HiP-HOPS

Safety Analysis Tool

User Manual

For HiP-HOPS V2.5.902

January 2023

Contents

1. INTRODUCTION 4

2. ALL ABOUT HIP-HOPS 5

2.1. Why use HiP-HOPS 5

2.2. Features of the tool 6

2.3. How the tool works 7
2.3.1. Safety Analysis 7
2.3.2. Decomposition of Safety Requirements 19
2.3.3. Optimisation 22

3. INSTALLATION 26

3.1. Installing HiP-HOPS 26

3.2. Installing Sentinel UltraPro drivers 27

3.3. Setting up Matlab Simulink for use with HiP-HOPS 27

3.4. Evaluation Version 28

3.5. Example models 29

4. USING HIP-HOPS WITH MATLAB SIMULINK 30

4.1. Naming Conventions 31

4.2. Component Failure Data Editor 32

4.3. Basic Event Editor 33

4.4. PCCF Editor 37

4.5. Output Deviation Editor 38

4.6. Exported Propagation Editor 40

4.7. Component Editor 41

4.8. Allocation Editor 43

4.9. Implementation Editor 44

4.10. Model Parameters Editor 45

4.11. Hazard Editor 47

4.12. Running HiP-HOPS 48

5. HIP-HOPS OUTPUT 49

5.1. Web Browser Analysis Output 49
5.1.1. Fault Tree Output 50
5.1.2. FMEA Output 53
5.1.3. SIL Decomposition 55
5.1.4. Warnings and Errors 56
5.1.5. Circular Logic 56

5.2. Spreadsheet Output 64

5.3. Optimisation Output 65

6. HIP-HOPS TUTORIALS 67

6.1. Tutorial 1: Analysing a simple model 67
6.1.1. Creating the model 67
6.1.2. Annotating the model 68
6.1.3. Analysing the model and viewing the output 71

6.2. Tutorial 2: Hierarchical models 74
6.2.1. Creating the model 74
6.2.2. Adding failures at the subsystem level 75
6.2.3. Analysis Results 77

6.3. Tutorial 3: Common Cause Failures 78
6.3.1. Creating the model 79
6.3.2. Adding a common cause failure 79
6.3.3. Analysis Results 81

6.4. Tutorial 4: NOT gates and non-coherent fault trees 82
6.4.1. Creating the system model 83
6.4.2. Failure logic 83
6.4.3. Analysis & Results 85

6.5. Tutorial 5: SIL Decomposition 89
6.5.1. The system model 89
6.5.2. Failure data annotations 90
6.5.3. SIL Decomposition 92
6.5.4. Results 93

6.6. Tutorial 6: Architectural Optimisation 96
6.6.1. The system model 96
6.6.2. Configuring optimisation options 99
6.6.3. Viewing the optimisation results 102

7. RUNNING HIP-HOPS AS A STANDALONE ENGINE 104

7.1. Command Line Parameters 104

1. Introduction

Over 20+ years, the Dependable Systems research group in the Department of Computer Science at the University of Hull has pioneered the development of novel methods and tools for dependability analysis and optimisation of complex safety critical engineering systems collectively known as **Hierarchically Performed Hazard Origin and Propagation Studies** (or **HiP-HOPS**).

Key features of the HiP-HOPS software tool include:

- Fast algorithms for bottom up dependability analysis via automatic synthesis of Fault Trees and Failure Models and Effects Analyses (FMEAs) based on hierarchical architectural models.

- Novel algorithms for top-down semi-automatic allocation of safety requirements in the form of Safety Integrity Levels - this work automates some of the processes for ASIL allocation specified in the new automotive safety standard ISO26262.

- A novel extension of dependability analyses with genetic algorithms that solves difficult multi-objective optimisation problems in the design of architecture and maintenance of safety critical systems.

This document is the user manual for the tool and explains both the background behind how the tool functions, how to use the tool, and some tutorial examples to better understand and highlight major features.

2. All about HiP-HOPS

2.1. Why use HiP-HOPS

Fault Tree Analysis (FTA) and Failure Modes & Effects Analysis (FMEA) are classical system analysis techniques used in reliability engineering. They are methods by which we can discover information about the potential faults in a system that we can then use to correct those faults. Both are widely used in the automotive, aerospace, nuclear, and other safety critical industries.

FTA is a deductive technique, which means it works from the top down. This is done by assuming a system failure has occurred and working backwards to try to determine what possible combinations of events might have caused it; the system failure then becomes the *top event* of the fault tree, and the individual component failures form the leaf nodes (or *basic events*), and are combined through logical gates such as OR and AND. The fault tree can then be analysed either qualitatively, to determine the smallest combinations of basic events needed to cause the system failure, or quantitatively, to obtain the probability of the top event occurring.

FMEA, by contrast, is an inductive technique, and works from the bottom up. It involves proposing a certain event or condition, and then trying to assess the effects of that initial event on the rest of the system. The end result is a table of failures and their effects on the system, which provide the analyst with an overview of the possible faults.

Both techniques are useful and provide valuable information about systems, but both suffer from the same flaw: they are primarily manual techniques, and the process of performing these analyses can be laborious, especially for larger and more complex systems. In such cases, it is more likely that the analyst will make a mistake, or that the results once obtained are too numerous to interpret efficiently.

This problem means that both FTA and FMEA tend to be performed only once, either after the system has been designed, in order to check its reliability, or after the system has been put into operation and has failed, in order to find out what went wrong. This is unfortunate, because both FTA and FMEA are potentially very useful when they are integrated into the design process itself, so that a system can be designed with safety and reliability in mind. By using these system analysis techniques as part of an iterative design process, it is possible to identify and remedy potential flaws and faults much earlier, thereby saving both time and effort and producing a more reliable product.

However, before FTA and FMEA can be incorporated into the design process in this way, they need to be automated in some fashion, so that they can be carried out much more quickly and efficiently, and thus maximising their contribution to the design. The HiP-HOPS Fault Tree Synthesis (FTS) tool is intended to achieve such a goal. By including reliability annotations as part of the system model, HiP-HOPS can examine the model and automatically construct and analyse both fault trees and FMEAs. The result is a semi-automated process which takes much of the burden off the system designer and speeds up the analysis considerably, allowing the designer to quickly identify weak points in the model and take steps to remedy them.

2.2. Features of the tool

HiP-HOPS (Hierarchically Performed Hazard Origin & Propagation Studies) works with the modelling package used by the designer to obtain information about the reliability of the system being modelled. By annotating the model with data describing how individual components can fail, HiP-HOPS can then take that data and produce a series of fault trees from it, and from those trees it can generate a wealth of reliability information presented in the form of an FMEA table.

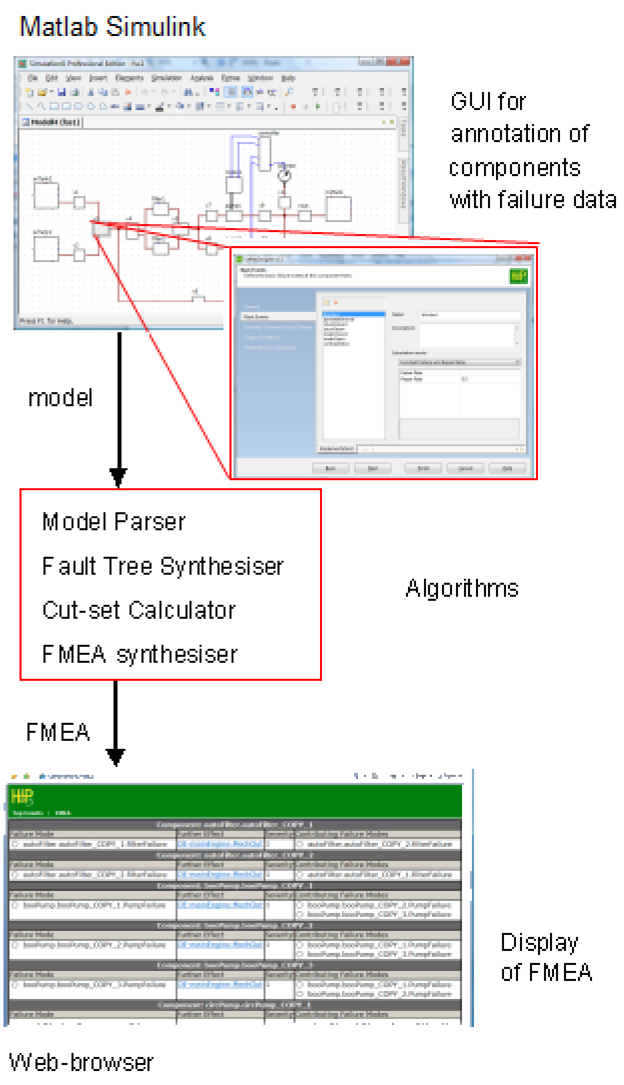


Figure 1: The HiP-HOPS Process

HiP-HOPS has been integrated with modelling package Matlab Simulink, allowing greater feedback to the user and a wider range of functionality.

The tool allows the user to load a model of a system to be loaded and parsed. Once this internal representation of the model has been loaded, the tool synthesises fault trees for every system failure in the model, and combines them to create the FMEA.

During this process, a lot of other useful analysis is carried out; the *minimal cut sets* are obtained, which are the minimum combinations of basic events required to cause the top event, and the *unavailability*, or probability, of the top event is also calculated.

These results are integrated into the resultant FMEA tables, which are presented in the form of hyperlinked web pages. This format allows the designer to locate a specific failure and click on it to find the effects it has on the rest of the system.

This whole process is illustrated in Figure 1, from the annotation of the model with the failure data at the top, through the fault tree synthesis and analysis, and culminating in the FMEA. Best of all, it does this in a matter of seconds or minutes, not days.

Thus, in summary, HiP-HOPS has the following capabilities:

- Works with Matlab Simulink to analyse annotated model files generated by that program;
- Can parse those models and synthesise fault trees from them;
- Can quickly generate minimal cut sets;
- Calculates unavailability for the top events;
- Generates FMEA and FTA results in the form of hyperlinked web pages;
- Can perform semi-automatic decomposition of safety requirements in the form of Safety Integrity Levels (SILs, ASILs, DALs etc);
- Can perform multi-objective optimisation of the models to produce a set of trade-off solutions to choice of alternative components and best sites for redundancy allocation.

2.3. How the tool works

The features of the tool can be divided into three main parts. The first part is safety analysis which performs a single pass analysis of the annotated model. The second is decomposition of SILs across the system architecture, which uses an optimisation process to find the 'cheapest' allocations. The third is optimisation which uses a multi-objective genetic algorithm to automatically improve the dependability characteristics of the model. The optimisation tool makes use of the analysis functions to evaluate the automatically generated alternative designs. These three functions will be discussed now in greater detail.

2.3.1. <u>Safety Analysis</u>

First the process of providing a single pass analysis is described.

Overview

HiP-HOPS analysis consists of three main phases. The first phase consists of annotating the system model with the failure data needed to produce the fault trees and perform the analysis. This phase must be integrated into the modelling tool used to design the system model, since they are so closely related. In this case, the failure data is entered via a graphical user interface within Matlab, as explained in section 4.

The second phase of HiP-HOPS analysis is the fault tree synthesis process. In this process, the tool examines the system model and its failure data and combines it to create a series of fault trees. It works by taking the failures of system outputs and working backwards through the model to determine which components caused those failures. These failures are then joined together using the appropriate logical operators to construct fault trees with the failures at the system outputs as the top events and the root causes as basic events.

The third and final phase of HiP-HOPS takes the newly constructed fault trees and analyses them. The result is an FMEA, which is a combination of all the information stored in the fault trees and presented in the form of a table listing the effects of each component failure on the rest of the system. As part of this process, more analysis is carried out on the fault trees, both *qualitative* (logical) and *quantitative* (numerical). This provides both the minimal cut sets of the fault tree and the unavailability of the top event.

Annotation Phase

Before any fault trees can be generated, the tool needs to know how the various components of the system are interconnected, and how each one can fail. The structural data is provided by the model itself, which shows the basic topology of the system and the connections between the various components and subsystems. The models can also be hierarchically arranged, so that systems can be decomposed into subsystems which each have their own components. The types of models which HiP-HOPS can be applied to are varied: fluidic systems, electrical or electronic systems, mechanical systems, and even more conceptual data flow based models are all suitable, as long as they can be augmented with failure data.

The failure data is what needs to be entered separately; each component or subsystem needs its own *local failure data*, which describes what can go wrong with that component and how it responds to failures elsewhere in the system. This is achieved by annotating the model with a set of failure expressions showing how deviations in the component outputs (*output deviations*) can be caused either by internal failures of that component (which become basic events) or corresponding deviations in the component's inputs. Such deviations include unexpected omission of output or unintended commission of output, or more subtle failures such as incorrect output values or the output being too early or late. This logical information explains all possible deviations of all outputs of a component, and so provides a description of how that component fails and reacts to failures elsewhere. Once done, the data can then be stored in a library, so that other components of the same type can use the same failure data. This avoids the designer having to enter the same information many times.

At the same time, numerical data can be entered for the component, detailing the probability of internal failures occurring and the severity of output deviations. This data will then be used during the analysis phase to arrive at a figure for the unavailability of each top event. Other information can be added at this stage to facilitate more advanced analysis, including the option to enter information for different implementations of a component to allow an optimisation algorithm to determine the most efficient choice of component to use, and in the future, also the addition of temporal data to describe more complicated failure behaviour.

Because all of this data needs to be entered for each component, it is necessary for this to be done as part of the modelling tool itself, where the parts of the system are readily visible. This requires a user interface to be created to allow for the data entry that is separate to the main part of HiP-HOPS itself, and instead integrated with the modelling tool directly.

In order to introduce some of the major concepts, a simple example of a standby recovery system, shown in
Figure 2, will be used.

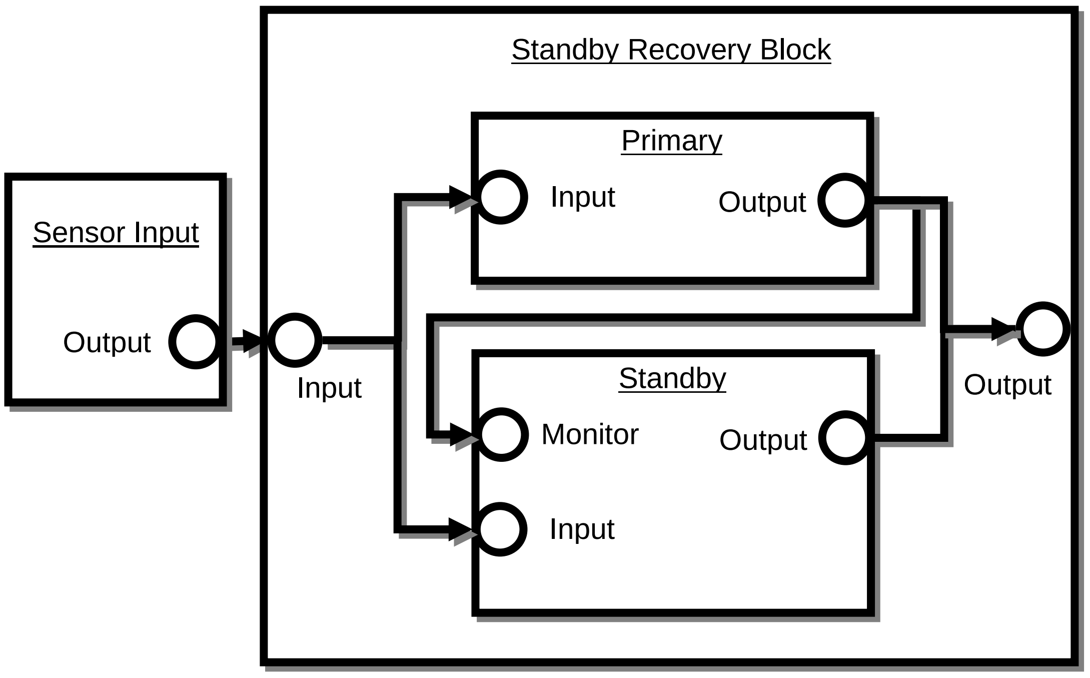


Figure 2: A simple example of a standby recovery system

It is composed of one input component, 'SensorInput', and one subsystem, 'Standby Recovery Block', which is composed itself of two sub-components, 'Primary', and 'Standby'. 'Primary' is the main subcomponent of the standby-recovery system and processes the input from the 'SensorInput'. The 'Standby' component monitors the output from 'Primary' and is designed to take over operation if it detects a failure of the 'Primary'.

Table 1 shows the failure modes of the components in the example system:

Component	Failure Modes	Probability
Standby Recovery Block	ElectroMagneticInterference	0.001
Primary	InternalFailure	0.03
Standby	InternalFailure	0.02
SensorInput	InternalFailure	0.05

Table 1: Failure modes for standby recovery system

Table 2 shows the failure annotations for the components in the example system.

Component	Output deviations	Failure expressions
Standby Recovery Block	Omission - Output	Omission – Primary.Output AND Omission – Standby.Output OR ElectroMagneticInterference
Primary	Omission – Output	Omission – Input OR Failure
Standby	Omission – Output	Omission – Monitor AND (Omission – Input OR Failure)
SensorInput	Omission - Output	Failure

Table 2: Failure data for standby recovery system

Table 2 defines that the omission of output of the SensorInput component is caused only by its internal failure.

The 'Monitor' port detects omissions of output of the 'Primary' component before activating the 'Standby' component. If the 'Primary' component is functioning then the 'Standby' cannot fail (or its failure is irrelevant) as it is inactive. If the 'Standby' component is thus activated then an output omission is caused by either an internal failure or the propagation of input omission.

An omission of 'Primary' output is caused by an internal failure or the propagation of an omission of input.

At the top level, the system output of the 'Standby Recovery Block' has an output deviation of type 'Omission' that is caused by either the failure mode 'electromagnetic interference', or by the conjunction of omissions occurring at the outputs of both the primary and standby components.

The Boolean failure expressions in the component annotations each describe a Component Fault Tree (CFT) that describes propagation of failure through it. However, such a fault tree is incomplete because its leaf nodes will not all be failure modes – some may be input deviations, which are failures originating in other components. Similarly the top node is an output deviation that may be relevant in further components in the system.

These CFTs for the example are shown in Figure 3 (for the 'Standby Recovery Block'), Figure 4 (for the 'Primary' component), Figure 5 (for the 'Standby' component), and Figure 6 (for the 'SensorInput' component). In these diagrams the circles with an arrow denote an input deviation where the arrow is entering the circle and an output deviation where the arrow is leaving the circle.

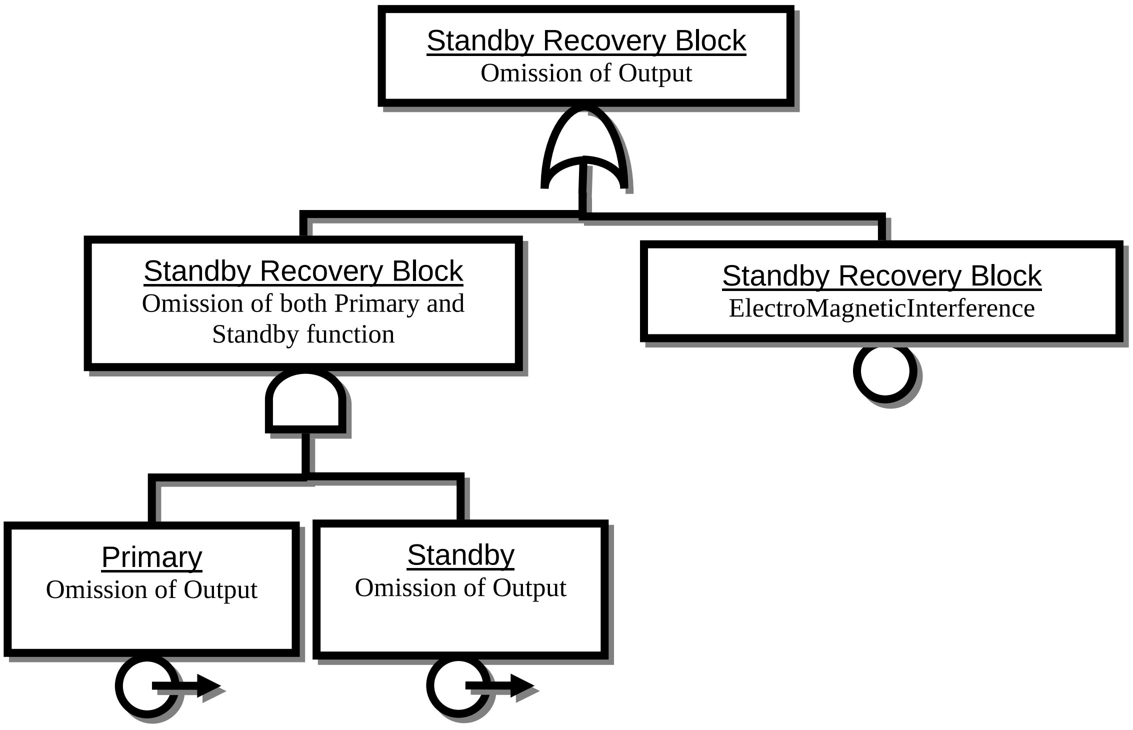


Figure 3: CFT described by annotation for standby recovery block subsystem

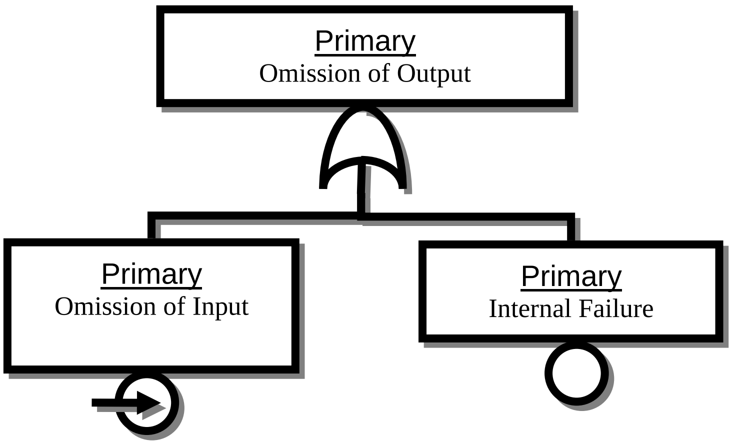


Figure 4: CFT described by annotation for primary component

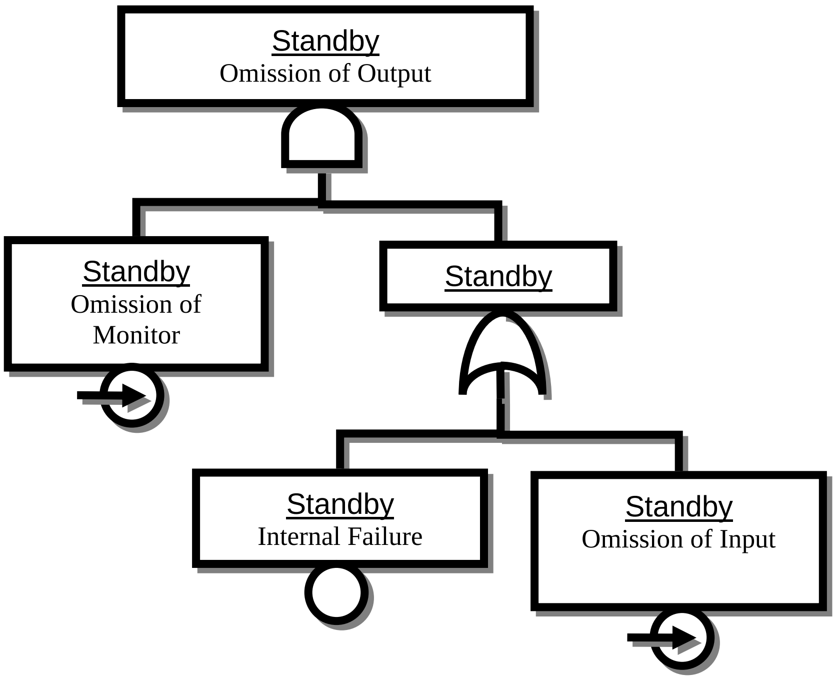


Figure 5: CFT described by annotation for standby component

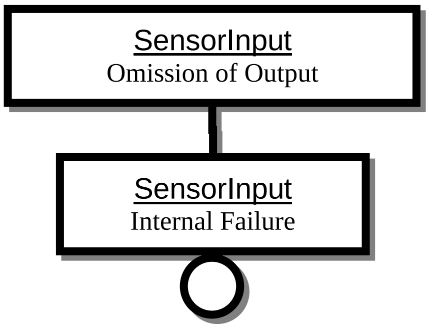


Figure 6: CFT described by annotation for SensorInput component

Once the components of the model have been fully annotated, the manual phase of HiP-HOPS is complete and the remaining stages are fully automatic.

