Fault Tree Analysis (FTA) Support for PASTA

Orhan Tekin

Bachelor Thesis
August 30, 2023

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
M. Sc. Jette Petzold

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Risk analysis is performed during and after system development to ensure the safety and reliability of a system. An analysis is performed to avoid potential undesired events such as the loss of a life or an injury. Risk analysis techniques identify the causes of these undesired events and allow analysts to understand the risks better. Fault Tree Analysis (FTA) is one of these analysis techniques. Fault Trees (FTs) represent the structure of the system. They are constructed and analyzed in FTA, which is usually done manually. Naturally, that is tedious and time-consuming. The development of tools that automate both the construction and the analysis of these FTs can solve this problem.

In this thesis, tools supporting FTA are explored. Additionally, this thesis adds standard FTA support to Pragmatic Automated System Theoretic Process Analysis (PASTA), which already offers a Domain Specific Language (DSL) for System-Theoretic Process Analysis (STPA). PASTA is a Visual Studio Code (VSCode) Extension that is implemented with Langium and Sprotty. FTA support includes creating FTs textually as well as generating a visualization and performing an analysis of the FT automatically. An evaluation of the DSL revealed that every standard FT can be created and analyzed with the cut set analysis. The DSL automatically lays out the generated graph, which is not provided by most of the tools. In the future, PASTA will be improved to support different extensions of FTA before it can be used in realistic use cases. Still, it is a good alternative for basic and less complex cases.

Contents

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Outline . . . 2

2 Foundations **3**
2.1 Fault Tree Analysis . . . 3
2.1.1 Constructing the Fault Tree . . . 3
2.1.2 Qualitative Analysis . . . 4
2.1.3 Quantitative Analysis . . . 6
2.2 STPA . . . 7
2.3 Used Technologies . . . 8
2.3.1 Langium . . . 8
2.3.2 Sprotty . . . 8

3 Related Work **11**
3.1 ITEM ToolKit . . . 11
3.2 A.L.D. RAM Commander . . . 12
3.3 Galileo . . . 12

4 FTA-DSL **15**
4.1 DSL . . . 15
4.2 Visualization . . . 17
4.3 Analysis . . . 18
4.4 Combining FTA with STPA . . . 19

5 Implementation **23**
5.1 Language Server . . . 23
5.1.1 DSL . . . 23
5.1.2 Diagram Generation . . . 24
5.2 Visualization . . . 25
5.3 Analysis . . . 26

6 Evaluation **29**
6.1 Pressure Tank Example . . . 29
6.2 Comparison . . . 31

7 Conclusion **33**
7.1 Summary 33
7.2 Future Work 34

Bibliography **35**

List of Abbreviations **37**

List of Figures

2.1 The graphical representations of gates [RS15]. 4
2.2 Example of an FT [RS15]. 5
2.3 Conversion of an FT to a BDD [RS15]. 6

3.1 Overview of the Galileo tool [DSC00]. 13

4.1 Visualization of failed checks. 17
4.2 Highlighting a cut set in the FT. 20
4.3 Combination of FTA and STPA. 21

5.1 The classes for the graph elements. 24
5.2 Cut sets of an example FT. 27

6.1 Pressure Tank in the FTA-DSL [VGR+81]. 30

Chapter 1

Introduction

Risk is present in many aspects of our lives and describes the possibility for unsafe events to occur. The consequences of ignoring or misjudging risks are often death, pain or serious losses [AR10]. Systems need to be checked for safety in order to prevent such events. Safety is a system property that can only be determined for the system as a whole [Lev20]. The safety of a system cannot be assured even if every individual *component* is safe. However, the failure of individual *components* can lead to system failure. In the construction and maintenance of a system, low priority risks are discarded in order to reduce the number of considered threats [BDJ07]. Naturally, threats that pose bigger risks to a system have a higher priority. Risks can be managed and prevented with risk analysis, which is the focus of this bachelor thesis.

Risk analysis of systems during development is important to prevent serious injury or death. It allows the analyst to understand the risk by determining potential threats and their probabilities [VK09]. Potential threats, such as the loss of life, can be caused by the failure of a system, which can be mitigated by applying risk analysis techniques. Risk analysis ensures that critical assets such as medical devices and nuclear power plants operate safely [RS15].

Fault Tree Analysis (FTA) is a risk analysis technique that evaluates and increases the safety of a system. FTA is a structured, systematic approach that provides insight on potential failures of a system. It identifies vulnerabilities as well as strategies to prevent them. The basic idea of FTA is that complex system failure often originates from the failure of basic system components or other events. A Fault Tree (FT) is a graphical representation of a system that breaks down the reason for a system failure. The analysis of FTs computes combinations of events, which are the reason that the *top event* occurs [Fro97]. The *top event* is an undesirable outcome where the safety and reliability of a system is impaired.

1.1 Problem Statement

Creating and analyzing FTs can be a time-consuming and error-prone process when done manually without software support. A tool that automatically creates and analyzes FTs would make FTA easier and faster to apply. For the hazard analysis technique System-Theoretic Process Analysis (STPA), such a tool already exists and is called Pragmatic Automated System Theoretic Process Analysis (PASTA). It offers a DSL and further features [PKH23]. FTA analyzes component failures and their consequences, whereas STPA focuses on the unsafe communication between components. Adding FTA support for PASTA is the goal of this thesis.

1.2 Outline

The next chapter explains the foundations for the thesis, such as the basics of FTA and STPA as well as used technologies. Chapter 3 presents already existing tools that support FTA. Afterwards, Chapter 4 outlines the conceptual ideas for the developed DSL and Chapter 5 shows its implementation. The DSL is evaluated by performing FTA to a known example in Chapter 6. In the same chapter, the DSL is compared to other already existing tools supporting FTA, where the advantages and differences of other approaches are outlined. Finally, Chapter 7 concludes the thesis with a summary and possible future work.

Chapter 2

Foundations

This chapter introduces fundamental concepts for understanding the contributions of this thesis. Section 2.1 describes the basic idea of FTA and Section 2.2 briefly introduces necessary aspects of STPA. Lastly, Section 2.3 presents technologies used for the actual implementation.

2.1 Fault Tree Analysis

Fault Tree Analysis (FTA) is a systematic analysis technique that determines if a system is dependable enough in the case of hardware failure [RS15]. The goal of FTA is to analyze possible causes of undesired events, such as system-level failures, by identifying component-level failures. Analyzing a system requires translating it to a Fault Tree (FT). FTs are graphical models that are used to assess the dependability of a system.

FTA techniques can be divided into qualitative and quantitative. Qualitative FTA identifies potential causes of system-level failures, such as the failure of multiple *system components*, based on the structure of the FTs. Quantitative FTA computes values such as failure probabilities for FTs. It evaluates the likelihood of the *top event* based on the probabilities of the component-level failures.

Section 2.1.1 describes FTs and explains how a system is translated to one. After that, Section 2.1.2 and Section 2.1.3 outline qualitative and quantitative analysis techniques, respectively.

2.1.1 Constructing the Fault Tree

The Fault Tree (FT) is a tree-structured model consisting of nodes and edges. Nodes can be *system components*, *conditions*, *gates* or the *top event*. *System components* make up the system and are the leaves of the FT.

An event is the failure of a subsystem or individual component. An event is called *intermediate* when it is triggered by one or many other events, or it is called *basic* otherwise. The *top event* is the root of the tree and it is an undesired event. For example, if the power supply of a system malfunctions, it can result in a system shutdown, which can have significant consequences. Avoiding these undesired events is the purpose of the analysis.

Nodes can also be logical *gates* that have system components attached. Logical *gates* process binary input signals into binary output signals using a boolean function. This function changes depending on the type of the gate. In the context of FTA, signals are events. Gates in standard FTs can be of the following types:

- AND The output event occurs if all input events occur.
- OR The output event occurs if any of the input events occur.
- INHIBIT The output event occurs if the input event and condition occur.
- k/N The output occurs if at least k of N input events occur.

Extensions of FTA, such as dynamic FTA, provide more gates in addition to these four. An example is the *XOR* gate, where the output event occurs if only one of the inputs occur. The graphical representation of gates can be seen in Figure 2.1.

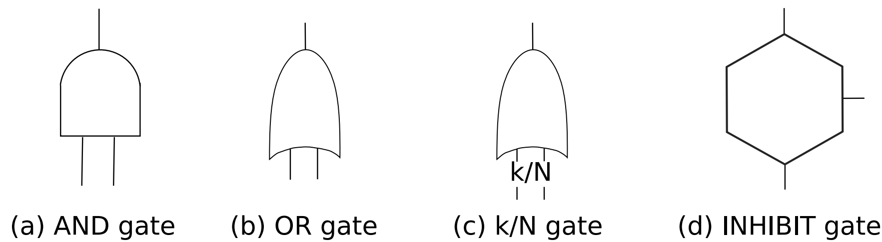


Figure 2.1. The graphical representations of gates [RS15].

