Context Tables in the System-Theoretic Process Analysis Domain Specific Language

Jana Kreiß

Bachelor Thesis
September 28, 2022

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

System-Theoretic Process Analysis (STPA), developed by Leveson, is a relatively new risk analysis procedure used to analyze systems during their development. The aim of STPA is to find possible safety flaws before the systems are brought into operation, in order to minimize the potential risks of accidents or losses caused by the systems. While the procedure is able to find all kinds of safety flaws in a system's design, it is tedious to conduct on paper. Because of this, Petzold developed a Domain Specific Language (DSL) as a Visual Studio Code (VS Code) extension that fully supports STPA to offer an alternative that makes the procedure less time-consuming and more manageable.

This thesis introduces the implementation of context tables into the mentioned DSL. The context tables act as visual support for analyzing a system's structure for system behavior that can, under worst-case circumstances, be leading to the risks that are aimed to be prevented. In the DSL, the context tables can be displayed in a view next to the STPA file editor provided by the extension. The tables can be edited by defining Rules, which have been implemented into the DSL's grammar. A Rule describes conditions for unsafe system behavior, which are immediately integrated into the context tables as soon as the Rule has been fully defined, updating their results.

It is concluded that the tables offer options to further optimize the DSL in regard to time-efficiency and user-friendliness. However, improvements can still be made, for example by implementing a feature to automatically generate results from the tables.

Acknowledgements

First, I want to thank my advisor Jette Petzold for offering me to work on her DSL and giving me the topic and motivation for my thesis. Moreover, without her help getting me acquainted with the DSL, her continuous feedback to any progress I made and her advice and provided resources to help me along the way, the conception of this thesis would not have been possible.

Next, I want to thank Prof. Reinhard von Hanxleden for giving me the opportunity to write this thesis, offering me thorough feedback on progress I presented as well as help with my thesis presentation when I needed it.

Furthermore, I owe gratitude to the Real-Time and Embedded Systems Group for welcoming me as a member of the group and inviting me to several enjoyable events. The encouragement and feedback regarding my thesis was also much appreciated.

Finally, I must thank my sister and my family for giving me unwavering support throughout my studies. You helped me make decisions without which I would not have been able to do any of this, and I will always be grateful for all that you have done to help me.

Contents

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

2 Foundations **3**
2.1 STPA . . . 3
2.2 The STPA DSL . . . 4
2.2.1 Extension . . . 4
2.2.2 Language Server . . . 5
2.2.3 Hazards . . . 6
2.2.4 Control Structure . . . 7
2.2.5 Unsafe Control Actions . . . 8
2.3 Context Tables . . . 9
2.4 Rules . . . 11

3 Related Work **13**
3.1 XSTAMPP . . . 13
3.2 WebSTAMP . . . 13
3.3 An STPA Tool . . . 15

4 The Context Table Web-View **19**
4.1 The View . . . 20
4.2 Context Tables . . . 20
4.3 Rules . . . 21

5 Implementation **23**
5.1 Context Table View . . . 23
5.2 Assembling the View's Content . . . 23
5.3 Language Server Communication . . . 25
5.4 Rules . . . 25

6 Conclusion **29**
6.1 Summary . . . 29
6.2 Future Work . . . 29

Bibliography **31**

List of Abbreviations **33**

List of Figures

2.1 The STPA extension. 5
2.2 The same control structure as in Listing 2.3 depicted in a diagram the DSL generated. . 8

3.1 The first four rows of a context table for the control action “Pumping Insulin” in *WebSTAMP* [SPP+19]. 14
3.2 Confirmation request displayed by the tool [SPP+19]. 14
3.3 Creating rule `R1` [SPP+19]. 14
3.4 Manually determining a Unsafe Control Action (UCA) [SPP+19]. 15
3.5 A context table for the control action “open” in *An STPA Tool* [ST14]. 16
3.6 Defining rules in *An STPA Tool* [ST14]. 17

4.1 The context table concept for the control action “Manual Braking” with type “provided”. 21

5.1 The context table view in the STPA DSL extension. 24
5.2 The communication between the DSL components when the user starts the context table view. 26

6.1 Example of a simplified context table [Tho13]. 30

List of Tables

2.1 Excerpt of an example context table [Tho13]. 10
2.2 Example rule table “Open door command provided“ modeled after Gurgel et al. 11

Listings

2.1 Excerpt of the grammar definition of the STPA DSL. 6
2.2 Hazards in the DSL. 6
2.3 An excerpt of a control structure in the DSL. 7
2.4 Example of UCAs in the DSL. 9
4.1 The Rule concept. 22
5.1 The DSL's grammar updated with Rules. 27
5.2 Rules in the STPA DSL extension. 27

Chapter 1

Introduction

Technology has become an increasingly significant part of people's day-to-day lives. There are many aspects where humanity has become reliant on machines, for instance cars, buses and trains for the daily commute or the life-saving appliances in hospitals. Thus, when a machine or a system fails to work correctly, it can have fatal consequences on its environment depending on the situation. In consequence, hazard analysis is an important part of the development of all systems and needs to be executed with utmost precision. Many types of system failures exist, such as component failures or the ones caused by the environment, all of which need to be considered. Another type that must be prevented is communication between system components which leads to undesired behavior. This happens when errors in the system's design exist, which cause the behavior to appear by default.

Diverse hazard analysis techniques have been developed in order to improve system safety concerning this issue. One of these procedures is the STPA designed by Leveson, which analyzes a system in four steps [LT18]. First, the purpose of the analysis is outlined, which in essence also establishes its foundation. Then, the system structure is modeled in a functional model diagram called the control structure. Afterwards, the results of the first two steps are analyzed together, with the aim of finding possible design flaws which could compromise the system's integrity. The fourth and final step serves as a conclusion to the analysis, in which the design flaws as well as all of their consequences are delineated. All four steps are further detailed in Section 2.1. STPA counts as a well-established procedure for hazard analysis and can be used to find a wide range of safety issues. In comparison with other techniques it consistently exceeded in the quality of results as well as the minimization of time and resources used in the analysis [LT18].

However, despite its well-structured approach, applying STPA manually is tedious, complex and prone to human errors, especially if the system is significantly intricate. In order to aid in this matter, Petzold developed a DSL to systematize the procedure [Pet22]. The language aims to make STPA execution easier for system developers. It is implemented as a VS Code[1] extension and freely available to download with Github[2]. This thesis aims to improve the already available DSL tools and further automate the analysis procedure.

1.1 Problem Statement

Although the DSL already supports the entire STPA procedure, improvements to the language can still be made [Pet22]. In this thesis, a special focus lies on the third step of STPA: Analyzing the system's structure for potential causes of undesired behavior. This is done by looking at all actions the system components can take, and consequently analyze if these actions can, under certain environments, be a potential catalyst for unsafe system behavior. In this case, these actions are called UCAs.

The more complex the system gets, the more arduous and harder to survey the analysis becomes [Pet22]. Most systems used in day to day life, such as cars, airplanes, hospital equipment, etc., have

[1] https://code.visualstudio.com/

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

elaborate control structures, which need thorough risk analyses to combat the significant risk of system failure. Manually determining UCAs in cases as these is an option considerably prone to human error. Goal of this thesis is to improve the aforementioned DSL to make it a more available tool for practical STPA on complex systems.

One alternative to the manual procedure of determining UCAs are the context tables introduced by Thomas [Tho13]. Here, a table serves to apply every possible set of environmental conditions to an action that one of the system components takes, filling the table's rows. Following this, it can then be determined for each row whether UCAs can be identified.

According to Petzold, the main disadvantage of the DSL are the missing context tables [Pet22]. Changing this is the fundamental purpose of this thesis. In order to make the DSL a more viable tool for conducting practical STPA, the focus of this thesis lies on implementing the context table procedure. The rule-based approach will be applied to make the tables as optimized as possible. Said approach can automatically determine UCAs for context table rows by utilizing user-defined rules, all of which is further illustrated in Section 2.4. This implementation will make the tedious, error-prone process of identifying UCAs almost fully automatic.

1.2 Outline

The next chapter will introduce the thesis foundations. In Chapter 3, related work considering automating context tables will be discussed, as well as their (dis-)advantages. Afterwards, Chapter 4 discusses the concepts which went into the development of the context table implementation. Chapter 5 goes into the actual implementation in three parts: web-view foundations, the table including visualization, communication as well as data management, and rules. Finally, Chapter 6 concludes the thesis with a summary and potential future work.

Chapter 2

Foundations

Context tables are a procedure specifically designed to simplify the process of finding UCAs in the third step of STPA. Base knowledge about the tables as well as related topics needs to be present in order to implement this. More concretely, the foundations of STPA, the DSL, context tables and rules need to be understood.

This chapter is split into four sections, which detail these topics in the above-mentioned order: Section 2.1 explains the STPA process, followed by Section 2.2, which elaborates on the DSL, as well as on the components relevant for the implementation. Subsequently, Section 2.3 discusses context tables, going into important types of data used to create them. Lastly, Section 2.4 introduces the rules used to determine UCAs.

2.1 STPA

The STPA procedure is a well established method for identifying potential causes of accidents during a system's development [LT18] and serves to prevent system failures precipitated by poor development practices. Risks can be effectively identified and proactive measures against them can be introduced in consequence of the analysis.

STPA as introduced by Leveson analyzes a system in four consecutive steps [LT18]:

1. **Define the purpose of the analysis.** In the beginning step, foundational information of the analysis needs to be determined, such as the system's general purpose and boundaries. Moreover, the potential hazards and losses are to be identified and indexed. In this context, a loss refers to losing something of value to stakeholders, e.g. human lives, environmental damage or information leak. Furthermore, a hazard is defined as a system state / set of conditions which, together with a specified set of worst-case environmental conditions, lead to a loss. Usually, the losses that are caused are referenced in each hazard's definition. The desired scope of the analysis should be regarded while identifying hazards. For instance, the analysis could serve only for data security hazards, or perhaps instead could encompass multiple scopes, such as performance, safety and environmental damages.

