Safety Analysis of the Steam Boiler in SCCharts

Tokessa Hamann

Bachelor's Thesis
September 30, 2024

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.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

Risk analysis is a crucial aspect of system development to ensure safety and minimize potential losses. One of these analysis is System-Theoretic Process Analysis (STPA), developed by Leveson, which is used during system development. Although STPA is effective in identifying safety flaws, conducting the analysis manually, is time-consuming and complex. To streamline the process, Petzold developed the Pragmatic Automated System-Theoretic Process Analysis (PASTA) tool, which fully supports STPA and aims to make the procedure more efficient. However, PASTA has not yet been tested on larger systems, which is the focus of this thesis. For this reason, STPA is applied to the steam boiler system specified by Abrial et al. and the performance of PASTA on this larger system is evaluated.

In parallel, the thesis also models the steam boiler using Sequentially Constructive Statecharts (SCCharts), a visual language suited for specifying safety-critical reactive systems. SCCharts have been previously used to model the steam boiler with a focus on manual verification of automatic generated code and object-oriented features, but not with an emphasis on verification of the steam boiler model. This thesis shifts the focus towards verification using Linear Temporal Logic (LTL) formulas.

The evaluation demonstrates that both PASTA and SCCharts are capable of handling larger systems effectively, but also identifies missing features that could enhance the visualization and usability of the tools. Recommendations are proposed to address these limitations and improve the overall user experience in future iterations.

Acknowledgements

First of all, I want to thank Prof. Dr. Reinhard von Hanxleden for the opportunity to write this thesis and for the constructive and helpful feedback during its conception.

Moreover, I want to thank the entire Real-Time and Embedded Systems group for their warm welcome. In particular, I want to express my gratitude to my advisor M. Sc. Jette Petzhold, for her continuous guidance and support during both the analysis and writing process. I would also like to thank Dr.-Ing. Alexander Schulz-Rosengarten for his insightful ideas on modeling the steam boiler and his assistance with challenges related to SCCharts.

Furthermore, I am deeply thankful to my family for their steadfast support throughout my studies, especially my sister Florina Hamann who proofread parts of this thesis. Finally, I would like to thank my friends who have always supported me. Special thanks go to Finn Evers, Merlin Felix and Yorik Hansen for their feedback, help and good company during the entirety of this thesis and especially during the last days.

Contents

1 Introduction **1**

1.1 Problem Statement 1

1.2 Outline 2

2 Preliminaries **3**

2.1 Steam Boiler 3

2.2 System-Theoretic Process Analysis 5

2.2.1 Define Purpose of the Analysis 5

2.2.2 Model the Control Structure 6

2.2.3 Identify Unsafe Control Actions 6

2.2.4 Identify Loss Scenarios 7

2.2.5 STPA Outputs and Traceability 9

2.3 Context Tables 9

2.4 Sequentially Constructive Statecharts 11

2.5 Linear Temporal Logic Formulas 13

2.6 Used Technologies 14

2.6.1 KIELER 14

2.6.2 Pragmatic Automated System-Theoretic Process Analysis 15

3 Related Work **17**

3.1 STPA Tools 17

3.1.1 STAMP Workbench 17

3.1.2 STPA Capella 19

3.1.3 SafetyHAT 20

3.2 Risk Analyses of the Steam Boiler 21

3.2.1 Fault Tree Analysis (FTA) of the Steam Boiler 22

3.2.2 Hazard and Operability Analysis (HAZOP) on a flame tube boiler 22

3.2.3 HAZOP on a coal-fired boiler 24

3.3 Models 24

3.3.1 The Steam Boiler in Statecharts and Z 24

3.3.2 The Steam Boiler in SCCharts 26

3.4 Verification 27

4 The Steam Boiler in PASTA **29**

4.1 Analysis 29

4.1.1 Define Purpose of the Analysis 29

4.1.2 Model the Control Structure 31

4.1.3 Identify Unsafe Control Actions 36
4.1.4 Identify Loss Scenarios 41
4.2 Challenges 43
4.2.1 Automatic Identifier Updates 43
4.2.2 Context Table 44
4.2.3 Loss Scenarios 45

5 The Steam Boiler Model in Sequentially Constructive Statecharts 47
5.1 Modelling 47
5.1.1 Steam Boiler Modelling Process 47
5.1.2 Approaches for Developing a Verifiable Model 54
5.1.3 Verification 55
5.2 Challenges 58

6 Evaluation 61
6.1 Analysis of PASTA 61
6.1.1 User guidance 61
6.1.2 Visualization 63
6.1.3 Language Server Updates 64
6.1.4 Context Table 65
6.1.5 Loss Scenarios 66
6.2 Steam-boiler SCChart Model 67
6.2.1 User Guidance 67
6.2.2 Visualization 68
6.2.3 Debugging 69
6.2.4 Comparison to other Models 70

7 Conclusion 73
7.1 Summary 73
7.2 Future Work 73
7.2.1 Evaluation of PASTA's LTL Formulas 74
7.2.2 Steam Boiler Model 74
7.2.3 Evaluation of KIELER's Model Checker 75

Bibliography 77

List of Abbreviations 81

List of Figures

2.1 Physical units of the steam boiler [SDH19]. 4
2.2 Control structure door controller [Tho13]. 7
2.3 Traceability between STPA outputs [Lev18]. 9
2.4 SCCharts elements [Sch24b]. 12
2.5 ABO SCChart [HDM+14]. 13
2.6 Illustration of the intuitive semantics of LTL [BK08]. 14

3.2 Stakeholder table in STPA Capella [Con24]. 19
3.3 Loss creation and tracing in STPA Capella. 20
3.5 Fault tree of the water level [PTL13]. 23
3.6 Main components and their services in the architectural model [BW96]. 25
3.7 The steam boiler divived in four subsystems [dT06]. 26

4.1 Losses, hazards and system-constraints of the steam boiler. 30
4.2 The steam boiler control strcuture. 31
4.3 Process model of the Operator in code form and as diagram. 34
4.4 Challenges with hazard IDs. 44

5.1 Abstracted steam boiler model 48
5.2 The booting of the physical units. 49
5.3 Unit fail protocol of the physical units. 51
5.4 Unit fail protocol of the program. 51
5.5 Program2 SCChart. 52
5.6 Model of the water level sensor. 53
5.7 Water level ranges. 54
5.8 The Valve in SCCharts. 56
5.9 The model checker table in KIELER. 57

7.1 Physical units of the steam boiler [SDH19]. 75

List of Tables

2.1 Unsafe Control Actions (UCAs) for the *open door* control action [Tho13]. 8
2.2 Context Table for automated door controller [Tho13]. 10
2.3 Rule Table for *open door* control action. 10
2.4 Process model flaws [Tho13]. 11

Chapter 1

Introduction

Risk is an inherent aspect of human life and a constant factor in our daily existence [AR10]. As technologies have become an increasingly significant part of people's day-to-day lives, safety in complex systems became of high importance, particularly in industries such as transportation, energy, and manufacturing, where system failures can lead to fatal consequences. Diverse hazard analysis techniques have been developed in order to improve system safety. One of them is the System-Theoretic Process Analysis (STPA) [Lev18]. It is a safety analysis method designed to identify potential hazards by focusing on unsafe interactions between components rather than failures of individual components. However, despite its well-structured approach, applying STPA manually can be a complex, and error-prone process, especially for intricate systems, as it often requires significant time and effort without software support. While visual diagrams illustrating relationships between STPA elements can aid understanding, creating them manually is labor-intensive. In order to aid in this matter, Petzold developed a tool aimed at systematizing STPA named Pragmatic Automated System-Theoretic Process Analysis (PASTA) [PKH23]. It is intended to streamline the safety analysis process by offering structured workflows and automation features. However, its effectiveness on larger, more intricate systems has not yet been thoroughly tested, presenting a gap in understanding its scalability and practicality.

One well-established system used for academic research and testing of safety methodologies is the steam boiler problem, first specified by Abrial et al. [Abr05]. The steam boiler provides a well-defined and sufficiently complex system for testing tools like PASTA.

SCCharts are particularly well-suited for safety-critical systems due to their deterministic nature and the fact that they eliminate problems related to race conditions. These features make SCCharts a powerful tool for modelling systems that require high reliability and predictability. Two SCChart models of the steam boiler system already exist, the first one focused primarily on the evaluation of how well manual user verification can be done on source code automatically generated by the model compiler [SDH19] and the second one explored object-oriented features in SCCharts [Sch24a]. Nonetheless, no model exists that focused on the verification of the system.

1.1 Problem Statement

In order to close the gaps mentioned above, this thesis seeks to evaluate how effectively PASTA supports the application of STPA on a larger system such as the steam boiler by conducting a STPA and to uncover potential challenges or limitations that may arise. In addition to the safety analysis, the steam boiler will be designed using SCCharts, with a focus on verifying the system's behaviour. The verification of the model will be done with the help of Linear Temporal Logic (LTL) formulas. The objective is to evaluate the applicability and limitations of SCCharts when modelling a larger system focused on verification.

1.2 Outline

The next chapter introduces the foundations for this thesis, such as the steam boiler specification, the STPA process, SCCharts, LTL formulas and used technologies. Chapter 3 reviews existing tools supporting STPA and other risk analyses conducted on the steam boiler. Furthermore, it presents existing steam boiler models and their verification approaches from the literature. In Chapter 4, the process of the steam boiler STPA in PASTA is explained and it addresses the challenges that were encountered during the analysis. Afterwards, Chapter 5 describes the process and approaches used to model the steam boiler in SCCharts. It shows the verification process on a small example and addresses the challenges that arose during the modelling. In Chapter 6, the features of PASTA and SCCharts are evaluated for the usage of analysing or modelling the steam boiler. Furthermore, additions to the tools are proposed that would improve the user experience. The chapter also compares the different SCChart models to show their differences and similarities. Finally, Chapter 7 concludes the thesis with a summary and potential future work.

Chapter 2

Preliminaries

It is essential to have a foundational knowledge of STPA, SCCharts, LTL formulas and the technologies used, to understand the evaluation of PASTA and the modelling method in SCCharts. Section 2.1 outlines the steam boiler specification, which is used in the following analysis and modelling. After that, Section 2.2 explains the STPA process and Section 2.3 the context tables that can be used for its third and fourth step. Section 2.4 introduces SCCharts and Section 2.5 the LTL formulas, which can be used to verify the SCChart model. Subsequently, Section 2.6 discusses the technologies used for the analysis and implementation.

2.1 Steam Boiler

The *Steam Boiler Specification Problem* [Abr05] is a well-established system used for academic research and testing of safety methodologies. The behaviour of the control program of the steam boiler is not trivial, as the requirements lay emphasis on a fault-tolerant behaviour of the system. Figure 2.1 illustrates the system defined in the specification, which consists of the following entities:

- the steam boiler
- a valve to evacuate water
- a device to measure the amount of water in the steam boiler
- four pumps to inject water into the boiler
- four pump controllers
- a device to measure the output steam from the boiler
- an emergency stop switch
- the message transmission system between the steam boiler and the control program

These entities are physically limited. The water level must lie in-between M_1 and M_2, because the steam boiler could be seriously damaged otherwise. It ideally lies between the safety levels N_1 and N_2 for optimal steam production, while C denotes the maximal capacity of the steam boiler. Furthermore, the variables U_1 and U_2 are the maximum gradient increase and decrease of steam respectively.

The overall operation of the program consist of the reception of messages from the physical units, their analysis, and a following transmission to the units with new control commands. For this purpose, the program takes the global state of the physical environment into account and switches, based on these states, to five different modes: *initialization*, *normal*, *degraded*,

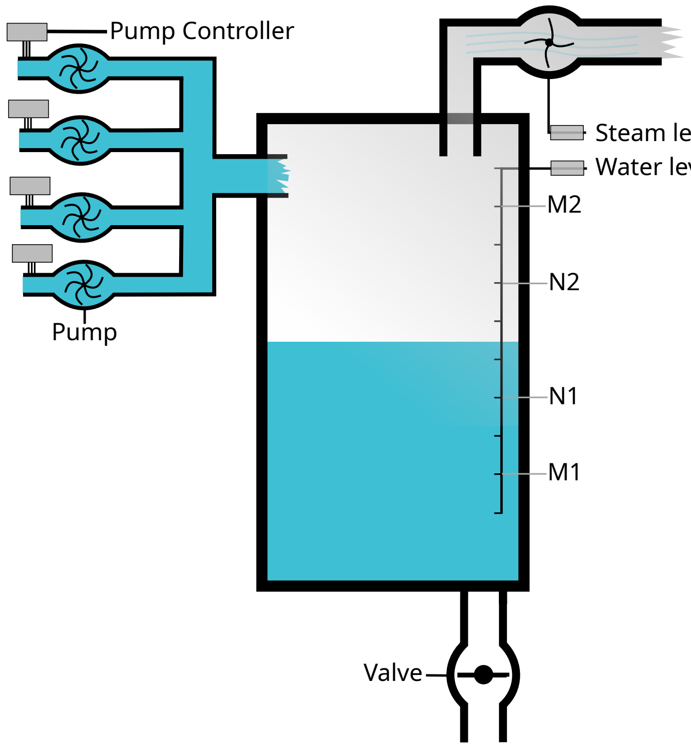


Figure 2.1. Physical units of the steam boiler [SDH19].

rescue and *emergency stop*. In the *initialization* mode, the program waits for the units to get ready and then ensures that the water level is between normal operating limits. It does this by either emptying the water via the valve or increasing the water level by activating the pumps. In *normal* mode, the program tries to maintain the safety water level. This is done by activating and deactivating the pumps alone, because the controller cannot control the heat of the steam boiler. It will stay in this mode until a failure is detected. The *degraded* mode tries to maintain a safe water level despite failures of one or more non-vital pieces of equipment. The mode changes back to *normal* after the components got repaired by the operator. When the water sensor has failed, the program goes into *rescue* mode, where the water level needs to be estimated via the other sensor information for operating the steam boiler. Should an error also be detected in these sensors, the program goes into *emergency stop* mode. In the *emergency stop* mode, the program has been instructed to stop, because either the water level is near to one of the critical limits, multiple pieces of equipment have failed, there is a transmission failure between the program and physical equipments, or the operator stopped the program manually.

2.2 System-Theoretic Process Analysis

System-Theoretic Process Analysis (STPA) [Lev18] is a hazard analysis technique based on System Accident Model and Processes (STAMP), which is used to identify potential causes of accidents during a system's development and develop a safe system. In contrast to traditional hazard analysis techniques, such as Fault Tree Analysis (FTA) [Ves81], Failure Modes and Effects Criticality Analysis (FMECA) [Rob93], Event Tree Analysis (ETA) [II05], or Hazard and Operability Analysis (HAZOP) [Kle99], where the hazards arise from component failures, STPA also considers emergent properties that lead to accidents. This means that STPA assumes that accidents can also arise from unsafe interactions among system components, none of which have necessarily failed. As a result, STPA finds all the scenarios that traditional analyses can find, along with additional scenarios previously undetected. The results of STPA can be used to create new requirements, identify mitigations, evaluate design decisions, define test cases, develop leading indicators of risk, etc. [Lev18]. STPA is an iterative process that can be applied before an accident has occurred and then updated and modified over time when new information arises. The analysis of the system is split into four consecutive steps, which will be discussed in more detail in the next subsections.

2.2.1 Define Purpose of the Analysis

In the first step, the purpose of the analysis is defined. For this, *stakeholders* and *losses* need to be identified. A *loss* has hereby a broader definition. It involves something of value for the stakeholder, e.g. loss of life, damage to environment, or loss of revenue.

Afterwards, the system that needs to be analysed and its boundaries are specified, so that system-level *hazards* can be identified. The system is a set of components that work together to achieve some common goal. It may contain subsystems or be part of a larger system. A *hazard* is a system state or set of conditions that lead to a loss in worst-case environmental conditions. It can be further refined into sub-hazards.

For the hazards, system-level *constraints* are defined, which specify system conditions or behaviours that need to be satisfied to prevent hazards. Each sub-hazard should also be covered by a sub-constraint.

As an illustrative example, consider a train with a basic automated door control system [Tho13]. A *loss* that could be studied is the loss of life or injury to a person. This results in the following hazards:

H-1: Doors close on a person in the doorway [L-1]
H-2: Doors open when the train is moving or not in a station [L-1]
H-3: People are unable to exit during an emergency [L-1]

System-level constraints can then be derived from these hazards:

SC-1: Doors must stay open if a person is in the doorway [H-1]
SC-2: Doors must stay closed when the train is moving or not in a station [H-2]
SC-3: Doors must be able to open during an emergency [H-3]

2.2.2 Model the Control Structure

In the second step, a structure capturing functional relationships and interactions between system components is built, called *control structure*. It is composed of feedback control loops that enforce constraints on the behaviour of the overall system modelled. A control loop consists of a controller, controlled processes as well as control-feedback loops between the controllers with each other or controlled processes. The controller sends *control actions* to enforce constraints on a controlled process. Moreover, it has a *control algorithm* that determines the action of the controller and a *process model* that represents the controller's beliefs about its environment. The vertical axis of the control structure indicates control and authority of the different components. In this structure, downward arrows represent the control actions and upward arrows the feedback of the control-feedback loops.

During this, *responsibilities* can be assigned to each entity of the control structure. The *responsibilities* are refinements of the system-level constraints that define what each entity needs to do, so that as a unit they enforce the constraints.

Coming back to the door controller example, a possible *control structure* could look like Figure 2.2. The control structure consists of the *automated door controller* with its control algorithm and process model. It gets input information about the train and emergencies. Furthermore, it commands the *door actuator* to open or close the door, which in turn uses mechanical force to move the *physical door*. The door then gives feedback of its mechanical position to the *door sensor*, which sents the feedback of the position and if the door is clear back to the controller.

An example of a responsibility for the door actuator can be:
R-1: Open or close the door when commanded by the door controller [SC-1, SC-2, SC-3]

2.2.3 Identify Unsafe Control Actions

In this step, the control structure is analysed to identify control actions that can lead to hazards in a specific context and worst-case environment. These control actions are called Unsafe Control Actions (UCAs) and can be dangerous in four ways:

- 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
- The control action lasts too long or is stopped too soon

These types should be considered for all control actions, although some may not be applicable in every case. Analogous to system-level constraints, *controller constraints* are identified, which define the specific behaviours that controllers must follow to prevent UCAs.

Possible UCAs for the control action *open door* are shown in Table 2.1

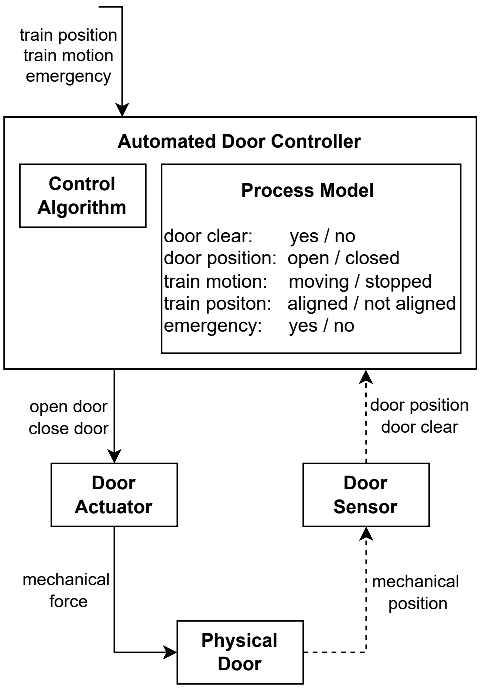


Figure 2.2. Control structure door controller [Tho13].

2.2.4 Identify Loss Scenarios

In the last step, *loss scenarios* are identified. These *loss scenarios* describe causal factors, which can lead to an UCA and therefore hazards. There are two types of loss scenarios that need to be considered in the STPA: Scenarios that lead to an UCA and scenarios where an improperly executed or not executed control action leads to a hazard.

The first case can further be divided into *Unsafe controller behaviour* and *Causes of inadequate feedback and information*. For the first type, the process involves working backward from the UCA to the factors for the loss scenario. There can be different reasons why the controller executes unsafe behaviour, for example: failures of the physical controller, an inadequate control algorithm, unsafe control input from another controller, or an inadequate process model. For finding causes of inadequate feedback and information, the system needs to be analysed to find their source. For the second case, factors that involve the control path as well as factors that relate to the controlled process must be considered to find scenarios that can lead to hazards. It is important that these factors are not reduced to a single one but are seen as a whole scenario. Otherwise, non-trivial cases where components interact between

Table 2.1. UCAs for the *open door* control action [Tho13].

control action	**not provided**	**provided**	**wrong timing or order**	**stopped to soon / applied to long**
open door	UCA-1: Door open command not provided when train is stopped at platform and person in doorway [H-1] UCA-2: Door open command not provided when train is stopped and emergency exists [H-3]	UCA-3: Door open command provided when train is moving and there is no emergency [H-2] UCA-4: Door open command provided when train is stopped unaligned with platform and there is no emergency [H-2]	UCA-5: Door open command is provided more than X seconds too late after train stops during an emergency [H-3]	N/A

themselves and combination of factors leading to hazards may be overlooked.

Applying the fourth step of the analysis to the example above results in the following loss scenarios as example for both cases of *Unsafe controller behaviour* and for *Causes of inadequate feedback and information*:

Scenario 1 for UCA-1: The door actuator fails when the train is stopped at a platform and a person is in the doorway, causing the *open door* control action not to be provided [UCA-1]. As a result, the controller closes the door on a person in the doorway [H-1].

Scenario 2 for UCA-1: The train is stopped at a platform and a person is in the doorway, but the door actuator does not provide the control action to open the door [UCA-1]. This is the case because the door actuator believes that the doorway is clear. This flawed process model will occur if the signal indicating that the doorway is occupied is not received. The indication may not be received when needed if any of the following occur:

- obstruction of the door clear sensor
- door clear feedback is delayed
- failure of door clear sensors
- etc.

As a result, the controller closes the door on a person in the doorway [H-1].

Scenario 3 for UCA-1: The automated door controller sends the *open door* command upon reaching the station, but the door is not opened due to actuator failure. As a result, the controller closes the door on a person in the doorway [H-1].

2.2.5 STPA Outputs and Traceability

The overall traceability between the different elements of the analysis, which were shown in the previous section, can be seen in the traceability diagram in Figure 2.3. The results of STPA can be used for creating requirements for the system, identifying design recommendations, defining test cases, designing more effective safety management systems and more [Lev18].

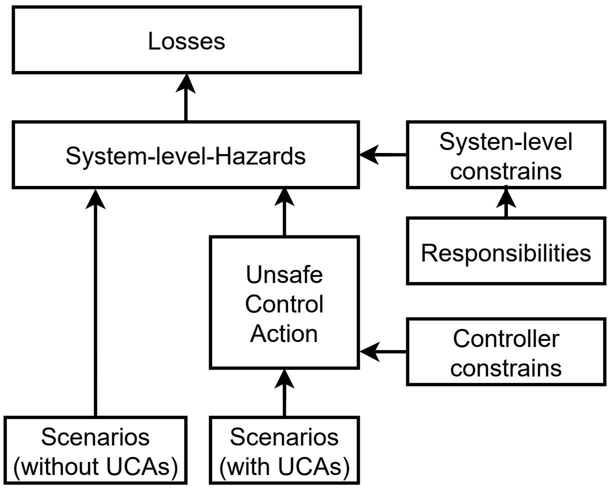


Figure 2.3. Traceability between STPA outputs [Lev18].

2.3 Context Tables

Applying STPA manually can be particularly tedious, especially in complex systems, making partial automation a valuable approach to reduce the workload. Thomas developed an automation method for creating contexts that help identify UCAs, as well as detecting conflicts between safety requirements and other system requirements [Tho13]. Thomas first formalized UCAs, to support the UCA generation process, by splitting the UCAs up in four components: the source controller, type, control action, and context. The context is defined by the process variables of the process model and their respective values. *Context tables* were then introduced [Tho13], to systematically conduct step three of STPA. These tables assist in identifying the full set of UCAs from the system's control structure. Each context table analyses one control action combined with the different UCA types, across all possible contexts.

The context table is structured as follows. In the first column is the control action that is to be examined for hazardous behaviour. All possible combinations of values of the individual process variables of the associated process model are then listed in the following columns. Each row represents a possible combination of the values of the variables, whereby some rows are taken together using the *any* value. The last columns show whether a control action is unsafe. This is determined by the analyst, who decides whether the control action in each row could realistically result in a listed hazard from the behaviour outlined in that context. If this is the case, a new UCA is identified and added to the set of UCAs.

The automated door controller example with the *open door* control action is considered again. In the context where the train is stopped and is aligned with a platform, but people are standing in the doorway, and there is no emergency, hazardous consequences are detected. The hazard occurs when the control action is never or too late provided. This means that a new UCA needs to be defined. Part of the context table for the control action *open door* is shown in Table 2.2. The *stopped too soon / applied too long* column is omitted because it is not applicable in the case of the *open door* control action.

Table 2.2. Context Table for automated door controller [Tho13].

