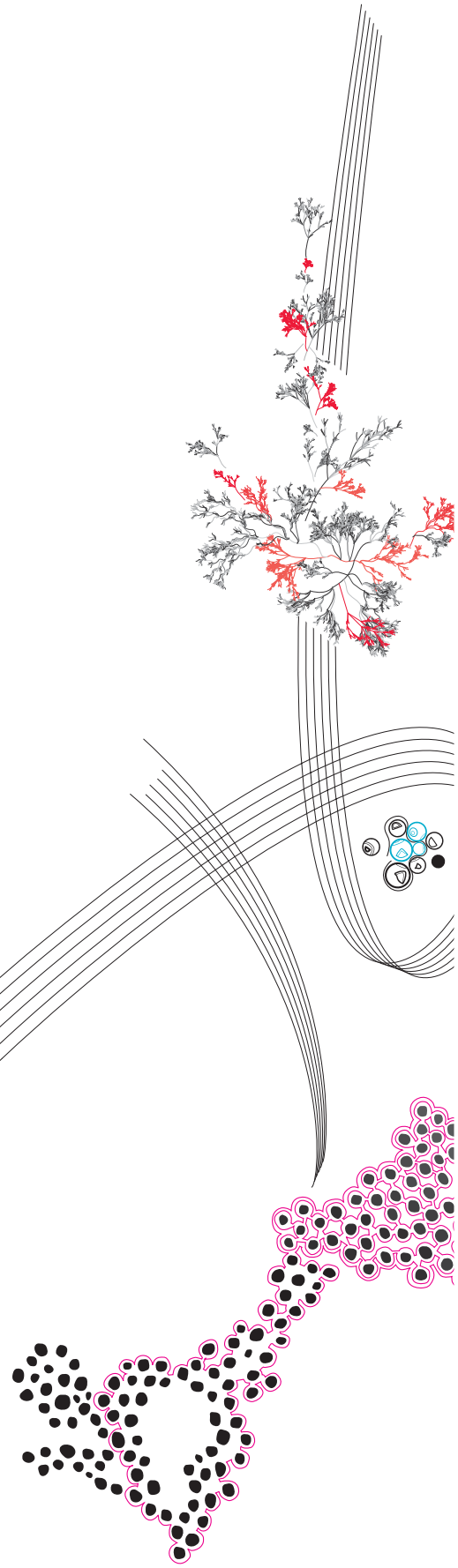


MSc Computer Science
Final Project

Generating Piping & Instrumentation Diagrams from Static Fault Trees

W. F. A. Bos

Supervisors:

Matthias Volk
Mariëlle Stoelinga
Anne Remke
Pavel Krčál
Marc Bouissou

February, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

Static Fault Trees (SFTs) can visualize and quantify risks. One domain which uses SFTs is within thermohydraulic systems of nuclear power plants. Fault Tree Analysis (FTA) is a research field providing diverse insights into these risks. While an SFT focuses on event risk and probability, understanding the event's location within a system requires the system's design and a graphical representation of the power plant.

Piping and Instrumentation Diagrams (P&IDs) serve as graphical representations of industrial plant designs. Existing software can already generate SFTs from P&IDs, including P&IDs of a nuclear power plant's thermohydraulic systems.

However, experts often manually create SFTs, lacking a digital, formal P&ID for the power plant. A formal P&ID offers a different perspective on the plant's risks and events in comparison to existing SFTs. It encompasses process flows, instrumentation, and alarms, providing additional and varied information beyond SFTs.

If an automated process could generate P&IDs from SFTs, these resulting P&ID(s) serve as an initial formal representation. This enhances consistency, reduces design faults, and helps prevent conflicts and hazards.

This Master's Thesis presents an initial algorithm for automatic P&ID generation from SFTs. It defines a P&ID using a graph structure and outlines the algorithm in three steps. First, it determines component labels and types based on a naming convention in the SFT. The second step addresses components related to power plant system maintenance. Finally, the algorithm establishes an initial topology using Post-Order Depth-First traversal of the SFTs. The Python tool implementing this algorithm is the SFT to P&ID Converter (SPC).

The validation process for the SPC involves two inputs: four basic examples and a RiskSpectrum case study. This yields four insights. First, the algorithm serves as an initial formal method for inferring basic P&IDs from SFTs in the nuclear power plant domain. Second, P&IDs defined as graphs provide a way to formally show the relationship between SFTs and P&IDs. Third, it marks a preliminary step in inferring P&IDs from SFTs in other domains. Lastly, current limitations of the approach involve Basic Event ordering, P&ID Configurations, and Scalability.

Keywords: Static Fault Trees, SFTs, Piping and Instrumentation Diagrams, P&IDs, Risk Analysis, Risk Management, Power Plants

Contents

Chapter 1

Introduction

Fault Trees (FTs) [72] are trees which visualize and quantify the risks of various systems and items. It is possible to create FTs of situations existing in trains, coffee machines and many other domains. There are many different types of FTs (e.g. DFTs, SFTs [68, 38, 30, 79]). Furthermore, many FT extensions exist (e.g. Fuzzy FTs, Repairable FTs [74, 61]). In particular, they offer insights into their reliability, availability and safety [46, 68, 27, 28]. Usually, performing Fault Tree analysis (FTA) gains these insights. Many FTA methods exist (e.g. MTBF, MTTF, MTTFF) to analyse the risks described in FTs.

Various industries (e.g. the energy, aerospace & transportation, and chemical & process industries [3, 72, 29, 11]) use models based on Fault Trees (FTs) for safety assessment. Especially FTs for nuclear power plants are notable for their complexity and level of detail. One type of FT is a Static Fault Tree (SFT). I define SFTs, Top Events (TEs), Basic Events (BEs) and their gate types AND, OR and VOT(k) in Section 2.1.

A limitation of SFTs is that they only show information about the risk and probability of an event happening. Another limitation is that they only show the impact of this risk on the whole system. I also want to know where that event happens and what impact that has on the components of the power plant itself. For that, I need a graphical representation of the power plant. This is where Piping and Instrumentation Diagrams (P&IDs [77]) come in.

A P&ID is a graphical representation of industrial plant designs. They represent the process flow, instrumentation and alarms within such a plant. Creating and using P&IDs increases consistency and decreases design errors. It helps to identify and correct issues in the design because they contain a lot of versatile information. An example of this is when a plant consists of many links which can carry different fluids, gases or signals. A P&ID helps to identify areas where those links may lead to potential conflicts or hazards. Typically, reliability analysts build SFT models manually. They transfer the system information present in system design representations together with the failure behaviour analysis into SFTs. Examples of such system design representations are P&IDs and single-line diagrams for electrical systems [48].

Another alternative is so called Model-Based Safety Assessment (MBSA), which:

- Encapsulates the reliability knowledge in a tool,
- Allows analysts to create safety models closely resembling the system models and,
- Automatically generates the SFTs.

This thesis uses an MBSA tool called RiskSpectrum ModelBuilder (RSMB[1]), which is

[1] https://www.riskspectrum.com/riskspectrum-modelbuilder

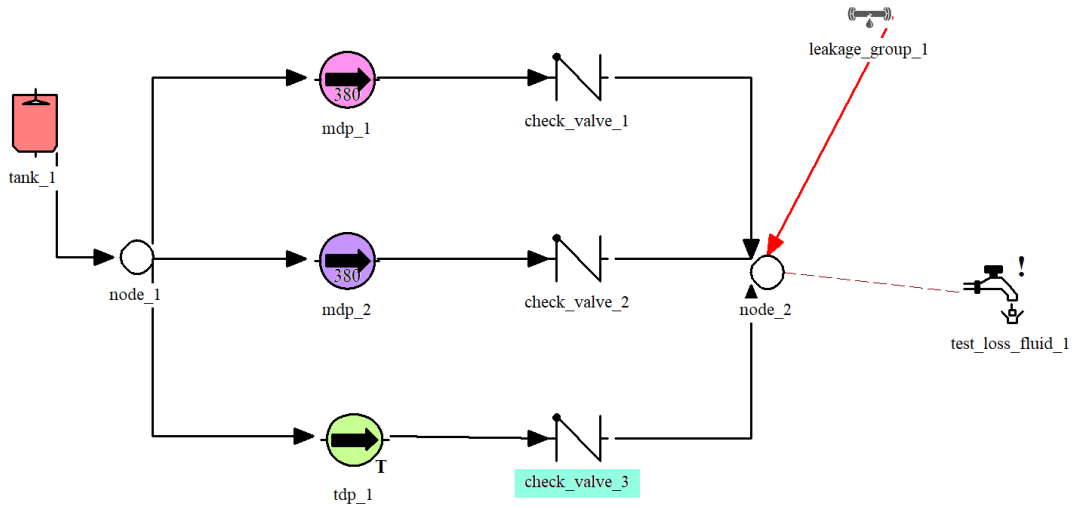


FIGURE 1.1: P&ID of a simple fluid system with three pumps and check valves in parallel.

based on KB3 [8]. Electricité de France (EDF) is developing KB3 since the early 1990's. Nuclear safety is a major concern for EDF, which operates 58 nuclear power plants in France. This is why this company developed an MBSA approach and KB3; speeding up and improving the quality of safety studies. It is also the reason for working towards inferring P&IDs from SFTs in this thesis. RiskSpectrum ModelBuilder commercializes and further develops KB3 in a partnership between EDF and RiskSpectrum AB. RiskSpectrum AB is one of the market leaders with a proven commercial suite of risk software for modelling, quantifying, and monitoring risk and reliability. Of the world's nuclear power plants, 60% uses a licence of the RiskSpectrum software. The company released its first software of this kind in 1986 and has since developed a suite of tools for risk, reliability, and availability assessment.

Subsection 2.3.1 further discusses MBSA, RSMB and KB3.

1.1 Problem example

Figures 1.1 and 1.3 show an example of an SFT generated from a P&ID. The P&ID in Figure 1.1 shows a part of a plant. The plant contains:

- A tank;
- Two motor-driven pumps ("mdp_1" and "mdp_2")
- One turbine-driven pump ("tdp_1")
- Three check valves ("check_valve_1", "check_valve_2" and "check_valve_3")

The P&ID also has two nodes ("node_1" and "node_2"). After the first node, the fluid flows in parallel over the three pumps and check valves. The second node brings these fluid streams back together. For this plant, I want to visualize the risk of insufficient pumping. In this case, it means no fluid reaches "node_2". This is a failure event and will be the Top Event (TE) in the generated SFT. The event can happen when:

1. There exist only closed check-valves or;
2. There exist only broken check-valves or;
3. There exist only inactive pumps or;
4. There exist only broken pumps or;
5. The tank has no fluid output or;
6. Any other combination of mentioned events in which no fluid reaches “node_2”.

To visualize the measurements, the P&ID contains a “leakage_group_1” and a “test_loss_fluid_a”. The first one groups the measurements. The second one tests the measurements to the condition that there should be some fluid at “node_2”. Components within the P&ID can contain failure events. For each failure event, RSMB can generate an SFT. When components are in parallel, the SFT generation process uses an AND gate. When they are in series, the process uses OR gates. The actual SFT generation process is substantially more complex than this. This complexity is necessary to correctly reflect failures and their propagation throughout the system. Important to note is that the component names always correspond to the descriptions of Basic Events and/or gates in the resulting SFTs.

The P&ID also contains non-visual data. The components “mdp_1” and “mdp_2” are already running and both adjacent check valves are open. The state of “tdp_1” is currently on “standby”. The state of the check valve adjacent to “tdp_1” is currently closed. The SFT generation process takes this into account since changing these states contains risks. This non-visual data becomes very clear in Figure 1.3. This figure shows the generated SFT. For the motor-driven pumps, insufficient pumping only happens in two ways. Either the operation of the pump fails, or the operation of the tank fails (e.g. “MDP_1_FO” and “TANK_1_FO”). The turbine-driven pump takes two more things into account. This is the opening of the check-valve, and the actual liquid flow through the links (“CHECK_VALVE_3_FO” and “TDP_1_FD”). In the labels, the abbreviations “FO” and “FD” mean “Failure of Operation” and “Failure of Demand”. The colours in both Figures 1.3 and 1.1 depict the link between the Basic Events in the SFT and the components in the P&ID.

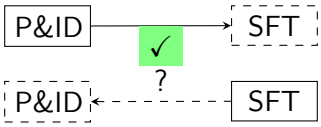


FIGURE 1.2: Simple representation of the problem

In this example, the SFT visualizes the risks of unwanted events happening in a power plant. An expert created this SFT without having an actual P&ID of that power plant. The problem is that there is only the SFT describing when there is insufficient pumping. I would love to also infer the P&ID as another source of analysis for the power plant. Figure 1.2 shows a simple representation of this problem. Currently, there already exists software which generates SFTs from P&IDs. Hence, the green checkmark with the arrow representing this. The question mark represents the process in the opposite direction. This is what I want to find a solution for. The solution could offer one P&ID with the highest probability of it being a good P&ID. It may also offer many P&IDs to an expert to choose the best option from. The latter one is especially useful if the use case differs per P&ID.

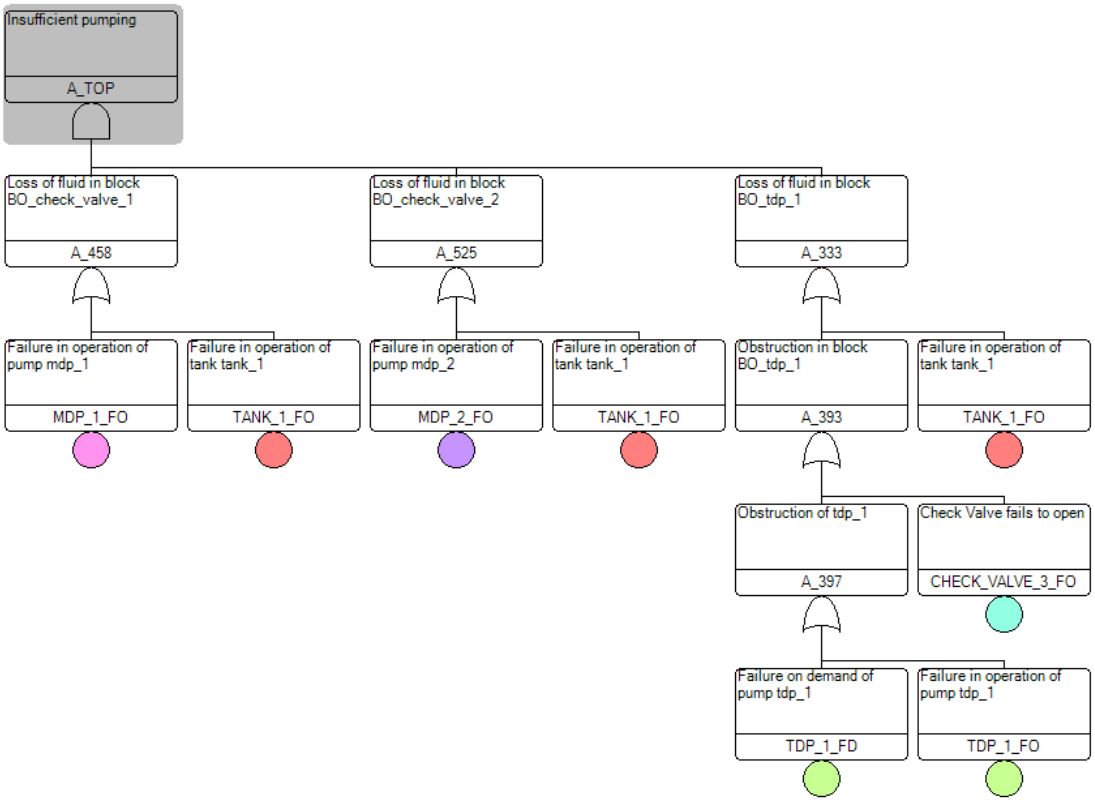


FIGURE 1.3: Generated SFT of the P&ID depicted in Figure 1.1

1.2 Problem context

One domain in which risk analysis and hazard prevention play a big role is the energy sector [32, 69, 52]. Especially within electricity-producing power plants. Every type of power plant has its own risks. In the current digital age, electricity is essential to have. Emergency services without electricity will not function. Current communication methods rely on electricity. The financial sector depends on computers having electricity. In other words, society relies on electricity to thrive and survive. This reliance will only increase given the planned energy transition addressing climate change [76, 6].

1.3 Problem relevance

Luckily, both governments and electricity companies are aware of the problem's context. They invest in managing risks related to power plants. Either with funds or with legislation. The industry currently uses SFTs as a way to visualize, quantify and analyse risks. Currently, there are many proven methods to generate SFTs [35, 55], of which some already do use existing P&IDs [75, 34, 64].

There are situations where the transformation in the other direction becomes relevant. This is because P&IDs are very versatile. It would help many disciplines to generate P&IDs from SFTs. These disciplines use it to analyse flows, controls and alarms of a power plant. Furthermore, it is easier to apply research into predictive maintenance or other analysis methods on P&IDs [13, 5]. Other industries than the energy sector also use P&IDs.

Often, industry experts create the SFTs by hand or use software made for this purpose.

Automatic generation from a P&ID is impossible because of one of the following reasons:

- The P&ID is missing or does not exist
- The P&ID only exists physically
- A digital, non-formal P&ID exist

So, both SFTs and P&IDs are well-known models. There is value in having them both to manage risks within power plants. Solving the problem is relevant because it helps to have both SFTs and P&IDs as a representation of a power plant.

1.4 Research questions

This section discusses the research questions (RQs) to specify and clarify the scope of the research. Since P&IDs exist in a wide variety of domains, there also exist many types and definitions. Hence, the first question focuses on formally defining a P&ID:

RQ 1: Is it possible to formally define a P&ID?

P&IDs can have different symbols representing many different objects and processes. Therefore, it is necessary to divide RQ 1 into multiple parts. The first subquestion focuses on the structure of a P&ID. Since SFTs are graphs (see Section 2.1), it would be nice to define a graph structure for P&IDs. The second subquestion reduces the type of P&IDs — and the domain — for which to eventually infer P&IDs. Hence, the research will focus on one metamodel. Section 3.4 elaborates on the concept of a metamodel.

The metamodel I will focus on is the EXPSA metamodel in RSMB. This metamodel focuses on the definition of a P&ID for thermohydraulic systems. Section 2.3 explains this metamodel in detail. The representation of the metamodel is text-based and can be quite explicit and descriptive. This makes it more complex to read and interpret. Hence, I want to answer the following sub-questions:

RQ 1.1: How can we use graphs as a formal structure to define P&IDs?

RQ 1.2: How does the EXPSA metamodel relate to a formal definition of a P&ID based on a graph structure?

Section 2.2 proposes an idea and an initial definition to answer RQ 1.1 with. RQ 1.2 follows from RQ 1.1 since it influences the characteristics of a graph-based P&ID.

Next to the RQs on the formal definitions, the following RQs focus on the process itself:

RQ 2: To what extent can we infer P&IDs from (manually created) SFTs?

RQ 2.1: To what extent can we determine what type of P&ID node we need to create?

RQ 2.2: What information do we need to infer relationships or connections between nodes in P&IDs and how can we use that information?

The sub-questions focus on altering and applying existing methods for inferring P&IDs, P&ID topology, and node types. Chapter 3 discusses related works and gives inspiration for possible answers to RQ 2. After that, Chapters 4 until 6 elaborate on a proposal for an algorithm to answer RQ 2, the algorithm's implementation and its verification & validation. Subsection 4.2 is especially interesting for RQ 2.1, whereas Subsection 4.4 goes deeper into RQ 2.2. Then, Chapters 7 and 8 show the results and discuss accomplishments and limitations to the proposed algorithm. This thesis ends with two chapters summarizing and concluding the research, as well as suggesting areas for future work.

Chapter 2

Background

The following Sections provide background information. More specifically, this Chapter covers three topics: Fault Trees (Section 2.1), Piping & Instrumentation Digarams (Section 2.2) and the Figaro language (Section 2.3).

2.1 Fault Trees

Fault Trees (FTs) [72] are, despite their name suggests, Directed Acyclic Graphs (DAGs) [71]. Their vertices visualise and represent events with a certain risk. An FT has two types of nodes: Events and Gates. There is a distinction between a *Top Event (TE)* and a *Basic Event (BE)*. An FT always has one TE at the root of the tree. BEs usually are the leaves of the FT and are Booleans. The Gates of a FT show how to evaluate the Boolean values of the BEs to determine whether the TE fails. The amount of gate types depends on the type of the FT.

FIGURE 2.1: SFT symbols

One type of FTs is a Static Fault Tree (SFT). For my purpose, I derive a formal definition from the definition of (Dynamic) Fault Trees (DFTs) given by Ruijters and Stoelinga [68] and Junges et al. [38]. It is worth noting that the formal definition of Ruijters and Stoelinga follows Codetta-Raiteri et al. [61].

SFTs can make use of six different symbol types. Figure 2.1 shows these types. They are:

(a) An AND gate (a logical AND gate)

(b) An OR gate (a logical OR gate)

(c) An VOT(k/N) gate (a voting gate)

(d) An event (a failure description)

(e) A BE (signalling the event is a Basic Event)

(f) A Transfer node (referencing other input- or output FTs)

Example 2.1.1. Figure 2.2 shows an example of an SFT containing every symbol type except the Transfer symbol. On the bottom are the BEs of this SFT. These nodes are the leaves of the DAG. The BEs are an empty battery (E11), a broken engine (E12) or one tire failure (E21 - E25). BEs can fail constantly or optionally. The former means a component can be working forever or broken forever. The latter means a component fails based on a probability distribution. This distribution then indicates how likely it is for the component to fail within a certain amount of time T. When a BE fails it evaluates to True.

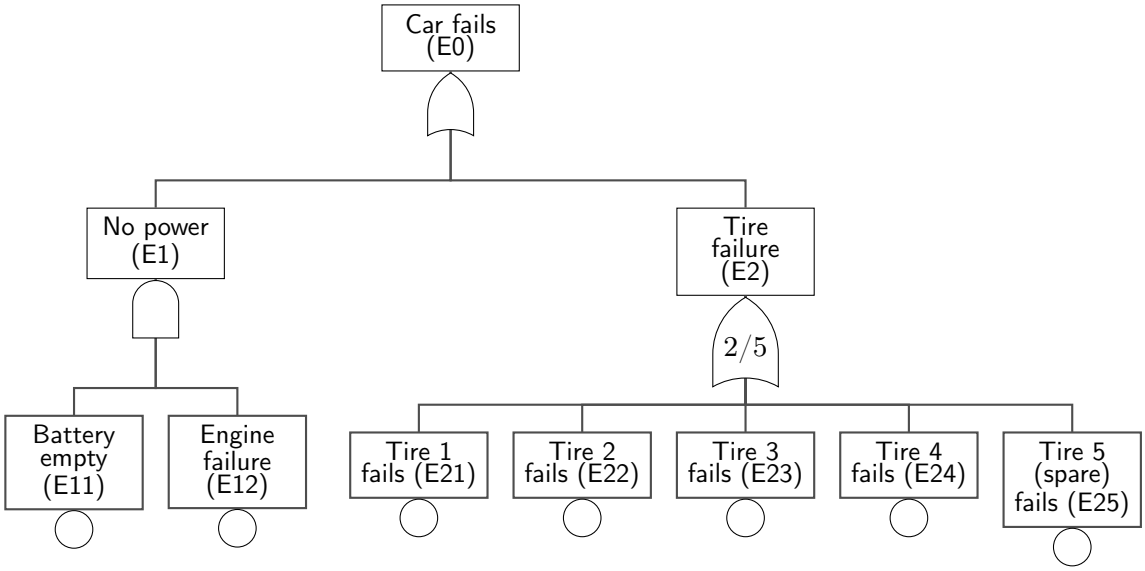


FIGURE 2.2: Simple SFT modelling failures of driving a car

On the top of the SFT, there always is the TE. The nodes and gates between itself and the BEs evaluate the TE. For Example 2.1.1, the car failing is the TE. The nodes between the TE and the BEs are the gates. There are three types of gates: OR, AND and VOT(k) gates. These gates evaluate their children (either BEs or other gates) based on their type. They fail if respectively one, all or k number of children fail. Sometimes, other gate types exist, but for my research, this is out of scope.

A formal, mathematical definition of an SFT is, however, within scope. Again, Ruijters and Stoelinga [68] offer a clear one:

Definition 2.1.2 (SFT)**.** Let $GateType = \{And, Or\} \cup \{VOT(k/N) \mid k, N \in \mathbb{N}^{>1}, k \leq N\}$. Then, an SFT consists of a five-tuple $SFT = \langle SLA, BE, G, GT, I \rangle$.

- SLA: The label of the SFT.
- BE: The set of Basic Events.
- G: The set of gates.
- $GT : G \to GateTypes$: Function describing each gate's type.
- $I : G \to P(E)$: Function describing the gate's inputs.

The following rules and conditions apply:

- $BE \cap G = \varnothing$ where $E = BE \cup G$ is the set of *elements*

- $I(g) \neq \varnothing$
- $GT(g) = VOT(k/N) \rightarrow |I(g)| = N$

The graph formed by $\langle E, I \rangle$ is a DAG with a unique root TE. The TE is reachable from all other nodes.

It is possible to perform both qualitative and quantitative analysis on an FT. Usually, a bigger FT is more complex to analyse since there are more nodes and gates to consider. The qualitative analysis of the FT in Figure 2.2 is quite trivial. It is clear that the k in the VOT(k) increases when there are more tires for the car. Hence, the chance of having three or fewer functional tires is lower. This does assume the failure probability of one tire failing, stays the same. It is common to use (minimal) cut sets for qualitative evaluation [68].

One method of quantitative FT analysis is by using failure probabilities. For this purpose, each BE has a failure probability function. With this, it is possible to calculate the chance of the TE failing. The complexity of this calculation depends on the size of the fault tree, as well as the probability distribution of the BE failing. At one point, this calculation can become too complex. Methods exist to estimate this probability instead of calculating it like using the Minimal-Cut-Set (MCS). Lee et al. [46] reviewed a selection of algorithms to calculate the MCSs of FTs back in 1985. In 2015, Ruijters and Stoelinga [68] surveyed the state-of-the-art in modelling, analysis and tools for FT analysis. Within the nuclear power plant domain, the MOCUS algorithm (with cutoff) is the standard method [63].

2.2 Piping & Instrumentation Diagrams

Piping & Instrumentation Diagrams [77] are diagrams describing a variety of (industrial) processing plants. The diagrams can include (emergency) processes, alarms and controls. Examples of processing plants are food processing, pulp-and-paper and wastewater treatment plants. This research focuses on nuclear power plants. P&IDs usually describe plants making non-discrete products. They consist of predetermined symbols which visualize and represent the components of the plant. Each type of component (tanks, valves, pumps etc.) has its own symbol. The size and orientation of the symbol are not linked to the component's size and orientation. Lines between symbols represent links between the plant's components. Amongst others, these links can be fluid pipes, gas pipes or signals. The length of the lines does not represent the length of the physical connection between the plant's components. Different lines (e.g. dashed, dotted or straight) show different connection types (e.g. fluid, gas or signals) between components.

The EXPSA metamodel in RiskSpectrum ModelBuilder (RSMB) uses a variety of symbols in their P&IDs. Figure 2.3 shows twelve of them. Almost all correspond to types of components which physically exist in the system the P&ID represents. Exceptions are symbols A, D and E; these symbols represent abstract components. Section 2.3 elaborates on this metamodel.

Appendix A shows an overview of the object types in the EXPSA metamodel. For now, you can ignore the colours in the Appendix. RSMB also uses symbols for other component types. The component types defined in Definition 2.2.1 are based on the twelve symbols visible in Figure 2.3.

Developing P&IDs is a field of research on its own since P&IDs are multidisciplinary, and they can serve many purposes. P&IDs can describe most aspects of a (chemical) process plant, either graphically or textually. There is a disadvantage of especially complex and big P&IDs. For one person to understand such a P&ID, they need knowledge of many

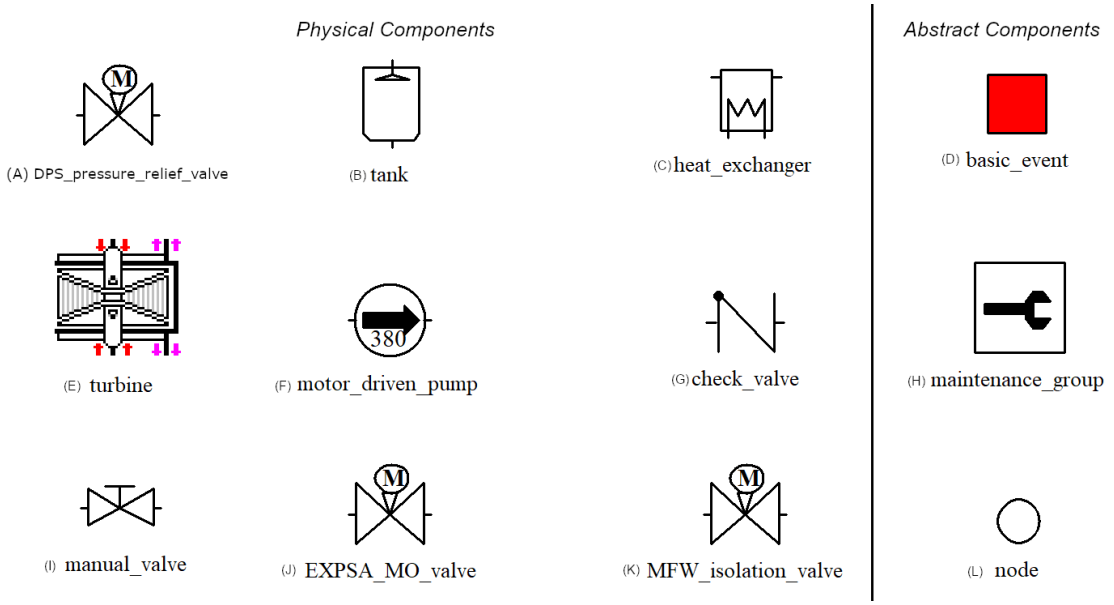


FIGURE 2.3: Examples of symbols for components in RSMB

different fields. More developed projects need more developed P&IDs, either because of the sheer size of the plant(s) or because of regulations.

One way to formally define P&IDs is as a directed graph;

Definition 2.2.1 (P&ID)**.** Let $PID = \langle PLA, C, L \rangle$ be a P&ID as a directed graph. Then PLA is the label of the PID, C is the set of components (vertices) and L the set of links (edges) of PID. Furthermore, let:

- $ComponentTypes = \{DPS\_pressure\_relief\_valve,\ tank,\ heat\_exchanger,\ basic\_event,\ turbine,\ motor\_driven\_pump,\ check\_valve,\ maintenance\_group,\ manual\_valve,\ EXPSA\_MO\_valve,\ MFW\_isolation\_valve,\ node\}$

and

- $LinkTypes = \{FluidPipe\}$

One element in C consists of a tuple $C = (CLA,\ CT)$:

- CLA: The label of C.
- CT: The ComponentType of C.
  $CT \in ComponentTypes$ holds.
  Section 4.2 elaborates on classifying CT for P&IDs of thermohydraulic systems.

One element in L consists of a four-tuple $L = \langle LLA, LT, CA, CB \rangle$:

- LLA: The Label of the Link.
- LT: The Type of the Link. $LT \in LinkTypes$ holds.
- CA: The Component at the head of the Link. $CA \in C$ holds.
- CB: The Component at the tail of the Link. $CB \in C$ holds.
  Section 4.4 describes determining CA & CB in more detail for P&IDs of thermohydraulic systems.

Note that there currently is only one type available in the set of LinkTypes. EXPSA does define three more classes which can link component: *comp_ test_ link, block_ link and dependency_ link*. These classes are used to signal abstract relations between components instead of physical ones. Hence, there exists only one type of physical link. This set can, in the future, be extended to eventually deal with gas pipes and/or electricity cables.

Now that an initial definition exists for P&IDs as directed graphs, the following extension and properties are necessary for validating the methodology (see Section 6.2). Let $PID = \langle PLA, C, L \rangle$ be a P&ID. Then:

- $|L|$ is the size of the set of links of a P&ID
- $CLA = \{CLA \mid (CLA,\ CT)\ \in\ C\}$
  This is the set of component labels of a P&ID.
- $\rho : CLA \to PLA$: A function describing the relation between CLA and PLA.
  More specifically, this function retrieves the PLA from CLA. The way this function works differs across domains. Sections 4.1 and 5.4.4 describe this function in more detail for P&IDs of thermohydraulic systems.
- $Component_a \Rightarrow Component_b$: component a is adjacent to component b.
  $\exists L \in PID : Component_a = CA \land Component_b = CB$
- $Component_a \nRightarrow Component_b$: component a is not adjacent to component b.
  $\nexists L \in PID : Component_a = CA \land Component_b = CB$
- $AbstractComponentTypes = \{node,\ maintenance\_ group,\ basic\_ event\}$
  is the set of abstract (non-physical) component types.
  $AbstractComponentTypes \subseteq ComponentTypes$ holds here.
- $AbstractComponents(PID) = \{c \mid c \in C \land\ CT \in AbstractComponentTypes\}$
  is the set of abstract components in a P&ID.
- $PhysicalComponents(PID) = \{c \mid c \in C \land\ CT \notin AbstractComponentTypes\}$
  is the set of physical components in a P&ID.

Definition 2.2.1 offers one idea to answer RQ 1.1 and RQ 1.2 with, which then can be used as a data structure for answering RQ2. Note that for simplicity and easier validation (see Sections 6.2.1 , 7.1 and 7.2), connections between PID components are unidirectional in this thesis. This means that the following holds:

$$\forall L \in PID : CA \neq CB \land CA \Rightarrow CB \longrightarrow CB \nRightarrow CA$$

Figure 2.3 already shows a visual representation of P&ID components, which offers a fitting introduction to the following example:

Example 2.2.2. Figure 2.4 shows a trivial example of a P&ID's visual representation in RSMB. The diagram shows a plant which only moves a liquid from one tank to another with a motor-driven pump and a manual valve. There are no checks or alarms on any node. Furthermore, there is no redundancy. Note that this is only one example of a P&ID. There exists a wide variety of P&ID types and a wide variety of symbols.