2. **Model the control structure.** Here, the system structure concerning controllers is identified. A control structure is a functional model which represents a system with controllers, controlled processes as well as control-feedback loops between the controllers with each other or controlled processes. The different actors in the system as well as the control actions that can be taken within it are determined in this step.

3. **Identify UCAs.** In this step, the control structure in the previous step is analyzed for behavior that could trigger hazards and therefore cause losses. More specifically, control actions executed in a specified, for it unsafe context, are sought after, which consequently are labeled as UCAs.

4. **Identify Loss Scenarios**. This last step serves as a conclusion to the previous steps of the analysis. With the UCAs determined in the previous step, the loss scenarios potentially caused by them are to be declared. This way, system developers can take measures to prevent these scenarios.

2.2 The STPA DSL

The STPA DSL developed by Petzold aims to simplify the execution of STPA by providing an as optimized as possible interface for the procedure [Pet22]. While other tools supporting STPA already existed before the DSL's development, Petzold discovered that all of them had advantages and disadvantages concerning time and resources. Consequently, she aspired to create the DSL in a way that makes use of all advantages and keeps the disadvantages minimized. After the main development had been finished, it was concluded that the main advantage of the new DSL was the visualization component, which can depict both the control structure of a defined system and the currently executed STPA process in two separate diagrams. As mentioned in Section 1.1, the main disadvantage turned out to be the missing context tables, which this thesis aims to fix.

The DSL is implemented as a VS Code[1] Extension. VS Code is an Integrated Development Environment (IDE) owned by Microsoft, which possesses a so-called Extension API. Accessing it allows programmers to customize and enhance almost every part of the IDE. This includes adding custom components, creating web views, which will be discussed in more depth in Chapter 5, and supporting new programming languages[2]. Petzold made use of the last stated feature especially, creating a new language out of the STPA process. This new language accepts files with the ending `.stpa`, reads them and checks the current progress of the STPA process contained in the file with the help of the built-in language server (more discussed in Section 2.2.2). The entire STPA process with all four distinct steps is fully supported [Pet22].

2.2.1 Extension

Figure 2.1 shows an example view of how the extension looks like on startup. The standalone message displaying "`Activating STPA extension`" indicates a successful initialization of the DSL.

An `.stpa` file, when completed, encompasses all four steps of an entire STPA process [Pet22]. The DSL is defined in a way that it consists of multiple aspects, those of which being: losses, hazards, system-level constraints, control structure, responsibilities, UCAs, controller constraints, loss scenarios and safety requirements. In the `.stpa` file, these aspects have to be defined in the order above. As can be seen in Figure 2.1, most aspects are to be written with syntax "`<ID> <String> [<Ref>, ...]`", except for losses, which do not possess references, loss scenarios, which may also have syntax "`<ID> for <Ref> <String> [<Ref>, ...]`", and the control structure.

In Figure 2.1, on the top right of the window, below the window's visualization menu, one can find an assortment of buttons serving as the web view options. Clicking the most left of these buttons opens up the STPA diagram view.

Once the diagram view has been started up, the two types of diagrams mentioned in Section 2.2 are generated. Both the control structure diagram (example in Figure 2.2) and the STPA diagram provide a better overview of the current process and its components. The DSL relies on an open-source web-based framework called Sprotty[3] [Pet22], whose components are responsible for rendering and creating the

[1] `https://code.visualstudio.com/`

[2] `https://code.visualstudio.com/api`

[3] `https://projects.eclipse.org/projects/ecd.sprotty`

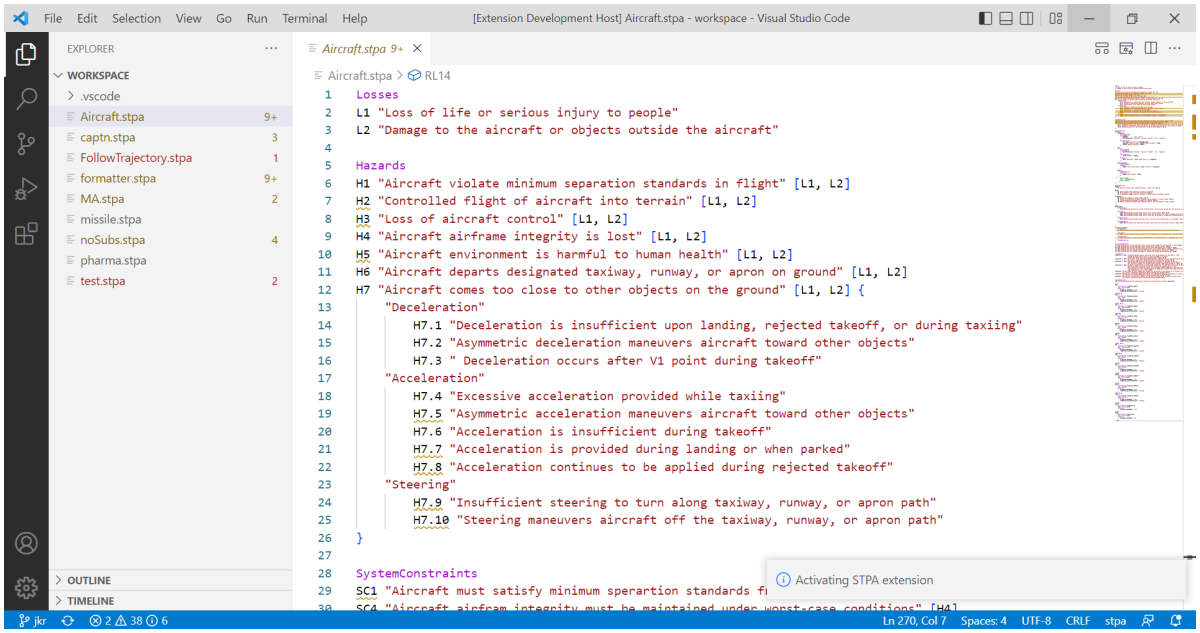

Figure 2.1. The STPA extension.

diagrams, as well as their user events. Sprotty communicates with the language server to send and receive data to make this possible.

2.2.2 Language Server

The language server handles everything concerning the DSL's language, including grammar definition, language and file validation, scopes and language data [Pet22]. It was built under use of Langium, which provides the tools for language parsing, language data management and editing support for the extensions.

Since the DSL's grammar is written exclusively using Langium, its definition in the `.langium` file is closely leaning on the Extended Backus-Naur Form (EBNF), a family of notations used to describe context-free grammars. The determining factor of a EBNF syntax is to define the grammar using terminal symbols and non-terminal production rules [ISO96], which is maintained in Langium. For instance, in Listing 2.1, the production rule for a `Loss` is defined by `name = ID`, which refers to the definition of `ID`, followed by `description = STRING`, which requires a String type input, similar to a terminal symbol. Directly underneath, a `Hazard` is described by the same two equations followed by an optional non-empty list of Loss references. Finally, the `Hazard` rule allows another optional list of newly defined `Hazards` to exist as the original `Hazard`'s subcomponents.

The language server provides multiple services, which in turn provide access to a variety of data, such as specific states, currently open documents and the language model itself. Various other components, such as the diagram components as well as the main extension component, need access to the data. The server provides a connection for this reason, over which these other components can communicate with it via notification sending and receiving when called.

```
Loss:
        name=ID description=STRING;

Hazard:
        name=SubID description=STRING
        ('['refs+=[Loss] (',' refs+=[Loss])*']')?
        ('{' (header=STRING? subComps+=Hazard+)*'}')?;
```

Listing 2.1. Excerpt of the grammar definition of the STPA DSL.

2.2.3 Hazards

As mentioned in Section 2.1, hazards are system states constructing hazardous system behavior that may lead to a loss [LT18]. These system states are worst-case consequences of unsafe and undesired behavior of the system, which are aimed to be prevented. For instance, in Listing 2.2, hazard `H6` describes the aircraft departing its designated taxiway / runway / apron, which can cause substantial harm to human lives, the aircraft and the environment. This harm is listed in the two losses `L1` and `L2` the hazard references.

```
H6 "Aircraft departs designated taxiway, runway, or apron on ground" [L1, L2]
H7 "Aircraft comes too close to other objects on the ground" [L1, L2] {
        "Deceleration"
                H7.1 "Deceleration is insufficient
                      upon landing, rejected takeoff, or during taxiing"
                H7.2 "Asymmetric deceleration maneuvers aircraft toward other objects"
                H7.3 "Deceleration occurs after V1 point during takeoff"
        "Acceleration"
                H7.4 "Excessive acceleration provided while taxiing"
                H7.5 "Asymmetric acceleration maneuvers aircraft toward other objects"
                H7.6 "Acceleration is insufficient during takeoff"
                H7.7 "Acceleration is provided during landing or when parked"
                H7.8 "Acceleration continues to be applied during rejected takeoff"
        "Steering"
                H7.9 "Insufficient steering to turn along taxiway, runway, or apron path"
                H7.10 "Steering maneuvers aircraft off the taxiway, runway, or apron path"
}
```

Listing 2.2. Hazards in the DSL.

As can be seen in Listing 2.2 above, the hazards are to be written with the general syntax discussed in Section 2.2.1. Furthermore, each hazard should reference at least one loss each. Optimally, the references cover all identified losses. As an additional option, hazards can have sub-hazards [Pet22], such as hazard `H7`. These sub-hazards serve as a refinement of the identified higher-level hazards and can be assigned to hazard-individual categories. In Listing 2.2, these categories are named `Deceleration`, `Acceleration` and `Steering`. It can also be seen that a category serves to contain a theme to be reflected in each of the contained sub-hazards. If a hazard contains sub-hazards, other aspects such as UCAs or loss scenarios will usually only reference the appropriate sub-hazards instead of the

higher-level hazard. However, while sub-hazards are in fact hazards with their own ID, they do not reference losses. Instead, the references of the high-level hazards count for all their subcategories.