Depending on the gate type, one or multiple system component failures can lead to failure propagation and eventually to system failure [RS15]. FTs are directed acyclic graphs and can be analyzed to detect vulnerabilities in the system. They graphically show how *basic* events can lead to the top event, as can be seen in Figure 2.2. The system consists of the following components: non-redundant system bus (B), power supply (PS), redundant CPUs (C1 and C2) and redundant memory units (M1, M2 and M3). They are the leaves of the tree and represent *basic* events. Additionally, the FT has the following nodes: condition (U), logical gates (G1-G6) and the top event (System Failure). Gates indicate how severe a failure is. The top event occurs when the failure propagates from the *basic* events to the root of the tree. In this case, the top event occurs when the output event of the *INHIBIT* gate G1 occurs. G1 indicates that system failure can only happen under the *condition* that the system is in use. G2 is an *OR* gate and the output event of G2 is the input event of G1. It indicates that system failure occurs if B or G3 fail and the condition U holds. In the same way, the entire tree can be analyzed. G3 is an *AND* gate with the inputs G4 and G5. Both are redundant units that need to fail for the failure to propagate through G3. G4 is a subsystem that fails when C1, PS or G6 fails and G5 fails when C2, PS or G6 fails. In Figure 2.2, PS is duplicated for layout purposes and is only meant to represent a single component. Lastly, G6 is a *k/N* gate and it fails when two of the three input events occur.

2.1.2 Qualitative Analysis

Qualitative analysis techniques aim to identify potential causes of the top event. Common qualitative analysis techniques are minimal *cut sets*, minimal *path sets* and *common cause failures*. The *cut set* computation is the focus of the DSL presented in this thesis.

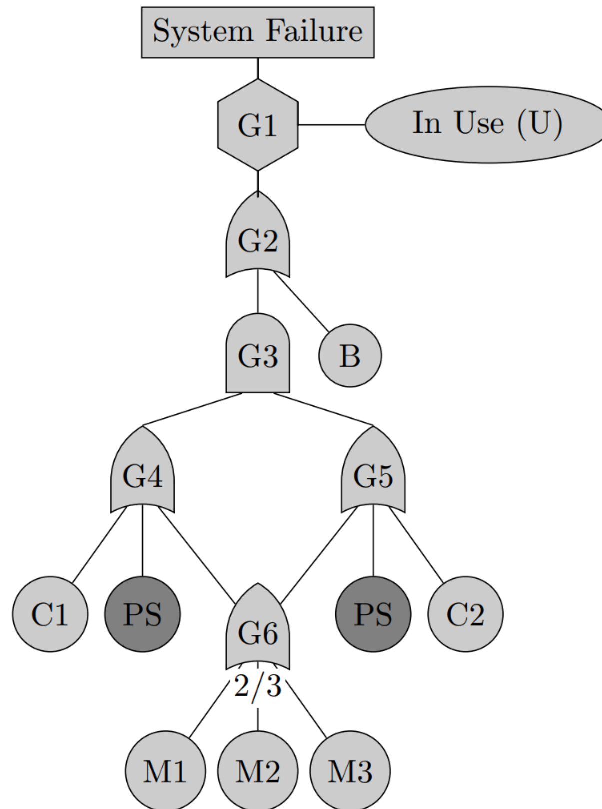


Figure 2.2. Example of an FT [RS15].

The Fault Tree (FT) can be used to compute the *cut sets*. A *cut set* is a set of components and conditions that together lead to the top event if each element in the set fails. Identifying cut sets helps in designing robust systems [RS15]. Since cut sets are vulnerabilities of a system, eliminating them improves system stability. The goal of the analysis is to compute all minimal cut sets. A cut set is minimal if by removing any component or condition from the set, it would not be a cut set anymore. All elements from a minimal cut set are necessary for the top event to occur. Knowing minimal cut sets can help in finding the most important weak points of a system, such as single points of failure, and deal with them.

There are several algorithms that can determine the cut sets of FTs. In the Binary Decision Diagram (BDD) method, FTs are converted into BDDs. A BDD is a directed acyclic graph, where the leaves are labeled with either 0 or 1 [RS15]. Except for the leaves, every node of the BDD represents a system component or condition from the FT. Nodes are labeled with a variable x_i and have two children. The path to one child ($x_i = 1$) represents the case when the parent component failed and the other ($x_i = 0$), when it did not fail. This variable is marked on the edge to the child node.

If the FT has n system components and conditions, the BDD represents a boolean function that takes n variables and returns either 0 or 1. The system fails if the function returns 1 and it will not fail otherwise. An example of converting an FT to a BDD can be seen in Figure 2.3. The FT consists of one AND gate with two OR gates attached. Each of the OR gates has two components attached.

The complexity of the BDD depends on the order of the components. Certain parts of BDDs are redundant and can be reduced. Additional parts can be reduced depending on the order of the components. Therefore, components are ordered before the conversion. In this case, the order does not matter because every component is on the same layer. Starting with component E1 as the root, two child nodes are attached. One edge represents the case when E1 failed and is marked with 1. The other edge is marked with 0 and E1 did not fail in that case. Components are now gradually attached to the BDD. The next in order is E2. E2 should be the child of both edges, but that is not the case for the edge marked with 1, as can be seen in Figure 2.3. In the FT, E1 and E2 are attached to an OR gate. If one of them fails, the failure already propagates through the gate. In other words, the failure of E2 does not matter if E1 has already failed. It then depends on the second OR gate of the FT, whether the top event occurs or not. Components E3 and E4 are attached to that gate. If E3 fails, an edge marked with 1 is created in the BDD and the path reaches a leaf labeled with 1. Otherwise, it depends on E4, whether the top event occurs or not. The same is done in the case where E1 did not fail.

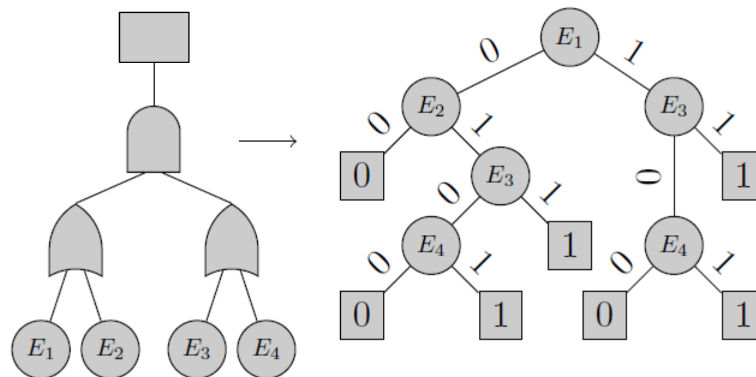


Figure 2.3. Conversion of an FT to a BDD [RS15].

2.1.3 Quantitative Analysis

Quantitative analysis computes the probability for *basic* events and the top event to occur [VCG11]. The analysis can be divided into stochastic and importance measures [RS15]. Stochastic measures compute values such as failure probabilities for a system component.

They are used to evaluate if a system should be shut down for repairs or if it can continue to operate safely with certain component failures. Importance measures determine the importance of a set of components to the reliability of the system. The reliability of a system is the probability of a failure not occurring in the lifetime of the system.

A common quantitative analysis technique is the *Expected Number of Failures*. It computes the expected number of times the top event occurs in a specified time limit. This technique is used for systems in which avoiding failures is especially important and the system is operating for a known period of time.

2.2 STPA

STPA is a hazard analysis technique that is used to develop a safe system based on the System-Theoretic Accident Model and Processes (STAMP) by Leveson [Lev14]. Accidents in STPA can be caused by either unsafe interactions between components or component failure. The results of STPA can be used in different ways, such as the creation of requirements, creating test cases and plans, etc. [LT18].

STPA can be used for any system property due to the broad definition of a *loss* in STPA. Events are considered *losses* if preventing them benefits stakeholders. Examples are the loss of life or serious injury to people. System states that under certain worst-case environmental conditions lead to losses are called *hazards*. Each *hazard* causes one or more losses under these worst-case conditions. A control structure in STPA is a system model consisting of feedback control loops. Controllers in the control loop use control actions to restrict and control a process of the system. Unsafe Control Actions (UCAs) are control actions that cause hazards in a specific context and worst-case environment. There are four types of UCAs:

- ▻ Not providing the control action leads to a hazard.
- ▻ Providing the control action leads to a hazard.
- ▻ Providing a potentially safe control action but too early, too late, or in the wrong order leads to a hazard.
- ▻ The control action lasts too long or is stopped too soon leads to a hazard.

Causal factors leading to a UCA or hazard are called *loss scenarios*. A *loss scenario* describes a situation where a UCA or hazard happens.

There are similarities between FTA and STPA. Losses need to be prevented and can be compared to the top event in FTA. *Basic* events lead to the top event in FTA and loss scenarios lead to losses in STPA.

2.3 Used Technologies

This section presents the technologies used for the implementation of the FTA-DSL. Section 2.3.1 explains Langium, which is used to define the DSL, and Section 2.3.2 outlines Sprotty, which is used for the visualization.