This is only one example of a P&ID. Nowadays, there is a variety of online tools[1] to create digital P&IDs. Furthermore, there exist various file formats which can describe P&IDs. One standardized notation is in DEXPI [56]. This thesis uses graphical representations of components from the EXPSA metamodel to denote P&IDs.

[1] https://www.edrawsoft.com/5-best-pid-software-and-tools.html

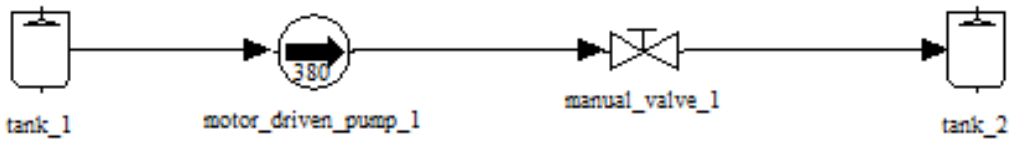


FIGURE 2.4: Simple P&ID showing an industrial plant. The plant moves a liquid from one tank to another. It has a motor-driven pump and a manual valve.

2.3 The Figaro Language

Figaro [8] is a modelling language aimed at modelling discrete state space systems and processes within those systems (amongst other goals). The main purpose of this language is to evaluate, study and visualize these systems and processes. This decreases the time necessary to perform reliability analysis and the risk of human errors.

2.3.1 Model-Based Safety Assessment (MBSA)

As mentioned in Chapter 1, Model-Based Safety Assessment (MBSA) is an alternative to manually building SFT models. Model-Based Safety Assessment can:

- Encapsulate the reliability knowledge in a tool;
- Allow analysts to create safety models closely resembling the system models and;
- Automatically generate the SFTs.

One such tool is RiskSpectrum ModelBuilder (RSMB), which is based on KB3. EDF has been developing KB3 since the early 1990's. A basic concept of KB3 is encapsulating reliability information into Knowledge Bases. An analyst builds graphical models of the system based on pre-defined components (e.g. P&ID components). A Knowledge Base defines the behaviour of these components, including failures and failure propagation. An (expert) user can use the Figaro modelling language to specify a Knowledge Base. So, KB3 allows (expert) users to define their own Knowledge Bases suitable for the domain and analysis type of their choice.

RiskSpectrum ModelBuilder commercializes and further develops KB3 in a partnership between EDF and RiskSpectrum AB[2]. One of the aims is to integrate this tool with the RiskSpectrum software suite. A special tool in this software suite is the event tree/fault tree modelling and analysis tool RiskSpectrum PSA (RSPSA). This currently is a leading tool in Probabilistic Safety Assessment (PSA) for nuclear power plants.

This thesis uses RiskSpectrum ModelBuilder (RSMB) together with the EXPSA Knowledge Base as the tool to represent P&IDs. Furthermore, RSMB can automatically generate fault trees from them and export those to RiskSpectrum PSA (RSPSA).

2.3.2 Language Levels

KB3 – and thus also RSMB – is based on the Figaro language. There are two "levels" in Figaro, called level 0 and level 1 by analogy with rule-based expert systems [7]. Figaro 0 is a simple sublanguage of Figaro 1; it can only describe specific systems. Figaro 1 contains

[2] https://www.riskspectrum.com/

sophisticated constructions making it possible to write generic models of components. These components then will fit in different system topologies, with various numbers of connections to other components. Some of the sophisticated constructions in Figaro 1 are similar to first order quantifiers ($\forall$ and $\exists$). It is possible to use Figaro 1 to define metamodels (see Section 3.4) suitable for categories of systems. Such a metamodel is in fact a set of classes described in the Figaro 1 language, usually called a Knowledge Base (KB). An example is the EXPSA KB for thermohydraulic systems. It is thus possible to consider Figaro as both a domain-specific modelling language (Khan et al. [41]) and a generic one (Kriaa et al. [43]).

Once a user of KB3 or RSMB has loaded a knowledge base in the tool, they have a dedicated tool enabling them to describe a system graphically. RSMB can then transform this system (in this thesis, a P&ID) into a Figaro 0 model. The user then can create simulations or generate SFTs for various Top Events based on this model.

2.3.3 Occurrence & Interaction Rules

In both models and metamodels, Figaro offers occurrence rules and interaction rules. *Occurrence rules* model the events which can occur on an object. They also define when and in which conditions those events happen. When generating SFTs, occurrence rules define the BEs of an SFT. *Interaction rules* model the consequences of these events. They express the propagation of state changes from an object to the objects it is connected to. These rules determine the logical structure of an SFT. The software of RiskSpectrum uses backpropagation over these rules to generate the SFTs of a P&ID (cf. Section 3.1).

As mentioned in the previous subsection, some constructions are similar to first order quantifiers. With the addition of occurrence and interaction rules, parts of Figaro look very similar to first-order logic conditions. Do keep in mind that Figaro only uses finite sets of objects.

2.3.4 Alternatives

Kriaa et al. [43] mention two alternatives for Figaro. Both alternatives are Domain-Specific Languages (DSLs) within the safety domain. These alternatives are AltaRica [58] and AADL [25].

The disadvantage of AltaRica is that it is less friendly to users than Figaro. This is because system analysts have to work with the AltaRica language itself to change a model. Figaro has the advantage of offering a graphical user interface.

AADL differs from both languages. It focuses more on electronic systems (processors, buses, buffers, etc.). Furthermore, it is less concise, and it cannot model dynamic failure propagation.

2.3.5 Figaro and P&IDs

For my research, I will use the Figaro language to describe both the P&ID models and the metamodel as a definition for a P&ID. I chose to use Figaro because of the following reasons:

- It provides an appropriate formalism for defining models and metamodels;
- Using Figaro's inference rules to generate SFTs from P&IDs is a proven and validated method;
- The language is legible and associable with graphic representations.

For the full documentation, syntax description and accompanying examples I would like to point the reader to the Visual Figaro IDE[3]. One can find the Figaro files of the example used in Section 1.1 on the GitLab repository[4] under "/doc/". The metamodel I will use in my research (EXPSA) is also available here.

[3] https://sourceforge.net/projects/visualfigaro/
[4] https://gitlab.com/wbos/spc

Chapter 3

Related Works

Unfortunately — as far as I know — there exists no relevant literature about inferring P&IDs from SFTs. However, there are three subjects giving more context:

1. Converting P&IDs into FTs
2. Inferring P&IDs from other models
3. Using FTs to infer other models

Next to these subjects, there also exist relevant papers on *Model-Driven Engineering* (MDE). Throughout this proposal, I already use several concepts and definitions related to MDE.

For now, the field of Fault Tree Analysis (FTA) is out of scope. So far, FTA did not help with answering the research questions. I do keep FTA in mind since it can become relevant at a later stage of solving the problem.

3.1 Converting P&IDs into FTs

This section focuses on ways to convert P&IDs into FTs. It starts with a look at RSMB's algorithm. After that, it looks at four alternatives.

Renault et al. [64] describes how they infer FTs from P&IDs using Figaro. It starts from the Figaro 0 model of a given system. One can obtain this model after applying generic rules of a Figaro Knowledge Base on the objects (in our case: pumps, valves, tanks etc.) of this system. The Figaro 0 model describes:

1. The probability of occurrence of component failures (via occurrence rules) and,
2. How these failures can propagate in the system (via interaction rules).

It is possible to obtain the FT gates by backwards-chaining on the interaction rules. The occurrence rules determine the probabilistic parameters of the basic events. This whole process actually starts by merging all rules contained in the various objects into one single set. Because of this, it is impossible to trace back from the fault tree where (meaning: in what object) the rules originally were. This is why we can only rely on a rigorous naming scheme of basic events to be able to find object names. Nevertheless, the paper of Renault et al. [64] is still the most relevant for the proposed methodology in Section 4.

Next to Figaro, there exists AltaRica [58] and AADL [25] as Domain-Specific Languages (DSLs [42]) within the safety domain. Prosvirnova [58] introduces AltaRica as DSL and presents an algorithm which compiles Guarded Transition Systems (GTS) — the underlying

mathemetical formalism of AltaRica — into FTs Feiler and Delange [24] generate FTs from AADL models using the SAE AADL Error Model V2 Annex (EMV2) standard [1]. Subsection 2.3.4 further discusses these alternatives

Taylor [75] developed an algorithm for synthesizing FTs from a flow chart, wiring diagram or a P&ID. The paper describes a four-step process:

1. Hazard classification
2. Identification of necessary hazard causes
3. Localization of hazard causes
4. FT construction

Taylor focuses especially on the last step. This step assumes a P&ID has been constructed and represented as a Mealy Machine. Then, a user of their algorithm selects a mission failure event which becomes the TE of the SFT. The rest of the SFT is then composed of mini fault-trees which are event outputs of earlier mentioned Mealy Machines. The next step is to compose the SFTs by traversing the Mealy Machines. Here, redundant loops could occur. This can happen when a part of a Mealy Machine is traversed more than once. Taylor does describe how the algorithm deals with possible loops.

Hunt et al. [34] give an overview of modelling the decomposition, flow, propagation and tracing of P&IDs for fault-tree synthesis using FAULTFINDER. In FAULTFINDER, their FT generation algorithm FLTGEN uses many different kinds of diagrams and tables based on the initial P&ID.

Currently, no source exists using the methodologies of Talor and Hunt et al. on Fault Trees to synthesize P&IDs. However, because of the complexity of those methods, they could serve as inspiration for this aim.

3.2 Inferring P&IDs from other models

On the topic of inferring P&IDs from other models, I only found papers related to the conversion from P&ID images (e.g. JPEG, PNG or PDF files) to object-oriented or database-compatible data formats.

I want to highlight two papers which go deeper into this process as a whole. Both Arroyo et al. [2] and Paliwal et al. [57] describe their own method of deriving and/or digitizing P&IDs for further processing. For us, Paliwal et al. may have an interesting dataset of 500 P&IDs. This dataset could be useful in the proposed methodology in Section 4.

There are four more papers [73, 59, 21, 26] I found interesting related to inferring P&IDs from image data. These papers have a narrower scope. They describe different methodologies for component and information detection within the bigger process. I choose to use a different method for my research. Based on one or more of these four papers, future research may be done into using different component and information detection methods to infer P&IDs from SFTs. A first step here would be to replace the input for these methods for SFT image data. Because I expect this results in poor quality P&IDs, a second step could be to alter the methods.

Lastly, I would like to point towards one paper describing a data structure in which P&IDs could be saved. Oeing et al. [56] describe the DEXPI standard. DEXPI is a "machine-readable, manufacturer-independent exchange standard for P&IDs". It seems mostly focused on making P&IDs accessible for AI-related research.

3.3 Using FTs to infer other models

For the last subject, I found five papers which infer other models from FTs. All these papers transform FTs into other models or data structures with the goal of improving Fault Tree Analysis. The first paper which gives a stat-of-the-art overview is published by Ruijters and Stoelinga [68]. They review quantitative and qualitative analysis methods for extended, repairable, dynamic and standard fault tees with many examples.

Liu et al. [49] propose the Cut Sequence Set Algorithm (CSSA) which generates the Cut Sequence Set (CSS) of an FT. A cut sequence is an ordered list of BEs which induce the TE. This differs from Minimal Cut Sets (MCSs) in that a CSS is the ordered sequence. The order in MCSs is irrelevant. Hence, CSSA provides another qualitative method of FTA.

For a quantitative method, a model-to-model transformation exists from Dynamic Fault Trees (DFT) to Generalized Stochastic Petri Nets (GSPN). Codetta-Raiteri [60] uses graph transformation rules for this transformation. The resulting GSPN can then be used to generate a Continuous Time Markov Chain (CTMC) of the system. A CTMC is useful for determining the reliability over time. Codetta-Raiteri does identify a problem with this method, however. Analysing DFTs by transforming it into other models remains a state space solution. Because of the state explosion problem [78], this can become computationally expensive.

Fortunately, this problem inspired Junges et al. [37] to reduce DFTs through graph rewriting. They presented rewriting rules which yielded timing gains. These gains were up to two orders of magnitude within their set of benchmarks. Despite rewriting resulting in the same model type, simpler SFTs might help to infer P&IDs. The extent to which this helps, depends on which properties of the original SFT are preserved.

The final paper which is interesting is about the uniform analysis of FTs through model transformations. Here, Ruijters et al. [67] present a unified metamodel describing a variety of FTs, attack trees (ATs) and combined attack-fault trees. Furthermore, they provide model-to-model transformations between different formalisms of FTs and ATs.

3.4 Model-Driven Engineering (MDE)

Model-Driven Engineering [39] is one of many fields of research within computer science. Usually, when engineering software, we make models and diagrams which focus on architecture. This principle is also known as Model-Driven Architecture (MDA). MDE is the combination of MDA with a *process* and *analysis*. In this case, it offers another perspective on the problem, which helps me to describe and define some related concepts. Furhtermore, MDE is relevant because the research questions focus on combining an MDA with a process and analysing both this process and the architecture.

Within MDE, I would like to introduce the reader to the following three definitions:

- Models;
- Metamodels;
- Model-To-Model Transformations.

Models are abstract representations of an existing system, usually made with a specific purpose in mind. Both P&IDs and SFTs can be seen as models of industrial systems, but with widely different purposes, advantages and disadvantages. P&IDs are a more general-purpose type of model, describing both the structure and many other concepts of the

industrial system. SFTs, however, are used to show a certain risk of an event happening in this industrial system.

Metamodels can be defined in many ways which all offer their own perspective on the concept of a metamodel.

Seidewitz [70] views a metamodel as something that defines what can be expressed in valid models of a specific modelling language. Kühne [45] gives a variation of definitions of which the simplest, in my opinion, are: "a model is an instance of a metamodel", and "a metamodel is a model of models". The paper elaborates more on the concept of both models and metamodels related to MDE.

Figaro can be used both to describe P&IDs and the metamodel of this P&ID. This metamodel can be vastly different between P&IDs for different kinds of industries. Since the research questions focus on inferring P&IDs from SFTs, this process seems similar to a *uni-directional model-to-model transformation*. Well-known related languages or frameworks for defining model-to-model transformations are ATL[1], MMT[2] and QVT[3]. The use of any languages or frameworks related to MDE is out of scope in this thesis. Future research could focus on defining the adhoc solution in ATL, MMT or QVT.

[1] https://www.eclipse.org/atl/
[2] https://wiki.eclipse.org/Model_to_Model_Transformation_-_MMT
[3] https://www.omg.org/spec/QVT/1.3/About-QVT/

Chapter 4

Methodology

This chapter focuses on the methodology of inferring P&IDs from SFTs. The methodology uses an algorithm which iterates over a list of SFTs. For each SFT it creates a fresh, empty P&ID and fills this by following three steps. Figure 4.1 shows these three steps. The first step (Section 4.2) consists of determining P&ID components from the BEs of an SFT. The second step (Section 4.3) removes each BE of the SFT, which relates to maintenance. The third step (Section 4.4) determines the topology of the P&ID (i.e. the links) by a post-order depth-first traversal of the SFT.

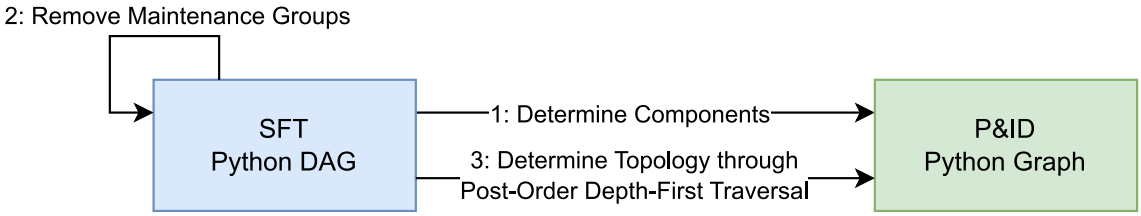


FIGURE 4.1: Three steps for inferring a single P&ID from a single SFT.

4.1 High-level Algorithm Overview

The main focus of this chapter is on translating a single SFT $\langle SLA, BE, G, GT, I\rangle$ to a single PID $\langle PLA, C, P\rangle$. However, it is possible that multiple SFTs exist for one system. This can happen because of (e.g.) different system configurations, different system representations, and the use of transfer gates. Ideally, the algorithm merges the information of these SFTs into one resulting P&ID. Hence, the method needs to reduce a tuple of $(SFT_1, SFT_2, \ldots, SFT_n)$ to a single P&ID.

The algorithm does this before and after following the aforementioned three steps for inferring a single P&ID from a single SFT. It first determines whether a P&ID already exists for the SFT it is now converting. Then, as a post-processing step (see also Section 5.4.4), it merges the resulting P&ID with the already existing one.

Listing 4.1 shows the pseudocode for converting a list of SFTs to a list of P&IDs. Most of this method focuses on the aforementioned post-processing. It starts with creating an empty list for the resulting PIDs. Then, it iterates over the list of SFTs, converting each SFT to a P&ID. Line six, calls a different method to convert a single SFT to a P&ID.

After this converting step, it looks at the current list of resulting P&IDs whether a P&ID with the same ID exists. If that is the case, it merges both P&IDs. Subsection 4.1.1 explains how the algorithm merges two P&IDs. If no P&ID with the same ID exists, it can

simply add the P&ID to the resulting list. For merging two P&IDs, the algorithm simply takes the union of two P&IDs. Thus, let:

$$PID_A = \langle PLA, C, L \rangle \text{ and } PID_B = \langle PLA', C', L' \rangle$$

Then the following holds:

$$PLA = PLA' \longrightarrow PID_{Merged} = \langle PLA, \{C \cup C'\}, \{L \cup L'\} \rangle$$

```
Method: infer_pids_from_sfts(sfts: List[SFT]) -> List[PID]
    result = []
    For sft in sfts:
        converted_pid = convert_sft_to_pid(sft)
        existing_pids: Set[PID] = {p for p in result if
                                   converted_pid.graph_id in p.graph_id}
        If len(existing_pids) == 1:
            pid_to_replace = existing_pids.pop()
            pid_merged = merge(pid_to_replace, converted_pid)
            result.remove(pid_to_replace)
            result.append(pid_merged)
        Else:
            result.append(converted_pid)
    Return result
```

Listing 4.1: Infer P&IDs from SFTs

Listing 4.2 shows the conversion of a single SFT according to the three steps mentioned at the start of this chapter. The SFT's SLA (see Definition 2.1.2) determines the PID's PLA. First, the algorithm splits the SLA on the hyphen character. This uses the fact that a naming scheme exists based on the EXPSA metamodel. The algorithm uses the first part as the ID for the fresh P&ID. So, let:

$$SLA = SLA_1 - SLA_2 - \ldots - SLA_n$$

Here, $SLA_1 - SLA_2 - \ldots - SLA_n$ represent the substrings obtained after splitting SLA based on the hyphen character. Then:

$$PLA = SLA_1$$

```
Method: convert_sft_to_pid(sft: SFT) -> PID:
    pid = new PID(directed=False)
    pid.pla = sft.sla.split('-')[0]

    determine_pid_components_based_on_sft(pid, sft)
    delete_sft_maintenance_groups_to_prevent_linking(sft)
    insert_pid_pipes_based_on_sft(pid, sft)

    return pid
```

Listing 4.2: Convert SFT to P&ID

4.1.1 Merging two P&IDs

Before explaining the three steps in more detail, this subsection explains how the algorithm merges two P&IDs. Listing 4.3 shows the pseudocode for merging two P&IDs. It first creates a fresh, directed P&ID which gets the label of the first input P&ID. Then, it stores the union of the components in the result. After that, the algorithm does the same for the links and returns the merged P&ID.

```
Method: union(pid_a: PID, pid_b: PID) -> PID
    res = new PID(pla = pid_a.pla, directed = True)
    res.c = union(pid_a.c, pid_b.c)
    res.p = union(pid_a.p, pid_b.p)
    return res
```

Listing 4.3: Merge two P&IDs

Subsection 8.3.4 discusses alternatives to taking the union as a method for merging two P&IDs.

4.2 Determining P&ID Components from SFT BEs

The first step of inferring a P&ID consists of determining its components. For this, the algorithm assumes that the IDs of an SFT's BEs follow a naming schema. This is possible because the naming scheme is based on the EXPSA metamodel. So, the naming scheme does suit larger P&IDs for the use case described in Chapter 1.

The general idea is to use this naming schema to deduce a component's label (CLA), type (CT) and ID (CID). As mentioned, CT and CID are a function of CLA (see also Section 2.2). The algorithm determines the CLA by iterating over the BEs of the SFT.

Listing 4.4 shows the pseudocode to obtain this. For each BE, it looks at its ID. The ID consists of the type of the P&ID component (two characters, CT), followed by the actual ID of the component (a number, CID). An example BE ID is *TK01* where *TK* is CT (a tank) and 01 is CID. The algorithm uses the full ID of the BE as the label (CLA) for the new component. When it is impossible to determine CT, the algorithm looks at the fault code of the BE. For now, a BE's fault code can only determine the *maintenance_group* component type. Note that:

1. The first match statement uses a naming convention to match the component type. This naming convention is *not* part of the EXPSA metamodel. Hence, it can be easily adapted to a domain-specific naming scheme.

2. The fault code also follows a naming convention *n*ot part of the EXPSA definition.

3. Currently, the method in the pseudocode chooses to alter the P&ID given in the parameter of the method, instead of creating and returning a list of components. This is because it is consistent with the current implementation. Of course, a different implementation can choose the alternative approach.

Contrary to the naming convention, the component types *are* based on the EXPSA metamodel. Each component type can be found in the classes of this metamodel (see also Appendix A).

```
Method: determine_pid_components_based_on_sft(pid: PID, sft: SFT)

    For basic_event in sft.be
        basic_event_id = basic_event.id
        component = new Component(basic_event_id)
        Match basic_event_id[0:2]:
            Case "GT":
                component.type = TURBINE
            Case "HX":
                component.type = HEAT_EXCHANGER
            Case "VC":
                component.type = CHECK_VALVE
            Case "VM":
```

```
                component.type = EXPSA_MO_VALVE
            Case "VH":
                component.type = MANUAL_VALVE
            Case "VI":
                component.type = MFW_ISOLATION_VALVE
            Case "VS":
                component.type = DPS_PRESSURE_RELIEF_VALVE
            Case "PM":
                component.type = MOTOR_DRIVEN_PUMP
            Case "TK":
                component.type = TANK
            Default Case:
                component.type = NODE

        If component.type == NODE
           and basic_event.faultcode == M:
            component.type = MAINTENANCE_GROUP

        pid.add_component(component)
```

Listing 4.4: Merge two P&IDs

4.3 Removing SFT BEs: Maintenance Groups

There is one component type in the EXPSA metamodel which the algorithm treats differently: *maintenance_group*. This component type is an abstract component instead of a real, physical one. In this specific case, the resulting P&ID must show this component, without linking it — *yet* — to real, physical components. Section 8.2.3 elaborates more on the topology w.r.t. *maintenance_group* components.

Currently, the algorithm removes BEs with a fault code indicating maintenance groups from the SFT. This happens between the steps of deducing components and determining the topology, because no link types exist in our P&ID definition for *maintenance_group* components (see Definition 2.2.1). Since the fault code of BEs also follows a naming scheme, the algorithm can use this to match the fault code.

Example 4.3.1. Figure 7.10 shows two SFTs containing BEs related to *maintenance_group* components. There are two unique BEs (three in total) which this step removes from their respective SFTs: *RHR-TR01-M* and *RHR-TR02-M*. Note that the SFT containing both BEs needs reducing afterward to prune the parent gate.

Listing 4.5 shows the pseudocode. The algorithm iterates over the BEs of the SFT. If the fault code of a BE matches, it removes the BE from the set of BEs. After that, the algorithm makes sure the BE is removed as input from all the gates. It does this by making use of the graph structure: it removes all edges incident to the removed BE. As the last step, it reduces the SFT. This makes sure that the SFT only contains gates with an input, which is necessary to determine the topology of the P&ID.

```
Method: delete_sft_maintenance_groups_to_prevent_linking(sft:SFT)
        for basic_event in sft.be:
            if basic_event.faultcode == FaultCode.M:
                sft.be.remove(basic_event)

                for edge in basic_event.incidence:
                    sft.i.remove(edge)

```

```
9                    reduce_sft(sft, True)
```

Listing 4.5: Remove BEs with Maintenance Group Faultcode from SFT

4.4 Inferring P&ID Topology

In the third and last step, the algorithm infers the topology of (or; links between) the P&ID's components. One interesting and vital characteristic of the relation between two distinct P&ID components is whether they are in series or in parallel. Therefore, the algorithm focuses on this aspect and makes three assumptions:

1. When two BEs are children of an AND gate, the corresponding components are linked in parallel.
   When one BE fails, the parent gate does *not* fail (and thus, the components under this gate are redundant and in parallel).

2. When two BEs are children of an OR gate, the corresponding components are linked in series.
   When one BE fails, the parent gate also fails (and thus, the whole chain of components which are in series fails).

3. When two BEs are children of a VOT gate, this gate behaves like an AND gate.

Since a VOT gate behaves like an AND gate, this section will omit further mentions of this gate type concerning the behaviour of the algorithm after detecting a VOT gate.

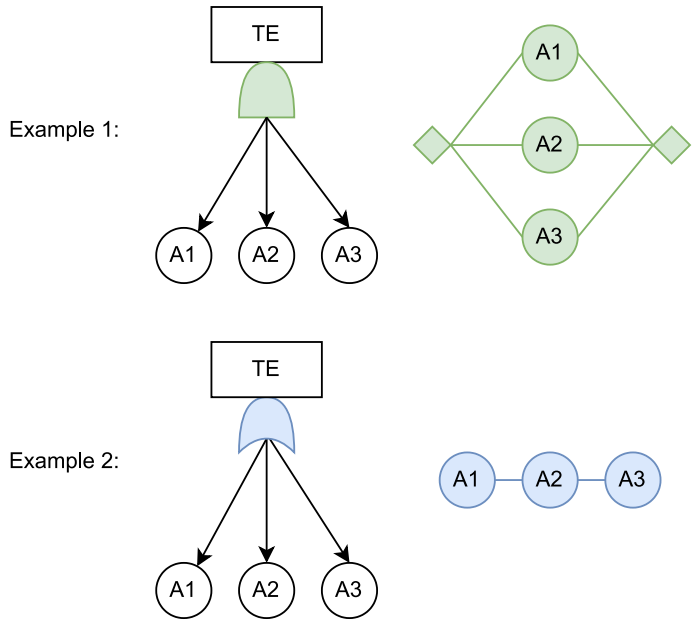


Figure 4.2: Topology determination with only BEs as children

Since the SFTs are DAGs, it is possible to traverse the graph. The algorithm for inferring the topology uses this by traversing an SFT from top to bottom.

Figure 4.2 shows two simple examples for which the algorithm can use the aforementioned assumptions while traversing. Example 1 shows an AND gate with three BEs as children, all corresponding to distinct P&ID components. This translates to the three

components being in parallel. Note that it is necessary to create extra nodes to show that the components are in parallel. Example 2 is very similar, except there now is an OR gate instead of the AND gate. This translates to the three components being in series. In this section, AND gates and their resulting structures will be green in figures. For OR gates, the figures will use blue.

Note that domain-specific knowledge is necessary for the order of the components to make sense. Furthermore, these examples avoid choosing between depth-first or breadth-first traversal. This is because they only have BEs as children.

Figure 4.3 shows two examples which also contain gates as children. Now, the order of the components does matter and the algorithm needs to apply either dept-first or breadth-first traversal.

Example 3 shows an initial AND gate, which has an OR gate as a child. This signals that the children of the OR gate are in series and the children of the AND gate in parallel. On the right, this is visible through B1, B2 and B3 being in series. Then, this substructure is in parallel with A1 and A3. Again, note the two extra nodes. Both nodes connect to A1 and A3. The first node connects to the first component of the OR gate's children. The second node connects to the last component of the OR gate's children.

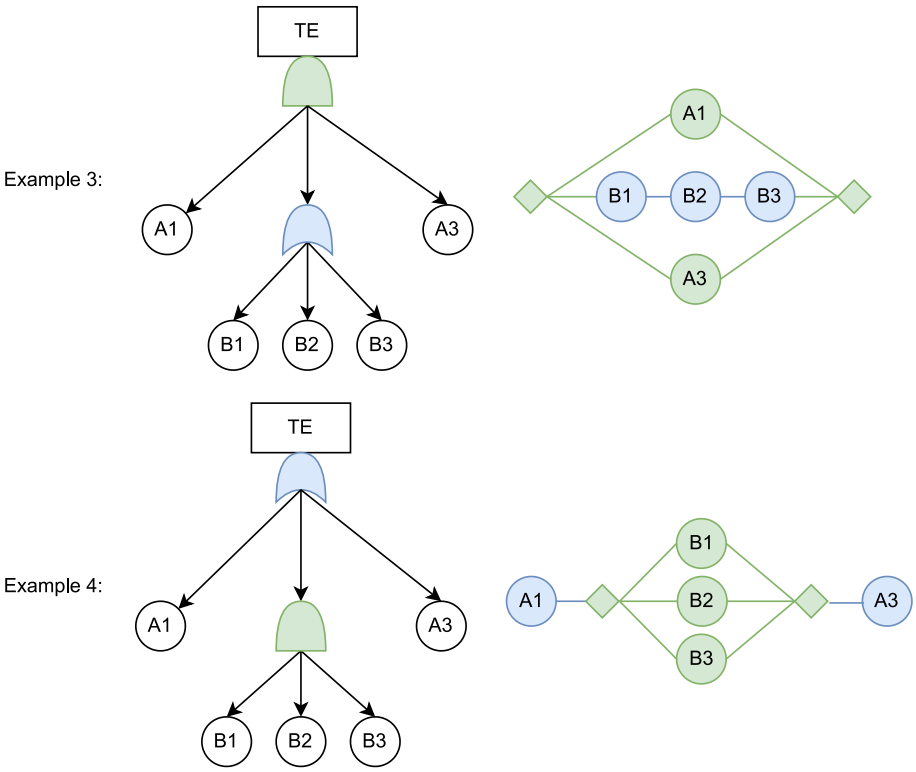


FIGURE 4.3: Topology determination with Gates and BEs as children

For Example 4, the OR and AND gate switched positions. The children of the AND gate are still in parallel, and the children of the OR gate still in series. Now, the extra nodes function as input and output of the substructure resulting from the AND gate. Hence, A1 connects to the first node. Then, the AND gate's sub-structure follows. After that, the second node connects to A3.

From examples three and four, it becomes quite clear the algorithm needs Post-Order Depth-First traversal. It is essential to know the resulting substructure of a child when it is a gate. Furthermore, determining whether and how to connect the extra nodes can only

be done after processing a gate's children.

Listing 4.6 shows the high-level pseudocode for retrieving a list of CLA tuples and adding links based on those tuples. The algorithm uses a list instead of a set since it makes use of the order in which CLA tuples are added. Note that in the implementation, the algorithm needs to ensure that each tuple is unique. This is because it then conforms to the formal definition of a P&ID where only unique links can exist. Furthermore, the algorithm assumes a fully reduced SFT as input (meaning that a gate always has one or more inputs).

On line three the algorithm calls a method to traverse over the SFT's gates and retrieve a list of CLA tuples (see Subsection 4.4.1). For this, it passes through the TE of the SFT (since it assumes the TE is always a Gate). Then, it starts tracking a counter, which functions as a generator for the Links' labels. After that, it creates a link for each CLA tuple in the list of CLA tuples. Here, the first element is the tail of the link and the second element is the head. Note that the link counter increases for each link that the algorithm creates and adds to the P&ID.

The next subsections continue elaborating on the details of the Post-Order Depth-First traversal algorithm.

```
1  Method: insert_pid_links_based_on_sft(pid: PID,
2                                        sft: SFT)
3      _, _, cla_tuples = post_order_depth_first_traverse_element(
4                                    sft.top_event
5                                )
6      pipe_counter = 1
7      for cla_tuple in cla_tuples:
8          pid.add_edge(str(pipe_counter),
9                       pid.find_component(cla_tuple[0]),
10                      pid.find_component(cla_tuple[1]))
11         pipe_counter += 1
```

Listing 4.6: Traverses an SFT and creates Links

4.4.1 Post-Order Depth-First Traversal (PODF)

For PODF, the algorithm recursively traverses each node of the SFT. After traversing, the algorithm returns a list of CLA tuples representing the CA and CB of a link which the algorithm then needs to create. Traversing starts at the TE. When traversing an element of the SFT, the algorithm looks at whether this element is a BE or a Gate. If it is a BE, it returns the BE's ID as input and output and an empty set (indicating there are no links). If it is a Gate, it checks the type of the gate. For each gate type, a method exists returning an input, output and a set of string tuples. This way, the method indicates the input, output and links in a "substructure" of a P&ID. All methods handling gates make a recursive call for each child.

Listing 4.7 shows the pseudocode for the high-level PODF algorithm. Lines four up until six show the case when the element is a BE. Lines seven up until nine show the case when the element is an OR gate. Lines ten up until twelve show the case when the element is an AND (or VOT) gate. Subsections 4.4.2 and 4.4.3 elaborate more on the two methods handling gate elements.

```
1  Method: post_order_depth_first_traverse_element(
2                          element: Gate | BasicEvent):
3      input, output, cla_tuples = "", "", set()
4          if isinstance(element, BasicEvent):
5              input, output, cla_tuples = \
```

```
                        element.id, element.id, set()
        elif element.gate_type == GateType.OR:
            input, output, cla_tuples = \
                        podf_traverse_or_gate(element)
        elif element.gate_type in (GateType.AND, GateType.VOT):
            input, output, cla_tuples = \
                        podf_traverse_and_vot_gate(element)
        return input, output, cla_tuples
```

Listing 4.7: Traverses an SFT and creates Links

4.4.2 Traversing an AND gate

When the algorithm traverses an AND gate, it knows that it has to put the children of this gate in parallel if there is more than one child. This means that in this case, it has to:

1. Create a component with Node as type, which is the start (or input) of the parallel structure
2. Create a component with Node as type, which is the end (or output) of the parallel structure
3. Traverse each child of the AND gate and deal with it properly by 1) copying the child's links; 2) linking the input node to the child substructure's input and 3) linking the output node to the child substructure's output.