2.2.4 Control Structure

Defining the control structure constitutes the second step of the STPA. It provides a detailed look at the system divided into its subcomponents, which, in this context, are controllers and controlled processes. For instance, in the defined control structure visible in Listing 2.3 and Figure 2.2, there are four components visible: the `FlightCrew`, the `BSCU`, `OtherSubsystems` and `Wheels`. The `FlightCrew` is a controller only, while the other three are controlled processes in addition. This is indicated by the fact that they possess lower positions in the hierarchy as expressed by their hierarchy levels. While in other functional models, the `FlightCrew` may rather be part of the environment instead of a direct component of the system, they do exert a controlling force on the system. Therefore, they need to be directly included in the control structure as a component.

```
1  ControlStructure
2  Aircraft {
3          FlightCrew {
4                  hierarchyLevel 0
5                  processModel {
6                          BCSUmode        : ["on", "off"]
7                          aircraftPosition: ["docked", "taxiing", "takeoff",
8                                             "air", "landing"]
9                  }
10                 controlActions {
11                         [mc "Manual Controls"] -> Other Subsystems
12                         [powerOff "Power Off BSCU", powerOn "Power On BSCU"] -> BSCU
13                         [manual "Manual Braking"] -> Wheels
14                 }
15         }
16         OtherSubsystems {
17                 hierarchyLevel 1
18                 feedback {
19                         [modes "Other system modes", states "states"] -> FlightCrew
20                 }
21         }
```

Listing 2.3. An excerpt of a control structure in the DSL.

In the DSL, the control structure does not follow the usual syntax described in Section 2.2.1. Furthermore, it does not reference other aspects. Instead, the components defined inside the control structure aspect only have references to other components in some of their properties, such as in `controlActions`.

The control structure is defined component-wise. As can be seen in Listing 2.3, the single components are defined in the scope of the system, which is also given an ID, in this case `Aircraft`. The components' definitions consist of various properties, most of them optional. The only required property is the hierarchy level that defines the level a component is placed on in the control structure diagram, as can be observed in Figure 2.2. While the other properties are optional, two of them are

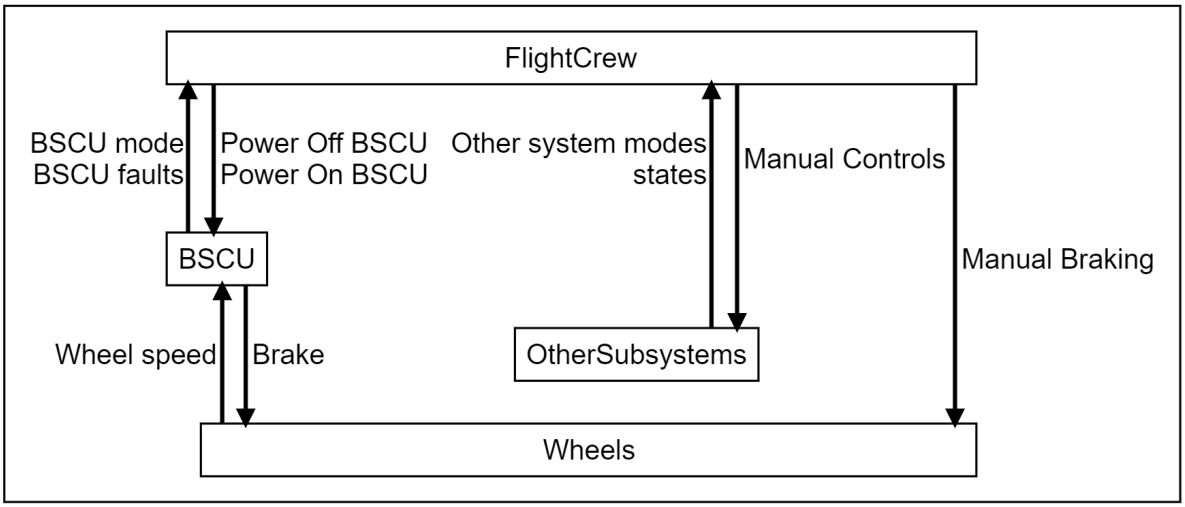

Figure 2.2. The same control structure as in Listing 2.3 depicted in a diagram the DSL generated.

important for the generation of context tables: the process model and the control actions. The process model contains the context variables of a controller, which are further discussed in Section 2.3. The control actions are the actions the controller under which they are listed can take. As mentioned in Section 1.1, they are a key component of the context tables, as they are the target of the analysis conducted with them.

2.2.5 Unsafe Control Actions

A UCA, as defined in Section 2.1, describes system behavior which is considered unsafe in the sense that it could cause hazardous system behavior to occur, which, in turn, may lead to one or more losses [LT18].

Obtaining a set of UCAs is the goal of step three of the STPA. This is done by focusing on the control actions of the different components of the control structure built in the prior STPA step, and analyzing them under all possible environmental conditions important to the controller that executes the action. These environmental conditions are also called the context in which a control action is executed in. A context consists of multiple process variables, which, in the DSL, are defined as the property `processModel`, seen in Listing 2.3 where component `FlightCrew` possesses the two context variables `BCSUmode` and `aircraftPosition`. For every possible context, the person conducting the analysis has to determine if there is the possibility of the executed control action causing a hazard. Furthermore, there are various ways in which a control action can be executed. These varieties are called types. For instance, two basic control action types are “provided“ and “not provided“, which determine if a control action is executed or not. The latter type is especially important for cases where a control action needs to be executed for the system to behave safely, therefore not providing the action may cause a hazard and consequentially has to be identified as a UCA.

The UCAs are composed of four different parts in the following order: the controller, the type, the control action and the context. For example, in Listing 2.4, `UCA2` is identified by controller `BSCU`, control action type `provides`, control action `Brake` and context `during takeoff`. Furthermore, each UCA needs to reference at least one hazard, which the UCA may to usually followed by the identifiers. `UCA2` refers to two hazards in its reference list indicated by the brackets: `H7.3` and `H7.6`, both of which can be

looked up in Listing 2.2.

In the DSL, the UCAs themselves are written with the general syntax discussed in Section 2.2.1. However, unlike other aspects, they are all sorted into categories. First, they get assigned to the control action they result from. It is of note that every control action is called by the controller which executes it. Following, they are categorized by their control action type. In Listing 2.4, UCAs of control action `BSCU.brake` are listed. Here, the possible types listed are the two base types `notProviding` and `providing`, in the addition of two more types, `tooEarly/Late` and `stoppedToSoon`. The UCAs are split into these categories in order to create a better overview and ensure no type gets forgotten, instead of listing them all one after another.

```
1  UCAs
2  BSCU.brake {
3          notProviding {
4                  UCA1 "BSCU Autobrake does not provide the Brake control action
5                        during landing roll when the BSCU is armed" [H7.1]
6          }
7          providing {
8                  UCA2 "BSCU Autobrake provides Brake control action
9                        during takeoff" [H7.3, H7.6]
10                 UCA5 "BSCU Autobrake provides Brake control action
11                       with an insufficient level of braking during landing roll" [H7.1]
12                 UCA6 "BSCU Autobrake provides Brake control action
13                       with directional of asymmetrical braking
14                       during landing roll" [H7.1, H7.2]
15         }
16         tooEarly/Late {
17                 UCA3 "BSCU Autobrake provides the Brake control action too late
18                       after touchdown" [H7.1]
19         }
20         stoppedTooSoon {
21                 UCA4 "BSCU Autobrake stops providing the Brake control action too early
22                       when aircraft lands" [H7.1]
23         }
24 }
```

Listing 2.4. Example of UCAs in the DSL.

2.3 Context Tables

Context tables, introduced by Thomas [Tho13], are used as a more systematic approach to conducting step three of STPA. Thus, they serve to identify the set of all UCAs from the control structure of the system to be analyzed. Each context table analyzes one control action together with a type, either "provided" or "not provided", under all possible contexts. A context, as explained in Section 2.2.5, consists of the action's context variables in the control structure. In each row, the set of variables is initialized with a unique set of values, until all combinations have been listed. For each row, the person conducting the analysis needs to decide if the control action should be considered unsafe. This

is the case if a listed hazard is a realistic consequence of the behavior displayed in the context table row, assuming the list of hazards created in the first STPA step is complete. If so, a new UCA has been identified and needs to be added to the set of UCAs. For instance, in the second row of Table 2.1, it needs to be determined if "`Door open command not provided`" is hazardous under the context that the train has stopped but is not aligned with a platform, there are no people standing in the doorway and there is no emergency. In this case, no hazardous consequences could be detected, so there is no UCA to be defined.

Table 2.1. Excerpt of an example context table [Tho13].

Control Action	**Train Motion**	**Emergency**	**Train Position**	**Door State**	**Hazardous if not provided in this context?**
Door open command not provided	Train is stopped	No emergency	Aligned with platform	Person not in doorway	No
Door open command not provided	Train is stopped	No emergency	Not aligned with platform	Person not in doorway	No
Door open command not provided	Train is stopped	No emergency	Aligned with platform	Person in doorway	Yes

A context table is constructed using the following method: First, a column for the control action is created. This is done by selecting a controller or controlled process from the control structure, looking at their control actions and picking one. Next, the type of the control action needs to be determined. Usually, only the two base types, "provided" and "not provided", are regarded in different tables, meaning there are two tables per action. The chosen type is to be mentioned along with the control action, as can be seen in the first column in Table 2.1 where the chosen control action is "`Door open command`" and the type "`not provided`".