Synthesis Phase

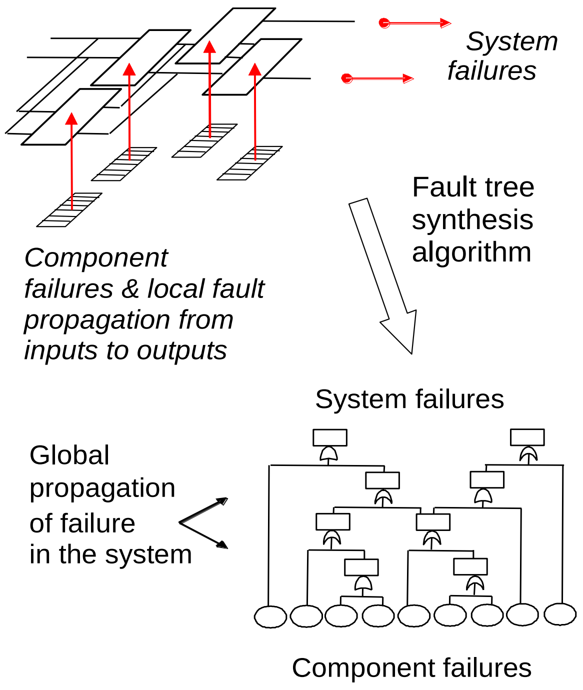

System failures
Fault tree synthesis algorithm
Component failures & local fault propagation from inputs to outputs
System failures
Global propagation of failure in the system
Component failures

Figure 7: The synthesis of fault trees from the system model

Once the local failure data has been entered into the system model, the model can then be given to HiP-HOPS proper for synthesis to take place. This phase functions by examining how local failures of components propagate through the model and cause failures at the outputs of the system. It is therefore necessary to be able to identify which parts of the model function as outputs of the system, so that the failures of these parts become top level failures of the system as a whole. This is done by defining *hazards*, top-level system events with safety or reliability implications, which then link to output deviations within the system.

In the case of an engine, for example, then one system output could be the application of motive power. A failure to provide this would be one possible hazard and would be treated as a starting point for the synthesis. In addition, unexpected output (i.e. the engine moving when it should not) could be another hazard.

For each of these hazards, HiP-HOPS generates a local fault tree describing the causes of that hazard in terms of output failures of components within the system. The tool then examines the causes of those output failures, which are typically internal faults and inputs to those component, and in turn travels back to see what output failures may be propagated to the inputs. It then makes local fault trees for those output failures, and so on until there are no connected components remaining. It then goes back through and combines the local fault trees into a single fault tree for each possible hazard, which shows how that hazard is caused by combinations of malfunctions or other failures of components elsewhere in the system. For our engine, this means we would have one fault tree for “no engine power on demand” and one fault tree for “unexpected engine power”.

The synthesis phase begins with the causes of the system-level hazards, which are always output deviations. The algorithm locates the CFT for each output deviation and traverses the tree until it locates a terminal input deviation. The input port that is associated with that input deviation is then selected.

The algorithm then follows the connections from the selected component port to the output ports of the connected components. The output deviation matching the failure class of the connected input deviation is then selected. Its CFT is joined to the input deviation from the connected component.

This cycle of traversing CFTs and connections is repeated until there are no more unconnected input deviations and the system fault trees are complete.

In the standby recovery example in
Figure 2, the system output is the omission of output of the 'Standby Recovery Block'. This is the top node of a CFT in the 'Standby Recovery Block' and marks the start of the synthesis for this example.

This CFT has two input deviations at its leaves, 'omission of Primary output' and 'omission of Standby output', and so the algorithm finds the corresponding components where it discovers further CFTs. The top nodes of these are added as child branches to the input deviation leaf nodes and the process is repeated, each time connecting output deviation CFTs to input deviation leaf nodes.

The omission of 'Primary' output is partly caused by an omission of its input, so the connection at the input is followed to the 'SensorInput' component where the output deviation that exists there is connected to the growing system fault tree. As the deviation of 'SensorInput' output is only caused by an internal failure, the propagation of that branch is terminated.

The 'omission of Standby input' branch is treated in the same way and thus a complete system fault tree, describing the propagation of failure throughout the whole model, is synthesised from the failure expressions. The result of synthesis of the example model is shown in
Figure 8. It can be seen from this diagram how the system fault tree is composed of the mini fault trees shown in Figure 3 to Figure 6. The output deviations that are the top nodes of the component fault trees are shaded.

In the next HiP-HOPS phase the fault tree is analysed to extract quantitative and qualitative information.

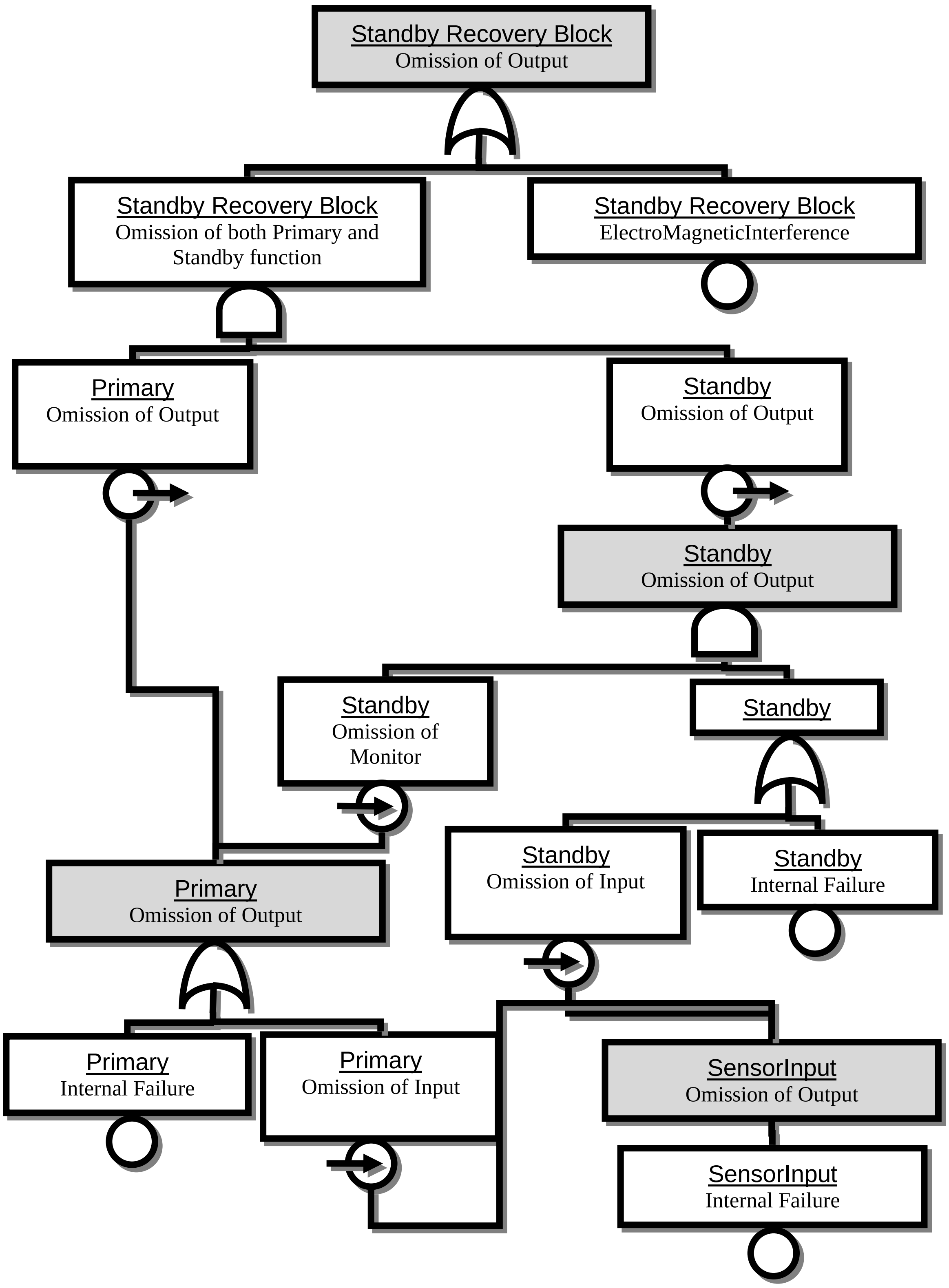


Figure 8: Fault tree synthesised from standby recovery example.

Analysis phase

The result of the synthesis process is a set of one or more interconnected fault trees and therefore the next stage of the HiP-HOPS technique is to analyse those fault trees using FTA. The fault trees represent the propagation of failure logic through the system, but they can often be large and complex. By reducing the fault trees to their minimal cut sets we retain the relationship between the basic events and the top level system event but strip out the intermediate propagation paths and any redundant causes.

The main computational expense when minimising the cut sets is the redundancy checking. Several methods of increasing the performance of this process, including modularisation, fault tree contraction, and use of cut set cataloguing, are applied for this purpose.

The following Boolean laws can be applied to obtain minimal cut sets:

- The Law of Absorption: E1 + ~~E1.E2~~ = E1
    - The cut set containing E1.E2 was removed as the action of E1 alone is sufficient to cause the top event and is therefore in its minimal form.
    - E1 + E1 = E1 is also a form of Absorption

- The Laws of Idempotence: E1.~~E1~~ = E1 and E1 + ~~E1~~ = E1
    - The former removes repeated events within cut sets and the latter removes repeated cut sets.

HiP-HOPS also supports the use of NOT gates and non-coherent fault trees. A NOT gate is typically used to indicate that a basic event must *not* occur (which is called a "complement" event). For example, we can say that an output deviation is caused by a value failure of one input as long as there is not an omission of the other input, e.g.:

- Omission-out = Value-In1 AND NOT Omission-In2

However, the use of NOT gates (indicated by the '~') introduces potential contradictions and implicit causes according to the Consensus Law:

- Contradictions: E1.~E1 = 0
- The Consensus Law: E1.E2 + ~E1.E3 = E1.E2 + ~E1.E3 + **E2.E3**

Generating and testing these implicit causes can incur a significant performance penalty, so it is recommended to avoid NOT gates where possible.

In order to keep the number of checks to a minimum the cut sets are checked for redundancy as they are created so that redundant combinations are quickly identified and removed. This ensures that they cannot affect or be combined with more cut sets later in the traversal of the fault tree.

Once the minimum cut sets have been identified, they can subsequently be used for quantitative analysis to calculate the system unavailability Q_S (where basic events have quantitative data) using the approximate Esary-Proschan method:

$$Q_S = 1 - \prod_{i=1}^{n}(1 - Q_{cut_i})$$

(Where n is the number of independent cut sets and Q_{cuti} is the unavailability of the cut set i).

The system failure frequency can also be similarly calculated using the Esary-Proschan method:

$$\omega_S = \sum_{i=1}^{n} \omega_{cuti} \prod_{j=1, j \neq i}^{n} (1 - Q_{cutj})$$

(Where ω_{cuti} is the frequency of cut set i and Q_{cutj} is the unavailability of cut set j).

The unavailability and frequency of a single cut set are given by:

$$Q_{cut} = \prod_{i=1}^{n} Q_i$$

and

$$\omega_{cut} = \sum_{j=i}^{n} \omega_j \prod_{i=1, i \neq j}^{n} Q_i$$

(Where Q_i and w_j are the unavailability and frequency of each of the n *events in the cut set.)*

In addition to the quantitative analysis that can be performed on the minimal cut sets, a further qualitative stage can be applied to generate an FMEA. Figure 9 shows the inverse relationship between the diagnostic failure propagation information in the fault trees, where the component failure modes that cause a system failure can be determined, and the causative nature of the FMEA, where a basic event (or combination of several events) have an effect on the system level.

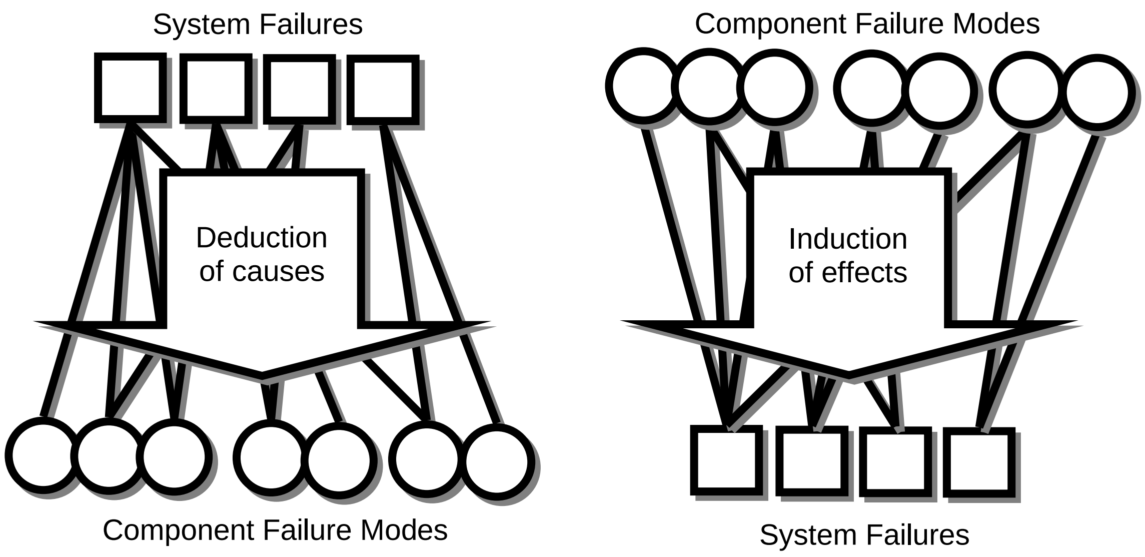
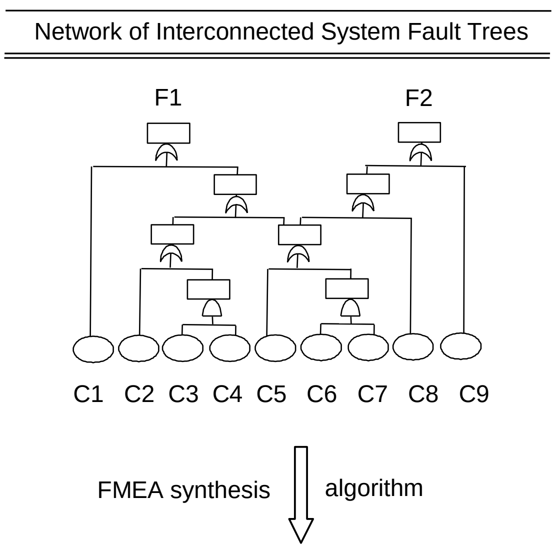


Figure 9: Inverse relationship between fault trees (left) and FMEA (right)

The minimal cut sets contain the non-redundant propagation of failure in the fault tree and an algorithm is used to catalogue each component failure mode in each fault tree and note which system failures they cause and in combination with which other component failure modes. This information is the core of an FMEA.

The deductive nature of this process is important for safety analysis as it allows large combinations of basic events to be considered in the FMEA, unlike traditional manual methods that could only consider single points of failure or fault injection simulation methods that are similarly limited by combinatorial explosion.

The last step is to combine all of the data produced into an FMEA, which is a table that concisely illustrates the results. The FMEA shows the direct relationships between component failures and system failures, and so it is possible to see both how a failure for a given component affects everything else in the system and also how likely that failure is. However, a classical FMEA only shows the *direct effects* of single failure modes on the system, but because of the way this FMEA is generated from a series of fault trees, HiP-HOPS is not restricted in the same way, and the FMEAs produced also show what the *further effects* of a failure mode are; these are the effects that the failure has on the system when it occurs in conjunction with other failure modes. Figure 10 shows this concept.



Multiple Failure System FMEA

Component failure	Direct effects on the system	Effects caused in conjunction with (other events)
C1	F1	-
C2	F1	-
C3	-	F1 (C4)
C4	-	F1 (C3)
C5	F1,F2	-
C6	-	F1,F2 (C7)
C7	-	F1,F2 (C6)
C8	F2	-
C9	F2	-

Figure 10: The conversion of fault trees to FMEA

In Figure 10, F1 and F2 are system failures, and C1 – C9 are component failures. For C3, C4, C6 and C7, there are no direct effects on the system – that is, if only one of these components fail, nothing happens. However, they do have further effects; for example, C3 and C4 both occurring in conjunction will cause F1 to occur.

The FMEAs produced, then, show all of the effects on the system, either singly or in combination, of a particular component failure mode. This is especially useful because it allows the designer to identify failure modes that contribute to multiple system failures (e.g. C5 in the example of Figure 10). These common cause failures represent especially vulnerable points in the system, and are prime candidates for redundancy or extra reliable components.

2.3.2. Decomposition of Safety Requirements

It is common practice in safety standards like IEC 61508 or ISO 26262 to indicate the stringency of the safety requirements applying to a system function or element with a Safety Integrity Level (or SIL). Different standards have different conventions, but in general SILs represent a qualitative measure of required risk reduction, typically on a scale, e.g. from 1 to 4 with 1 being least strict/dependable and 4 being most strict/dependable. ISO 26262 uses Automotive SILs (ASILs), which range from A (least dependable) to D (most dependable), while ARP 4754 defines Development Assurance Levels (DALs), ranging from A – E, with E being least dependable and A being the most.

SILs are typically defined as part of risk analysis in relation top-level safety hazards. The higher the risk the hazard represents (either in terms of severity, likelihood, or both), the higher the SIL likely to be assigned to it. The intention is that if the corresponding system components are developed to the appropriate standard of safety represented by the SIL, the risk will be brought down to acceptable levels.

However, decomposition of SILs can be a complex problem. There can be many, many contributing components to any given hazard, and if *all* components inherited the SIL of the hazard they contribute to, then the cost of developing the system could be prohibitively expensive.

As such, most standards allow for a higher SIL to be met by a suitable combination of independent components with lower SIL values. The simplest way to think of this is to assign an integer to each SIL and sum them. For example, ISO 26262 allows an ASIL D requirement to be met by a pair of independent components like so (where QM = quality management only, i.e., no special safety requirements):

Component 1	Component 2	Overall
QM (0)	ASIL D (4)	0 + 4 = 4 (ASIL D)
ASIL A (1)	ASIL C (3)	1 + 3 = 4 (ASIL D)
ASIL B (2)	ASIL B (2)	2 + 2 = 4 (ASIL D
ASIL C (3)	ASIL A (1)	3 + 1 = 4 (ASIL D)
ASIL D (4)	QM (0)	4 + 0 = 4 (ASIL D)

This helps reduce cost by distributing the burden of meeting a safety requirement over multiple independent components (e.g. typically, two ASIL B components would be less expensive than a single ASIL D component). However, it also produces a combinatorial challenge of assigning suitable SILs to individual components (or even failure modes of those components) that minimises cost while continuing to meet overall system safety requirements.

Because HiP-HOPS builds up a model of causation and failure propagation via its fault tree synthesis, it knows which component failure modes contribute to which hazards. Thanks to the AND gates in the fault trees, it also knows how combinations of failure modes lead to the hazards. This information can therefore be used as the basis of an automatic SIL decomposition process. An exhaustive evaluation of the search space would be infeasible, since the number of potential allocations increases very quickly with every AND gate, so a Tabu search-driven optimisation process is applied.

To perform SIL decomposition, each hazard needs to be assigned an integer SIL value. A cost heuristic may also be defined; otherwise, a default heuristic where cost = SIL x 10 is used (e.g. SIL 1 = 10, SIL 4 = 40 etc). Then HiP-HOPS will attempt to assign a SIL to every basic event such that the overall requirements are met while the total cost is minimised.

For example, consider a simple scenario with three components, C_A, C_B, and C_C, each with a single internal failure mode, A, B, and C respectively. Two hazards are defined:

- H1 = A AND B
- H2 = B OR C

where H1 = SIL 4 and H2 = SIL 2.