Listing 4.8 shows the pseudocode for this. Lines two up until five generate the node components. Currently, it simply keeps track of a *node_counter* variable in the background which starts at one. This ensures that every node component in a P&ID eventually has a unique CLA. Then, lines six and seven deal with an AND gate with only one child by ignoring the node components and simply traversing the child.

After that, it traverses over the children of the AND gate. For each child, it first copies the child's CLA tuples if necessary. Then, the algorithm creates a tuple for the input of the child's "substructure" and the input node. Lasly, it creates a tuple for the output of the child's "substructure" and the output node.

```
1  Method podf_traverse_and_vot_gate(gate: Gate):
2      input_node_cla = "NODE" + str(node_counter)
3      node_counter += 1
4      output_node_cla = "NODE" + str(node_counter)
5      node_counter += 1
6      cla_tuples = set()
7      if len(gate.children) == 1:
8          return post_order_depth_first_traverse_element(gate.children[0])
9      for child in children:
10         child_input_cla, child_output_cla, child_cla_tuples = \
11             post_order_depth_first_traverse_element(child)
12         if len(child_cla_tuples) > 0:
13             cla_tuples = \
14                     cla_tuples.union(child_cla_tuples)
15         if child_input != "":
16             cla_tuples = \
17                     cla_tuples.union({(input_node_cla, child_input)})
18         if child_output != "":
19             cla_tuples = \
20                     cla_tuples.union({(child_output, output_node_cla)})
21     return input_node_cla, output_node_cla, cla_tuples
```

Listing 4.8: Traverse an AND gate in PODF

Let's now apply this algorithm on two examples.

Example 4.4.1. In Figure 4.3, Example 1 shows an AND gate with three BEs as children. The algorithm starts at the top AND gate. It first creates two nodes. Then, it traverses the three children *A1*, *A2* and *A3*. The children have a mapping to a component with *A1*, *A2* and *A3* as CLAs, and the algorithm links both nodes to this component. The result of traversing this AND gate hence is:

1. *Node1* as input
2. *Node2* as output
3. CLA tuples $\{(Node1, A1), (A1, Node2), (Node1, A2), (A2, Node2), (Node1, A3), (A3, Node2)\}$

Since this AND gate is the TE, it eventually ignores the resulting input and output.

Example 4.4.2. In Figure 4.3, Example 3 shows an AND gate with an OR gate as a child. The algorithm starts at the top AND gate. It first creates two nodes. Then, it traverses the first child *A1*. This child has a mapping to a component with *A1* as CLA, and the algorithm links both nodes to this component. It continues traversing the second child, which is an OR gate. After traversing this, the algorithm has:

1. *B1* as input
2. *B3* as output
3. CLA tuples $(B1, B2), (B2, B3)$

The first node then is linked to *B1* and the second node to *B3*. Furthermore, the algorithm passes through the CLA tuples resulting from traversing the OR gate. As the last step, it traverses *A3* and links this to both nodes. The result of traversing this AND gate hence is:

1. *Node1* as input
2. *Node2* as output
3. CLA tuples $\{(Node1, A1), (A1, Node2), (Node1, B1), (B1, B2), (B2, B3), (B3, Node2), (Node1, A3), (A3, Node2)\}$

4.4.3 Traversing an OR gate

When the algorithm traverses an OR gate with multiple children, it knows it has to put the children of this gate in series. This means that when traversing the children of the OR gate, it has to do the following steps:

1. Use the first child's input CLA as input CLA;
2. Use the last child's output CLA as output CLA;
3. Link a previous child's output CLA to the current child's input CLA and;
4. Copy any other CLA tuples of the gate's children.

Listing 4.9 shows the pseudocode for this. Lines six and seven cover step one. Line twelve makes sure the last child's output gets passed on as output CLA. Lines eight up until ten deals with step three. Note that the first part of the condition on line eight deals with the edge case when traversing the first child of the gate. The last step is visible on line eleven.

```
Method: podf_traverse_or_gate(gate: Gate):
    input_cla, output_cla, cla_tuples = "", "", set()
        for child in gate.children:
            child_input_cla, child_output_cla, child_cla_tuples = \
                post_order_depth_first_traverse_element(child)
            if input_cla == "":
                input_cla = child_input_cla
            if output_cla != "" and child_input_cla != "":
                cla_tuples = \
                    cla_tuples.union({(output_cla, child_input_cla)})
            cla_tuples = cla_tuples.union(child_cla_tuples)
            output_cla = child_output_cla
        return input_cla, output_cla, cla_tuples
```

LISTING 4.9: Traverse an OR gate in PODF

Let's now apply this algorithm on two examples.

Example 4.4.3. In Figure 4.3, Example 2 shows an OR gate with three BEs as children. The algorithm starts at the top OR gate. It traverses the three children *A1*, *A2* and *A3*. The children have a mapping to a component with *A1*, *A2* and *A3* as CLAs. When traversing *A1*, it saves that as input. Then, with *A2* it links this to *A1* and saves *A2* as output. Lastly, by traversing *A3*, it links this child to *A2* and overrides the earlier output. The result of traversing this OR gate hence is:

1. *A1* as input
2. *A3* as output
3. CLA tuples $\{(A1, A2), (A2, A3)\}$

Example 4.4.4. In Figure 4.3, Example 4 shows an OR gate with an AND gate as a child. The algorithm starts at the top OR gate. It first saves *A1* as input. Then, it traverses the second child, which is the AND gate. Subsection 4.4.2 earlier discussed this exact example of traversing an AND gate. After traversing this, the algorithm has:

1. *Node1* as input;
2. *Node2* as output and;
3. CLA tuples $\{(Node1, B1), (B1, Node2), (Node1, B2), (B2, Node2),$ $(Node1, B3), (B3, Node2)\}$

So, with this information, the code of Listing 4.9 links *Node1* to *A1*. Furthermore, it copies the child's CLA tuples as well as passing on *Node2* as output CLA.

Lastly, it traverses the last child under the OR gate. It links *A3* to *Node2* and saves *A3* as output CLA. The result of traversing the OR gate hence is:

1. *A1* as input
2. *A3* as output
3. CLA tuples $\{(A1, Node1), (Node1, B1), (B1, Node2), (Node1, B2), (B2, Node2),$ $(Node1, B3), (B3, Node2), (Node2, A3)\}$

Appendix B offers three more examples for the PODF algorithm. You can apply the algorithm and see whether you get the same result.

Chapter 5

Implementation

This chapter elaborates on the implementation of the methodology. More specifically, it discusses the implementation of a Python tool for this purpose. The name of the tool is the *"SFT to P&ID Converter" (SPC)*. Section 5.1 gives a high-level overview of this tool. The second section of this chapter discusses the requirements and the design. Lastly, this chapter focuses on the Model-View-Controller pattern [47, 9] and how the SPC Tool implements this.

5.1 High-Level Implementation Overview

The process within the SPC consists of seven steps, of which three are part of the methodology discussed in Chapter 4. Figure 5.1 shows a simplified overview. Firstly, steps A.1 and A.2 import an .RSA file to an SFT and pre-processes it. Then, it follows the three steps of the methodology. Lastly, steps B.1 and B.2 post-processes one or more P&IDs and export them into a .KBI file.

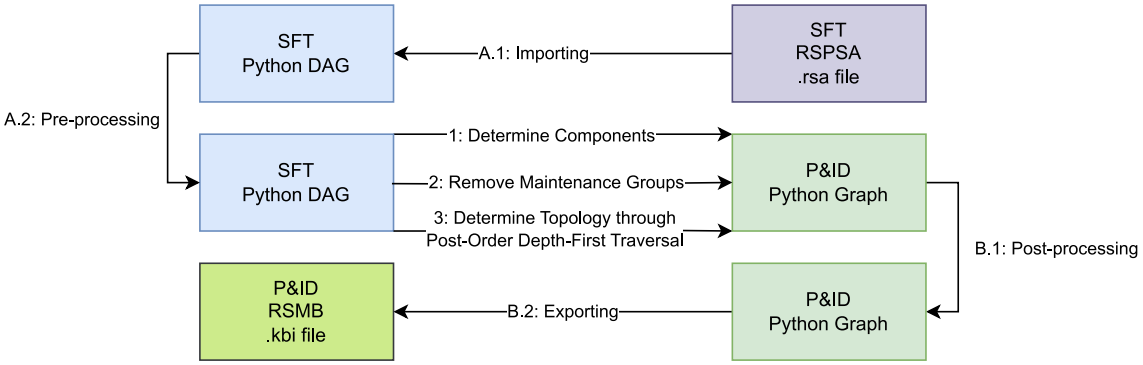


FIGURE 5.1: Simplified overview of the process within SPC

5.1.1 Directory Structure

Appendix G shows the directory structure the SPC's repository. There are five main directories within this project:

- *doc*
  Contains documents like figaro files, requirements, and design diagrams.

- *img*
  Contains images showing examples of SFTs and the EXPSA metamodel.

- *resources*
  Contains RSA files, KBI files, KBE files (see Section 6.2), numpy files and files containing Minimal Cut Sets (MCSs)

- *src*
  Contains the source code, structured through the Model-View-Controller Pattern (See Section 5.4).

- *test*
  Contains the code to test the source code.

5.2 Design & Requirements

Through a number of design choices and tooling decisions, the SPC uses the following patterns and/or tools:

- The Model-View-Control pattern as source code architecture.
  The reason for this is that it is a well-known architecture within the software development industry. Furthermore, it helps with separating responsibilities across different components within the SPC Tool.

- Python as the programming language.
  An alternative to Python is using Java or a framework based on this (e.g. the Spring framework[1]). However, since there is no need for features next to Python and because of recent familiarity with Python, the choice for it was sensible. A Pipfile manages the dependencies of the SPC.

- Git[2] as Version Control System (VCS) [82].
  Git is currently the industry standard for VCS, experienced personally as well as surveyed by Stack Overflow[3].

- GitLab as the remote VCS.
  The University of Twente has a GitLab server running for educational purposes, so it is sensible to also use it for the SPC Tool. Furthermore, there is recent experience with Python, GitLab and its CI/CD features.

- The Python package Tox[4] as environment orchestrator, focused on testing.
  In this case, the purpose was to learn about and familiarize with Tox and its relation with CI/CD.

- SonarQube[5], SonarLint and PyCharm as IDE and for code analysis.
  These tools offer functionality to improve code quality. A personal SonarQube server is available for this purpose, and previous experience with it is present.

[1] https://spring.io/
[2] https://git-scm.com/doc
[3] https://survey.stackoverflow.co/2022/#version-control-version-control-system
[4] https://tox.wiki/en/latest/index.html
[5] https://docs.sonarsource.com/sonarqube/latest/

Next to these choices and tools (and the reasoning behind them), Appendix H shows a part of the design with two Class Diagrams and a Call Hierarchy Graph. Lastly, SPC's repository contains documentation on a number of requirements. The requirements follow the MoSCoW structure.

5.3 Development Environment

This section discusses the development environment of the SPC. More specifically, it shortly explores Python, dependencies, CI/CD, versioning and testing.

Because Python 3.10.2 offers pattern matching by using match/case statements, the SPC uses this Python version. The root spc directory contains the following files:

- .gitignore
  A file defining what Git should ignore for Version Control.
- .gitlab-ci.yml
  This file defines a CI/CD pipeline for the remote GitLab repository. The pipeline configuration includes linters, maintainability metrics, testing and coverage reports, and static analysis through SonarQube.
- Pipfile
  This file shows the dependencies of the SPC.
- README
  This file contains essential information for repository users.
- setup.py
  The setup.py file configures basic information for publishing the SPC as Python package.
- sonar-project.properties
  This file configures variables a SonarQube instance can use to statically analyse the SPC project.
- tox.ini
  The tox.ini file contains configuration variables for testing the SPC tool in a variety of Python environments.

In the *src/pid_ dataset* directory, a main.py file exists for exploring the data structure of one of the 500 P&IDs discussed in Section 3.2. For this, the SPC requires the numpy version to be equal to or higher than v1.23.0 because of *distutils* getting deprecated[6].

One P&ID in this dataset exists of seven numpy files. These files contain meta-data, lines, symbols, words and links between the latter three entities. However, for the SPC to interpret this dataset, it needs the P&IDs to conform to the EXPSA metamodel. Subsection 8.4.1 discusses this further as future work.

The README.md in the root directory of SPC's repository explains how to use the SPC tool.

[6] https://numpy.org/doc/stable/release/1.23.0-notes.html

5.4 Model-View-Controller (MVC)

As mentioned earlier in this chapter, the SPC's architecture uses the MVC pattern. The first subsection focuses on the Model; the implementation of the SFT and P&ID datastructures as graphs. Then, two subsections focus on the View; importing SFTs from RSA files and exporting P&IDs to KBI files. Finally, one subsection elaborates on the Controller; the implementation of the methodology. The full source code is present in the repository of the SPC Tool[7].

5.4.1 Model: SFTs and P&IDs as Graphs

Within the model, there are three files:

- *src/model/graph.py*
  This file defines the data structures for a Graph, a Vertex and an Edge. It also contains a GraphError class for some basic exception handling.

- *src/model/sft.py*
  This file defines the data structures for an SFT, a Gate and a Basic Event, based on a Graph. It also contains two enums which define GateTypes and Fault Codes. The latter one is necessary to support maintenance group component recognition (see Section 4.3). Next to that, the SFTAttributes class defines the Top Event of an SFT and performs bookkeeping by splitting the list of vertices based on whether it is a Gate or a Basic Event.

- *src/model/pid.py*
  This file defines the data structures for a P&ID, a Link and a Component, based on a Graph. It also contains an enum which defines Component Types.

Figure H.2 shows the class diagram for the model.

5.4.2 View: Importing SFTs from RSA Files

The *src/model_io* directory is the "View" within the MVC architecture. This directory contains three files, of which *src/model_io/sft_io.py* supports importing SFTs from RSA files. Currently, it contains 20 methods to parse an RSA file line by line. The documentation in the source code describes the purpose of each method. Essentially, for each line, the first three characters define what to parse. This can be an SFT, BE, Gate, TE or an input of a Gate. Sometimes, Gate inputs contain elements which do not exist yet; while parsing it takes this into account. Parsing currently is a sequential process. Note that RSPSA exports RSA files in UTF-16 LE BOM encoding.

Figure H.3 shows the call flow graph for *sft_io.py*. In the repository, the *resources/SFTs/rspsa_expsa_examples.RSA* file is an example of a representative RSA file. Riskspectrum AB. distributes a document describing the structure of RSA files together with the RiskSpectrum PSA software.

5.4.3 View: Exporting P&IDs to KBI Files

The second file is *src/model_io/pid_io.py*, which supports exporting P&IDs to KBI files. Currently, it contains seven methods to generate a KBI file. Again, the documentation in the source code describes the purpose of each method. The process is similar to the process

[7] https://gitlab.com/wbos/spc/-/tree/main/src?ref_type=heads

for inferring P&IDs from SFTs. First, it generates the KBI lines which define a P&ID. Then, it generates the components and links respectively. For motor-driven pumps, there are EXPSA classes for each system; while generating it takes this into account.

When importing a KBI file, RSMB needs location data for the P&ID's components. In our implementation, the algorithm takes this into account by placing the components in three columns. For this, it generates coordinates (x, y) and transforms these into valid KBI syntax. It is also possible to provide coordinates to links in KBI files. This can improve the visualization of a P&ID and will be discussed in Section 8.3 Currently, the SPC Tool places both the P&IDs themselves, and a P&ID's components in three columns.

In the repository, the *resources/PIDs_result_generated/rspsa_expsa_examples.kbi* file is an example of a representative KBI file. Riskspectrum AB. distributes a document describing the structure of KBI files together with the RiskSpectrum ModelBuilder software.

5.4.4 Controller: Method Implementation

Next to implementing the methodology, the controller also pre-processes SFTs and post-processes P&IDs. The controller consists of four files:

- *src/controller/graph_controller.py*
  This file contains three methods for retrieving the union, intersection or substraction of two Graphs. It also provides a different way to retrieve the union of two Graphs' edges.

- *src/controller/sft_controller.py*
  This file contains 1) three methods for retrieving the union of two SFTs and 2) a class (SFTController) for pre-processing SFTs. It is impractical to use the union method of a generic graph because of the preference of explicitly typing classes, despite Python using dynamic typing. The SFTController performs two tasks related to pre-processing: 1) reducing SFTs (see also Subsections 8.2.2 and 8.2.3) and 2) checking SFTs for internal transfer gates.

  Internal transfer gates refer to other SFTs about the same system (e.g. the *Residual Heat Removal System* in Subsection 7.2.1). This is different from external transfer gates. These are gates referring to SFTs about a different system (e.g. the *Emergency Core Cooling System* referring to the *Component Cooling Water System* in Subsection 7.2.3).
  For internal transfer gates, the SFTController takes the union of two SFTs when the SFTs are about the same system. Subsection 8.2.6 discusses options for dealing with external transfer gates.

  Figure H.1 shows the class diagram the SFTController.

- *src/controller/pid_controller.py*
  This file contains one method for retrieving the union of two P&IDs. Again, it is impractical to use the union method of a generic graph because of the preference of explicitly typing classes, despite Python using dynamic typing.

- *src/controller/sft_to_pid.py*
  This file contains a class (SFTToPIDConverter) which implements the methodology described in Chapter 4. One method (*infer_pids_from_sfts*) performs post-processing of P&IDs by retrieving the union of two P&IDs. The implementation only retrieves the union of these P&IDs when it detects that a P&ID already exists

for the same system. It takes Python's call by object reference system into account. However, for future work (see also Section 10.4), an implementation needs to take the following aspects into account:

- Consider whether a change in one of the original P&IDs should also change the merged P&ID and alter the implementation accordingly.
- The amount of dynamic programming and bookkeeping involved. Consider whether the implementation of the P&ID keeps track of extra data next to the formal definition. Then, alter the implementation accordingly.

Subsection 4.1.1 elaborates further on merging two P&IDs.

Chapter 6

Verification and Validation

This chapter elaborates on how to verify and validate that the algorithm works. More specifically, this chapter elaborates on:

- testing the correct implementation of the algorithm explained in Chapter 4 using the four examples in Figures 4.2 and 4.3 (Subsection 6.1);
- the process to validate that the algorithm works (Subsection 6.2);

Therefore, this chapter starts with a section on testing relevant parts of the SPC tool. This section will have two subsections focused on unit tests and integration tests. The second section focuses on validating the methodology.

6.1 Implementation Verification

This section focuses on checking that SPC's implementation of our methodology is correct via testing. Both unit and integration tests exist; the two subsections elaborate on them separately. In the SPC's GitLab repository, the CI/CD pipeline does include a job to run all tests and generate a coverage report[1].

The system tests, which are part of the validation process, are also a way to verify a correct implementation. Hence, running the unit tests, integration tests and system tests locally shows a full picture of the tests' coverage.

6.1.1 Unit Tests

For each method in the source code, a corresponding unit test exists. Now, there are multiple methods which call upon a chain of other methods. One can argue a unit test then becomes an integration test. However, the aim here is to test the method under test instead of all other methods.

6.1.2 Integration Tests

Since the focus is on verifying a correct implementation of the methodology, the *test/controller/test_sft_to_pid* file uses the examples discussed in Section 7.1 as integration tests. To also account for the EXPSA naming convention, the examples contain BE IDs which conform to this convention. In this way, it was also possible to — in a relatively early stage — detect a correct implementation for a case study (see Section 7.2) as well.

[1] https://gitlab.com/wbos/spc/-/pipelines

6.2 Methodology for Validation

This section will focus on the process to validate our method. The idea here is to test the whole workflow so that it is possible to compare an original P&ID with a resulting, inferred P&ID. The reasons to use system tests are two-fold:

- System tests can be done by one individual
- The resulting P&IDs are directly usable for RSMB stakeholders. This will also enable potential A/B Testing [81] in the future (see Section 8.4.2).

An alternative to system tests is to approach P&ID comparison as a Graph Isomorphism problem. Subsection 8.3.2 elaborates more on this approach.

An overview of the system is given in Appendix C. The sections in this chapter use terminology of the mentioned system models to explain the approach to validating the algorithm. Note that RSMB can export both .KBI and .KBE files. For the purposes of the SPC's implementation and the algorithm's validation, both files work on different abstraction levels. A .KBE file roughly represents an RSMB Study. A .KBI file contains the P&IDs of an RSMB Study. The difference between .KBI and .KBE is less relevant since the SPC can only export to .KBI files. Nevertheless, the README of the SPC's repository gives slightly more information about the difference[2].

Figure 6.1 shows an overview of the process. For a system test, the idea is to start with a set of P&IDs and get automatically generated SFTs from RSMB and RSPSA. Then, the SPC needs RSA files exported from RSPSA, which contain the SFTs. The SPC converts these to KBI files containing the resulting P&IDs. RSMB can then import these files. Since the set of original P&IDs and the set of resulting P&IDs are bijective, it is possible to compare them based on the ID of the P&ID. Hence, there are six steps:

1. Making/retrieving P&IDs in RSMB, including failure modes (which are the equivalent of TEs).
2. Generating SFTs and exporting these to RSPSA.
3. Exporting the SFTs to RSA files.
4. Using the SPC to convert the SFTs back to P&IDs and retrieve KBI files (see Chapter 5).
5. Importing the KBI files into RSMB.
6. Visually comparing the P&IDs from steps one and five.

The actions necessary for step five are the same as the ones for step one. A manual for the first five steps is available on the SPC GitLab[3]. An alternative to a visual comparison is to solve the graph isomorphism problem for P&IDs. A colouring algorithm where the CLAs are treated as colours could be a solution for this (see Section 8.4.1). The validation process only contains a visual comparison because of time constraints.

[2] https://gitlab.com/wbos/spc
[3] https://gitlab.com/wbos/spc/-/blob/main/doc/Validation-Process-Manual.pdf

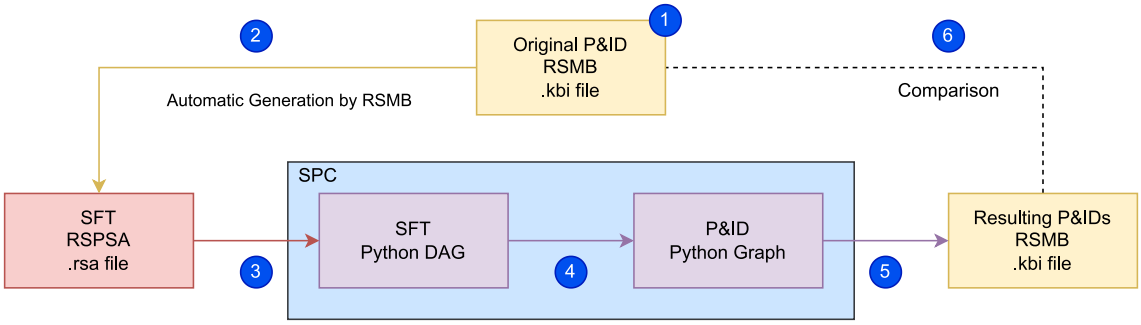


FIGURE 6.1: Methodolody Validation Process Overview

6.2.1 Visually comparing P&IDs

This subsection will elaborate on comparing two P&IDs visually. After importing the KBI files back into RSMB, it is possible to visually compare the original P&ID to the resulting one. This is possible since the input and output P&IDs are still relatively small. Furthermore, a formal and extensive validation takes significantly more time. Lastly, it is more important to validate that the method is sound for basic, visual elements in the transformation instead of validating its completeness.

The goal of comparing the original and resulting P&IDs is to answer the following questions:

1. Is the number of components the same?
2. Is the number of physical components the same?
3. Are the Component Labels (CLAs, see Definition 2.2.1) of the original P&ID present in the imported P&IDs?
4. Are the physical components which are in parallel in the original P&ID, also in parallel in the imported P&ID and vice versa?
5. Are the physical components which are in series in the original P&ID, also in series in the imported P&ID and vice versa?
6. Are there any unexpected components in the resulting P&ID?
7. Are there any unexpected links in the resulting P&ID?

Definition 2.2.1 elaborates on properties which are necessary to further formalize the comparison. The following Boolean statements and properties then tell something on the validity of our method:

1. $|C_{orig}| = |C_{result}|$
2. $|PhysicalComponents(PID_{original})| = |PhysicalComponents(PID_{result})|$
3. $CLAs(PID_{original}) \subseteq CLAs(PID_{result})$
4. $\forall r, s \in PhysicalComponents(PID_{original}), r \neq s \wedge r \nRightarrow s$
   $\exists a, b \in PhysicalComponents(PID_{result}), a \neq b \wedge a \nRightarrow b,$
   $a.CLA = r.CLA \wedge b.CLA = s.CLA$

5. $\forall r, s \in PhysicalComponents(PID_{original}), r \neq s \wedge r \Rightarrow s$
   $\exists a, b \in PhysicalComponents(PID_{result}), a \neq b \wedge a \Rightarrow b,$
   $a.CLA = r.CLA \wedge b.CLA = s.CLA$

6. $\forall cla \in CLAs(PID_{result}) \exists cla \in CLAs(PID_{original})$

7. $\forall i \in Links(PID_{result})\ \exists j \in Links(PID_{original}) :$
   $i.CA.CLA = j.CA.CLA\ \wedge\ i.CB.CLA = j.CB.CLA$

Note that statements four, five and seven only look at the adjacency of two vertices. Section 8.4.3 discusses validating P&ID similarity through Graph Isomorphism, which includes connectedness of two vertices.

Chapter 7

Results

This chapter discusses the results of comparing original P&IDs and the resulting P&IDs of the methodology validation process. More specifically, this chapter elaborates on:

- P&IDs based on the examples used when verifying the implementation and (Subsection 7.1);
- P&IDs based on a case study provided by RiskSpectrum (Subsection 7.2).

The emphasis in these sections will be on:

- Describing the original P&IDs;
- Discussing aspects related to applying the validation process on these P&IDs and;
- Comparing the original P&IDs with the resulting P&IDs.

7.1 Simple Examples

After applying the validation process to the examples of Figures 4.2 and 4.3, there exist four original P&IDs, SFTs, RSA files, KBI files and resulting P&IDs (see also Figure 6.1). This section will compare the original and resulting P&IDs for each example. Each subsection below covers one example. The KBE, RSA and KBI files can be found in the SPC repository in the *resources/PIDs_original*, *resources/SFTs* and *resources/PIDs_results_generated* directories[1].

For the examples of Figures 4.2 and 4.3, no original P&IDs existed yet in RSMB. Hence, it was necessary to create these, such that the P&IDs would generate SFTs corresponding to ones in these two figures. Since no automatic process exists for this, this was done manually (see the aforementioned manual on the SPC's GitLab repository[2]). Furthermore, the examples consist of simple SFTs. For Examples 1, 2 and 3, manually creating an original P&ID was relatively simple because of this; exactly one RSMB study exists with exactly one original P&ID. This P&ID is the original input P&ID.

Unfortunately, it was impossible to find a suitable P&ID which would generate the SFT of Example 4 within a reasonable timeframe. The main reason this takes more time is because of the generation process of RSMB. This process is different from earlier beliefs. Hence, Subsection 7.1.4 uses an expected P&ID as original for the comparison.

[1] https://gitlab.com/wbos/spc/-/tree/main/resources
[2] https://gitlab.com/wbos/spc/-/blob/main/doc/Validation-Process-Manual.pdf

Note that in all examples' original P&IDs, the *"RHR_ T1_ FAIL"* and *"leakage_ group"* nodes visualize the location where the eventual TE can fail. Our examples fail when no fluid reaches the end *RHR_ node*.

Since there is only one SFT for each P&ID, there is also only one SFT in each RSMB study and hence, in each RSPSA Project. There are two important differences between the RSMB and RSPSA SFTs. The first difference is visible in the labels of the Gates and BEs. RSPSA SFTs do follow the EXPSA metamodel naming scheme and emphasize TEs with a grey background. The second difference is visible in the description: there is more information about how the component fails.

7.1.1 Example 1

Figure 7.1 shows the original and resulting P&IDs of Example 1. Figure 7.2 shows the generated and exported SFTs from RSMB and RSPSA for Example 1.

When comparing the P&IDs of Figure 7.1, there are two abstract components missing, including their incident edges. So, the comparison violates the third property described in Subsection 6.2.1. However, when only looking at physical components, the third property is still valid. This is also the case for the other examples in the following two subsections for examples two and three.

The resulting P&ID also shows the CLAs of the node components being capitalized. Since node components are abstract components, capitalization of CLA matters less. Nevertheless, the implementation needs to check whether this conforms to the EXPSA naming convention.

For the seven properties, this means the comparison violates properties one and three.

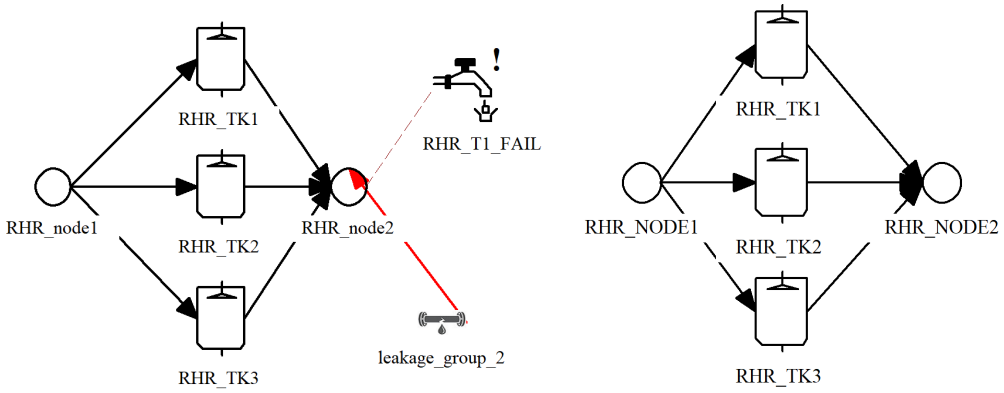


FIGURE 7.1: Original and Resulting RSMB P&ID of Example 1

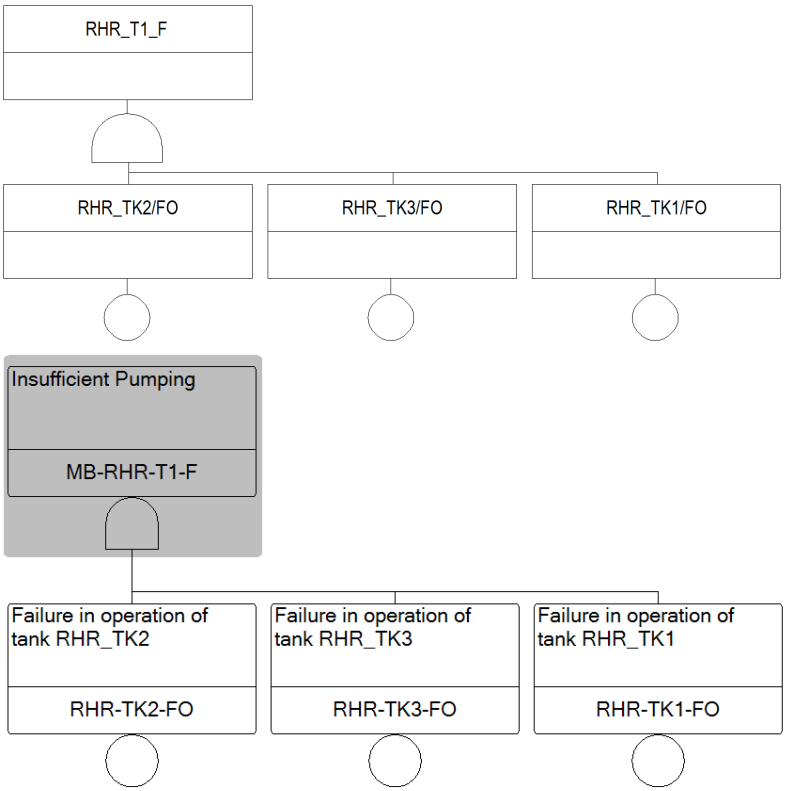


FIGURE 7.2: RSMB and RSPSA SFTs of Example 1

7.1.2 Example 2

Figure 7.3 shows the original and resulting P&IDs of Example 2. Figure 7.4 shows the generated and exported SFTs from RSMB and RSPSA for Example 2.

When comparing the P&IDs of Figure 7.3, there is one major difference visible (next to the one already described in the previous subsection). The order of the components now starts with the check valve, followed by the tank and then the pump. This is because of two reasons. Firstly, the SPC tool works with lists and preserves the ordering. When components are in a certain order under an OR gate, it assumes that this is also the order in which the components should be placed in series within the P&ID. Secondly, RSMB generates the SFT such that the BE of the check valve is on the left, the BE of the tank in the middle and the BE of the pump on the right. Subsection 8.2.1 discusses further on this.

For the seven properties, this means the comparison violates properties one, three, five and seven.