Moving on, the context variables for the action need to be assembled. Since context variables are directly defined as a controller's property, all the for the chosen action relevant ones are found as one of its executor's properties. The set of variables is assembled after the control action column as one more column each, with the variable as its name. For instance, in Table 2.1, the variables relevant for "`Door open command`" are "`Train Motion`", "`Emergency`", "`Train Position`" and "`Door State`". The columns' cells below the header are to be filled with the values the variables can take, one per cell each. In each new row created, the set of values must be different to all prior rows created. If there are no more combinations to add, no more new rows must be added.

Lastly, a result column labeled "Hazardous?" or similar is to be added to the table. This column is reserved for the results of the table analysis and is filled using the method described in this section's first paragraph, meaning for each of the tables rows, the described behavior is analyzed with the help of the available list of hazards. If one of the hazards is a realistic potential consequence, the behavior is to be considered unsafe, and a new UCA should be added to the list of UCAs.

There is, however, an alternative, more systematic procedure to this method called the rule-based approach. This procedure will be discussed in the following section.

2.4 Rules

In the prior section's introduced approach to filling context tables, each row of the hazardous-column needs to be processed manually by the person conducting the analysis. As explained in Section 1.1, this procedure can easily become too complex and, therefore, error-prone.

The rule-approach to filling the hazardous-column constitutes an algorithmic alternative. Here, so-called rules are defined for control actions with type as to which context constitutes determining it as unsafe [GHB15]. Rules were first suggested by Thomas, and following this, a proper approach to finding UCAs based on rules was introduced by Gurgel et al. in 2015. In their approach, rules are defined directly before creating the context tables. Much like the latter mentioned, they are defined in one table per control action with base type ("provided" or "not provided") each. One example of this can be seen in Table 2.2, for a control action called "`Door open command`" with type "`provided`". Two rules are indexed in the table: "`R1`" holds if the train is not aligned with a platform and there is no emergency and "`R2`" if the train is moving and there is no emergency. A variable gets assigned with the expression "`ANY`" if its value assignment does not influence the validity of the rule. For instance, "`R1`" holds *no matter* the train motion.

Table 2.2. Example rule table "Open door command provided" modeled after Gurgel et al.

Index	Train Position	Train Motion	Emergency
R1	not aligned with platform	ANY	no emergency
R2	ANY	moving	no emergency

Once the rules have been defined, the context tables are created using the method introduced in Section 2.3. However, instead of checking each table row with each of the hazard list's entries, the rule tables are used to systematically confirm the affected rows as hazardous while the remaining ones are not hazardous [GHB15]. For example, with "`R1`", all rows of the context table for "`Door open command provided`" where "`Train Position`" is set to "`not aligned with platform`" and "`Emergency`" to "`no emergency`" are judged as hazardous.

In comparison to the manual method, the rule-based approach is a noticeably more time-efficient and adaptable to changes [GHB15], feasibly shifting the focus of the UCA identification process from analyzing each context for each control action to simply adding and removing rules.

Chapter 3

Related Work

As mentioned in Section 2.2, other existing tools supporting STPA were considered during the creation of the DSL. In this chapter, three of these tools are introduced with focus on how they integrate context tables to identify UCAs. Each of these tools provide different approaches, which will be analyzed and evaluated concerning the use of context tables in Petzold's DSL.

3.1 XSTAMPP

The Extensible STAMP Platform (XSTAMPP)[1] is an open-source program written by Abdulkhaleq et al. and based on the Eclipse Plug-in-Development Environment (PDE)[2] and the Rich Client Platform (RCP)[3] [AW16].

XSTAMPP offers a variety of different plugins to the user, allowing them to flexibly assemble their user interface fitting to the application area of the analysis they conduct. One of the plugins offered is its predecessor, *A-STPA*, which only provides support for basic STPA. Another important one is called *XSTPA*, which extends the program with support for context tables and other improvements for STPA suggested by Thomas [Tho13]. *XSTPA* automatically generates context tables from the process model found in the user-defined control structure. Once hazardous combinations have been defined by analyzing the tables, the user can let the program generate UCAs with a built-in algorithm. The set of UCAs are defined as Linear Temporal Logic (LTL) specifications, which, in the context of XSTAMPP, can directly be used for model checking and testing.

In conclusion, XSTAMPP offers support for context tables, and can effectively generate a set of UCAs, which additionally can be used for other features of the program. However, analysts need to manually analyze and complete the context tables. While the tables make the UCAs identification process more time-efficient, there is still room for improvement in the context of simplification. By implementing rules, *XSTPA* could provide a tool that lets users determine sets of rows as hazardous at once, instead of needing to define each row individually.

3.2 WebSTAMP

WebSTAMP is a web application based on System-Theoretic Accident Model and Processes (STAMP) that partially supports STPA [SPP+19]. More specifically, the program focuses on identifying UCAs and loss scenarios. As control structure construction is not a feature of the application, this needs to be done externally. Once the necessary process model variables are provided, context tables will automatically be created, an example of which can be seen in Figure 3.1. A context table offers various categories such as the base options "provided" and "not provided", "`Wrong order of Control Action`" and many

[1] https://github.com/SE-Stuttgart/XSTAMPP

[2] https://www.eclipse.org/pde/

[3] https://wiki.eclipse.org/Rich_Client_Platform

more. After the generation of a context table, each of the cells of the hazard category columns will display a "?", signifying the cell is not analyzed yet.

#	Glucose Level	Reservoir level	Battery level	Pump Operational Status	Index Rule	Control Action provided	Control Action not provided	Wrong order of Control Action	Control Action provided too early	Control Action provided too late	Control Action stopped too soon	Control Action applied too long
1.	Below	Below	Low	Transmitting	R1	Hazardous	?	?	?	?	?	?
2.	Below	Below	Low	Not transmitting	R1	Hazardous	?	?	?	?	?	?
3.	Below	Below	Normal	Transmitting	R1	Hazardous	?	?	?	?	?	?
4.	Below	Below	Normal	Not transmitting	R1	Hazardous	?	?	?	?	?	?

Figure 3.1. The first four rows of a context table for the control action "Pumping Insulin" in *WebSTAMP* [SPP+19].

If a cell is identified as hazardous, the analyst can select the cell to display `Hazardous`, as can be seen in the `Control Action provided` column in Figure 3.1. Once a cell has been marked as hazardous, the application will automatically ask to add the hazardous combination as a UCA, an example of which can be seen in Figure 3.2, corresponding to the first row in Figure 3.1. Otherwise, if a cell is identified as not hazardous, a blank space (" ") can be displayed.

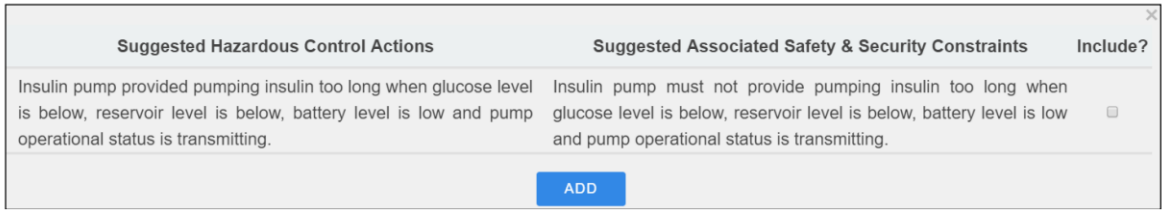

Figure 3.2. Confirmation request displayed by the tool [SPP+19].

Additionally, rules are supported. With the user interface shown in Figure 3.3, new rules can be created by first choosing one of the UCA categories and values for at least one of the process model variables and then clicking on the "`Add new rule`" button. After going back to the context table interface, the new rules are displayed in a column to the left of the hazard category columns if they apply. In Figure 3.1, the in Figure 3.3 created rule called `R1` applies to all four rows.

Figure 3.3. Creating rule `R1` [SPP+19].

As an alternative to using the context tables, a user can manually identify UCAs (see Figure 3.4). Similar to rules, the user needs to define the category and at least one of the process model variable. A text for the new UCA is automatically generated by the tool along with a fitting constraint. Unlike defining UCAs with the context tables, process model variables can be left unassigned if their values do not influence the validity of the UCA.

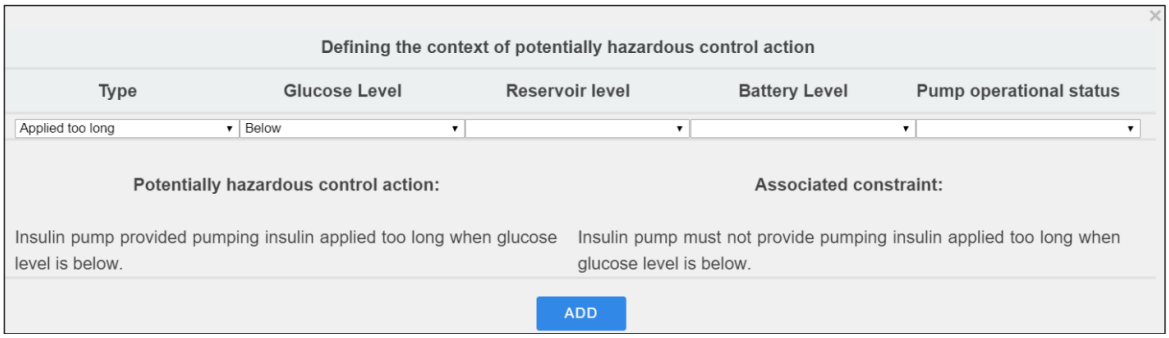

Figure 3.4. Manually determining a UCA [SPP+19].

WebSTAMP's user interfaces are easy to comprehend and provide many alternatives to make the process of identifying UCAs time-effective and manageable. The user can decide how to combine the different approaches on the basis of how effective they are in analyzing the provided control structure. As a result, it maintains flexibility with various levels of control structure complexity. However, since it only supports the creation of UCAs and loss scenarios, other aspects of STPA such as losses and hazards can not be included. Therefore, hazards cannot be referenced in the created UCAs, which means they remain partially incomplete. An inclusion of these missing aspects could help bring more clarity over the conducted STPA process.