Control Action	Door Clear	Door Position	Train Motion	Train Position	Emergency	never provided	provided anytime	too early / late
open door	any	closed	moving	any	no	no	yes	yes
open door	any	closed	stopped	not aligned	no	no	yes	yes
open door	any	closed	stopped	any	yes	yes	no	yes (too late)
open door	no	closed	stopped	aligned	no	yes	no	yes (too late)
open door	...							

Gurgel et al. [GHD15] introduced a rule-based approach that represents an algorithmic alternative to the manual method of filling context tables. This way, the approach aids in the identification of hazardous contexts, which is otherwise both time-consuming and prone to errors due to the complexity and large number of possible contexts. *Rules* are defined as logical expressions of variable states, and mean that those states represent a hazardous context. For variables where the specific state is irrelevant, the keyword *any* is used, allowing flexibility in rule creation. These rules are defined directly before creating the context tables, with each rule specifying the conditions under which a control action is considered unsafe. For instance, a rule table for the *open door* control action can be seen in Table 2.3. The first rule applies when the train is moving and there is no emergency and the second rule applies if there is no emergency, but the train is not aligned with the platform.

Table 2.3. Rule Table for *open door* control action.

Index	Door Clear	Door Position	Train Motion	Train Position	Emergency
R1	any	any	moving	any	no
R2	any	any	any	not aligned	no

Once the rules are established, they must be checked for conflicts, redundancy, and correctness. The context tables are then created using these rules, systematically confirming, which rows are hazardous, rather than manually checking each one. For example, all rows in the context table where *Train Motion* is *moving* and *Emergency* is *no* would be judged as hazardous, when the first rule is applied. This rule-based method significantly reduces the effort and time required for the process by partly automating the identification of hazardous contexts, making it more efficient and adaptable to changes. It shifts the focus of the UCAs identification process from analysing each context manually to refining and managing the rules, thus enhancing both the accuracy and the efficiency of the analysis.

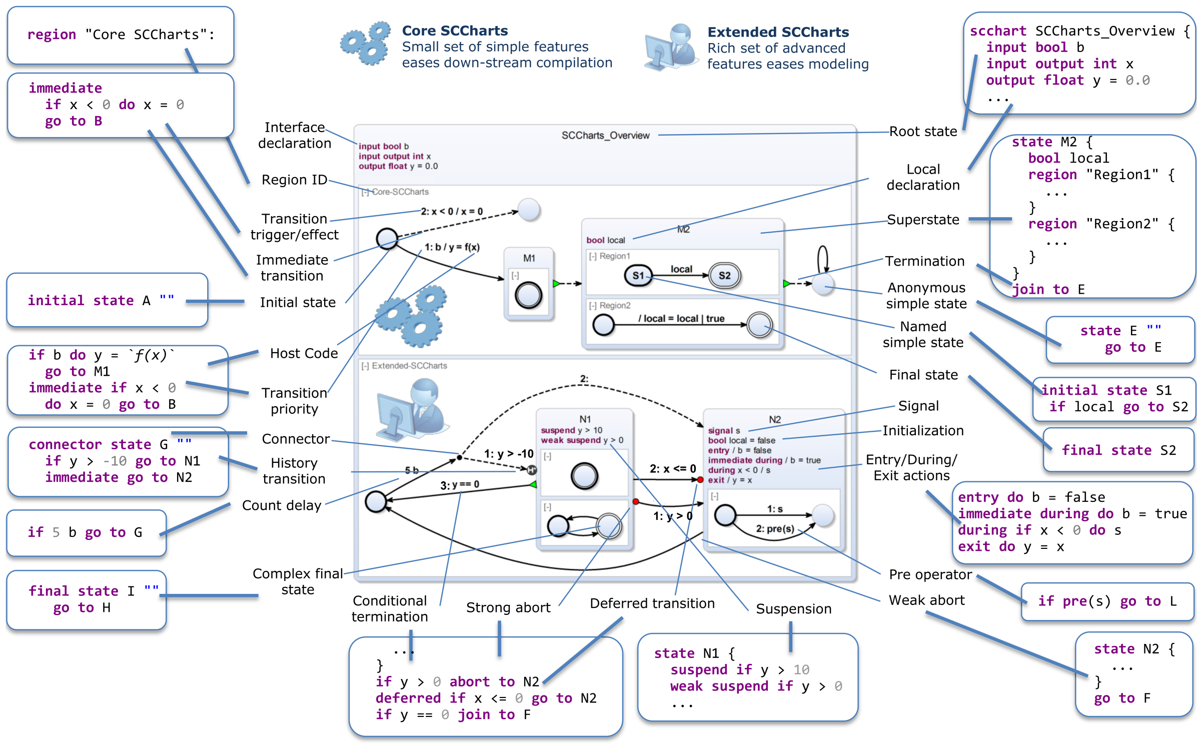
However, context tables can not only help with step three of STPA, but also with the fourth step, the identification of loss scenarios. They assist in this process by revealing scenarios that lead to UCAs by uncovering a flawed process model. Some possible types of process model flaws can for example be seen in Table 2.4. This technique also provides traceability by allowing the exact hazards relevant to each process model flaw to be readily determined from the existing context tables. Given the UCAs and context tables, basic scenarios can even be generated automatically, but identifying causal factors still require manual effort. However, this approach reduces redundancy between UCA identification and scenario analysis, ensuring that only scenarios that definitively lead to a loss are considered.

Table 2.4. Process model flaws [Tho13].

Hazardous control action	Process model flaws
Open door command provided when train is moving	Controller incorrectly believes train is not moving
Open door command provided when train is stopped unaligned and there is no emergency	Controller incorrectly believes train is aligned Controller incorrectly believes there is an emergency
Open door command not provided when train is stopped and an emergency exists	Controller incorrectly believes train is moving Controller incorrectly believes no emergency exists
Open door command not provided when train is stopped at platform and person in doorway	Controller incorrectly believes train is moving Controller incorrectly believes the door is clear Controller incorrectly believes train is not aligned

2.4 Sequentially Constructive Statecharts

Sequentially Constructive Statecharts (SCCharts) are a synchronous statecharts dialect with Sequentially Constructive (SC) semantics. It was introduced by von Hanxleden et al. [HDM+14] for specifying safety-critical reactive systems. SCCharts use a similar notation to statecharts [Har87] and are inspired by SyncCharts [And96]. Furthermore, they are based on the SC Model of Computation (MoC), which provides them with deterministic concurrency. There are two dialects of SCCharts, the core and the extended SCCharts, which are built upon the core dialect. The elements of both can be seen in Figure 2.4.



```
region "Core SCCharts":
```

```
immediate
  if x < 0 do x = 0
  go to B
```

```
initial state A ""
```

```
if b do y = `f(x)`
  go to M1
immediate if x < 0
  do x = 0 go to B
```

```
connector state G ""
  if y > -10 go to N1
  immediate go to N2
```

```
if 5 b go to G
```

```
final state I ""
  go to H
```

```
  ...
}
if y > 0 abort to N2
deferred if x <= 0 go to N2
if y == 0 join to F
```

```
state N1 {
  suspend if y > 10
  weak suspend if y > 0
  ...
```

```
scchart SCCharts_Overview {
  input bool b
  input output int x
  output float y = 0.0
  ...
```

```
state M2 {
  bool local
  region "Region1" {
    ...
  }
  region "Region2" {
    ...
  }
}
join to E
```

```
state E ""
  go to E
```

```
initial state S1
  if local go to S2
```

```
final state S2
```

```
entry do b = false
immediate during do b = true
during if x < 0 do s
exit do y = x
```

```
if pre(s) go to L
```

```
state N2 {
  ...
}
go to F
```

Figure 2.4. SCCharts elements [Sch24b].

The core dialect is a minimal set of simple features designed for easy compilation. It consists of states, variables, regions, and transitions, which can optionally have triggers and effects. Figure 2.5 shows the `ABO` SCChart, a well-established introductory example that shows the core features of SCCharts. The SCChart produces the outputs *O1* and *O2* according to the input *A* and *B*. In the first transition, *O1* and *O2* are set to false. After that, both *A* and *B* can set *O1* to true, which can happen with a different timing. Furthermore, *A* overrides the boolean *B* to true. After both boolean *A* and *B* were true, *O1* is set to false and *O2* to true in the last transition.

One of the core elements of SCCharts are *regions*, e.g. `HandleA` and `HandleB`, which are used to compose SCCharts hierarchically and to express concurrency, when they are in the same state. A state is called a *superstate* if it has one or more regions e.g. `AB`. The regions are executed as soon as the superstate is entered. Each region must have an initial state e.g. `init` and might have a final state e.g. `GotAB` in which it terminates. A superstate only terminates if all its regions reached their final state. SCCharts react in synchrony and in discrete ticks. A *tick* is a program reaction that consists of a finite amount of smaller computation, which are considered to take no time [Hal92]. SCCharts behave in a control-flow manner whereby they transition from one state to another. This transition goes as follows: each active state checks its

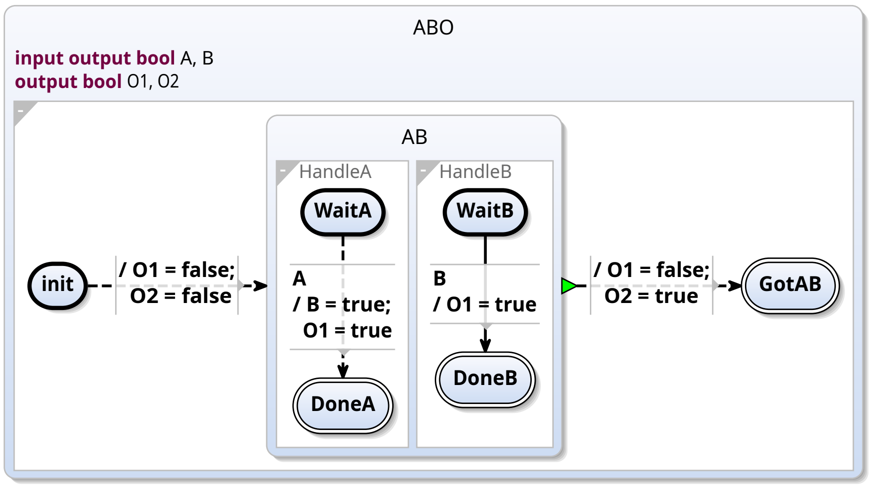


Figure 2.5. ABO SCChart [HDM+14].

available outgoing transitions—-these can be ordered by priority labels—-whether its trigger expression holds e.g. `is B true?`, takes the first match, executes its effect sequence e.g. `set O1 to true`, and passes activity on to the target state. This continues until no further transition can be taken, which marks the end of the tick. Transitions have a specific timing, concerning the discrete ticks. They can either be *immediate* or *delayed*. Immediate transition can be taken immediately during execution, while delayed transitions require that at least one tick has passed after their source state was entered, which means a delayed transitions gets ignored in the initial tick after entering the state. An immediate transition would be that in `HandleA` and a delayed transition that in `HandleB`. Furthermore, SCCharts can have multiple values in a tick for signals, channels and local variables due to their SC. This is because SCCharts act according to the Initialize-Update-Read Protocol (IURP). It allows a variable to be *relatively updated* several times after it has been initialized by an *absolute write*. This means in the `ABO` example that *B* is false in the absolute write, then it gets updates relatively to true in `HandleA` and after that, the transition in `HandleB` reads *B* so that the transition can be taken.

The extended set is a rich set of advanced features for easier modelling that are reducible to the core set. It allows writing SCCharts that can be referenced by states of other SCCharts [SMS+15]. This referenced state then has to provide a *binding* for all input and output variables of that SCChart module. In addition to classical statechart design, SCCharts also provide a dataflow notation [Smy21]. In special dataflow regions, SCCharts can be instantiated as actors and control logic is expressed as equation systems.

2.5 Linear Temporal Logic Formulas

Linear Temporal Logic (LTL) formulas express the abstract order in which events occur. For example, *event A occurs after event B*. The basic LTL formulas are built out of atomic propositions, the boolean connectors such as conjunction $\wedge$, and negation $\neg$, and two temporal modalities the next-operator X and the unil-operator U. They are defined as the LTL formulas over the set $\mathcal{S}$ of atomic propositions and $a \in \mathcal{S}$ are formed by the following grammar:

$$\varphi ::= \mathit{true} \;\mid\; a \;\mid\; \varphi_1 \wedge \varphi_2 \;\mid\; \neg\varphi \;\mid\; \mathtt{X}\varphi \;\mid\; \varphi_1 \,\mathtt{U}\, \varphi_2$$

Formal semantics of LTL are not covered in this thesis but can be looked up in the book by Baier and Katoen [BK08]. Figure 2.6 demonstrates the intuitive semantics of LTL that are used in this work. The operators G, which represents *always* and F representing *eventually* can be expressed using the until-operator as follows:

$$\mathtt{F}\varphi \equiv \mathit{true}\,\mathtt{U}\,\varphi \qquad\qquad \mathtt{G}\varphi \equiv \neg\mathtt{F}\neg\varphi$$

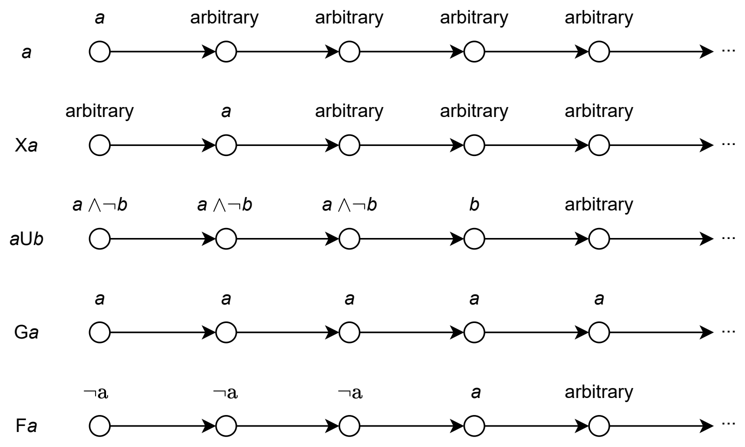


Figure 2.6. Illustration of the intuitive semantics of LTL [BK08].

2.6 Used Technologies

This section discusses the technologies involved in the application of STPA and the modelling of the steam boiler with SCCharts. Section 2.6.1 explains how KIELER is employed, which is used for modelling the steam boiler in SCCharts and visualizing this model. Following that, Section 2.6.2 introduces PASTA, which is used for the safety analysis of the steam boiler.

2.6.1 KIELER

The KIELER project[1] is a research project about enhancing the graphical model-based design of complex software systems, developed by the Real-Time and Embedded Systems group at Kiel

[1] https://github.com/kieler

University. In the past, the tool KIELER was built around the Eclipse Integrated Development Environment (IDE)[2], but today is also supported for Visual Studio Code[3] [Ren18; Dom18; KRD+24].

Part of the KIELER project are the semantics, which focus on the meaning of the symbols used in the modelling languages, particularly synchronous languages. This part of the project primarily deals with SCCharts, including specialized automatically generated graphical views, model-transformation-based compilation, and simulation. The KIELER tool hereby uses a text-first approach, for generating the diagrams. This means that the source model is edited textually, whereby the diagram is automatically generated.

Another, relatively new part of the KIELER project is the *Kieler Process Analysis*, which performs a safety analysis of a system. This part covers the PASTA tool, which combines STPA with pragmatics-aware modelling and visualization techniques. PASTA is discussed in more detail in Section 2.6.2.

In the context of this thesis, the tools from the KIELER project are primarily used for performing a STPA in PASTA, which visualizes the control structure and traceability diagram, and using SCCharts to model and visualize the steam boiler.

2.6.2 Pragmatic Automated System-Theoretic Process Analysis

The tool Pragmatic Automated System-Theoretic Process Analysis[4], developed by Petzold et al. [PKH23], is designed to support STPA. It is implemented as a Visual Studio Code (VSCode) extension[5] and provides a new language derived from the STPA process. This language evaluates the current progress of the STPA process using the built-in language server. PASTA differentiates itself from other STPA tools through its automatic visualization with a text-first approach, bridging the gap between purely textual and purely graphical tools [PH23]. PASTA offers two main advantages. First, its visualization capabilities, which can display both the control structure of a defined system and the traceability diagram in a separate figure. It also utilizes extensive filtering features, allowing multiple views of the underlying model to handle large graphs. For example, users can choose to view a single figure, toggle the visibility of individual STPA elements, or display only one UCA along with its connections to other components. Second, the entire analysis is conducted in a purely textual format, eliminating the need for users to familiarize themselves with a new User Interface (UI).

[2]KIELER in Eclipse

[3]KIELER in VSCode

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

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

Chapter 3

Related Work

This chapter reviews existing research and tools relevant to the topics covered in this thesis. First, Section 3.1 examines various tools designed to support STPA, highlighting their features. The discussion then shifts in Section 3.2 to existing risk analyses conducted on the steam boiler system, which serve as a foundation for further safety studies. Additionally, this chapter explores different modelling approaches applied to the steam boiler in Section 3.3. Finally, Section 3.4 looks into the verification of the steam boiler.

3.1 STPA Tools

Several tools already support the use of STPA, with the main challenge being to assist in its application by offering a systematic process, aiding in repetitive tasks, and producing a comprehensive list of safety requirements. This section presents three tools designed to support STPA.

3.1.1 STAMP Workbench

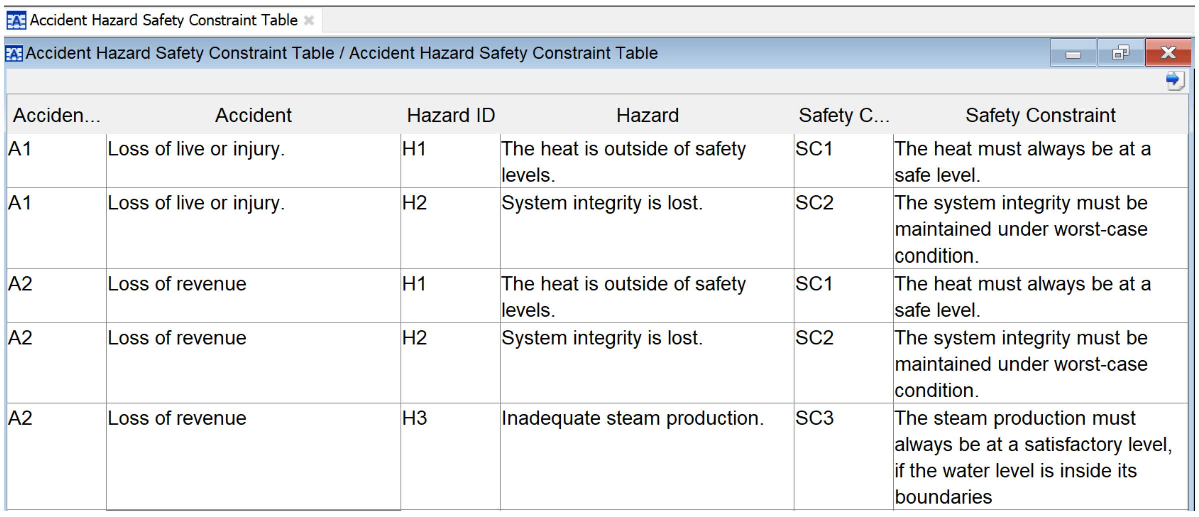
The STAMP Workbench[1] was created by the Information-technology Promotion Agency (IPA) in Japan as an open-source project. The tool offers various views for different steps in the STPA process and displays a description of the view when hovering over it. This description entails the *Purpose*, *Input*, *Processing*, *Output*, and `Remarks` of the specific view.

Initially, preconditions can be entered into a table by providing an Identifier (ID) and a description. Losses, hazards, and constraints are displayed in a unified view, represented by a table with columns for each aspect, as illustrated in Figure 3.1a. Linking is achieved implicitly by placing the related components in the same row. Each component is assigned an ID and a description.

Another available view is the *Component Extracting Table* in which components can be added to the control structure. For each component, attributes such as name, responsibility, control action, feedback, I/O, and remarks can be specified. Using this table, the control structure diagram can be automatically generated, if the components that should be shown are selected. However, it can also be manually drawn using drag-and-drop functionality. A control structure can be seen in Figure 3.1b. In the *Control Structure Diagram*, the automatically

[1] https://www.ipa.go.jp/en/digital/complex_systems/stamp_workbench.html

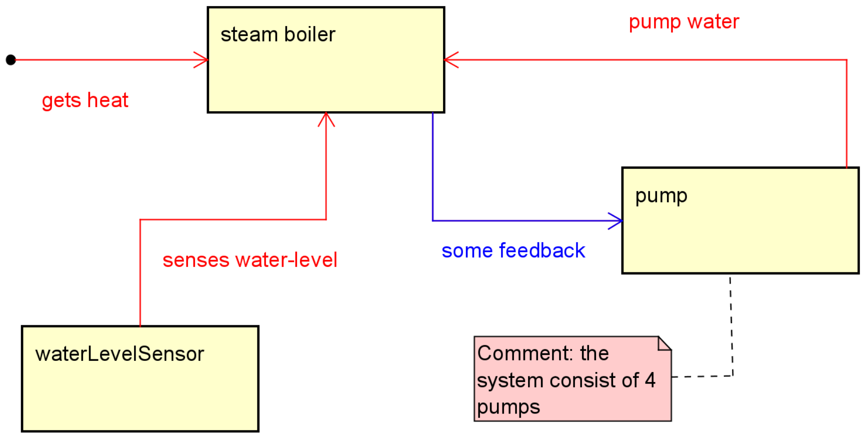
[2] https://www.ipa.go.jp/en/digital/complex_systems/stamp_workbench.html


Accident Hazard Safety Constraint Table

Accident Hazard Safety Constraint Table / Accident Hazard Safety Constraint Table

Acciden...	Accident	Hazard ID	Hazard	Safety C...	Safety Constraint
A1	Loss of live or injury.	H1	The heat is outside of safety levels.	SC1	The heat must always be at a safe level.
A1	Loss of live or injury.	H2	System integrity is lost.	SC2	The system integrity must be maintained under worst-case condition.
A2	Loss of revenue	H1	The heat is outside of safety levels.	SC1	The heat must always be at a safe level.
A2	Loss of revenue	H2	System integrity is lost.	SC2	The system integrity must be maintained under worst-case condition.
A2	Loss of revenue	H3	Inadequate steam production.	SC3	The steam production must always be at a satisfactory level, if the water level is inside its boundaries

(a) STAMP Workbench table example.



(b) Control structure in the STAMP Workbench.

Figure 3.1. Views in the STAMP Workbench[2].

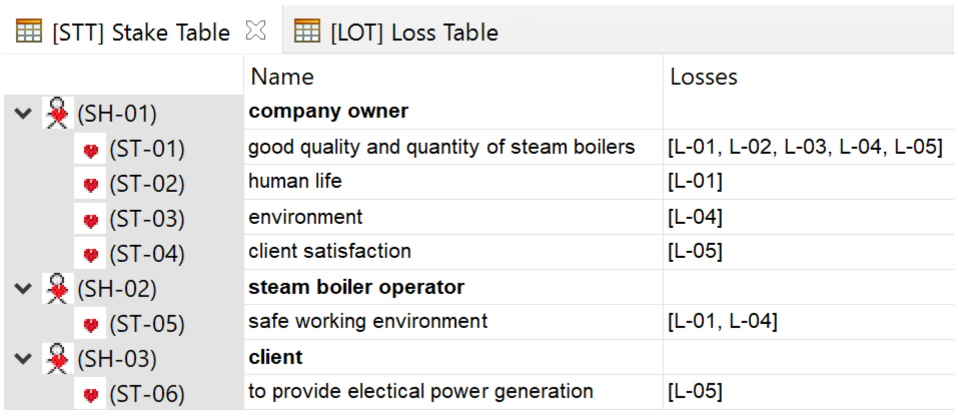
generated control structure is customizable and additional elements such as comments can be added.

For the UCA view, a table is used where control actions are automatically populated by extracting them from the control structure. However, this table does not function as a context table, as proposed by Thomas [Tho13]. To identify causal factors, a control loop diagram specific to each control action can be accessed, showing only the relevant components. Additionally, hint words for identifying causal factors are provided, and results are recorded in the `HCF Table`. In this view, the relevant UCA must first be selected, after which a table for documenting causal factors is displayed. Each entry includes an ID, the causal factor, the associated hint word, and scenarios. Finally, countermeasures can be recorded in a dedicated countermeasures table, which also displays the associated `HCF` and UCA.

3.1.2 STPA Capella

STPA Capella [CRD+22] is an experimental add-on tool for Capella[3], which is a tool for model-based systems engineering build on *Eclipse*. It is open-source and was developed by Oliver Constant[4]. It enables the users to perform STPA analyses in a model-based fashion, either standalone or in combination with Capella or Arcadia system architecture modelling. Albeit experimental, it has been successfully used in several real-world projects.

STPA Capella has a built in concept that allows the user to define stakeholder and their values, which are first entered in one table and then linked to the respective losses in another table. This can be seen in Figure 3.2.