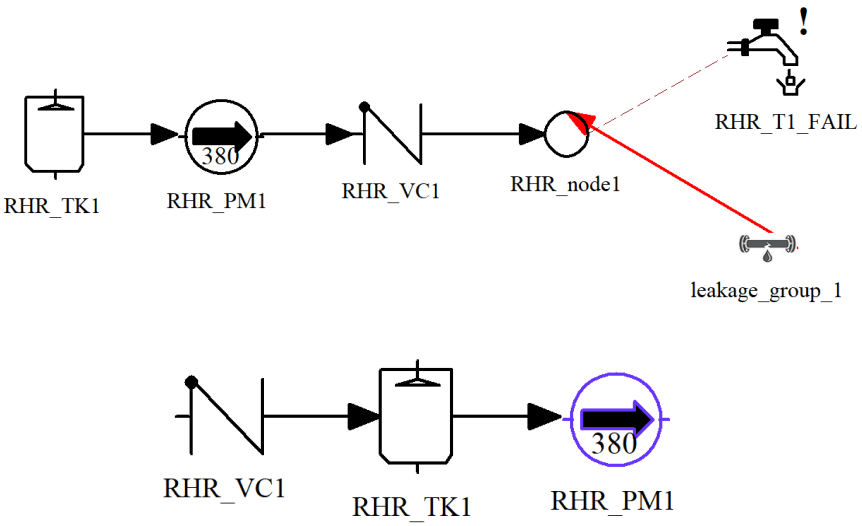


FIGURE 7.3: RSMB Original and Resulting P&IDs of Example 2

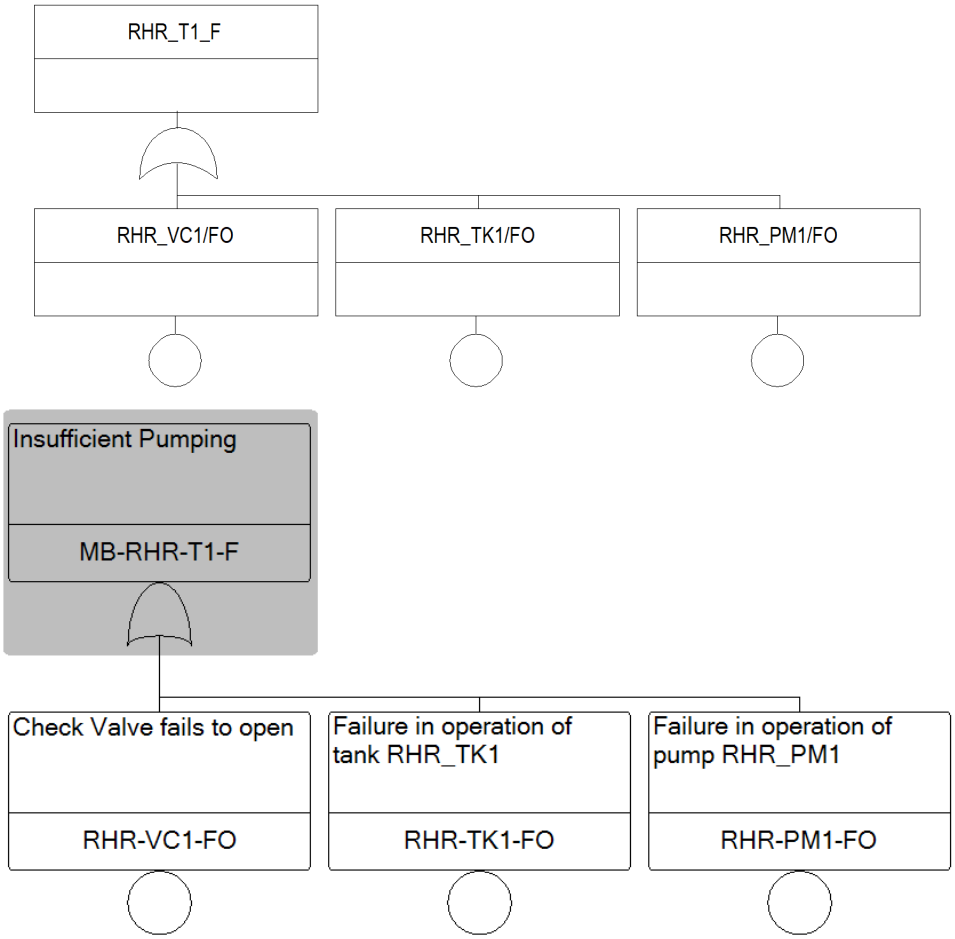


FIGURE 7.4: RSMB and RSPSA SFT of Example 2

7.1.3 Example 3

Figure 7.5 shows the original and resulting P&IDs of Example 3. Figure 7.6 shows the generated and exported SFTs from RSMB and RSPSA for Example 3.

When comparing the P&IDs, the issues of the first two examples are also visible here. The same abstract components are missing in the resulting P&ID on the right side. Next to that, the change in ordering is present again in the serial substructure.

For the seven properties, this means the comparison violates properties one, three, five and seven.

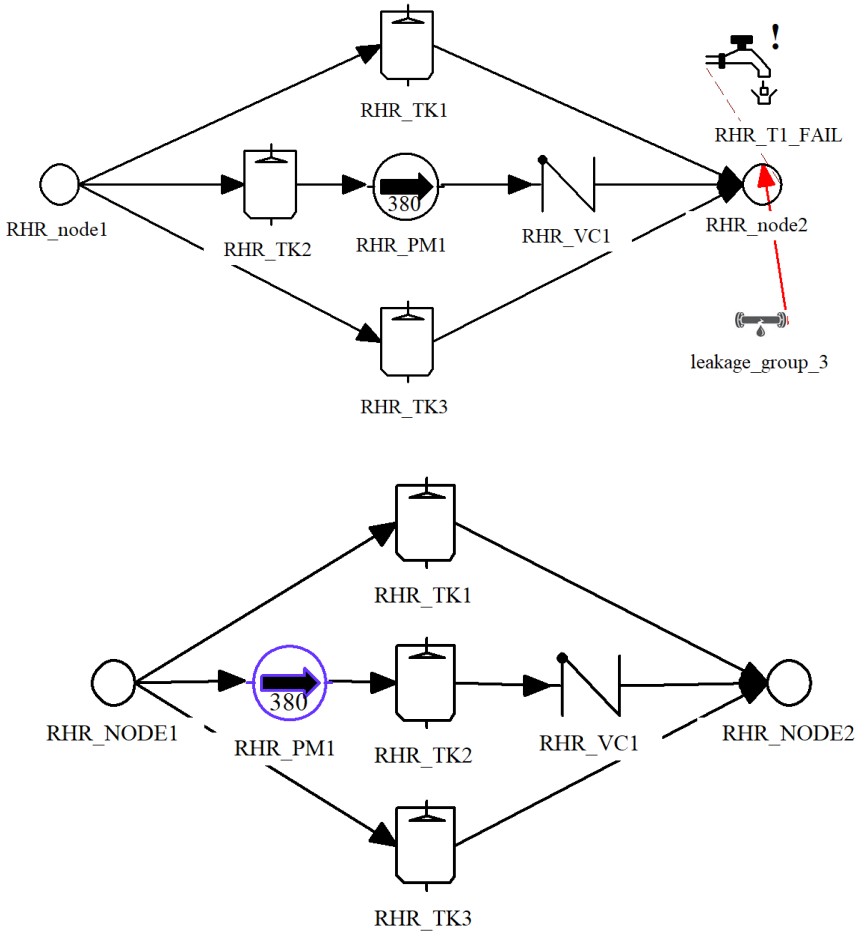


FIGURE 7.5: RSMB Original and Resulting P&IDs of Example 3

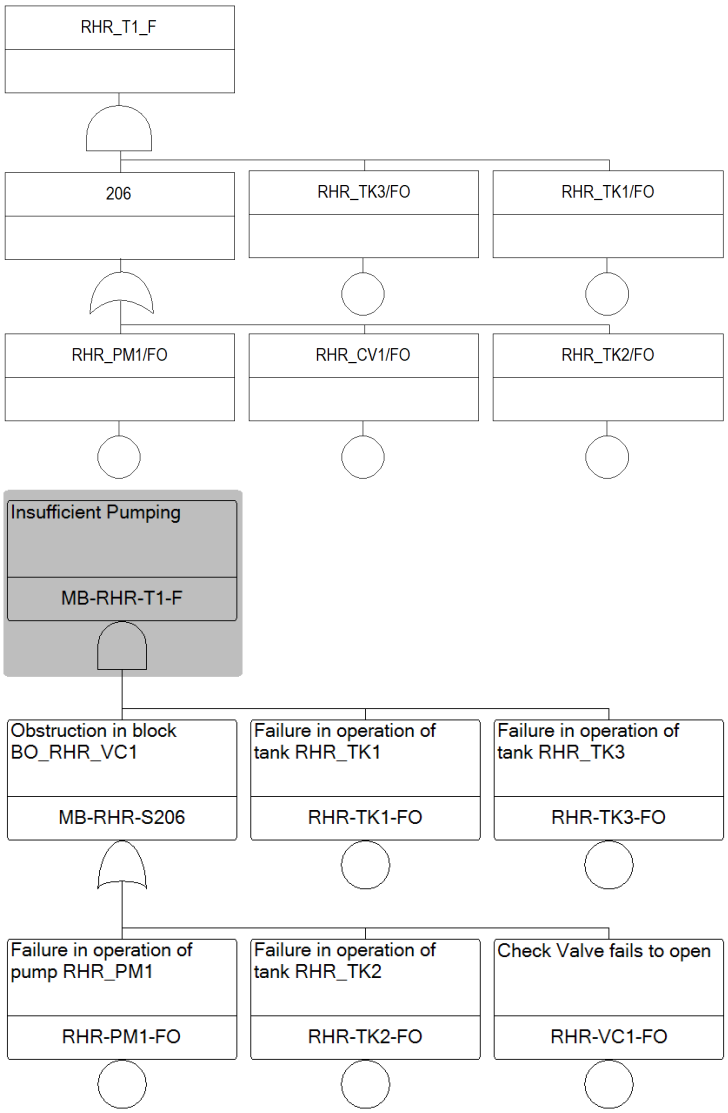


FIGURE 7.6: RSMB and RSPSA SFT of Example 3

7.1.4 Example 4

As mentioned at the start of this Section, it was impossible to find a suitable P&ID which would generate the SFT of Example 4 within a reasonable timeframe. Instead, it is possible to swap around the AND and OR gates in the RSPSA SFT of Example 3 within RSPSA. This then results in the SFT of Example 4, without changing the order of components w.r.t. the SFT of Example 3. Now, the SPC can work with this SFT to retrieve the resulting P&ID. Unfortunately, this means there is no original P&ID. Instead, this subsection compares the resulting P&ID to an expected P&ID.

Figure 7.7 shows the expected and resulting P&IDs of Example 4. Figure 7.8 shows the RSPSA SFT for Example 4.

Just like the other examples, the same two issues are visible. Now, the ordering is that the parallel structure comes at the start of the overall serial structure, instead of in the middle.

For the seven properties, this means the comparison violates properties one, three, five and seven.

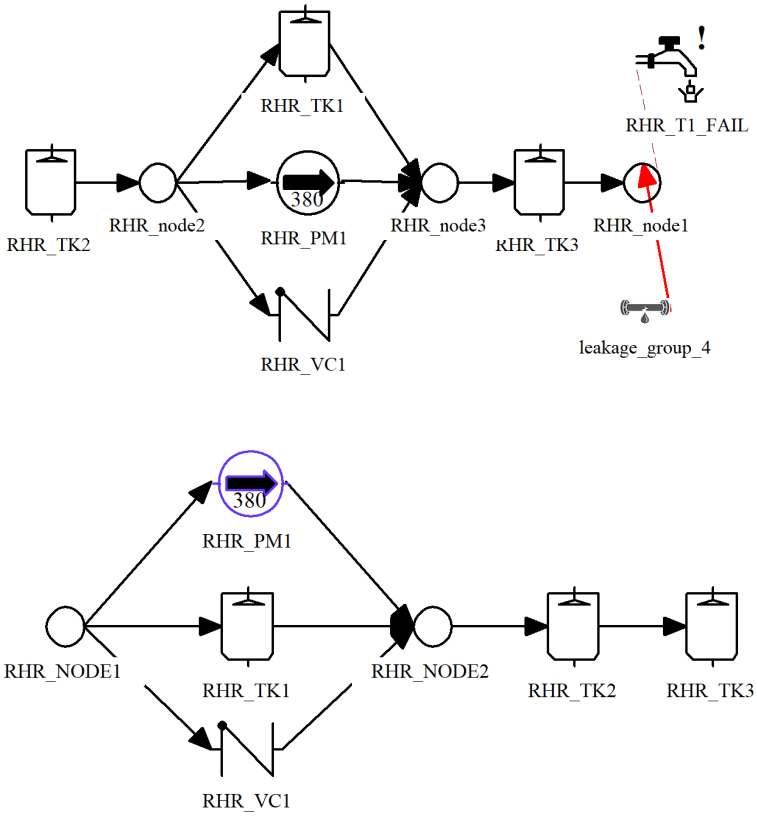


FIGURE 7.7: RSMB Original and Resulting P&IDs of Example 4

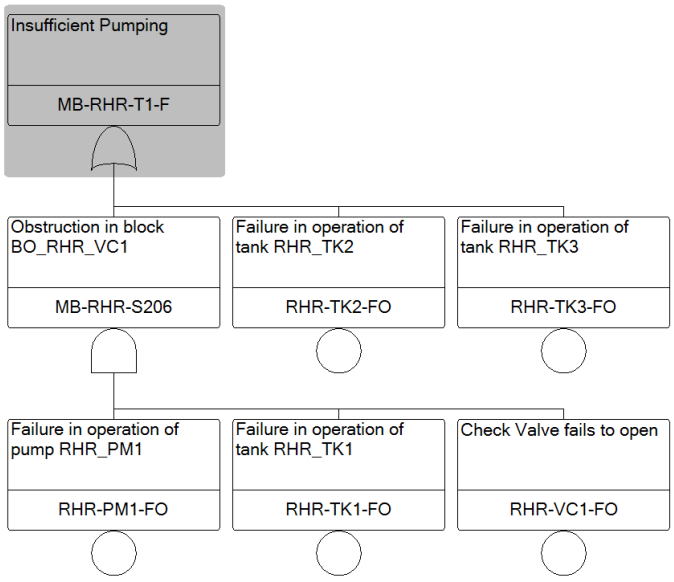


FIGURE 7.8: RSPSA SFT of Example 4

7.2 Case Study

This section focuses on a larger case study as input. The main emphasis is on the difference with the four examples of the previous section.

Riskspectrum has an RSMB project with two studies based on the EXPSA metamodel. This project represents a PSA model of a simplified Boiling Water Reactor (BWR) plant. Johan Gustafsson [31, Figure 9] describes the PSA model in their Masters' thesis back in 2012. The model contains the following features;

- Individual trains, which consists of a series of components
- Redundant (parallel) trains
- Dependencies to support systems and electric supply
- Common Cause Failures (CCFs)
- Maintenance and Operator Actions
- Configuration Management

Because the model contains these features, they are representative for larger, real-life models. Hence, it is possible to use the model:

1. For demonstrating the capabilities of RSMB and RSPSA
2. As case study for research purposes.

The P&IDs in the RSMB project show the structure of the thermohydraulic systems within the example plant. Since the EXPSA metamodel already defines failure mode(s) of components, P&IDs contain this information by default. For full system failures, abstract components are visible, similar to the ones of the previous section. The case study contains a simplified model (and hence P&IDs) since:

1. The original P&IDs of nuclear power plants are classified because of national security, and;
2. Simpler P&IDs are easier to explain and reason about. It is easier to remember a shorter list of BEs and/or Components

As a consequence of this:

1. P&IDs with more components, links and failure modes use similar structures, and;
2. The scalability of the algorithm for bigger P&IDs is out of scope;

The model consists of two RSMB studies with seven P&IDs in total. Two KBE files (*rsmb_expsa_demo.kbe* and *rsmb_expsa_mfw_demo.kbe*) for these studies are available in the SPC repository in the *resources/PIDs_original* directory[3].

Subsections 7.2.1 until 7.2.3 and Appendix D show all RSMB P&IDs of the case study. The P&IDs show the following systems:

- *Main Feed Water System (MFW)*
  Injects coolant into the inner part of the reactor (the Reactor Pressure Vessel).

[3] https://gitlab.com/wbos/spc/-/tree/main/resources/PIDs_original

- *Emergency Feed Water System (EFW)*
  Injects coolant into the Reactor Pressure Vessel (RPV) when the MFW system fails. The EFW uses water from a separate Demineralized Water Storage container (DWST) as the coolant source.

- *Depressurization System (DPS)*
  The EFW and MFW systems inject coolant under high pressure. When both fail, the cooling circuit needs to be depressurized for the next safety systems to work.

- *Emergency Core Cooling System (ECC or ECCS)*
  Injects coolant into the RPV when both the MFW and EFW systems fail. It usually requires depressurization of the cooling circuit by the DPS to work.

- *Residual Heat Removal System (RHR)*
  Removes heat from the coolant which the DPS and ECC systems use. This coolant is water taken from a pool present in the outer part of the reactor (the Reactor Containment).

- *Component Cooling Water System (CCW)*
  Support system for the EHW, EEC and RHR systems to cool components.

- *Service Water System (SWS)*
  Support system for the CCW system to provide water.

There is another system present in this case study, for which no original graphical system description exists within the RSMB project. This system is the *AC Power System (ACP)*. Usually, Single-Line Diagrams (SLDs) visually show the electrical systems, instead of P&IDs. Section 10.4 shortly discusses SLDs as potential future work.

For the original P&IDs, it is possible to generate 30 SFTs, export them to RSPSA and generate the RSA files. However, the SPC uses an RSPSA Project with a different set of SFTs for internal system tests. This set is based on the same case study and contains only 20 SFTs. When comparing the resulting P&IDs of both inputs, it was initially impossible to compare them. The reasons for this were:

1. Component labels and BE fault codes diverged from the naming convention.
   Two examples are 1) using *ECCS* as system name instead of *ECC* and 2) using *MA* as maintenance group failure code instead of *M*.

2. There were more failure modes defined within components.
   This still causes some CCW, ECC, EFW and SWS systems' components to have fluid pipes with the same source and destination component (see Subsection 8.2.2). This also probably causes a difference in the amount of generated SFTs and their sizes.

3. Because there were more failure modes, it is possible to define more SFTs of one system, with different TEs for different parts of a system.
   This was the case for the ECC, EFW and RHR systems.

The workaround for the first issue was to simply change the component labels so that they conform to the naming convention. For the third issue, it was sufficient to simply delete the extra SFTs. Each of the ECC, EFW and RHR systems had two redundant SFTs. After this, the structure of the resulting P&IDs (see Appendix F) is recognizable and comparable with the ones of the RSPSA EXPSA case study. There are still some differences between the results. These can be found in:

- The order of components in the CCW system (see also Section 8.2.1).
- A missing substructure in the ECC system (see also Section 7.2.3).
- A couple of redundant extra nodes and fluid pipes adjacent to these nodes in the MFW system. The expectation is that this is because of the unaddressed second issue.

Because of these differences and time limitations, this section continues elaborating on the SFTs and resulting P&IDs based on using the RSPSA Project with 20 SFTs as input for the SPC. Note that RiskSpectrum manually created this set of SFTs based on the original P&IDs. As a side effect, this represents the actual problem (see Chapter 1) more closely.

The rest of this section focuses only on the Residual Heat Removal System (RHR), Main Feed Water System (MFW) and Emergency Core Cooling System (ECC) systems. This is because these systems show the most interesting comparisons between P&IDs at the end of this subsection. More specifically, the RHR P&IDs will be fairly equivalent, the MFW system is interesting because of a VOT gate, and the ECC P&IDs will show an interesting difference. The original P&IDs, SFTs and resulting P&IDs of the other systems can be found in Appendices D and E. Furthermore, the RSA files of the original 30 SFTs (*rsmb_expsa_demo.rsa* and *rsmb_expsa_mfw_demo.rsa*) are available in the SPC repository in the resources/SFTs directory[4]. There, the RSA file of the 20 SFTs of the RSPSA project (rspsa_expsa_examples.rsa) is also available. Because the main difference between RSMB SFTs and RSPSA SFTs is clear from the previous section, this section will only focus on the RSPSA SFTs. Note that for the RHR, MFW and ECC systems, there are three SFTs each.

7.2.1 RHR

The RHR system consists of thirteen components and ten fluid pipes. Figure 7.9 shows the original and resulting P&IDs for the RHR system. The first P&ID is the original P&ID, displayed above the resulting P&ID.

[4] https://gitlab.com/wbos/spc/-/tree/main/resources/SFTs

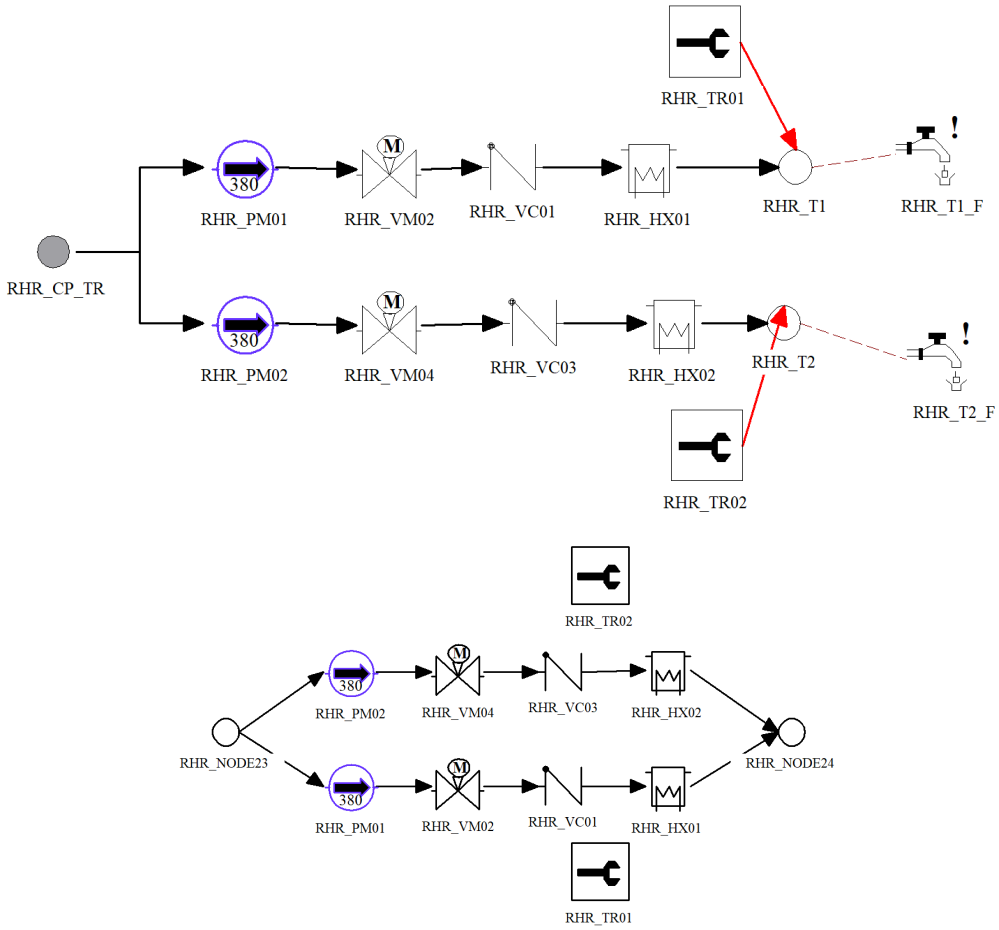


FIGURE 7.9: Original and Resulting P&IDs of the RHR System

When comparing the original and resulting P&IDs of the RHR system, the issue of missing abstract components also persists here. This also happens for the MFW and ECC systems in the following subsections. An exception is visible for the maintenance groups in the resulting P&ID for the RHR system. The method does recognize this type of abstract component based on the fault code in the BE ID.

These maintenance group components are, however, not linked to their respective series of components. This is also visible in the last node; instead of two nodes, there is now one. Where the original P&ID clearly shows that each series has separate failure tests at the end nodes, this information is missing in the resulting P&ID. Subsection 8.2.3 elaborates further on this.

For the seven properties, this means the comparison violates properties one, three and seven.

Figure 7.10 show the SFTs for the RHR system. The first SFT with TE *@RHR-1* has an AND gate with two children @RHR-4 and @RHR-3.

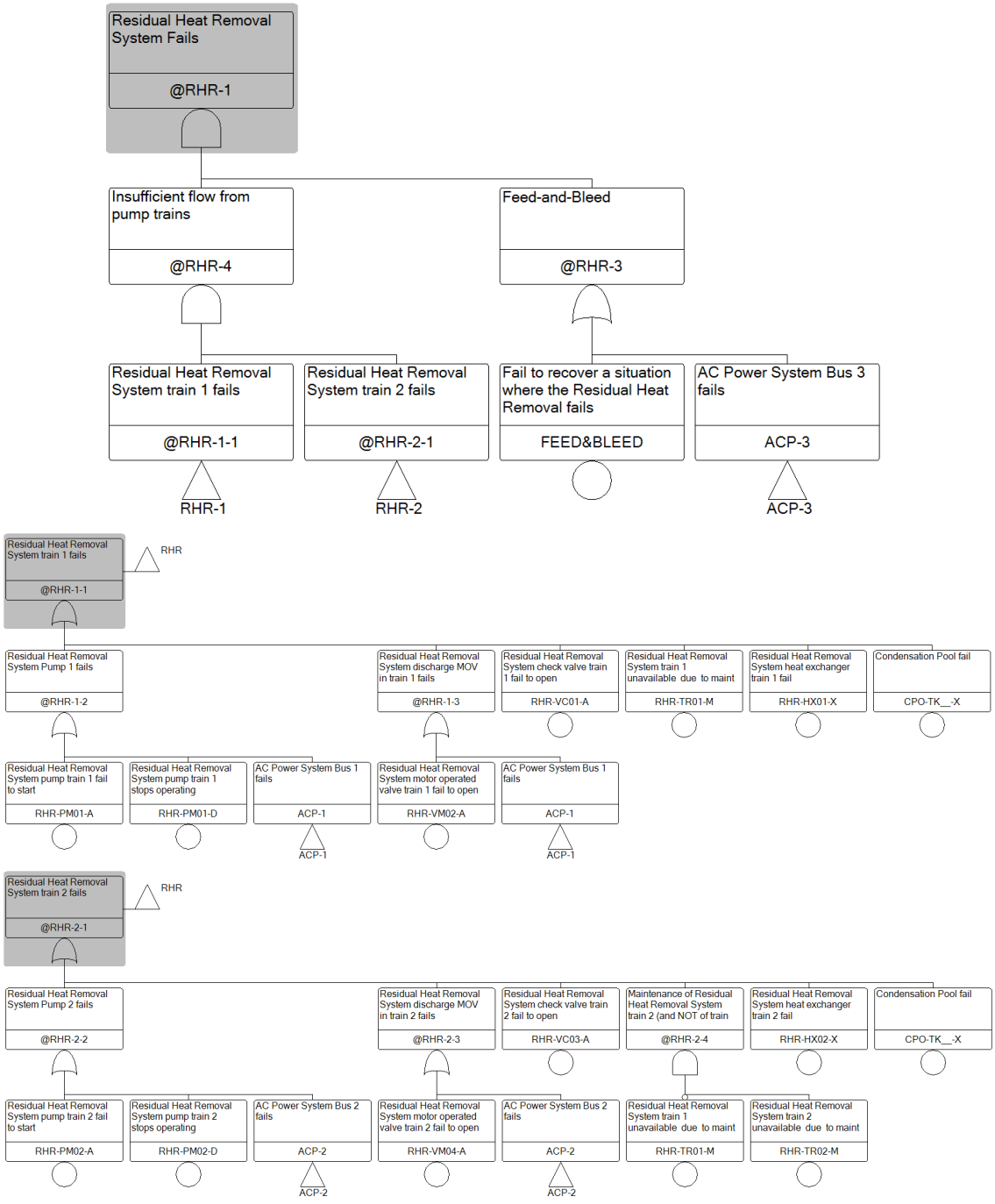


FIGURE 7.10: SFTs of the RHR System

Below the @RHR-3 OR gate, there are only two children. The first child is a BE, which is a full system recovery failure nonspecific to one component. The second child is a transfer event to the ACP system which – as mentioned earlier in this section – is out of scope.

Below the @RHR-4 AND gate, there are two transfer events to the other SFTs with TEs @RHR-1-1 and @RHR-2-1. Both SFTs are similar in structure with exception to the BEs and Gate related to the maintenance group components. The first child of both SFTs consists of an OR gate to two BEs. Each SFT only has the two BEs related to

one of the pumps, and both BEs only have a different fault code. The other children either are BEs related to component failures or gates leading to such BEs. Note that the SFT starting with TE @RHR-1-1 only has BEs concerning the components *RHR_PM01*, *RHR_VM02*, *RHR_VC01*, *RHR_HX01* and *RHR_TR01*. Furthermore, the SFT starting with TE @RHR-2-1 only has BEs concerning the components *RHR_PM02*, *RHR_VM04*, *RHR_VC03*, *RHR_HX02*, *RHR_TR01* and *RHR_TR02*. Hence, when ignoring the maintenance group components, each SFT covers a set of components which is in series in the original P&ID.

7.2.2 MFW

The MFW system consists of nine components and ten fluid pipes. Figure 7.11 shows the original and resulting P&IDs for the MFW system. The first P&ID is the original P&ID, displayed above the resulting P&ID.

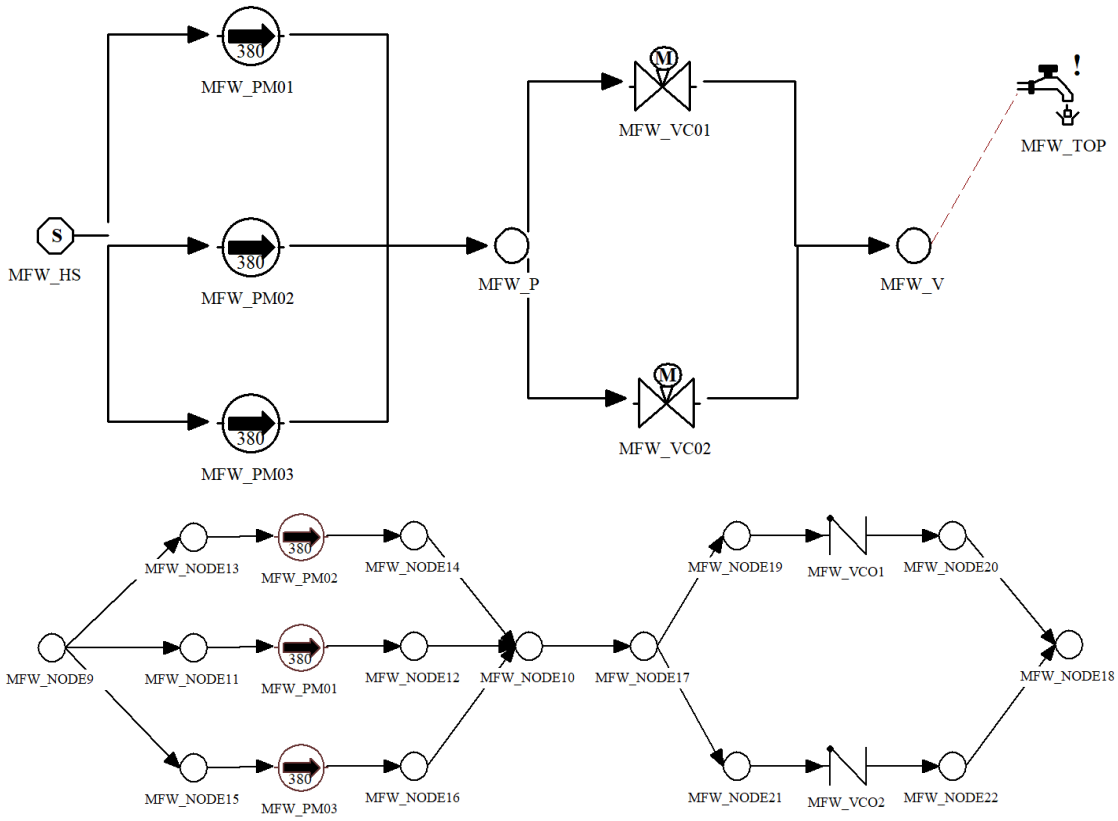


FIGURE 7.11: Original and Resulting P&IDs of the MFW System

For the MFW system, note that RSMB can the failure criteria in a P&ID such that two out of three pumps must fail. This can be done by altering the *MFW_P* node's configuration. As a result, there will be a VOT gate in the SFTs for the MFW system.

Note that the original P&ID uses electrical valve symbols instead of check valve symbols: this is a graphical error. Since the CLA still uses VC, the resulting P&ID is correct in determining the component type.

When comparing the original and resulting P&IDs of the MWF system, there are significantly more nodes visible in the resulting P&ID. This happens because two BEs,

which map to the same component, are placed below an AND gate. The algorithm hence sees this single component as "being in parallel" with itself. Subsection 8.2.2 elaborates more on this issue.

Again, one abstract component is missing, in line with other comparisons. For the seven properties, the comparison violates properties one, three, six and seven.

Figure 7.12 shows the SFTs for the MFW system. The first SFT with TE *@MFW-1* has an OR gate with two children @MFW-PUMP and @MFW-ISOL. These children are transfer events to the other two SFTs.