3.3 An STPA Tool

An STPA Tool was developed by Suo and Thomas at the Massachusetts Institute of Technology (MIT) [ST14]. It focuses on specifying hazards, drawing the safety control structure and identifying UCAs. For identifying UCAs, the tool provides support for context tables and the rule-based approach. The tables are automatically generated from the control structure drawn by the user.

An example of a context table generated by the tool can be seen in Figure 3.5, where it can be assumed that the control actions `open` and `close` refer to train doors. Unlike the previously discussed *WebSTAMP*, the context tables of *An STPA Tool* include a column to display the control action in each row. Furthermore, while *WebSTAMP* offered "provided" and "not provided" as categories next to "too early", "too late" and more, *An STPA Tool* uses them as types in an additional `Type` column right next to the `Control Action` column. Two columns are dedicated to the analysis of the rows, allowing users to identify rows as generally hazardous and hazardous specifically when the control action is executed too early or too late. Since hazards are a feature of the tool, if a hazardous combination is identified in a row, the row's hazard column cells directly reference the hazards by ID after they have been entered into the cell. Otherwise, if the row is identified to be not hazardous, both columns are simply left blank.

The last two columns of the table bind in the rule-based approach, with one column for referencing rules if they apply, and the other for potential rule conflicts. Rules are defined by clicking the `Rule definition` button shown near the upper right corner of Figure 3.5, which opens the user interface shown in Figure 3.6. Here, the user can define new rules with the help of given parameters. In order to create a new rule, the user needs to select the type as either `Provided` or `Not provided`, apply a value to at least one of the process model variables, and finally choose the hazards the rule should refer to. The new rules are listed in a box below the rule definition parameters for more clarity. Below that, an

And/Or table can be found displaying a control algorithm for the current control action by listing the rule conditions in one column per rule, so that the user can determine possible safety requirements for the execution of the control action.

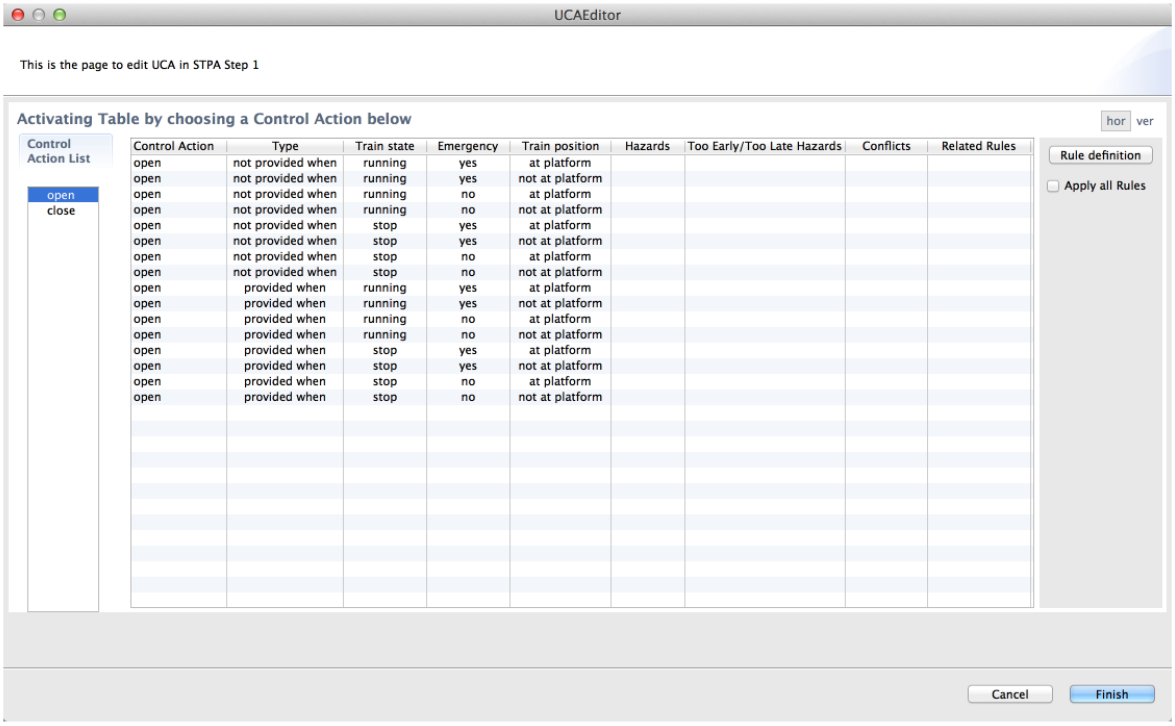

Figure 3.5. A context table for the control action "open" in *An STPA Tool* [ST14].

An STPA Tool provides comprehensible context tables, especially by additionally listing the control action and type in the table's first two columns. Because of this, a row's context can simply be read by going through the columns from left to right. After the generation of a context table, all rows are first assumed to not contain hazardous combinations until otherwise defined by the user. This improves the tables' time-efficiency, as the user does not need to work through each of a table's cells for it to be completely analyzed. Moreover, rules can easily be defined and included with the provided interface and the various rule features it offers to the user. Especially the ability to detect rule conflicts and the And/Or table ensure effective options to keep the current rule status assessable, although the And/Or table could benefit in clarity from the implementation of column descriptors.

However, including the `Type` column in the table instead of having "provided" and "not provided" as two more hazard categories, like *WebSTAMP* does, generates tables that have double the row count in comparison. While this approach remains relatively inconsequential with smaller tables, with more complex ones it may lead to a significant decrease in time-efficiency. Furthermore, this approach to types and hazard categories also persists in the rule definitions, where a user can only bind in the types "provided" and "not provided". On the other hand, they cannot differentiate between generally hazardous behavior or hazardous behavior only if the action is executed too early or too late, even though the distinction can be made in the two hazard columns of the context table. Displaying the type as two more hazard categories could be an effective alternative in the context of *An STPA Tool*, since the tables work on the basis of a row not being hazardous until found otherwise, meaning the time-effort

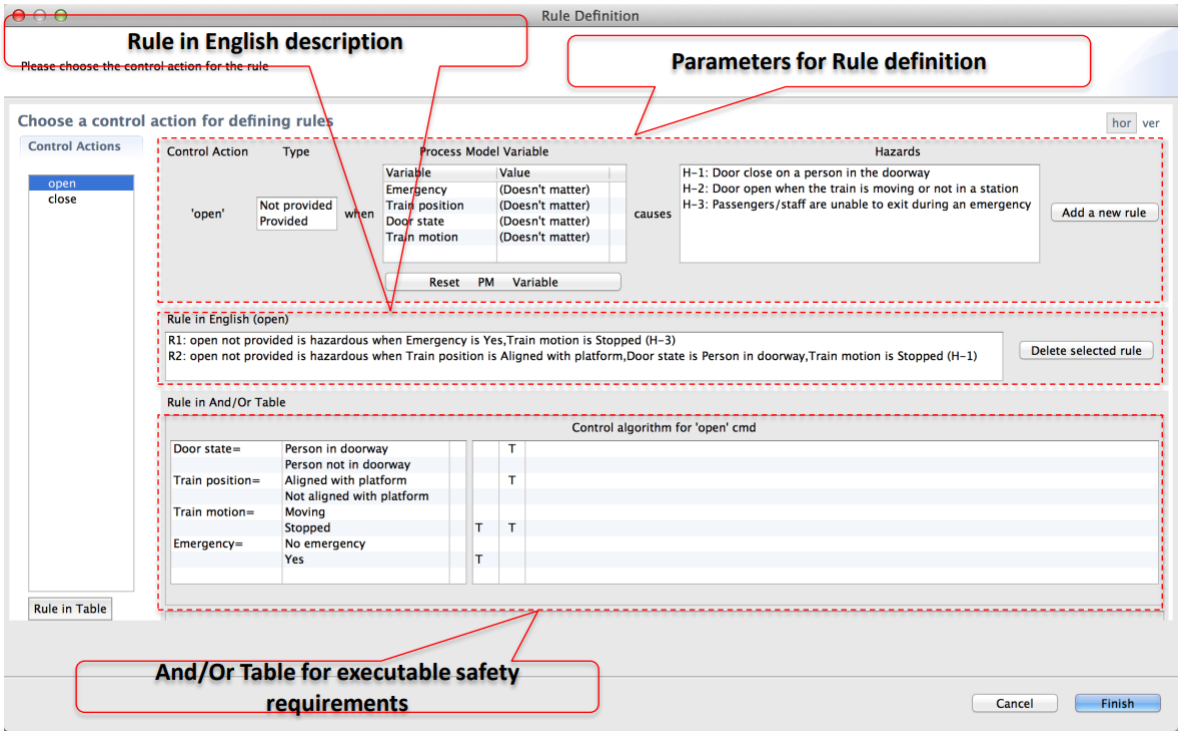

Figure 3.6. Defining rules in *An STPA Tool* [ST14].

of analyzing a context table would remain largely unchanged. Furthermore, since there currently are only two existing hazard categories, increasing the number of hazard columns to four would not make the table too incomprehensible, assuming *WebSTAMP*'s seven categories were accounted as not making the table too complex.

Chapter 4

The Context Table Web-View

This thesis focuses on implementing context tables for Petzold's DSL. A challenge to overcome with this is to find an approach that suits the functionality of the DSL's already existing components. The DSL includes both textual and graphical elements for various steps of STPA, as both approaches offer advantages concerning efficiency while being able to balance out disadvantages of the respective other approach. For instance, as can be seen in Section 2.2.4, the user writes a system's control structure as a text adhering to the DSL's grammar, and has the option of opening up a diagram view (see Section 2.2.1) depicting the structure as a graph. Here, the advantage of the graphical component lies in the overview it gives users regarding their current progress. However, having to manually construct a well-structured layout of the graph with methods such as drag-and-drop and pop-ups would be tedious and time-inefficient [GKR+14]. The DSL circumvents this by having the user define the control structure textually with a concise grammar. As a diagram based on the structure's definition can be displayed, the graphical overview advantage is not lost.