2.3.1 Langium

Langium[1] is an open source language engineering tool created by TypeFox[2]. The goal of Langium is to simplify the process of creating a DSL by providing a grammar language that describes the syntax and structure of the language. Three files are needed to create a DSL with Langium. A *grammar file* that describes the syntax and structure of the language, a file defining the module and a main file. The main file starts the language server and creates the connection to the client and the services defined in the module.

Grammars consist of entry, terminal and parser rules[3]. *Terminal rules* start with the keyword `terminal` and create an atomic symbol. *Parser rules* start with the name of the rule and a colon. They determine the structure of objects to be created by the parser and construct the Abstract Syntax Tree (AST), which represents the syntactic structure of the language. The following example shows the use of such a rule:

```
Component: name=ID description=STRING;.
```

This rule creates an object of type `Component` with the properties `name` and `description`. Both need to match their respective terminals `ID` and `STRING`.

The *entry rule* starts with the keyword `entry` and defines the starting point of the parsing step by matching other parser rules. Langium also offers services with default implementations, such as validation and document highlights. In the module of the language server, new services can be registered and the default implementations can be overridden with custom ones. Langium can be used in combination with Sprotty (Section 2.3.2) by defining a `DiagramGenerator` service, which translates an AST to an SGraph. If Langium is used in combination with Sprotty, the `addDiagramHandler` method needs to be called in the main file of the grammar in order to define reactions for requests sent to the language server.

2.3.2 Sprotty

Sprotty[4] is an open source web-based framework for rendering diagrams using Scalable Vector Graphics (SVG). Animations and transitions for modifying diagrams are included and the rendering is stylable with Cascading Style Sheets (CSS). Colors can be added in these CSS files and class selectors make it possible to only style elements with a specific class attribute[5].

[1] `https://langium.org/`

[2] `https://www.typefox.io/`

[3] `https://langium.org/docs/grammar-language/`

[4] `https://www.typefox.io/blog/sprotty-a-web-based-diagramming-framework`, `https://github.com/eclipse/sprotty`

[5] `https://www.w3schools.com/css/css_selectors.asp`

Sprotty can be combined with the Eclipse Layout Kernel (ELK) for an automatic diagram layout. Sprotty applications contain the client, which is implemented in TypeScript, and the optional server component. The server holds the semantic model and knows how to map it to diagram elements while the client only holds a model of the current diagram. The client renders the model and interacts with the user. Client and server communicate using JSON notifications.

Sprotty integrates with the Language Server Protocol (LSP)[6]. Since Langium does that as well, it can be combined with Sprotty[7]. Combining Sprotty with a *language server* requires a language server, a webview and the extension. The *extension* starts the language server and creates the webview. It also connects the *webview* to the language server when the diagram needs to be shown. The Sprotty client is integrated within the webview and Langium can be used for the language server.

Sprotty applications need a container module for configuring the diagram elements and a `SprottyStarter` for creating the container. Diagram elements can be configured by binding their type to a model element and a view. Sprotty also provides default implementations that can be extended to create custom model elements.

[6] `https://www.typefox.io/blog/using-sprotty-in-vs-code-extensions`

[7] `https://sprotty.org/`

Chapter 3

Related Work

Tools that simplify the application of FTA already exist. These tools offer features for improving the efficiency and support the usage of FTA. They provide assistance in the creation of FTs and automate the analysis. This chapter provides an overview of a few of these tools and their approaches.

3.1 ITEM ToolKit

The ITEM ToolKit[1] by ITEM software is a commercial software that supports FTA, as well as other reliability and safety analyses such as Reliability Block Diagrams (RBDs). The software allows users to build, analyze and evaluate FTs with a graphical interface that contains a system and a diagram window. Gates and events can be added to the diagram window manually via drag-and-drop and they will be positioned automatically. Both qualitative and quantitative analysis techniques for FTs as well as uncertainty and sensitivity analysis are supported. Uncertainty analysis can be performed if input uncertainties, such as failure rates and probabilities, are known. This analysis helps in understanding how uncertainties or variations in *basic* event probabilities can affect the result of a quantitative analysis. Sensitivity analysis can be performed when input uncertainties are not known. This analysis can determine how sensitive the computed values of a quantitative analysis are to the input data [RS15]. Understanding how changes in input parameters or data influence the result of the analysis can help identify which input parameters have the most significant impact on the outcome. The tool uses BDDs for its analysis and can also perform an approximation method [RS15]. The user can assign failure rates and probabilities to events and gates and evaluate the top event probability, cut sets and importance measure of events and gates.

ITEM ToolKit updates the diagram elements with the result of FTA and displays the results in a dialog box[2]. Additionally, it is possible to highlight a *critical path* in the diagram. A *critical path* is a group of events with the highest probability of occurrence among all sets of events. ITEM ToolKit has a wide range of features that make the software very versatile but at the same time very complex to use for new users. Additionally, a user does not get an overview of every computed minimal cut set. These are used internally for further computations. Listing every minimal cut set is useful, especially for smaller and less complex FTs. Allowing the user to select and highlight a cut set would improve the visual clarity of the FT.

[1] https://www.itemsoft.com/item_toolkit.html

[2] https://www.yumpu.com/en/document/read/53771959/item-toolkit

3.2 A.L.D. RAM Commander

A.L.D. RAM Commander[3] is a tool that specializes in reliability, availability and maintainability (RAM) analysis. The software is used to assess and improve the performance of complex systems in industries such as aerospace, energy and transportation. It performs FTA as part of its RAM Commander toolkit. It automatically generates FTs from the reliability and risk analysis techniques Failure Mode, Effect and Criticality Analysis (FMECA), Failure Mode and Effect Analysis (FMEA) [BK94] and RBDs [ČČ11]. These techniques evaluate and handle potential failures in complex systems. The FTs are then used to perform FTA. The software provides a graphical interface for modeling FTs. Users define the top event and can add events as well as gates by filling the parameters of a small data window.

RAM Commander is suitable for large-scale and critical applications, such as complex systems and models. Users of RAM Commander can generate a result report to display the results of FTA[4]. Reports such as minimal cut sets or sensitivity analysis are available in the reports menu. Similar to ITEM ToolKit, the tool is complex and offers a wide range of features that may take new users some time to understand. Additionally, manually adding every gate and laying it out can be tedious for larger FTs. RAM Commander follows a purely graphical approach. In a textual approach, creating and editing FTs is more time-efficient since the graph is generated and laid out automatically.

3.3 Galileo

Galileo is a software tool designed to support precise, modular, dynamic FTA [DSC00]. Analyzing large FTs can be impossible due to the computational complexity. That is why the tool can split complex FTs into independent subtrees. Two subtrees are independent of each other if they do not share any system components. Subtrees are analyzed and the results are combined into a result for the entire FT.

The tool enables engineers to edit and display FTs in textual and graphical form [SDC99]. The graphical view allows users to create FTs using shapes for the various gates and connectors that model the relationships between gates. The textual view describes the same FTs using a textual language, making it easier and faster to edit but more difficult to comprehend.

A gate in Galileo has the following syntax :

```
<gate name> <gate type> <component 1> ... <component n>
```

This will create and visualize a gate of the given type and it will add a box with the given name on top of the gate, as seen in Figure 3.1. The name of the gate serves as a headline for all nodes attached.

Galileo generates a result report after the analysis is finished[5]. The report detects independent subtrees and computes the unreliability for each subtree as well as for the entire tree.

[3] https://aldservice.com/RAMS-Reliability-Availability-Maintainability-and-Safety-Software.htm

[4] https://www.aldsoftware.com/download/ramc/UserManual/html/index.html?welcome_to_ram_commander.htm

[5] https://www.cse.msu.edu/~cse870/Materials/FaultTolerant/manual-galileo.htm

Unreliability refers to the probability of a subtree or tree not functioning as intended. The FTA-DSL presented in this paper offers the following alternative:

```
<gate identifier> = <component1> <gate type> <component2> ...
```

Here, an infix notation is used for the gate definition in contrast to the prefix notation used by Galileo. The infix notation makes the definition more intuitive and improves the readability. The equality operator makes it apparent that we are defining a gate and the gate type is a separator for each component.

Additionally, a headline is redundant and makes the visualization more detailed than needed. The FT will be more compact without losing essential details by removing the headline. In Galileo, the components are only listed when defining a gate. Listing every component with a description before defining the gates would make the textual view easier to understand. This will prevent users from accidentally using a new name for components already defined in a gate. Furthermore, my proposed DSL can be combined with the safety analysis technique STPA. It can generate FT templates with the results of STPA.