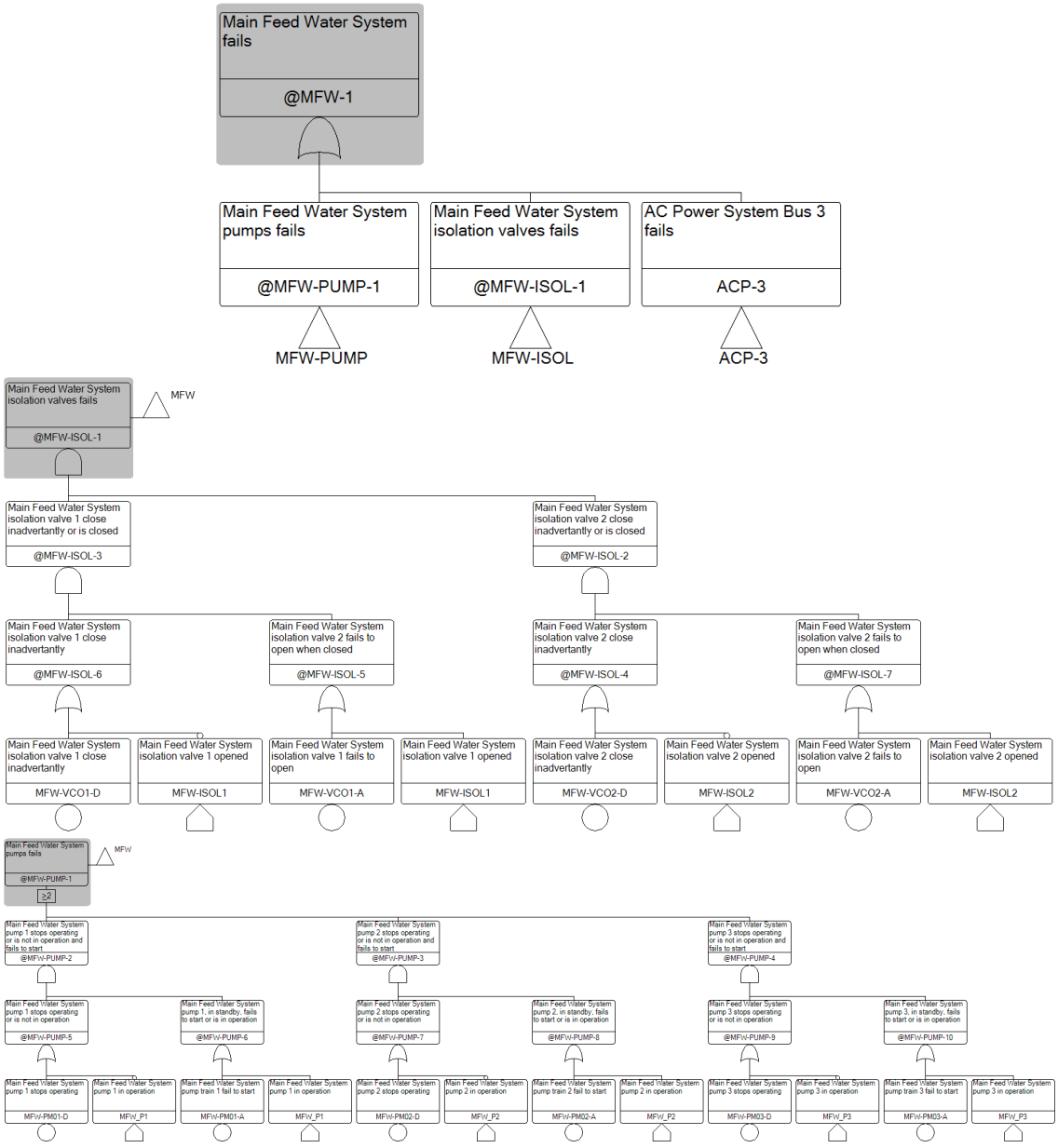


FIGURE 7.12: SFTs of the MFW System

The SFT with TE *@MFW-ISOL-1* has an AND gate with two more OR gates as children, which each has in turn two OR gates as children. Each OR gate has a BE and a new type of node as a child. This type of child is a House Event. House Events can be used to configure a system, such that only a part of the SFT is relevant in a given state

of the system. For the MFW system, this is out of scope. In the next subsection, House Events *do* have an impact on the resulting P&ID.

The SFT with TE *@MFW-PUMP-1* has a VOT gate where 1/3 children can fail. Each child is an AND gate with two more OR gates as children. The structure here is similar to the *@MFW-ISOL-1* SFT and now covers three pumps instead of the two check valves.

Currently, the extra safety information of the VOT gate is ignored. No visible component shows that 1/3 children can fail. However, it would be beneficial if this information can be kept. Subsection 8.2.4 discusses this improvement.

7.2.3 ECC

The ECC system is very similar to the RHR system. Figure 7.13 shows the original and resulting P&IDs for the ECC system. The first P&ID is the original P&ID, displayed above the resulting P&ID.

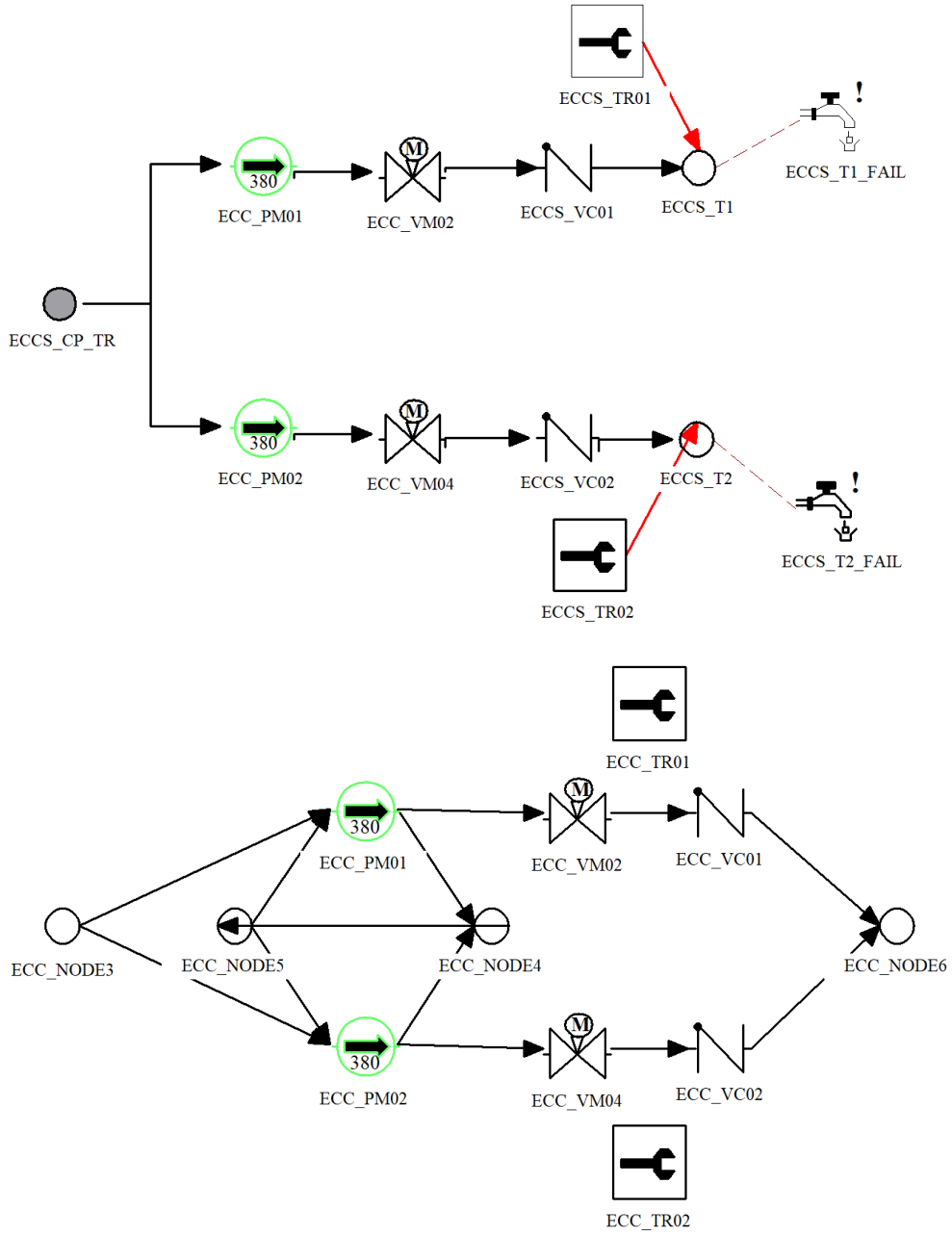


FIGURE 7.13: Original and Resulting P&IDs of the ECC System

There is one big difference between the original P&IDs of the ECC and RHR systems. The difference lies in the missing heat exchanger components. Furthermore, the failure modes are different, which is visible in the SFTs.

In the resulting P&ID, the left part shows two parallel motor-driven pumps *ECC_PM01* and *ECC_PM02*. Note that *ECC_NODE5* and *ECC_NODE4* show a smaller parallel structure within the bigger one started by *ECC_NODE3* and ended by *ECC_NODE6*. Furthermore, a link goes from *ECC_NODE4* to *ECC_NODE5*. After the motor-driven pumps, there are two motor operated valves (*EXPSA_MO_valves*) and two check valves. Again, two components of the same type are in parallel, just like the pumps. Note that there are now two series of a pump and two valves. Just like the RHR system, two

maintenance groups are visible.

When comparing the original and resulting P&IDs of the ECC system, there are three things standing out. The first two things are repeating issues:

- The same abstract components are missing in the resulting P&ID — again.
- The resulting P&ID shows the maintenance groups just like the one of the RHR system. This also comes with the same issue of missing nodes, as well as missing links between the maintenance groups and these nodes.

Lastly, the resulting P&ID contains extra nodes and links around the pumps. This is where House Events have an impact on the resulting P&ID.

Figure 7.14 shows the SFTs for the ECC system. These SFTs are similar to the SFTs of the RHR system. The first SFT with TE *@ECC-1* has an OR gate with two children @ECC-3 and @ECC-9. These children are both AND gates.

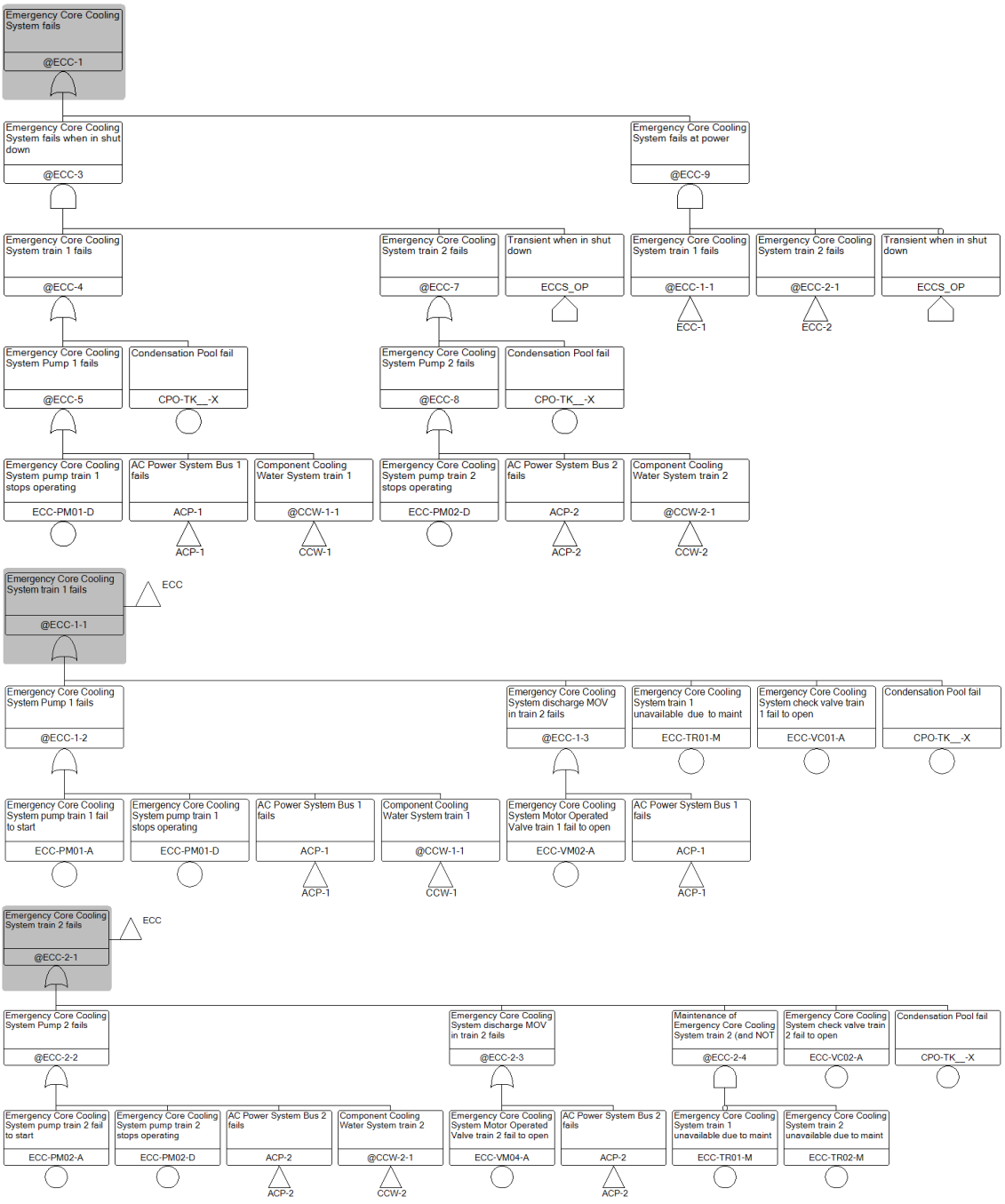


FIGURE 7.14: SFTs of the ECC System

The @ECC-3 gate's descendants consist mostly of OR gates, BEs related to the condensation pool, a house event and transfer gates to other systems. Note that there are two BEs as descendants of the @ECC-3 gate, describing failures of both pumps. Hence, the resulting P&ID shows these pumps in parallel. This is different from the RHR system where this structure is missing.

The extra structure and the added house events cause our method to add an extra parallel structure in the resulting P&ID. The way in which the algorithm traverses the SFT is clearly visible when looking at the node labels. First, it starts with the smaller structure, creating *ECC_NODE3* and *ECC_NODE4* and making sure that *ECC_PM01*

and *ECC_PM02* are in parallel. Then, the algorithm continues with the bigger structure, which structurally is similar to the resulting P&ID of the RHR system (ignoring the heat exchanger of the RHR system). Lastly, it connects both structures by creating a fluid pipe from *ECC_NODE4* to *ECC_NODE5*. Subsection 8.2.5 discusses this issue further.

For the seven properties, the comparison violates properties one, three, six and seven.

Chapter 8

Discussion

This chapter further discusses and evaluates the results. The first section focuses on the accomplishments of the presented approach. After that, the three remaining sections elaborate on limitations on and future improvements for respectively the methodology, the implementation and the verification & validation process.

8.1 Accomplishments

Based on the results, it is clear that the SPC Tool can recognize the CLAs and CTs of physical components correctly from SFTs. Furthermore, the methodology successfully infers an initial topology. The physical components which are in series in the original P&ID are also in series in the resulting P&ID and preserve ordering. The algorithm also recognizes two parallel substructures of the original P&ID correctly. In general, the approach is useful because it offers:

1. An initial, formal method to infer basic P&IDs from SFTs given the EXPSA meta-model;
2. Insight into the formal relationship between SFTs and P&IDs when both are defined as graphs and;
3. A first step towards inferring P&IDs from SFTs within other domains.

Of the three systems discussed in Section 7.2, the approach handles inferring a P&ID for the RHR system (see Subsection 7.2.1) the most successful. Both in the RHR and ECC systems, the SPC Tool recognizes the *maintenance_group* component type. While initially out of scope, it was necessary to support this because the topology of the resulting P&ID differs significantly when the algorithm ignores this abstract component type. Now that it is supported, the resulting P&IDs show that there are maintenance related safety concerns which need attention. It also gives perspective to improve the methodology using fault codes in general for component type recognition, as well as improving the topology of the resulting P&ID (see Subsection 8.2.3)

8.2 Methodology

This section elaborates on limitations and improvements with respect to the methodology. First, it focuses on the order of BEs and the impact on the resulting P&IDs. One subsection follows discussing a phenomenon where a gate has two children BEs concerning the

same physical component. After that, the third subsection discusses maintenance group components. Subsection 8.2.4 evaluates VOT gates. The fifth subsection elaborates on multiple resulting P&IDs when house events in SFTs can result in multiple configurations for one P&ID. Lastly, two subsections discuss 1) transfer gates within SFTs referring to other systems and 2) scalability when the size and/or number of SFTs increase.

8.2.1 Ordering of Basic Events

Section 7.1 shows that the ordering of BEs in SFTs is important. This is because the method is based on a Post-Order Depth-First traversal of the SFT. It preserves the order of traversing BEs (and thus, their corresponding components) in the resulting P&IDs.

One way to improve the order is to alter the order of components which are in series. This can be done by improving either pre-processing SFTs or post-processing resulting P&ID(s). For this, additional knowledge on hydraulic systems is necessary, which is currently not present in SFTs. In the case of the EXPSA metamodel, usually the flow goes from tanks to pumps, followed by check valves and then heat exchangers. This gives a simple heuristic to 1) recognize which components are in the wrong order and 2) reorder components in the correct order. The heuristic could be implemented either in the SPC Tool itself, or by using another tool (e.g. Groove[1]). There are three options for this situation:

- Keep the current resulting P&ID;
- The SPC tool chooses one of a range of different orderings, or;
- The SPC tool offers the user a choice between different orderings;

A more drastic change can be to change the algorithm which determines the topology of the resulting P&ID. Instead of using a Post-Order Depth-First traversal of the SFT, a different method may offer an initial topology with the correct order of components when placed in series.

8.2.2 Self-loops & Parallel Single Components

It is common for an SFT to have two or more descendant BEs under an OR gate which concerns the same physical component. The IDs of these BEs then contain a different fault code. For sibling BEs, this is not a problem as visible in the SFTs and P&IDs of the SWS system (see Figures E.4 and D.4). However, this changes once these BEs are descendants (e.g. by adding a gate between the BE and its parent). Our algorithm then links this physical component to itself in the resulting P&ID, causing self-loops. Examples are visible in Figures F.5 and F.3. This can be solved by removing self-loops in post-processing.

The same phenomenon can happen with AND gates. In that case, our method places a component 'in parallel' with itself, creating extra nodes around a single component. Subsection 7.2.2 shows an example of this in the resulting P&ID of the MFW system.

A better way to solve both issues is to take fault codes into account when reducing SFTs. After the step of removing maintenance groups (see Section 4.3), the additional information fault codes give is redundant. The IDs of BEs can be stripped of their fault codes and SFT reduction then removes the duplicate BE descendant.

[1] https://groove.ewi.utwente.nl/

8.2.3 Maintenance Group Components

The algorithm currently only recognizes maintenance group components and places them in the P&ID without linking them to any other component. After that, it removes all BEs related to maintenance groups from the SFT such that the PODF algorithm can infer a correct topology. An advantage is that the resulting P&ID now can contain maintenance related components. Unfortunately, the disadvantage is that these components do not show in what way maintenance impacts the system.

In the resulting P&IDs visible in Section 7.2, no type of link exists to connect maintenance group components. The links visible in the original P&IDs are abstract relations between components instead of physical ones (see Definition 2.2.1). However, if the definition includes these types of links, determining to which components the maintenance groups should link is non-trivial, especially when using the PODF traversal algorithm. Multiple questions need answering:

- How to retain the serial and parallel structures of physical components?
- Would using a look-ahead be sufficient in finding a component to link the maintenance group to?
- If a look-ahead is sufficient, how big is that look-ahead?
- Can a maintenance group be incident to more than one component?
- Without reducing SFTs, gate recursion may occur. What does this mean with respect to the topology of maintenance groups?

The SFTs of both the RHR and ECC systems (see Figures 7.10 and 7.14) may help with providing answers, since they suggest a pattern. For each train (series of components) a maintenance group exists. Hence, it would make sense to connect a maintenance group to one of the components within that train. However, the SFT with TE *@ECC-2-1* includes both maintenance groups. Here, one maintenance basic event is negated, which suggests both maintenance groups having an impact on this train. This expresses that simultaneous maintenance of both trains is impossible. Automatic modelling this in the resulting P&ID would improve the methodology. In that case, the algorithm first could ignore the maintenance group, to later link them to one of the physical components. The challenge here lies in determining which component to link to. It may be an option to leave this choice to expert judgement.

8.2.4 VOT Gate Details

Currently, the methodology does not differentiate between AND and VOT gates when applying PODF traversal. In an ideal scenario, since this is safety-relevant information, the resulting P&ID should represent the VOT threshold. The implementation of the SPC tool does already support importing the VOT gate value from an RSA file (see Subsection 5.4.2. Furthermore, the data structure of a P&ID also supports a node containing this value.

However, the conversion itself is still non-trivial. This is because:

1. The algorithm currently creates nodes after traversing the SFT and,
2. The recursive PODF function currently only supports BE IDs;

In other words, the algorithm knows it needs to create a node, but not whether this node is linked to a VOT or an AND gate. One option with which this can be solved is to alter the method by returning component objects in tuples, instead of strings.

When implementing the solution, special attention needs to go towards importing and exporting information relevant to the VOT gate. RSMB and RSPSA both use a different definition of a VOT gate, where k/n children must *not* fail. Hence, an implementation should invert this value (so; $k' = n - k$). The SPC tool does not do this yet because of the structure of the RSA files. An RSA file respresents the VOT gate value k at the gate definition. After that, the children follow and thus, n only is known after defining all the children.

The inversion of the k is also part of the reason why the SPC tool does not support the KBI export yet. For future work, Listing 8.1 shows how KBI files represent VOT gate information.

```
<Objet Action="CREER" Desc="MFW Pumps" Nom="MFW_P" Page="Main_page" Type="node">
  <Position X="366" Y="314"/>
  <Affectation Nom="number_of_trains_required" Profil="MAIN" Valeur="2"/>
</Objet>
```

Listing 8.1: VOT gate information representation in KBI files

8.2.5 P&ID Configurations & House Events

Another issue showing from the results of the case study is related to house events and configurations. This is especially visible in the comparison of the ECC P&IDs. Here, setting house events to *True* or *False* can offer two configurations. This is visible in the resulting P&ID of the ECC system: there are two substructures, as described in Subsection 7.2.3. The first configuration only includes the pumps in the resulting P&ID. The second configuration offers the full resulting P&ID without the extra parallel substructure visible around the pumps.

Improving the SFT reduction based on graph transformation rules by Junges et al. [37] can be considered. A Python library tool called dftlib[2] exists which may help with this. This then fully removes house events and, hence, removes the parallel substructure. However, the disadvantage is that sometimes, this information must be retained. Therefore, there are three other options:

- Keep the current resulting P&ID;
- The SPC tool chooses one of the substructures as resulting P&ID or;
- The SPC tool offers the user a choice between the resulting P&ID or one of its substructures;

8.2.6 Transfer Gates

Currently, the methodology does resolve internal transfer gates (see also Subsection 5.4.4). This is visible in the results of the case study in Section 7.2. What the SPC tool still ignores are transfer gates in SFTs referring to other SFTs about different systems. The SFTs of the ECC system (see Subsection 7.2.3) show an example of this. Here, there are a couple of transfer gates linking to the *CCW-1* SFT.

A naive way to show a link between these systems is to add a node showing that this link exists. Slightly more complex is to link components between two systems using interfaces,

[2] https://github.com/volkm/dftlib

which is the way in which RSMB supports this functionality. This would mean 1) parsing the EXPSA metamodel within the SPC Tool and 2) an extension of the P&ID definition since support for power-related systems is necessary. Lastly, combining all SFTs of all systems into one big SFT as a pre-processing step may be an option to support external transfer gates. The PODF algorithm then only has to traverse one big SFT, which results in multiple P&IDs.

8.2.7 Scalability

While the algorithm works for easy examples and a simple case study, it is yet unclear what the performance is for bigger examples. In other words, it is necessary to look at the scalability, both in the size of SFTs and the amount of SFTs for one system. Both time and space complexity are interesting to measure for different parts of our methodology. The expectation is that the complexity of a PODF algorithm is similar to the complexity of a Depth-First Search algorithm for graphs: $\mathcal{O}(V + E)$. Since the current approach only traverses SFTs – and hence, DAGs – research relevant to parallel DFS for DAGs is interesting here [54].

8.3 Implementation

Currently, there is one major limitation concerning the implementation of the SPC Tool. This limitation concerns the initial positioning of a P&ID (and its components) in RSMB.

8.3.1 P&ID Visualization

At the moment, the SPC Tool places P&IDs within an RSMB Study into three columns (see Section 5.4.3). It does the same for the components of each P&ID. This means that user effort is necessary to get to the visualization of the resulting P&IDs in Chapter 7. Despite that, the effort is lower than a situation in which all P&IDs (and hence, the components of a P&ID) are in one location. Unfortunately, with the current implementation, scalability still will become an issue because some manual positioning remains necessary.

One way to improve the visualization is to take the ordering of the components into account (see also Subsection 8.2.1). In the case of the EXPSA metamodel, tanks can then be displayed on the left side, pumps in the middle and check valves on the right. An automatic graph layout algorithm could offer a solution here. Enright and Ouzounis [22] describe an automatic graph layout algorithm for similarity visualization. Additionally, Böhringer and Paulisch [12] elaborate on using constraints to achieve stability in automatic graph layout algorithms.

Another improvement can be made by using coordinates for links. While the positions of links are determined by the source and target components, RSMB does support rerouting the links. The KBI file format allows for coordinates in links to visualize them differently. Figures 7.9 and 7.11 already show the difference this can make.

8.3.2 Importing and Exporting Graphs

Section 5.4 discusses importing SFTs from RSA files, as well as exporting P&IDs to KBI files. When generalizing this to graphs, it is useful to also support exporting and importing them to the DOT language developed by the Graphviz project[3]. Currently, the SPC Tool already contains a file (*src/model_io/graph_io.py*) with methods offering this support.

[3] https://graphviz.org/doc/info/lang.html

There are already a range of tools available for further processing of graphs described in DOT files, which warrant further exploring.

8.3.3 SFT Pre-processing

Subsection 5.4.4 elaborates on the implementation of the methodology. Here, an SFTController exists which first reduces the SFTs, before checking whether internal SFT transfer gates exist. After checking — and possibly getting the union of two SFTs — it might be necessary to again reduce the resulting SFT. The current validation process shows no need for this yet, but proof is necessary, showing that the union of two completely reduced SFTs results in another completely reduced SFT. If no proof exists, efficiently reducing the resulting SFT needs to look at the method of merging the two original SFTs.

8.3.4 P&ID Post-processing

When looking at P&ID post-processing, the current implementation uses the union of two P&IDs as method to merge them (see Subsections 4.1.1 and 5.4.4). An advantage of this is keeping all the information of the two P&IDs, meaning merging is lossless. However, when dealing with bigger SFTs and P&IDs, the size of the resulting P&ID could become an issue. In this case, taking the disjoint union ($PID_A + PI_B$) or intersection ($PID_A \cap PID_B$) of both P&IDs may offer a better solution.

8.4 Method Validation

There are also limitations and suggestions for improvements concerning the validation of the methodology. The subsections below divide the limitations into three parts: P&ID Similarity, P&ID Quality and SFT Similarity. With P&ID Similarity, the first subsection focuses on the Graph Isomorphism Problem and finding (or synthesizing) a benchmark of P&IDs. After that, the second subsection focuses on Expert Judgement, Model Checking and A/B Testing. The third subsection looks at comparing SFTs as another validation method.

8.4.1 P&ID Similarity: Graph Isomorphism Problem

An alternative to system tests is to treat the comparison of two P&IDs as a graph isomorphism problem (assuming that the order of a sequence stays the same, see Subsection 8.2.1).

Babai proposed a quasi-polynomial time algorithm back in 2015 to solve this problem for all graphs [4]. However, a P&ID may fall into a specific class of graph for which a lower complexity algorithm may work as well. This will probably depend on the P&ID's metamodel; not every type of P&ID will fall into one or more specific classes of graphs. Given the planar nature of the P&IDs in this thesis, the recommendation is to look at algorithms for planar graphs. Furthermore, P&IDs may have restricted bounds on graph parameters. Hence, algorithms for other bound-parameter graphs are another topic for future research.

Next to the graph isomorphism problem, it is necessary to assemble and/or synthesize a benchmark of P&IDs.

Benchmark of P&IDs

Currently, the amount of P&IDs for a proper benchmark is limited. This is partly because of the heterogeneous nature of P&IDs. Furthermore, within the domain of nuclear power

plants, the designs of these plants are usually state secrets. Section 3.2 already refers to a paper from Palawi et al. describing a dataset of 500 P&IDs [57]. A challenge with this dataset is that:

- The dataset must be converted to work with the EXPSA metamodel first or;
- The validation process must be adapted to work with the metamodel of the P&IDs in this dataset.

As an alternative, it may also be possible to synthesize a bigger set of P&IDs based on different metamodels. One option is to use example P&IDs from the DEXPI standard (see Section 10.3), available on GitLab[4]. The implementation of the SPC does contain a script to interpret a single P&ID of this dataset. Section 5.3 discusses this script and

8.4.2 P&ID Quality

Next to the similarity between original and resulting P&IDs, it is necessary to look at the quality of the resulting P&ID. This subsection suggests three methods of evaluating this: Expert Judgement, A/B Testing [81] and Model Checking.

Expert Judgement & A/B Testing

With expert judgement, it is possible to retrieve direct feedback from experts on the quality of a resulting P&ID. There is a variety of structured and unstructured methods for this, e.g. through a survey or with interviews. For a more structured approach, it is possible to use a framework which also measures software system quality attributes. Chen et al. [18] give examples of those attributes.

Another approach is A/B Testing. Here, two versions of the SPC can output two different P&IDs. An expert then can give feedback on which of the two they prefer and why. Young [81] elaborates more on using A/B Testing as a user experience research methodology. An example company which uses A/B Testing is Spotify[5]

Model Checking

Model-checking provides a formal way to measure the quality of a P&ID. Instead of asking Experts their objective opinion, model-checking looks at features of graphs and/or P&IDs and measures these. Future work is necessary to determine which features are interesting. There are multiple sources providing inspiration for the Model Checking approach. Khan et al. [41] presented a probabilistic model-checking tool for Figaro. Rensink et al. [66] compares two approaches to model-checking graph transformations. Rensink [65] also worked on model-checking graph grammars.

8.4.3 SFT Similarity

Another approach to validate the methodology is to:

1. Generate a P&ID with the SPC from an SFT;
2. Use RSMB to generate a new SFT;
3. Compare the resulting SFT with the original one and;

[4] https://gitlab.com/dexpi/TrainingTestCases/-/tree/master/dexpi%201.3/example%20pids
[5] https://engineering.atspotify.com/2020/10/spotifys-new-experimentation-platform-part-1/

4. Compare both SFTs' minimal cut sets.

Measuring the similarity of two SFTs is fundamentally the same as measuring the similarity of two P&IDs, as discussed in Subsection 8.4.1. One difficulty lies in the implementation of the SPC. At the moment, the SPC cannot define mission failures within P&IDs. This means information is missing for RSMB to determine how to generate an SFT from the imported P&ID. Once the SPC offers this functionality, it would be possible to use this as another validation method. In that case, the Formal Methods and Tools group of the University of Twente offers FFORT; a collection of fault trees gathered from scientific literature and open industrial reports [6]. Here, it might be possible to retrieve a benchmark of suitable SFTs (assuming a corresponding P&ID metamodel exists).

[6] https://dftbenchmarks.utwente.nl/

Chapter 9

Conclusion

This thesis proposes an initial algorithm which infers Piping and Instrumentation Diagrams (P&IDs) from Static Fault Trees (SFTs). The algorithm achieves this by Post-Order Depth-First traversal of the SFTs. A first implementation of this algorithm is available as a Python tool: the SFT to P&ID Converter (SPC). This tool provides a first step towards validating algorithms with the purpose of inferring P&IDs from SFTs. The validation process uses five inputs for the SPC: four basic examples covering basic SFT structures and a case study provided by RiskSpectrum. Visually comparing the original P&IDs of these inputs with the resulting ones from the SPC:

1. Shows that the algorithm can be used as an initial formal method to infer basic P&IDs from SFTs within the domain of nuclear power generators;
2. Provides the insight that P&IDs defined as graphs provide a way to formally show the relationship between SFTs and P&IDs, and;
3. Offers a first step towards inferring P&IDs from SFTs within other domains.

However, the algorithm also comes with limitations concerning: the ordering of Basic Events, Self-Loops, Parallel Single Components, Maintenance Group Components, VOT Gate Details, P&ID Configurations, Transfer Gates and Scalability. Despite the SPC offering a first validation method through visual comparison, this is only possible for smaller SFTs and smaller sets of SFTs.

9.1 Research Questions

Subsection 1.4 defines two main research questions, both having two sub-questions. Below, this section first gives an answer to RQ 1 and then to RQ 2.

RQ 1: Is it possible to formally define a P&ID?

RQ 1.1: How can we use graphs as a formal structure to define P&IDs?

RQ 1.2: How does the EXPSA metamodel relate to a formal definition of a P&ID based on a graph structure?

Section 2.2 discusses an initial, formal graph structure for P&IDs and hence answers RQ 1.1. Here, the components of a P&ID form the vertices of a graph, and the links are the edges.

An answer to RQ 1.2 is more complex. On a detailed level, Subsection 5.4.3 elaborates on the transformation of the internal P&ID graph datastructure in Python to a KBI file. While this file can be imported into RiskSpectrum ModelBuilder (RSMB) – and hence the KBI file essentially contains a P&ID based on the EXPSA metamodel – this is still on a detailed level.

Now, Section 2.3 does elaborate more on the EXPSA metamodel. Furthermore, Definition 2.2.1 takes the EXPSA metamodel into account with a limited amount of types for its elements. Hence, at the moment, the EXPSA metamodel mostly specifies and limits the used definition of a P&ID as a graph. Future work is necessary to determine whether a P&ID definition can represent all parts of the EXPSA metamodel (see Section 10.1).

The second RQ and its sub-questions are as follows:

RQ 2: To what extent can we infer P&IDs from (manually created) SFTs?

RQ 2.1: To what extent can we determine what type of P&ID node we need to create?

RQ 2.2: What information do we need to infer relationships or connections between nodes in P&IDs and how can we use that information?

Section 8.1 shows that the proposed algorithm can successfully recognize component types of physical components. The naming scheme related to the EXPSA metamodel is crucial here. However, this is within the domain of nuclear power generators only. If a naming scheme exists for other domains, then the SPC could implement functionality supporting these naming schemes. So, the answer to RQ 2.1 is that it is possible to determine it within domains where naming scheme exists.