We could naively test every possible combination of SILs to the three failure modes, but most of these will either fail to meet the requirements imposed on the hazards (i.e., A + B >= 4, B >= 2, and C >= 2) or will be unnecessarily strict and thus unnecessarily expensive (e.g. assigning SIL 4 to all three).

In this case, the example is small enough that we can explore the possibilities manually. For A and B, we know they must total 4, thus there are 5 "optimal" allocations:

A	B
0	4
1	3
2	2
3	1
4	0

However, we also have a second constraint: B and C must both be at least 2. Thus we can eliminate the last two rows of this table, since either B would fail to meet the requirement (e.g. A=3, B=1) or, if we increase B, we increase cost for no gain (A=2, B=3 is more expensive than needed given that A=2, B=2 is sufficient).

Therefore we have three possible "valid" allocations that are not needlessly strict. We can also calculate the cost of each possible allocation using the default heuristic (cost = SIL x 10):

A	B	C	Cost
0	4	2	60
1	3	2	60
2	2	2	60

In this case all possibilities are "optimal", as none are cheaper than any other while all respect the overall constraints. However, with a different cost heuristic, the results may change. For instance, if the cost heuristic was instead 10^{SIL} (so SIL 1 = 10, SIL 2 = 100, SIL 3 = 1000, SIL 4 = 10000), the results change:

A	B	C	Cost
0	4	2	10100
1	3	2	1110
2	2	2	300

Then in this case, the third option (allocation SIL 2 to each) is the clear victor.

As mentioned, the complexity of this process increases rapidly with the number of components, hazards, and AND gates in the logic, meaning that to perform it exhaustively (let alone manually) soon becomes infeasible. An optimisation process is used to instead explore the search space and arrive at suitable allocations much more efficiently, though it is not guaranteed to find all optimal solutions.

Care must also be taken to ensure that the AND logic truly does represent independent components. If there is a common cause then typically failures cannot be considered independent and thus it would be inappropriate to decompose SILs across them.

2.3.3. Optimisation

HiP-HOPS and a multi-objective optimisation algorithm have been combined to generate un-dominated trade-off solutions that meet dependability criteria. The principle could apply to any optimisation algorithm but the algorithm used is the NSGA-II algorithm, a multi-objective genetic algorithm that uses a Pareto-based selection method to encourage a wide, even spread of trade-off results.

Figure 11 shows a fuel oil service system for a cargo ship. The example will be expanded and used to illustrate concepts throughout this section.

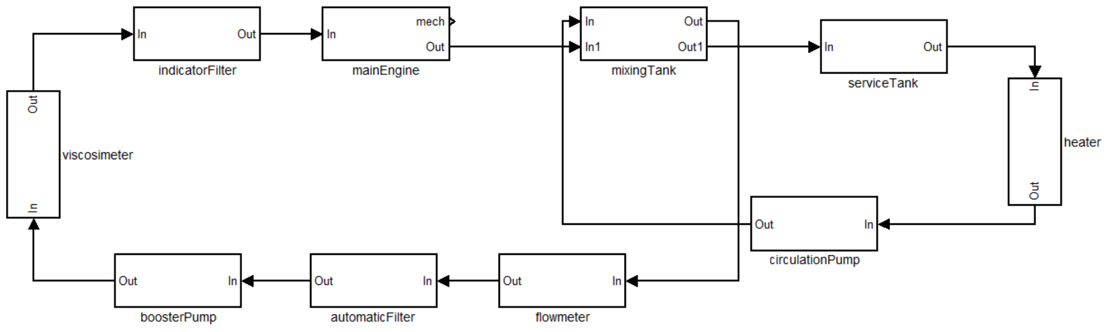


Figure 11: Fuel oil service system for a cargo ship.

When the fuel oil system fails, there is a loss of engine propulsion (OmissionEnergy-mainEngine.mech) that can lead to the ship becoming grounded as a result of drifting.

The following tables contain the implementation failure data for the components in the example in Figure 11. The main engine provides the system output (an omission of energy at the mechanical output of the engine) which is caused by an omission of flow of oil at the input. It has no internal failures.

mainEngine		
Output Deviation	**Description**	**Failure logic (Propagation)**
OmissionEnergy-mech (System output)	**Omission of energy at the mechanical output of the mainEngine caused by an omission of flow of oil at the input**	**OmissionFlow-In**

The other components in the system (indicator filter, viscosimeter, pre-heater, circulation pump, mixing tank, flow meter, automatic filter, booster pump, and service tank) each contain 3 alternative subsystems that define different levels of parallel redundancy (Figure 12 to Figure 14). In each case the propagation of the omission of flow of oil is combined in the 'AND' block so that both (or all three) redundant components must fail to cause failure of the subsystem.

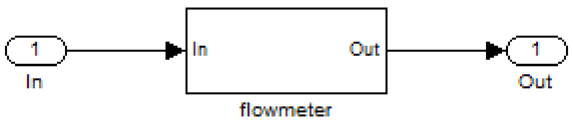


Figure 12: Alternative 1: no redundancy

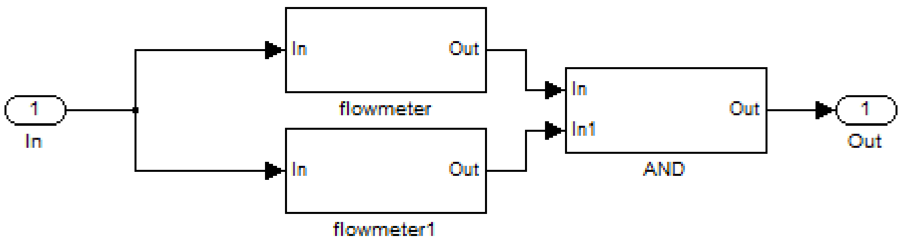


Figure 13: Alternative 2: one parallel redundancy

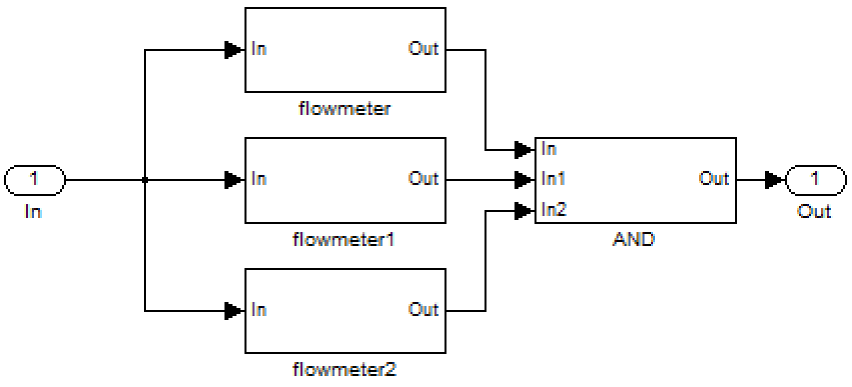


Figure 14: Alternative 3: two parallel redundancies

All of the subcomponents have the same failure propagation logic. The omission of flow of oil at the output is caused by the omission of flow of oil at the input, or an internal failure of the component.

All other system subcomponents		
Output Deviation	**Description**	**Failure logic (Propagation)**
OmissionFlow-Out	**Omission of flow of oil at the output can be caused either by an omission of flow of oil at the input or an internal failure mode of the component**	**OmissionFlow-In or [component]Failure**

Each of the subcomponents has 3 alternative implementations with different costs and the internal failure modes have different failure rates.

	Alternative 1		**Alternative 2**		**Alternative 3**	
Components	**Cost**	**Failure Rate**	**Cost**	**Failure Rate**	**Cost**	**Failure Rate**
Indicator filter	1500	5.0E-7	2500	2.0E-7	3222	1.0E-7
Viscosimeter	2500	2.5E-6	3178	1.0E-6	3814	5.0E-7
Pre-heater	2000	6.7E-6	2505	5.0E-6	3956	1.0E-6
Circulation pump	6000	3.2E-5	13380	2.0E-5	18000	7.0E-6
Mixing tank	2000	1.6E-5	2963	8.0E-6	4444	2.0E-6
Flow meter	2000	1.0E-5	3000	1.0E-6	4444	5.0E-7
Automatic filter	2000	1.0E-5	2647	5.0E-6	3529	1.0E-6
Booster pump	5000	3.2E-5	10682	2.0E-5	12500	5.0E-6
Service tank	1500	1.6E-5	1957	5.0E-6	2739	1.0E-6

It is possible to flag both components and individual implementations for inclusion/exclusion from consideration during the optimisation.

During optimisation the genetic algorithm can generate different potential solutions by selecting between specified alternative implementations of the components and subsystems. In some cases this can be simply replacing a component with a more reliable or less expensive (but functionally equivalent) version. It is also possible to select between replacement subsystems which may employ different fault tolerant architectures such as parallel redundancy or standby recovery.

The combination of these two mechanisms allow all aspects of the architecture to be altered by the genetic algorithm.

As the algorithm promotes survival of the fittest, it is necessary to evaluate the fitness of each new individual and be able to compare it to other individuals in the population.

- The evaluation uses the model based evaluation tool HiP-HOPS to calculate the objective values of the individuals.

- Cost and weight values are calculated by summing the values for all the components used in the solution.

- Unavailability is calculated through HiP-HOPS analysis, first traversing the model to mechanically synthesise interconnected system fault trees, converting the fault trees into their minimal cut sets, and finally using an order 3 inclusion exclusion algorithm as detailed in the existing HiP-HOPS specification.

- Reliability is equal to *(1 – unavailability)* where repair rate = 0.

- A safety metric could be established qualitatively from the cumulative severity of all single points of failure in a given design.

- Risk can be calculated from failure probabilities and severities of the top events of the fault trees produced by the tool.

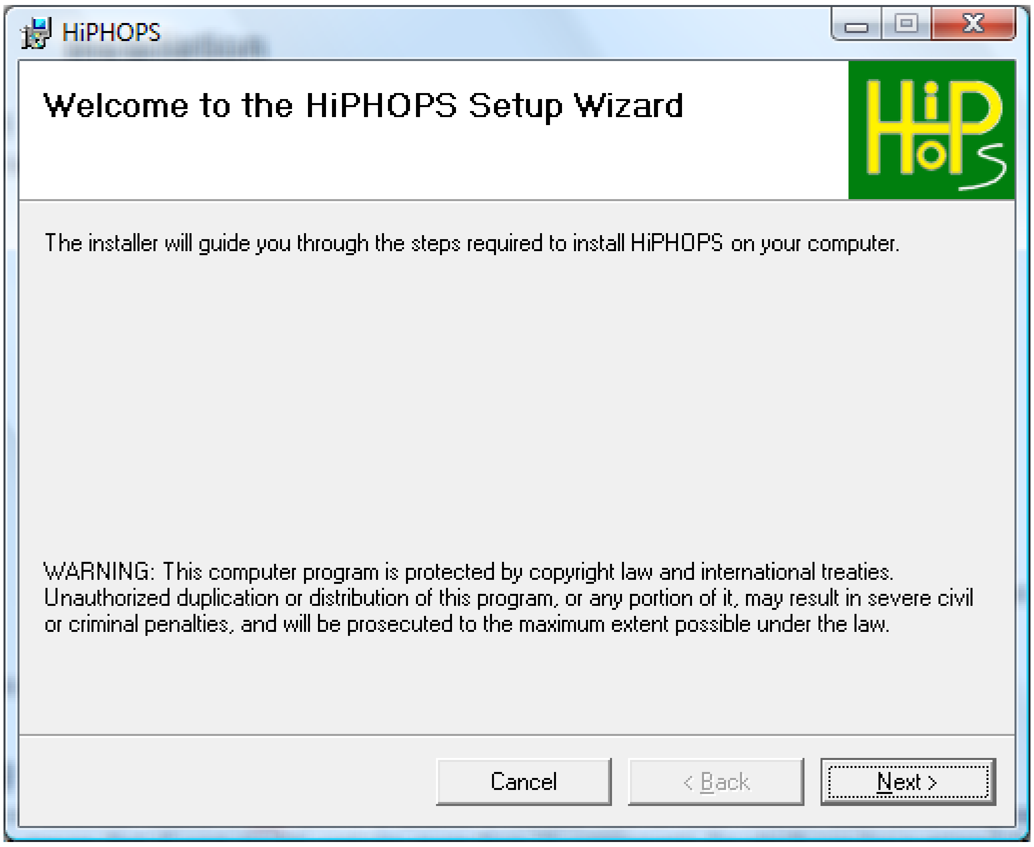
Once the model has been evaluated, calculating the values that correspond to the objectives specified by the user, the values are stored with the individual encoding. The evaluation is complete and the individual will not be changed therefore the model can be deleted for memory efficiency.

3. Installation

3.1. Installing HiP-HOPS

Installing HiP-HOPS is very simple:

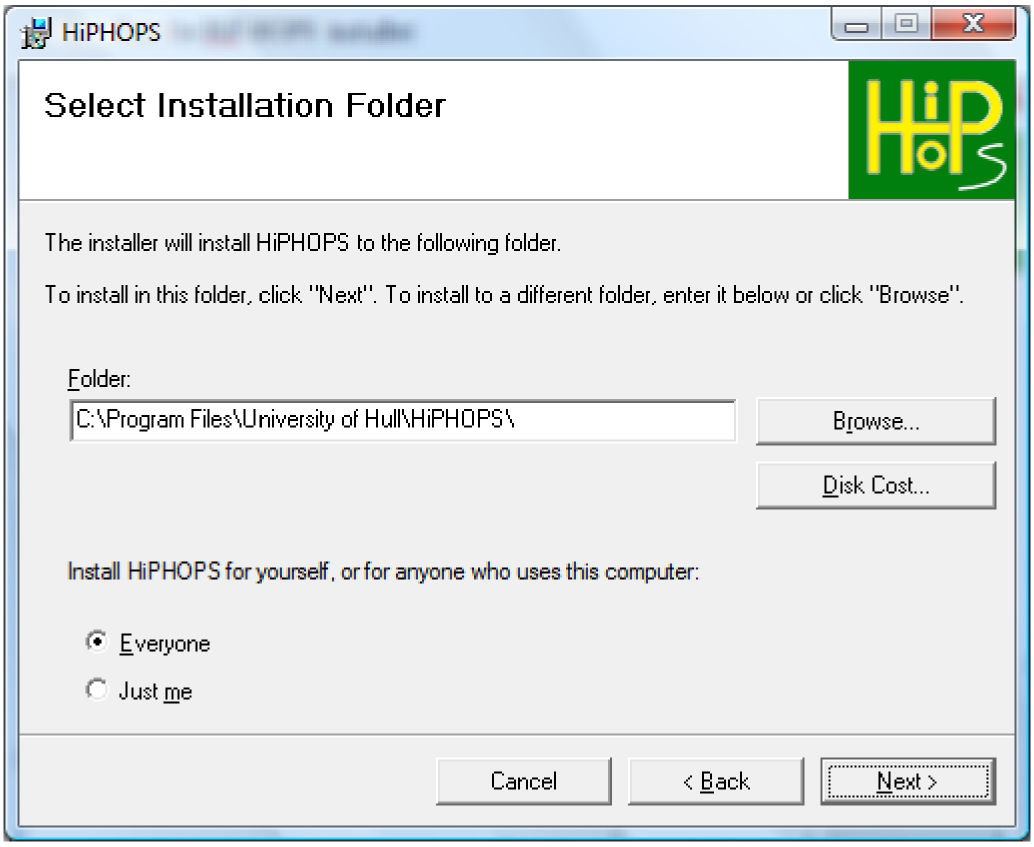
1. Run the HiP-HOPS installer:



The installer will guide you through the steps required to install HiPHOPS on your computer.

WARNING: This computer program is protected by copyright law and international treaties. Unauthorized duplication or distribution of this program, or any portion of it, may result in severe civil or criminal penalties, and will be prosecuted to the maximum extent possible under the law.

2. Select a directory to install to and choose which profiles to make HiP-HOPS visible for:



HiPHOPS

Select Installation Folder

The installer will install HiPHOPS to the following folder.

To install in this folder, click "Next". To install to a different folder, enter it below or click "Browse".

Folder:
C:\Program Files\University of Hull\HiPHOPS\

Install HiPHOPS for yourself, or for anyone who uses this computer:

[✓ Everyone]
Just me

3. Click "Next";

4. Click “Next” again to confirm;

5. Close the installer when finished.

HiP-HOPS can be uninstalled by using the Programs and Features settings menus.

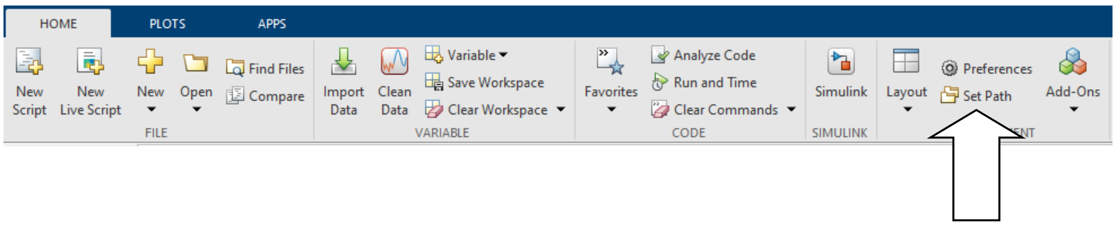
3.2. Installing Sentinel UltraPro drivers

The non-evaluation version of HiP-HOPS requires the Sentinel UltraPro hardware usb key to be connected to the PC. To detect the key, the Sentinel drivers need to be installed. The driver installer can be downloaded from each version posted on the Downloads section of the HiP-HOPS website.

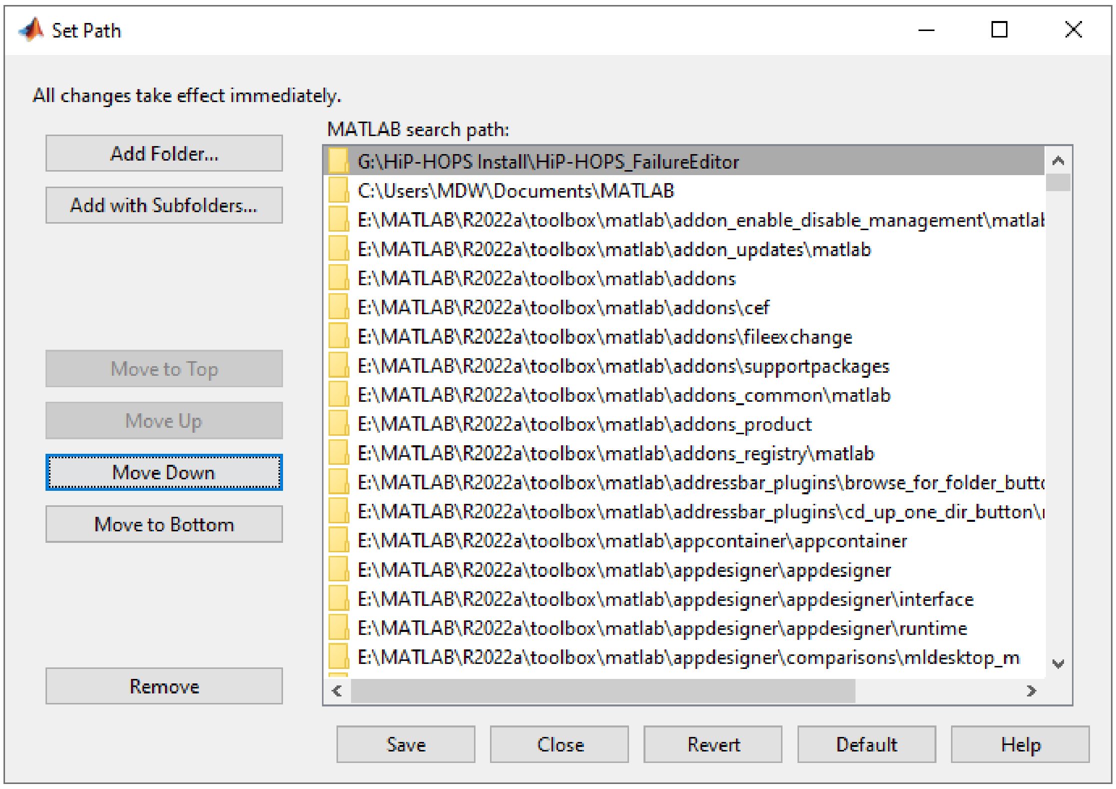
3.3. Setting up Matlab Simulink for use with HiP-HOPS

An interface for using HiP-HOPS within Matlab Simulink has been developed to allow the HiP-HOPS to be applied to Simulink models. The commands are operated through the use of buttons in the HiP-HOPS Launcher.

When you first install HiP-HOPS you need to add the Interface folder to the Matlab path. To do this click on the Set Path button in the "Home" ribbon:



This opens the path menu, as shown below.



All changes take effect immediately.

MATLAB search path:

```
G:\HiP-HOPS Install\HiP-HOPS_FailureEditor
C:\Users\MDW\Documents\MATLAB
E:\MATLAB\R2022a\toolbox\matlab\addon_enable_disable_management\matlal
E:\MATLAB\R2022a\toolbox\matlab\addon_updates\matlab
E:\MATLAB\R2022a\toolbox\matlab\addons
E:\MATLAB\R2022a\toolbox\matlab\addons\cef
E:\MATLAB\R2022a\toolbox\matlab\addons\fileexchange
E:\MATLAB\R2022a\toolbox\matlab\addons\supportpackages
E:\MATLAB\R2022a\toolbox\matlab\addons_common\matlab
E:\MATLAB\R2022a\toolbox\matlab\addons_product
E:\MATLAB\R2022a\toolbox\matlab\addons_registry\matlab
E:\MATLAB\R2022a\toolbox\matlab\addressbar_plugins\browse_for_folder_butt
E:\MATLAB\R2022a\toolbox\matlab\addressbar_plugins\cd_up_one_dir_button\
E:\MATLAB\R2022a\toolbox\matlab\appcontainer\appcontainer
E:\MATLAB\R2022a\toolbox\matlab\appdesigner\appdesigner
E:\MATLAB\R2022a\toolbox\matlab\appdesigner\appdesigner\interface
E:\MATLAB\R2022a\toolbox\matlab\appdesigner\appdesigner\runtime
E:\MATLAB\R2022a\toolbox\matlab\appdesigner\comparisons\mldesktop_m
```

Click on the 'Add Folder' button and use the file dialog window to navigate to the location where you installed HiP-HOPS. Select the HiP-HOPS_FailureEditor folder and click the 'OK' button. Click on 'Save' to commit the path entry and then you can click on 'Close' to exit the window.