The context table implementation should follow the same approach: The tables should be automatically generated out of a chosen component of the control structure. They should then be available in a graphical view which can be accessed similarly to the diagram view. However, while the majority of the table's content is automatically generated, the "Hazardous" column still needs to be filled out manually by the user. Instead of filling out the table's columns directly in the graphical view, the tables' hazard columns should instead be edited textually by extending the language's grammar with a new aspect: Rules (see Section 2.4). As they have been developed to systematize the analysis of context tables, rules are a fitting tool to implement the desired approach. Once the user has defined rules for a system, the table view should update to include them in the evaluation of the "Hazard" column. Implementing rules gives users an iterative method of identifying UCAs while the graphical view offers an updated overview over the current status with each conducted iteration.

Therefore, the goal of this thesis is the implementation of context tables into the DSL which have a visualization component that generates a table view out of the DSL's textual control structure aspect, and rules as a textual component with which the tables' hazard columns are edited.

The following sections discuss the concepts for the implementation: Section 4.1 explains the view concept for the context tables and how it should be integrated into the DSL, followed by Section 4.2, which discusses the concepts for the context tables. Finally, Section 4.3 specifies the concepts for rules and how they are defined in the DSL's grammar.

4.1 The View

The view displaying the context tables should be made accessible similarly to the diagram view introduced in Section 2.2.1: A button for the table view must be implemented to be positioned on the same bar as the diagram view button. Clicking on the button will open the context table view, splitting VS Code's workspace screen into two parts, the one on the left showing the current STPA file, and the one on the right displaying the table view.

Since the context table view needs data from hazards, the control structure and rules to generate tables, the DSL must send the mentioned information to the view. This should be done by the extension sending notifications and receiving the necessary data back. As the DSL must maintain multiple STPA documents' data, the context table view, knowing which document it belongs to, should provide a document identifier to the extension. Following this, the extension notifies the DSL of this identifier and receives the needed context table data as a response.

The inclusion of rules (see Section 2.4) makes working with context tables an iterative process: First, rules are defined, and then the progress is inspected for correctness. Following this, the process begins anew with the adaptation of the existing rules, definition of new rules or deletion of old rules, until the analysis has been completed. In regard to the DSL, context tables are meant to be a visual component helping with keeping an overview over the created progress. For each inspection done in the mentioned iterative process, the analyst needs updated context tables. Therefore, whenever the user updates their current STPA file, meaning i.e. that they save changes, the context table view must update as well.

In order to keep the table view as manageable as possible, only one context table is displayed at a time. This context table is associated with a certain control action and a type (see Section 2.2.5 and Section 4.2), both mentioned in the table. In order to switch between context tables, selection elements for both control action and type must be implemented into the view. When changing the selected control action or type with these elements, the view component immediately updates and replaces the now old table with a new one fitting to the currently selected options.

4.2 Context Tables

There are a variety of design choices to be made when creating a context table. However, the final concept made for the DSL' context tables mostly leans on their original definition and introduction by Thomas [Tho13].

As discussed in Section 4.1, one table is displayed in the view at a time. Inspired by the types used by *An STPA Tool*, the generated table is associated with a certain control action paired with one of the following types: "provided", "not provided" or "both". However, instead of depicting all types in one table, the user selects a type to be paired with the control action. The selection of both the control action and type can be changed anytime.

The selected type determines how the hazard evaluation column is structured. As can be seen in Figure 4.1, type "provided" has three different hazard categories: "anytime", "too early / too late" and "stopped too soon / applied too long". These hazard categories reference the ones used to compartmentalize UCAs in the DSL (see Listing 2.4), with "anytime" being used in place of UCA category "providing". On the contrary, when type "not providing" is selected, the hazard column has no subcategories, as the scenario of a control action being not present makes them redundant. The last type, "both" exists to depict both the "provided" and the "not provided" option in one table. In this case, the table is structured like the one for the "provided" type visible in Figure 4.1, with the

only difference being that the “not provided” type is represented with an additional fourth hazard subcategory called “never”.

In summary, the selectable types “providing” or “not providing” introduce context tables with fewer columns than a table that contains both types as hazard subcategories. These tables are therefore easier to manage and well suited for smaller screens such as laptop screens. However, if a user wants a bigger table containing both “providing” and “not providing” as hazard subcategories, they can select type “both” and view columns “anytime” and “never” respectively.

Control Action	Context Variables		Hazardous?		
	aircraftPosition	BSCUMode	anytime	too early / late	stopped too soon / applied too long
Manual Braking provided	docked	yes	No		
"	docked	no	No		
"	taxiing	yes	No		
"	taxiing	no	No		H7
"	takeoff	yes	No		
"	takeoff	no	H3	H3	H6
"	air	yes	No		
"	air	no	No		
"	landing	yes	No		
"	landing	no	H7	H7	H7

Figure 4.1. The context table concept for the control action “Manual Braking” with type “provided”.

The control action and type are written in all cells of the first column of the table, the “Control Action” column, to make the context table easier to read. For instance, in Figure 4.1 the context table is generated for “Manual Braking provided”, since the selected control action is “Manual Braking” and the type “provided”. For type “both”, the control action will also be listed as provided, whereas for “not provided”, it will be listed as not provided.

The context variables (see Section 2.3) are collected and displayed in the “Context Variables” column as sub-columns. The “Hazardous?” column and its potential subcategories forms the last part of the context table. As can be seen in Figure 4.1, each subcategory is represented in one sub-column. The hazard columns’ cells are filled out textually by defining rules instead of directly editing the table’s cells in the view, adapting them to the DSL’s existing approach to STPA. If no rules are defined in the current STPA file, all cells of the “Hazardous?” columns will display “No”. Otherwise, if the view component received rule data (see Section 4.3), each of the affected cells of the “Hazardous?” column will be filled with the list of hazards the corresponding rule references. For instance, in row 6 of Figure 4.1, a rule which references hazard `H7` holds when “Manual Braking” is provided and the “aircraftPosition” is set to “taxiing” and “BSCUMode” to “no”. The rules should be traceable from the hazard references displayed in the “Hazard” column’s cells. Furthermore, neighboring “No” entries in a context table row’s cells will be combined to one cell for better readability. The same will not be done for neighboring matching hazard references as they might be caused by different rules.

4.3 Rules

Rules are an essential part of the conceptual context tables’ functionality, as they are used to modify the “Hazardous?” column’s cells. They must be integrated into the DSL’s grammar introduced in Section 2.2.2, so that they can be defined in STPA files and subsequently used by the context table view component.

Listing 4.1 shows the concept developed for the rule implementation. This concept translates the rule approach discussed by Gurgel et al. (see Section 2.4) into a grammatical rule close to the ones used for the DSL's Langium grammar definition (see Listing 2.1). It also references multiple definitions already in use in the Langium file, more specifically "`ID`", "`Command`", "`Variable`" and "`HazardList`".

```
1       Rule:
2               name = ID, '{'
3                       'Control Action: ', action = [Command]
4                       'Type: ', type = Type
5                       'Context: ', '{'
6                               vars += [Variable], {', ', vars += [Variable]}*
7                       '}'
8               '}'
9               hazards += HazardList
```

Listing 4.1. The Rule concept.

Rules will be implemented as an additional aspect, next to "`Losses`", "`Hazards`", "`ControlStructure`" and more. As can be seen, to fully define a rule, it must receive a `String` type ID. Furthermore, it must reference an existing control action, defined as "`Command`" and at least one context variable. While the concept keeps the `Context` references short for the sake of readability, both the name and value of each variable should be listed in the implementation. Additionally, the rule must have a type, which, just like the rules, still must be added to the grammar file. The type can be defined as one of the four following: "anytime", "too early / too late", "stopped too soon / applied too long" or "never". This way, control action types and hazard subcategories are bound into the rule definition. For example, a rule which has received type "never" will only potentially affect cells in the "Never" column when control action type "both" is selected, or in the "Hazardous?" column if control action type "not provided" is selected. Lastly, the rule also references a "`HazardList`", which, in the DSL's grammar file, is defined as a non-empty list of existing `Hazard` items, meaning the rule must refer to at least one `Hazard`.

With this Rule definition, the context table view is able to adapt to the DSL's grammar structure. In addition, the context table view is able to get all rule data necessary to update the table's "Hazardous?" column. This, according to the concept presented in Figure 4.1, encompasses the control action, its type or a hazard subcategory, context variables with assigned values and the referenced hazards. In conclusion, it is a concept fitting for both integration into the DSL and use for the context tables.

Chapter 5

Implementation

The context table implementation encompasses the following parts: the *context table view*, *view content assembly*, *language server communication* and *Rules*, which are discussed in this chapter. Section 5.1 explains the new view for context tables and its integration into the DSL. Following this, Section 5.2 describes the process of constructing the view's visual components. Section 5.3 deals with the communication between the view and language server, which is needed to exchange the data necessary to build the context tables. Finally, Section 5.4 focuses on the rules, their integration into the DSL's grammar and their functionality in the extension.

5.1 Context Table View

The context tables are displayed in a new view closely following the concept discussed in Section 4.1. The view has been integrated into the DSL by extending its startup files in the *extension* folder with a command that makes the view available once the extension has been activated. In order to display the context table view, the user presses a newly added button located next to the one starting the diagram view introduced in Section 2.2.1.