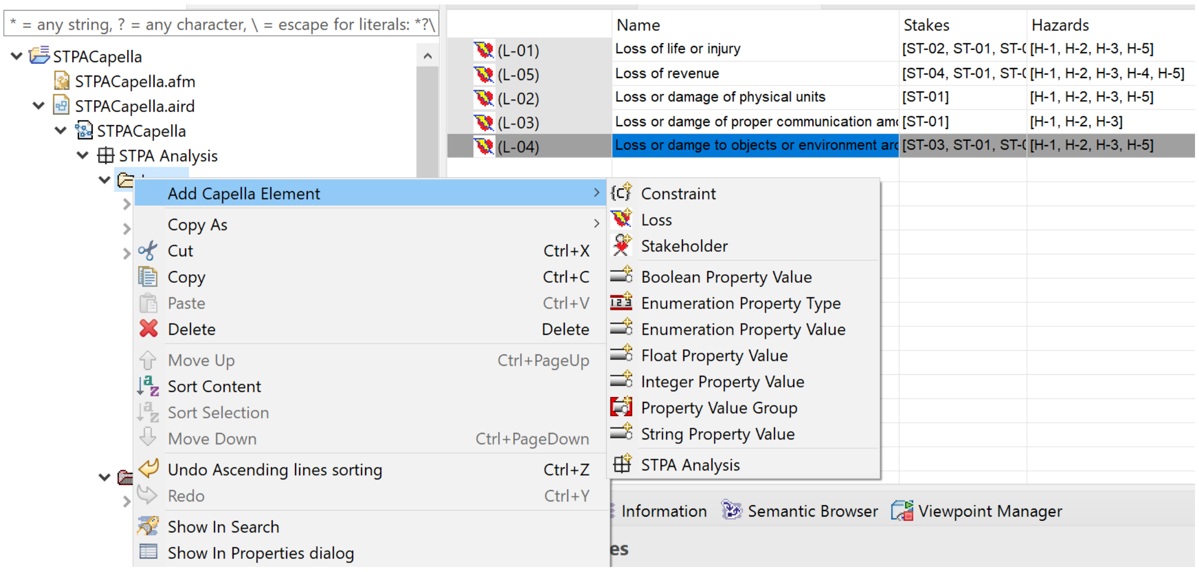
	Name	Losses
(SH-01)	**company owner**	
(ST-01)	good quality and quantity of steam boilers	[L-01, L-02, L-03, L-04, L-05]
(ST-02)	human life	[L-01]
(ST-03)	environment	[L-04]
(ST-04)	client satisfaction	[L-05]
(SH-02)	**steam boiler operator**	
(ST-05)	safe working environment	[L-01, L-04]
(SH-03)	**client**	
(ST-06)	to provide electical power generation	[L-05]

Figure 3.2. Stakeholder table in STPA Capella [Con24].

Afterwards the losses, hazards and system-level constraints can be defined in their respective table. The process of defining a new element is as follows: First, the correct directory needs to be clicked to add a new Cappella element. For example, a loss can be created by right-clicking on the *Losses* directory first, then *Loss* can be selected and described in individual input fields in the loss table. This process is shown in Figure 3.3a. After that, the traceability can be added to each element. This is done in a separate window, which is shown for a connection to a hazard in Figure 3.3b.

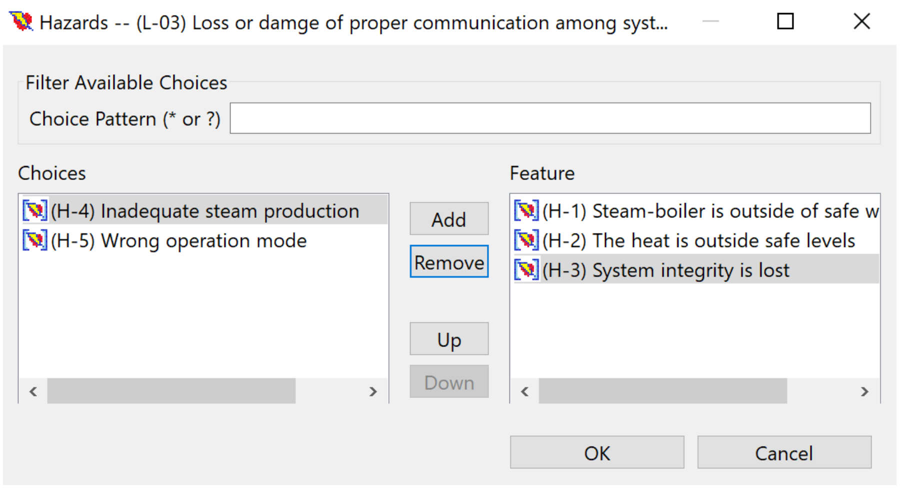
In the next step, the control structure is defined by adding the Capella Elements for components and actions. Furthermore, the responsibilities can be specified in an extra table for each component. In the same directory, also the UCAs are identified. In the last step, loss scenarios and constraints are specified in the same manner as the losses and hazards.

[3]https://mbse-capella.org/
[4]https://github.com/labs4capella/stpa-capella



	Name	Stakes	Hazards
(L-01)	Loss of life or injury	[ST-02, ST-01, ST-(	[H-1, H-2, H-3, H-5]
(L-05)	Loss of revenue	[ST-04, ST-01, ST-(	[H-1, H-2, H-3, H-4, H-5]
(L-02)	Loss or damage of physical units	[ST-01]	[H-1, H-2, H-3, H-5]
(L-03)	Loss or damge of proper communication am	[ST-01]	[H-1, H-2, H-3]
(L-04)	Loss or damge to objects or environment ar	[ST-03, ST-01, ST-(	[H-1, H-2, H-3, H-5]

(a) Creating a new loss.



Choices	Feature
(H-4) Inadequate steam production	(H-1) Steam-boiler is outside of safe w
(H-5) Wrong operation mode	(H-2) The heat is outside safe levels
	(H-3) System integrity is lost

(b) Selecting hazards that concern a loss.

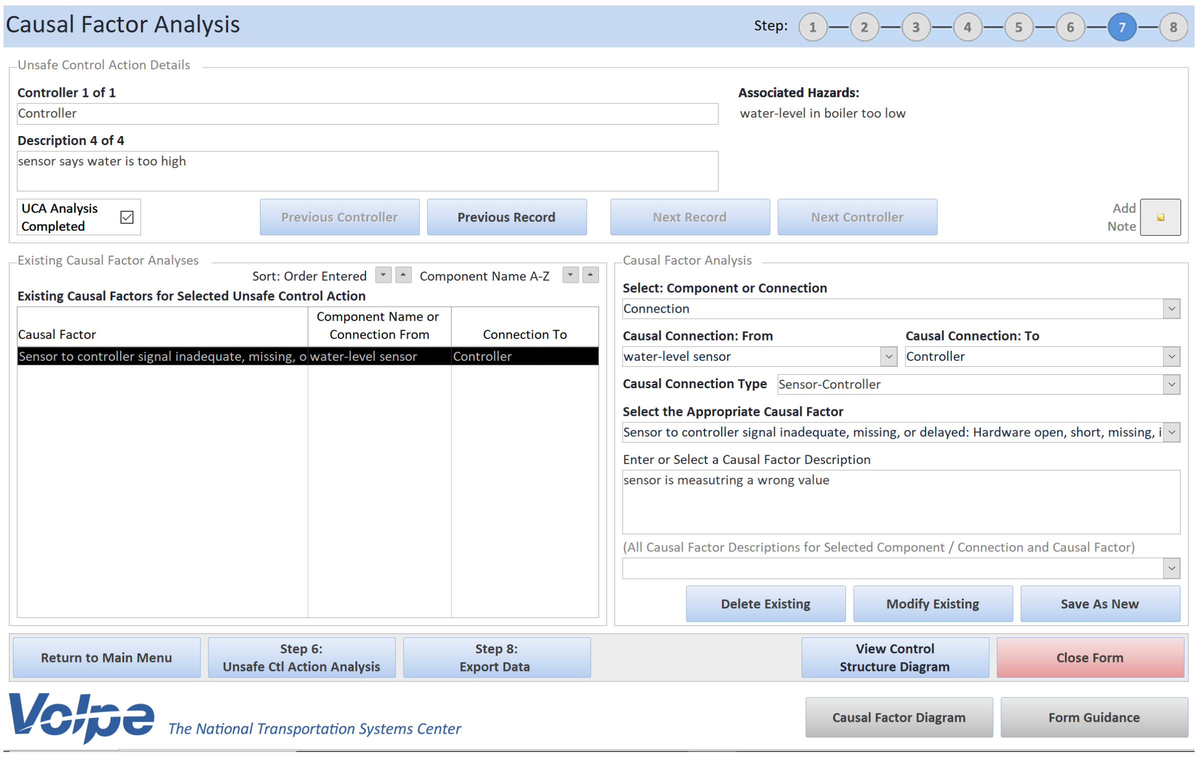
Figure 3.3. Loss creation and tracing in STPA Capella.

3.1.3 SafetyHAT

Safety Hazard Analysis Tool (SafetyHAT)[5] was developed at the Volpe National Transportation Systems Center in the USA and is designed to assist analysts in learning STPA [BV+14]. It accomplishes this by offering selection options, such as potential causes for loss scenarios. The STPA process is broken down into eight steps. The first three steps involve inputting system information such as the components of the system, their type and connection as well

[5]`https://www.volpe.dot.gov/infrastructure-systems-and-technology/advanced-vehicle-technology/safetyhat-transportation-system`

as their respective control actions. This is followed by four steps dedicated to the core STPA, the identification of losses, hazards, UCAs and loss scenarios. The view of the last STPA step can be seen in Figure 3.4. All steps have a form guidance that explains how to use the view and the *Causal Factor Analysis* also provides the user with a `Causal Factor Diagram`. Finally, the analysis results can be exported to Excel.



Causal Factor Analysis

Step: 1 — 2 — 3 — 4 — 5 — 6 — 7 — 8

Unsafe Control Action Details

Controller 1 of 1: Controller

Description 4 of 4: sensor says water is too high

Associated Hazards: water-level in boiler too low

UCA Analysis Completed ✓

Existing Causal Factor Analyses

Sort: Order Entered | Component Name A-Z

Existing Causal Factors for Selected Unsafe Control Action

Causal Factor	Component Name or Connection From	Connection To
Sensor to controller signal inadequate, missing, o	water-level sensor	Controller

Causal Factor Analysis

Select: Component or Connection: Connection

Causal Connection: From: water-level sensor

Causal Connection: To: Controller

Causal Connection Type: Sensor-Controller

Select the Appropriate Causal Factor: Sensor to controller signal inadequate, missing, or delayed: Hardware open, short, missing, i

Enter or Select a Causal Factor Description: sensor is measutring a wrong value

(All Causal Factor Descriptions for Selected Component / Connection and Causal Factor)

Figure 3.4. Causal factor analysis view in SafetyHAT[a].

[a] `https://www.volpe.dot.gov/infrastructure-systems-and-technology/advanced-vehicle-technology/safetyhat-transportation-system`

SafetyHAT was primarily developed for use in transportation system analysis, which is why it includes transportation-specific guide phrases and causal factors. Additionally, the categories for UCAs are expanded based on experience gained from applying STPA to transportation systems. However, users have the flexibility to modify both the categories and the guide phrases for causal factors.

3.2 Risk Analyses of the Steam Boiler

The steam boiler specification or similar steam boilers have already been analysed with different risk analysis techniques with the goal of identifying possible unsafe scenario and defining requirements the system needs to uphold in order to be safer. Three of the existing analysis are presented in this section.

3.2.1 FTA of the Steam Boiler

Prokhorova et al. [PTL13] are conducting a FTA on the steam boiler. They first define the functional and safety requirements the system needs to fulfil. An example for the functional requirements would be that `When the water level is between N1 and M1, the pump shall be switched on`. Moreover, an example for the safety requirements would be `During the system operation, the water level shall not exceed the predefined safety bounds`. The main hazard of the system is identified as the overflow or lack of water. For this reason, the fault tree shown in Figure 3.5 is based on the water level. It shows what the process is for the water level to exceed it bounds. For this, the water level can be either too low or too high. The case that the water level is too low is thereby fully developed. On the left side is shown how the defect of the water level sensor and another unit can cause the water level to be incorrectly determined if the steam boiler has no system to detect and tolerate these defects, which then leads to a too low water level.

3.2.2 HAZOP on a flame tube boiler

Oliveira et al. [OR18] performed a HAZOP on two steam boilers in real life that are located at the University Hospital of Santa Maria (HUSM). A steam boiler is defined as equipment used to generate and accumulate steam at pressures higher than atmospheric pressure, using any source of heat. The steam boilers at the HUSM are two flame tube boilers, which are characterized by internal circulation of the combustion gases in operation with liquid or gaseous fuels. The HAZOP application was limited to the boilers' water and pressure of the steam flow. Two Knots of the flame tube boilers system were considered for the analysis. The first one was located in the boilers' water supply and the second was situated in the steam power of the pressure vessel. The procedure to identify possible risk was as followed. First were the "Connections" identified, which are the probable critical points in the system. Afterwards, guide words able to cover the possible deviations of the evaluated system were determined and deviations and proposition of mitigation alternatives were assessed.

The risk analysis then revealed that the most dangerous factors are, on the one hand, always triggered by a water level that is too high. This risk could be minimised by elaborating and implementing a maintenance plan and a calibration plan. In addition, too low a water level due to incrustation in the water pipe is recognised as a severe risk. According to the analysis, this could be minimised by implementing an alarm system when the water level is too low. On the other hand, excessively high steam pressure is always very risky, and the scenario where the pressure register closes during the boilers' operation is seen as particularly dangerous. It is recommended to elaborate and implement a maintenance plan, the substitution of the manual register by an automatic controller and investing in qualification for boilers operator. Finally, it is considered very dangerous if the pressure is too

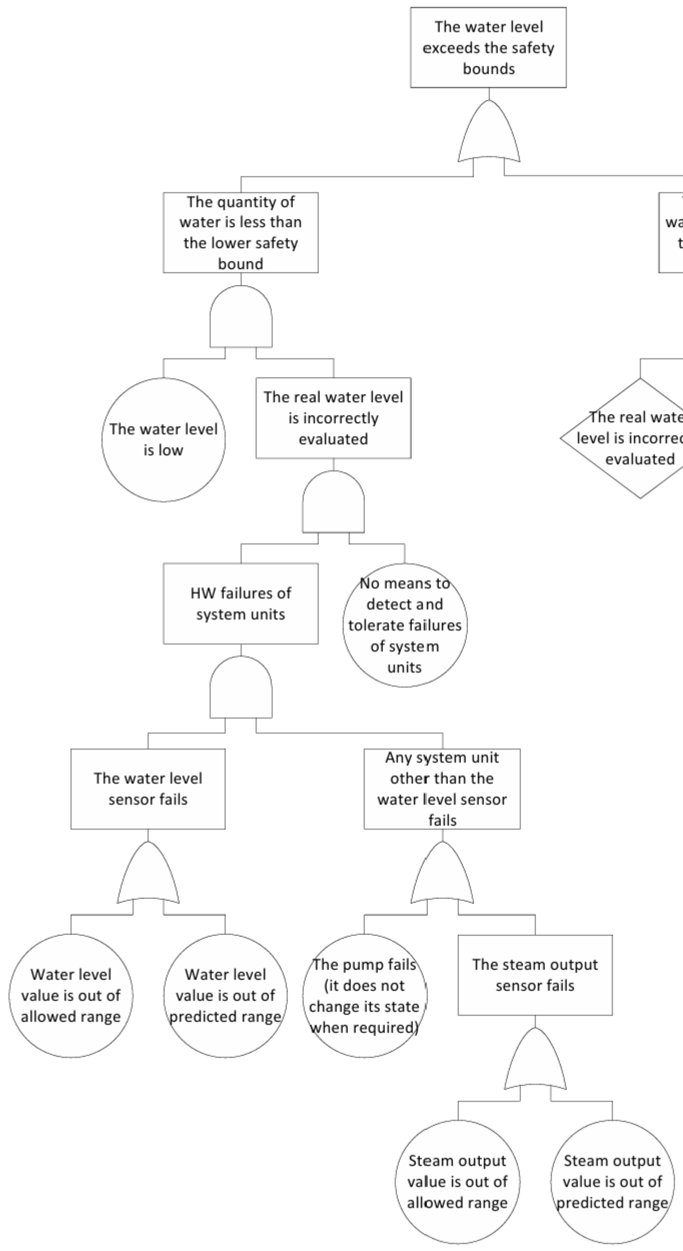

The water level exceeds the safety bounds
The quantity of water is less than the lower safety bound
The quantity of water is higher than the upper safety bound
The water level is low
The real water level is incorrectly evaluated
The real water level is incorrectly evaluated
The water level is high
HW failures of system units
No means to detect and tolerate failures of system units
The water level sensor fails
Any system unit other than the water level sensor fails
Water level value is out of allowed range
Water level value is out of predicted range
The pump fails (it does not change its state when required)
The steam output sensor fails
Steam output value is out of allowed range
Steam output value is out of predicted range

Figure 3.5. Fault tree of the water level [PTL13].

low due to obstruction or leakage in the oil pipe. Here it is also recommended to create a plan and to invest in the boilers' course of operation for the operators.

3.2.3 HAZOP on a coal-fired boiler

Rahma et al. [RH] conducted a HAZOP analysis on a steam boiler used for producing electricity. It is a PT PJB's coal-fired boiler system, which consists of a feed water system (steam drum), steam system (superheater), and fuel system (furnace).

Possible losses that were identified, are the loss of production or property as well as injury or death of bystanders. The analysis determined two study nodes with extremely high-risk levels: the feed water setting to the steam drum has a damaged feed water pump, and the steam pressure on the steam drum has a damaged safety valve. As a result, the water level is low and the boiler trips. The controls implemented by PT PJB to minimise these risks are for the first case that the pump is getting repaired or replaced and that a feed-water flow sensor and a steam drum level sensor are getting installed. This is done so that this particular error is detected early. For the second case, possible action to minimise the risks are the inspection or repair of the safety valve, doing a leak check as well as checking the sensor level and the pressure.

3.3 Models

The steam boiler specification has already been modelled with different methods. Three of the existing models are presented in this section.

3.3.1 The Steam Boiler in Statecharts and Z

Büssow et al. [BW96] modelled the steam boiler specification in their work with statecharts and Z. They first split the system in three compatible views: the architectural model, the reactive model and the functional model.

The architectural model of a system describes the relationships between the classes of components used in the system as well as the actual configuration of the system components themselves.

The two other views are primarily concerned with the specification of the behaviour of single components of the embedded control system. The functional model, defines a component's data structure, including data invariants and transformation relations, capturing its local state and input/output operations. From this model, constraints such as safety properties can be derived. The second view, the reactive model, focuses on a component's interactions with other components, detailing its life-cycle and managing timing constraints. It specifies how external operations are requested or supplied during state changes.

Figure 3.6 shows the architectural model of the steam boiler, which primary components are the unit manager and the steam boiler control system. Two components were added that differ from the specification. One being the `MonitoredPump`, which monitors the pumps and their controllers. The other being the `UnitManager`, which was introduced to distinguish between the non-periodic processing of signals and their periodic transmission.

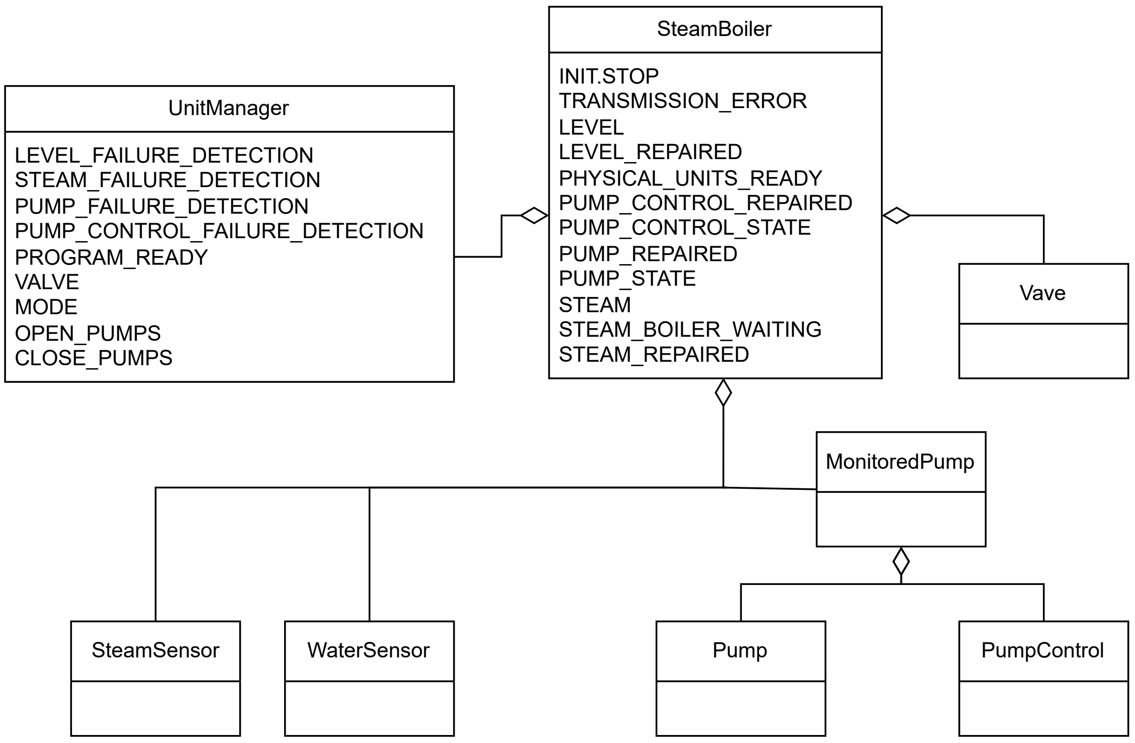


Figure 3.6. Main components and their services in the architectural model [BW96].

The steam boiler system is structured around a unit manager that controls several physical units. The unit manager plays a key role by handling two tasks: sampling sensor data and status information from actuators (e.g., pumps) and periodically transmitting this data to the main control system. Additionally, it processes control messages sent between the steam boiler control system and the physical units, managing communication errors by sending transmission error messages if problems occur.

The unit manager requests data from sensors and actuators periodically and transmits it to the central control system, while also receiving and forwarding operating mode data to the physical units. It handles failure detections, repair acknowledgments, and any communication errors between the physical units and the main control system.

The reactive model splits the model in an `NoEmergency` and in an `Emegerency` state. The system can only run in the first one. Furthermore, it is specified that the pumps need an extra state for balancing the pressure.

The functional model then defines the values each unit can have and at what value the state of a unit changes. Moreover, it does define when the system is considered to be in a specific mode and how the transmissions should work.

3.3.2 The Steam Boiler in SCCharts

The steam boiler was already modelled in SCCharts twice. In the work from Smyth et al. [SDH19] the steam boiler was heavily abstracted. It can have no transmission failures and the signals are all modelled as booleans. This is the case, because the focus was on a user study on manual user verification of different source codes that were generated by automatic code generators. The SCChart was compiled to C with three different approaches and it was shown that manual verification can be time-consuming and is error prone if the user has no clear mapping between states and transition of the original model and the generated code. Therefore, the participants performed better if the generated code followed a state pattern that preserves original model structures and names.

In the work from Schulz-Rosengarten [Sch24a] the focus lies on using Object-Orientated (OO) design principles to model the steam boiler so that the OO features of SCCharts can be evaluated. For this reason, the model is heavily built upon modularisation with the help of inheritance, extending interfaces and using abstract classes for transmission failures. The results of the evaluations illustrates that the new OO SCCharts are capable of expressing an OO architecture and its implementation in one model, while various derived views can visualize different structural aspects.

The similarities and differences of the two SCChart models and the one designed in this work are discussed in more detail in Section 6.2.4.