You can test that the path was set up correctly by attempting to start the HiP-HOPS Launcher. To do this, type '`hiphops`' (without quotes) into the Matlab command window. If successful, the launcher will appear.

You are then ready to use HiP-HOPS in Matlab. Information on how to conduct modelling and failure annotation is found in the following sections.

3.4. Evaluation Version

The evaluation version of HiP-HOPS is limited to models with no more than 20 annotated components. This means that if your model contains more than 20 components for which you have entered failure data, i.e., basic events and output deviations, then the evaluation version will not run. If you are running the evaluation version, it will say so in the command window when you run it.

3.5. Example models

The installation process creates some example Matlab model files in the 'Examples' subfolder of the HiP-HOPS installation folder. These are the same models described in section 6 and can be studied to see how various HiP-HOPS features are applied in practice. These six examples/tutorials are:

1. **tutorial1_standbyRecovery.slx**: A basic model following the standby-recovery system described above, used to explain basic modelling and annotation concepts.

2. **tutorial2_standbyRecoverySubsystem.slx**: The same system, but introduces subsystems and subcomponents to explain hierarchical modelling.

3. **tutorial3_commonCauseFailures.slx**: The standby-recovery system again, but introduces common cause failures.

4. **tutorial4_notgates.slx**: A simple 3 component example demonstrating how NOT gates can be used in the failure logic.

5. **tutorial5_silDecomposition.slx**: A more complex example covering a hybrid braking system. As well as being a more realistic system, it demonstrates the use of the SIL decomposition feature.

6. **tutorial6_optimisation.slx**: This model describes the ship fuel oil system mentioned above and is used to demonstrate the optimisation concept. Its various subsystems are also included as separate files (e.g. optimisationExampleFlow2.mdl, optimisationExampleFlow3.mdl, etc).

4. Using HiP-HOPS with Matlab Simulink

NB: If using HiP-HOPS with Simulink for the first time, the path to HiP-HOPS must first be set up as described in section 3.3.

To use HiP-HOPS with Matlab Simulink, you need to open or create a Simulink model and launch HiP-HOPS by typing '`hiphops`' (without quotes) into the Matlab command window. If successful, the launcher will appear:

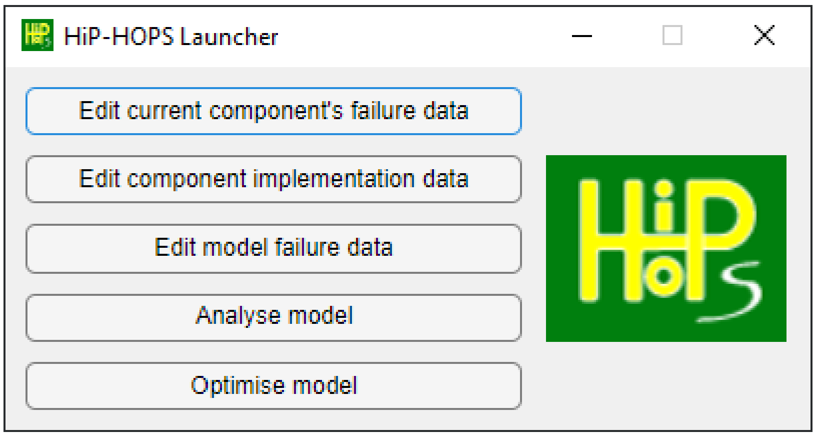

HiP-HOPS Launcher
Edit current component's failure data
Edit component implementation data
Edit model failure data
Analyse model
Optimise model

Figure 15: HiP-HOPS Launcher

The Launcher is the main access point to all HiP-HOPS functions and should be kept open while modelling (though you can move the Launcher around as you like). The five buttons in the Launcher are as follows:

1. **Edit current block's failure data:** This button allows you to edit the failure data of the current component with the Component Failure Data Editor (see section 4.2). In order to function, a component (block) must be selected in the Simulink window. It will open an interface that enables editing of basic events and output deviations etc.

2. **Edit component implementation data:** This button opens the Component Editor (see section 0) for the component implementations (for optimisation purposes) but also for perspectives and common cause failures if multi-perspective modelling is being used. As with the first button, it requires a component (i.e., a block) to be selected in the Simulink window.

3. **Edit model failure data:** This enables editing of the model-level information using the Model Parameters Editor (4.10), of which the most important are the hazards (which serve as the starting point for analysis).

4. **Analyse model:** This button begins a HiP-HOPS safety analysis of the current model.

5. **Optimise model:** This button begins an optimisation of the current model.

Each interface will be described in further detail below. Note that all of the editor windows have a "Save and Close" button (and a "Cancel" button) at the bottom. If you wish to save the data, it is important to click the "Save and Close" button. Pressing Cancel or closing the window with the X in the top-right will discard any data, whether newly added data or editing of existing data.

4.1. Naming Conventions

Before discussing the interface, it is important to note the following naming conventions, both within the HiP-HOPS editor windows and within any Simulink models used with HiP-HOPS:

Failure to follow these conventions is likely to lead to an error when running HiP-HOPS.

Simulink Components and Ports
All Simulink components and their input/output ports should be named with a combination of letters, numbers, and underscores, but must start with a letter. Upper case and lower case letters are both permitted. Accented letters (e.g. é) should also work. **Do not use spaces in names.**

Valid examples: `component1, Valve, particulate_filter, output1, in1, SignalOut, fuel_in`

HiP-HOPS Basic Events, Common Causes, and Exported Propagations
These should all follow the same rules as components and port names: they should start with a letter, followed by any combination of letters, numbers, and underscores.

Valid examples: `blocked, short_circuit, tempOver90, floodInC5`

HiP-HOPS Input and Output Deviations
These consist of two parts, separated by a dash. The first part is the failure class, a custom user-defined name for a particular type of failure (e.g. omission, commission). These failure classes should all follow the same naming rules as above. The second part, after the dash, is the name of the port where the deviation occurs and as such should follow the rules for ports as described above.

Valid examples: `omission-out1, commission-input, too_high-signalIn, LowPressure-FuelOut`

Uniqueness
In almost all cases, it is important that names are unique within their context. Components in the same subsystem should have unique names, ports within the same component should have unique names, basic events within a component should all be uniquely named, and so forth.

Fully qualified names
In some circumstances, HiP-HOPS requires the use of *fully qualified names*, which allow it to correctly identify a component anywhere in the system hierarchy. The format of a fully qualified name is as follows:

perspective :: subsystem . component . port

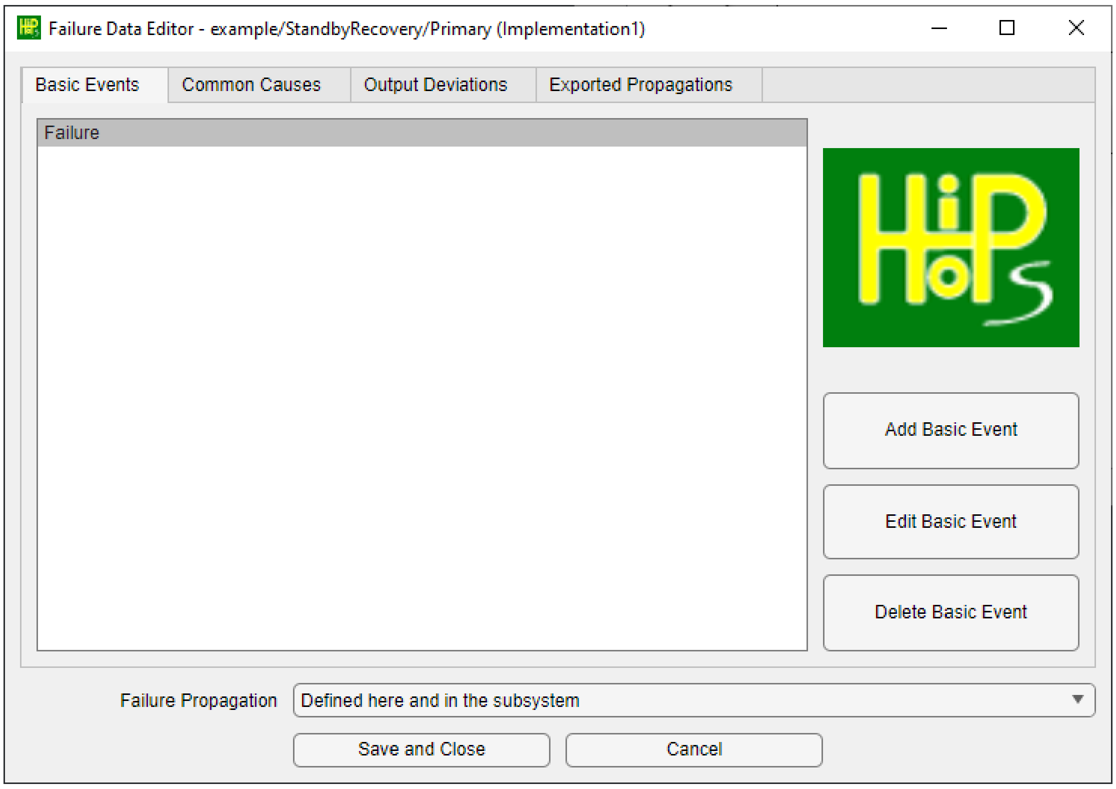
But the exact format depends on what is being referenced. For ports and basic events, the above is accurate, e.g.:

• hardware::signalProcessor.out	(a port)
• hardware::cpu.overheating	(a basic event)
• valve.in1	(a port in a model without perspectives)
• c1.c1b.c1b4.output	(a port in a deep component hierarchy)

For deviations, the failure class always goes first:

- omission-hardware::signalProcessor.out (an omission at a port)

4.2. Component Failure Data Editor


Failure Data Editor - example/StandbyRecovery/Primary (Implementation1)

Failure Propagation: Defined here and in the subsystem

Figure 16: The Component Failure Data editor

The Component Failure Data Editor is the most used editor. It allows you to annotate the selected component with failure information. This information is divided into four categories, each of which has a tab of its own:

- **Basic Events** allows editing of the basic events (component failures).
- **Common Causes** allows editing of the potential common cause failures (PCCFs) in this component.
- **Output Deviations** allows definition of the deviations at the component's output as well as the logical expressions describing their causes.
- **Exported Propagations** allows definition of deviations that export to a different perspective.

All four tabs feature a list of currently defined failure data of that category on the left along with three buttons on the right: an Add button (to create a new entry of the given type), an Edit button (to edit the selected entry), and a Delete button (to delete the selected entry).

At the bottom of the interface is a drop-down selection box that enables you to specify how this component propagates failures across different hierarchical levels. There are three options:

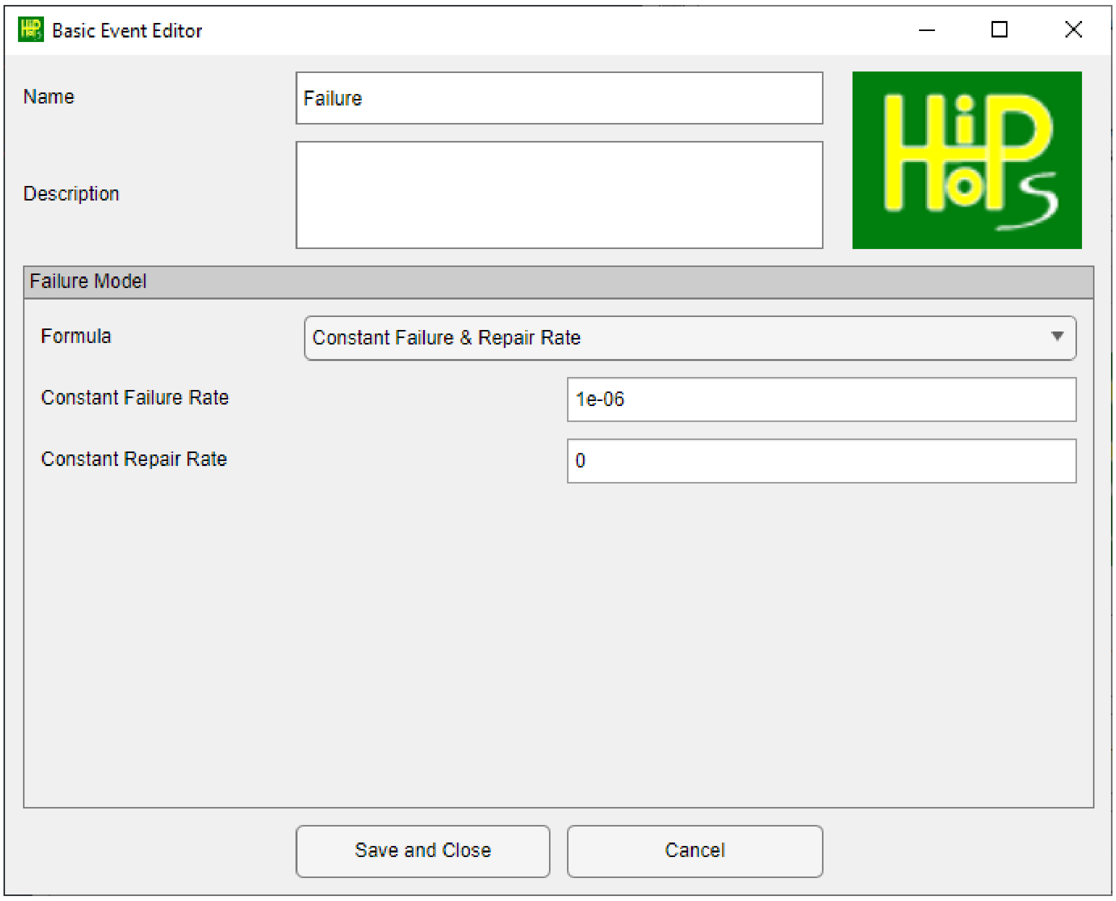
- **Defined here and in the subsystem**: If selected, the component will propagate both failures defined on the current level as well as from any output deviations defined in the component's subsystem (if it has one). This is the default and most commonly used option.
- **Defined here only**: The component will propagate only failures defined on the current level. Any output deviations defined in the component's subsystem (if it has one) are ignored. This can be useful when attempting to diagnose problems with the model's failure data, since it allows you to temporarily ignore subsystem data without deleting it.
- **Defined in the subsystem only**: If selected, the component will propagate only output deviations defined in the component's subsystem (if it has one).

As mentioned earlier, to save any data that has been added or edited, the Save and Close button must be pressed.

Note also that all HiP-HOPS windows except for the Launcher are modal. **You will not be able to edit anything else until you have closed them.**

4.3. Basic Event Editor

Adding or editing a basic event will open up the Basic Event Editor:



Basic Event Editor

Name	Failure
Description	

Failure Model

Formula	Constant Failure & Repair Rate
Constant Failure Rate	1e-06
Constant Repair Rate	0

Figure 17: Basic Event Editor

This allows you to edit the information for a basic event (i.e., a component failure), including its name, its description, and (if appropriate) the quantitative failure model and associated parameters.

Basic event names can contain any combination of letters, numbers, and underscores, but **must begin with a letter**. No spaces or other symbols are permitted. Note also that the names of all basic event for a given component must be unique, though basic events of different components can have the same name.

The description field has no such limitations and is intended primarily for the benefit of the user.

The Failure Model allows you to add quantitative failure data to the basic event which will allow HiP-HOPS to estimate its probability. There are a number of different formulae, each with different parameters, and each will yield a different unavailability for the basic event. These can be selected from the drop down menu, changing the available parameter fields in the process, and you can then fill in the appropriate parameters.

The current formulae are listed below.

Constant Failure and Repair Rate

Parameters:

λ	- Failure Rate
μ	- Repair Rate

This is the currently implemented calculation method, and assumes an exponential distribution. The formula for unavailability is as follows:

$$u = \frac{\lambda}{\lambda + \mu} \times (1 - e^{-(\lambda+\mu)t}) \qquad (4.1)$$

Constant Failure and Mean Time To Repair (MTTR)

Parameters:

λ	- Failure Rate
MTTR	- Mean Time To Repair

Very similar to above, with the exception that the repair data is entered as the MTTR instead. This is then converted to the repair rate μ (assuming μ = 1 / MTTR) and the unavailability calculation 4.1 above is used.

Mean Time To Failure(MTTF) and constant Repair

Parameters:

MTTF	- Failure Rate
μ	- Repair Rate

Very similar to above, with the exception that the repair data is entered as the MTTF instead. This is then converted to the failure rate λ (assuming λ = 1 / MTTF) and the unavailability calculation 4.1 is used.

Mean Time to Failure and Repair

Parameters:

MTTF	- Mean Time To Failure
MTTR	- Mean Time To Repair

Like the last, except that both the failure and the repair data are entered as mean times. These values will be converted and the formula in 4.1 used.

Fixed Unavailability

Parameters:

Unavailability	- The constant unavailability

This is the option to choose if you already knows the unavailability of the event.

Binomial Failure Model

Parameters:

λ	- Failure Rate
μ	- Repair Rate
n	- Number of components
m	- Minimum components needed to fail to cause subsystem failure
T	- The time of operation of the subsystem

This model is useful for representing situations where *m* failed components out of *n* will result in failure, such as in a voter. The formula is as follows:

$$u = \sum_{k=m}^{n} \left(\frac{n!}{k! \times (n-k)!} \times q^{k} \times (1-q)^{(n-k)}\right) \tag{4.2}$$

Poisson Failure Model

Parameters:

λ	- Failure Rate
n	- Number of components in operation at any one time
s	- Number of spare components available
t	- Time of operation of the subsystem

This method can be used to model the effects of limited numbers of replacement components. Formula:

$$u = \sum_{k=0}^{s} \left(\frac{x^{k} \times e^{-x}}{k!}\right) \tag{4.3}$$

Dormant Failure Model with periodic inspection

Parameters:

λ	- Failure Rate
MTTR	- Mean time to repair
T	- Time between inspections

This model can be used when a component is a standby component and only activated when its primary fails. Formula:

$$u = \frac{\lambda t - (1 - e^{-\lambda t}) + \lambda \times MTTR \times (1 - e^{-\lambda t})}{\lambda t + \lambda \times MTTR \times (1 - e^{-\lambda t})} \qquad (4.4)$$

Variable Failure Rate

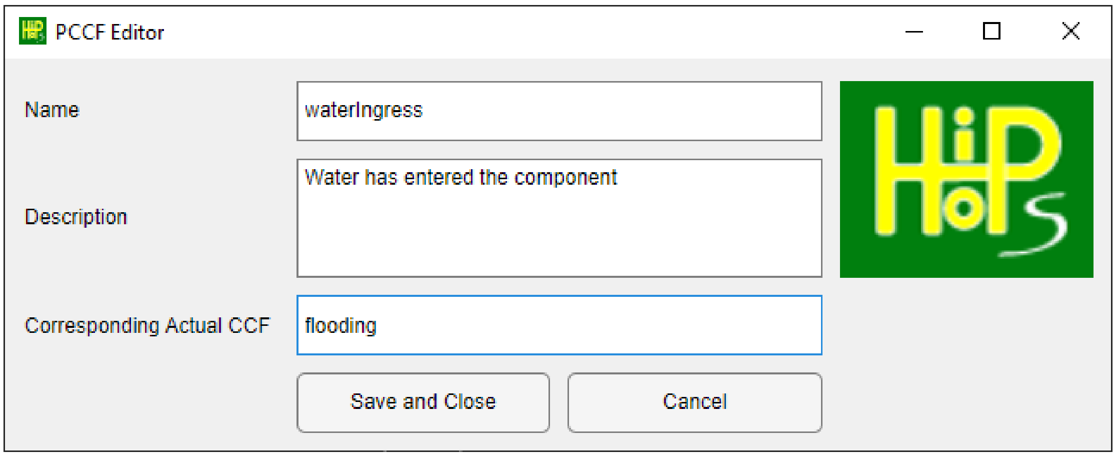
Parameters:

Slope parameter 1
Scale parameter 1
End interval 1
Scale parameter 2
End interval 2
Slope parameter 3
Scale parameter 3

This model allows the failure of the event to be described as a bath tub with Weibull variable failure rates.

4.4. PCCF Editor

Changing to the Common Causes tab allows you to open the Potential Common Cause Failure (PCCF) Editor:



Name	waterIngress
Description	Water has entered the component
Corresponding Actual CCF	flooding

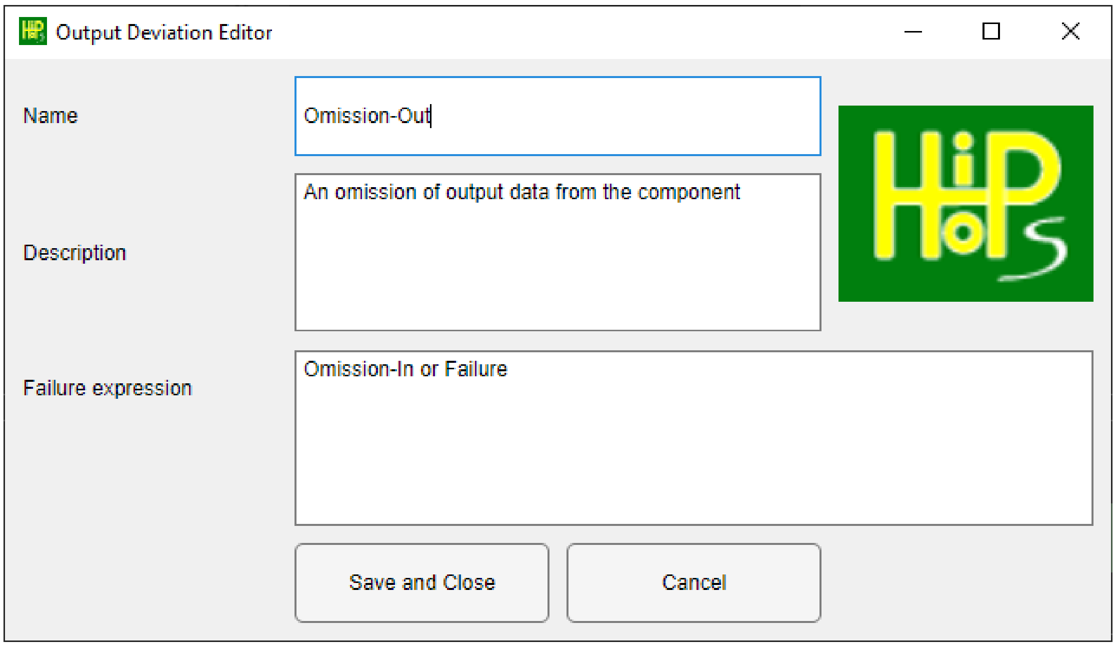
Figure 18: PCCF Editor

A PCCF acts like an indirect basic event. It is a placeholder with a name and a description, but its actual failure data comes from a corresponding Actual Common Cause Failure (ACCF) defined at the system level (see section XYZ).