Once the button has been pressed, the command for the context table view triggers the activation of "ContextTablePanel.ts", which defines the general behavior of the context table view. This includes the restriction to only one table view being able to exist at a time, as well as it being opened up in a new panel right next to the current STPA file, as can be seen in Figure 5.1. Furthermore, it provides update functions ensuring that the context tables are updated every time the user saves changes in the current STPA file.

5.2 Assembling the View's Content

The view possesses three main visual components, as can be seen in Figure 5.1: two selection elements and a context table. Additionally, text has been added to clarify the use of the components. Both selection elements allow the user to choose the context table to display. The upper selection element provides all available control actions listed with their respective controllers as options. Choosing a different control action will cause the view to update itself and display the context table for the now selected control action. On the contrary, the selection element below allows the user to switch between three types to be paired with the current control action in the context table: "provided", "not provided" and "both". All three types function as described in Section 4.2.

The displayed context table is structured into three main column types: "Control Action", "Context Variables" and "Hazard" columns. The "Control Action" column displays the currently selected control action and the paired type. Following this, the "Context Variables" column lists all context variables (see Section 2.3) as sub-columns. In these sub-columns cells, the respective variables are assigned values. Each row of the context table contains a different combination of assigned context variable

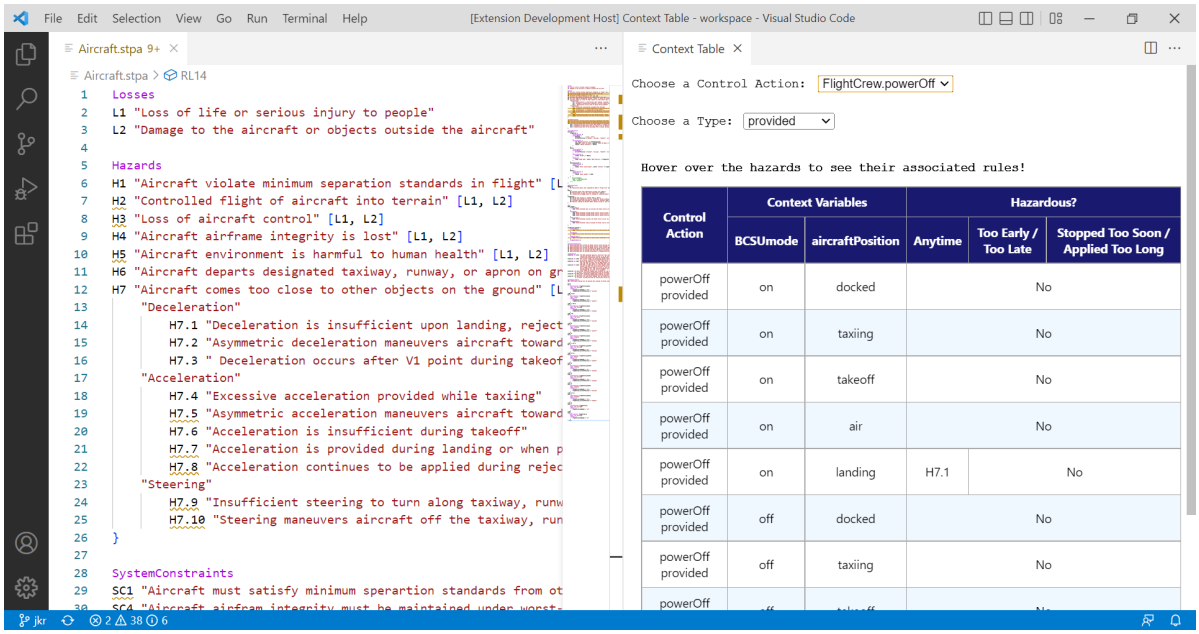

Figure 5.1. The context table view in the STPA DSL extension.

values, creating the environments to analyze the control action in. Finally, the "Hazard" column structure varies depending on the currently selected type, as discussed in Section 4.2.

In order to build the view's visualization components, "ContextTablePanel.ts" constructs a plain Hypertext Markup Language (HTML) structure fitted to the view's frame and integrates a script called "Main.ts" to add the desired components as HTML sub-elements. The mentioned script lies within a new folder called "context-table" and receives the necessary data (control actions, context variables and rules) to build the visual components from a message sent by "ContextTablePanel.ts".

After the data has been received, the selectors are created as HTML selection elements. While the control action selector uses the items of the obtained list of control actions as its options, the type selector uses a predefined list to build its options. If the message from "ContextTablePanel.ts" was sent because of saved changes in the STPA file, "Main.ts" remembers and maintains the selectors' currently chosen options on updating the view. Furthermore, both selectors attain a listener each that triggers when a different option is selected. These listeners are responsible for re-initializing the displayed context table with the selected options.

After the selectors have been fully assembled, the context table is created. This is done by first collecting the data to assign to the table, namely the current control action, its controller and context variables, and the current type. In the STPA file's control structure, context variables are controller attributes. "Main.ts" receives them as a list containing context variable lists, one for every controller each. Since the control action selector displays the actions coupled with the controller they belong to, its current option for the controller is checked, then the list of context variables belonging to it is taken.

After gathering the necessary data, the table's rows are created. First, the upper header row marking the three main column types is created. Then, the second header row containing the context variables and, depending on the selected type, sub-columns for the "Hazard" column, is added. Finally, the regular rows are created: First, a value combination of the collected context variables is constructed, which is then used to create a row. This is done by first adding the control action paired with the type

to the "Control Action" column, then the given values to their respective context variable.

Lastly, it is checked if Rules apply to the row. An iteration through the received list of Rules is done, comparing each of their data to the row's control action, context and type. The iterative process lasts until either a fitting rule is found or all rules have been checked. If no rule applies to the row, all the "Hazard" column's cells are combined and filled with a "No", as can be seen in the first row of Figure 5.1. On the contrary, if a rule applies, its referenced hazards are written into the affected cell, which, in addition, gains a flag displaying the rule ID when the user hovers the mouse cursor above the cell.

Once rows for all possible value combinations have been created, the table has been fully constructed. In order to make the visual components more user-friendly, Cascading Style Sheets (CSS) is used for styling the context table. The colors were chosen to be mellow and working with both VS Code's light and dark color themes.

5.3 Language Server Communication

In order to build the context tables, the view requires certain data from the current STPA file, namely the controllers' control actions and context variables, as well as the rules. As all STPA files' data is maintained by the language server implemented in the "language-server" folder, "context-dataProvider.ts" has been added to it, which gathers the needed data when prompted.

The context table view communicates with this class by sending and receiving back notifications (see Figure 5.2). First, the activation of the context table view triggers the sending of the current STPA file's Uniform Resource Identifier (URI), which acts as the file's resource ID within the extension. This URI notification is also sent every time changes in the current STPA file are saved while the context table view is active. In addition to this, the extension creates a listener waiting for a return notification. The language server catches the URI notification and collects the required context table data from the STPA document found with the received URI. Since components outside the language server cannot work with types such as `Rule` and `Hazard`, the attributes of those types, all of which have the `String` type, are collected instead and put into lists.

After gathering the `String` data for the context tables, the language server sends a notification containing the data back to the extension, which consequently delegates over to "ContextTablePanel.ts". The latter checks the data for completeness and validity, then sends it in a message to "Main.ts". If "ContextTablePanel.ts" evaluates the received data as incomplete or invalid, a notification about this error is sent to VS Code's debug console.

5.4 Rules

As discussed in Section 4.3, rules are used to fill out the context tables' "Hazard" columns with hazard references when they apply. Therefore, the DSL's Langium grammar has been extended with the `Rule` aspect. This implementation is shown in Listing 5.1: The language model gained an additional component called "`Rules`", in which a list of `Rule` type language elements can be defined.

The `Rule` type is defined directly below: Each new rule must first receive an ID, after which the behavior of the rule must be defined in a block marked by braces. In order to completely define a `Rule`, a control action, a type and a context needs to be given. The `controlAction` is written with syntax "`<Controller>.<Action>`" (see Listing 5.2), with the controller being defined as the `Node` type and the control action as the `Command` type in the grammar. Knowing the controller to every control action is

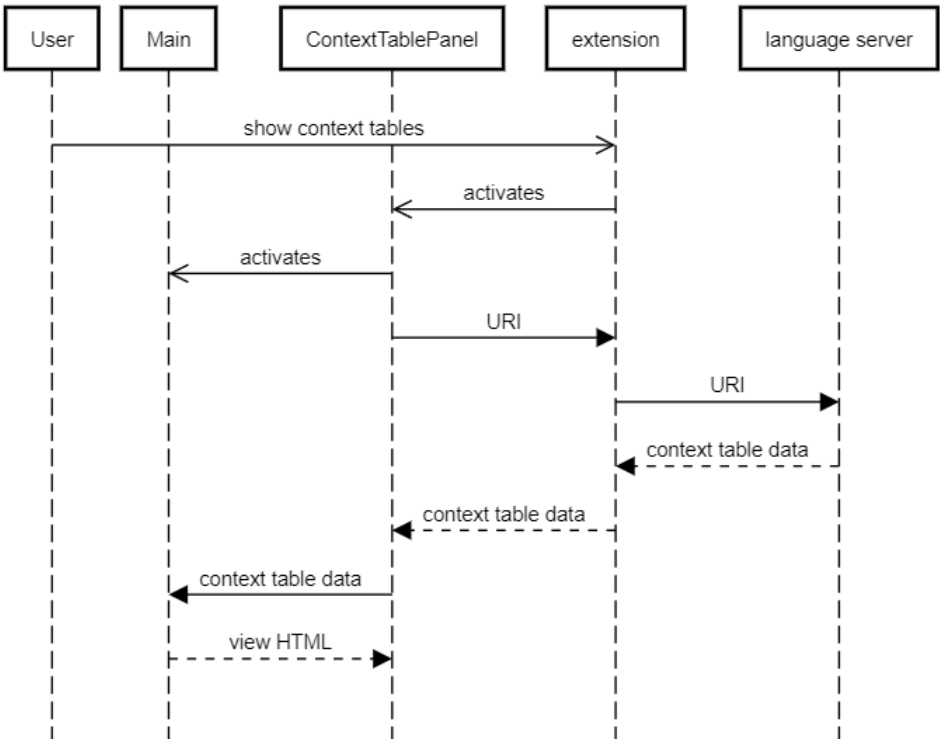

Figure 5.2. The communication between the DSL components when the user starts the context table view.

important for creating the context tables, as the controller defines the context variables that apply to the action.