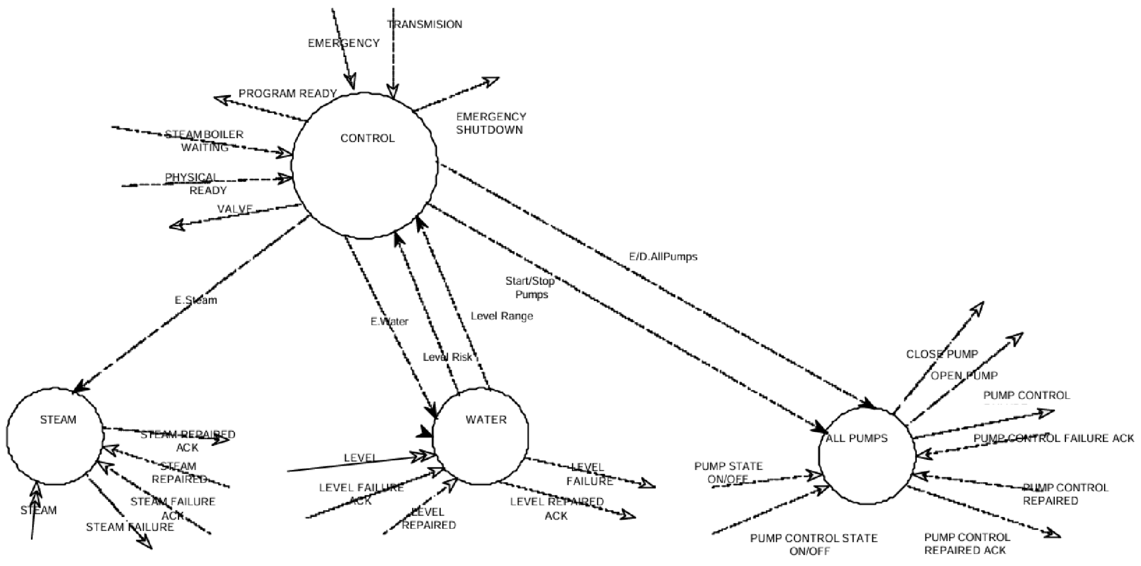


Figure 3.7. The steam boiler divived in four subsystems [dT06].

3.4 Verification

The work by Riva et al. [dT06] verifies the steam boiler. For this, a model was created that consists of four subsystems and can be seen in Figure 3.7. This was done in order to avoid the state explosion problem. A natural approach to solving this problem is to divide the system into components and to perform local verifications on separate components and then to deduce a number of global properties for the whole system.

The verification of the system's correctness involves ensuring that 55 declarative properties derived from system requirements are satisfied. Due to the complexity and size of the system, a modular verification approach was adopted to manage state explosion and resource constraints. For this, properties were translated into LTL formulas for formal verification. Global properties were broken down into local properties relevant to each component and each component was verified under certain assumptions about its environment. To enhance efficiency, assumptions were automatically generated based on the behavior of input events.

Chapter 4

The Steam Boiler in PASTA

The focus of this thesis is for one part to perform a STPA in PASTA on a bigger system, to later evaluate the usefulness of the tool in aiding in the analysis. For this reason, a risk analysis of the steam boiler was carried out, which made it possible to identify possible risks and scenarios that could lead to them. This chapter presents the process of the STPA that was performed with the help of PASTA and addresses the challenges that occurred and how these were solved.

4.1 Analysis

This section explains the process of performing a STPA in PASTA. It explores the individual steps of the STPA and how it was carried out for the steam boiler in PASTA. Additionally, it explains in detail certain special cases where either the specification was deviated from and the reasons behind these decisions or where a specific approach was chosen due to the limitations of the tool.

4.1.1 Define Purpose of the Analysis

The first step of STPA is to identify the losses of the system. For this, Leveson [Lev18] advises to first identify the stakeholders such as users, operators or producers and determine their stake in this system. Each of these values can then be translated to a loss. This approach has the advantage that the analyst first familiarizes themselves with all aspects of the system and its users, making it less likely that losses will be overlooked. In addition, a basic understanding of the system is already established. A feature to do this does not yet exist in PASTA, so that one can only record the stakeholders separately for oneself or immediately identify the losses.

```
Losses
L1 "Loss of life or injury"
L2 "Loss or damage to physical units"
L3 "Loss or damage to proper communication among system components"
L4 "Loss or damage to objects or environment around the system"
L5 "Loss of revenue"
```

Listing 4.1. Losses of the steam boiler.

Possible stakeholder are the companies who produce and use the steam boiler respectively. They value human life, the environment, a good product or production and the satisfaction of the client. Other possible stakeholders are the steam boiler operator and the residents that live near the companies. They value a safe working and living environment. These stakes can be translated to the losses seen in Listing 4.1.

The hazards, that can lead to losses, are then identified and shown in Listing 4.2.

```
Hazards
H1 "Steam-boiler is outside of safe water levels" [L1, L2, L3, L4, L5] {
   H1.1 "Water level is too low"
   H1.2 "Water level is too high"
}
H2 "The heat is outside safe levels" [L1, L2, L3, L4, L5]
H3 "System integrity is lost" [L1, L2, L3, L4, L5]
H4 "Inadequate steam production" [L5]
H5 "Wrong operation mode" [L1, L2, L4, L5] // also STOP
```

Listing 4.2. Hazards of the steam boiler.

For each hazard a system-level constraint is defined. In this case, a constraint only states that the hazard cannot occur even in a worst-case environment. For example, the system-level constraint for *H2* "`The heat is outside safe levels`" would be `SC2` "`The heat must always be at a safe level`". All system-level constraints can be found on `github`. The connection of the losses, hazards and constraints values can be seen in Figure 4.1.

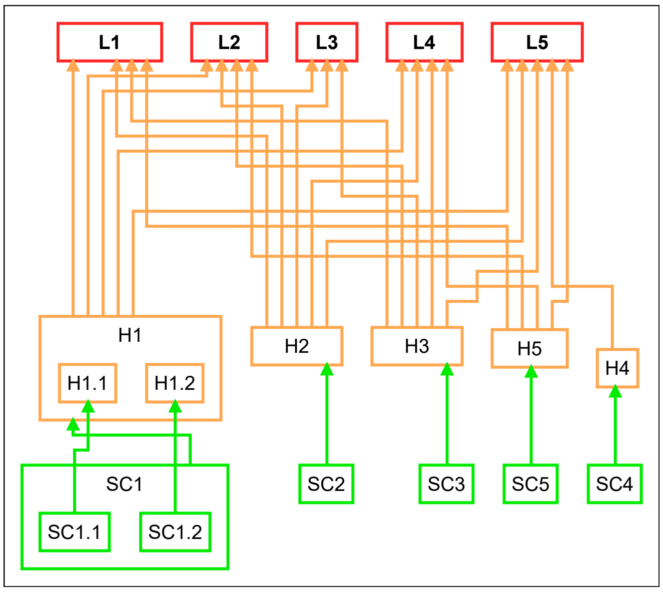


Figure 4.1. Losses, hazards and system-constraints of the steam boiler.

The losses and hazards were kept to a minimum, as is good practice. An exception was only made for the distinction between the loss of physical units and the loss of communication between the components. This is the case, because according to the specification of the steam boiler, the system can still continue to operate if certain physical components fail, whereas it immediately goes into an emergency stop if there is a problem with the communication. Furthermore, this thesis introduced a hazard that specifies that the system can also have a problem with the heat if it is outside a certain range. This analysis therefore also considers the heater of the steam boiler, which is not considered in the specification, as it cannot be regulated by the steam boiler controller. *H1* is also subdivided into *subhazards*, as this is the main hazard in the specification and the feature of the subhazard can also be presented in this way. Such a subdivision would also have been possible for *H2*, but this was not done as the main focus should be on the physical units of the specification and one example is sufficient for evaluating the feature.

4.1.2 Model the Control Structure

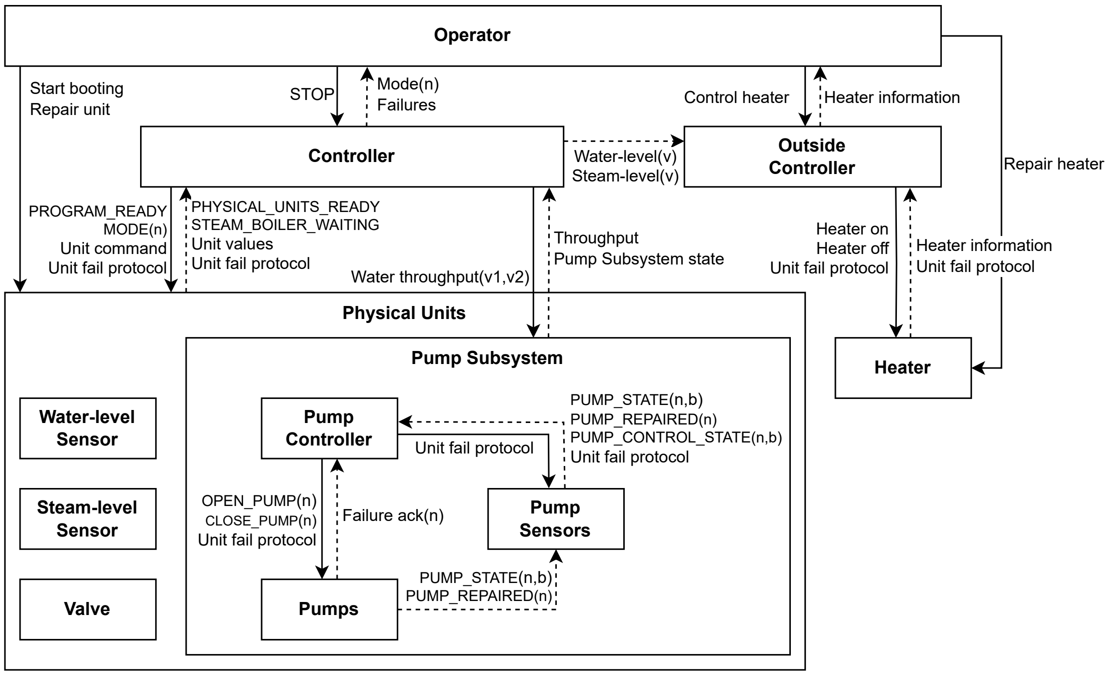


Figure 4.2. The steam boiler control strcuture.

In the next step, the *control structure* for the steam boiler was defined. An abstraction of it can be seen in Figure 4.2, as the original control structure, which PASTA created, was too large for this explanation. The original control structure can be found on `github`. In the abstracted version of the control structure, as well as in the original, there are control actions and feedback actions, which are capitalized. This means that these signals are defined in the specification. All other signals have been added for a more complete analysis of the steam boiler.

The system is monitored by an *Operator*. This Operator is responsible for starting and stopping the system and repairing the physical units. It also receives information about the mode of the system, component failures and transmission failures. n denotes here the five different modi, that the steam boiler system can be in.

The system is managed by two controllers. These are separated, as one of the controllers, called the *Outside Controller*, operates the *Heater*, which is not part of the actual steam boiler specification of Abrial et al. [Abr05] and is for that reason kept separate from the rest. The Outside Controller is responsible for activating and deactivating the Heater. For this purpose, it also receives information about the water level and steam level from the *Controller*, where v denotes the values for the respective level. Moreover, it gets information about the Heater - its state and heat value - from the Heater itself, as the physical heater and its sensor are modelled as one module. This is because this work focuses on the steam boiler specification and therefore the control structure should not be enlarged too much by other units. For this reason, other components such as tubes, the deaerator, or combustion chamber, are also not listed.

The control action from the Outside Controller and the feedback from the Heater also has a so called *unit fail protocol*. This is defined as follows: the control unit detects a failure of a physical component and as a consequence sends a FAILURE_DETECTION control action to the concerning unit. The unit then has to answer with a FAILURE_ACKNOWLEDGEMENT feedback when it received this action. After the Operator has repaired the physical unit, it sends a REPAIRED signal as feedback and waits in turn for the REPAIRED_ACKNOWL-EDGEMENT from the control unit. This protocol is used by every physical component and their respective controller if a failure was detected.

The other controller, simply called the *Controller*, is responsible for managing the units named in the specification. In this case, it performs the tasks that the program has assumed in the specification. It boots the program, transfers the individual modes that the system is in at a certain point in time to the components and controls the physical units. For being able to do this, it receives the values that the system outputs, such as the position of the valve or the water and steam level. It also receives the STEAM_BOILER_WAITING signal, with which the program starts booting, and PHYSICAL_UNITS_READY when all physical units have finished booting. In addition, it executes the unit fail protocol with the *Valve*, the *Steam-Level Sensor* and the *Water-level Sensor*.

The specification does not explicitly state what happens if the Valve breaks, as it is assumed that this will not happen. Since this assumption is unlikely to be met in a real system, it is also assumed in this analysis that the Valve can break. It follows the same unit fail protocol as the other physical units in the event of a failure and subsequent repair.

Contrary to the specification, the Controller does not control the pumps and their sensors directly, but indirectly through the *Pump Subsystem*. This subdivision was carried out so that the individual controllers have fewer *process variables*, as this allows the *context tables* in PASTA to be better utilized. The process variables of the control structure will be discussed in more detail later in this subsection and the context tables are considered in more detail as part of step three of STPA in Section 4.1.3. The Controller regulates the Pump Subsystem by forwarding the minimum and maximum throughput that all the pumps should have, so that the system can control the pumps accordingly. The required throughput is determined by the values of the remaining sensors. Then the Pump Subsystem gives feedback actions of the current throughput and the general state of the system back to the Controller. The state here is whether all units are working, parts have failed or everything has failed. This information needs the Controller in order to control the system.

The Pump Subsystem consists of three components: the Pump Controller, Pump Sensors and Pumps. There are four of each of the pumps and sensors, as stated in the specification. It should also be noted here that the pump controller referred to by Abrial et al. [Abr05] is in this case the Pump Sensor and the Pump Controller is a controller in the sense of a controller in STPA, which regulates units. The subsystem works as follows: the Pump Controller sends the *control action* to open or close a pump. `n` stands for the respective pump that is to execute the action. The pumps then give their state as feedback for the sensor. The `n` again stands for the respective pump, and `b` is the respective state: open or closed. The sensor then passes this feedback on to the Pump Controller with its own state. In this case, the state is again denoted by `b`, which now stands for water flowing or water not flowing in the respective pump. The Pump Controller also handles the *unit fail protocol* for the pumps and their sensors. It only forwards the error status to the Controller so that a repair can be carried out. The protocol is executed directly with the Pump Sensors, and these also transmit the REPAIR signals of the Pumps because the sensors recognizes the repairs. However, the FAILURE_ACKNOWLEDGEMENT is passed directly from the Pumps to the Pump Controller as feedback.

The Operator and all controllers also have a process model with process values, which they need to decide which control action is to take for a specific context. The Operator has process variables that tell them for all physical units whether these are defective so that they can repair them. Moreover, they know if it came to a transmission failure and the mode of the system, so that they can stop or start it in an emergency. Figure 4.3 shows what this looks like in code in PASTA and what it looks like as a diagram.

The Outside Controller only needs to know about the defects of the Heater and, if transmission failures occur, to either pass this on to the Operator or apply the unit fail protocol. The protocol needs three variables: one variable to know if the Heater acknowledged the fail and two variables in which the defect of the Heater is split into. One of the defective variables indicates whether the Heater was previously faulty, while the other indicates whether the Heater is now faulty. This is done to determine whether the Heater has been repaired. It is considered repaired if it was previously defective and is now functioning correctly, indicating that the issue has been resolved and the defect no longer persists. The Outside Controller also knows the state of the Heater and its heat in order to be able to switch it on and off appropriately. Certain ranges are specified for the heat in which it can be located. The ranges were defined in this way because different UCAs can be found depending on the range. The specific variable ranges, which are specified in the code, can be used later for automated

```
  processModel {
   pumpSubsystemDefective: [yes=[true], no=[false]]
   valveDefective:         [yes=[true], no=[false]]
   levelSensorDefective:  [yes=[true], no=[false]]
   steamSensorDefective:  [yes=[true], no=[false]]
   transmissionDefective: [yes=[true], no=[false]]
   mode:                 [initialization=[0], normal=[1],
                           degraded=[2], rescue=[3], emergencyStop=[4]]
   heaterDefective: [yes=[true], no=[false]]
}
```

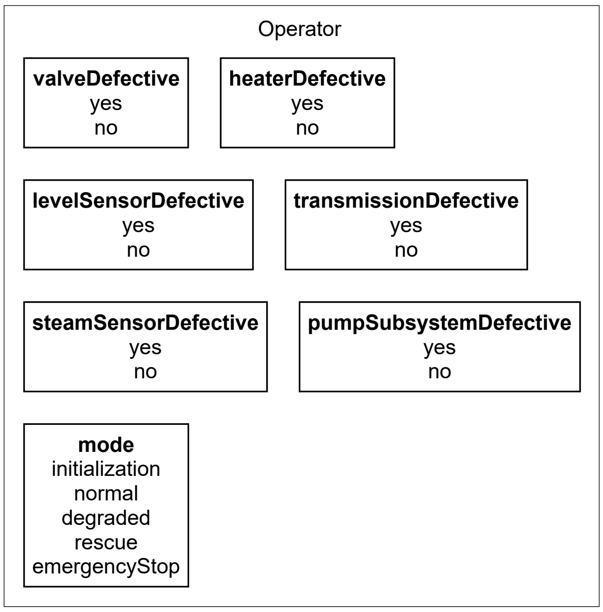


Figure 4.3. Process model of the Operator in code form and as diagram.

creation of models and also serve to improve understanding of the process variables. For this reason, they were defined for all variables. The heat variable then looks as follows:

```
heat: [belowLimit=[MIN, H1], belowNormal=[MIN,L1], normal=[L1, L2],
      aboveNormal=[L2,MAX], aboveLimit=[H2, MAX]]
```

The variables indicatie hereby the physical limit of the Heater. The heat must lie in-between H_1 and H_2, because the steam boiler and heater could be seriously damaged otherwise. It ideally lies between the security levels L_1 and L_2 for optimal steam production.

The Controller has process variables that specify the failures of transmissions and all units. But this time no *DefectiveBefore* variable exists, because it is only needed to know if a unit was repaired. This is the case, because the UCAs for the unit fail protocol were only defined exemplary for the Heater and omitted for the rest of the units, because it is the same for all of them. Moreover, it does allow the controller to have fewer *process variables*, which allows the *context tables* to be better utilized. The only other difference is the Pump Subsystem, because this does not have a binary state: defective and not defective. Instead, only parts can be defective, as it is an entire system of units. The Controller must therefore react differently if only one part is defective, as the system can then still run, whereas it must stop if all pumps are defective. It also has variables that indicate, which mode the system is in and whether it is ready to start at all. The second is the case when the Controller has received the PHYSICAL_UNITS_READY signal from all units. Finally, it has variables that describe the values and states of the units. The values are again divided into ranges, as with the Heater. In addition to the water level, there is also a calculated water level. This is required if the water level sensor is defective, and the system goes into rescue mode, as the water level can no longer be measured reliably.

Lastly, the Pump Controller has process variables that specify the failures of transmissions and all different pumps and their respective sensors. It also has variables that describe the value and states of the components. The value is again divided into ranges, as before, and all throughputs are combined to one. There is no variable that indicates whether the water in the pumps is flowing, which would be the sensor value for the pump sensors. This is the case, because the variable is only used to know if there is a defect with the pump or their respective sensor. But since there are already variables indicating that defect, no additional variable is needed. The mode is required because the subsystem not only has to decide when to open or close the pumps, but also when to start or stop the whole system.

All components were then analysed to identify the respective responsibilities they need to adhere, so that the system-level constraints can be fulfilled. They all are responsible to uphold the unit fail protocol. They do this by returning an acknowledgement for a fail signal, if they are a physical unit, and an acknowledgement for a repair signal, if they are a controller. In addition, the controllers must first recognize the error and inform the units while the units are informing the controllers of a repair. Otherwise, the tasks of the components are generally to listen to the higher hierarchical unit when it gives a control action and to update it with feedback. The controllers also have the responsibility to make decisions on states and the sensors have the responsibility to only pass on correct data. The Controller's responsibilities are shown in the following as an example:

```
Controller {
  R5 "Must adhere to start and stop command from the Operator" [SC1, SC3, SC4, SC5]
  R6 "Decides what state the valve should have" [SC1, SC3, SC4, SC5]
  R7 "Decides what throughput the pumpSubsystem shall have" [SC1, SC3, SC4]
  R8 "Decides what mode the steam-boiler system should be in
      (also decides the state of the PumpSubsystem)" [SC1, SC3, SC4, SC5]
  R9 "Detects failures and acts accordingly
      (send failure to physical units / change mode)" [SC1, SC3, SC4, SC5]
  R10 "Needs to inform Operator of important system data" [SC2, SC3, SC4, SC5]
 }
```

4.1.3 Identify Unsafe Control Actions

In the third step, the control actions of the control structure were analysed to find the UCAs. PASTA offers two options for defining UCAs. Firstly, the traditional method, where all UCA types are analysed for a control action and then the UCA with the associated context and hazards are written down textually. Secondly, context tables can be used. The analyst specifies rules for contexts that are seen as unsafe. These contain the concrete values of the process variable and the hazards to which these can lead, when they are provided or not provided. The UCAs described in this way are then displayed in the context table. These types are specified beforehand. For example, part of the UCAs for the control action *valve* from the Controller, with the type `not provided`, `provided` and `too late`, are shown with the traditional method in Listing 4.3 and with the use of context tables in Listing 4.4. In this work, the second method was chosen due to the advantages described in Section 2.3.

```
UCAs
Controller.valve {
   notProviding {
      UCA70 "Valve command to open the valve is not provided
           when mode is initialization, the waterLevel is higher then N2
           and the valve is closed." [H1.2, H4]
      UCA71 "Valve command to close the valve is not provided
           when mode is initialization, the waterLevel is inside its boundaries
           and the valve is open." [H1.1, H3, H4]
      UCA72 "Valve command to close the valve is not provided
          when mode is initialization, the waterLevel is lower then N1
          and the valve is open" [H1.1, H3, H4]
   }
   providing {
      UCA73 "Valve control action is provided, when mode is normal and therefore not
           initialization as needed for the command" [H1, H3, H4, H5]
   }
   tooEarly/Late {
      UCA79 "Valve command to close the valve is provided too late,
           when mode is initialization, the waterLevel is above N2
           and the valve is open." [H1.2, H4]
      UCA80 "Valve command to open the valve is provided too late,
           when mode is initialization, the waterLevel is above N2
           and the valve is closed." [H1.1, H3, H4]
   }
  }
```

Listing 4.3. UCAs in traditional notion in PASTA.

```
  Context-Table
  RL16 {
    controlAction: Controller.valve
    type: not-provided
    contexts: {
        UCA70 [mode=initialization, waterLevel=aboveBoundary, valve=close]
              [H1.2, H4]
        UCA71 [mode=initialization, waterLevel=insideBoundary, valve=open]
              [H1.1, H3, H4]
        UCA72 [mode=initialization, waterLevel=lowerBoundary, valve=open]
              [H1.1, H3, H4]
    }
  }
  RL17 {
    controlAction: Controller.valve
    type: provided
    contexts: {
        UCA73 [mode=normal] [H1, H3, H4, H5]
    }
  }
  RL18 {
  controlAction: Controller.valve
  type: too-late
  contexts: {
      UCA79 [mode=initialization, waterLevel=upperBoundary, valve=close]
            [H1.2, H4]
      UCA80 [mode=initialization, waterLevel=insideBoundary, valve=open]
            [H1.1, H3, H4]
  }
}
```

Listing 4.4. UCAs written with the help of context tables in PASTA.

The `mode` has five possible values, which are the modi described in the specification. Moreover, the ranges of the process variable *waterLevel* from Listing 4.4 are divided in five different ranges that are shown in Listing 4.5.

```
 waterLevel: [belowBoundary=[0,M1], lowerBoundary=[0,N1],
           insideBoundary=[N1, N2], upperBoundary=[N2, MAX], aboveBoundary=[M2, MAX]]
```

Listing 4.5. The ranges of the waterLevel process variable.

As mentioned before, the variables M_1 and M_2 are those between which the water level must lie, because the steam boiler could be seriously damaged otherwise. Between N_1 and N_2 is the ideal water level range for optimal steam production. The `valve` process variable is binary.

The UCAs shown in Listing 4.4 and all others have been identified as follows. First, I selected a controller for which all control actions were to be defined. Then, for each control action, all possible UCA types were analysed and contexts were found that lead to hazards. The control actions for the unit fail protocol were first skipped and then only listed once as an example, which will be explained in more detail later. As a final step, I looked at the control table for the respective control actions to check whether a context does not yet lead to a UCA that is actually unsafe. The context tables were not used more actively in the writing of the UCAs because of the challenges described in Section 4.2.2.

Furthermore, the use of the method with context tables itself can cause issues with certain UCAs when attempting to describe their contexts precisely. This is the case because a context can no longer be detailed textually, and must instead rely on the meaningfulness of the process variables. Such a case occurs with the control action *setThroughput* from the Controller. In this situation, a UCA occurs if the throughput range is not set to a different value when the water level becomes too high or too low. If the status of the pumps is not changed in this context, it can lead to a loss of integrity of the steam boiler and inadequate steam production. However, the water level can be determined in two different ways, depending on which mode the program is currently in. In the optimal scenario, the water level is determined by the water level sensor, but if the system is in rescue mode, the measured *waterLevel* of the sensors can no longer be used, because of a sensor failure. This means that the water level needs to be calculated with the values of the other sensors, so that the system can use the *calculatedWaterLevel*, which has the same ranges as the *waterLevel*. This differentiation results in different contexts for UCAs. It is only unsafe not to provide a control action if the specific water level matches the mode of the system. However, with PASTA there is no possibility to introduce a `not` in a rule to define that the measured water level is always taken, except in rescue mode. This is discussed in more detail in Section 6.1.4. The only way to describe this would be to list all other modes individually, that use *waterLevel*. However, this option was discarded as the additional UCAs would have made the analysis more complex and time-consuming. Instead, there is a comment at this point that indicates the circumstance of the correct mode that has to be taken, as shown in Listing 4.6.

```
RL20 {
 controlAction: Controller.setThroughputRange
 type: not-provided
 contexts: {
    UCA83 [waterLevel=lowerBoundary] [H1.1, H3, H4] // mode not rescue
    UCA84 [waterLevel=upperBoundary] [H1.2, H4] // mode not rescue
    UCA85 [mode=rescue, calculatedWaterLevel=lowerBoundary] [H1.1, H3, H4]
    UCA86 [mode=rescue, calculatedWaterLevel=upperBoundary] [H1.2, H4]
 }
}
```

Listing 4.6. UCAs for control action *setThroughputRange* type `not provided`.