As with basic events, a PCCF name can consist of letters, numbers, and underscores, but must begin with a letter. They must also be unique.

4.5. Output Deviation Editor

The third tab allows editing of output deviations via the Output Deviation Editor:


Output Deviation Editor

Name: Omission-Out

Description: An omission of output data from the component

Failure expression: Omission-In or Failure

Figure 19: Output Deviation Editor

As described previously, an output deviation relates the failures propagated from a component's output to a logical combination of propagated failures received at the inputs and any relevant internal failure modes of the component.

The name of an output deviation consists of two parts, separated by a dash (-). The first part is the *Failure Class* and should be used consistently across the whole model for all failures of this type. A commonly used set of failure classes are:

- **Omission** — A lack of output from the component port when expected
- **Commission** — Output when not expected (e.g. due to a short circuit)
- **Value** — A value failure, i.e., the output occurs when expected but the value is incorrect.

These may often be abbreviated (e.g. O, C, V) but again it is important to be consistent: if "Omission" is used in one place, "omission" in another, and "O" in a third, HiP-HOPS will treat them all as separate failure classes, thus preventing them from propagating correctly. To ensure propagation, the failure class of an output deviation must match the corresponding failure class of the input deviation referred to in the next component.

Other failure classes can be defined as necessary. For example, Value may be further distinguished into Timing (or Early/Late) failures and direction (e.g. High/Low). In other cases it might be convenient to refer to specific parameters, e.g. HighPressure, LowVoltage etc. Again, you are welcome to define whichever failure classes you please, but for them to propagate from one component to the next, they must be used consistently.

The second part of the name is the name of the port, which can be observed from the Simulink window.

The description field is free for the user to record important information about the output deviation but has no effect on the HiP-HOPS analysis.

The final field most certainly does, however. The Failure Expression is a Boolean logical expression that describes the causes of the output deviation. Operands can include input deviations (which follow the same naming convention as output deviations, i.e., `FailureClass-PortName`), basic events, and PCCFs. Note that for the expression to be valid, any basic events or PCCFs must be defined as described in the preceding sections.

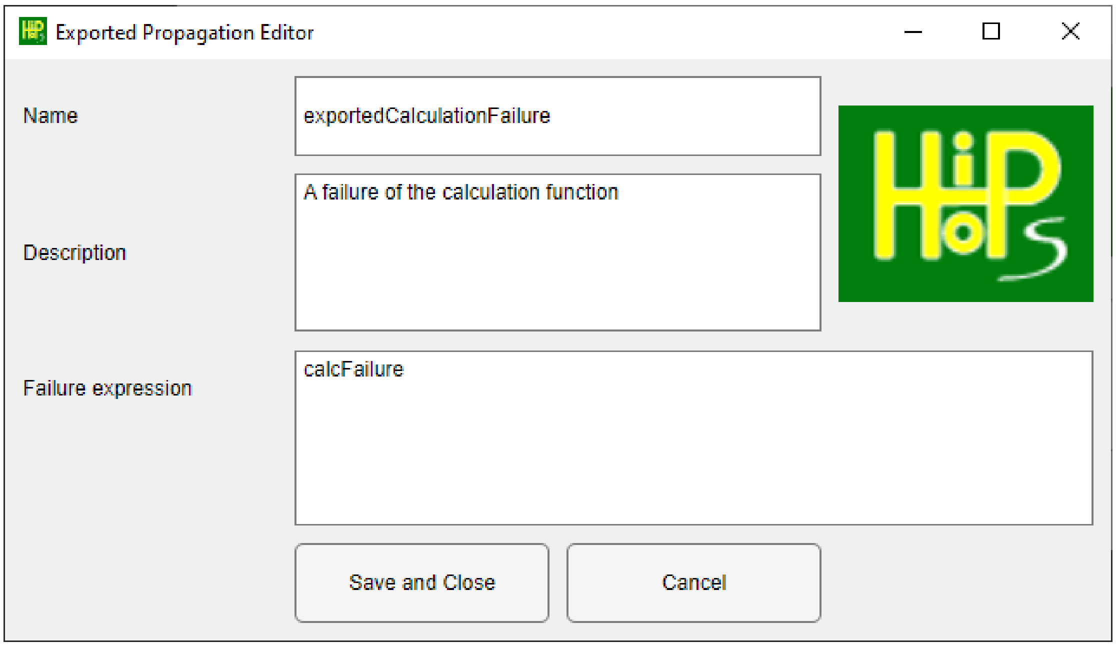
Operators include Boolean operators AND and OR (or * and + as shorthand) and round brackets/parentheses to define operator precedence (AND has higher precedence than OR). NOT (or ~ or !) is also a possible operator, but it is an unary operator (it takes only one parameter and binds to the operand to its immediate right). Note however that the inclusion of NOT will result in a non-coherent fault tree and incurs a performance penalty during analysis (see section 6.4 for more).

The following are all valid expressions:

- `failure`
- `failure OR omission-in1`
- `(omission-in1 OR omission-in2) AND basicEvent OR pccf1`
- `(omission-in1 or omission-in2) and basicEvent or pccf1`
- `(omission-in1 + omission-in2) * basicEvent + pccf1`
- `NOT omission-in1 AND failure`

4.6. Exported Propagation Editor

The fourth and final tab allows editing of exported propagations:


Exported Propagation Editor

Name: exportedCalculationFailure

Description: A failure of the calculation function

Failure expression: calcFailure

Figure 20: Exported Propagation Editor

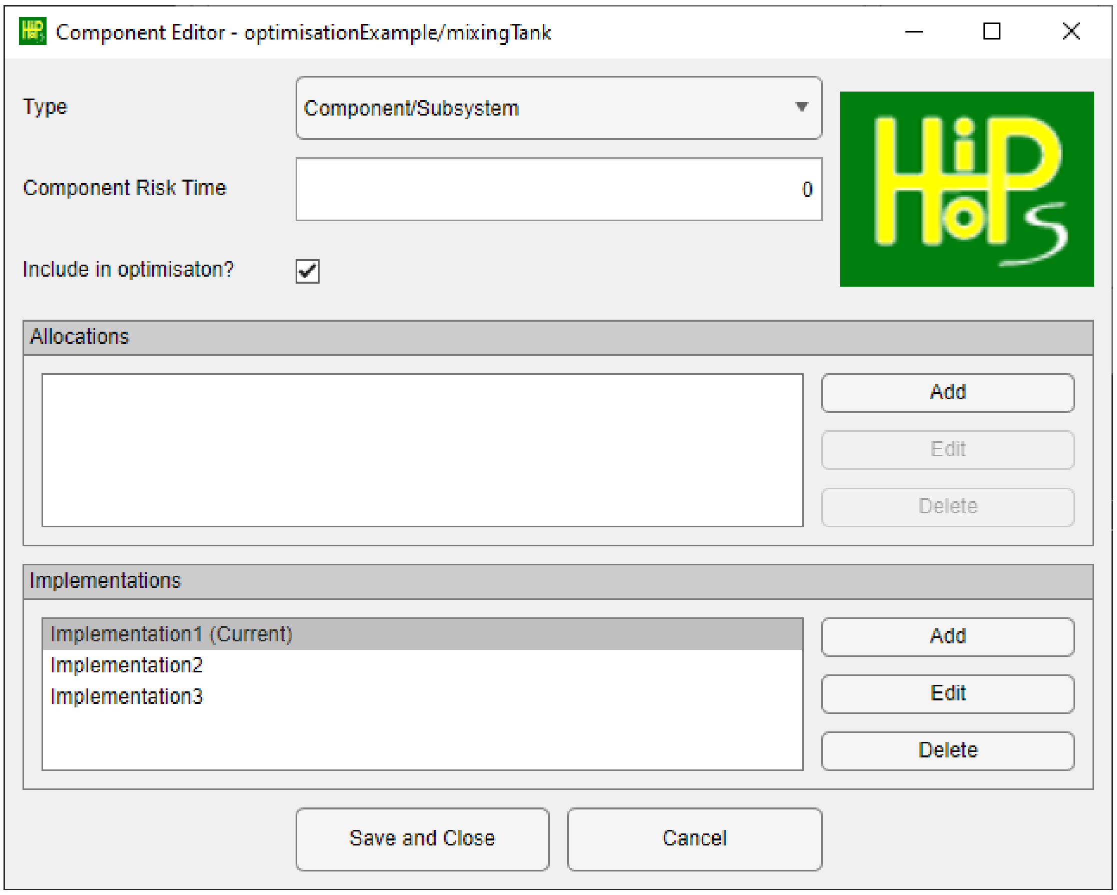
An exported propagation is a type of output deviation that propagates to a component in a different perspective rather than via an output port. For this propagation to be meaningful, the current component must be *allocated* to another (see section XYZ). The target component can then receive propagations exported from this component.

The interface is very much like the Output Deviation Editor and works in much the same way except for the name. An exported propagation is named much like a basic event, i.e., a combination of letters, numbers, and underscores, though it must begin with a letter. Notably, it must also be unique not just within the current component but across all components allocated to the target.

The description field and failure expression are the same as for output deviations.

4.7. Component Editor

Clicking on the "Edit component implementation data" button in the Launcher opens the Component Editor:


Component Editor - optimisationExample/mixingTank

Type: Component/Subsystem

Component Risk Time: 0

Include in optimisaton? ✓

Allocations

Implementations

Implementation1 (Current)
Implementation2
Implementation3

Figure 21: Component Editor

The above is the default form of the Component Editor and is primarily used to add or edit implementation information. Implementations are used to add variability during optimisation; the idea is that each component may be implemented in a number of different ways (e.g. by components from different manufacturers), each of which fulfil the same purpose while having differing failure characteristics.

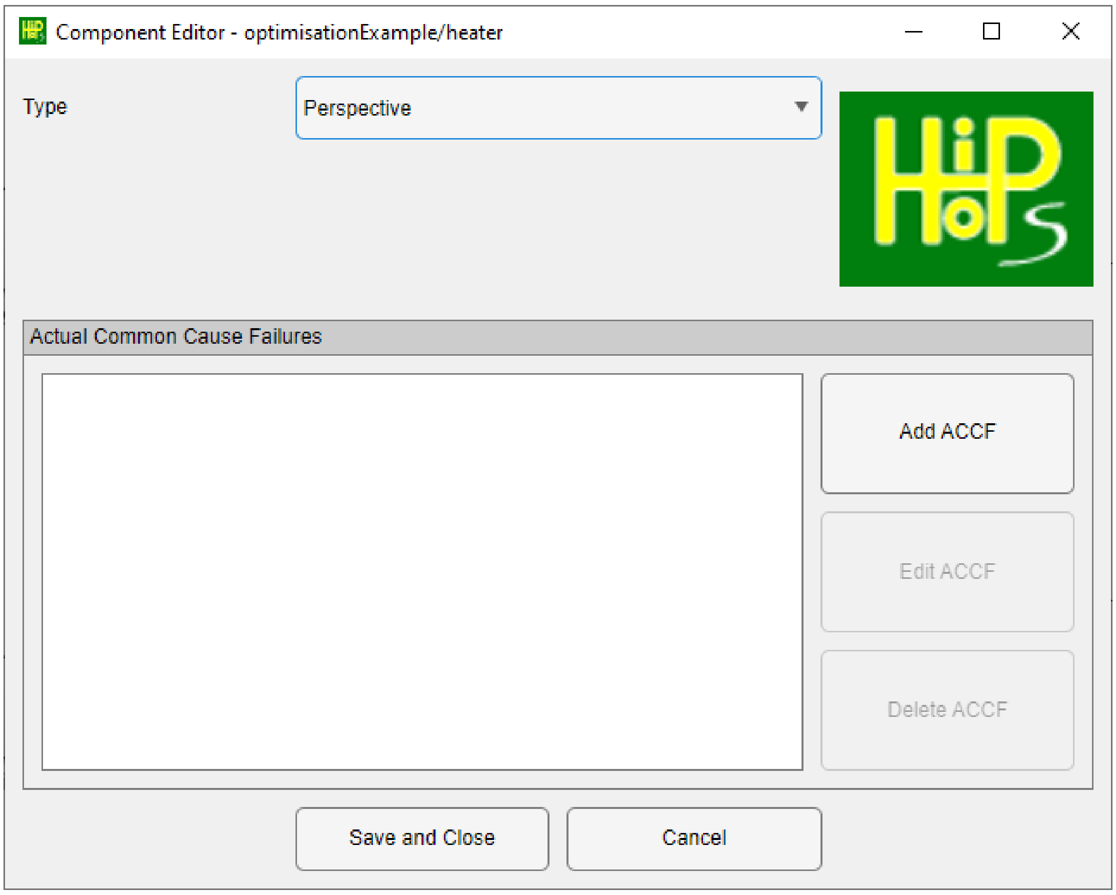
The list of implementations defined for a component are shown in the list at the bottom, along with buttons to Add, Edit, or Delete them. The current implementation is indicated by the "Current" suffix in brackets. Note that there is always at least one implementation added by default. The 'current' implementation is the one used by HiP-HOPS to perform its analysis, though optimisation will make use of all available implementations.

You can optionally choose to exclude a component from optimisation using the check box, which can be useful for debugging purposes in that it temporarily reduces the size of the optimisation space without deleting any information.

A component may also define its own risk time (i.e., the time during which the component is at risk/may experience failures), which overrides the global risk time defined at the model-level. This may be used when the component has a different risk time to the rest of the system, e.g. because it is dormant part of the time or because it only runs intermittently. Basic events of this component will then use this risk time value for quantitative calculations instead. If left blank, the global value is used instead.

Additionally, the Component Editor supports multi-perspective editing and common cause failures (CCFs). Top-level components can be turned into perspectives using the 'Type' drop-down menu, which results in a simpler version of the editor, as shown below. Components can also be allocated to a target component in another perspective using the allocations list, with associated buttons. The current allocation is indicated as 'Current' in brackets, like the current implementation.

Note that only top-level components can become perspectives. If there at least one top-level component is a perspective, all top-level components must be set as perspectives.


Component Editor - optimisationExample/heater

Type: Perspective

Figure 22: Component Editor for perspectives

In this version of the interface, the only relevant information is the list of Actual Common Cause Failures (ACCFs) defined. Buttons are available to add/edit/delete them as appropriate. The ACCF editor (opened by adding or editing an ACCF) is the same as the Basic Event Editor, described in section 4.3.

4.8. Allocation Editor

Adding or editing an allocation opens the Allocation Editor:

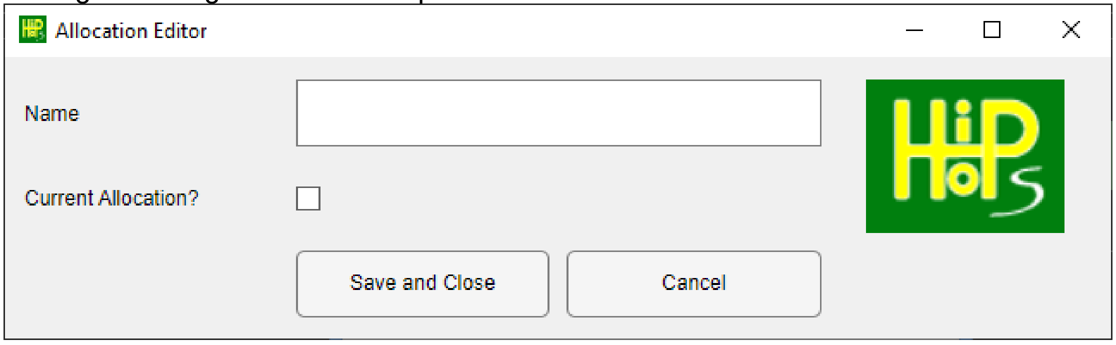


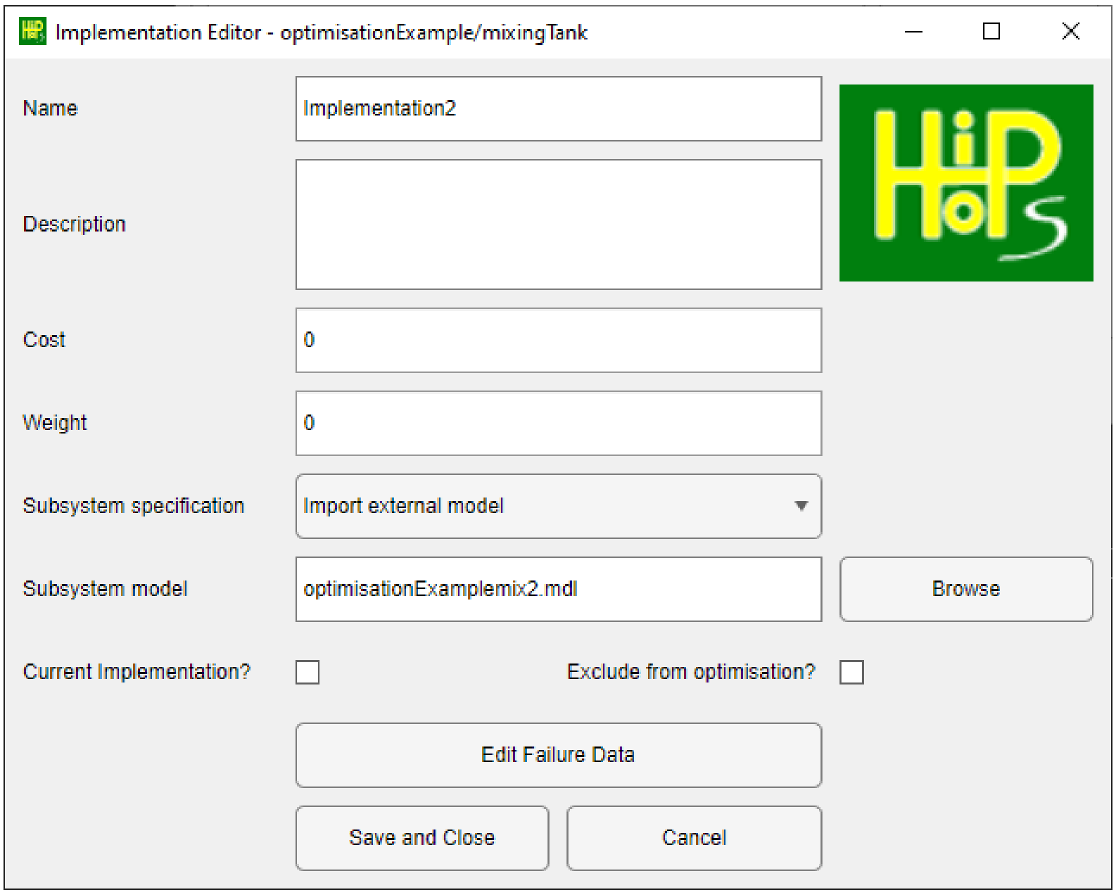
Figure 23: Allocation Editor

There are only two fields here. The first is the **fully qualified name** of the target component that this component should be allocated to. A fully qualified name is of the format *perspective::subsystem.component* and must include the perspective in this case. If multiple subsystems are involved, these are included as well, e.g. *subsystem.subsubsystem.component*.

The second field is the check box to indicate which allocation is the current (default) allocation. Optimisation will use all available allocations but standard analysis will use only the current allocation.

4.9. Implementation Editor

Adding or editing an implementation opens the Implementation Editor:


Implementation Editor - optimisationExample/mixingTank

Name	Implementation2
Description	
Cost	0
Weight	0
Subsystem specification	Import external model
Subsystem model	optimisationExamplemix2.mdl
Current Implementation?	☐
Exclude from optimisation?	☐

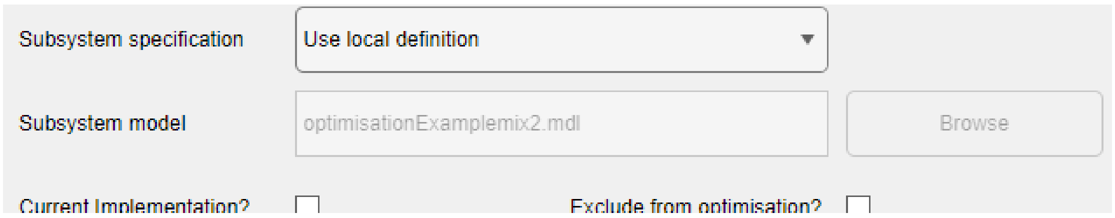
Figure 24: Implementation Editor

The name should follow the standard naming conventions, i.e., it can contain letters, numbers, and underscores, but should start with a letter and should be unique within its context — the component in this case.

The description is primarily for user reference and plays no part in analysis or optimisation.

Cost and weight allow parameterisation of this implementation for the purposes of optimisation and can be objectives of the optimisation process (e.g. to minimise cost and/or weight). Note that units are not specified, thus care must be taken to ensure that all costs/weights use consistent units across the model.

The subsystem specification field is more complex. Two options are available here. The first option is "Use local definition", which means that *all* implementations use the subsystem currently defined in the Simulink model. In this case, the Subsystem model field and 'Browse' button are disabled:



The second option is to define a separate subsystem for each implementation. Since Simulink does not support this directly, we instead achieve this by having separate model files for each implementation. The Subsystem model field allows you to specify this subsystem model; clicking Browse will open up a file dialog where you can select the right file.

Note that the path is stored relative to the current model path. Care should also be taken to ensure that the external model has the inputs and outputs to match those of the current component.