The `Type` (see Section 4.3) is introduced as another new language type, which is defined as a `Sting` element. Technically, any `String` can be used for the definition of a `Rule`. However, to have a `Rule` apply to the context tables, the given `String` must adhere to the following norms:

1. For type “provided”, the `Strings` “`provided`” or “`anytime`” can be used.
2. For type “not provided”, the `Strings` “`not provided`” or “`never`” can be used.
3. For type “too early / late”, the `Strings` “`too early`” or “`too late`” can be used.
4. For type “stopped too soon / applied too long”, the `Strings` “`stopped too soon`” or “`applied too long`” can be used.
5. Any uppercase letter variants of the above-mentioned `Strings` apply as well. For instance, the `Strings` “`Provided`” and “`anyTime`” will also be evaluated as type “provided”.

The `context` must contain at least one context variable belonging to the defined control action’s controller, therefore written with syntax “`<Controller>.<Variable> = <Value>`”. These variables must have a value assigned to them each. As can be seen in Listing 5.1, this can technically be any `String`. However, in order to have the `Rule` apply to a context table, the value must be one of the values defined for the respective variable defined in the control structure. Furthermore, after completing the `Rule` block, a list of hazards referencing at least one `Hazard` element needs to be defined for the rule.

```
grammar Stpa

entry Model
        ('Losses' losses+=Loss*)?
        ('Hazards' hazards+=Hazard*)?
        ('SystemConstraints' systemLevelConstraints+=SystemConstraint*)?
        ('ControlStructure' controlStructure=Graph)?
        ('Responsibilities' responsibilities+=Resps*)?
        ('UCAs' allUCAs+=ActionUCAs*)?
        ('ControllerConstraints' controllerConstraints+= ContConstraint*)?
        ('LossScenarios' scenarios+=LossScenario*)?
        ('SafetyRequirements' safetyCons+=SafetyConstraint*)?
        ('Rules' rules+=Rule*)?;

Rule:
        name=ID '{'
                'controlAction:' system=[Node] '.' action=[Command]
                'type:' type=Type
                'context:' '{'
                        system=[Node] '.' vars+=[Variable] '=' values+=STRING
                        (',' system=[Node] '.' vars+=[Variable] '=' values+=STRING)*
                '}'
        '}'
        list=HazardList;

Type:
        value=STRING;
```

Listing 5.1. The DSL's grammar updated with Rules.

```
RL12 {
        controlAction: FlightCrew.powerOn
        type: "anytime"
        context: {
                FlightCrew.BCSUmode = "off",
                FlightCrew.aircraftPosition = "takeoff"
        }
} [H7.3]
RL13 {
        controlAction: FlightCrew.mc
        type: "not provided"
        context: {
                FlightCrew.BCSUmode = "off"
        }
} [H3]
```

Listing 5.2. Rules in the STPA DSL extension.

Listing 5.2 depicts how defined rules look in the STPA file. Since rules, just like the control structure, are defined by a variety of attributes, their definition was designed to look similar to the definition of said structure (see Figure 2.2). This way, the rules fit in with the DSL's other aspects.

Chapter 6

Conclusion

This chapter concludes the thesis with Section 6.1, which gives a summary of the thesis. Finally, Section 6.2 will give an outlook on future work that can be done to improve the context tables.

6.1 Summary

In this thesis, context tables have been implemented into the STPA DSL built by Petzold in order to make the identification of UCAs more manageable and time-efficient for users. The implementation consists of a textual and a visual component: The context tables are integrated into the extension as a new view similar in function to the DSL's diagram view. This table view can be displayed next to the current STPA file and helps the user to conduct a complete UCA analysis of the control structure. On the contrary, rules have been added to the DSL's grammar as a textual component the user can define in the STPA file. Defined rules reference hazards and apply them to context table rows they affect.

As the implementation creates context tables for all controllers' control actions with all possible contexts, it offers a method that guarantees a complete analysis of the control structure with the aim to identify all possible UCAs. However, instead of filling out the context tables manually, this is done solely by textually defining rules. Rules have the ability to affect multiple table rows depending on its defined context, therefore making the process of filling out the context tables more time-efficient. Once rules have been defined and the changes to the STPA file have been saved, the tables update automatically and adapt to any additions or changes made to the rules and control structure. Thus, the user can keep a consistent overview over the made progress regarding identified UCA. As soon as rules apply to the tables, the user can read UCAs from their respective rows and manually write them into the STPA's UCA section.

In conclusion, the context table implementation provides options to make the UCA identification process more user-friendly, manageable and time-efficient. Therefore, the STPA tools viability has been increased. Nonetheless, improvements to the context table can be made such as the automatic generation of UCAs from the context tables.

6.2 Future Work

The context table implementation can be improved by a variety of features. As discussed in Section 6.1, with the current context table implementation, while the tables are automatically generated and defined rules immediately apply to them, the user still needs to manually read the UCAs from the tables and write them into the STPA file. However, since the context table view already contains all necessary data (controller, control actions, type, context, hazards) to construct UCAs, a feature that lets the user directly generate UCAs from the tables can be implemented. This feature would reduce the manual work for the user to identifying rules, with everything else being generated automatically as soon as the control structure and rules are defined.

It is possible for multiple Rules to apply to one hazard cell in a table's "Hazard" column. Currently, the view will take the first one it finds and then stop the search, therefore ignoring potential other Rules that also apply to the cell. In order to give users an opportunity to analyze these Rule conflicts, a feature that marks conflicting Rules with a warning in the STPA file can be implemented. Furthermore, while Rule types applying to the context tables are predefined by norms described in Section 5.4, the DSL does not make the user aware of these type limitations. Another warning feature could be added to the Rule type, which informs the user if an invalid type has been chosen for a Rule.

The context tables can still be made more assessable to users by implementing an option which simplifies them. Figure 6.1 shows the general idea of the simplification in the first three rows: Context variable "`Train Position`" has two known values, but its value assignment does not affect the result of the "Hazard" column in those rows. Consequently, these originally six rows were combined to three rows, making the context table more manageable without any context being lost. This can be especially valuable for context tables with a great amount of context variables. However, since in the DSL context tables are initialized with all "Hazard" column cells displaying "No" if no rules are defined, this simplification method will always reduce the tables to one row in those cases. Therefore, it is best implemented as an option for the user to choose when it best suits them.

Control Action	**Train Motion**	**Emergency**[10]	**Train Position**	**Hazardous control action?**		
				If provided any time in this context	**If provided too early in this context**	**If provided too late in this context**
Door open command provided	Train is moving	No emergency	(doesn't matter)	Yes	Yes	Yes
Door open command provided	Train is moving	Emergency exists	(doesn't matter)	Yes[11]	Yes	Yes
Door open command provided	Train is stopped	Emergency exists	(doesn't matter)	No	No	Yes
Door open command provided	Train is stopped	No emergency	Not aligned with platform	Yes	Yes	Yes
Door open command provided	Train is stopped	No emergency	Aligned with platform	No	No	No

Figure 6.1. Example of a simplified context table [Tho13].

Finally, an additional feature can be implemented that, when the user clicks on a hazard reference in a table's "Hazard" column, the STPA file jumps to the definition of this hazard. This way, the readability of the context tables can be increased, and the resulting UCAs can be logically connected to the rest of the conducted STPA even before they are written down.

Bibliography

[AW16] Asim Abdulkhaleq and Stefan Wagner. “XSTAMPP 2.0: New Improvements to XSTAMPP Including CAST Accident Analysis and an Extended Approach to STPA”. In: *MIT* (2016).

[GHB15] Danilo Lopes Gurgel, Celso Massaki Hirata, and Juliana De M. Bezarra. “A Rule Based Approach for Safety Analysis Using STAMP/STPA”. In: *IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)* (2015).

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

[ISO96] ISO. “Iso/iec 14977:1996(e), Information technology — Syntactic metalanguage — Extended BNF”. In: *International Organization for Standardization* (1996).

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

[Pet22] Jette Petzold. “A Textual Domain Specific Language for System-Theoretic Process Analysis”. MA thesis. Kiel University, 2022.

[SPP+19] Fellipe G. R. Souza, Daniel P. Pereira, Rodrigo M. Pagliares, Simin NadjmTehrani, and Celso M. Hirata. “WebSTAMP: a Web Application for STPA and STPA-Sec”. In: *MATEC Web of Conferences* (2019).

[ST14] Dajiang Suo and John Thomas. “An STPA Tool”. In: *STAMP 2014 Conference at MIT* (2014).

[Tho13] John Thomas. “Extending and Automating a Systems-Theoretic Hazard Analysis for Requirements Generation and Analysis”. PhD thesis. Massachusetts Institute of Technology, 2013.

List of Abbreviations

DSL	Domain Specific Language
STPA	System-Theoretic Process Analysis
UCA	Unsafe Control Action
VS Code	Visual Studio Code
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
IDE	Integrated Development Environment
EBNF	Extended Backus-Naur Form
XSTAMPP	Extensible STAMP Platform
STAMP	System-Theoretic Accident Model and Processes
MIT	Massachusetts Institute of Technology
LTL	Linear Temporal Logic
PDE	Eclipse Plug-in-Development Environment
RCP	Rich Client Platform
URI	Uniform Resource Identifier