Section 4.4 discusses the part of the proposed algorithm which synthesizes the topology of a P&ID. Since this currently uses Post-Order Depth-First traversal, it needs the whole SFT, if not multiple ones. More specifically and answering RQ 2.2: it uses the structure of SFTs to infer relationships or connections between the components of a P&ID. However, Section 8.2 elaborates on significant limitations to this approach which warrant further improvements, in particular the ordering of components.

This means that with respect to RQ 2, the proposed algorithm offers an initial method to infer P&IDs from (manually created) SFTs within the domain of nuclear power generators. It gives promising results, but comes with significant limitations which warrant further improvements.

Chapter 10

Future Work

This last chapter discusses future work regarding five research directions. It starts with a section on the EXPSA metamodel, shortly discussing:

- Minimal Cut Sets (MCSs),
- Involving another Figaro feature,
- Finding interesting features for model-checking as validation method,
- Exploiting Fault Tree symmetries to infer P&IDs and,
- Determining a complete representation of the EXPSA metamodel in the P&ID definition.

Then, three sections follow on using different types of source models, using an intermediate model and targeting other P&ID metamodels. The last section suggests involving Artificial Intelligence (AI) given the recent trend towards data-driven solutions.

10.1 EXPSA

There are a couple of things warranting future work regarding the current translation from SFTs to P&IDs when using the EXPSA metamodel, next to the points present in Chapter 8. Section 3.3 already refers to using Minimal Cut Sets (MCSs) for Fault Tree Analysis (FTA). Since an MCS also is one way to represent the structure of an SFT, existing algorithms to determine the MCS of an SFT might be quicker than the proposed algorithm in this paper. On the other hand, the list of MCSs could become large quickly.

Next to that, Subsection 2.3.3 describes Figaro's feature concerning Occurrence and Interaction Rules. Future work lies in investigating how Interaction Rules can provide information to improve the results of the current methodology.

Another recommendation is to look at Model Checking as a formal way to measure the quality of a P&ID. Section 8.4.2 looks at determining the quality of a resulting P&ID when comparison with an original P&ID is impossible. As a first step, it is necessary to determine which features of graphs (or P&IDs) are interesting with respect to Model Checking.

Symmetry within SFTs might also be an approach to improve the quality of the resulting P&IDs. Jimenez-Roa et al. [36] already perform data-driven inference of Fault Tree models by exploiting symmetries. An approach might be to use the same symmetries to infer P&IDs from SFTs.

Lastly, as discussed in Section 9.1, future work lies in determining whether a P&ID definition can represent all parts of the EXPSA metamodel. The suggestion is to start focussing on two aspects: the component type hierarchy (see also Appendix A), and representing Occurrence and Interaction rules.

10.2 Different Source Models

As mentioned in Chapter 1, a variety of FTs exist. Each FT type and extension is a possible source model that may be interesting to infer P&IDs from. Examples are DFTs, BDMPs, Repairable FTs and Fuzzy FTs. Next to that, Codetta-Raiteri [60] and Khan et al. [40] focus on a translation from DFTs and BDMPs (resp.) to Generalized Stochastic Petri Nets (GSPNs) [50, Section 5] and Continuous Time Markov Chains (CTMCs) [51]. GSPNs will probably be a major challenge because of the dynamic interactions between components. A first step here can be to check whether a GSPN complies with rules of a certain P&ID metamodel or Knowledge Base. The same approach could be an option for Event Trees [23], which show a range of possible consequences of a failure.

Note that inferring P&IDs in this case probably only makes sense if the source model has some connection to P&IDs. In general, not everything (e.g. a model of a banana) is suitable for inferring P&IDs.

10.3 Using an Intermediary Model

Another area in which future work is possible is applying the concept of Intermediate Representations (IRs). Chow [19] reports on the increasing significance of IRs within the field of compiler construction. Since both other sections both recommend different source and target models, it seems that this concept is also applicable with respect to P&IDs and FTs.

One suggestion for an IR — or in this case, Intermediary Model — is to use Reliability Block Diagrams [17]. Catelani et al. [16, 15] already use an approach with RBDs to perform reliability assessment on P&IDs. Distefano et al. [20] look at Dynamic RBDs (DRBDs) versus Dynamic FTs (DFTs) and define a mapping of DFT elements into the DRBD domain. Furthermore, they investigated the possibility to invert this mapping.

Another option is to look at the DEXPI[1], which is an industry standard for exchanging P&IDs. On GitLab, example P&IDs exist which could be useful[2].

10.4 Different P&ID Metamodels

Different metamodels next to EXPSA already exist. Krčál et al. [44] and Bäckström et al. [10] describe the existence of seven (three and four resp.) examples.

One of the examples is shared and has similarities with EXPSA; the "Spent Fuel Pool" (SFP) metamodel. This metamodel exists for failures within a specific system in nuclear power plants, where failure chains happen over longer periods of time, meaning weeks instead of hours. It requires a more dynamic metamodel, which SFP provides.

Krčál et al. also describe metamodels for "Digital Instrumentation&Control" (DIC) and "Heterogeneous Power Production" (HPP). Especially DIC is interesting, since it enables

[1] https://dexpi.org/
[2] https://gitlab.com/dexpi/TrainingTestCases/-/tree/master/dexpi%201.3/example%20pids

creating models for detecting faulty input signals. When detected, it is possible to respond properly, depending on the detected faulty input.

Bäckström et al. describe the following metamodels next to SFP and EXPSA:

- Miniplant
  Focuses on systems in processing or production plants. Blocks within these systems can differ in capacity and their reliability data. More generically, other metamodels for availability analysis and/or process modelling also exist. Note that SFT generation may or may not be supported, depending on the metamodel.

- "Non-Repairable Mission System" (NRMS)
  Focuses on systems which cannot repair failures while in operation. An example of this is a submarine. Usually, cold spares of components add redundancy to these kinds of systems as a way to improve reliability.

Another option is to look at One-Line (or Single-Line) Diagrams. Quite a bit of research exists on these diagrams, which started back in the seventies [14, 48, 33, 80, 62]. The use of these diagrams is common for electrical systems (see also Section 7.2), which makes it interesting to also infer from SFTs.

10.4.1 Extending the P&ID Definition

The current graph definition of a P&ID takes the EXPSA metamodel into account. Hence, the proposed algorithm currently cannot support links other than fluid pipes. Two component types present in the EXPSA metamodel can – therefore – not be linked to other components; the *Basic_Event* and the *Diesel_Generator*. Thus, the graph definition of a P&ID must expand accordingly to support other metamodels.

10.5 Artificial Intelligence (AI)

Lastly, another option is to look at a data-driven approach. Recently, a trend is visible towards such solutions and, more specifically, Artificial Intelligence (AI). With respect to inferring P&IDs from SFTs, the following research directions for AI models could be interesting:

- Generation of component labels;
- Classification of component types;
- Classification of link types;
- Regression estimation for the number of components and/or links in a P&ID;
- Generation of full P&IDs from a selection of SFT features with different weights;

One method which may be useful is model-based Reinforcement Learning [53]. This especially becomes interesting when a naming scheme is missing or incorrectly followed. When a naming scheme exists and is correctly followed, the big drawback against RL is that it cannot give hard guarantees about the quality of the resulting P&ID.

A challenge lies in the amount of available data to train and test models. Organizations often classify their datasets as company (or state) secrets. Furthermore, the datasets often concern a specific use case, which can result in overspecialization of an AI model.

Acknowledgements

After nine-and-a-half years of studying at the University of Twente, there are many people who have helped or influenced me. I want to dedicate this Master's thesis to all these people who have changed, challenged and supported me, both in good and bad times. However, there is still a long list of persons, groups and organizations I want to mention specifically.

First of all, my parents and my sister: Martin, Anja and Jolijn. You have always been there for me and continue to do so in any way possible, supporting me through thick and thin. Words cannot describe how much you mean to me. There regularly are moments that you know me better than myself. I love you, and I hope you will stick around in this world for as long as possible

Then, my grandma Ellie. Since she only can speak and read Dutch, the rest of this paragraph will be the only Dutch part of my thesis. Oma, dankjewel voor alles in de afgelopen jaren. Alle punten, broodjes ei en gehaktballen zijn altijd heerlijk en het is altijd gezellig om langs te gaan. Bij jou komt gezin en familie altijd op één, en vergeet je af en toe jezelf daarin. Echter zie ik je de laatste jaren ook echt zelf meer genieten, onder andere van alle tripjes met vriendinnen.

I want to continue with thanking all my supervisors. Quite a bit of patience was necessary, and you always supported my choices and offered constructive feedback in the nicest way possible. I also really appreciate all the help with the ESREL extended abstract which we published back in September. Special thanks are in order for Matthias, who even after taking on a new job at the Eindhoven University of Technology kept supervising me, often meeting twice a week. Next to that, I want to thank Pavel, Ola and Marc for all the online meetings, feedback and three-monthly licences for using RSMB and RSPSA. Apologies for keeping you from your retirement a bit longer, Marc!

After my supervisors, I want to highlight two persons without whom I would not be here today: Helen Leavis and Annet van Royen - Kerkhof. They have kept and continue keeping me alive week in, week out with their expertise, professionalism and humanity. What you do for a living has an invaluable impact on both your patients and their environments. It means the world to me, and I think I can speak for any other patient of yours.

Now, a number of wonderful, stunning and lovely human beings follow: Beitske, Jaimie, Rolf, Soof, Margot, Birgit, Myrthe. All of you are very close friends. We have enjoyed glorious, happy, sad, frustrating, terrifying, emotional, playful, inspiring and – most of all – loving moments and conversations. I am eternally grateful for that and hope we can continue this for a long time.

There are also many who have supported me through various boards, support groups, hobbies, coaching and advising. Hence, I thank Bas, Jan, Ivan, Erik, Cynthia, Kasper, Lotte, Meike, Alina, Sanne, Ellen, Lilian, Boudewijn, Geert, Geert, Evert-Jan, Hanna, Joran en Thomas. You all have made a difference in my life, and I could not have done this without you.

Lastly, I want to thank all the people involved in one or more of the following organizations and groups. All these organizations and groups have contributed to my personal development during my study years through invaluable experiences, connections, and journeys. Hence, I want to thank Studenten Harmonie Orkest Twente (SHOT), Inter-Actief Personal Computing (IAPC), International Student Productions Enschede (InSPE), Universitair Medisch Centrum Utrecht (UMC Utrecht), Koninklijke Nederlandse Hockeybond (KNHB), Luleå University of Technology (LTU) and my 2014 Do-Group Lorem Ipsum.

I have made many memories and stories with all of you. While many of you have moved on, your stories and memories will stay close to my heart.

Bibliography

[1] AS-2C Architecture Analysis and Design Language. *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex*, sep 2015. URL: https://doi.org/10.4271/AS5506/1A, doi:https://doi.org/10.4271/AS5506/1A.

[2] Esteban Arroyo, Mario Hoernicke, Pablo Rodríguez Carrion, and Alexander Fay. Automatic derivation of qualitative plant simulation models from legacy piping and instrumentation diagrams. *Comput. Chem. Eng.*, 92:112–132, 2016. doi:10.1016/j.compchemeng.2016.04.040.

[3] IAEA (Corporate Author). *Development and Application of Level 1 Probabilistic Safety Assessment for Nuclear Power Plants Specific Safety Guide*. International Atomic Energy Agency, 2010.

[4] László Babai. Graph Isomorphism in Quasipolynomial Time, January 2016. arXiv:1512.03547 [cs, math]. URL: http://arxiv.org/abs/1512.03547, doi:10.48550/arXiv.1512.03547.

[5] Martin Bergmann, Frank Hertling, and Sebastian Marquardt. How Deep Graph Analysis Based on Piping & Instrumentation Diagrams Can Boost the Oil & Gas Industry. In *ADIPEC 2020*, Abu Dhabi, November 2020. OnePetro. URL: https://onepetro.org/SPEADIP/proceedings/20ADIP/3-20ADIP/D031S090R002/452583, doi:10.2118/202811-MS.

[6] Dmitrii Bogdanov, Manish Ram, Arman Aghahosseini, Ashish Gulagi, Ayobami Solomon Oyewo, Michael Child, Upeksha Caldera, Kristina Sadovskaia, Javier Farfan, Larissa De Souza Noel Simas Barbosa, Mahdi Fasihi, Siavash Khalili, Thure Traber, and Christian Breyer. Low-cost renewable electricity as the key driver of the global energy transition towards sustainability. *Energy*, 227:120467, July 2021. URL: https://www.sciencedirect.com/science/article/pii/S0360544221007167, doi:10.1016/j.energy.2021.120467.

[7] Marc Bouissou. Automated Dependability Analysis of Complex Systems with the KB3 Workbench: the Experience of EDF R&D. In *CIEM 2005*, page 9, Bucharest, Romania, 2005. URL: http://ciem.upb.ro/2005/index.html.

[8] Marc Bouissou, Henri Bouhadana, Marc Bannelier, and Nathalie Villatte. Knowledge Modelling and Reliability Processing: Presentation of the Figaro Language and Associated Tools. *IFAC Proceedings Volumes*, 24(13):69–75, October-November 1991. URL: https://www.sciencedirect.com/science/article/pii/S1474667017513683, doi:10.1016/S1474-6670(17)51368-3.

[9] James Bucanek. Model-View-Controller Pattern. In *Learn Objective-C for Java Developers*, pages 353–402. Apress, Berkeley, CA, 2009. doi:10.1007/978-1-4302-2370-2_20.

[10] Ola Bäckström, Marc Bouissou, Pavel Krcal, and Pengbo Wang. Flexibility of Analysis Through Knowledge Bases. In *Proceedings of the 31st European Safety and Reliability Conference (ESREL 2021)*, pages 1402–1409. Research Publishing Services, 2021. URL: https://rpsonline.com.sg/proceedings/9789811820168/html/561.xml, doi:10.3850/978-981-18-2016-8_561-cd.

[11] Ola Bäckström, Yuliya Butkova, Holger Hermanns, Jan Krčál, and Pavel Krčál. Effective Static and Dynamic Fault Tree Analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9922 LNCS:266–280, 2016. Publisher: Springer, Cham ISBN: 9783319454764. URL: https://link.springer.com/chapter/10.1007/978-3-319-45477-1_21, doi:10.1007/978-3-319-45477-1_21.

[12] Karl-Friedrich Böhringer and Frances Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 43–51, New York, NY, USA, March 1990. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/97243.97250, doi:10.1145/97243.97250.

[13] Gabriele Calzavara, Eleonora Oliosi, and Gianluigi Ferrari. A time-aware data clustering approach to predictive maintenance of a pharmaceutical industrial plant. In *International Conference on Artificial Intelligence in Information and Communication, ICAIIC 2021, Jeju Island, South Korea, April 13-16, 2021*, pages 454–458. IEEE, 2021. doi:10.1109/ICAIIC51459.2021.9415206.

[14] R. Canales-Ruiz, D. Toral Garibay, and A. Alonso-Concheiro. Optimal Automatic Drawing of One-Line Diagrams. *IEEE Transactions on Power Apparatus and Systems*, PAS-98(2):387–392, March 1979. Conference Name: IEEE Transactions on Power Apparatus and Systems. URL: https://ieeexplore.ieee.org/abstract/document/4113496, doi:10.1109/TPAS.1979.319329.

[15] Marcantonio Catelani, Lorenzo Ciani, and Matteo Venzi. Reliability assessment for complex systems: A new approach based on RBD models. In *2015 IEEE International Symposium on Systems Engineering (ISSE)*, pages 286–290, September 2015. URL: https://ieeexplore.ieee.org/abstract/document/7302771, doi:10.1109/SysEng.2015.7302771.

[16] Marcantonio Catelani, Lorenzo Ciani, and Matteo Venzi. RBD Model-Based Approach for Reliability Assessment in Complex Systems. *IEEE Systems Journal*, 13(3):2089–2097, September 2019. Conference Name: IEEE Systems Journal. URL: https://ieeexplore.ieee.org/abstract/document/8401963, doi:10.1109/JSYST.2018.2840220.

[17] Marko Čepin and Marko Čepin. Reliability block diagram. *Assessment of Power System Reliability: Methods and Applications*, pages 119–123, 2011.

[18] Lianping Chen, Muhammad Ali Babar, and Bashar Nuseibeh. Characterizing Architecturally Significant Requirements. *IEEE Software*, 30(2):38–45, March 2013. URL: http://ieeexplore.ieee.org/document/6365165/, doi:10.1109/MS.2012.174.

[19] Fred Chow. Intermediate Representation: The increasing significance of intermediate representations in compilers. *Queue*, 11(10):30–37, October 2013. URL: https://dl.acm.org/doi/10.1145/2542661.2544374, doi:10.1145/2542661.2544374.

[20] Salvatore Distefano and Antonio Puliafito. Dynamic Reliability Block Diagrams VS Dynamic Fault Trees. In *2007 Annual Reliability and Maintainability Symposium*, pages 71–76, January 2007. ISSN: 0149-144X. URL: https://ieeexplore.ieee.org/abstract/document/4126327, doi:10.1109/RAMS.2007.328095.

[21] Eyad Elyan, Carlos Moreno Garcia, and Chrisina Jayne. Symbols Classification in Engineering Drawings. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2018. ISSN: 2161-4407. doi:10.1109/IJCNN.2018.8489087.

[22] Anton J. Enright and Christos A. Ouzounis. BioLayout—an automatic graph layout algorithm for similarity visualization. *Bioinformatics*, 17(9):853–854, September 2001. doi:10.1093/bioinformatics/17.9.853.

[23] Clifton. A. Ericson. Event Tree Analysis. In *Hazard Analysis Techniques for System Safety*, pages 223–234. John Wiley & Sons, Ltd, 2005. Section: 12 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471739421.ch12. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471739421.ch12, doi:10.1002/0471739421.ch12.

[24] Peter Feiler and Julien Delange. Automated Fault Tree Analysis from AADL Models. *ACM SIGAda Ada Letters*, 36(2):39–46, May 2017. URL: https://dl.acm.org/doi/10.1145/3092893.3092900, doi:10.1145/3092893.3092900.

[25] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction:. Technical report, Defense Technical Information Center, Fort Belvoir, VA, February 2006. URL: http://www.dtic.mil/docs/citations/ADA455842, doi:10.21236/ADA455842.

[26] Wei Gao, Yunfei Zhao, and Carol Smidts. Component detection in piping and instrumentation diagrams of nuclear power plants based on neural networks. *Progress in Nuclear Energy*, 128:103491, October 2020. URL: https://www.sciencedirect.com/science/article/pii/S0149197020302419, doi:10.1016/j.pnucene.2020.103491.

[27] Hayette Gatfaoui. From Fault Tree to Credit Risk Assessment: An Empirical Attempt. *The IUP Journal of Risk & Insurance*, 3:7–31, January 2006.

[28] Hayette Gatfaoui. From Fault Tree to Credit Risk Assessment: A Case Study. *International Research Journal of Finance and Economics*, March:379–401, March 2008.

[29] Stanley S Grossel. Guidelines for hazard evaluation procedures, aiche center for chemical process safety, john wiley & sons, inc., hoboken, nj (2008), 542pp., $125.00, isbn 978-0-471-97815-2, 2008.

[30] R. Gulati and J.B. Dugan. A modular approach for analyzing static and dynamic fault trees. In *Annual Reliability and Maintainability Symposium*, pages 57–63, January 1997. ISSN: 0149-144X. URL: https://ieeexplore.ieee.org/abstract/document/571665, doi:10.1109/RAMS.1997.571665.

[31] Johan Gustafsson. Reliability analysis of safety-related digital instrumentation and control in a nuclear power plant. Master's thesis, KTH Royal Institute of Technology, Stockholm, May 2012. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-107186.

[32] Gholamreza Heravi and Ashkan Gholami. The Influence of Project Risk Management Maturity and Organizational Learning on the Success of Power Plant Construction Projects. *Project Management Journal*, 49(5):22–37, October 2018. URL: http://journals.sagepub.com/doi/10.1177/8756972818786661, doi:10.1177/8756972818786661.

[33] Jing Hong, Yue Li, Yiran Xu, Chen Yuan, Hong Fan, Guangyi Liu, and Renchang Dai. Substation One-Line Diagram Automatic Generation and Visualization. In *2019 IEEE Innovative Smart Grid Technologies - Asia (ISGT Asia)*, pages 1086–1091, May 2019. ISSN: 2378-8542. URL: https://ieeexplore.ieee.org/abstract/document/8881340, doi:10.1109/ISGT-Asia.2019.8881340.

[34] A. Hunt, B. E. Kelly, J. S. Mullhi, F. P. Lees, and A. G. Rushton. The propagation of faults in process plants: 6, overview of, and modelling for, fault tree synthesis. *Reliability Engineering & System Safety*, 39(2):173–194, January 1993. URL: https://www.sciencedirect.com/science/article/pii/095183209390041V, doi:10.1016/0951-8320(93)90041-V.

[35] Lisandro Arturo Jimenez-Roa, Tom Heskes, Tiedo Tinga, and Mariëlle Stoelinga. Automatic inference of fault tree models via multi-objective evolutionary algorithms. *CoRR*, abs/2204.03743, 2022. arXiv:2204.03743, doi:10.48550/arXiv.2204.03743.

[36] Lisandro Arturo Jimenez-Roa, Matthias Volk, and Mariëlle Stoelinga. Data-Driven Inference of Fault Tree Models Exploiting Symmetry and Modularization. In Mario Trapp, Francesca Saglietti, Marc Spisländer, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 46–61, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-14835-4_4.

[37] Sebastian Junges, Dennis Guck, Joost-Pieter Katoen, Arend Rensink, and Mariëlle Stoelinga. Fault trees on a diet: automated reduction by graph rewriting. *Formal Aspects Comput.*, 29(4):651–703, 2017. doi:10.1007/s00165-016-0412-0.

[38] Sebastian Junges, Dennis Guck, Joost-Pieter Katoen, and Mariëlle Stoelinga. Uncovering dynamic fault trees. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 299–310. IEEE Computer Society, 2016. doi:10.1109/DSN.2016.35.

[39] Stuart Kent. Model driven engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002. doi:10.1007/3-540-47884-1_16.

[40] Shahid Khan, Joost Pieter Katoen, and Marc Bouissou. Explaining Boolean-Logic Driven Markov Processes using GSPNs. *Proceedings - 16th European Dependable*

Computing Conference, EDCC 2020, pages 119–126, September 2020. Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781728189369. doi:10.1109/EDCC51268.2020.00028.

[41] Shahid Khan, Matthias Volk, Joost-Pieter Katoen, Alexis Braibant, and Marc Bouissou. Model checking the multi-formalism language FIGARO. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pages 463–470. IEEE, 2021. doi:10.1109/DSN48987.2021.00056.

[42] Tomaz Kosar, Pablo E. Martínez López, Pablo Andrés Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, 2008. doi:10.1016/j.infsof.2007.04.002.

[43] Siwar Kriaa, Marc Bouissou, and Youssef Laarouchi. A new safety and security risk analysis framework for industrial control systems. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 233(2):151–174, April 2019. Publisher: SAGE Publications. doi:10.1177/1748006X18765885.

[44] Pavel Krčál, Ola Bäckström, and Helena Troili. Control Logic Encoding using RiskSpectrum ModelBuilder. In *Probabilistic Safety Assessment & Management*, volume 16, Honolulu, Hawaii, 2022. International Association for PSAM. URL: https://www.iapsam.org/PSAM16/papers/PA131-PSAM16.pdf.

[45] Thomas Kühne. Matters of (meta-)modeling. *Softw. Syst. Model.*, 5(4):369–385, 2006. doi:10.1007/s10270-006-0017-9.

[46] W. S. Lee, D. L. Grosh, F. A. Tillman, and C. H. Lie. Fault Tree Analysis, Methods, and Applications - A Review. *IEEE Transactions on Reliability*, R-34(3):194–203, August 1985. Conference Name: IEEE Transactions on Reliability. doi:10.1109/TR.1985.5222114.

[47] A. Leff and J.T. Rayfield. Web-application development using the Model/View/-Controller design pattern. In *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, pages 118–127, September 2001. URL: https://ieeexplore.ieee.org/abstract/document/950428, doi:10.1109/EDOC.2001.950428.

[48] Imre Lendak, Aleksandar Erdeljan, Darko Čapko, and Srđan Vukmirović. Algorithms in electric power system one-line diagram creation. In *2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 2867–2873, October 2010. ISSN: 1062-922X. URL: https://ieeexplore.ieee.org/abstract/document/5641927, doi:10.1109/ICSMC.2010.5641927.

[49] Dong Liu, Weiyan Xing, Chunyuan Zhang, Rui Li, and Haiyan Li. Cut sequence set generation for fault tree analysis. In Yann-Hang Lee, Heung-Nam Kim, Jong Kim, Yongwan Park, Laurence Tianruo Yang, and Sung Won Kim, editors, *Embedded Software and Systems, [Third] International Conference, ICESS 2007, Daegu, Korea, May 14-16, 2007, Proceedings*, volume 4523 of *Lecture Notes in Computer Science*, pages 592–603. Springer, 2007. doi:10.1007/978-3-540-72685-2_55.

[50] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. Modelling with Generalized Stochastic Petri Nets. *ACM SIGMETRICS Performance Evaluation Review*, 26(2):2, August 1998. URL: https://dl.acm.org/doi/10.1145/288197.581193, doi:10.1145/288197.581193.

[51] John I McCool. Probability and statistics with reliability, queuing and computer science applications. *Technometrics*, 45(1):107, 2003.

[52] Mohammad Modarres, Taotao Zhou, and Mahmoud Massoud. Advances in multi-unit nuclear power plant probabilistic risk assessment. *Reliab. Eng. Syst. Saf.*, 157:87–100, 2017. doi:10.1016/j.ress.2016.08.005.

[53] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Model-based Reinforcement Learning: A Survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, January 2023. Publisher: Now Publishers, Inc. URL: https://www.nowpublishers.com/article/Details/MAL-086, doi:10.1561/2200000086.

[54] Maxim Naumov, Alysson Vrielink, and Michael Garland. Parallel Depth-First Search for Directed Acyclic Graphs. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, IA3'17, pages 1–8, New York, NY, USA, November 2017. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3149704.3149764, doi:10.1145/3149704.3149764.

[55] Meike Nauta, Doina Bucur, and Mariëlle Stoelinga. LIFT: learning fault trees from observational data. In Annabelle McIver and András Horváth, editors, *Quantitative Evaluation of Systems - 15th International Conference, QEST 2018, Beijing, China, September 4-7, 2018, Proceedings*, volume 11024 of *Lecture Notes in Computer Science*, pages 306–322. Springer, 2018. doi:10.1007/978-3-319-99154-2_19.

[56] Jonas Oeing, Wolfgang Welscher, Niclas Krink, Lars Jansen, Fabian Henke, and Norbert Kockmann. Using artificial intelligence to support the drawing of piping and instrumentation diagrams using DEXPI standard. *Digital Chemical Engineering*, 4:100038, September 2022. URL: https://www.sciencedirect.com/science/article/pii/S2772508122000291, doi:10.1016/j.dche.2022.100038.

[57] Shubham Paliwal, Arushi Jain, Monika Sharma, and Lovekesh Vig. Digitize-pid: Automatic digitization of piping and instrumentation diagrams. *CoRR*, abs/2109.03794, 2021. URL: https://arxiv.org/abs/2109.03794, arXiv:2109.03794.

[58] Tatiana Prosvirnova. *AltaRica 3.0: a Model-Based approach for Safety Analyses. (AltaRica 3.0 : une appoche orientée modèles pour la Sûreté de Fonctionnement)*. PhD thesis, École Polytechnique, Palaiseau, France, 2014. URL: https://tel.archives-ouvertes.fr/tel-01119730.

[59] Rohit Rahul, Shubham Paliwal, Monika Sharma, and Lovekesh Vig. Automatic information extraction from piping and instrumentation diagrams. *CoRR*, abs/1901.11383, 2019. URL: http://arxiv.org/abs/1901.11383, arXiv:1901.11383.

[60] Daniele Codetta Raiteri. The conversion of dynamic fault trees to stochastic petri nets, as a case of graph transformation. In Hartmut Ehrig, Julia Padberg, and Grzegorz Rozenberg, editors, *Proceedings of the Workshop on Petri Nets and Graph Transformations, PNGT@ICGT 2004, Rome, Italy, October 2, 2004*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 45–60. Elsevier, 2004. doi:10.1016/j.entcs.2005.02.005.

[61] Daniele Codetta Raiteri, Mauro Iacono, Giuliana Franceschinis, and Valeria Vittorini. Repairable fault tree for the automatic evaluation of repair policies. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 659–668. IEEE Computer Society, 2004. doi:10.1109/DSN.2004.1311936.

[62] N. Raman, H. P. Khincha, and K. Parthasarathy. AUTOMATIC GENERATION OF POWER SYSTEM ONE-LINE DIAGRAMS. In M. Ramamoorty, editor, *Automation and Instrumentation for Power Plants*, IFAC Symposia Series, pages 225–229. Pergamon, Oxford, January 1989. URL: https://www.sciencedirect.com/science/article/pii/B9780080341972500367, doi:10.1016/B978-0-08-034197-2.50036-7.

[63] Antoine Rauzy. Toward an efficient implementation of the MOCUS algorithm. *IEEE Trans. Reliab.*, 52(2):175–180, 2003. doi:10.1109/TR.2003.813160.

[64] I. Renault, M. Pilliere, N. Villatte, and P. Mouttapa. KB3: computer program for automatic generation of fault trees. In *Annual Reliability and Maintainability. Symposium. 1999 Proceedings (Cat. No.99CH36283)*, pages 389–395, January 1999. doi:10.1109/RAMS.1999.744149.

[65] Arend Rensink. Model checking graph grammars. In *Proc. of AVOCS*, volume 3, 2003.

[66] Arend Rensink, Ákos Schmidt, and Dániel Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256, pages 226–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-540-30203-2_17, doi:10.1007/978-3-540-30203-2_17.

[67] Enno Ruijters, Stefano Schivo, Mariëlle Stoelinga, and Arend Rensink. Uniform analysis of fault trees through model transformations. In *2017 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–7, January 2017. doi:10.1109/RAM.2017.7889759.

[68] Enno Ruijters and Mariëlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Comput. Sci. Rev.*, 15:29–62, 2015. doi:10.1016/j.cosrev.2015.03.001.

[69] Suzanne Schroer and Mohammad Modarres. An event classification schema for evaluating site risk in a multi-unit nuclear power plant probabilistic risk assessment. *Reliab. Eng. Syst. Saf.*, 117:40–51, 2013. doi:10.1016/j.ress.2013.03.005.

[70] Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, 2003. doi:10.1109/MS.2003.1231147.

[71] Claude E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27(3):379–423, 1948. doi:10.1002/j.1538-7305.1948.tb01338.x.

[72] Michael Stamatelatos, William Vesely, Joanne Dugan, Joseph Fragola, Joseph Minarick, and Jan Railsback. *Fault tree handbook with aerospace applications*. NASA Headquarters, Washington, DC 20546, August 2002. URL: https://www.mwftr.com/CS2/Fault%20Tree%20Handbook_NASA.pdf.

[73] Wei-Chian Tan, I-Ming Chen, and Hoon Kiang Tan. Automated identification of components in raster piping and instrumentation diagram with minimal pre-processing. In *IEEE International Conference on Automation Science and Engineering, CASE 2016, Fort Worth, TX, USA, August 21-25, 2016*, pages 1301–1306. IEEE, 2016. doi:10.1109/COASE.2016.7743558.

[74] Hideo Tanaka, L. T. Fan, F. S. Lai, and K. Toguchi. Fault-Tree Analysis by Fuzzy Probability. *IEEE Transactions on Reliability*, R-32(5):453–457, December 1983. Conference Name: IEEE Transactions on Reliability. URL: https://ieeexplore.ieee.org/abstract/document/5221727, doi:10.1109/TR.1983.5221727.

[75] J.R. Taylor. An Algorithm For Fault-Tree Construction. *IEEE Transactions on Reliability*, R-31(2):137–146, June 1982. Conference Name: IEEE Transactions on Reliability. doi:10.1109/TR.1982.5221276.

[76] Jinfang Tian, Longguang Yu, Rui Xue, Shan Zhuang, and Yuli Shan. Global low-carbon energy transition in the post-COVID-19 era. *Applied Energy*, 307:118205, February 2022. URL: https://www.sciencedirect.com/science/article/pii/S0306261921014720, doi:10.1016/j.apenergy.2021.118205.

[77] Moe Toghraei. *Piping and Instrumentation Diagram Development*. John Wiley & Sons, March 2019. Google-Books-ID: ICONDwAAQBAJ.

[78] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996. doi:10.1007/3-540-65306-6_21.

[79] Jianwen Xiang and Kazuo Yanoo. Formal static fault tree analysis. In *The 2010 International Conference on Computer Engineering & Systems*, pages 280–286, November 2010. URL: https://ieeexplore.ieee.org/abstract/document/5674869, doi:10.1109/ICCES.2010.5674869.

[80] Zhu Yongli and O.P. Malik. Intelligent automatic generation of graphical one-line substation arrangement diagrams. *IEEE Transactions on Power Delivery*, 18(3):729–735, July 2003. Conference Name: IEEE Transactions on Power Delivery. URL: https://ieeexplore.ieee.org/abstract/document/1208349, doi:10.1109/TPWRD.2003.813819.

[81] Scott W. H. Young. Improving Library User Experience with A/B Testing: Principles and Process. *Weave: Journal of Library User Experience*, 1(1), 2014. URL: http://hdl.handle.net/2027/spo.12535642.0001.101, doi:https://doi.org/10.3998/weave.12535642.0001.101.

[82] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. Version Control System: A Review. *Procedia Computer Science*, 135:408–415, January 2018. URL: https://www.sciencedirect.com/science/article/pii/S1877050918314819, doi:10.1016/j.procs.2018.08.191.

Appendix A

EXPSA Metamodel

This appendix shows an overview of the object types within the EXPSA metamodel. Section 2.2 shows and discusses most object types with respect to the definition of a P&ID as graph. Later in the same Chapter, Section 2.3 elaborates on Figaro and the EXPSA metamodel.