Another problem occurs with the pumps and their sensors. According to the specification, there is only ever one signal that controls the respective pumps and sensors, which has the number of the pump or sensor as an attribute. This has also been modelled in the control structure and is represented by an `n` as described in Section 4.1.2. However, in this way it is no longer possible to distinguish in the contexts of the UCAs exactly, which unit the signal went to, as it is recognized by PASTA as one signal. This could be avoided by modulating a single signal for each pump and sensor. Nevertheless, this solution was rejected as it would have made the control structure much more unclear. This decision has no major influence on the UCAs, which consider the control actions *openPumps* and *closePumps*, as the throughput is already considered as a whole and therefore the number of units is abstracted for the respective control action. However, a restriction exists for the unit fail protocol. It can still be decided whether the control action *pumpFailDet* is not provided for the correct unit. This is the case because if there is no transmission failure and the program got no acknowledgement from the unit, the control action could not have been provided for the correct unit. In all other cases, it must be assumed that the control action went to the wrong unit, even if it is not explicitly specified by the process variables, but only noted in a comment. A similar problem exists with all signals that are not binary. This means that they can either be received by different units, or transmit more than two different values. With the control action `mode`, for example, it is not clear, which mode the system is being changed to, only that it is being changed. For this reason, there are also comments at these UCAs that indicate, which mode the system is to be changed to, in order for the UCA to occur.

All UCAs for the unit fail protocol were only identified exemplary for the Outside Controller when the Heater fails, as they are the same for all unit controller pairs, except that the respective failure is from a different component. The Heater was selected here as component and not a unit from the specification because the controller, which manages the Heater, had only a few process variables, so that the context table could still be used. The problems that arose with more process variables are described later in Section 4.2.2. The unit fail protocol comprises two control actions, the *FailDet* and the *RepAck*. The *heaterDefective*, *heaterDefectiveAck*, *heaterDefectiveBefore* and *transmissionFailDet* signal are binary. It is considered a transmission failure if either a signal has been received, whose presence is aberrant, or if a signal has not been received, whose presence is indispensable. The latter is recognized by a timeout. It could either occur because the controller was delayed in giving the signal or because the signal was lost in between. The type *stoppedTooSoon* can be neglected for both control action, as the signals are not continuous. Otherwise, it is for example considered unsafe if the *heaterFailDec* was `not provided` when the heater is defective, the program detects no transmission failure, but there is also no acknowledgement from the heater, that would indicate that the control action was provided. This could result in a wrong mode and therefore a wrong temperature of the heater and possible loss of integrity of the steam boiler. For the *heaterRepAck*, two variables are used to decide if the repair acknowledgement is needed as described in Section 4.1.2. It is for example necessary to provide a *heaterRepAck* signal if the `heaterDefective` is false and `heaterDefectiveBefore` is true. All UCAs for the unit fail protocol can be seen in Listing 4.7.

```
RL38 {
 controlAction: OutsideController.heaterFailDec
 type: not-provided
 contexts: {
    UCA130 [heaterDefective=yes, heaterDefectiveAck=notReceived, transmissionFailDet=no]
         [H2, H3, H4]
 }
}
RL39 {
 controlAction: OutsideController.heaterFailDec
 type: provided
 contexts: {
    UCA131 [heaterDefective=no] [H4, H5]
 }
}
RL40 {
 controlAction: OutsideController.heaterFailDec
 type: too-late
 contexts: {
    UCA132 [heaterDefective=yes] [H2, H3, H4]
 }
}
RL41 {
 controlAction: OutsideController.heaterRepAck
 type: not-provided
 contexts: {
    UCA133 [heaterDefective=no, heaterDefectiveBefore=yes, transmissionFailDet=no] [H4, H5]
 }
}
RL42 {
 controlAction: OutsideController.heaterRepAck
 type: provided
 contexts: {
    UCA134 [heaterDefective=yes] [H2, H3, H4]
    UCA135 [heaterDefectiveBefore=no] [H5]
 }
}
RL43 {
 controlAction: OutsideController.heaterRepAck
 type: too-late
 contexts: {
    UCA136 [heaterDefective=no, heaterDefectiveBefore=yes] [H4, H5]
 }
}
```

Listing 4.7. Exemplary UCAs for the unit fail protocol, exemplified on the heater.

The following is a brief overview of the contexts of the remaining UCAs, which did not have any major problems. Firstly, the control actions of the Operator are considered. Here, a UCA occurs when the Operator initiates an *emergency_stop* if it is not an emergency or does not do so if it is an emergency. An emergency is hereby defined by the same circumstances in which the program switches in the emergency stop mode in the specification [Abr05]. The Heater should be stopped if the system is in *emergency_stop* or the Heater has an error, and should already be started in the initialization mode. In addition, the program should be started if there is no defect, and a physical unit should be repaired if it is defective.

The remaining UCAs of the Outside Controller occur when the Heater is switched on or off at an incorrect temperature. This means that the Heater is switched on when it is too hot and switched off when it is too cold. Additionally, these control actions should always be sent in time.

The remaining control actions of the Controller follow next. It outputs the PROGRAM_-READY signal if *waterLevel* is *insideBoundary*, the valve is closed and the steam is zero. The mode must be changed according to the specification. The remaining UCAs of the valve control action occur if the valve is used in an incorrect mode or at an incorrect water level. Moreover, the throughput should be promptly newly set if the water level goes out of the safety range.

Finally, the UCAs of the Pump Controller occur if it does not open or close the pumps at the correct water level or if the unit fail protocol is not executed as already described for the Heater. The complete list of all the UCAs determined in this work, which were not discussed in detail here, can be found on `github`.

4.1.4 Identify Loss Scenarios

In the final step, I defined loss scenarios for the respective UCAs. For these, the associated UCA and the connection to the respective hazards were specified, while the scenario and its context had to be written down textually. Some examples of loss scenarios from the control action valve are shown in Listing 4.8.

```
Scenario58 for UCA70
 "The Controller fails when the mode is initialization, the water level is above the
  boundary and the valve is still closed, causing the control action to open the valve not
  to be provided." [H1.2, H4]
Scenario60 for UCA70
 "The mode is initialization, the water level is above the boundary and the valve is still
  closed, but the Controller believes that the water level is inside its boundaries. So that
  the control action to open the valve is not provided. This could occur, when:
   - the water level sensor failed
   - the response from the water level sensor was delayed
   - due to wear over the time, has the sensor a drift in its data, leading to a wrong
     feedback
   - depending on the specific sensor used, there could be different inferences with the
     signal (temperature changes, water disturbances, impurities in water, scale)
   - error in the controllers' software, leading to a wrong interpretation of
   the water level" [H1.2, H4]
Scenario67 for UCA73
  "The mode is normal, but the Controller provides the valve control actions because it
   believes that the mode is initialization, resulting in falsely moving the valve. This
   flawed model will occur when the following holds:
    - The mode change from the Controller is delayed
    - The Controller calculated the wrong mode" [H1, H3, H4, H5]
Scenario72 for UCA79
  "The mode is initialization, the water level is above normal, the valve is closed. The
  processing of the correct valve state was delayed, so that the action to open the valve
  came too late." [H1.2, H4]
Scenario73 for UCA79
  "The mode is initialization, the water level is above normal, the valve is closed. The
   transmission of the correct valve state was delayed, so that the action to open the valve
   came too late and was already detected as a transmission failure." [H1.2, H4]
```

Listing 4.8. Loss Scenarios for the *valve* control action.

Most of the loss scenarios of *valve* and other control action can be traced back to following action sequences. The associated controller has a failure because of which a control action can not be sent. This is a possible scenario for all UCAs that have the type `not provided`. An example scenario with this sequence of events can be seen in `Scenario58`.

It is also possible for an UCA to occur if some part of the program was delayed. This can either happen, while a sensor or the controller is processing, or when the transmission happens. `Scenario72` and `Scenario73` are examples for this kind of scenario that leads to a too late UCA.

Another possibility for the occurrence of UCAs is the wrong belief of a controller, which means that a process variable has another value then the controller believes. The reasons for this wrong belief can be diverse. It is possible that the corresponding sensor, that is responsible for the value of the process variable, has failed. Other reasons are that due to wear over time the sensor has got a drift in its value, so that it does not transmit the real value any more. Such a wrong reading could also occur, when the sensors were not properly calibrated before the system was started or when it was not correctly installed or repaired in the first place. Moreover, environmental factors can lead to incorrect measurements due to interference. The environmental factors are varied and differ for the various sensors.

All sensors may encounter issues with rapid temperature changes, potentially resulting in incorrect measurements. Additionally, the water level sensor could provide inaccurate readings due to water disturbances, impurities of the water, or limescale build-up on the sensor over time. The steam level sensor might face challenges with wet steam, condensation, or blockages caused by limescale or other small particles. Dust and debris in the system, electrical noise, or excessive moisture, could negatively affect the single pump sensor. Lastly, the heat sensor could also experience problems due to dust and debris or excessive moisture, as well as electromagnetic interference, or electrical noise.

It is also possible that the program has a wrong belief, if the controller has a software error, which leads to a wrong interpretation of the correct value. Examples of these kinds of action sequences that lead to an incorrect belief are shown in `Scenario60`. The control algorithm can also be wrong in the sense that it does not misinterpret a value, but calculates the control action incorrectly with the values given to it. An example for this case would be a wrong system mode, shown in `Scenario67`.

Another critical aspect of the system that could lead to incorrect beliefs involves the transmissions. On the one hand, these can be delayed, so that value was delivered too late. On the other hand, a signal can be corrupted, so that it was misinterpreted as a different value or the transmission line is noisy, which leads the system to perceive the noise as a signal.

For the UCAs that handle the unit fail protocol, wrong beliefs about the status of a system component also occur, when the system dynamics are not well defined, leading a controller to a wrong interpretation of the status of a physical unit. This also occurs when the component's feedback is incorrect but still compatible with the system dynamics, so that a failure is not detected by the program. Since this work introduces additional components that may also be defective, I wrote a specification that has been added in the form of comments to these components, to indicate when the respective component is considered defective. The valve can be seen as defective, when the water level does not change if the valve should be open or when the water level changes even though the valve should be closed, the pumps are off and the system is still in initialization mode, so that the heater is still turned off. Moreover, the heater is defective, when its value lies outside the possible limits or when the program detects that the unit indicates a value, which is incompatible with the dynamics of the system.

In general, false beliefs are a major factor that can lead to UCAs of any type and therefore occur in many scenarios.

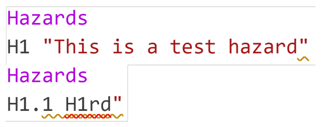
4.2 Challenges

A number of challenges arose during the analysis of the steam boiler, which are described in more detail below. It will be discussed how the respective challenges arose and what options were available to circumvent them. Furthermore, I explain the impact these resulting limitations had on the analysis.

4.2.1 Automatic Identifier Updates

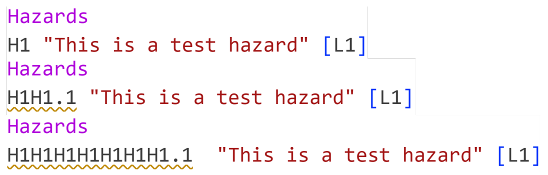
The challenge that was the most time-consuming to fix in this analysis, was due to the automated creation of the IDs. This is the case, because there are multiple different defects that can occur with the creation. Most of them can easily be fixed by undoing the previous action with `ctrl + Z`. However, this process can sometimes be more time-consuming if the program has performed numerous actions to create the IDs, as all these actions must now be undone. This is particularly time-consuming if the analyst does not immediately realize that the language server has broken the IDs or when there are a lot of changes. In this case, the incorrectly made updates have to be adjusted manually.

One of the possible ways, for the IDs to break is that changing the ID sometimes writes into the string behind it, which is used to describe the element, so that the analyst must change the text again manually. For example, it is possible to define a hazard for which the ID and the description are already written down, but not yet the connection to the loss. If a `.1` is now inserted after the ID, the ID is copied to the fourth last position in the string, and then everything between the new and old ID is deleted. However, if the loss connection is written down first and then the `.1` is inserted, the ID is inserted before the existing ID. In addition, each further keystroke inserts further IDs at the front. Both cases can be seen in Figure 4.4. On the left side is the sequence of the first described case shown and on the right side that of the second.



```
Hazards
H1 "This is a test hazard"
Hazards
H1.1 H1rd"
```

(a) Hazard IDs copied into description.



```
Hazards
H1 "This is a test hazard" [L1]
Hazards
H1H1.1 "This is a test hazard" [L1]
Hazards
H1H1H1H1H1H1H1.1  "This is a test hazard" [L1]
```

(b) Duplication of IDs in hazards.

Figure 4.4. Challenges with hazard IDs.

However, IDs have not only posed a challenge for hazards, but also for UCAs and loss scenarios. I was unable to determine the cause of the incorrect creation of the IDs. However, I had the feeling that a large system and a larger number of elements led to more problems occurring. For example, the IDs were usually determined incorrectly if many had to be updated. The language server could write the IDs incorrectly in various ways. For example, numbers could be duplicated, or written in the wrong order, keywords and comments were overwritten or IDs in general put at the wrong place.

Another challenge that occurred with the IDs was that the automatic updating of them sometimes took a lot of time. For instance, adding a new loss scenario at a higher position could require updating around a hundred IDs, causing the analyst to wait several minutes for the changes to take effect. However, the processing time was not always consistent. In some cases, the IDs were updated immediately, even in similar situations.

4.2.2 Context Table

More challenges occurred, while using the context tables. Firstly, the procedure used to create the context tables in PASTA is still very simple. This means that the table is created by brute force and can only ever be displayed in full for one control action, which makes its creation very slow. For this reason, tables with more than sixteen variables can no longer be displayed. In the control structure for the steam boiler was this the case for the Controller. The Pump Subsystem was created to reduce the amount of process variables in the Controller's process model. However, both the Controller and the Pump Controller had in the final control structure still too many process variables, so that the visualization of the context table cannot be used because the program stopped working.

Another challenge encountered while using the context tables was their limitation to only displaying controllers with a depth one. For example, it was not possible to select control actions of the Pump Controller. This is the case, because it is located within the Pump Subsystem, which is located inside the Physical Units, and as a result has a depth of three. Consequently, the table was unable to display the control action because the context table had not yet been expanded to include subsystems.

Lastly, there were two small challenges that made writing contexts more difficult. On the one hand, the fact is that not only process variables that are possible for the respective control action are suggested in the contexts, but also process variables from other controllers or variables that are not even process variables. It would speed up the process of UCAs context creation, if there would be a feature that only suggests process variables that actually match the respective control action. For example, in many IDEs if one wants to call a function of a class, only the possible functions of that class are displayed as suggestions. A similar feature would also be good for PASTA. On the other hand, it is not possible to write a context of an UCA, while having the context table open. It switches the view from the context table to the diagram after an automatic update that occurs after some time. This is problematic because one cannot see the context table, while writing the rules. This makes the process of identifying UCAs more time-consuming because the analyst needs to switch manually back to the context table. Moreover, sometimes this leads to a server crash, so that the program needs to be restarted.

The controller constraints were not specified in this paper because the challenges mentioned above required a lot of time and most constraints only say the opposite of the respective UCA. As an example for `UCA70`, the constraint for the controller would be `C70 "In initialization mode, if the waterLevel is aboveBoundary and the valve is closed, the Controller must give the control action valve to open the Valve." [UCA70]`.

4.2.3 Loss Scenarios

The biggest challenge with the loss scenarios were the amount of text that have to be written manually. This amount can significantly be reduced by automating the text for the contexts of the loss scenarios. At the moment, the process for writing loss scenarios is as follows. First, the ID of the scenario and the associated UCA are specified. Then the control action, the controller that executed it, the context and the actual scenario that triggers the UCA are written down textually in a string. This effort can be reduced, as the control action, controller and context are already known through the connection to the UCA and the context table, allowing them to be noted automatically.

It should also be considered whether this additional information is needed at all at this point or whether it would not be sufficient if it were added automatically when the report is created. By removing this information, the loss scenarios would gain in clarity, and it would be easier to recognize how they differ. However, the analyst would then no longer be able to find the context in the same place as a loss scenario when creating the report. This could be particularly confusing for users who have already carried out STPAs with other tools, as the context is also part of the scenario according to the STPA handbook [Lev18].

Due to these challenges and the large number of UCAs, writing the loss scenarios took a lot of time. As a result, there is not a scenario for every UCA. For time reasons, UCAs related to the operator were omitted, as they are not part of the specification. In addition, UCAs for the throughput were also excluded since they deviate from the specification. For the UCAs related to the `mode` control action, only the scenarios for `not-provided` were detailed further.

Chapter 5

The Steam Boiler Model in Sequentially Constructive Statecharts

The focus of this thesis is to design the steam boiler in SCCharts, to later evaluate the usefulness of the language in aiding in the modelling. For this reason, a steam boiler was modelled, which adheres very closely to the specification and focuses on verification. This chapter presents the process of designing the steam boiler in KIELER and addresses the challenges that occurred and how these were solved.

5.1 Modelling

This section explains the process of designing the steam boiler with SCCharts. It explores the individual approaches that were looked into to design a model that is easily to verify. Additionally, it shows the process of designing and explains in detail certain special cases where either the specification was deviated from and the reasons behind these decisions or where a specific approach was chosen due to the limitations of the tool.

5.1.1 Steam Boiler Modelling Process

When I started modelling the system, I was initially heavily guided by the specification, as this described how the individual components should communicate with each other. In addition, even before I started modelling, I considered a few approaches as to what features the model should display so that it can be verified later. These approaches are explained in more detail in Section 5.1.2. I also orientated myself on the steam boilers from Schulz-Rosengarten [Sch24a] and Smyth et al. [SDH19], which were already modelled in SCCharts but had different focuses. Here, I have adopted individual aspects from both of them that I found advantageous for the analysis. As an example, I took up a similar idea to the abstract classes from Schulz-Rosengarten, which were used to write the models for the unit fail protocol. In this way, the model for the program and for the units only needed to be written once and were then bound with the correct variables in the respective components. From Smyth's work, I had the idea of writing an environment and representing the whole system as a dataflow region. A more detailed comparison of the individual steam boiler models follows in Section 6.2.4.

After that I started modelling with the easiest component, which is in the case of the steam boiler system the valve. It had only three features that it needed to fulfil. First of all, it needed to open and close when given the `VALVE` command. Furthermore, it needed to detect a transmission failure when either the `VALVE` command was given outside the initialization mode or when the `booting` signal was given multiple times. The valve does not need to handle the unit fail protocol because it does not break according to the specification. The model was also used in Section 5.1.3 to show that the concepts considered in this thesis work, as it is relatively small and so that there are few requirements that the model must fulfil in order to be verified.

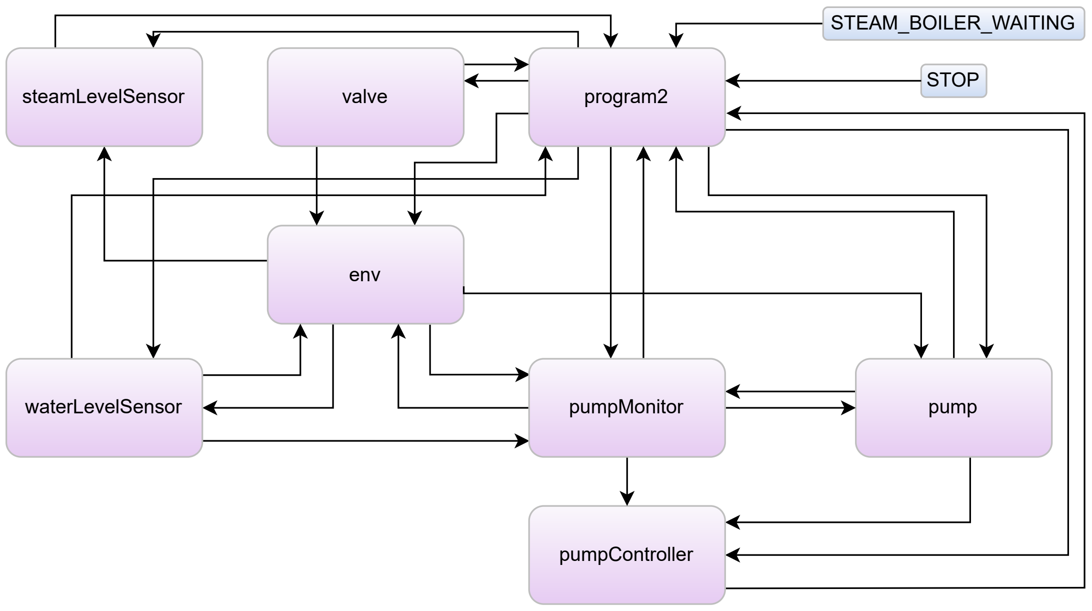


Figure 5.1. Abstracted steam boiler model

Figure 5.1 shows the composition of the main components of the steam boiler. The `program2` has a minimal version of the responsibilities that are mentioned in the specification. It initializes the steam boiler by checking the steam value and then adjusting the water level with either the valve or the pumps. Furthermore, it does change the modes according to the sensor values and detects transmission failure for its signals. Other functionalities such as starting and stopping the pumps or detecting when the physical units are defect, were outsourced to other models. This was done because it was unclear if the program module would work without compile error, which is explained in more detail in Section 5.2. The Pump Subsystem that was seen in the STPA was here again changed to the pumps and their respective sensors that are controlled by the program as given in the specification. The `pump` and `pumpController` are therefore in charge of the same aspects that are described in the specification. The `pumpMonitor` was added, which opens and closes the pumps according to the water level and calculates the pumps' throughput. Moreover, it also closes the pumps in case of the system switching into the emergency mode. Furthermore, the system consists of two other sensors, namely the water level sensor and the steam level sensor, which measure the respective variable's value and detect if there are inconsistencies with the values. They then pass this on to the program as a defect. The last physical unit that is part of the system is the valve. It is only used in the initialization mode for emptying the water out of the boiler. The last module of the model is the environment. It is responsible for simulating the physical conditions in which the steam boiler would be located. This means that it calculates the water level and steam level, and it indicates when a second has passed. This is required because, as described in Section 2.4, SCCharts do not have any built-in notation of time, so that the time needs to be modelled as signal. Moreover, it also determines the calculated water level when the water level sensor is defective. The last two components are the signals `STEAM_BOILER_WAITING` and `STOP`, which can only be given from outside the system, and start and stop the system. Furthermore, the system has a `Config` file, which is omitted in this graph. The `Config` decalres, for example, the amount of pumps in the system, what the maximum increase and decrease of the water and steam level is or after how much time a transmission is seen as a failure. The arrows in the diagram show how the individual components interact with each other.

These interactions can lead to different transmission failures that each unit must take care of. All physical units must ensure that they only receive the booting signal once. Additionally, all units have to handle the unit fail protocol, except the valve. There are a few additional signals that only individual components have. For example, the valve must ensure that it does not receive the `VALVE` signal outside the initialisation mode. In this mode, the program also has signals that may only be emitted at certain times. For example, the PHYSICAL_-UNITS_READY signal may only be received after PROGRAM_READY has been sent by the program and STEAM_BOILER_WAITING may only be received once.

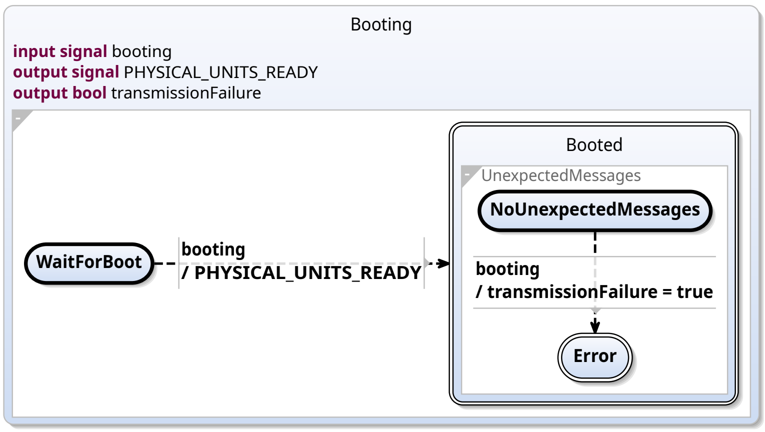


Figure 5.2. The booting of the physical units.

