'. The data exchanged between these two is encoded in a custom enumeration data type for each extended block. When multiple extended blocks from the same library blocks are to be used in one model, Stateflow's object model prefers these two enumeration data types to have different names and will throw a warning if they do not. This can be done in the Model Explorer page of the chart, in the input field "Enum name"[3].

Because MATLAB's table control does not support differing choices in drop-down menus for each row, a workaround has been employed that has the downside of requiring the user to select drop-down menus twice for them to open when opening the drop-down menu and selecting a new table row in one click.

[2]This could be automated further so that only parameter changes or values for new parameters would need to be specified by the user.

[3]Because this data type name is not accessible programmatically, this can currently not be automated.

Appendix C

Java Environment Configuration

The tests and use cases for the Chapters 4 and 5 require a properly-configured Java environment for MATLAB. This appendix provides step-by-step instructions for setting it up.

1. Download and install the current Java 8 Software Development Kit (SDK).
   During installation, choose to install the corresponding Java Runtime Environment. When installing to the default path, this creates a Java 8 SDK installation at a location similar to
   `C:\Program Files\Java\jdk1.8.<minor-version>_<update-id>`
   with version numbers replaced with those of the installed version. For the rest of this walk-through, this path is referenced as *JDK path*. A Java Runtime Installation is created at *JDK path*`\jre`. This path is henceforth referenced as *JRE path*.

2. Set the `MATLAB_JAVA` environment variable to *JRE path*.
   This makes MATLAB use the Java Runtime Environment of the Java SDK as it's internal Java environment and ensures that compiling and execution environment are the same.

3. Copy `startup.m` provided with this thesis to your MATLAB-directory (e.g. `C:\Users\<user-name>\Documents\MATLAB`) or — if you already have a `startup.m`-file at that location — merge it with the existing one.
   This script is executed with the startup of MATLAB and will automatically register the previously installed Java SDK with MATLAB, so that MATLAB uses it for compiling and debugging. It also enables the use of external Java debuggers for convenient debugging from the Integrated Development Environment (IDE) of your choice when MATLAB is started with the '-jdb' flag.

4. Add the provided JAR-files for the `org.willea.ftms` package to your Java class path.
   If you plan to make changes to the provided implementations, use the dynamic Java path (with the `javaaddpath`-command in MATLAB). If you do not need to make any changes or restarting MATLAB after each change is acceptable, use the static class path. This is done creating a file named `javaclasspath.txt` in your

MATLAB preferences folder that contains the appropriate paths to the provided JAR-files.