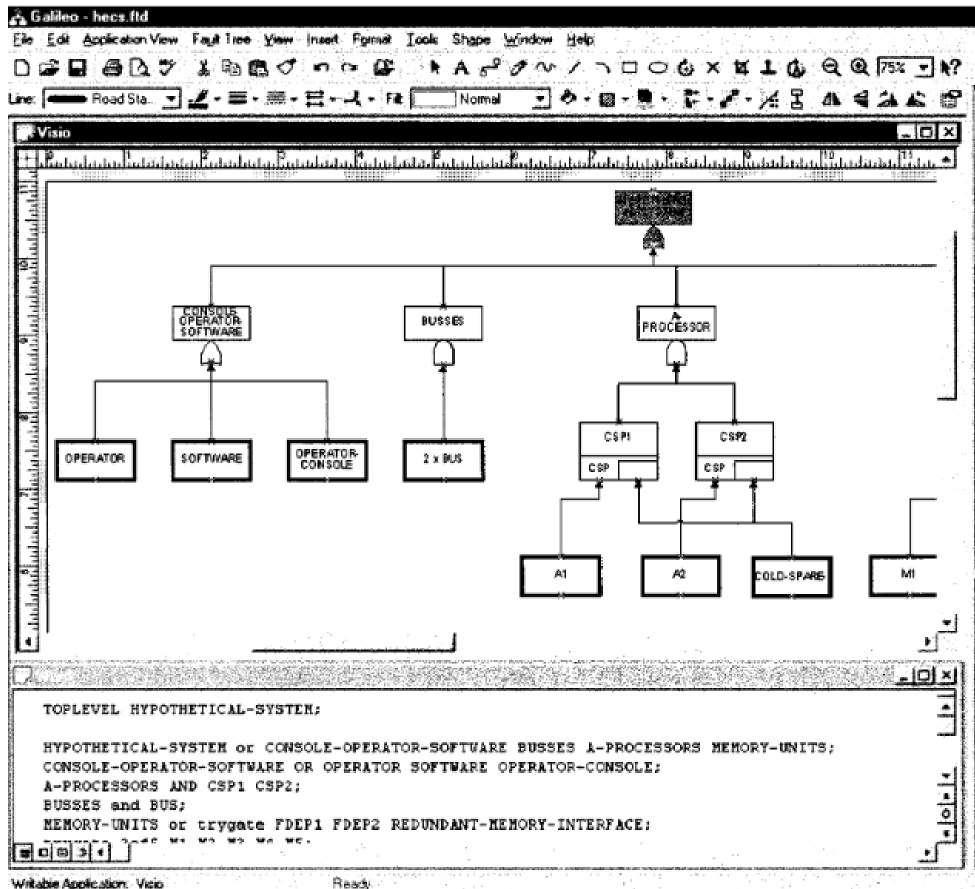


Figure 3.1. Overview of the Galileo tool [DSC00].

Chapter 4

FTA-DSL

The goal of this bachelor thesis is to develop a DSL to make FTA easier and faster to apply. There are two approaches when defining a DSL: the textual and the graphical approach [GKR+14]. In the graphical approach, the user can get an overview of all the diagram elements. However, adding graphical elements as a user is only possible via drag and drop or pop-ups in the diagram. The user needs to arrange the elements manually for a good layout. Combining the graphical with a textual approach is more time-efficient because it is possible to lay out graphical elements automatically. A DSL that uses both approaches still offers the visual clarity of the graphical approach. That is why the goal of this thesis is to define a textual DSL for FTA for which a visualization is automatically generated and laid out.

Analyzing FTs can also be easier and faster by doing it automatically. The user of the DSL can request the computation of all or just the minimal *cut sets*. Additionally, combining FTA and STPA should also be possible in the DSL but is not implemented in this thesis. After performing STPA, the user can select one of the *losses* defined in STPA and can create an FT template that can be modified. This is helpful if the user performed STPA and then wants to focus on analyzing possible reasons for the hardware failure of a system.

This chapter further explains the concepts of the DSL (Section 4.1), the visualization (Section 4.2) and the analysis of the graph (Section 4.3). Lastly, a general idea to combine FTA and STPA is presented in Section 4.4.

4.1 DSL

The main goal is a well-structured, easy-to-read and sensible grammar of the DSL such that different types of nodes of FTA are clearly separated. For that, the definitions of nodes are grouped by *components*, *conditions*, *gates* and the *top event*. Each group has its own caption,

```
1 Components
2
3 Conditions
4
5 TopEvent
6
7 Gates
```

Listing 4.1. Captions for the types of nodes in FTA.

```
1 Components
2 M1 "Redundant memory unit 1"
3 M2 "Redundant memory unit 2"
4 C1 "CPU1"
5
6 Conditions
7 U "In Use"
```

Listing 4.2. Definitions of components and conditions.

as shown in Listing 4.1. Several nodes can be defined for each group, except the top event, which can only be defined once. Some definitions of nodes reference already existing ones, which is why every node needs an ID. Components, conditions and gates can be referenced but only gates can reference other nodes.

Components and *conditions* have a description that explains their purpose and the following syntax: `<identifier> <string>`. Examples of such nodes can be seen in Listing 4.2. Here, the components M1, M2 and C1 are defined. Each of them starts with an identifier and ends with a string that describes them. *Condition* U is defined in the same way.

The definition of a *top event* takes a string that describes the undesired event and is followed by "=" as well as a reference to a gate or component. A top event can be defined as shown in Listing 4.3. Here, the string that contains the term "System Failure" describes the undesired event and becomes the label of the node in the diagram. It is followed by "=" and a reference to a gate G1. This represents an FT with system failure as the top event and an edge to the gate G1 that is directly below it.

```
1 TopEvent
2 "System Failure" = G1
```

Listing 4.3. Definition of the top event.

A node can also be of type *gate*. As explained in Section 2.1, gates can be of type *AND*, *OR*, *INHIBIT* and *k/N*. Gates also have IDs followed by "=" that signal the start of a definition. After that, the syntax varies for each gate, as can be seen in Listing 4.4.

```
1 Gates
2 G1 = U inhibits G2
3 G2 = G3 or B
4 G3 = G4 and G5
5 G4 = C1 or PS or G6
6 G5 = C2 or PS or G6
7 G6 = 2 of M1, M2, M3
```

Listing 4.4. Definitions of the gates.

For *AND* as well as *OR* gates, the definition takes a reference to a component or gate followed by the keyword `and` for AND gates or `or` for OR gates and finally another reference. This describes an AND or OR gate with two input events. More input events can be added by appending the respective keyword and a reference to a component or gate to the definition. *INHIBIT* gates take a reference to a condition followed by the keyword `inhibits` and a reference to a component or gate. Here, more references cannot be added. The *k/N* gate takes a number *k* followed by the keyword `of` and multiple references to components and gates separated with a comma. This way, the gates are easy to understand and the type of the gate can easily be determined from the syntax without explicitly defining them.

The DSL is defined in a way that is easy to understand at first glance. Even complex files need to implement the four captions to create an FT. Dividing every document into these captions improves the readability and structure of the file. Before referencing them in a gate, every component and condition needs to be defined under the respective caption. The user then knows where the references in the gate definitions originate from. Gate definitions contain already defined components and conditions and make up the structure of the FT.

Every component, condition and gate is checked to have a unique ID. If a check fails, an error is shown with a corresponding message, as seen in Figure 4.1.

This is done to give the user more visual clarity in the diagram since nodes are labeled with the *ID* of their textual counterpart. Checks are performed to prevent multiple diagram elements from having the same label.

```
Components
B "System bus"
B ""
```

(a) An example of an error message in the editor.

FaultTreeExample.fta D:\Informatik\FTADSL
Component has non-unique name 'B'.

(b) An example of an error message in the terminal

Figure 4.1. Visualization of failed checks.

4.2 Visualization

After creating the FT textually, the user can request an automatically generated diagram. A screenshot with a small textual example and the automatically generated diagram is shown in Listing 4.5. It shows how the DSL models the FT presented in Section 2.1.1. The visualization tab contains a graph representing the FT. That graph is laid out automatically with the layered algorithm from the Eclipse Layout Kernel (ELK). ELK[1] offers layout algorithms and an infrastructure that connects diagram editors to these algorithms. ELK computes positions and dimensions for diagram elements and it can be used in JavaScript with the elkjs library[2]. The layered algorithm sorts the nodes of a graph into layers. For example, in Listing 4.5 the top event is in the top-most layer and M1, M2, M3 are in the bottom-most layer. It aims to

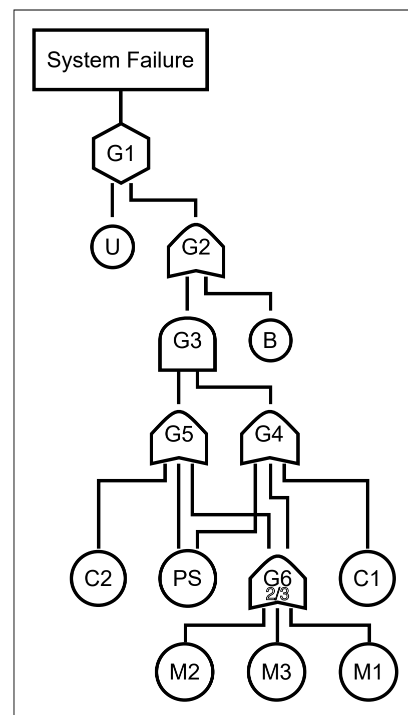
[1] https://projects.eclipse.org/projects/modeling.elk

[2] https://github.com/kieler/elkjs

improve the readability of the graph by reducing edge crossings. The top event is the root of the FT and every node is placed on a layer under its parent. Conditions are placed one layer below the INHIBIT gate that they are attached to and not next to it on the same layer as in other tools. This way, the similarities to an AND gate become more apparent.