The SCChart shown in Figure 5.2 shows the diagram all physical units reference to detect a transmission failure of the booting signal. The signal should only be sent once to indicate to the units that they should start booting. If they are ready, they must answer the program with a `PHYSICAL_UNITS_READY` signal. The booting protocol is then completed, so that the system switches to a final state. If the booting signal is detected again in this state it is detected as a transmission failure and the system switches in an error state.

Figure 5.3 is showing the model of the unit fail protocol for all physical units. It is initialized in the `Normal` state because there is no defect detected. If the unit then gets a fail message, it switches in the `Failed` state and acknowledges the fail message with an *failAckMsg*. The unit sends a repair message to the program and waits for the respective repair acknowledgement, after it got repaired. The system has a transmission failure if the repair acknowledgement is received in any other state or if the acknowledgement needs more ticks then specified in the `Config` for the `transmission_timeout`. The unit transitions into a final error state if a transmission failure is detected. Normally, this state would then be joined to a final error state on the highest depth level. This is done on the one hand to make the model easier to understand, as it is clear that the system has an error in this state. On the other hand, it is modelled so that the superstate cannot be left with an abort when an error has occurred. However, this option was not chosen here, as it triggered a compiler error. Instead, it is checked before each transition whether the *transmissionFailure* is true, as in this case there is an error in the system and no transition should take place. The last step is to switch back to the normal state when the unit has received the repair acknowledgement.

The specification does not define what happens if the failure message is sent at the same time or even before the repair message is acknowledged. For this reason, I have specified a behaviour that the program must follow. It exists two cases that needs to be considers: the signals are received at the same time, the failure message is received before the acknowledgement. In the first scenario, the model first transitions to `Normal` and then in the same tick to `Failed`. In the second scenario, it transitions directly to `Failed`. It should not actually be possible for the *failMsg* to be sent before the *repairAckMgs*, as the *repairMsg* is responded to immediately, while a new *failMsg* is delayed by at least one tick in the `program2`. However, if for some reason the *repairAckMsg* is lost and a new *failMsg* is already received, this is not recognized as an error by the program, as the new error is more important than the acknowledgement. The importance of the *failMsg* is therefore the reason that the state at the end of a tick in both cases is `Failed`.

Figure 5.4 shows the unit fail protocol of the program. It is quite similar to the model of the physical units as it shows the responding side. It also starts in `Normal` state and switches when the unit has a defect. The program then waits for the acknowledgement of the failure message. It is also not specified what happens if the acknowledgement and the repair message are received at the same time. In a similar form to the units diagram, I specified that a signal is more important than an acknowledgement. For this reason, the model always ends in the `Repaired` state. When both signal are present at the same time, the diagram first transitions to `FailAck` and in the tick then to `Repaired`. In the case, that the *repairMsg* is received first, it directly transitions to the `Repaired` state and is not classified as transmission failure. A transmission failure occurs, if the acknowledgement takes too long, if an acknowledgement or a repair message is received in the `Normal` state or when an acknowledgement is received twice.

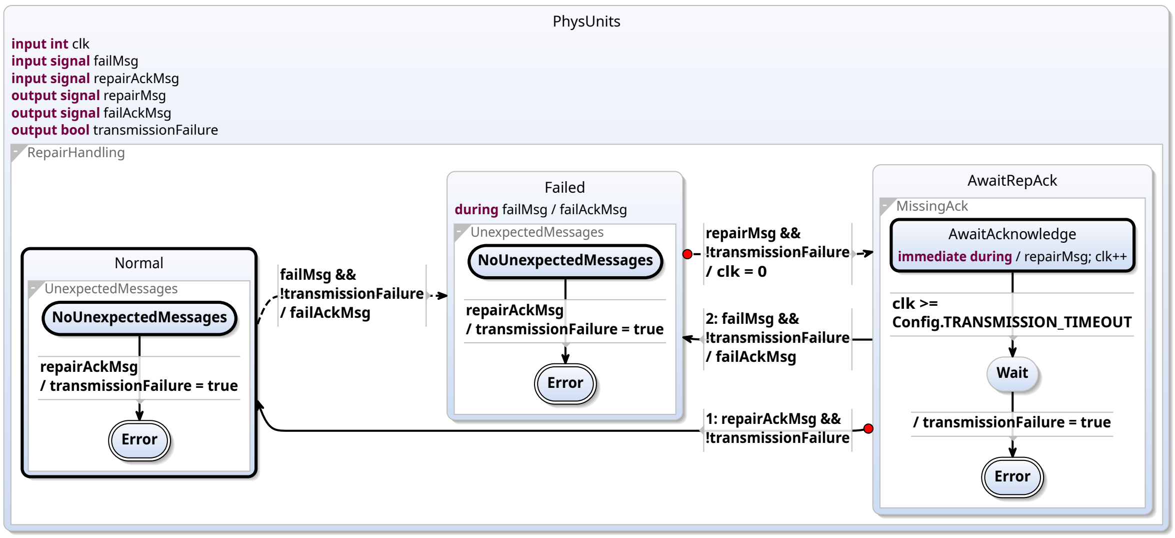

PhysUnits
input int clk
input signal failMsg
input signal repairAckMsg
output signal repairMsg
output signal failAckMsg
output bool transmissionFailure
RepairHandling
Normal
UnexpectedMessages
NoUnexpectedMessages
repairAckMsg / transmissionFailure = true
Error
failMsg && !transmissionFailure / failAckMsg
Failed
during failMsg / failAckMsg
UnexpectedMessages
NoUnexpectedMessages
repairAckMsg / transmissionFailure = true
Error
repairMsg && !transmissionFailure / clk = 0
2: failMsg && !transmissionFailure / failAckMsg
1: repairAckMsg && !transmissionFailure
AwaitRepAck
MissingAck
AwaitAcknowledge
immediate during / repairMsg; clk++
clk >= Config.TRANSMISSION_TIMEOUT
Wait
/ transmissionFailure = true
Error

Figure 5.3. Unit fail protocol of the physical units.

These error states are then joined to a final error state due to the advantages described above as this lead to no error. Lastly, the model transitions back to `Normal` if the repair message is no longer received. This is the case, because the repair message is sent as long as the acknowledgement was not received. It therefore must be assumed, that the acknowledgement was only received when no more repair message was sent.

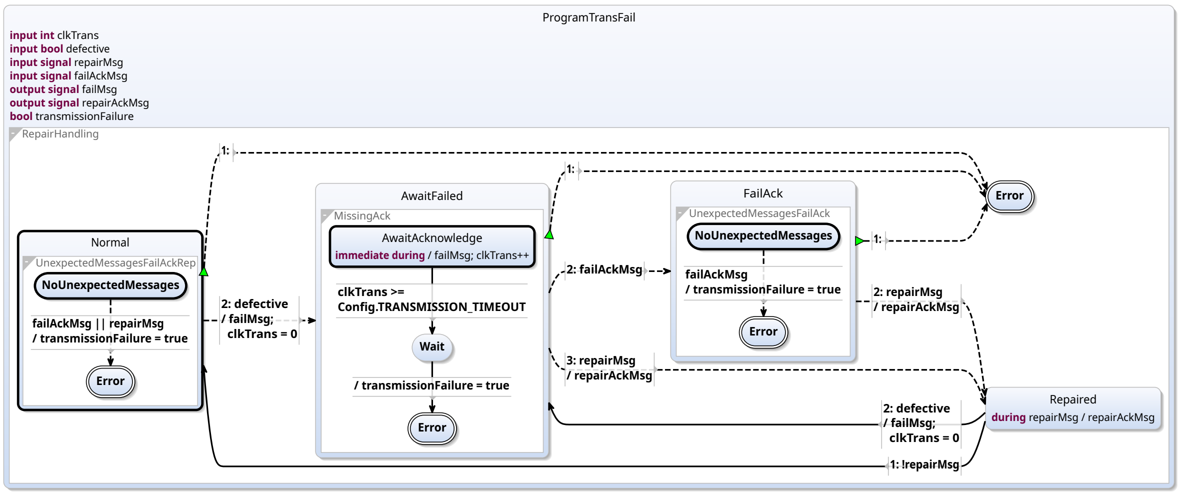

ProgramTransFail
input int clkTrans
input bool defective
input signal repairMsg
input signal failAckMsg
output signal failMsg
output signal repairAckMsg
bool transmissionFailure
RepairHandling
Normal
UnexpectedMessagesFailAckRep
NoUnexpectedMessages
failAckMsg || repairMsg / transmissionFailure = true
Error
1:
2: defective / failMsg; clkTrans = 0
AwaitFailed
MissingAck
AwaitAcknowledge
immediate during / failMsg; clkTrans++
clkTrans >= Config.TRANSMISSION_TIMEOUT
Wait
/ transmissionFailure = true
Error
1:
2: failAckMsg
3: repairMsg / repairAckMsg
FailAck
UnexpectedMessagesFailAck
NoUnexpectedMessages
failAckMsg / transmissionFailure = true
Error
1:
2: repairMsg / repairAckMsg
Error
Repaired
during repairMsg / repairAckMsg
2: defective / failMsg; clkTrans = 0
1: !repairMsg

Figure 5.4. Unit fail protocol of the program.

Figure 5.5 shows the SCChart of the program. The local declarations as well as input and output variables are omitted due to the clarity of the diagram. The collapsed regions consist of the unit fail protocol with the different physical units and the check for transmission failures

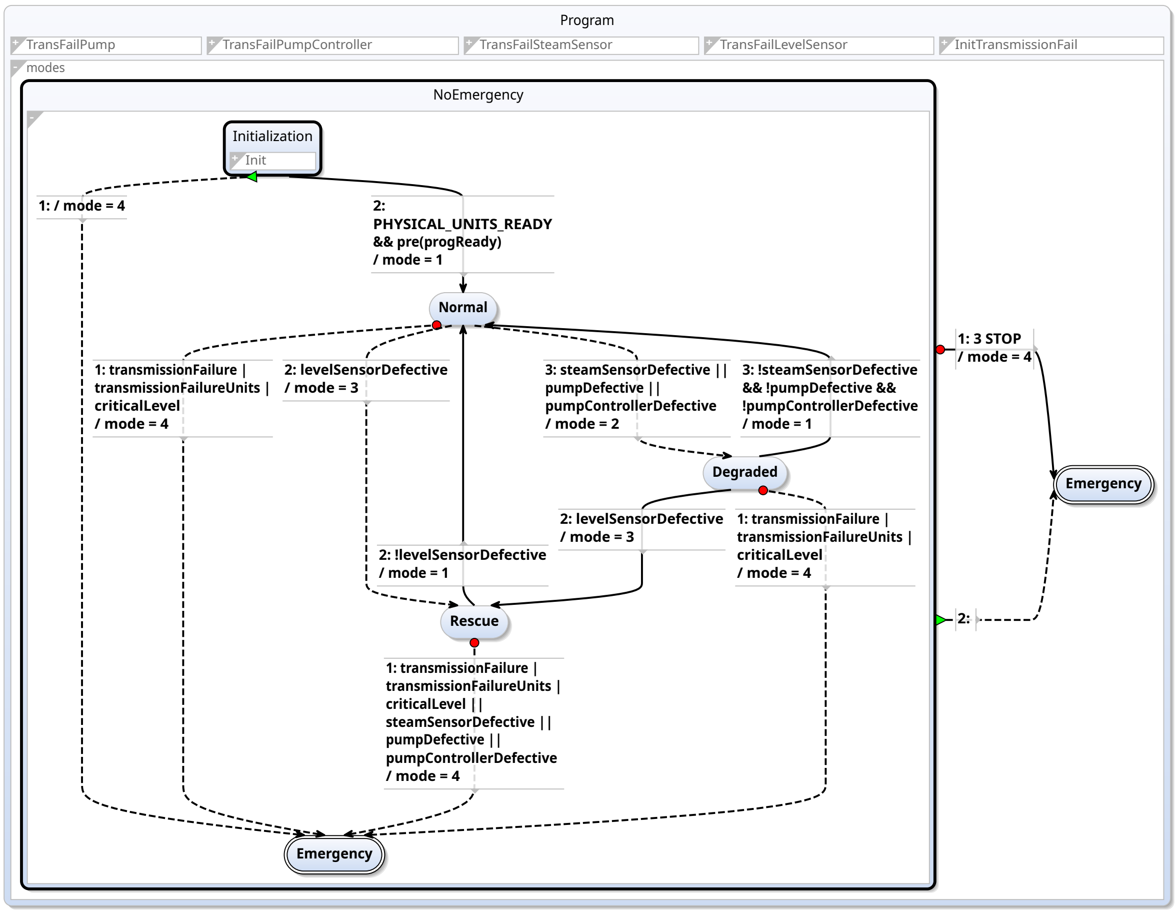
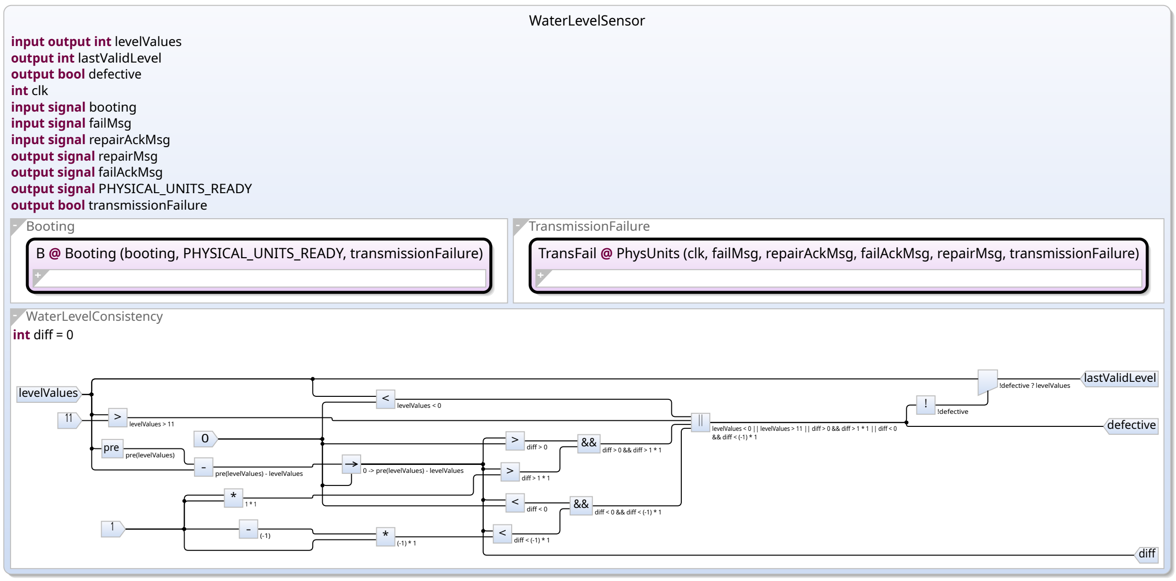

Program
TransFailPump
TransFailPumpController
TransFailSteamSensor
TransFailLevelSensor
InitTransmissionFail
modes
NoEmergency
Initialization
Init
1: / mode = 4
2: PHYSICAL_UNITS_READY && pre(progReady) / mode = 1
Normal
1: transmissionFailure | transmissionFailureUnits | criticalLevel / mode = 4
2: levelSensorDefective / mode = 3
3: steamSensorDefective || pumpDefective || pumpControllerDefective / mode = 2
3: !steamSensorDefective && !pumpDefective && !pumpControllerDefective / mode = 1
Degraded
2: levelSensorDefective / mode = 3
1: transmissionFailure | transmissionFailureUnits | criticalLevel / mode = 4
2: !levelSensorDefective / mode = 1
Rescue
1: transmissionFailure | transmissionFailureUnits | criticalLevel || steamSensorDefective || pumpDefective || pumpControllerDefective / mode = 4
Emergency
1: 3 STOP / mode = 4
Emergency
2:

Figure 5.5. `Program2` SCChart.

of the `STEAM_BOILER_WAITING` and `PHYSICAL_UNITS_READY` signals. The `mode` region shows how the program switches between the different modi described in the specification. If there was no error in the initialisation phase, the system goes into the normal mode. It switches to the degraded mode if a physical unit has a defect, and to the rescue mode if the water level sensor is defective. In this mode, the water level is calculated by the environment and no longer measured by the sensor. Because the logic of which water level value has to be taken is defined in the `plant2` module, the implementing the control logic for program as well as the Pump Monitor is independent of the source of the water level information. The system switches to emergency mode if a transmission error was detected or multiple critical physical components fail. The switch to a mode in which more units are defective is always immediate, whereby the switch back when the units are no longer defective is delayed. This is the case, because the defect of a component is more critical than the repair of a unit. After a failure, the superestate joins to a final emergency state, which can also be transitions to if the operator sends the `STOP` signal three times.

As described above, it is not the program that detects the failures of the physical units, but the physical units themselves, which then send the information to the program. This was done because the units already have all the information needed to detect an error, thus preventing the program from ever working with incorrect values and possibly making the wrong decisions.

As an example of failure detection of a physical unit, the model for the water level sensor is examined in more detail. The model can be seen in Figure 5.6. The `Booting` and `TransmissionFailure` regions contain the models described above for booting the unit and the unit fail protocol. In the last region, `WaterLevelConsistency`, it is determined whether the water level sensor has a defect and what the last valid water level was. A dataflow region was chosen for this region as it allows direct computations within a region and the region has otherwise no real state change.



```
WaterLevelSensor
input output int levelValues
output int lastValidLevel
output bool defective
int clk
input signal booting
input signal failMsg
input signal repairAckMsg
output signal repairMsg
output signal failAckMsg
output signal PHYSICAL_UNITS_READY
output bool transmissionFailure

Booting
B @ Booting (booting, PHYSICAL_UNITS_READY, transmissionFailure)

TransmissionFailure
TransFail @ PhysUnits (clk, failMsg, repairAckMsg, failAckMsg, repairMsg, transmissionFailure)

WaterLevelConsistency
int diff = 0
```

Figure 5.6. Model of the water level sensor.

There is an error in the water level sensor, as defined in the specification [Abr05], if the water level is outside the possible values. This means that it is either less than zero or greater than the capacity of the steam boiler. Otherwise, it is recognised as a defect if the water level changes faster than it should be able to according to the configuration. Furthermore, the water level is saved as `lastValidLevel` if the sensor is not recognised as defective. Original, the `defective` was defined with a config file as seen in Listing 5.1. The advantage of this approach was, that on the one hand, values can easily be changed and stayed the same in each module without the need to change all of them manually. On the other hand, it is more evident what each number stands for without having to look it up, making the diagram easier to understand. However, the region has now the real numbers at the places of the config file because the program has otherwise produced an error.

$$[0, M_1], (M_1, M_{1.1}), [M_{1.1}, N_{1.2}], (N_{1.2}, N_1], (N_1, N_{1.1}],$$
$$(N_{1.1}, N_{2.2}], (N_{2.2}, N_2), [N_2, N_{2.1}), [N_{2.1}, M_{2.2}], (M_{2.2}, M_2), [M_2, C), [C]$$

Figure 5.7. Water level ranges.

```
defective = levelValues < 0 || levelValues > scchart(Config).C ||
        (diff > 0 && diff > (scchart(Config).P * scchart(Config).NUM_PUMPS) )||
        (diff < 0 && diff < (-scchart(Config).P * scchart(Config).NUM_PUMPS))
```

Listing 5.1. Original code of the defective value of the water level sensor.

As a result of the challenges discussed in Section 5.2, it was not possible to produce a fully functional program. The single components are working and can be simulated but due to scheduler errors in the `plant2` module, which combines all components so that they can communicate, the model can not be compiled. The scheduler errors occur because of instantaneous loops.

5.1.2 Approaches for Developing a Verifiable Model

One idea to simplify the verification of the model was to severely limit the values of water level and steam level, as a large number of values slow down the verification considerably. Accordingly, it was decided to use a very small number of integers in the model to represent the respective values. The individual integers stand for different ranges in which the water- or steam level can be located. The ranges for the water level can be seen in Figure 5.7. C is again the maximum capacity of the boiler. In addition, the already known values for the optimal production of steam N_1 and N_2 and the safety values M_1 and M_2 in which the water level must be located, have been extended by further division of the values. Values with the suffix `.1` indicate that the value is slightly larger, and values with the suffix `.2` indicate a slightly smaller value. This results in an order relation for the variable N_1: $N_{1.2} < N_1 < N_{1.1}$. The more detailed categorisation is necessary as this is the only way to find problems with transitions. If these ranges are then translated into integers, the numbers from 0 to 11 are obtained.

One of the biggest challenges with model checker is the state explosion problem [CKN+12]. The problem describes the circumstance that as the number of state variables in the system increases, the size of the system state space grows exponentially. A model checker now needs to check the whole state space in order to verify the model, but this is difficult with the growth of the state space because the verification needs too long. As a result, the second idea is, to limit the amount of states that are used in the model to limit the state explosion.

The procedure for implementing this approach was to consider for each SCChart whether a state was really necessary or whether it can be combined with another without changing the desired behaviour. It was also necessary to consider whether states should be reduced or whether the diagram should still be easy for people to read. For example, the initial idea was to combine the *degraded* mode and the *normal* mode in the program. This would have been possible because both require the same water level and both have the same transitions to other modes. However, it would no longer have been possible to tell from the state whether all physical units were intact or whether some had a defect. In the end, I decided against this approach because the additional state makes the model easier to understand, and it is also more similar to the specification.

To further avoid the state explosion problem, the third idea was to modularise the system. Modular verification can be conducted by partitioning the system into components, each one corresponding to a subsystem in the specification. In turn, the global properties will be modularised in local properties, enabling each one to be verified over the proposed components. One idea here was to install a pump subsystem, as with STPA, for example, and then verify this in a modular way.

Only the first of these approaches was used in the actual model. This is due to the fact that the compiler created many instantaneous loops, which could often only be solved by deviating from the considerations. For example, a new `Wait` state was inserted at certain points where a loop would have been created without the additional tick the state provides. However, this does not correspond to the approach that as few states as possible should be used.

5.1.3 Verification

As proof of concept, the Valve was exemplary used to show that the SCChart models can be verified using the ideas from this work if the challenges from Section 5.2 had not occurred. The SCChart Valve can be seen in Figure 5.8. It has three regions: `Booting`, `TransmissionFailure` and `Valve`.

The `Booting` region was already described in detail above. The `TransmissionFailure` is responsible for monitoring that the *VALVE* signal only gets sent in initialization mode. That means that after the first change to a mode unequal zero, as that is the number for initialization, the Valve transmit a transmission failure if it receives another *VALVE* signal. Furthermore, is it not possible for the program to change the mode back to zero without restarting, which is why there is no transition back to `Initialization`. In the `Valve` region, the actual function of the Valve happens, namely that it either opens or closes when it receives the *VALVE* signal, depending on which state it is currently in. The state is then noted in the `valve` variable, where `true` indicates open and `false` indicates closed. It should be noted here that the state of the Valve cannot be switched in the first tick. Moreover, the *VALVE* signal either never occurs or occurs in even numbers, with a correct program sequence, as the valve must always be closed at the end and it starts in a closed position.

```
@LTL "F(booting -> F(PHYSICAL_UNITS_READY)) && F(booting -> X(G(booting -> transmissionFailure)
   || G(!booting)))"
@LTL "F(Mode != 0 -> X(G(VALVE -> transmissionFailure) || G(!VALVE)))"
@LTL "G(!VALVE && !valve) || G(VALVE -> (valve -> F(VALVE -> !valve)))"
```

Listing 5.2. LTL formualas for the valve.