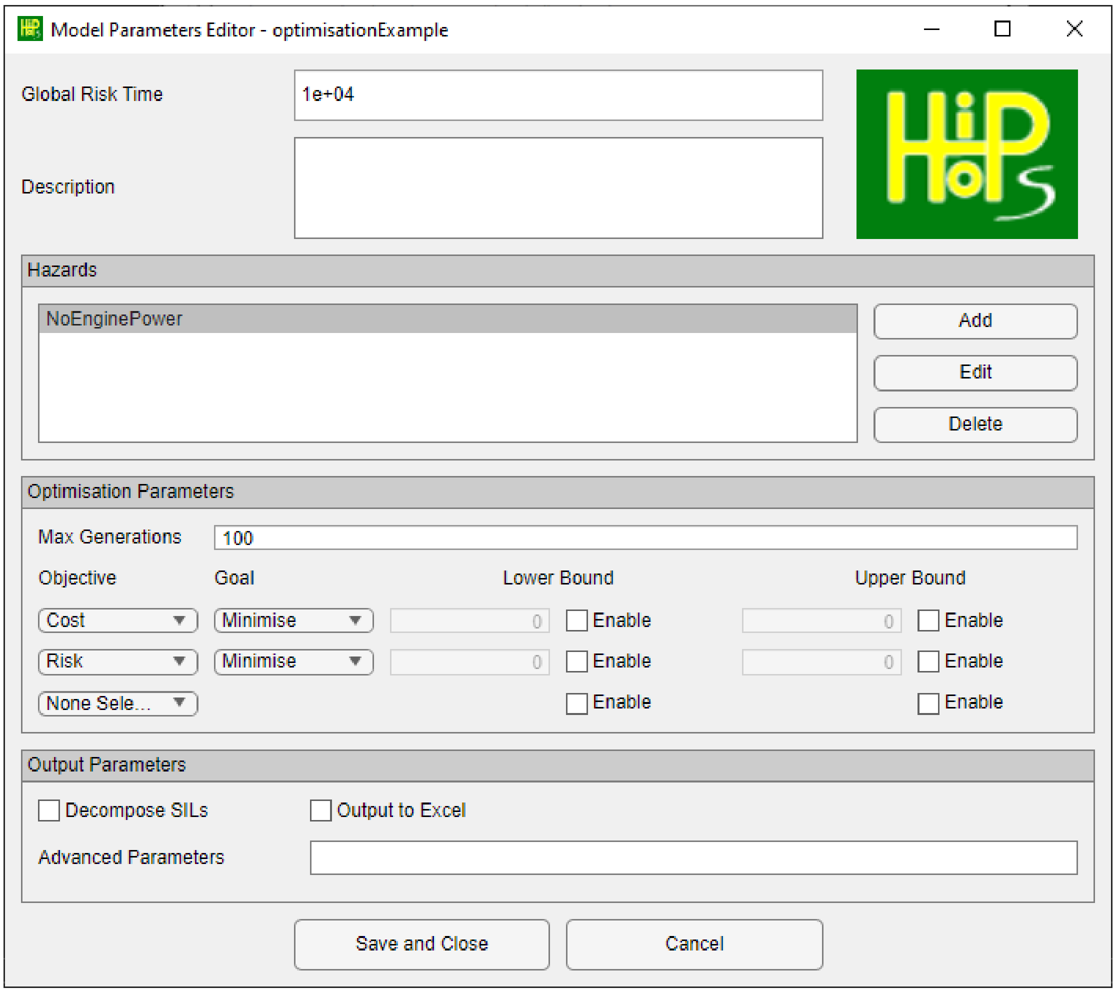
Any implementation can be set as 'current', meaning it will be the default implementation used in a standard analysis. Only one implementation can be set as current at any time.

Implementations can also be temporarily excluded from optimisation by checking the appropriate box. This can be useful for debugging purposes, since it excludes an implementation without deleting it.

Additionally, every implementation can have its own specific failure data. The "Edit Failure Data" button opens the Failure Data editor, allowing you to specify the information for the implementation being edited.

4.10. Model Parameters Editor

The third button on the Launcher, "Edit model failure data", opens the HiP-HOPS Model Parameters Editor, shown below.


Model Parameters Editor - optimisationExample

Global Risk Time: 1e+04

Description:

Hazards

NoEnginePower

Optimisation Parameters

Max Generations: 100

Objective	Goal	Lower Bound		Upper Bound	
Cost	Minimise	0	Enable	0	Enable
Risk	Minimise	0	Enable	0	Enable
None Sele...			Enable		Enable

Output Parameters

Decompose SILs
Output to Excel

Advanced Parameters:

Figure 25: Model Parameters Editor

This window allows editing of all the model-level parameters for the Simulink model.

At the top is the global risk time value, which is used in qualitative calculations to estimate probability and unavailability. It can be thought of in most cases as the expected operating lifetime of the system. Note however that no units are provided, so care must be taken to be consistent. For example, if the risk time is in hours, then the failure rates for basic events must be specified in terms of failures/hour; if the risk time is in years, then the failure rates would be failures/year instead.

Under this is the description field, which allows the user to input generic information about the overall system or model.

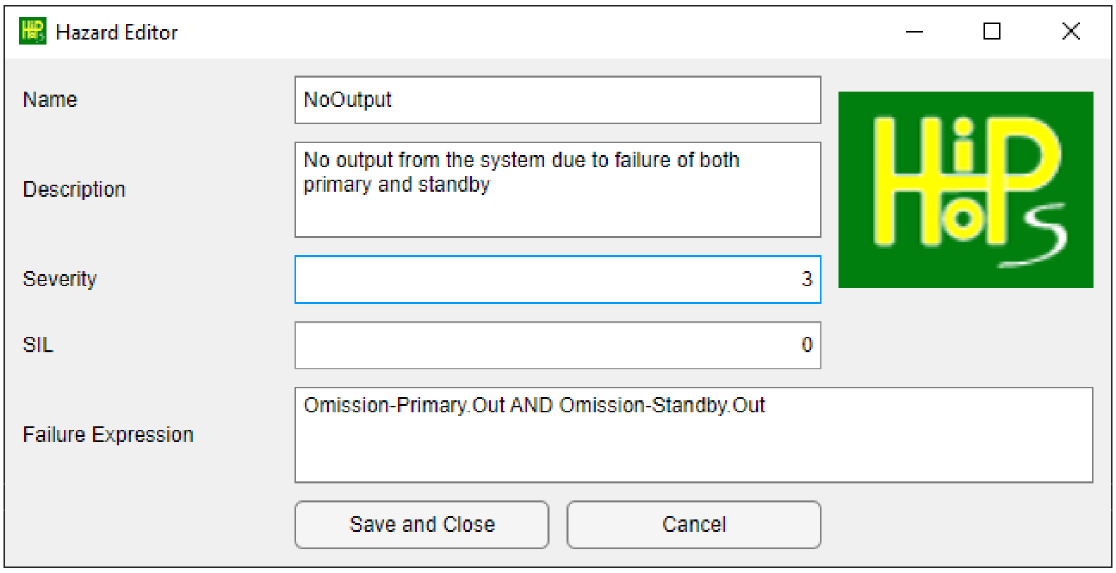
Next is the hazards list. While hazards serve as the starting point for a HiP-HOPS analysis, and can be thought of here as system-level failures, they may themselves be the output of a separate hazard and risk assessment process. Buttons are available to add, edit, or remove hazards.

Beneath the hazards are the optimisation parameters. These allows you to specify the maximum number of generations to run the optimisation for (generally, more generations yields better results but takes longer), as well as the objectives. Risk, cost, and weight are the available objective types. For each objective, the goal (minimise or maximise) and the lower & upper bounds can be set.

Finally, at the bottom are the output parameters that are passed to the HiP-HOPS executable. Ticking the "Decompose SILs" box will cause HiP-HOPS to attempt to decompose safety requirements. The "Output to Excel" box will cause HiP-HOPS to also output an Excel spreadsheet. The "Advanced Parameters" field allows you to enter more obscure parameters, which are described in section 7.

4.11. Hazard Editor

The final window is the Hazard Editor, which is opened when you add or edit a hazard.


Hazard Editor

Name	NoOutput
Description	No output from the system due to failure of both primary and standby
Severity	3
SIL	0
Failure Expression	Omission-Primary.Out AND Omission-Standby.Out

Figure 26: Hazard Editor

The first field is the name, which follows the usual naming rules. This name becomes the name of the fault tree that HiP-HOPS generates for each hazard and also serves as the 'effect' in the resultant FMEA.

The severity field is a custom numeric field where you can specify the severity for each hazard according to your own scale.

The SIL field is where a safety integrity level (or industrial equivalent, e.g. ASIL or DAL) can be set. This is used during SIL decomposition, as HiP-HOPS will attempt to distribute this value across all component basic events that cause the hazard.

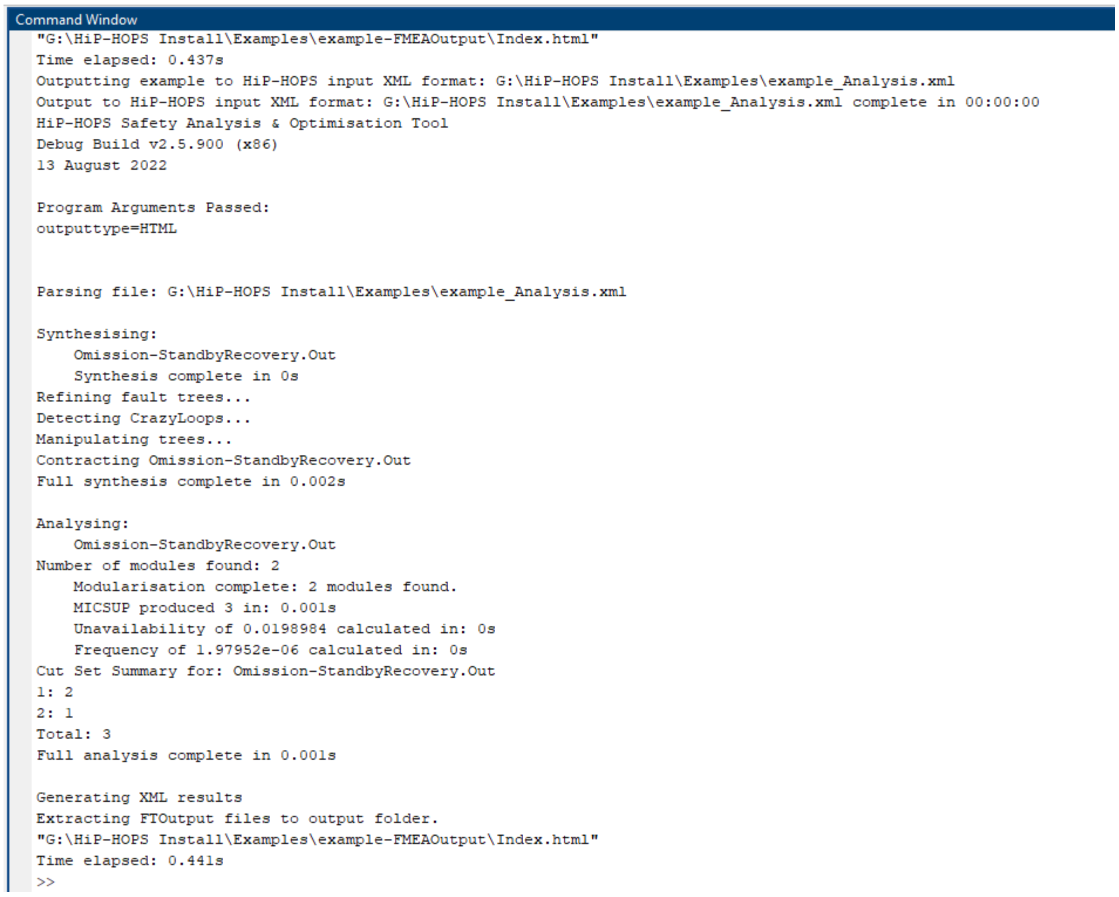
The final field is the failure expression field, which contains a logical expression that specifies the cause of the current hazard. It is similar to the logical expression for output deviations except that the only permissible operands are output deviations and these must be referred to using fully qualified names. Operators are the usual Boolean operators (AND, OR, NOT) and brackets/parentheses may be added to override operator precedence.

4.12. Running HiP-HOPS

Pressing the "Analyse model" or "Optimise model" buttons on the Launcher will pass the model to the HiP-HOPS engine to begin analysis or optimisation as appropriate.

Note that you should save the model before doing this or any recent changes may not be recognised by HiP-HOPS.

As HiP-HOPS runs, it will report status information to the Matlab command window:



```
"G:\HiP-HOPS Install\Examples\example-FMEAOutput\Index.html"
Time elapsed: 0.437s
Outputting example to HiP-HOPS input XML format: G:\HiP-HOPS Install\Examples\example_Analysis.xml
Output to HiP-HOPS input XML format: G:\HiP-HOPS Install\Examples\example_Analysis.xml complete in 00:00:00
HiP-HOPS Safety Analysis & Optimisation Tool
Debug Build v2.5.900 (x86)
13 August 2022

Program Arguments Passed:
outputtype=HTML


Parsing file: G:\HiP-HOPS Install\Examples\example_Analysis.xml

Synthesising:
    Omission-StandbyRecovery.Out
    Synthesis complete in 0s
Refining fault trees...
Detecting CrazyLoops...
Manipulating trees...
Contracting Omission-StandbyRecovery.Out
Full synthesis complete in 0.002s

Analysing:
    Omission-StandbyRecovery.Out
Number of modules found: 2
    Modularisation complete: 2 modules found.
    MICSUP produced 3 in: 0.001s
    Unavailability of 0.0198984 calculated in: 0s
    Frequency of 1.97952e-06 calculated in: 0s
Cut Set Summary for: Omission-StandbyRecovery.Out
1: 2
2: 1
Total: 3
Full analysis complete in 0.001s

Generating XML results
Extracting FTOutput files to output folder.
"G:\HiP-HOPS Install\Examples\example-FMEAOutput\Index.html"
Time elapsed: 0.441s
>>
```

Figure 27: Example output to Matlab command window

Any errors that occur will be reported here.

Depending on the options set in the Model Parameters, the analysis output should open automatically in your default web browser. Other output types (e.g. Excel spreadsheets, optimisation output) must be opened manually by navigating to the output directory — which is typically the same location as the Simulink model, but which can also be read from the output in the Command Window.

The HiP-HOPS output will be described in the following section.

5. HiP-HOPS Output

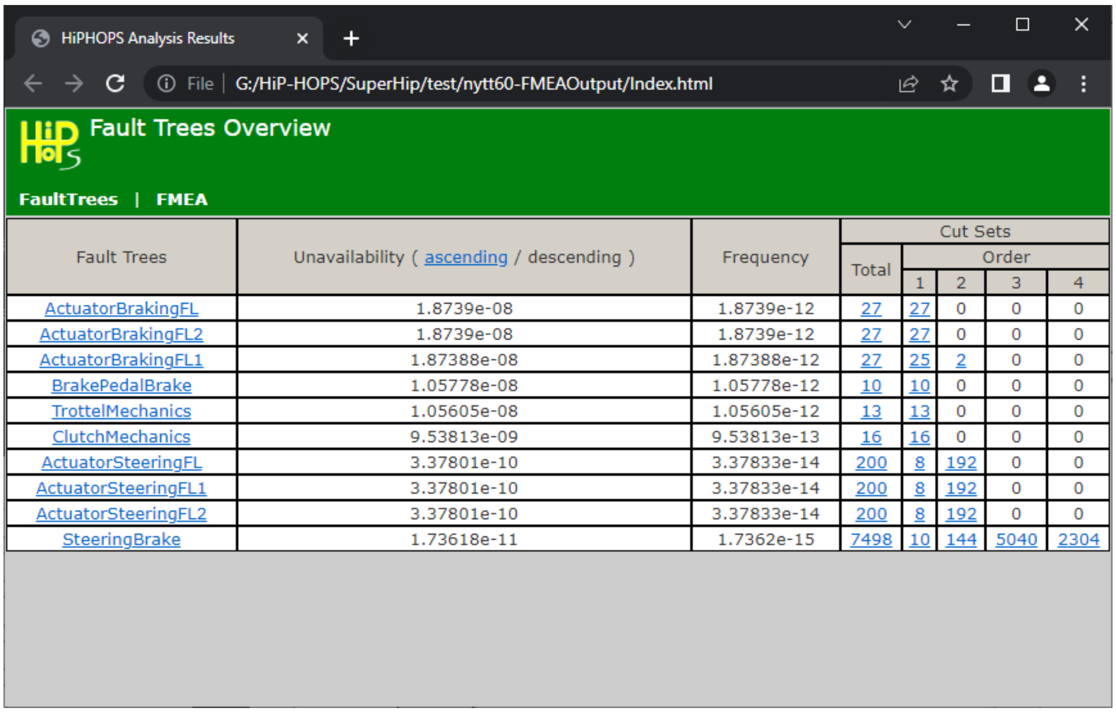
HiP-HOPS produces a variety of output depending on the parameters set (see section 7). By default, HTML & JavaScript files are produced which are viewable in a web browser. An Excel spreadsheet may also be generated. Standalone XML files can also be produced.

Standard HiP-HOPS output is generated in a sub-directory called <*modelName*>-FMEAOutput, in the same directory as the Simulink model they were generated from.

5.1. Web Browser Analysis Output

The default output for a HiP-HOPS analysis is a collection of XML and HTML files viewable in a browser. There will be an index.html file in the root, which can be opened in the browser. Data is in the 'model' subdirectory in the form of JavaScript files, while the 'FTOutput' subdirectory contains the various standard files HiP-HOPS uses to render the result in the browser.

The main index page looks like this:



HiPHOPS Analysis Results

File | G:/HiP-HOPS/SuperHip/test/nytt60-FMEAOutput/Index.html

Fault Trees Overview

FaultTrees | FMEA

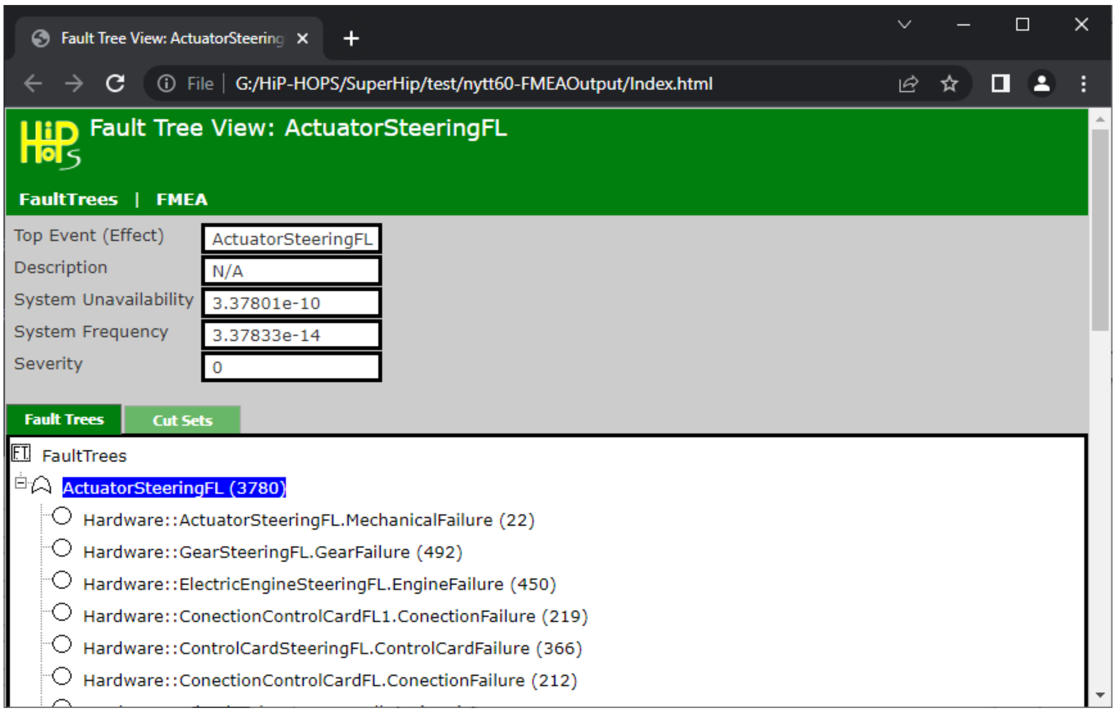
Fault Trees	Unavailability (ascending / descending)	Frequency	Cut Sets				
			Total	Order			
				1	2	3	4
ActuatorBrakingFL	1.8739e-08	1.8739e-12	27	27	0	0	0
ActuatorBrakingFL2	1.8739e-08	1.8739e-12	27	27	0	0	0
ActuatorBrakingFL1	1.87388e-08	1.87388e-12	27	25	2	0	0
BrakePedalBrake	1.05778e-08	1.05778e-12	10	10	0	0	0
TrottelMechanics	1.05605e-08	1.05605e-12	13	13	0	0	0
ClutchMechanics	9.53813e-09	9.53813e-13	16	16	0	0	0
ActuatorSteeringFL	3.37801e-10	3.37833e-14	200	8	192	0	0
ActuatorSteeringFL1	3.37801e-10	3.37833e-14	200	8	192	0	0
ActuatorSteeringFL2	3.37801e-10	3.37833e-14	200	8	192	0	0
SteeringBrake	1.73618e-11	1.7362e-15	7498	10	144	5040	2304

Figure 28: Analysis index page

This page shows an overview of all the fault trees that have been generated, their estimated unavailability, their failure frequency, and the cut set totals. The list can also be sorted in both ascending or descending order by clicking the relevant link in the unavailability column.

5.1.1. Fault Tree Output

Clicking on any of the fault tree names will open the fault tree view for that fault tree:



Fault Tree View: ActuatorSteering

G:/HiP-HOPS/SuperHip/test/nytt60-FMEAOutput/Index.html

Fault Tree View: ActuatorSteeringFL

FaultTrees | FMEA

Top Event (Effect)	ActuatorSteeringFL
Description	N/A
System Unavailability	3.37801e-10
System Frequency	3.37833e-14
Severity	0

Fault Trees | Cut Sets

```
FaultTrees
  ActuatorSteeringFL (3780)
    Hardware::ActuatorSteeringFL.MechanicalFailure (22)
    Hardware::GearSteeringFL.GearFailure (492)
    Hardware::ElectricEngineSteeringFL.EngineFailure (450)
    Hardware::ConectionControlCardFL1.ConectionFailure (219)
    Hardware::ControlCardSteeringFL.ControlCardFailure (366)
    Hardware::ConectionControlCardFL.ConectionFailure (212)
```

Figure 29: Fault Tree view

Here, the fault tree information is shown — its description, system unavailability & failure frequency, and the severity. Below are two tabs: one is a simple vertically-oriented representation of the fault tree, showing all of the gates and basic events, while the other lists the cut sets.