Each FTA node of the graph has a shape that depends on its node type. A condition or component is drawn as a circle. They share the same shape because both of them cannot reference other nodes. The top event is the only node drawn as a rectangle to emphasize its uniqueness. Lastly, gates are drawn based on their gate type, as described in Section 2.1.1. The nodes of the FT in Listing 4.5 have the described shapes and are positioned automatically. Edges are clearly spaced and only cross once in the diagram due to the layered algorithm. However, even in that case the readability of the graph is not compromised.

```
1  Components
2  M1 "Redundant memory unit 1"
3  M2 "Redundant memory unit 2"
4  M3 "Redundant memory unit 3"
5  C1 "CPU1"
6  C2 "CPU2"
7  PS "Power supply"
8  B "System bus"
9
10 Conditions
11 U "In Use"
12
13 TopEvent
14 "System Failure" = G1
15
16 Gates
17 G1 = U inhibits G2
18 G2 = G3 or B
19 G3 = G4 and G5
20 G4 = C1 or PS or G6
21 G5 = C2 or PS or G6
22 G6 = 2 of M1, M2, M3
```



Listing 4.5. Example of an FT in the DSL.

4.3 Analysis

The user can request the automatic computation of all or just the minimal *cut sets* of an FT. The cut sets of an FT are determined recursively with the node referenced by the top event as the starting node. Sets of components and conditions are computed and assigned for each recursively referenced node of the starting node. Depending on the type of the gate they are attached to, the computed sets of all the referenced nodes are combined differently. The

computed sets assigned to the starting node are the cut sets of the FT. Chapter 5 explains this in more detail.

All requested cut sets are printed in the console after computation. Listing 4.6 shows the minimal cut sets for the FT in Listing 4.5.

```
1 The resulting 6 minimal cut sets are:
2 [[U,C2,C1],
3 [U,PS],
4 [U,M2,M1],
5 [U,M3,M1],
6 [U,M3,M2],
7 [U,B]]
```

Listing 4.6. Minimal cut sets of the example.

After computing the cut sets, a dropdown menu appears on the sidebar of the diagram containing all cut sets. When the user selects a cut set in the menu, the components and conditions in the set are highlighted in the diagram. Each node in the set as well as every node and edge in their path to the top event is highlighted as well. Nodes and edges unrelated to the cut set become transparent, as can be seen in Figure 4.2. This way, the user can understand how a set of components and conditions lead to the top event in the FT.

The components C1 and C2 as well as the condition U are highlighted. Gates and edges that are on the path of these nodes to the top event are also highlighted. Every other node and edge becomes transparent. This shows that the set containing C1, C2 and U is a selected cut set. The set is also minimal because it is not a cut set anymore if any node is removed.

4.4 Combining FTA with STPA

FTA focuses on identifying component or subsystem failure and STPA identifies potential vulnerabilities in the system design and operation. Both techniques offer complementary approaches to safety analysis. Combining both techniques allows analysts to understand and develop safety measures more effectively. STPA identifies hardware issues and FTA further analyzes these issues.

That is why a feature of the DSL is the ability to combine FTA with STPA. The following is a possible idea to combine them. However, it is not implemented in the DSL.

Combining them can be done by performing FTA on the result of STPA. After performing STPA, the user can select one of the *losses* in the diagram. A new file is then created and a new diagram is generated in the visualization tab. In the file, an FT is created with the selected loss as the top event. Every *hazard* that, per definition, leads to that loss is added to an OR gate, which is attached to the top event. Another OR gate is attached to each hazard. Every *loss scenario* that leads to the hazard, if there are any, is attached to this gate. Additionally, every Unsafe Control Action (UCA) that causes the hazard is also attached to the gate. Loss scenarios lead not only to hazards but also to UCAs. That is why every corresponding loss scenario is added to a UCA in the form of an OR gate.

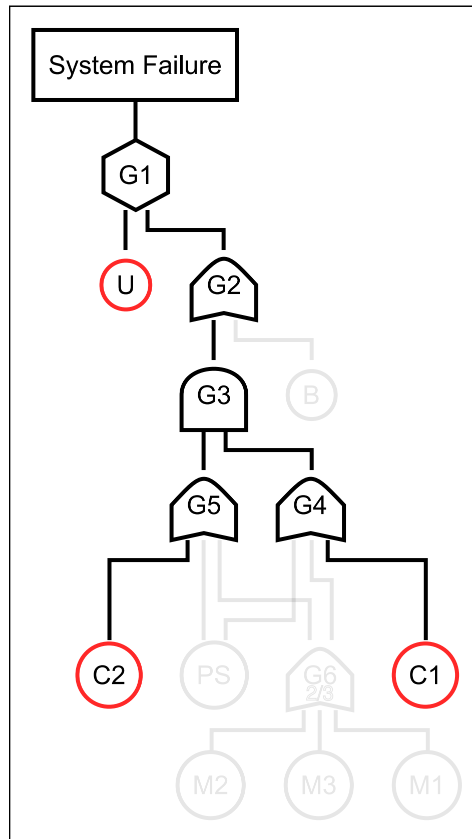
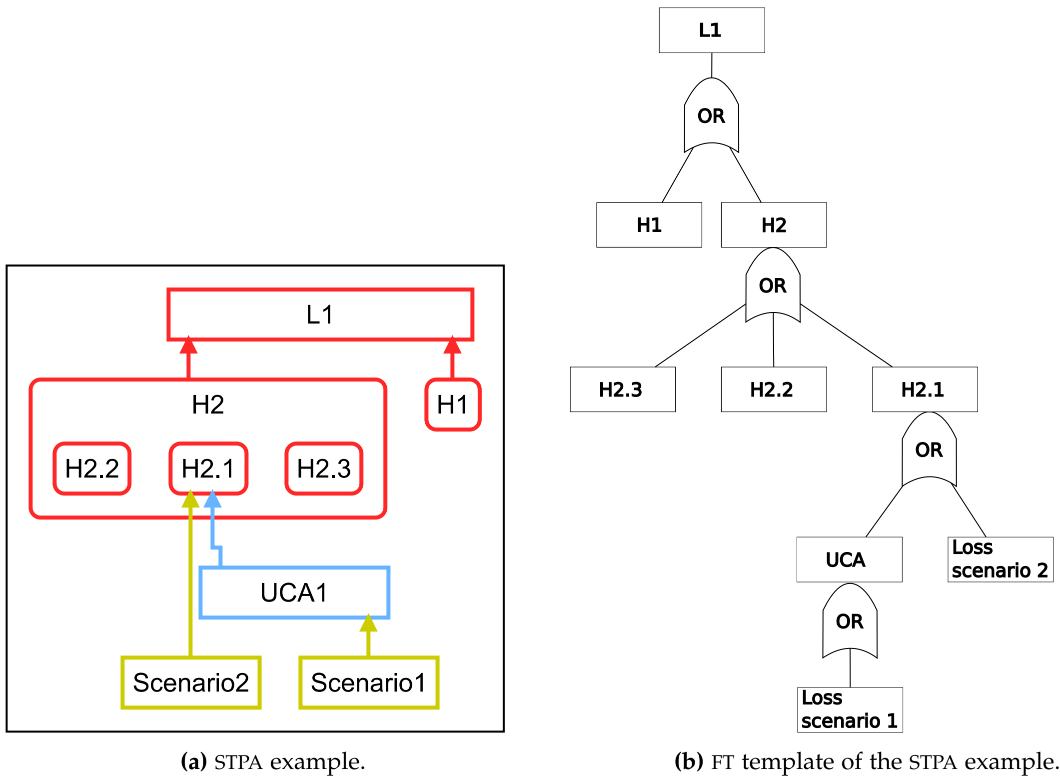


Figure 4.2. Highlighting a cut set in the FT.

This can be seen in Figure 4.3 with an example from the STPA handbook [LT18]. Figure 4.3a shows the generated diagram of the example from the handbook. It is about possible hazards and losses when using aircraft. There is loss L1, which represents the loss of life or serious injury. The loss of aircraft control and the aircraft coming too close to other objects are hazards H1 and H2. H2 has three *sub-hazards* H2.1, H2.2 and H2.3. H2.1 states that the deceleration of the aircraft is insufficient. H2.2 and H2.3 describe the cases in which the aircraft is asymmetrically or excessively accelerated. Additionally, there is a UCA that causes the sub-hazard H2.1. It describes that the crew does not provide a power off for a system component when abnormal behavior occurs. Finally, there are two loss scenarios, with one leading to sub-hazard H2.1 and the other leading to the UCA that causes H2.1.

Figure 4.3b shows how a template for this example could be generated automatically by selecting L1. L1 is the top event of the generated FT with H1 and H2 being attached to an OR gate below L1. The OR gate below H2 references H2.1, H2.2 and H2.3. An OR gate below H2.1 references the UCA and a loss scenario. Lastly, the UCA has an OR gate that references the other loss scenario.