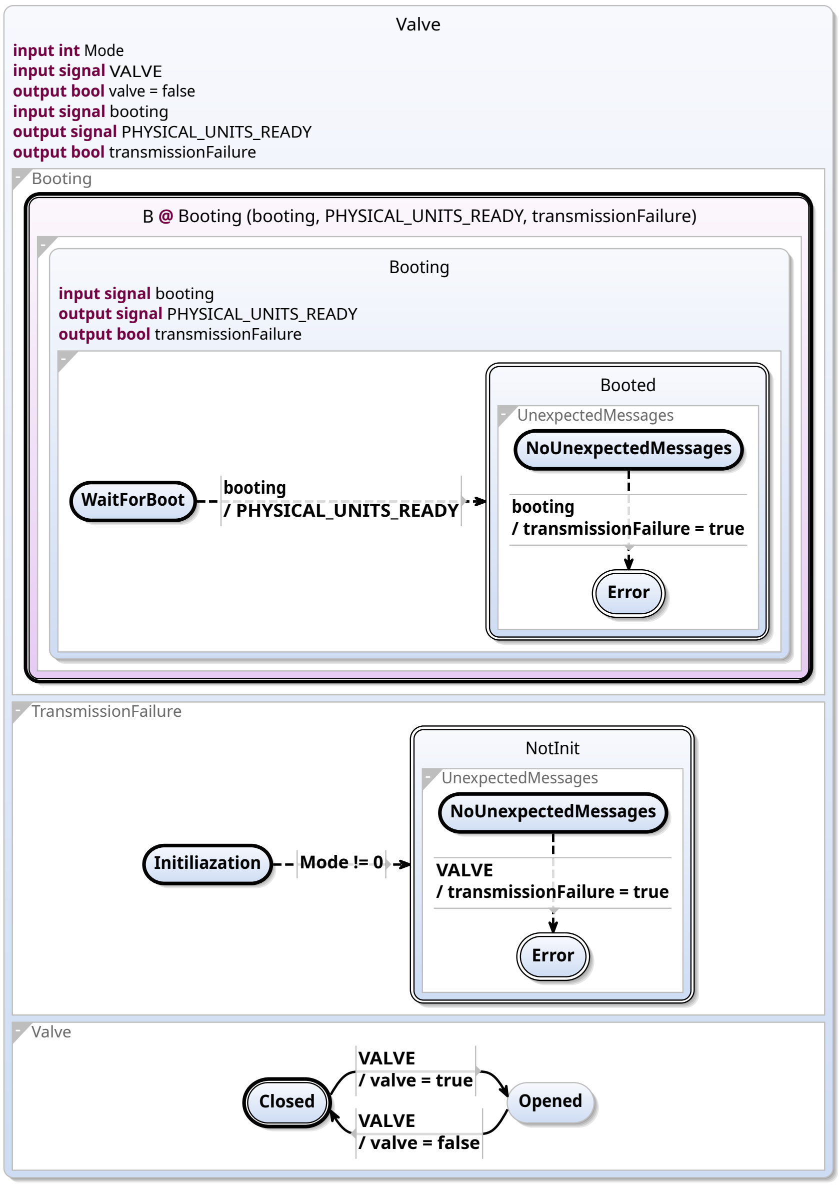

Valve
input int Mode
input signal VALVE
output bool valve = false
input signal booting
output signal PHYSICAL_UNITS_READY
output bool transmissionFailure
Booting
B @ Booting (booting, PHYSICAL_UNITS_READY, transmissionFailure)
Booting
input signal booting
output signal PHYSICAL_UNITS_READY
output bool transmissionFailure
WaitForBoot
booting / PHYSICAL_UNITS_READY
Booted
UnexpectedMessages
NoUnexpectedMessages
booting / transmissionFailure = true
Error
TransmissionFailure
Initiliazation
Mode != 0
NotInit
UnexpectedMessages
NoUnexpectedMessages
VALVE / transmissionFailure = true
Error
Valve
Closed
VALVE / valve = true
VALVE / valve = false
Opened

Figure 5.8. The Valve in SCCharts.

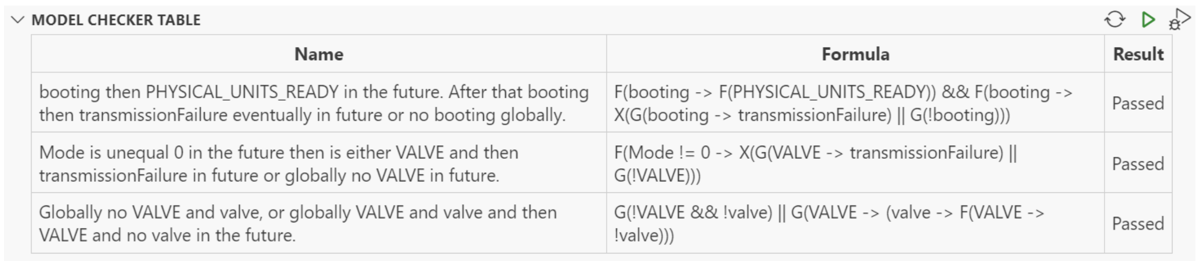
The other component needed for verification, apart from the model, are LTL formulas, which state the rules the model needs to follow in order to be verified. The rules for the Valve model can be seen in Listing 5.2.

The first rule is for the `Booting` region. It says that if there is eventually a `booting` signal, then is `PHYSICAL_UNITS_READY` in the future. Furthermore, if there is a booting signal then from the next step, if another `booting` signal then `transmissionFailure` is true or there is no more `booting` globally after the first `booting`. This rule ensures that the Valve sends a

PHYSICAL_UNITS_READY when it is booted and that it detects a transmission failure if the *booting* signal is issued more than once. The second rule is for the `TransmissionFailure` region. It declares that the `Mode` is eventually unequal to zero some time in the future. Then from the next step, if the signal `VALVE` is received, then `transmissionFailure` is true, or there is globally no signal `VALVE` in the future. This rule ensures that a transmission failure is also detected for the *VALVE* signal, as the signal may only be used in initialisation mode as defined in the specification. The last rule is for the `Valve` region. It states that the signal *VALVE* is either globally never present or it is globally first present and then sometime in the future again present. Hereby, valve is set true in the first instance and wrong in the second instance, which indicates if the valve is opened or closed. This rule therefore ensures that the *VALVE* signal only occurs in even numbers and that the valve boolean is set to the correct state. It is of course possible to define more than one rule for a region, but this was not necessary for the Valve example.

If the model is now verified using the rules mentioned, it is important to note that numerical variables are only permitted if they are integers that also have a specific range in which they can be located. This is because the possible values of the variables would otherwise be too large for them to be verified properly. With Valve, there is only one variable that is an integer, namely the `Mode`. A range from zero to four can be specified here, as it is clear that the mode can have five different values. In general however, the range should always be selected so that all possible integer values are within it, as otherwise possible failures of the rules will not be found. The range can be specified in `KIELER` with `@AssumeRange <start range>, <end range>`.

Running the rules in `KIELER` results in all rules being passed, indicating that the Valve has been verified. The appearance of the model checker in `KIELER` is shown in Figure 5.9, where it is evident that all rules passed and that individual names can be assigned to them. The circle with two arrows are used to reload the model checker, while the green arrow initiates the verification process. The adjacent arrow, labelled `Start Counterexample`, starts a simulation. However, it does not automatically display a counterexample if a rule has failed.



Name	Formula	Result
booting then PHYSICAL_UNITS_READY in the future. After that booting then transmissionFailure eventually in future or no booting globally.	F(booting -> F(PHYSICAL_UNITS_READY)) && F(booting -> X(G(booting -> transmissionFailure) || G(!booting)))	Passed
Mode is unequal 0 in the future then is either VALVE and then transmissionFailure in future or globally no VALVE in future.	F(Mode != 0 -> X(G(VALVE -> transmissionFailure) || G(!VALVE)))	Passed
Globally no VALVE and valve, or globally VALVE and valve and then VALVE and no valve in the future.	G(!VALVE && !valve) || G(VALVE -> (valve -> F(VALVE -> !valve)))	Passed

Figure 5.9. The model checker table in `KIELER`.

5.2 Challenges

The biggest challenge I faced while writing the model, was the creation of instantaneous loops due to how the compiler works. Most of the instantaneous loops developed due to the abortion of superstates. As a result, the `SCChart` was no longer able to compile. I used three different strategies to break the instantaneous loops. Deleting immediate transitions and joins, adding `pre` to values and inserting new states between transitions so that the original transitions take one tick longer. All these states have the name `Wait` in order to differentiates between them and the states used for the actual behaviour of the program. In order to find the instantaneous loops, `Eclipse` was used to see the marked loops. The reason why `VSCode` could not be used for this task is explained later.

The procedure for finding and eliminating an instantaneous loop is as follows. First, the diagram is used to recognise which variables trigger the instantaneous loop. However, it is difficult to recognise in the `Basic Blocks` view, where exactly the variables are set that lead to the instantaneous loop, which is why, the `Surface / Depth` view is used. This view is chosen because it is the last one in which states occur, and thus the compiler has already done all the conversions so that the chart is written in the core notation. Here, one has to search manually for the respective variable, because there is no tool that would allow one to search in diagrams. The only aspect that speeds up the search is to look in the `Basic Blocks` view to see in which region the loop happens and then finding this region in the diagram. However, this method only slightly speeds up the search for the right variable, especially if it is a large diagram. For this reason, eliminating instantaneous loops was very time-consuming and was therefore no longer done in the `plant2` model.

Another problem that occurred here, which slowed down fixing the loops even more, is that loading the diagram in `Eclipse` is very slow. If the diagram is moved or zoomed in on, it takes a few seconds for the change to actually take place. In addition, a stack overflow frequently occurs with larger programs, where the program prompts the user to close the program. However, this is not necessary and the program can still be used normally after the message was closed.

Another challenge is posed by the simulator, which can be compiled in `KIELER` in three different ways: netlist-based compilation, priotity-based compilation and state-based compilation. In addition, each compilation can be executed in C or Java. In this thesis, it was decided to use the netlist-based approach. This is because it is more powerful than the priority-based approach, and the state-based approach has problems with parallel regions, which are very common in the steam boiler model. In addition, C was used for the actual compilation, as this is slightly faster. However, the system needed to be restarted quite often in order to simulate the system because it otherwise did not find the correct `PATH` variable. This made the simulation very time-consuming, as it took some time to restart the program. In addition, it was not always immediately clear whether the model still had an error or whether the error was triggered by the program.

Despite these performance problems, *Eclipse* had to be used for a large part of the modelling instead of `VSCode`. This st the case because it was otherwise not possible to use signals correctly in dataflow regions. However, as this feature was used in `plant2`, the absence of the feature would mean that the diagram can no longer be displayed in later transformation steps, making it virtually impossible to solve instantaneous loops. *Eclipse* was also used due to the fact that VSCode does not mark the actual instantaneous loops in the diagram. It can still recognise the instantaneous loops and marks the scheduling error with an exclamation mark in the view selection but does not show the loop. For this reason, it is not really possible to find the loops in VSCode and to fix them later.

Chapter 6

Evaluation

In this chapter, the benefits of PASTA for risk analysis on large systems are evaluated and compared with other tools that were already mentioned. In addition, the function of SCCharts for modelling the steam boiler is evaluated and the steam boiler modelled in this thesis is compared with the steam boilers already available in SCCharts.

6.1 Analysis of PASTA

This section looks at the advantages and disadvantages of the PASTA tool and evaluates to what extent it is suitable for carrying out a STPA on a larger system such as the steam boiler and where there might still be potential for improvement.

6.1.1 User guidance

The user already needs a good knowledge of STPA, in order to use PASTA because the tool does not provide any guidance to explain the process. However, each step of STPA can be done with the help of PASTA and it also provides the help of context tables. PASTA does not currently have any documentation, so the notation of STPA elements must be learnt from examples. Such an example can be found in the `Details` page of the VSCode extension. Here, it is important to ensure that the example is always kept up to date, allowing the user to stay informed of all the features of PASTA. For example, the subhazards and subsystems are currently missing from the control structure and it is therefore unclear for a new user how to write them down or if this is even possible.

Furthermore, it is nowhere mentioned that snippets can be used. Snippets are a unique selling point of PASTA, as only PASTA uses such a system. They consist of control structure components and their control and feedback actions and are designed to help users create a control structure more quickly. For this reason, there are already ready-made snippets that map parts of the control structure, which occur in many systems, such as the loop: *controller -> actuator -> physical unit -> sensor -> controller*. These snippets are displayed graphically and can then simply be selected, whereby they are inserted at the end of the already existing control structure. They can also be customized at this place in the code. In addition, own snippets can be created by selecting the desired part of the control structure and then right-clicking on the command `add diagram snippet`. In this way, snippets can then be used again in new control structures. As a result, snippets not only speed up the creation of control structures, but also show beginners established substructures from which they can build their own control structure.

Another benefit of PASTA is that compared to other tools, the creation of new losses, hazards, etc. and their connections within the analysis with other elements is much faster. One of the tools that has been inspected in Section 3.1 is the STAMP Workbench. Here, a new loss is first added using the *Add Accident* button. Then a new hazard can be added with *Add Hazard* or an already written hazard can be selected for this loss with *Select Hazard*. However, if the written loss is then to be assigned a hazard that already exists, *Add Hazard* must first be pressed and then *Select Hazard* to choose an exsting hazard. This system can lead to a confusing appearance, as hazards are listed again after each loss associated with them, so that they are often duplicates, which are furthermore not automatically sorted by ID. A large table is quickly created, as the same principle is also applied to the system-level constraints in which the analyst has. If this process is now applied to the steam boiler, as was done in Section 4.1 in PASTA this would result in a table of the size twenty times six, whereas in PASTA it is fifteen lines of code. Similar problems occur with other tested STPA tools. With STPA Capella, the correct element such as *Losses* must first be right-clicked, then the correct Capella element selected and described in individual input fields. After that must the traceability be added again individually in a separate window. In SafetyHAT, all parts of the analysis must also be clicked through individually and added with many button clicks. These many buttons may help beginners who do not usually write much code or similar structural text, but it is much more time-consuming than simply typing. This is the case, because there the user needs to search for the right button, window, or key combination to add a particular element. The only real advantage of this system is that not only the IDs but also the entire descriptions of the individual elements are displayed, which eliminates the need for looking up IDs.

A similar feature also exists in PASTA, by choosing the *Automatic* option for displaying descriptions in the traceability diagram. When this option is activated, the diagram shows the descriptions of the STPA elements that are connected to the elements that are currently written. For example, when a new hazard should be defined the description of all the losses is shown in the traceability diagram. In this way, all descriptions of the elements that are currently being used are displayed and no IDs need to be memorised. However, it should be noted at this point that the feature is not yet working properly, and therefore no descriptions are displayed at all. However, if this bug is fixed, it would be a valid alternative for the selection options in other STPA tools. Furthermore, a feature could be added that allows parts of the analysis to be collapsed. This would work in the same way as with functions or classes in other programming languages, where only the name with the associated input is visible, while the rest is collapsed. In PASTA, only the heading such as *ControlStructure* could remain. In this way, it would be possible to view system-level constraints and responsibilities at the same time, which would be advantageous as they are linked to each other.

Another feature that should be added to help the user, is that if an STPA element is changed, all other elements to which it is linked should be marked. At the moment, either nothing happens or the link is deleted if the associated ID has not existed for a while. Here it would be useful to mark all elements that are linked to the changed element, as it is possible, for example, that the IDs have changed, which means that the links now lead to the wrong element. This would speed up the analysis as the changes cannot be overlooked so quickly. It would also be practical if there was a way for the analyser to automatically remove all the markers. This would be useful if, for example, only a spelling mistake has been corrected and for this reason, no other elements need to be adjusted. For example, this could be done, by clicking on a marked element and selecting `remove all markers` or by clicking on the changed element and selecting this option.

A further feature, that would help a new user get started, is the addition of stakeholders. Currently, PASTA supports no form of defining stakeholders, which can be used to identify losses. Here, a feature as described in Section 3.1.2 can be added. In PASTA, such an implementation would look somewhat different, as the stakeholders and their values are not recorded in a table, but in textual form. Here, for example, *values* could be recorded in the same form as *losses* and then linked to the respective *stakeholders*. This could also be shown in an additional diagram, with the stakeholders at the top and all the values they have below. If desired, the respective losses that were identified by the values could also be linked to the values so that these are then displayed below them.

6.1.2 Visualization

The visualisation was identified as an advantage of PASTA compared to other STPA tool, it will be examined in more detail in this subsection. One big advantage of PASTA visualisation process is that it happens automatically. This means that in PASTA the diagram is described textually and the layout of the individual components with the respective control and feedback actions is done by the program. As already mentioned in Section 3.1, STAMP Workbench and STPA Capella also have this feature. However, the automatic layout is not as advanced as in PASTA. Furthermore, in STAMP Workbench, a layout can be created manually for the diagram by adding individual components using drag-and-drop, which are then connected with arrows where actions are added. Moreover, SafetyHAT has no option to create a diagram itself so that only a file can be uploaded. The advantage of PASTA's approach is that changes can be made quickly to the diagram without having to manually adjust the entire layout. This is therefore very well suited to the iterative process of STPA. In addition, PASTA is the only program that displays a traceability diagram.

Another feature of PASTA that greatly helps with the overview in the diagram are the many different filters that PASTA has to offer. For example, it is possible to select which diagram is to be displayed, whether the process variables are to be shown in the control structure and which STPA elements are to be displayed. It is also possible to select only a specific control action so that all elements related to it are displayed. The filters make it easier to work with the diagrams and also ensure that they can be created more quickly. This is relevant because a challenge when creating the diagrams is that as the size of the system to be analysed increases, the program takes increasingly longer to do its work. With the model from this thesis, it can take minutes for the diagram showing the traceability to load. This only gets worse if the control structure is to be displayed at the same time.

For the control structure, a filter could also be added that allows subcomponents to be collapsed so that only the hierarchically higher components are displayed. This could create more clarity in systems with many subcomponents and perhaps also reduce the size of the structure so that it can be better shown to others.

In addition to the filters, there are also a few selection options that change the appearance of the diagram. For example, it is possible to select a colour style and decide, which element's description and not just the ID is displayed. In the last case, it would be a good addition if this selection was not just a drop-down menu where either one element or all elements can be selected, but if a checkbox is clicked, the label for an ID is to be displayed, as with the filters. Furthermore, it is possible to define a hierarchy level for each component of the control structure, which then can be activated. However, this hierarchy does not really work at the moment because the `OutsideController` has a hierarchy level of one while the `Operator` has a hierarchy level of zero, but the controller is shown higher in the control structure than the operator. It makes here no difference if the `Hierarchy` is enabled or not. So it would be nice if this feature would be functioning correctly as it would contribute to the clarity of the diagram.

The last feature, which helped a lot with the orientation in the code, is that parts of the diagrams can be clicked to get to the respective place in the code. So it is possible to click on either a controller or a respective STPA element so that the code jumps to exactly the place where the respective element is defined. This is particularly useful for larger systems, as it allows the program to be controlled quickly if changes or additions need to be made. In addition, the click on a specific element also highlights it and all other elements that are connected with it in the traceability diagram. Furthermore, clicking on another element results in also highlighting all of its connection additionally to the first element clicked. This feature creates a lot of clarity, especially with large diagrams, when only the connection of one element in particular is to be viewed or a group of elements. With this feature, it would also be desirable to add the possibility that clicking on the element again demarks it with all its connections. At the moment, it is only possible to deselect all elements, so that if the wrong element is clicked once, all elements have to be selected once again. Here, it could also be added that there exist an option that the description of only the highlighted elements are displayed while the rest is shown with their IDs.

6.1.3 Language Server Updates

The language server has two main tasks, that are relevant for the evaluation. On the one hand, it checks the code and marks if elements of the analyses or references are still missing or if the code has an error. Such an error can for example occur, when a process variable is used that is from a different controller or which has other values. The language server helps here to easily find these errors. Furthermore, the marking of elements if they are missing a reference is really helpful, to keep an overview of the elements for which parts of the analysis are still missing.

On the other hand, the language server is tasked with updating the IDs. This can be really helpful because elements can easily be added or deleted at any point without manually changing any of the IDs. However, one of the biggest challenges discussed in Section 4.2 were caused by wrong updates from the language server, which would mostly lead to incorrect IDs. Furthermore, this updates would increasingly take more time with the increasing number of UCAs and loss scenarios.

Nevertheless, many problems were already solved. As a result, the control actions of the subhazards can now also be selected in the context table, their control actions are getting marked, when still some of the analysis steps of a control action is missing, and the traceability diagram shows the connection always correctly.

6.1.4 Context Table

The context tables are a valuable addition to PASTA, as they offer a comprehensive overview of all possible contexts that can lead to UCAs. This systematic approach minimizes the likelihood of overlooking any cases. An improvement is that all different types of UCAs are displayed in a single table, saving time on switching and requiring the contexts to be considered only once for all types. However, the procedure used to create the context tables in PASTA is still very basic. The fact that the table is created by brute force and can only ever be displayed in full for one control action makes its creation very slow. For this reason, tables with more than sixteen variables can no longer be displayed, which severely limits the usefulness of the function for larger systems. It would then be necessary to abstract and analyse the system at a high-level first and perhaps take a closer look at individual parts later. It would also help if PASTA used logical simplification and thus combined rows. This would not only speed up the creation of the program's table, but also the analysis, as it would not require going through contexts that have already been recognised as unsafe.

However, PASTA already uses the rule-based approach, which was introduced in Section 2.3. This works in such a way that the written contexts of the UCAs represent the individual rules of the context tables. A process variable is automatically assigned the value `any` if it is not listed by the respective controller. This incorporation significantly reduces the effort and working time by partly automating the generation process.

In Section 4.1.3 it was noted that it was not possible to use a `not` in the context table, which would have made it easier to analyse the *setThrougput* control action. This is because the water level is only calculated in rescue mode and otherwise measured, so a `not` could have combined many UCA, as not all modes have to be listed individually, but simply `not rescue` can be written. However, this is not a feature in PASTA, as it does not occur in rules defined by Gurgel et al. [GHD15]. The reason for this is that a `not` could lead inexperienced analysts in particular to combine UCAs that actually lead to different hazards or contain a different scenario. For example, when analysing the steam boiler, there were UCA that I initially thought should be grouped together, but on closer inspection it turned out that they were actually different. For this reason, I would not include a `not` in PASTA at first, even if it could be useful in rare cases. And if this feature should be added, it should be pointed out that UCAs might be different after all and that the `not` should therefore be used with extreme caution.

In Section 2.3, it was also mentioned that context tables can also be used to identify loss scenarios. With PASTA, it was possible to use the context tables as a guide and to get an overview of the respective contexts in which the loss scenario occurs. It can also help to recognise process flaws. However, there is no automatic traceability that links the scenario with the respective UCA and thus the context. This would save the analyst a lot of time, as the context would not have to be written down again manually for each scenario, although it would already be known from the context table, which is redundant.

Other challenges and possible solution were already mentioned in Section 4.2.2. However, since PASTA is a tool that is currently under active development, the problem that only control actions that were not nested could be selected in the context table has already been fixed. This is why in the current version also control actions from the `Pump Controller` can be selected for the context table.

6.1.5 Loss Scenarios

In Section 4.2.3 were also challenges for the loss scenarios identified. One of these was that a lot of text needed to be written for the scenarios. Here, a feature can be added to link specific causes for loss scenarios, which often happen, such as incorrect beliefs. In Section 4.1.4, for example, was mentioned that a lot of the wrong beliefs were caused by failures or incorrect values for sensor reading. It is possible to add a new category, which is used to note subcauses for loss scenarios. These would work similar to the snippets that one writes down causes for loss scenarios and these can then marked and added to this section. However, contrary to the snippets this would not be an extra window in VSCode but only a textual block above the loss scenarios. Causes then also can be added directly in this block by just writing them in this part and adding an ID for later referencing. It is then possible to reference these subcauses in the loss scenarios, where they get documented automatically in the string. This would save a lot of time and repetitive work. For example, causes for the wrong belief of the water level are the following:

- the water level sensor failed
- the response from the water level sensor was delayed
- due to wear over the time, has the sensor a drift in its data, leading to a wrong feedback
- depending on the specific sensor used, there could be different inferences with the signal (temperature changes, water disturbances, impurities in water, scale)
- error in the controllers' software, leading to a wrong interpretation of the water level

This could look exemplarily like in Listing 6.1. The `SCA1` would then be replaced by the text in the `Subcasuses` either when the file is saved the next time or when the ID is clicked and the option `Write Subcasuses` is chosen.

```
Subcasuses
SCA1 "- the water level sensor failed
    - the response from the water level sensor was delayed
    - due to wear over the time, has the sensor a drift in its data, leading to a wrong
      feedback
    - depending on the specific sensor used, there could be different inferences with the
      signal (temperature changes, water disturbances, impurities in water, scale)
    - error in the controllers' software, leading to a wrong interpretation of the water level"

LossScenarios
Scenario1 for UCA1
"The mode is initialization and the water level is below normal. But the Controller believes
 that it is normal, so that it sends the control action progReady. This could occur, when:
 SCA1" [H1, H3, H4, H5]
```

Listing 6.1. Feature proposal for subcauses in PASTA.

It would also be good to have a feature that makes it possible to mark certain parts of a string differently. At the moment, the individual scenarios are very similar and long at the same time, which makes it difficult for the analyst to immediately differentiate between the individual scenarios. If the differences could be marked, it would be easier to recognised them quickly when reading through the loss scenarios at a later time.

6.2 Steam-boiler SCChart Model

This section looks at the advantages and disadvantages of SCCharts in the KIELER tool and evaluates to what extent it is suitable for designing the steam boiler with a focus on verification and where there might still be potential for improvement.

6.2.1 User Guidance

SCCharts have a number of features that make them well-suited for modelling the steam boiler. Firstly, they are deterministic, which is particularly important in a critical system, as it must always be clear how the system behaves in a given situation. It is also particularly useful when working with many signals that have an effect on the system. This is because it clearly determines the order in which transitions are taken.

Furthermore, SCCharts are particularly suitable for systems that work with different states, as they are state-oriented. With the steam boiler, it was therefore very intuitive to convert the specification into a model with states, as the modes already describe different states of the system.