For the fault tree diagram, each node contains the fully qualified name of the node — its perspective, subsystem (if any), component, and either port or basic event. The number in brackets is the ID value assigned by HiP-HOPS and can be used to determine whether two nodes are identical or not; if they have the same ID, they are identical.

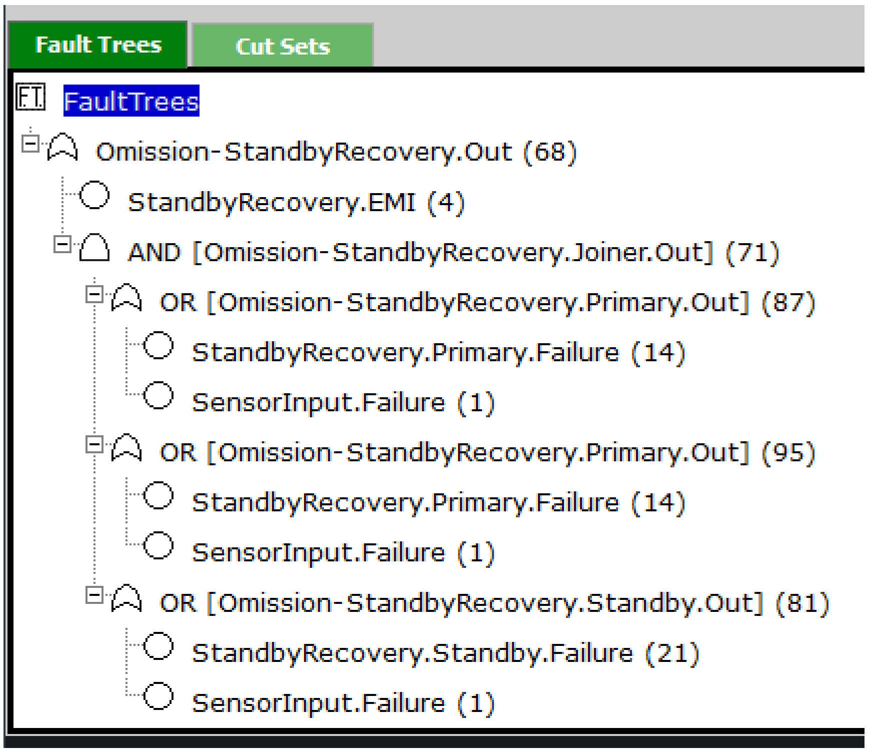
Each node also has a symbol:

AND gate
OR gate
○ Basic event / Failure Mode
Circle node (circular logic has been detected and broken at this point)
Input deviation
Output deviation
Proxy/Intermediate node
Normal event (a non-failure event)

△ Transfer node (references another location in the fault tree via ID)

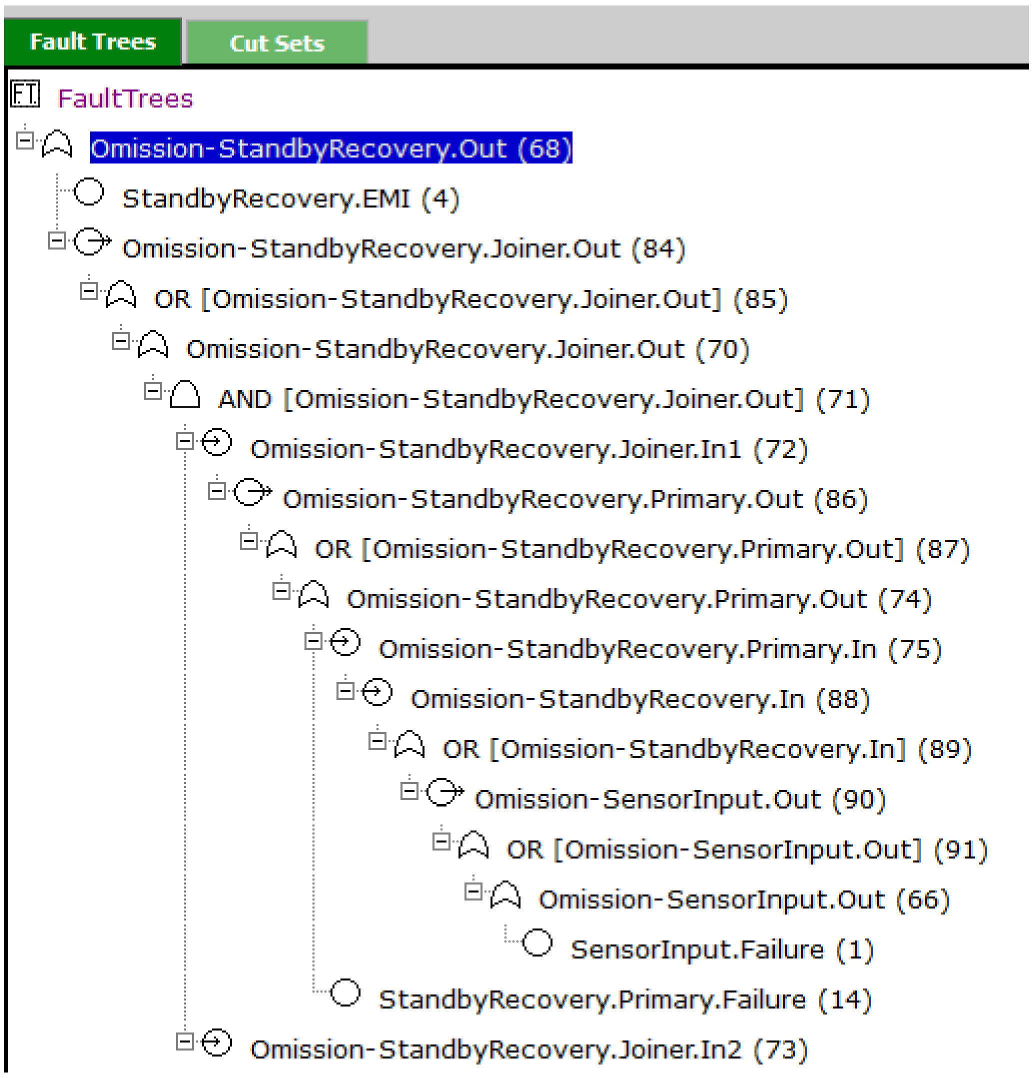
Some of these types of nodes may not be visible in all output modes. By default, HiP-HOPS will *contract* all fault trees by eliminating any redundant intermediate nodes (such as input/output deviations, proxy nodes, etc). This simplifies the fault tree and makes it much easier to read, but it also hides the full propagation information. Using the "-contract" argument tells HiP-HOPS to leave the full fault tree intact.

An example is shown below:



```
FaultTrees
  Omission-StandbyRecovery.Out (68)
    StandbyRecovery.EMI (4)
    AND [Omission-StandbyRecovery.Joiner.Out] (71)
      OR [Omission-StandbyRecovery.Primary.Out] (87)
        StandbyRecovery.Primary.Failure (14)
        SensorInput.Failure (1)
      OR [Omission-StandbyRecovery.Primary.Out] (95)
        StandbyRecovery.Primary.Failure (14)
        SensorInput.Failure (1)
      OR [Omission-StandbyRecovery.Standby.Out] (81)
        StandbyRecovery.Standby.Failure (21)
        SensorInput.Failure (1)
```

Figure 30: A contracted fault tree

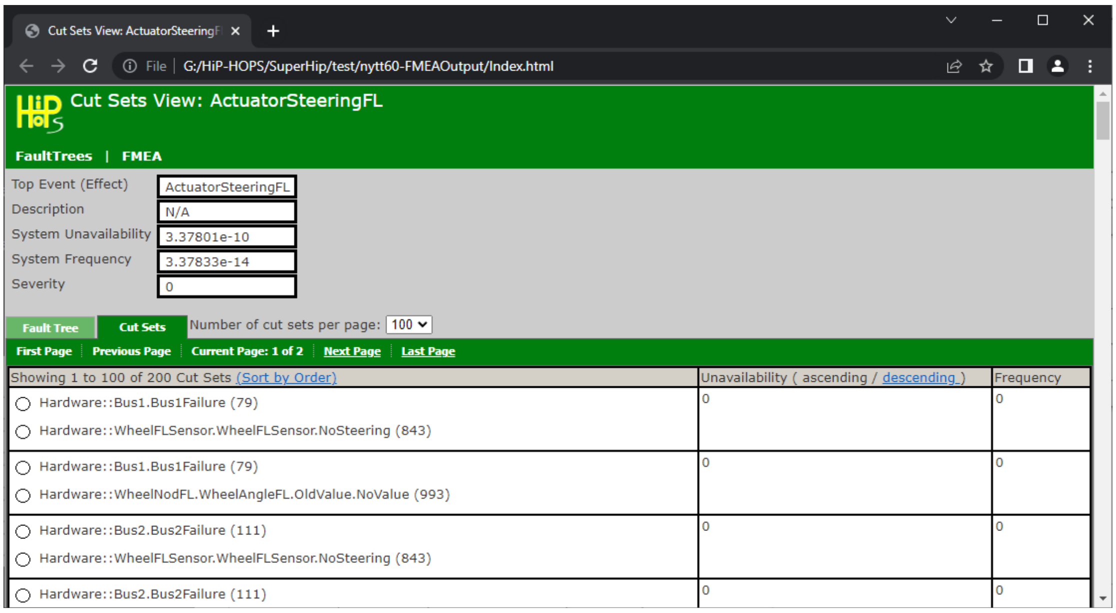


```
FaultTrees
Omission-StandbyRecovery.Out (68)
  StandbyRecovery.EMI (4)
  Omission-StandbyRecovery.Joiner.Out (84)
    OR [Omission-StandbyRecovery.Joiner.Out] (85)
      Omission-StandbyRecovery.Joiner.Out (70)
        AND [Omission-StandbyRecovery.Joiner.Out] (71)
          Omission-StandbyRecovery.Joiner.In1 (72)
            Omission-StandbyRecovery.Primary.Out (86)
              OR [Omission-StandbyRecovery.Primary.Out] (87)
                Omission-StandbyRecovery.Primary.Out (74)
                  Omission-StandbyRecovery.Primary.In (75)
                    Omission-StandbyRecovery.In (88)
                      OR [Omission-StandbyRecovery.In] (89)
                        Omission-SensorInput.Out (90)
                          OR [Omission-SensorInput.Out] (91)
                            Omission-SensorInput.Out (66)
                              SensorInput.Failure (1)
                    StandbyRecovery.Primary.Failure (14)
          Omission-StandbyRecovery.Joiner.In2 (73)
```

Figure 31: (Part of) the uncontracted fault tree

Both fault trees are logically equivalent and contain the same basic events (and the same IDs), but in the first tree the intermediate nodes have been stripped out for clarity. The second tree, however, makes it easier to see the propagation of failure through the model; for example, we can see how the SensorInput failure propagates from SensorInput.Out to StandbyRecovery.In and then to StandbyRecovery. Primary.In and so forth.

The second tab displays the minimal cut sets in tabular form:



Cut Sets View: ActuatorSteeringFL

G:/HiP-HOPS/SuperHip/test/nytt60-FMEAOutput/Index.html

Cut Sets View: ActuatorSteeringFL

FaultTrees | FMEA

Top Event (Effect)	ActuatorSteeringFL
Description	N/A
System Unavailability	3.37801e-10
System Frequency	3.37833e-14
Severity	0

Fault Tree | Cut Sets | Number of cut sets per page: 100

First Page | Previous Page | Current Page: 1 of 2 | Next Page | Last Page

Showing 1 to 100 of 200 Cut Sets (Sort by Order)	Unavailability (ascending / descending)	Frequency
○ Hardware::Bus1.Bus1Failure (79) ○ Hardware::WheelFLSensor.WheelFLSensor.NoSteering (843)	0	0
○ Hardware::Bus1.Bus1Failure (79) ○ Hardware::WheelNodFL.WheelAngleFL.OldValue.NoValue (993)	0	0
○ Hardware::Bus2.Bus2Failure (111) ○ Hardware::WheelFLSensor.WheelFLSensor.NoSteering (843)	0	0
○ Hardware::Bus2.Bus2Failure (111)	0	0

Figure 32: Fault tree cut sets table

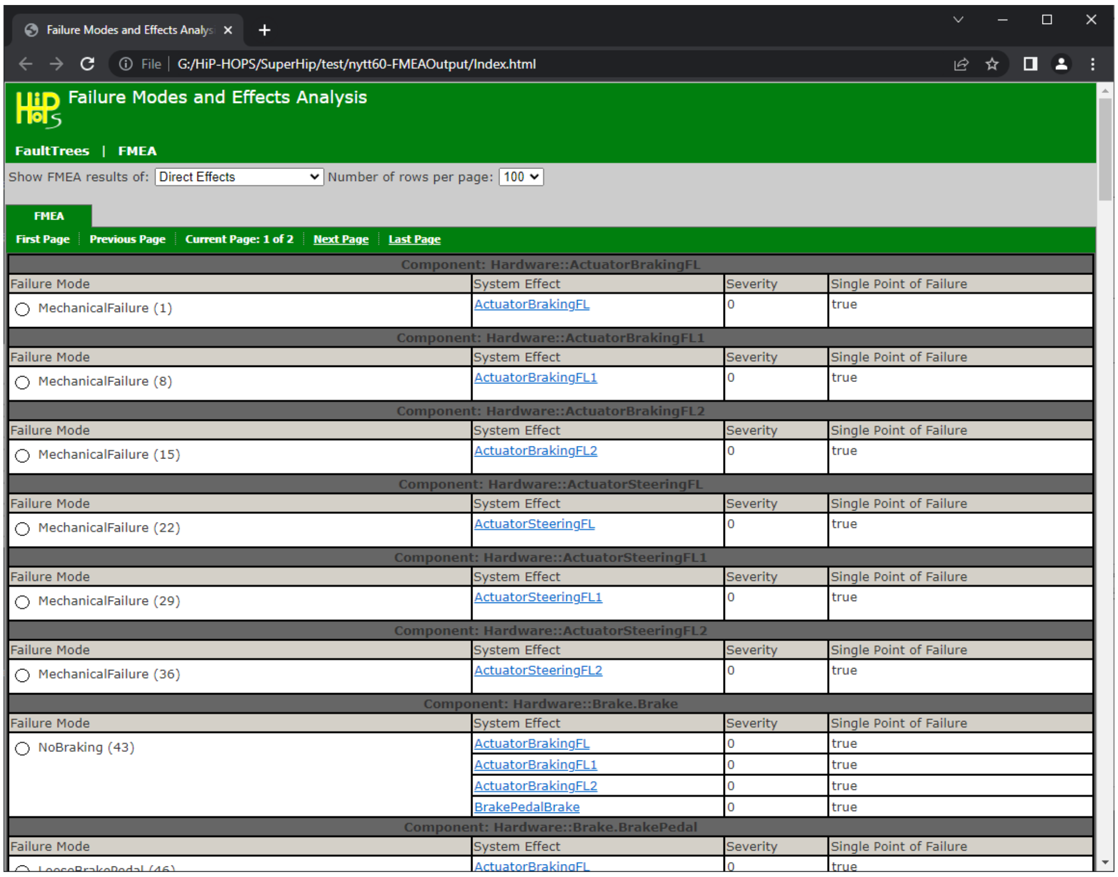
Each row of the table is a cut set. The constituent basic events are listed in the left-most column. Unavailability and frequency is shown, if data is available for it to be calculated.

Cut sets can be sorted by unavailability in ascending (default) or descending order, or by cut set order (i.e., how many basic events per cut set), by clicking the links in the table header. HiP-HOPS also paginates the cut sets for faster loading; by default, it displays 100 results per page, but this can be changed or switched off using the drop-down menu.

Clicking on the cut set totals from the main page also opens this table directly, except filtered by order. Thus if you click the number of order 1 cut sets for a given fault tree, the table will only display the order 1 cut sets, and so on.

5.1.2. <u>FMEA Output</u>

The other form of output is the FMEA, accessible from any page by clicking 'FMEA' at the top. The FMEA displays a table showing each effect (i.e., hazard) caused by each component failure mode, bunched by component:



Failure Modes and Effects Analys

File | G:/HiP-HOPS/SuperHip/test/nytt60-FMEAOutput/Index.html

Failure Modes and Effects Analysis

FaultTrees | FMEA

Show FMEA results of: Direct Effects Number of rows per page: 100

FMEA

First Page | Previous Page | Current Page: 1 of 2 | Next Page | Last Page

Failure Mode	System Effect	Severity	Single Point of Failure
Component: Hardware::ActuatorBrakingFL			
MechanicalFailure (1)	ActuatorBrakingFL	0	true
Component: Hardware::ActuatorBrakingFL1			
MechanicalFailure (8)	ActuatorBrakingFL1	0	true
Component: Hardware::ActuatorBrakingFL2			
MechanicalFailure (15)	ActuatorBrakingFL2	0	true
Component: Hardware::ActuatorSteeringFL			
MechanicalFailure (22)	ActuatorSteeringFL	0	true
Component: Hardware::ActuatorSteeringFL1			
MechanicalFailure (29)	ActuatorSteeringFL1	0	true
Component: Hardware::ActuatorSteeringFL2			
MechanicalFailure (36)	ActuatorSteeringFL2	0	true
Component: Hardware::Brake.Brake			
NoBraking (43)	ActuatorBrakingFL	0	true
	ActuatorBrakingFL1	0	true
	ActuatorBrakingFL2	0	true
	BrakePedalBrake	0	true
Component: Hardware::Brake.BrakePedal			
LooseBrakePedal (46)	ActuatorBrakingFL	0	true

Figure 33: FMEA view

Like the cut sets, this table is paginated for faster loading. Number of results per page is customisable via the drop-down menu.

From left to right, the columns show:

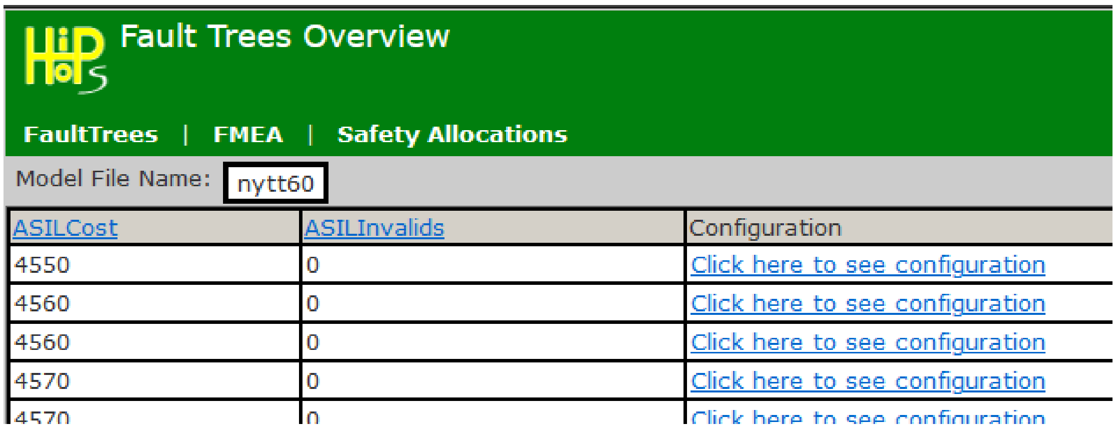
- The name and ID of the failure mode;
- The list of system effects (hazards) it causes, which can be clicked on to view the corresponding fault tree;
- The severity of the system effect;
- Whether or not the failure mode is a single point of failure.

The latter point is one of the unique features of HiP-HOPS, in that its FMEA shows not only single points of failure but also the effects of failure modes that must occur in conjunction. To view these "further effects", select either "Further Effects" or "Direct and Further Effects" from the first drop-down menu at the top. (By default, "Direct Effects" is selected, which only shows single points of failure; this is more in line with a typical FMEA and also reduces the number of results).

A further effect has "false" in the rightmost column, since it does not cause the system-level effect by itself. To view the other contributing events, you can click on the effect and it will take you to the cut sets view for that fault tree, filtered to show only the cut sets containing that failure mode.

5.1.3. SIL Decomposition

If SIL Decomposition mode was used (and SILs are available for decomposition), an additional link will appear at the top where you can view the safety allocations for the system. The table shows each identified allocation, displaying the total SIL cost (where 1 = cost 1, 2 = cost 10, 3 = cost 100, 4 = cost 1000), the number of invalid SILs (i.e., where it does not meet the requirements), and a link to view the system configuration.



Fault Trees Overview

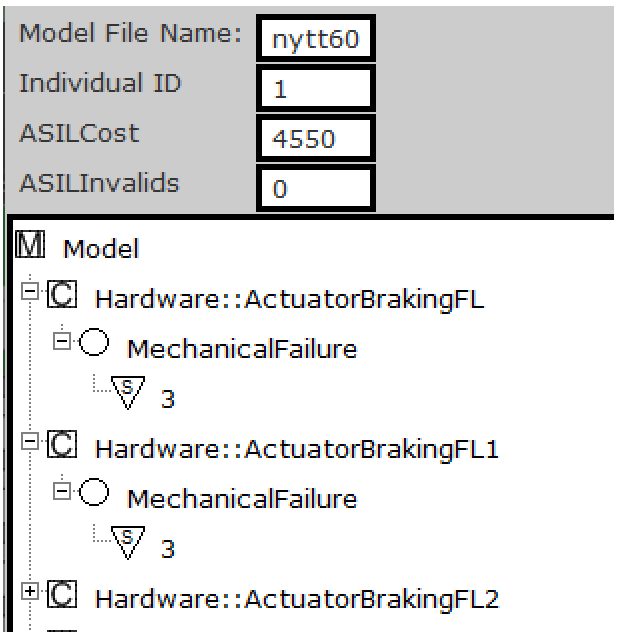
FaultTrees | FMEA | Safety Allocations

Model File Name: nytt60

ASILCost	ASILInvalids	Configuration
4550	0	Click here to see configuration
4560	0	Click here to see configuration
4560	0	Click here to see configuration
4570	0	Click here to see configuration
4570	0	Click here to see configuration

Figure 34: Safety Allocations view

Clicking one of the configuration links on the right opens a treeview for that configuration, showing which SIL is assigned to each basic event:



Model File Name:	nytt60
Individual ID	1
ASILCost	4550
ASILInvalids	0

```
Model
  Hardware::ActuatorBrakingFL
    MechanicalFailure
      3
  Hardware::ActuatorBrakingFL1
    MechanicalFailure
      3
  Hardware::ActuatorBrakingFL2
```

Figure 35: Configuration view

The tree shows the model as the root, then the components, then all the basic events of each component, and finally the SIL assigned to each basic event.