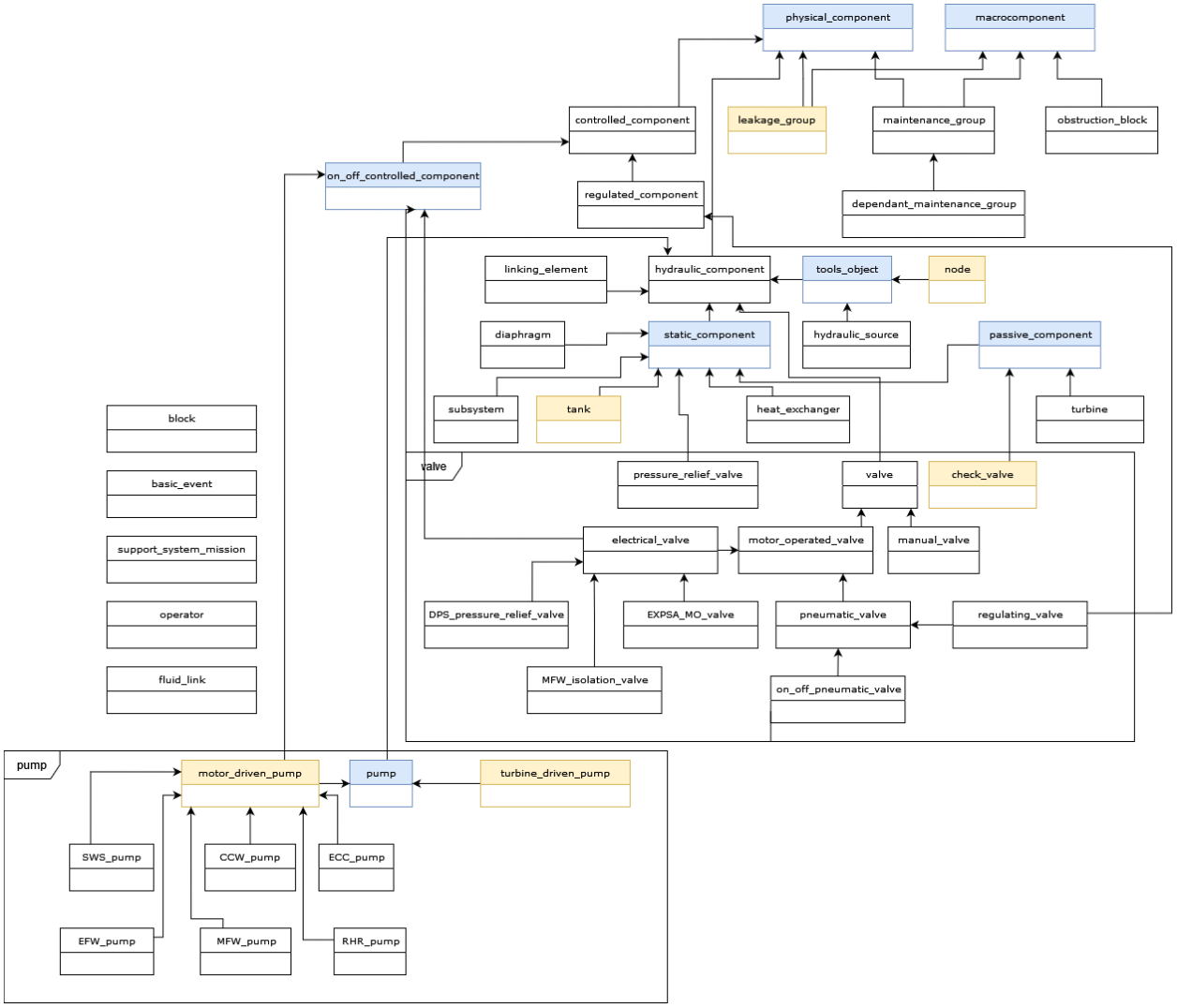


FIGURE A.1: Object types within the EXPSA metamodel

Appendix B

Methodology Algorithm Exercises

Section 4.4.1 describes the SPC's algorithm for determining the topology of a resulting P&ID. This appendix provides extra example exercises to apply this algorithm on.

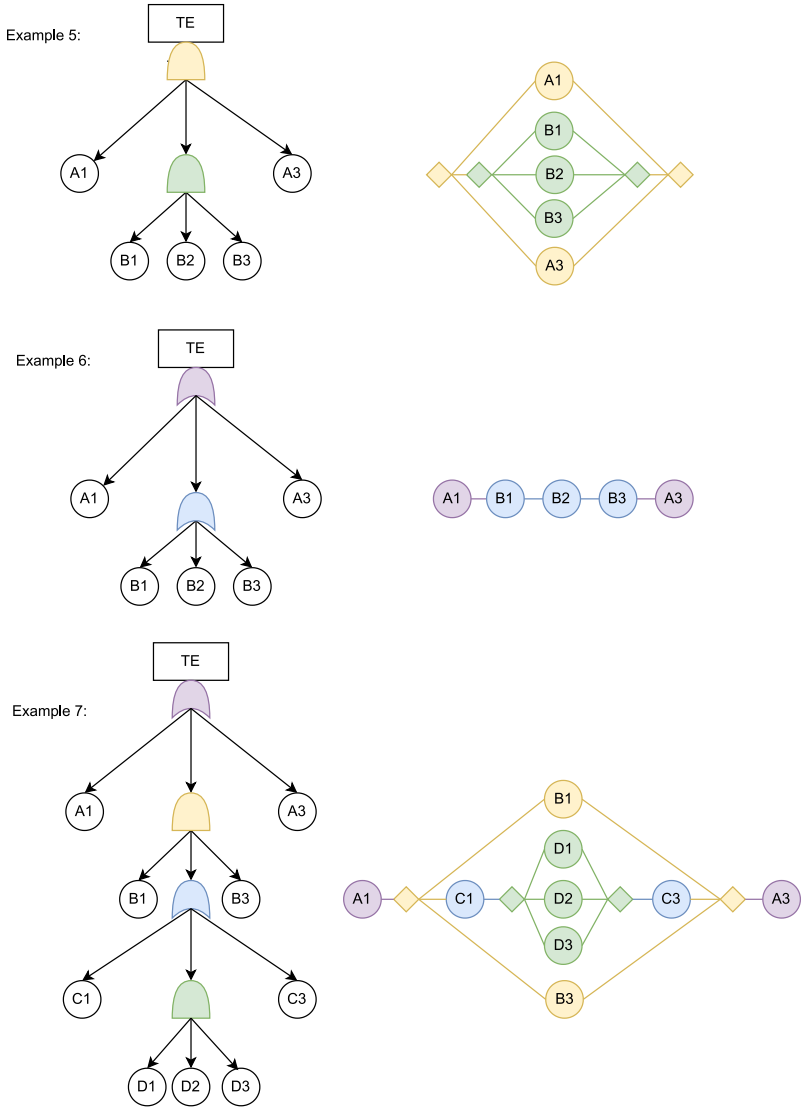


FIGURE B.1: Extra Example Exercises for Methodlogy Algorithm

Appendix C

System Overview

This appendix shows an overview of the system for validating the methodology (see Section 6.2). Figure C.1 shows UML-like relationship models for the whole system covering entities within RSMB, RSPSA and the SPC. The yellow, red and purple boxes show entities within RSMB, RSPSA and the SPC respectively. The yellow, red and purple arrows show actions or relations for which RSMB, RSPSA and the SPC are responsible (in that order).

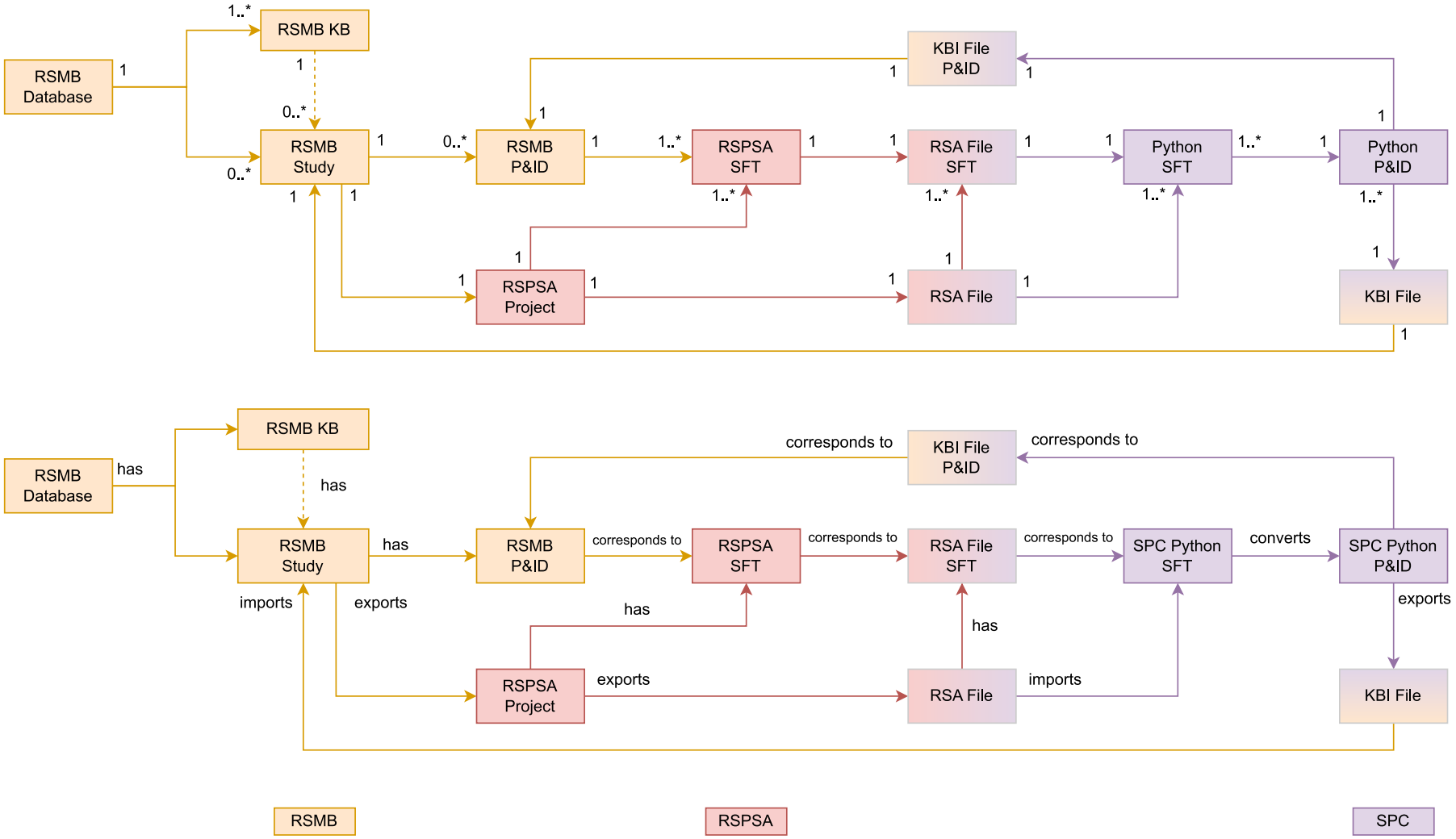


FIGURE C.1: UML-Like System Overview

Appendix D

Case Study: Remaining Original & Resulting P&IDs

This appendix shows the original and resulting P&IDs of the remaining thermohydraulic systems (see Section 7.2). The top P&ID in each figure is the original P&ID which is the input for the SPC. The bottom P&ID in each figure is the resulting P&ID which is the output of the SPC.

D.1 CCW System

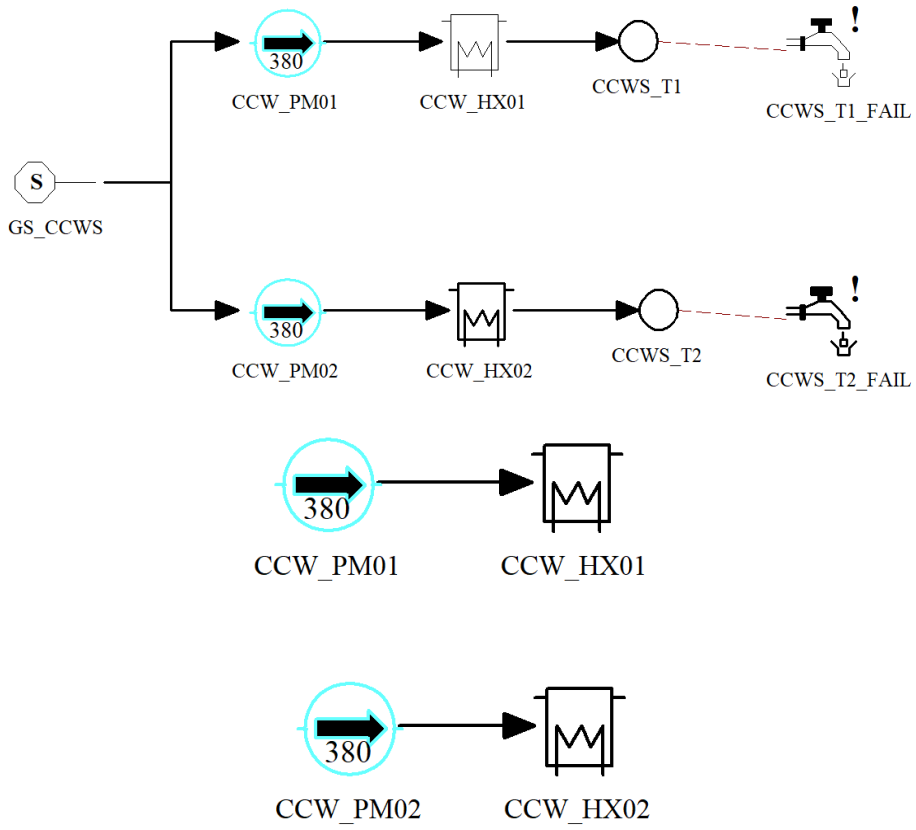


FIGURE D.1: Original and Resulting P&IDs of the CCW System in the Case Study

D.2 DPS System

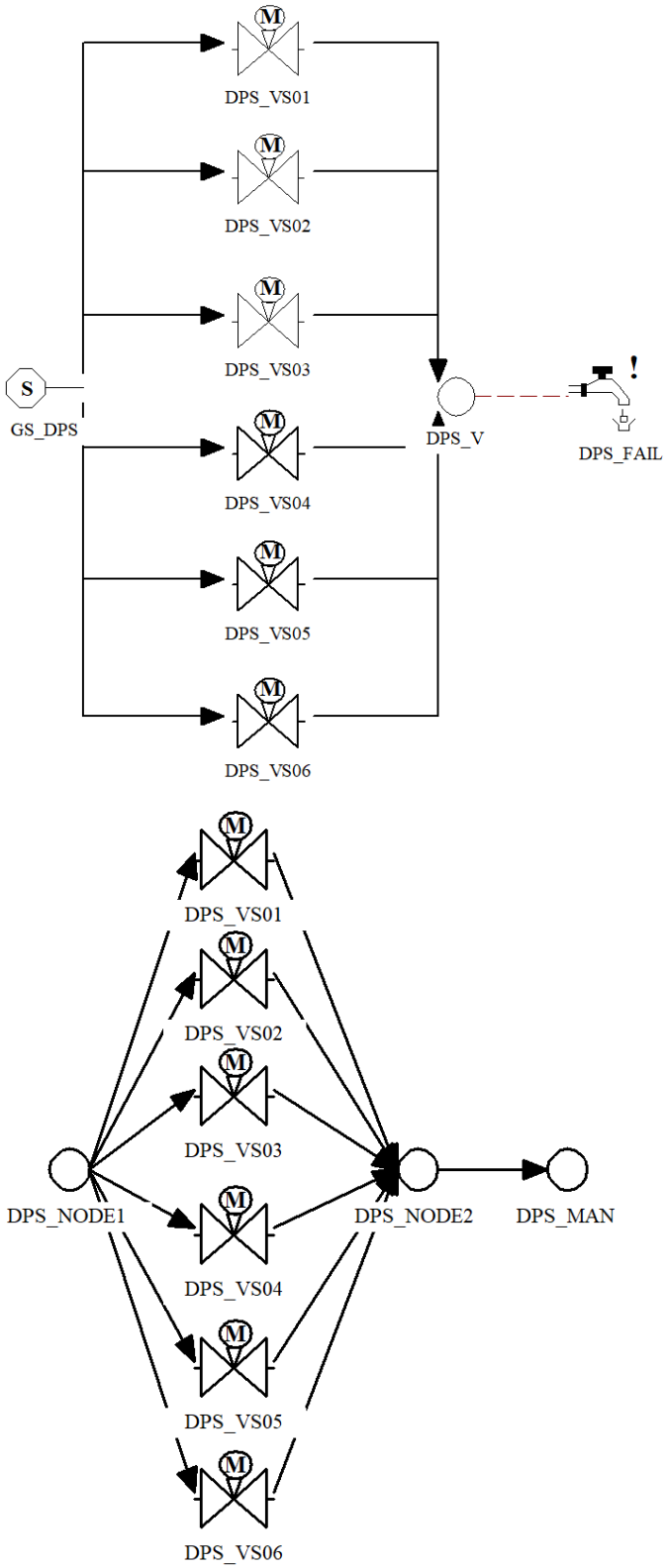


FIGURE D.2: Original and Resulting P&IDs of the DPS System in the Case Study

D.3 EFW System

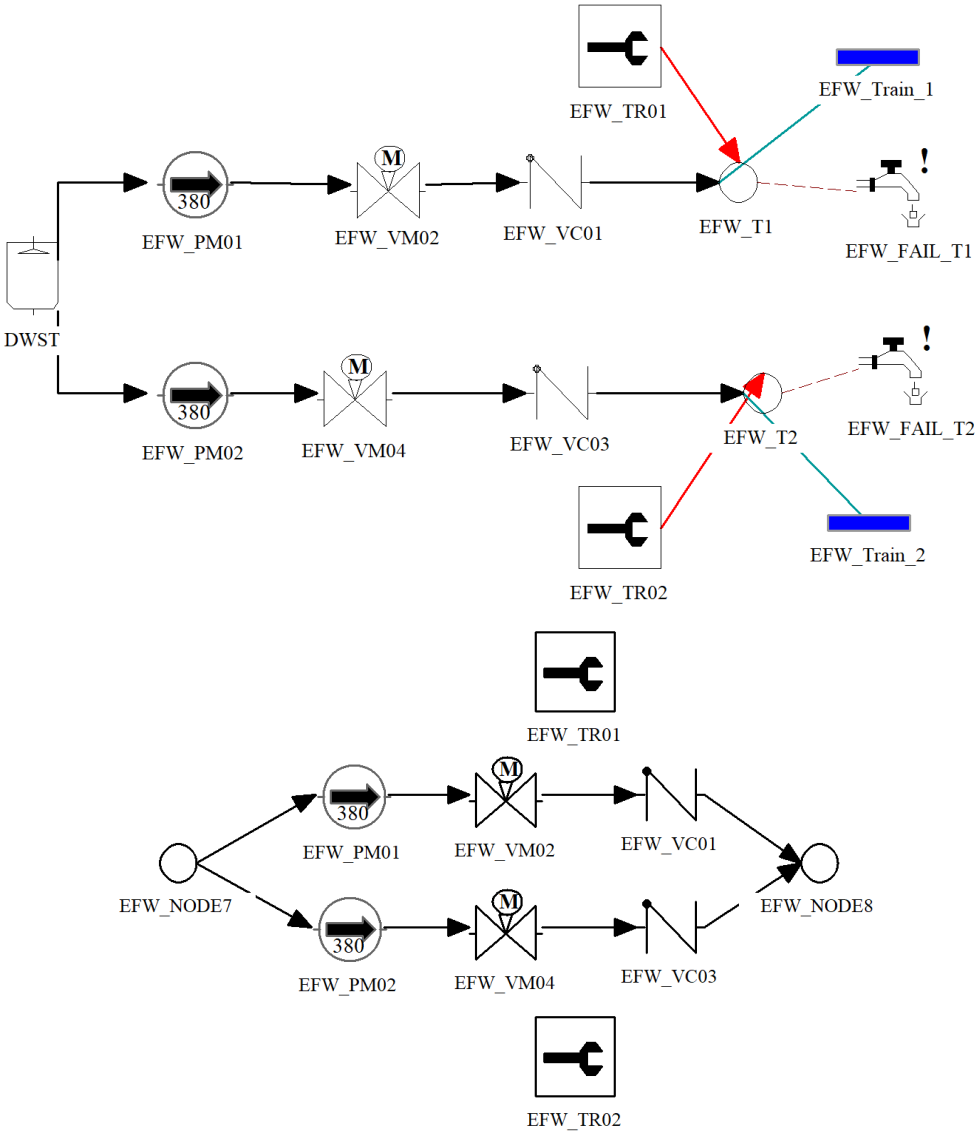


FIGURE D.3: Original and Resulting P&IDs of the EFW System in the Case Study

D.4 SWS System

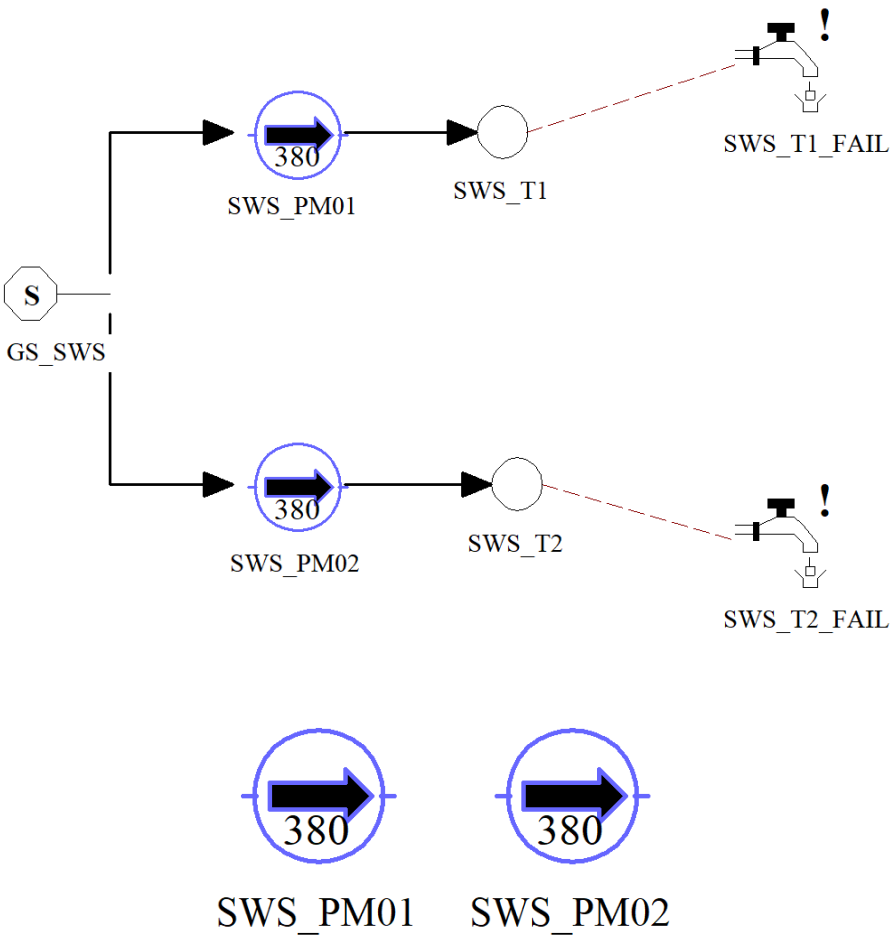


FIGURE D.4: Original and Resulting P&IDs of the SWS System in the Case Study

Appendix E

Case Study: RSPSA SFTs

This appendix shows the SFTs of the remaining thermohydraulic systems (see Section 7.2). These SFTs correspond with the SFTs after step 2 visible in Figure 6.1.

E.1 CCW System

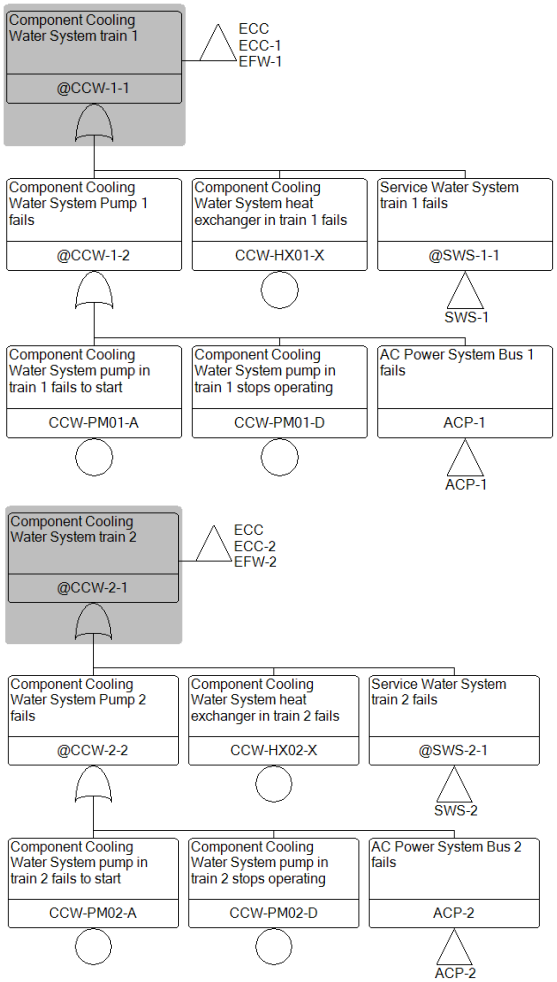


FIGURE E.1: SFTs of the CCW System

E.2 DPS System

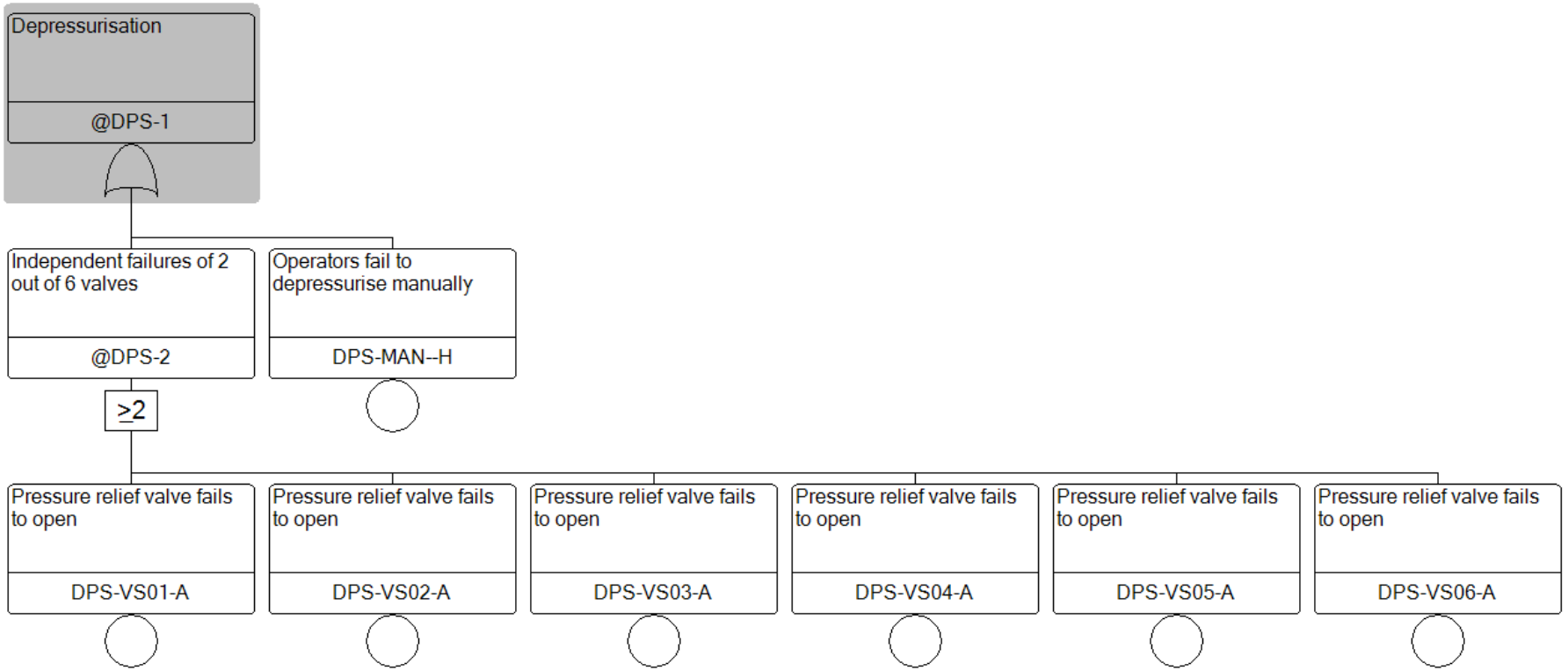
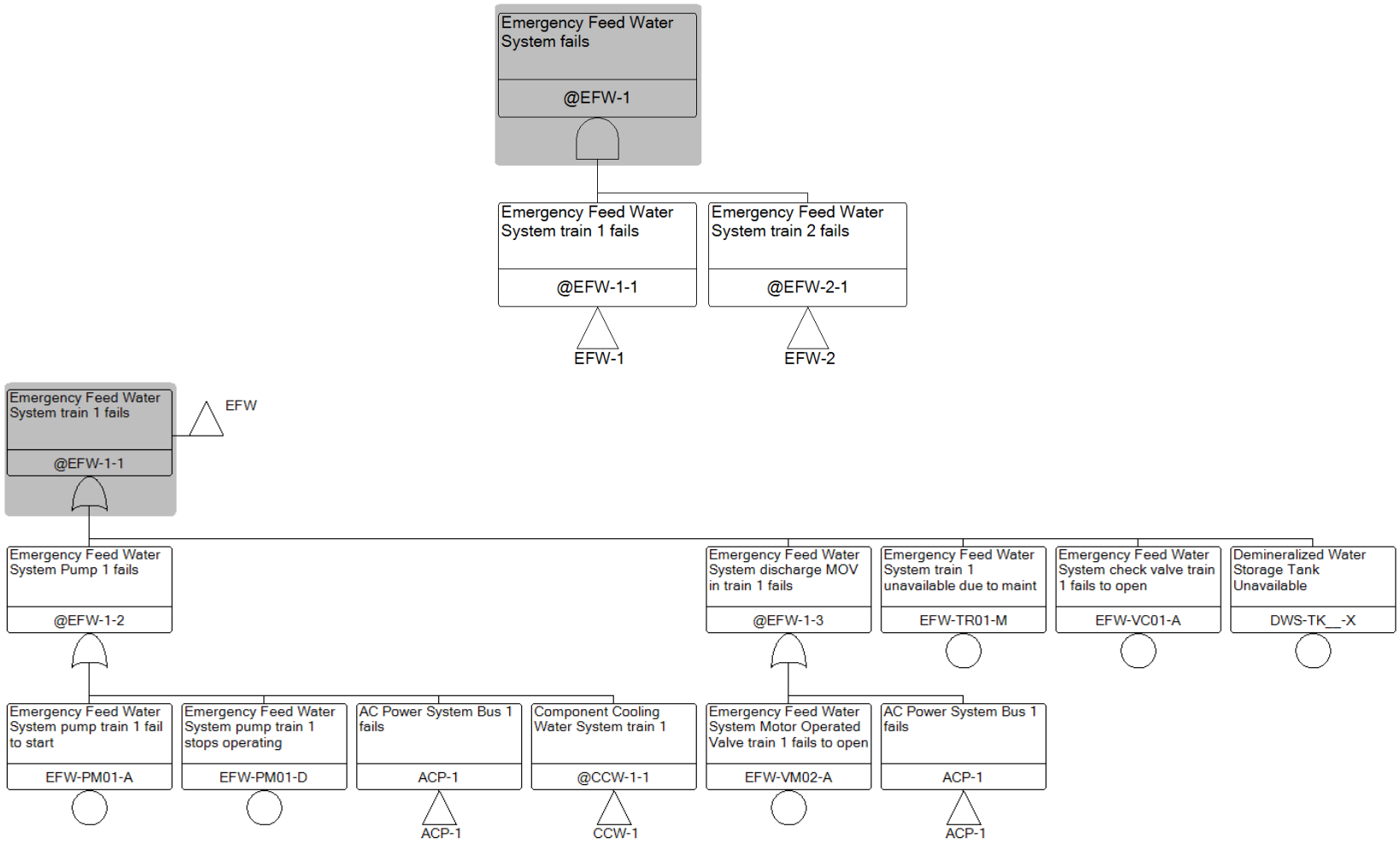