SCCharts also offer many different features that are useful for modelling. For example, regions are used to describe program sequences that run in parallel. Some superstates were also used, as certain processes often have to take place in a state. For example, the initialisation state in the `program2` has inner states, which ensure that the initialisation works correctly. There is a similar situation with the pumps, as they also follow a certain process to open and close, but this can only happen if the pump is not in the defect state. This could also be modelled differently with a flat hierarchy, but the hierarchical model is very intuitive. The use of references is also very practical when modelling a larger system. In this way, individual modules can be designed and then put together afterwards so that you have a modular structure.

There are many other features that can help with modelling that have not been mentioned here. However, this also highlights a challenge with SCCharts. There are many features that a user will not use or understand, especially at the beginning. One reason for this is the lack of a complete documentation and available documentation, which is only updated sporadically. Another reason is the sheer number of features in SCCharts which initially makes it unclear where they could be used best. For this reason, users often stick with the same features and model everything with them, even though another feature might be more suitable. Here it would be useful if the documentation was always kept up to date and if there was an example of how each feature is written in SCCharts and where typical use cases are.

Since the model checker was also used in this work, some ideas have arisen as to which features it could still be missing. The most important feature to add would be a correctly functioning button that provides a counterexample. At the moment, it is not always clear why an LTL formula has failed. For this reason, it would be practical to start a simulation that shows the user a counterexample where the LTL formula fails. This would save a lot of time, as it would not be necessary to think of an example manually, since the model checker would simply display the example that it has already found. Another good enhancement would be the addition of error messages when the model checker has an exception. These can be triggered by various circumstances, such as an incorrect LTL formula or the absence of an `@AssumeRange` before an integer variable. At the moment, however, there is no feedback indicating where the possible exception could come from. For this reason, it can take a long time before the cause can be found and rectified.

6.2.2 Visualization

An advantage of SCCharts is the variety of views of the model, as well as filter options. After the model has been compiled, it can be viewed in the KIELER view for each compilation step. This makes it easy to see exactly what the compiler is doing and at which step errors may still occur in the program. Due to the many filters and display options, it is possible to view exactly those parts of the diagram in the form that is currently required. For example, declarations can be omitted, the length of the labels of a transition can be adjusted and the progression of the diagram can be swapped from top to bottom to left to right. In this way, the diagram can display exactly what the user wants, which also facilitates the presentation of a system.

Another feature that helps with the overview is that the diagram automatically adjusts how much of it is visible depending on the zoom level. This means that if a larger system is modelled, which has a hierarchical structure, only the names of the superstates are displayed when zoomed out, but no longer their inner regions, because these then become too small. This function ensures that even large systems remain readable and that the rough relationships between the states can be quickly recognised.

The only problem with the diagrams is that they sometimes take a long time to load. This phenomenon is particularly noticeable in the *Eclipse* version of KIELER, where the using of the diagrams also frequently leads to stack overflows. If a diagram is to be enlarged or moved, this action is usually delayed by several seconds, resulting in a lag in the diagram. This limits the usefulness of the diagram, as it is much more time-consuming to work with it.

One feature that could be added is that clicking an element in the diagram leads to jumping to the position of the element in the code. This could work similarly to PASTA. For example, clicking on a transition would take the user to that transition in the code, while clicking on a state or region would take the user to the start of that element. This would make it easier to navigate through the code, as the diagram is often used for modelling, and it would not be necessary to manually search for the point in the code that needs to be updated.

Another feature that already exists in PASTA and would also be well suited for SCCharts is the ability to click on a signal in the dataflow region and then see its connections highlighted. This would make it easier for the user to recognise how certain components are connected, which can quickly become confusing, especially in large diagrams with many connections.

Another feature that would save a lot of time would be the addition of a search function in the diagrams. This would allow the diagram to be searched for a region, state, variable or other elements. It would also be good to have a setting that allows automatic zoom, as otherwise the element will be found, but the user will not be able to see it because it is too small at the current zoom level. It would also be good if there was an option that highlighted all the elements searched for and not just one.

6.2.3 Debugging

The debugging of large systems has already been recognised in other reports as one of the biggest weaknesses of SCCharts [SMS+19]. This fact has not changed since the report was written. In this work, the system was mainly debugged to find and repair instantaneous loops. This was a particular challenge because the variables were created during compilation and therefore had to be found in the chart. Here it would be good if the feature mentioned in Section 6.2.2 was added, because it would speed up the search for the variables in the diagram considerably.

It would also be good if the VSCode version added the feature that instantaneous loops can also be displayed.

Another challenge while debugging in SCCharts is that it is very difficult to recognise the actual error from the error messages alone. In the case of instantaneous loops, these are schedule errors in which the basic blocks are specified between which no schedule can be found. However, this error message alone is of little use if the basic blocks cannot be displayed or if these errors are not marked there. Other error messages are also of little help to the user, as these contain only the stacktrace of the compiler leading to the error, but as the user does not know the internal code of the compiler, it is often difficult to understand the actual error from the message and then correct it. Here it would be good if the error messages were adapted so that they can also be understood by a user who has no insights into the development of KIELER.

6.2.4 Comparison to other Models

The steam boiler has already been modelled twice in SCCharts, but the focus of those models differed from that of this thesis. The first being a user study on manual user verification of different source codes, which were generated for a steam boiler, while the second focussed on object-orientated approaches to model the steam boiler to test the OO features of SCCharts. In order to better differentiate between the models, the following section looks at the differences and similarities between them.

The Steam Boiler by Smyth

The biggest differences between the steam boiler that was modelled in this thesis and the model from Smyth et al. [SDH19] is that Smyths model omits all transmission failure related protocols and shortens the unit fail protocol. The model assumes that all transmission are reliable and correct so that no retransmits and timeouts are needed. Furthermore, the model has no checks if a signal is sent at the correct time or is a transmission failure. The model simplifies the protocol for component failures. In both the specification and the steam boiler model from this paper, the program detects that a component has failed. It then sends a failure message to the affected component that must acknowledge this signal. After the component has been repaired at a later time, it sends a repair signal to the program, which the program must acknowledge. In the case of Smyths model, a failure is triggered by the user clicking on a component in the visualisation of the steam boiler. The repairing of the signal is then done in the same way. Both the failure and the repair message expect no acknowledgement from either the physical unit or the program.

A major difference between Smyths model and the specification is that the heater was also modelled in this version. This means that there are different modes that the heater can adopt, in which it emits different amounts of heat. If the steam boiler is in initialisation or emergency stop mode, the heater is off and cools down slowly, in normal mode the heater heats up quickly to a higher temperature and in degraded or rescue mode the heater heats up more slowly to just half the maximum temperature that it can have in normal mode. The water level is then also calculated with the heat of the heater. However, the steam level is never changed and just stays at zero for the whole time the system is running. Furthermore, the model has no valve module, but the valve is part of the environment as it can only open to release water, but has no functionality that checks if it is only used in initialisation mode or if there is a transmission failure.

The model has some other features where it differs from the specification, it has four different pumps but they can only be activated or deactivated all at once. This means that their functionality is more similar to the model in this thesis where only one pump is used, with the only difference being that different pumps can break. However, this breakage has no influence on the functionality of the pumps in Smyth model. Furthermore, the model uses only booleans and no signals for its variables.

The SCChart, which is used to change modes, is very similar in both models. The only differences are that in Smyths model the manual `STOP` only needs to be triggered once that in rescue mode the failure of the steam level sensor or the pump controller does not switch the system into emergency stop mode, and, as already mentioned, there are no transmission failures. In addition, in Smyths model the program takes over the activation of the pumps, as is also defined in the specification. In the model in this thesis, this was outsourced to the *Pump Monitor* due to the modularisation of the system.

Another difference is that Smyths model does not recognise whether a unit has a defect by the values of the physical units, but a defect is triggered from outside by clicking on a component. The model has also been visualised so that it shows the current status of the steam boiler and the user can interact with it.

Another difference is that all physical units of Smyths model use object-orientation to inherit from a *physicalDevice* class. This provides a region that takes care of booting and processing the defect of the unit. In the model from this thesis, different SCCharts are also used in several other modules, but this is done by referencing the SCChart and not by inheritance.

Furthermore, it would not be possible to verify the model because it uses floats for the water and steam level variables as well as for the throughput of each pump and heat value. However, the state explosion problem would not be such a big problem, but this is rather due to the fact that Smyths model was abstracted so heavily that it does not have all functions that were described in the specification and thus the number of states is kept lower.

The two models are very similar in other respects. They both use an environment and calculate their system in a dataflow region. In addition, physical units such as the pump or the sensors are modelled in the same way, apart from the differences already mentioned.

The Steam Boiler by Schulz-Rosengarten

The biggest difference in Schulz-Rosengartens steam boiler is the use of object-orientated methods in his steam boiler model. He therefore defined interfaces for each physical unit, that are extended by the module. Furthermore, the *controller*, which operates the whole system, inherits from all physical units and the *MonitoredPumpControl* inherits from all other pump related modules. Additionally, abstract classes were defined for all different kinds of transmission failures. There exist classes for a missing signal, an unexpected signal and a transmission timeout. These are then used for every signal that can have a transmission failure. Furthermore, it exits an abstract class for the water level, as it can be either measured or calculated.

Schulz-Rosengarten added two new modules for regulating the pumps and their controllers, the *MonitoredPump* module and the *MonitoredPumpControl*. These are tasked with activating the pumps, getting their throughputs and monitoring their failures. This model also has four different pumps and controllers that can be activated independently. This is the reason why it is closer to the specification than the model in this thesis in that regard. However, the pumps do not need five seconds to open, which then differs from the specification and my model.

The `SCChart`, which is used to change modes, is very similar in both models. The only differences are that in Schulz-Rosengartens model the transmission failures are modelled by using the abstract classes, the management of the water level is part of the `Controller` In addition, in Schulz-Rosengartens model the program takes over the activation of the pumps, as is also defined in the specification. In the model in this thesis, this was outsourced to the *Pump Monitor* due to the modularisation of the system.

Another difference between the models is that only my model has an environment. That means that the model from Schulz-Rosengarten needs to be updated manually. There is no module that calculates the water or steam level, and the `PHYSICAL_UNITS_READY` signal needs to be sent from the outside. This means that the units have no booting time and the `PROGRAM_READY` signal is not checked for transmission failures. As a result, the user has a lot more work to simulate the steam boiler, because they have to make sure that the entered values match the behaviour of the steam boiler so that no defects occur. In addition, the user must also signal that units have been repaired.

It would also not be possible to verify the model because it also uses floats for the water and steam level variables as well as for the transmission timeout. Additionally, the state explosion problem would be a challenge in verifying the model because the model uses many states due to the object-orientated approach. As a result, the verification process would take a long time and would probably not be feasible.

The unit fail protocol is in both models almost the same. The only difference being that in the model from Schulz-Rosengarten the inner regions are dataflow region that are using the abstract transmission failures classes to define a failure. Furthermore, the sensor modules are also very similar, as they detect a failure and have the unit fail protocol in a dedicated region.

Chapter 7

Conclusion

This chapter serves as a summary for the contributions of this thesis. Section 7.1 includes the STPA of the steam boiler and the design of steam boiler model focused on verification. Furthermore, it provides an overview of the evaluation of the applicability and limitations of the PASTA tool and SCCharts for this process. Additionally, Section 7.2 explores potential future work to expand the analysis of the tools as well as the refinement and visualization of the model.

7.1 Summary

In this thesis, a full STPA of the steam boiler was conducted in PASTA. It was shown that many different attributes and components lead to UCAs and that the analysis of the system already has a large size so that it needed to be abstracted and cut in some parts.

In the second part, the steam boiler was modelled using SCCharts with the focus on verification. For this reason, approaches were identified to enhance the model's verifiability. These approaches being: limiting the range of values and the amount of states, modularising the system for partial verification. It was then shown on the valve module that a verification with LTL formulas is possible.

The evaluation of the tools revealed that both can be used to either analyse or model larger systems such as the steam boiler. However, this process still contains some challenges that must first be overcome in order to obtain a functioning product. In addition, features have been identified that support the user, and new features have been proposed to improve visualisation and the user experience. Some of this being the addition of stakeholders, the logical simplification in context tables, or the marking of changes from connection in PASTA, or the addition of a search function in diagrams in SCCharts.

In conclusion, both tools show much potential to be used on larger systems. Nonetheless, still some features exist that need to be worked on, in order to have a good user experience.

7.2 Future Work

There are multiple functions of PASTA as well as KIELER that still need evaluation, which were not tested in this thesis. Furthermore, the model of the steam boiler must be modified so that it is compileable.

7.2.1 Evaluation of PASTA's LTL Formulas

PASTA has multiple features that were not used in this work. One of them is the ability to automatically generate LTL formulas for a written analysis. These could then be used for the verification of a model of the analysed system. In the future, LTL formulas could be written for all modules of the steam boiler model and then compared to the formulas created by PASTA. This could be used to evaluate how well the automatic creation of the LTL formulas works and where work might be still needed.

7.2.2 Steam Boiler Model

The steam boiler model presented in this work is only functioning in its single components. These can be compiled and simulated. However, the complete model represented by `program2` still has many instantaneous loops and can therefore not be compiled. In the future, these loops need to be fixed in order to get a working model.

Furthermore, the model can be visualised to help show the functioning of the steam boiler. This can be done with `kviz` as it was already shown in the steam boiler model by Smyth et al. [SDH19]. In order to animate the Scalable Vector Graphics (SVG) of the steam boiler shown in Figure 7.1, a `kviz` file needs to be written, which specifies the changes in the SVG if a specific signal occurs or a value changes. For example, a physical unit turns red, when it is defect, the water level changes, the valve opens and closes, or the rotors are turning when the pumps are open. Moreover, a display needs to be added to show, which mode the system is in and what values the sensors are measuring. In Listing 7.1 is shown how the turning red of a physical unit can be displayed when it is defective. The example chosen here, is the defect of the pump. The `image` tag references the correct SVG, with the keyword `handle` is the signal defined that changes the appearance of the steam boiler. `pump` is the ID of the SVG element that changes its appearance and after that follows the javascript line, which changes the colour either to red or black, depending on the state of the pump.

```
image "elevator.svg"

handle failMsgPump
in "#pump"
with (elem, status)
'elem.style.fill = status ? "red";'

handle failMsgPump
in "#pump"
with (elem, status)
'elem.style.fill = status ? "black";'
```

Listing 7.1. Unit defect in kviz.

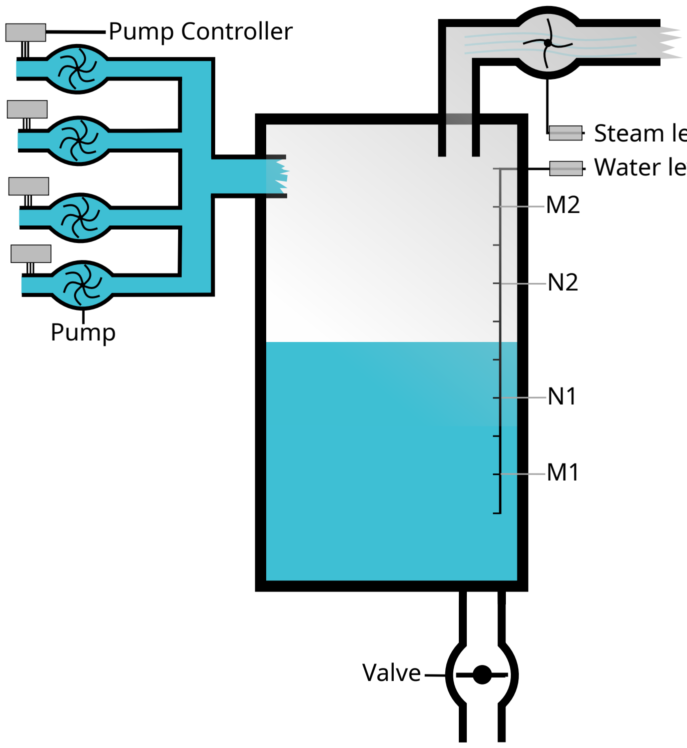


Figure 7.1. Physical units of the steam boiler [SDH19].

7.2.3 Evaluation of `KIELER`'s Model Checker

Future work would also include, to verify the steam boiler model with the LTL formulas defined in Section 7.2.1. For this, the model needs to be updated so that it works and is also suitable for the verification process. This means, for example, that all integer values must have an `@AssumeRange` so that the model checker knows what range it needs to test for the variables. After the model is verified, the model checker can also be evaluated. The model checker was developed by Stange in his master thesis [Sta19], where it was evaluated on smaller models. However, its performance on larger systems has not yet been tested. Thus, the steam boiler model presents a valuable opportunity to assess its effectiveness on a more complex system.

Bibliography

[Abr05] Jean-Raymond Abrial. “Steam-boiler control specification problem”. In: *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control* (2005), pp. 500–509.

[And96] Charles André. “Synccharts: a visual representation of reactive behaviors”. In: *I3S, Sophia-Antipolis, France, Tech. Rep. RR* (1996), pp. 95–52.

[AR10] Terje Aven and Ortwin Renn. *Risk management and governance: concepts, guidelines and applications*. Risk, Governance and Society. Springer Berlin, Heidelberg, 2010. ISBN: 9783642139260. DOI: https://doi.org/10.1007/978-3-642-13926-0.

[BK08] Christal Baier and Joost P. Katoen. *Principles of model checking*. English. United States: MIT Press, May 2008. ISBN: 978-0-262-02649-9.

[BV+14] Christopher Becker, Qi Van Eikema Hommes, et al. *Transportation systems safety hazard analysis tool (safetyhat) user guide (version 1.0)*. Tech. rep. John A. Volpe National Transportation Systems Center (US), 2014.

[BW96] Robert Büssow and Matthias Weber. “A steam-boiler control specification with statecharts and z”. In: *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the Book Grow out of a Dagstuhl Seminar, June 1995)*. Ed. by Jean-Raymond Abrial, Egon Börger, and Hans Langmaack. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 109–128. ISBN: 978-3-540-49566-6. DOI: 10.1007/BFb0027233.

[CKN+12] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. “Model checking and the state explosion problem”. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1. URL: https://doi.org/10.1007/978-3-642-35746-6_1.

[Con24] Oliver Constant. *Github*. 2024. URL: https://github.com/labs4capella/stpa-capella.

[CRD+22] *ERTS 2022 proceedings*. June 2022, pp. 95–100, 191–200. URL: https://hal.science/hal-03704287.

[Dom18] Sören Domrös. “Moving model-driven engineering from eclipse to web technologies”. In: (Nov. 2018). URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf..

[dT06] Claudio de la Riva and Javier Tuya. “Automatic generation of assumptions for modular verification of software specifications”. In: *Journal of Systems and Software* 79.9 (2006). Selected papers from the fourth Source Code Analysis and Manipulation (SCAM 2004) Workshop, pp. 1324–1340. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2005.11.570. URL: https://www.sciencedirect.com/science/article/pii/S0164121205001664.

[GHD15] Danilo Lopes Gurgel, Celso Massaki Hirata, and Juliana De M. Bezerra. “A rule-based approach for safety analysis using stamp/stpa”. In: *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*. 2015, 7B2-1-7B2–8. DOI: 10.1109/DASC.2015.7311464.

[Hal92] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Vol. 215. Springer Science & Business Media, 1992.

[Har87] David Harel. “Statecharts: a visual formalism for complex systems”. In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. “Sccharts: sequentially constructive statecharts for safety-critical applications: hw/sw-synthesis for a conservative extension of synchronous statecharts”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 372–383. ISBN: 9781450327848. DOI: 10.1145/2594291.2594310. URL: https://doi.org/10.1145/2594291.2594310.

[II05] Clifton A. Ericson II. “Event tree analysis”. In: *Hazard Analysis Techniques for System Safety*. John Wiley & Sons, Ltd, 2005. Chap. 12, pp. 223–234. ISBN: 9780471739425. DOI: https://doi.org/10.1002/0471739421.ch12. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471739421.ch12. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471739421.ch12.

[Kle99] Trevor Kletz. *Hazop and hazan - identifying and assessing process industry hazards*. Fourth. Institution of ChemicalEngineers, 1999.

[KRD+24] Maximilian Kasperowski, Niklas Rentz, Sören Domrös, and Reinhard von Hanxleden. “KIELER: a text-first framework for automatic diagramming of complex systems”. In: *Diagrammatic Representation and Inference, 14th International Conference, DIAGRAMS ’24*. To be published, 2024.

[Lev18] Leveson, Nancy G. and Thomas, John P. *STPA handbook*. MIT Partnership for Systems Approaches to Safety and Security (PSASS). Cambridge, Massachusetts, U.S., 2018.

[OR18] Marcos Lucas de Oliveira and Janis Elisa Ruppenthal. "Using the hazop procedure to assess a steam boiler safety system at a university hospital located in brazil". In: *Revista Gestão da Produção Operações e Sistemas* 13 (Sept. 2018), pp. 259–275. DOI: 10.15675/gepros.v13i3.1959.

[PH23] Jette Petzold and Reinhard von Hanxleden. "Tool Support for System-Theoretic Process Analysis". In: *Electronic Communications of the EASST* 82 (2023).

[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)*. 2023, pp. 559–567. DOI: 10.1109/DSN58367.2023.00058.

[PTL13] Yuliya Prokhorova, Elena Troubitsyna, and Linas Laibinis. "A case study in refinement-based modelling of a resilient control system". In: *Software Engineering for Resilient Systems: 5th International Workshop, SERENE 2013, Kiev, Ukraine, October 3-4, 2013. Proceedings 5*. Springer. 2013, pp. 79–93.

[Ren18] Niklas Rentz. "Moving transient views from eclipse to web technologies". MA thesis. Kiel University, Department of Computer Science, Nov. 2018. URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf.

[RH] Ratih Andhika Akbar Rahma and Achmad Hasanudin. "Risk assessment analysis in boiler system with hazard and operability study (hazop)". In: *Jurnal Ilmiah Teknik Industri* 22.2 (), pp. 213–220.

[Rob93] Michael Rossi Robert Borgovini Stephen Pemberton. *Failure mode, effects and criticality analysis (fmeca)*. Reliability Analysis Center, Apr. 1993. URL: https://s3vi.ndc.nasa.gov/ssri-kb/static/resources/a278508.pdf.

[Sch24a] Alexander Schulz-Rosengarten. *Language design for reactive systems — on modal models, time, andobject orientation in lingua franca and sccharts*. Kiel Computer Science Series 2024/1. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, 2024. DOI: 10.21941/kcss/2024/1.

[Sch24b] Alexander Schulz-Rosengarten. *The sccharts syntax*. 2024. URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Syntax.

[SDH19] Steven Smyth, Sören Domrös, and Reinhard von Hanxleden. *A case-study on manual verification of state-based source code generated by kieler sccharts*. Tech. rep. Kiel: Faculty of Engineering, 2019.

[SMS+15] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Wechselberg, Carsten Sprung, Reinhard von Hanxleden, et al. "Sccharts: the railway project report". In: *Bericht Des Instituts Für Informatik* 1510 (2015).

[SMS+19] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Sören Domrös, Lena Grimm, Andreas Stange, and Reinhard von Hanxleden. "Sccharts: the mindstorms report". In: Kiel Computer Science Series (2019). URL: https://nbn-resolving.org/urn:nbn:de:gbv:8:1-zs-00000358-a4.

[Smy21] Steven Smyth. *Interactive model -based compilation— a modeller -driven development approach*. Kiel Computer Science Series 2021/1. Dissertation , Faculty of Engineering ,Kiel University . Department of Computer Science , CAU Kiel, 2021. DOI: 10.21941/kcss/2021/1.

[Sta19] Andreas Achim Stange. *Model checking for sccharts*. Kiel, 2019. URL: https://wiki.rtsys.informatik.uni-kiel.de/bin/view/Theses/Completed%20Theses/.

[Tho13] John P Thomas IV. “Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis”. PhD thesis. Massachusetts Institute of Technology, 2013.

[Ves81] Vesely, W.E.,Goldberg, F.F.,Commission, U.S. Nuclear Regulatory. *Fault Tree Handbook*. NUREG-0492. Systems and Reliability Research, Office of Nuclear Regulatory Research, U.S. Nuclear Regulatory Commission, 1981. URL: https://www.nrc.gov/docs/ML1007/ML100780465.pdf.

List of Abbreviations

ETA	Event Tree Analysis
FMECA	Failure Modes and Effects Criticality Analysis
FTA	Fault Tree Analysis
HAZOP	Hazard and Operability Analysis
LTL	Linear Temporal Logic
PASTA	Pragmatic Automated System-Theoretic Process Analysis
SCChart	Sequentially Constructive Statechart
STPA	System-Theoretic Process Analysis
SVG	Scalable Vector Graphics
STAMP	System Accident Model and Processes
UCA	Unsafe Control Action
SC	Sequentially Constructive
MoC	Model of Computation
IURP	Initialize-Update-Read Protocol
IDE	Integrated Development Environment
VSCode	Visual Studio Code
UI	User Interface
ID	Identifier
IPA	Information-technology Promotion Agency
SafetyHAT	Safety Hazard Analysis Tool
OO	Object-Orientated