The described FT will be created textually in the editor. This serves as a template and can be edited by the user for the desired FT. Furthermore, the user can add details by adding more gates to the loss scenarios. It will automatically generate a diagram in the visualization tab and editing the FT textually will update the diagram. Currently, labels cannot be placed on top of gates in the DSL. This needs to be added together with this feature.



(a) STPA example.

(b) FT template of the STPA example.

Figure 4.3. Combination of FTA and STPA.

Chapter 5

Implementation

In this chapter, the implementation of the previously explained approach is presented as a VSCode Extension. The extension uses Langium (Section 2.3.1) for the DSL and Sprotty (Section 2.3.2) for generating the visualization. There already exists a VSCode Extension for STPA, which is called Pragmatic Automated System Theoretic Process Analysis (PASTA) [PKH23]. PASTA is the basis for the implementation presented in this section.

The FTA-DSL consists of a language server and a webview, which are explained in Section 5.1 and Section 5.2. The `language server` contains the actual DSL, the generation of a graph, the layout configuration for the layout of the graph and the analysis (Section 5.3) of the graph. In the `webview`, the visualization of the graph is generated, which can be shown to the user alongside the editor.

5.1 Language Server

The language server of the extension is based on Langium. As explained in Section 2.3.1, in order to create a Langium-based language server, the following files are needed: a grammar, a module and a main file. In the DSL, the module of FTA is called `FtaModule` and is defined in the *fta-module* file. Services registered in this module are the following: `FtaValidator`, `FtaDiagramGenerator`, `FtaLayoutConfigurator` and the `CutSetGenerator`. The `FtaValidator` improves the utility of the DSL by ensuring that every element defined in the editor has a unique ID. The `FtaDiagramGenerator` and the `FtaLayoutConfigurator` create the diagram elements and layout them properly. They are essential for the combination with Sprotty. Finally, the `CutSetGenerator` generates all cut sets of the current FT and displays them in the webview.

Section 5.1.1 explains the implementation of the DSL including grammar and validation and the implementation of the diagram generation is explained in Section 5.1.2

5.1.1 DSL

The grammar of the DSL is defined in Langium. In Langium, the language server can be implemented in TypeScript, which is also used for VSCode Extensions and Sprotty. Development and maintenance are easier with a single programming language.

The entry rule is named `ModelFTA` to not overlap with the entry model of STPA. It contains parser rules for every type of node in FTA. Parser rules are defined for every type of node in FTA based on the concept for the grammar (Section 4.1). For example, the rule for *components*

is defined as follows: `Component: name=ID description=STRING;`. Cross-references are used in the grammar to reference other nodes in a rule.

The `FtaValidator` overrides the default validator implementation by Langium and can only be used after the `ValidationRegistry` is overridden. The `FtaValidationRegistry` overrides the default `ValidationRegistry` and registers a method that contains a relevant check for the model. This method is implemented in the `FtaValidator` and it generates an error message when the check fails. It guarantees that all conditions, components and gates have unique IDs.

5.1.2 Diagram Generation

Sprotty needs an `SGraph` to visualize the FT. An `SGraph` is created with the `FtaDiagramGenerator` class, which contains the `generateRoot` method. This method creates the `SModelRoot` using the document the user edited. An overview of the classes used to generate the `SGraph` can be seen in Figure 5.1.

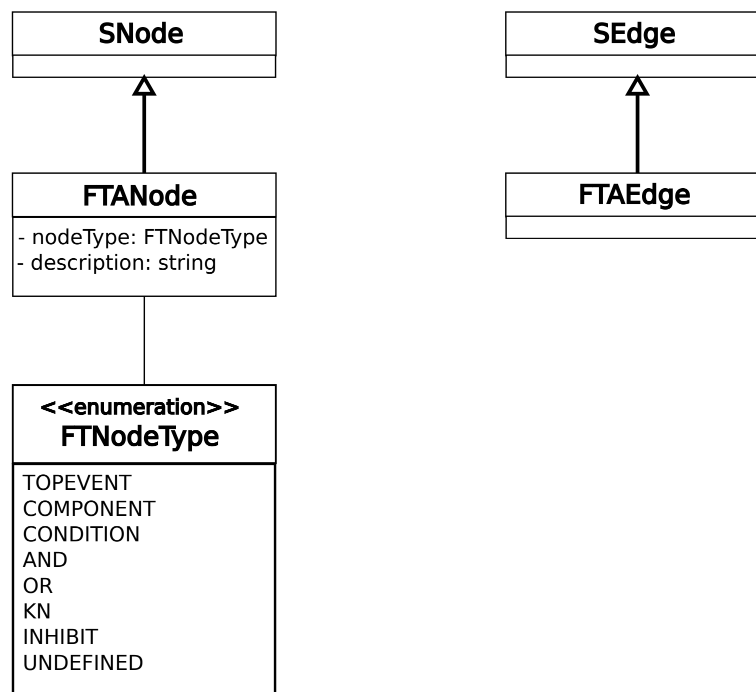


Figure 5.1. The classes for the graph elements.

The graph is created by translating each element, defined in the editor, into an `FTANode`. An `FTANode` extends the `SNode` class with the properties `nodeType` and `description`. The `nodeType` has the type `FTNodeType`, which is an enum with a value for each type of node in standard FTs. It is determined by examining the definition of the element. The `description` contains the description for components and conditions stated by the user and is empty for gates and

the top event. An FTAEdge is created between two FTANodes if one references the other. The FTAEdge class extends the SEdge class.

After the graph is generated, it needs to be laid out. As mentioned in Section 4.2, elkjs is used for the layout. A DefaultLayoutConfigurator is provided by Langium. It can be extended to configure the layout with elkjs. The FtaLayoutConfigurator class contains layout options for the FTA graphs. The direction of the graph is set to down. As explained in Section 4.2, the graph is laid out with the layered algorithm from ELK. It sorts the nodes of the graph into layers and positions them in the layers. A layered approach is well-suited for laying out FTs due to their hierarchical and tree-like structure. Here, the top event is the root and the top of the tree. Each node is positioned below the gate or top event that references it. The aim of the layered algorithm is to minimize the number of edge crossings. This can help to maintain visual clarity for complex FTs.

5.2 Visualization

The webview uses Sprotty for the visualization since it can be used in combination with Langium and the DSL already uses Sprotty to visualize STPA. It is sensible to use Sprotty for FTA as well. As explained in Section 2.3.2, a container module is needed. There already exists one in the extension where the STPA diagram elements are configured. The FTA elements can be added to this STPA container module. Diagram elements are configured by assigning a view class that renders them. Every element in Figure 5.1 needs to be configured this way. The container is created by the StpaSprottyStarter class, which extends SprottyStarter.

Additionally, diagram elements can be styled using Cascading Style Sheets (CSS). This is done in the ftaDiagram.css file for the FTA diagram elements. In that file, the colors of nodes and edges are defined based on the color theme of VSCode. For that, class selectors are used, which are introduced in Section 2.3.2. If an element has the fta-node class, the fill and stroke colors are set based on the color theme. The stroke color of edges is also set based on the color theme and whether the fta-edge class was assigned or not. Furthermore, by assigning the fta-hidden class to nodes and edges, the opacity of the element is set to 0.1. At the same time, nodes that do not have the fta-hidden class assigned appear highlighted. If an element has the fta-highlight-node class, the stroke color is set to red to further highlight the node. The fta-highlight-node class is assigned to nodes that are in the currently selected cut set.

In the fta-views file, the different views for the diagram elements are implemented. For the edges, path is used for the line. The element created for an FTANode is based on the type of the node. In the view-rendering file, each type of node has a render method, as explained in Section 4.2. In addition to the methods for the STPA elements, the file also contains methods for rendering *AND*, *OR*, *k/N* and *INHIBIT* gates.

5.3 Analysis

The Binary Decision Diagram (BDD) method presented in Section 2.1 is a graph-based approach that visualizes logical dependencies. A path in the graph where the leaf is labeled with a 1 is a cut set of the FT. BDDs are compact and can efficiently describe complex logical relationships with a simple graph. They are suitable for analyzing large FTs with numerous components because they reduce redundant nodes and edges in the graph. However, this approach requires the translation of the FT to a BDD. Additionally, algorithms to reduce the graph increase the runtime.

The following approach of the DSL is an alternative to the BDD method, where cut sets are computed recursively. Cut sets are computed from the FT directly and do not need to be translated to a BDD first. Both approaches describe how component failures and the top event are connected. They determine cut sets to explain how failure propagates through the system.