FIGURE E.2: SFTs of the DPS System

E.3 EFW System



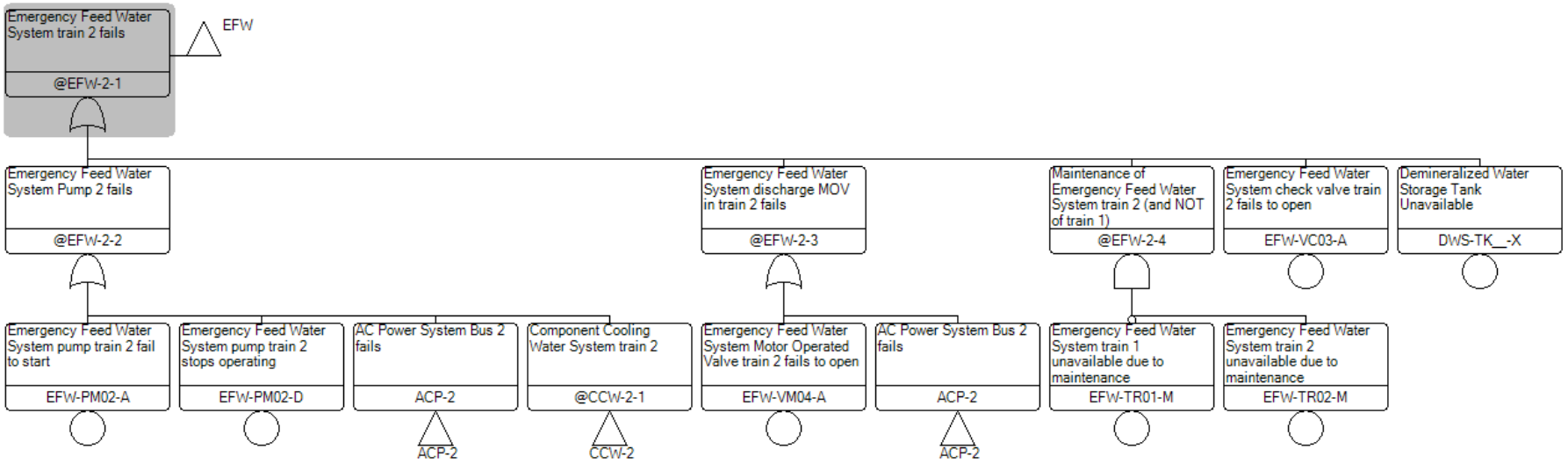


FIGURE E.3: SFTs of the EFW System

E.4 SWS System

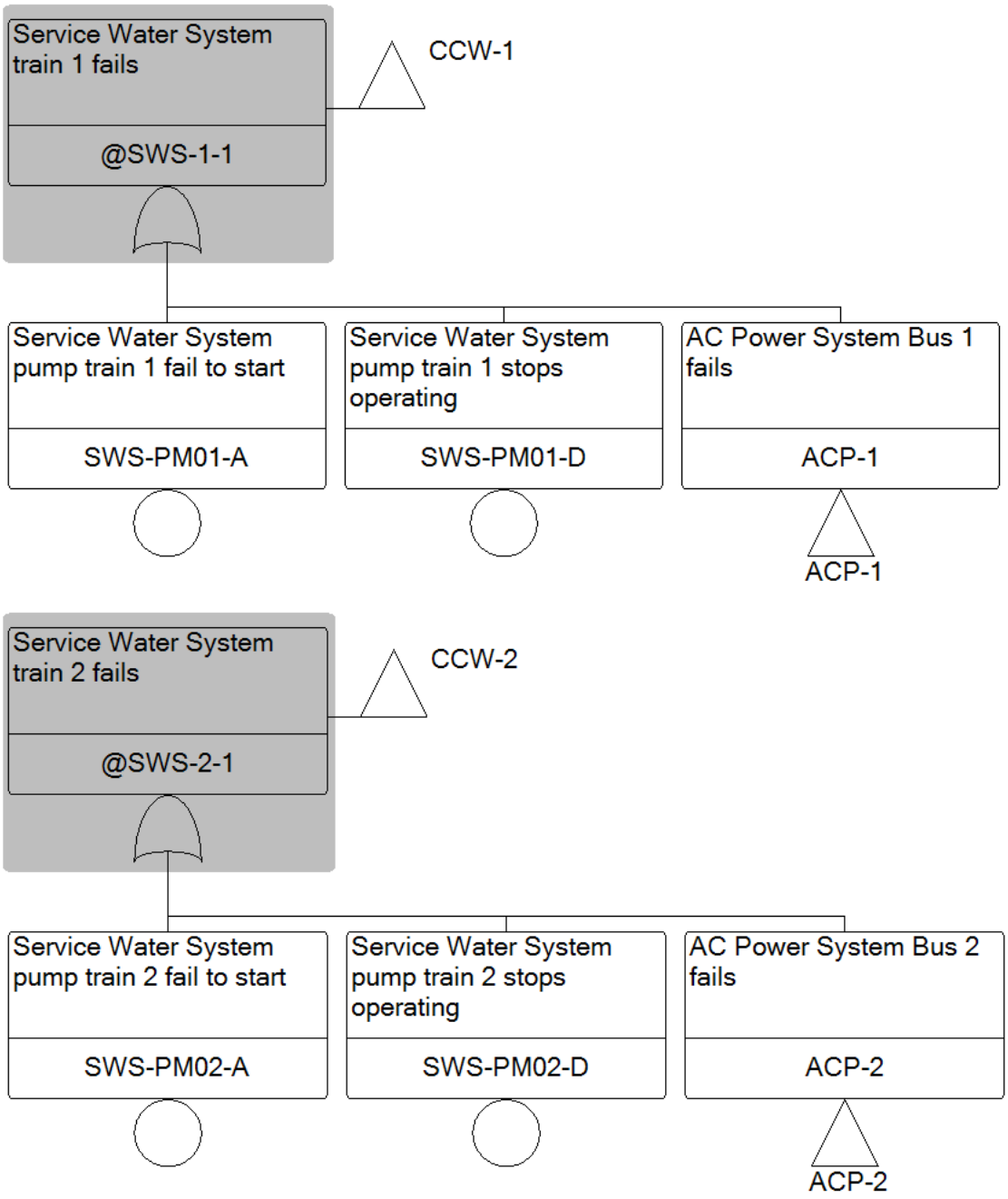


FIGURE E.4: SFTs of the SWS System

Appendix F

Case Study: Resulting P&IDs RSMB Project vs. RSPSA Project

Section 7.2 elaborates on using a case study as input of the SPC to validate the methodology. Here, one important detail is that the SPC uses a different set of 20 SFTs instead of the 30 SFTs RSMB generates from the original P&IDs. This appendix shows the resulting P&IDs of all thermohydraulic systems in the case study, for both sets of SFTs.

The top P&ID in each figure is the resulting P&ID of the set of 30 SFTs (the RSMB Project), after applying the workarounds described in Section 7.2. The bottom P&ID in each figure is the resulting P&ID of the set of 20 SFTs (the RSPSA Project). Subsections 7.2.1 until 7.2.3 and Appendix D use the bottom P&ID of each figure.

F.1 CCW System

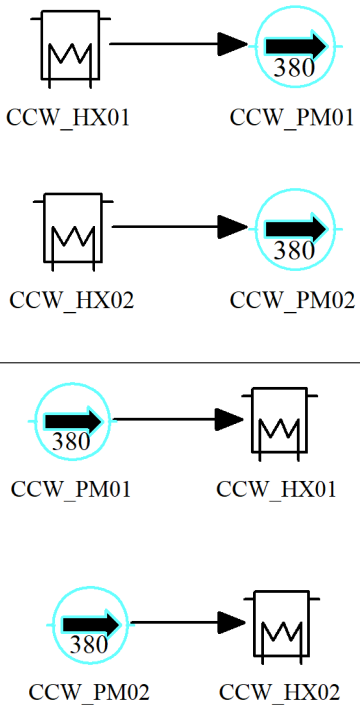


FIGURE F.1: Resulting P&IDs of the CCW System.
Above the version of the RSMB Project, below the version of the RSPSA Project.

F.2 DPS System

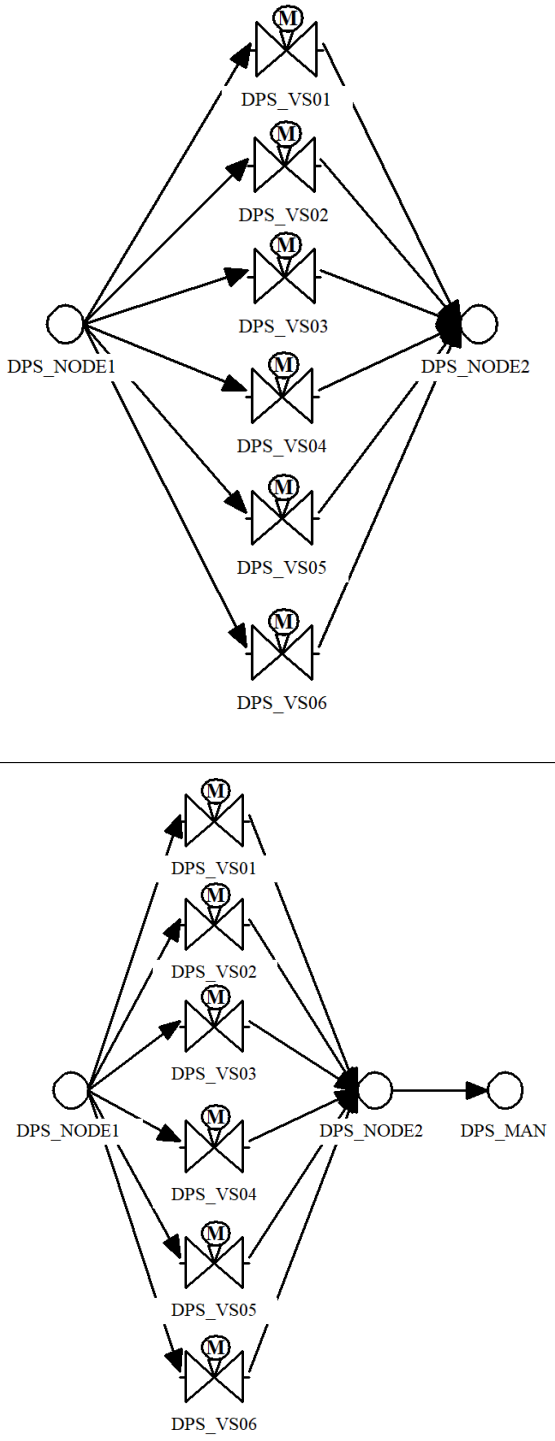


FIGURE F.2: Resulting P&IDs of the DPS System.
Above the version of the RSMB Project, below the version of the RSPSA Project.

F.3 ECC System

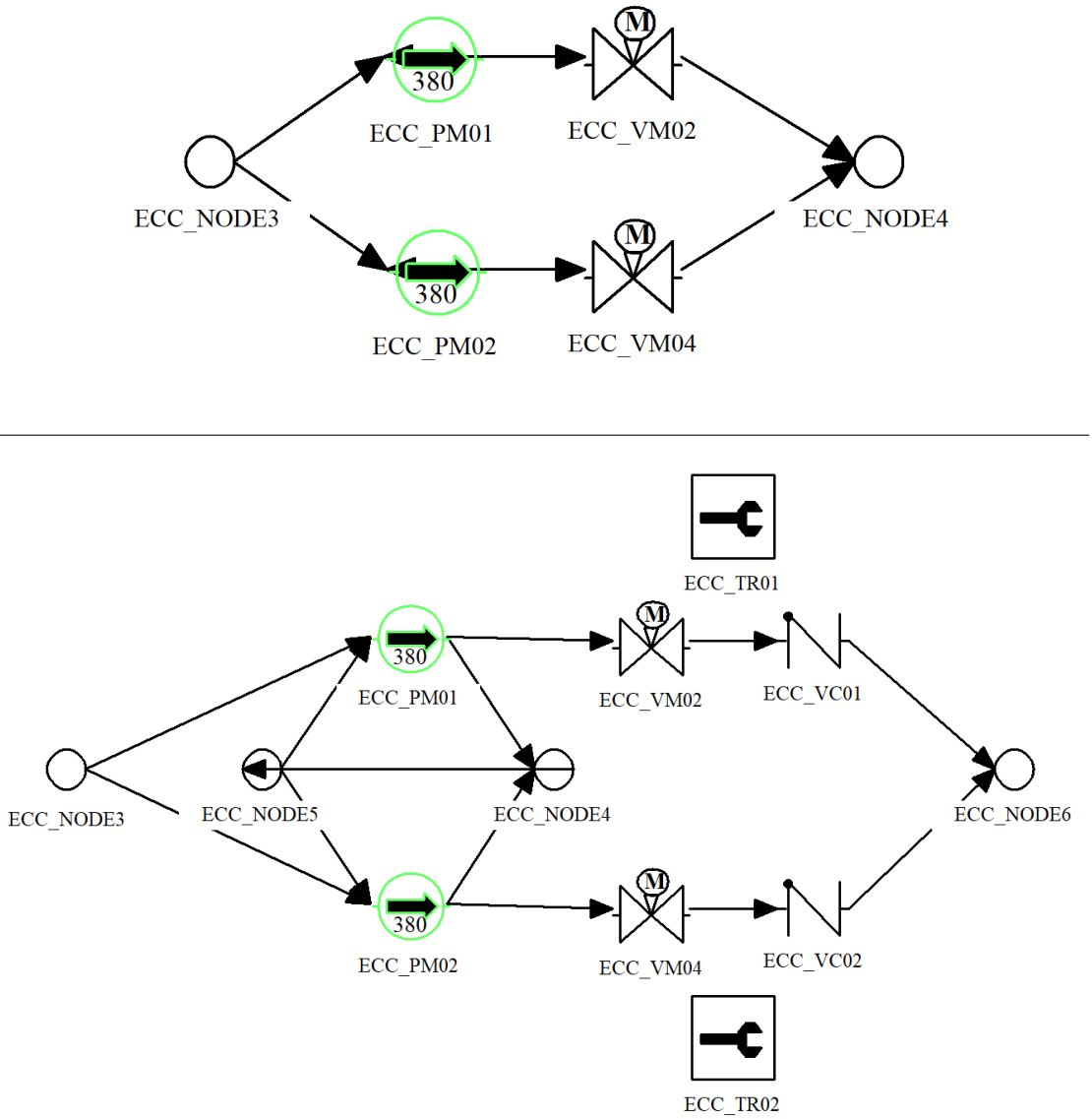


FIGURE F.3: Resulting P&IDs of the ECC System.
Above the version of the RSMB Project, below the version of the RSPSA Project.

F.4 EFW System

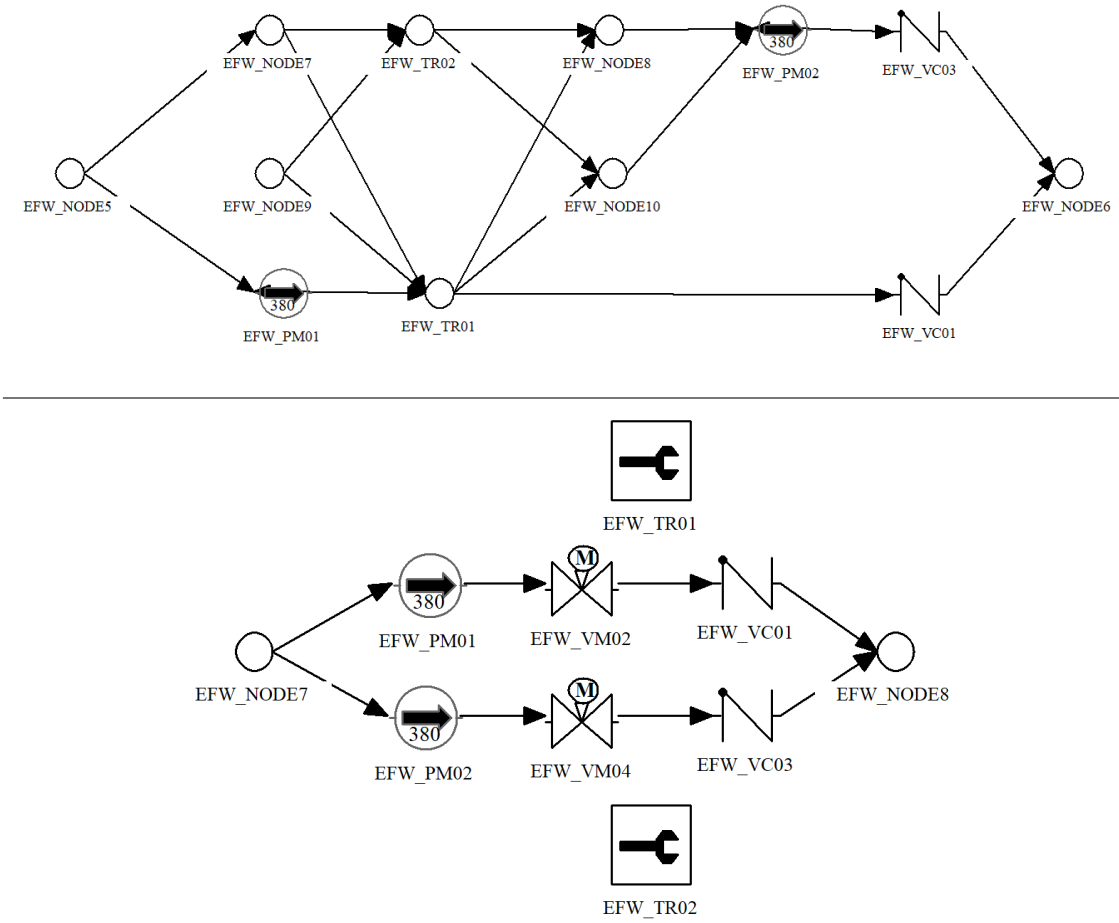


FIGURE F.4: Resulting P&IDs of the EFW System.
Above the version of the RSMB Project, below the version of the RSPSA Project.

F.5 MFW System

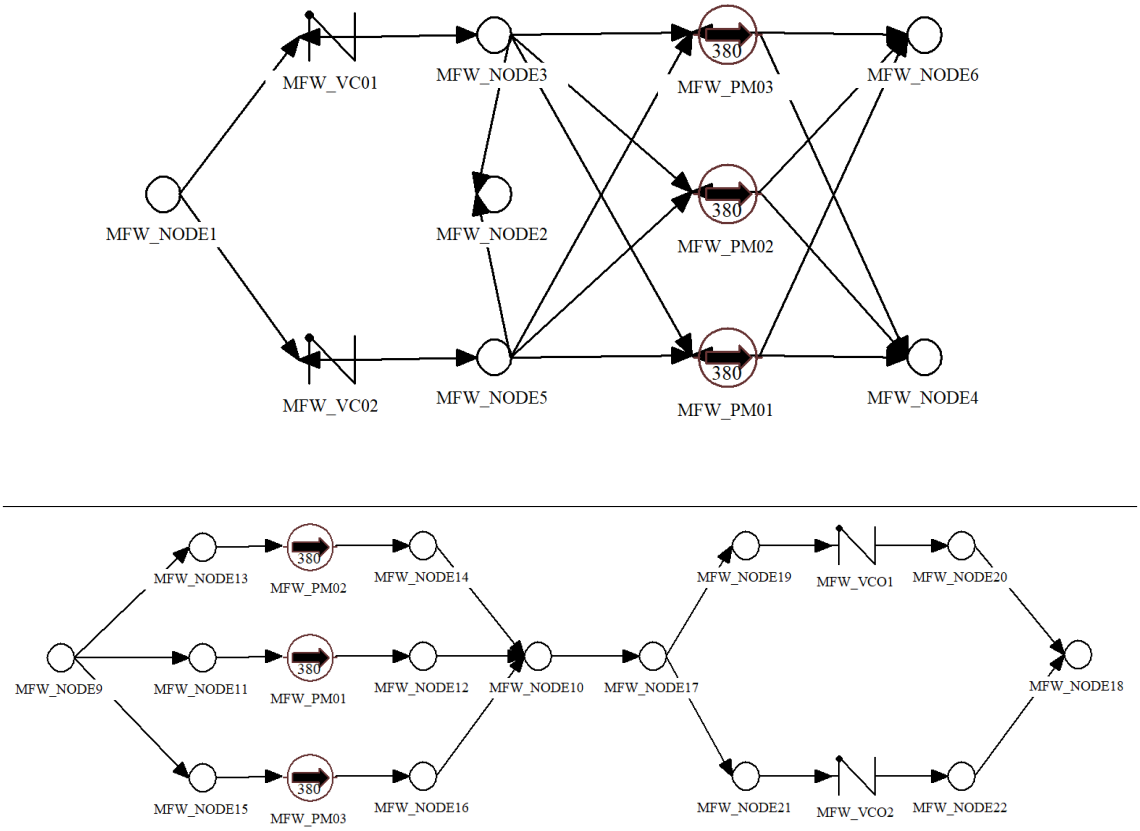


FIGURE F.5: Resulting P&IDs of the MFW System.
Above the version of the RSMB Project, below the version of the RSPSA Project.

F.6 RHR System

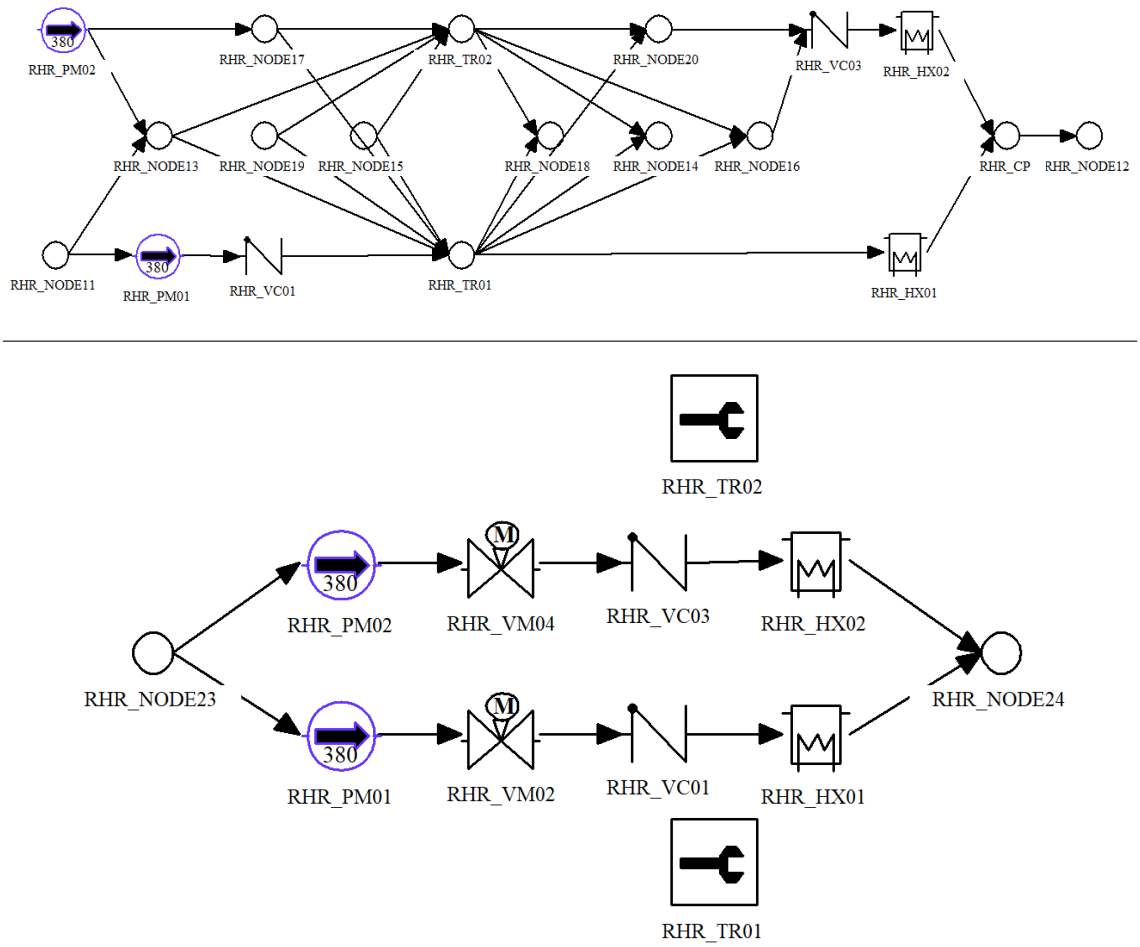


FIGURE F.6: Resulting P&IDs of the RHR System.
Above the version of the RSMB Project, below the version of the RSPSA Project.

F.7 SWS System

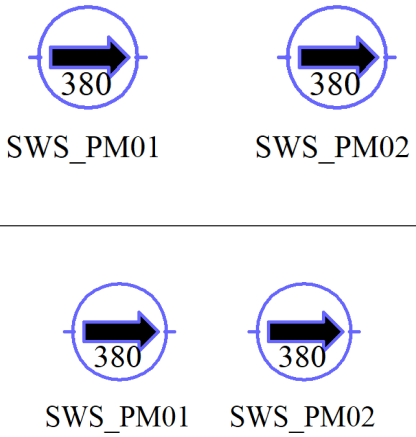


FIGURE F.7: Resulting P&IDs of the SWS System.
Above the version of the RSMB Project, below the version of the RSPSA Project.

Appendix G

SPC: Directory Tree

This appendix shows the directory tree of the SPC repository[1]. Subsection 5.1.1 elaborates further on the directory tree.

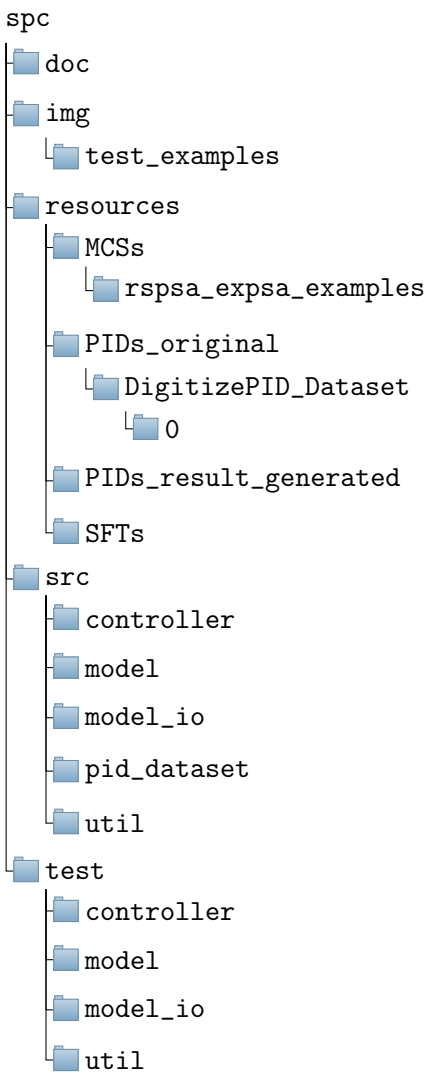


FIGURE G.1: The Directory Tree of the SPC Repository

[1] https://gitlab.com/wbos/spc

Appendix H

SPC: Design Diagrams

This appendix shows the design diagrams of the SPC implementation (see Section 5.2). Figure H.1 shows the Class Diagram of the SFTController. Section 5.4.4 explains the SFTController in more detail.

Figure H.2 shows the Class Diagram of the SPC's data model (see Section 5.4.1). Green classes correspond to the generic Graph structure. Purple classes are specific for P&IDs. Yellow classes are specific for SFTs.

Figure H.3 shows the Call Hierarchy Graph of *sft_io.py*, which imports RSA files and converts them to SFTs. Blue methods focus on defining the SFTs. Green methods focus on defining the Gates of the SFTs. Yellow methods focus on defining the Basic Events of the SFTs. Green methods focus on linking Gate inputs to the correct Gates in the SFTs.

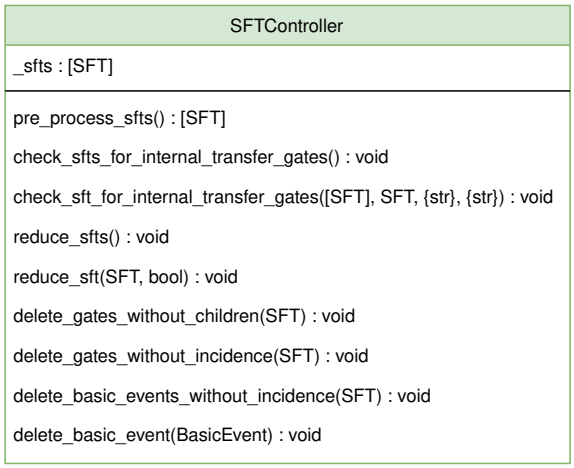


FIGURE H.1: Class Diagram of the SFTController class

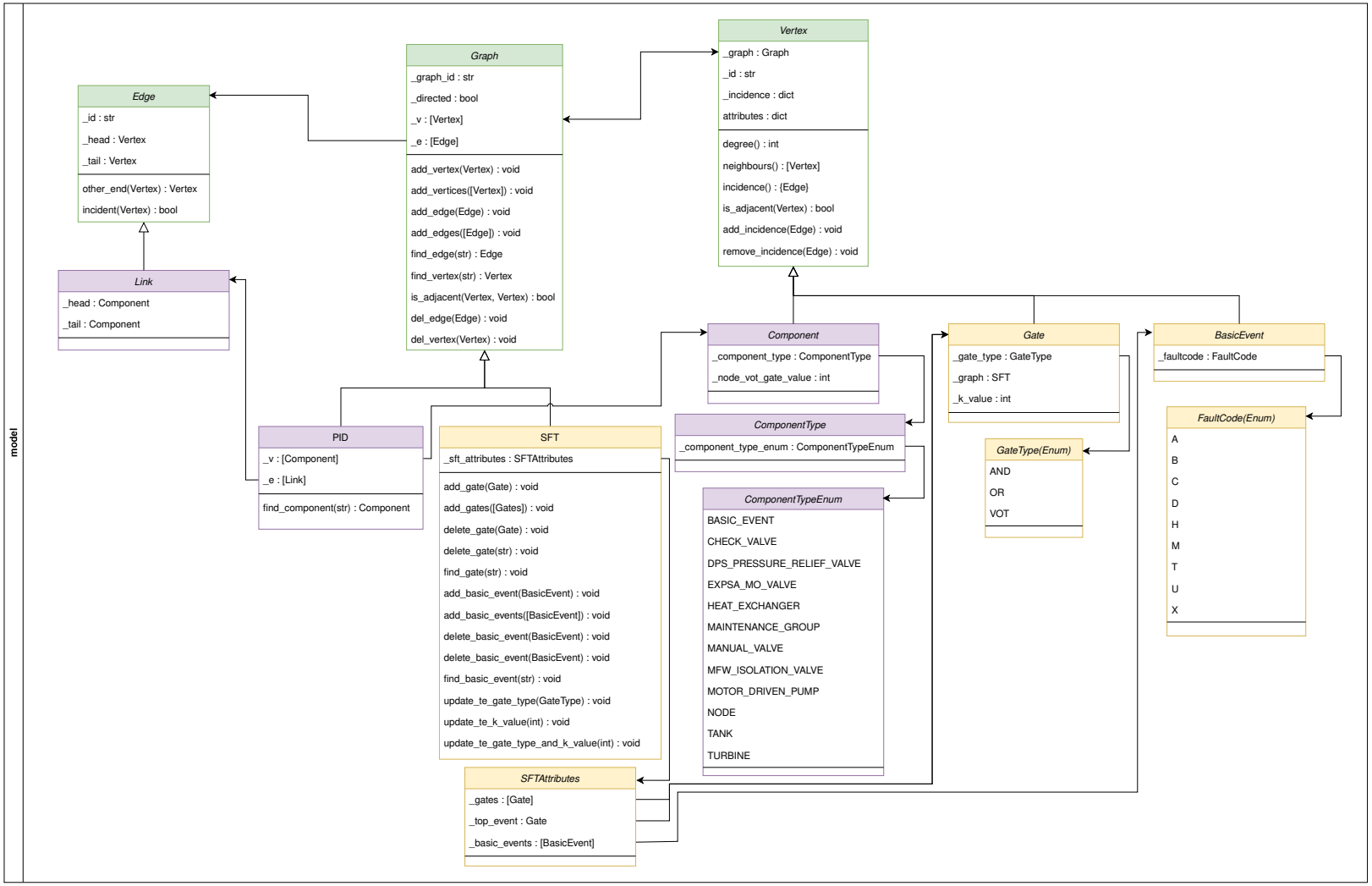


FIGURE H.2: Class Diagram covering the SPC's data model

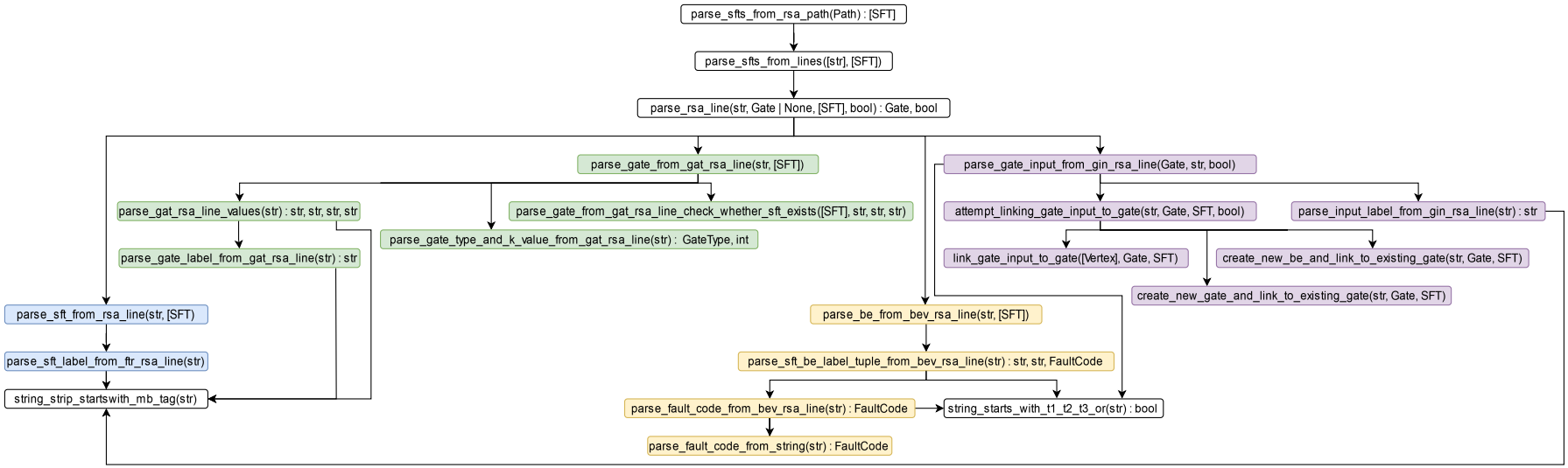


FIGURE H.3: Call Flow Graph of sft_io.py (which imports RSA files and converts them to SFTs within Python)