The analysis of the FT in the DSL is done in the `CutSetGenerator`. The `CutSetGenerator` contains the `generateCutSets` method, which computes the cut sets of the FT recursively. It starts with the node that is attached to the top event. If that node is a component, then the cut sets of the FT only contain a single component. Otherwise, the node has to be a gate and the algorithm calculates and evaluates every successor node of that gate. Successor nodes are the nodes that are referenced in the definition of a gate. In the evaluation, it is checked if the successor has successors too. In that case, the same procedure is done recursively until the successor is a leaf of the tree and therefore a component or condition. Here, the cut sets of each gate are computed starting from the bottom. In this context, a cut set is a set of components or conditions in which each element of this set needs to fail for the output event of a gate to occur. In other words, the failure propagates through the gate. Depending on the type of the gate, the cut sets of all successors are combined differently. The combination is assigned to the gate and represents the cut sets. For example, if components M1 and M2 are attached to an AND gate, then the output event will occur only if both M1 and M2 fail. That is why the evaluation of the gate results in a single list containing both components. Going further, if this AND gate and a component C are attached to an OR gate, then the output event of the gate will occur if either M1 and M2 or C fails. That is why the resulting cut sets of the OR gate will be two lists. One containing both M1 and M2 and the other containing only C. This can be seen in Figure 5.2.

The `CutSetGenerator` also contains the `determineMinimalCutSets` method. First, that method generates all cut sets with the `generateCutSets` method. After that, a filter checks every single cut set and determines if it is minimal or not. That is done in the `checkIfMinimalCutSet` method. This method checks whether the currently examined cut set contains any other cut set. If that is the case, the set is not minimal because there is a cut set that contains fewer elements. Otherwise, the cut set is minimal and will be added to the result list.

Every requested cut set is printed in the console after computation. This is done in the `language-extension` file. A request is sent to the language server when the user wants to compute the cut sets. These are handled in the `fta-message-handler` file. In that file, the computed cut sets are used to create a string that contains a new line after every cut set. This

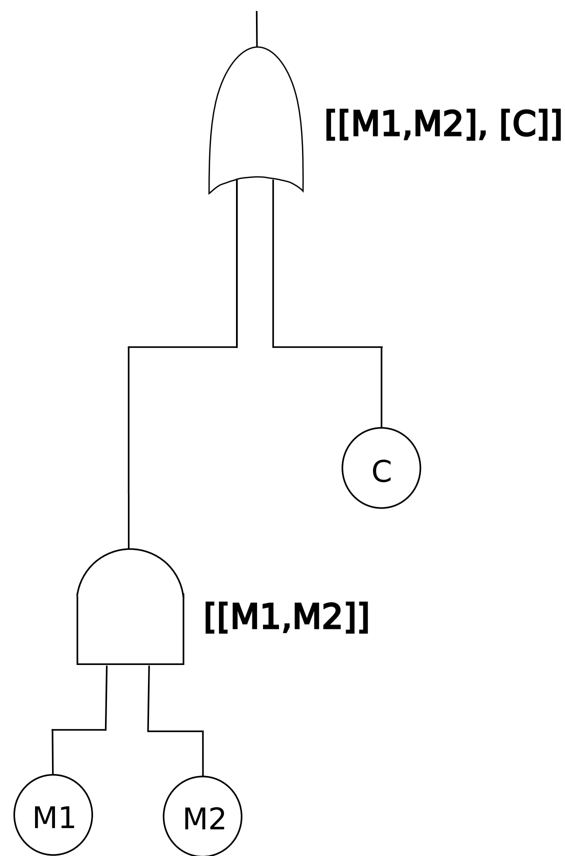


Figure 5.2. Cut sets of an example FT.

can be seen in Listing 4.6. The request returns the string to the `language-extension` file and it is printed.

Additionally, the language server sends the cut sets to the webview, where they are handled in the `CutSetRegistry` class. After receiving the cut sets, a dropdown menu is created with an entry for each of them. This dropdown menu is added to the sidebar of the DSL. In the `fta-views` file, the webview checks if an entry in the dropdown menu is currently selected. In that case, the webview iterates over every node and checks whether it is in the currently selected entry of the dropdown menu. The diagram is highlighted, as described in Section 4.3.

Chapter 6

Evaluation

In this chapter, the pressure tank example is introduced and the DSL is used to model it and compute the cut sets of the example (Section 6.1). Furthermore, the DSL is compared to other tools supporting FTA (Section 6.2).

6.1 Pressure Tank Example

This example involves a system containing a pressure tank that is used to store a liquid or gas by regulating the operation of a pump [VGR+81]. The system has a control part and an electrically isolated circuit that energizes the motor of the pump. A push button starts the system when pressed. If the pressure tank is empty, the pressure switch is closed and therefore the electrical circuits start the motor of the pump. It takes approximately one minute to fill the tank. When the tank is full, the pressure switch opens, which de-energizes the electrical circuits and turns off the pump. An outlet valve drains the tank until it is empty and the system can repeat the process. Thus, the pressure switch is opening and closing depending on the liquid level of the tank. A timer prevents tank rupture when the pressure switch is defective. The timer sends a timeout when the pressure switch fails to open and the tank is full. If the circuit that powers the motor of the pump is energized for longer than one minute, the timer provides an emergency shutdown by de-energizing it.

The goal is to analyze the potential failures of the pressure tank that could lead to a system-level failure. For that, system failure is the top event of the FTA. The pressure tank example can be modeled with the FTA-DSL. Listing 6.1 defines a simplified version by relabeling the components and gates. It is noticeable that every component only occurs once. Respectively, the generated FT for Listing 6.1 can be seen in Figure 6.1. Most of the gates are positioned on a new layer due to the structure of the FT. Components *One* to *Five* can cause system failure if any of them fail. For the rest, it takes a combination from Components *Six* to *Eight* and *Components Nine* to *Sixteen*.

```
Components
One ""
Two ""
...
Fifteen ""
Sixteen ""

TopEvent
"System Failure" = G0

Gates
G0 = One or Two or G1
G1 = Three or G2
G2 = Four or Five or G3
G3 = G4 and G5
G4 = G6 or G7
G5 = Six or Seven or Eight
G6 = Nine or Ten or Eleven
G7 = Twelve or Thirteen or G8
G8 = Fourteen or Fifteen or Sixteen
```

Listing 6.1. Modeling the pressure tank example in FTA [VGR+81].

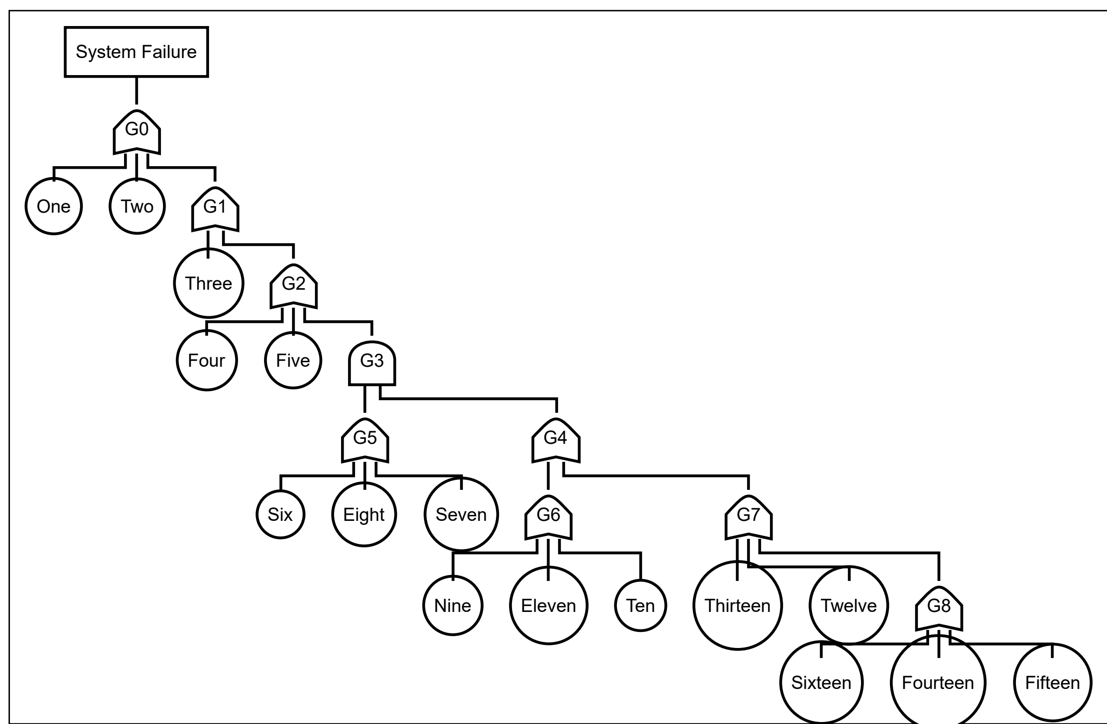


Figure 6.1. Pressure Tank in the FTA-DSL [VGR+81].

Computing the cut sets of the pressure tank example results in 29 cut sets as can be seen in Listing 6.2. The algorithm in the paper also computed these 29 cut sets [BC73]. In this example, all of the computed sets are also minimal. Generally, the algorithm applied in the paper will also compute non-minimal cut sets. That is why all of the resulting cut sets need to be filtered so that non-minimal sets are removed. The FTA-DSL can compute all or only the minimal cut sets depending on the used command.

```
The resulting 29 minimal cut sets are:
[[One],
[Two],
[Three],
...
[Fourteen, Eight],
[Fifteen, Eight],
[Sixteen, Eight]]
```

Listing 6.2. Cut sets of the pressure tank example.

6.2 Comparison

Compared with other tools such as ITEM ToolKit, the DSL only supports standard FTs and fewer event types. Many tools lack support for the *INHIBIT* gate due to its similarity to the *AND* gate but the DSL includes support for it. Therefore, in addition to the components, it is possible to add conditions in the DSL. *INHIBIT* gates are connected to one component and condition each. Additionally, the DSL supports the *AND*, *OR* and *k/N* gate. However, it does not support the *PRIORITY*, *XOR*, *NOT* and *NULL* gate. Nevertheless, these can be provided in future work.

Creating and editing FTs graphically can be tedious, even if tools such as ITEM ToolKit provide automatic node placement. That disadvantage does not exist in the DSL. The DSL automatically generates a visualization and lays it out. Editing and aligning the positions of nodes can be time-consuming if done manually. Changing the position of a node can require editing the entire graph. These are handled automatically in the DSL. When the user changes the FT textually, the entire layout of the FT is restructured.

Other tools do not show the user the computed cut sets and just use them internally. In contrast to the other tools, the DSL displays all computed cut sets and enables the user to highlight one of them specifically. Every node in the selected cut set as well as all nodes and edges that connect them to the top event are highlighted. The DSL computes and then filters them to get all minimal cut sets. However, the runtime can be reduced by computing the minimal cut sets directly. This can also be provided in future work.

All in all, the main advantage of the DSL is its simplicity as well as the automatic visualization and layout of the FT. Furthermore, another advantage of the DSL is the computation and visualization of the cut sets in the diagram as well as the combination of FTA and STPA.

Chapter 7

Conclusion

This chapter serves as a summary for the contributions presented in this thesis. It provides an overview of FTA and explains the presented DSL. Afterwards, it outlines the results of the evaluation of the DSL. Additionally, this chapter describes possible features for future work of the DSL.

7.1 Summary

Fault Tree Analysis (FTA) is a risk analysis technique that assumes basic *component* failure or other events to be the cause of undesired events. For example, an undesired event can be a system shutdown due to the power supply of the system malfunctioning. These events can have significant consequences and need to be avoided, which is the purpose of FTA. In this thesis, FTA support is added in PASTA. The DSL is implemented as a VSCode Extension with Langium and Sprotty. An editor in which Fault Tree (FT) elements can be defined is provided by the extension. A graph is generated based on the defined elements and is laid out automatically using ELK. FTs are hierarchically structured graphs that have an undesired event, which is called *top event*, as the root of the tree. Logical *gates* then further split this event into *basic* events, which represent the failure of a system component. Additionally, a validator is implemented to improve the utility. The goal of FTA is to determine how certain combinations of *basic* events can cause the top event. That is done with an analysis of the FT, which can be performed automatically with the DSL. Here, the user can choose to compute all *cut sets* or just the minimal ones. After computation, the user can select one of the computed cut sets. The selected set as well as every node and edge on the path to the top event is highlighted in the graph. Furthermore, this thesis provided a general idea on how to combine FTA with STPA. The results of STPA can be used to generate a template of an FT that depicts these results. Users can edit and extend this template to create the desired FT.

The evaluation of the DSL resulted in clear strengths and weaknesses. Its main advantages are the automatic visualization and layout of the FT as well as the computation and visualization of the cut sets in the diagram. The main disadvantage is that the DSL only supports standard FTs, which is why only a few events and gate types are supported.

In conclusion, the DSL supports and simplifies the application of FTA. Analysts can visualize the defined FT and then compute the cut sets. Therefore, the DSL is very suitable for showcasing and explaining FTA.

7.2 Future Work

Several features can be added to the DSL that can improve the utility and simplify the application of FTA. One of them is allowing and supporting more of the quantitative and qualitative analysis techniques, such as the *Expected Number of Failures* or computing the *critical path sets*. Additionally, failure probabilities for components can be defined [RS15]. The underlying idea of quantitative analysis techniques is that every component will fail at some point in time. These probabilities describe the likelihood of that failure. The DSL could also support more than standard FTs such as dynamic FTs. This way, more types of gates will be supported, such as the *PRIORITY*, *XOR*, *NOT* and *NULL* gate. As stated in Section 6.2, the runtime can be reduced by computing the minimal cut sets without calculating all cut sets first. The DSL can be improved to inform the user about all of the single points of failure in the computed minimal cut sets. They are the most important weak points of the system and need to be eliminated with the highest priority among all cut sets.

The clarity of the diagram can be improved by allowing the user to select whether the labels of gates should be shown or not. They are the identifiers of the nodes but do not offer essential information since the shape of the elements represents the gate type.

Another feature that can be implemented is the combination of FTA and STPA as explained in Section 4.4. For this, it should be possible to define an OR gate with just a single reference to a component or gate. Furthermore, the user should be able to add a label to a gate, which is visible on top of it in the diagram. In addition, the gates in the DSL can be improved to reference labels. The user can also add gates to the labels. However, labels are only optional and the user should be able to delete them for more visual clarity. They provide advantages when combining FTA and STPA but are not needed in FTA.

Another feature that allows the user to select a sub-branch of the tree and analyze it separately can be introduced. This way, analysts could isolate and identify potential problems in larger FTs more quickly.

Bibliography

[AR10] Terje Aven and Ortwin Renn. *Risk management and governance: concepts, guidelines and applications*. Vol. 16. Springer Science & Business Media, 2010.

[BC73] Richard E Barlow and Purnendu Chatterjee. *Introduction to fault tree analysis*. University of California Press Berkeley, CA, 1973.

[BDJ07] Koen Buyens, Bart De Win, and Wouter Joosen. "Empirical and statistical analysis of risk analysis-driven techniques for threat management". In: *The Second International Conference on Availability, Reliability and Security (ARES'07)*. IEEE. 2007, pp. 1034–1041.

[BK94] Abdelkader Bouti and Daoud Ait Kadi. "A state-of-the-art review of fmea/fmeca". In: *International Journal of reliability, quality and safety engineering* 1.04 (1994), pp. 515–543.

[ČČ11] Marko Čepin and Marko Čepin. "Reliability block diagram". In: *Assessment of Power System Reliability: Methods and Applications* (2011), pp. 119–123.

[DSC00] Joanne Bechta Dugan, Kevin J Sullivan, and David Coppit. "Developing a low-cost high-quality software tool for dynamic fault-tree analysis". In: *IEEE Transactions on reliability* 49.1 (2000), pp. 49–59.

[Fro97] Steve Frosdick. "The techniques of risk analysis are insufficient in themselves". In: *Disaster Prevention and Management: An International Journal* 6.3 (1997), pp. 165–177.

[GKR+14] Hans Grönninger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. "Textbased modeling". In: *arXiv preprint arXiv:1409.6623* (2014).

[Lev14] Nancy G Leveson. "Using stamp to develop leading indicators." In: *GI-Jahrestagung*. 2014, pp. 597–600.

[Lev20] Nancy Leveson. "Are you sure your software will not kill anyone?" In: *Communications of the ACM* 63.2 (2020), pp. 25–28.

[LT18] Nancy Leveson and John P Thomas. "STPA Handbook". In: *MIT Partnership for Systems Approaches to Safety and Security (PSASS)* (2018).

[PKH23] Jette Petzold, Jana Kreiß, and Reinhard von Hanxleden. "Pasta: pragmatic automated system-theoretic process analysis". In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2023, pp. 559–567.

[RS15] Enno Ruijters and Mariëlle Stoelinga. "Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools". In: *Computer science review* 15 (2015), pp. 29–62.

[SDC99] Kevin J Sullivan, Joanne Bechta Dugan, and David Coppit. "The galileo fault tree analysis tool". In: *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*. IEEE. 1999, pp. 232–235.

[VCG11] Johanna AE ten Veldhuis, Francois HLR Clemens, and Pieter HAJM van Gelder. "Quantitative fault tree analysis for urban water infrastructure flooding". In: *Structure and Infrastructure Engineering* 7.11 (2011), pp. 809–821.

[VGR+81] William E Vesely, Francine F Goldberg, Norman H Roberts, David F Haasl, et al. *Fault tree handbook*. Systems and Reliability Research, Office of Nuclear Regulatory Research, US . . ., 1981.

[VK09] David Valis and Miroslav Koucky. "Selected overview of risk assessment techniques". In: *Problemy eksploatacji* 4 (2009), pp. 19–32.

List of Abbreviations

DSL Domain Specific Language

STPA System-Theoretic Process Analysis

ELK Eclipse Layout Kernel

FMEA Failure Mode and Effect Analysis

FMECA Failure Mode, Effect and Criticality Analysis

RBD Reliability Block Diagram

BDD Binary Decision Diagram

FTA Fault Tree Analysis

FT Fault Tree

VSCode Visual Studio Code

STAMP System-Theoretic Accident Model and Processes

AST Abstract Syntax Tree

SVG Scalable Vector Graphics

CSS Cascading Style Sheets

LSP	Language Server Protocol
UCA	Unsafe Control Action
PASTA	Pragmatic Automated System Theoretic Process Analysis