5.1.4. Warnings and Errors

Errors can arise for a number of reasons, but the most common is an incorrect name. Errors are reported in the Matlab command window. For example, an incorrect port name in an input deviation may result in an error like this:

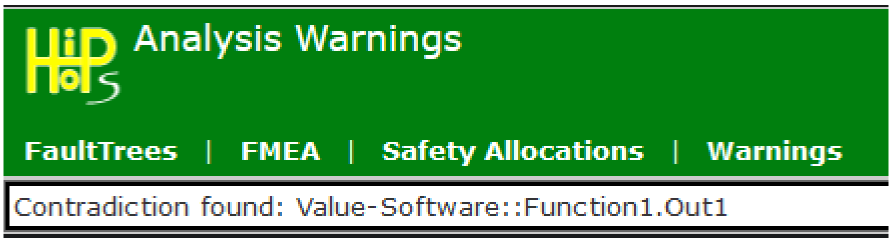
```
ERRORS: 1
ERROR:   Port   StandbyRecovery.Standby.missing   in   component
StandbyRecovery.Standby not found! [line number : 305]
```

The information provided should be enough to diagnose and fix the error. In this case, there is an output deviation in the Standby component that refers to a port called "missing" which does not exist.

The same can apply if an output deviation refers to a basic event that does not exist:

```
ERRORS: 1
ERROR:  Basic  event  NoSuch  is  referred  to  in  an  output
deviation    but    does    not    exist    in    component
StandbyRecovery.Standby [line number : 305]
```

It is also possible to have non-critical errors which do not prevent analysis but which do raise warnings. These are listed in both the command window and in the browser results view, where a new "Warnings" link will appear at the top.



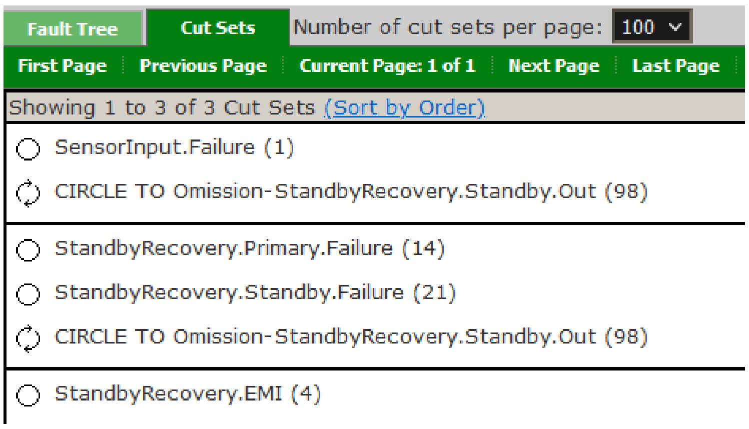
Contradiction found: Value-Software::Function1.Out1

Figure 36: Warnings in the output

The most common form of warning is a contradiction. These can be caused by danging input deviations (input deviations with no corresponding output deviation) or via circular propagation logic in the model.

5.1.5. Circular Logic

HiP-HOPS will detect circular logic and will in most cases be able to handle it appropriately by severing the loop. In this case, it will insert a "circle node" at the point where the loop was severed, effectively unrolling the loop to its largest extent. This appears in the results as a special type of node:



Showing 1 to 3 of 3 Cut Sets (Sort by Order)

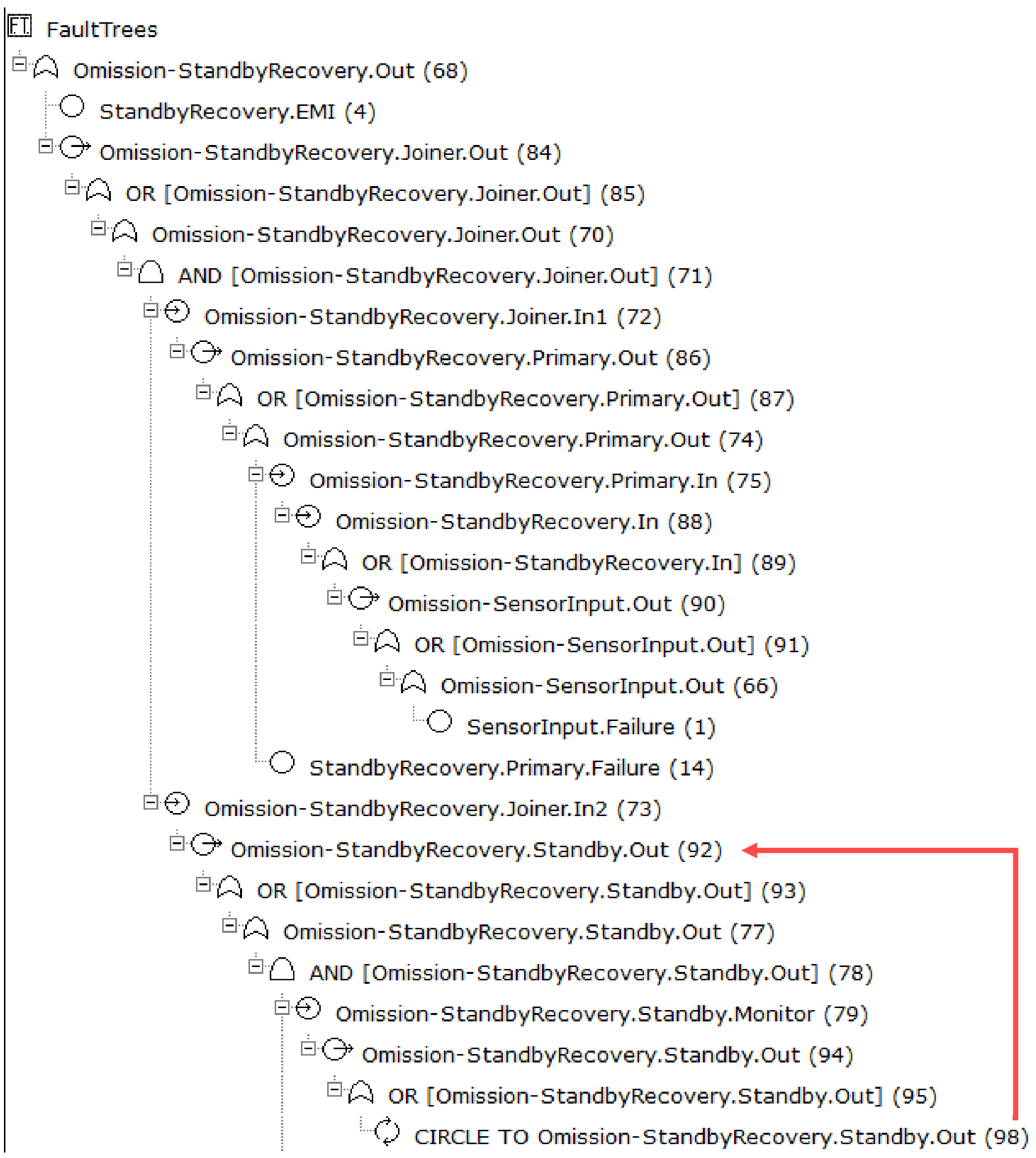
SensorInput.Failure (1)
CIRCLE TO Omission-StandbyRecovery.Standby.Out (98)

StandbyRecovery.Primary.Failure (14)
StandbyRecovery.Standby.Failure (21)
CIRCLE TO Omission-StandbyRecovery.Standby.Out (98)

StandbyRecovery.EMI (4)

Figure 37: Circular logic detected

Sometimes this circular logic is expected and harmless, but it should always be checked if it is detected. In this case we can see more information by viewing the tree in uncontracted mode:



```
FaultTrees
Omission-StandbyRecovery.Out (68)
  StandbyRecovery.EMI (4)
  Omission-StandbyRecovery.Joiner.Out (84)
    OR [Omission-StandbyRecovery.Joiner.Out] (85)
      Omission-StandbyRecovery.Joiner.Out (70)
        AND [Omission-StandbyRecovery.Joiner.Out] (71)
          Omission-StandbyRecovery.Joiner.In1 (72)
            Omission-StandbyRecovery.Primary.Out (86)
              OR [Omission-StandbyRecovery.Primary.Out] (87)
                Omission-StandbyRecovery.Primary.Out (74)
                  Omission-StandbyRecovery.Primary.In (75)
                    Omission-StandbyRecovery.In (88)
                      OR [Omission-StandbyRecovery.In] (89)
                        Omission-SensorInput.Out (90)
                          OR [Omission-SensorInput.Out] (91)
                            Omission-SensorInput.Out (66)
                              SensorInput.Failure (1)
                  StandbyRecovery.Primary.Failure (14)
          Omission-StandbyRecovery.Joiner.In2 (73)
            Omission-StandbyRecovery.Standby.Out (92)
              OR [Omission-StandbyRecovery.Standby.Out] (93)
                Omission-StandbyRecovery.Standby.Out (77)
                  AND [Omission-StandbyRecovery.Standby.Out] (78)
                    Omission-StandbyRecovery.Standby.Monitor (79)
                      Omission-StandbyRecovery.Standby.Out (94)
                        OR [Omission-StandbyRecovery.Standby.Out] (95)
                          CIRCLE TO Omission-StandbyRecovery.Standby.Out (98)
```

Figure 38: More circular logic (arrow added)

Here we can see that one of the causes of Omission-StandbyRecovery.Standby.Out is itself, via input at port Monitor. This is also apparent from the model:

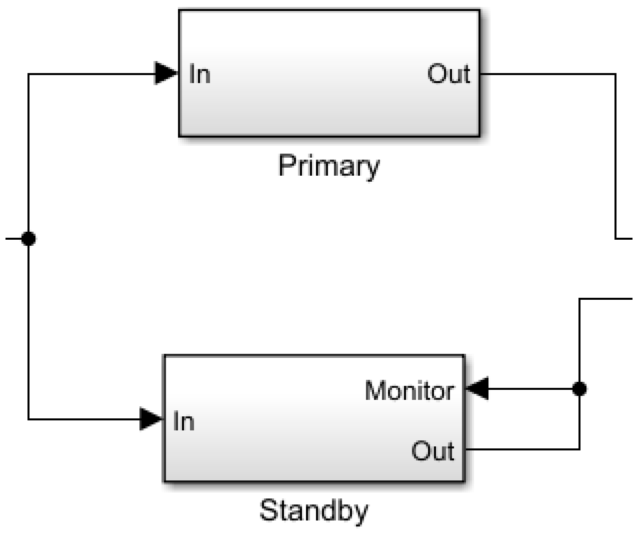


Figure 39: The circular model

Clearly this is an artificial example, but more complex situations can arise where the circular propagation goes through many components before looping around.

In some cases, the logic becomes too tangled for HiP-HOPS to continue — a so-called "Crazy Loop". This can arise when a circular loop has multiple entry points for the same fault tree. To illustrate the issue, consider the following fault trees.

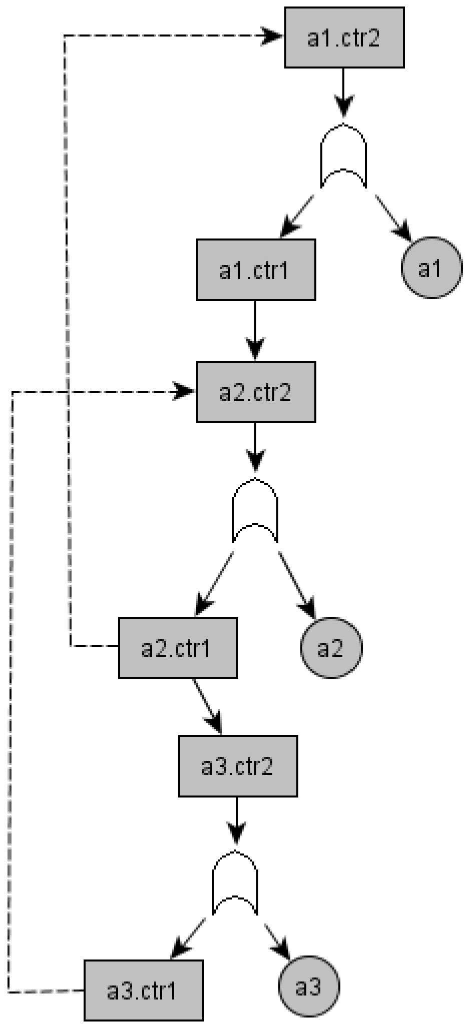
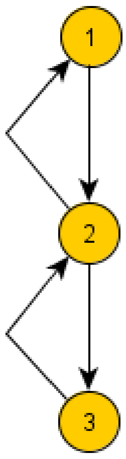
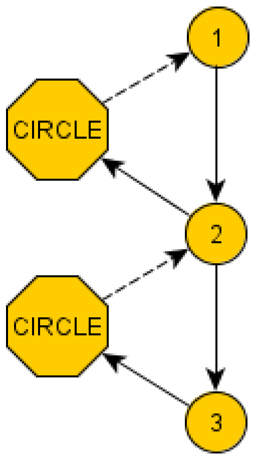


Figure 40: Complex Circular Loop 1

In this case, we have a fault tree with two loops, one from a3.ctr1 to a2.ctr2, and one from a2.ctr1 to the top, a1.ctr2. Conceptually, we can think of this situation like so:



And HiP-HOPS will correctly break these two loops and insert circle nodes accordingly:



However, consider a more complex scenario:

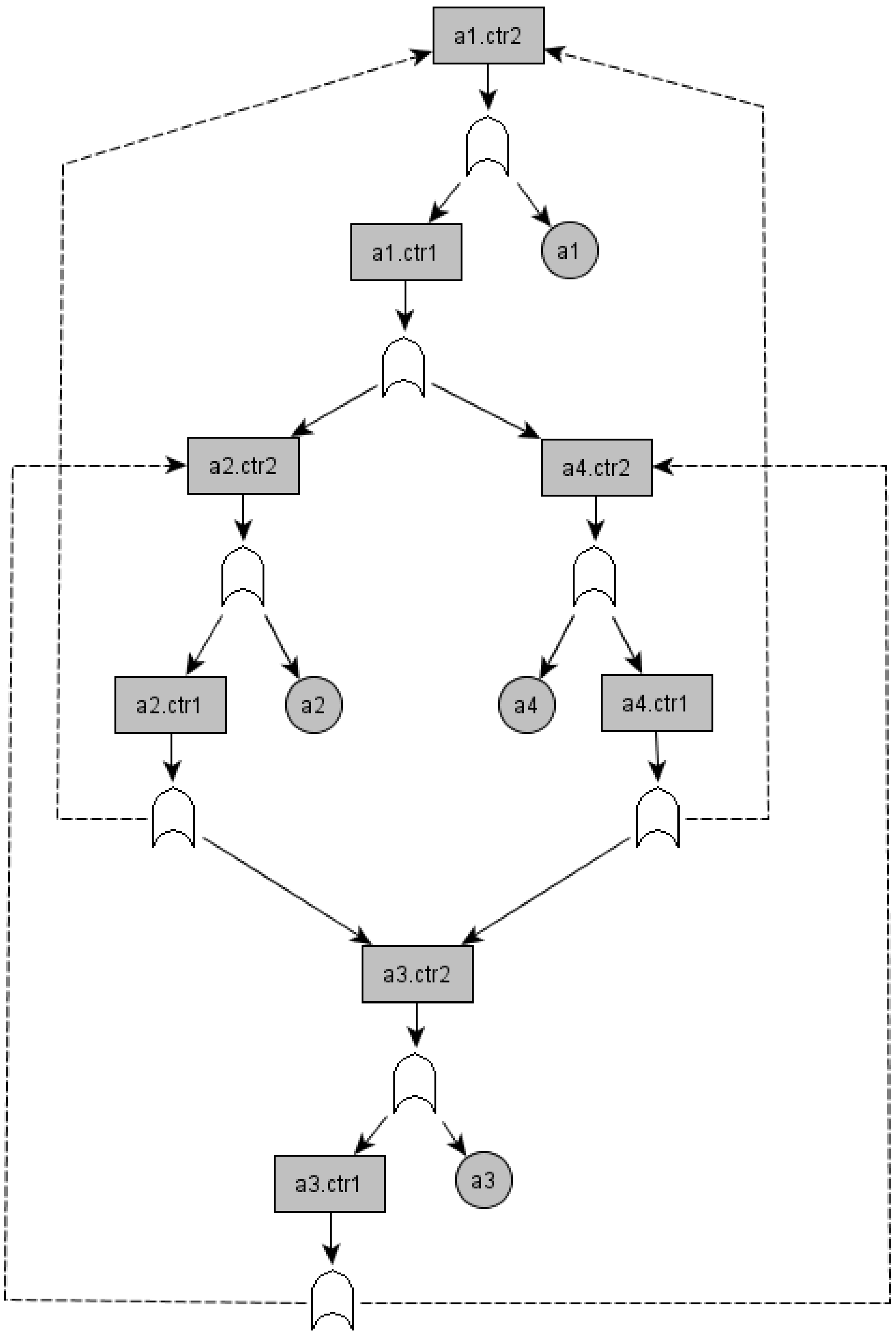
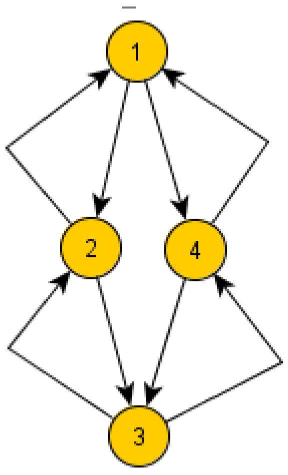


Figure 41: Uh oh

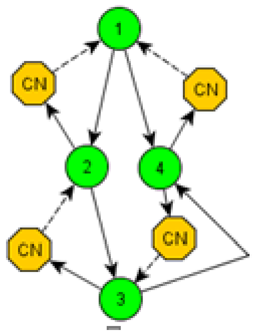
Again, we can conceptualise this tree as follows:



In this case, not only do we have multiple loops, but some of these loops have multiple entry points. The node labelled '1' is entered from both 2 and 4 (i.e., a2.ctr1 and a4.ctr1), while node 3 is entered from both 2 and 4 as well. In fact, all four of these nodes can be reached by two others in circular fashion.

The problem arises when we try to break these loops. Normally, we try to ensure the longest path length before each cut and inserted circle node. For example, if we traverse the network above from 1 → 2 → 3 → 2, we stop only when we loop back to 2, resulting in a maximum length path of 1 → 2 → 3 → (C), where (C) is the inserted circle node. Similarly, 1 → 2 → 1 would become 1 → 2 → (C) and so forth.

However, we can get non-deterministic results based on which path we follow first. Consider the maximum length path 1 → 2 → 3 → 4 → 3. In this case, we would cut only when we try to loop back to 3, i.e., we cut the link from 4 to 3 and get 1 → 2 → 3 → 4 → (C). So far so good, but now when we later follow the path 1 → 4 → 3 → 2 → 1, we find it has already been cut and only get 1 → 4 → (C) — and fail to reach node 3.

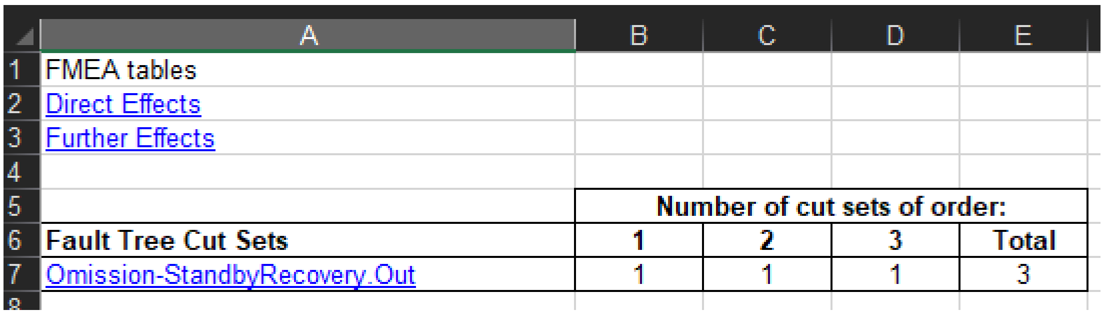


If we had happened to go down the other route first, we would get a first path of 1 → 4 → 3 → 2 → (C), but then 1 → 2 → (C).

This occurs due to the way HiP-HOPS processes paths of propagation through the interconnected fault trees. Because there is no safe way of severing the loops, HiP-HOPS instead generates an error message in this situation and aborts the analysis process. The user is then advised to try to remove or simplify the circular logic and try again.

5.2. Spreadsheet Output

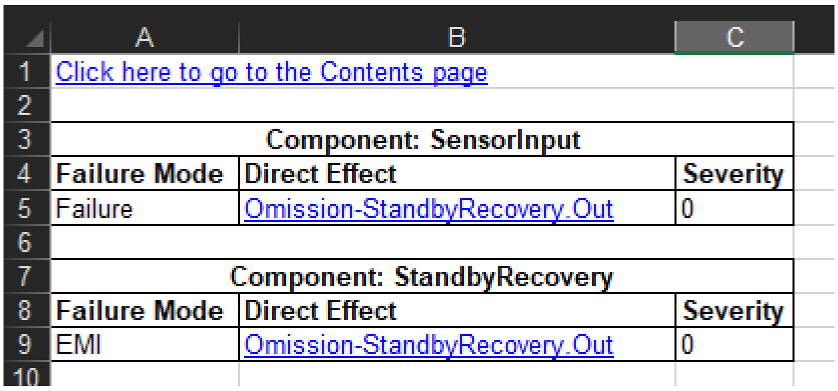
If the option to generate an Excel spreadsheet has been selected, then HiP-HOPS will output an XML file in the same output directory that can be opened in Excel. The structure of the spreadsheet is very similar to the browser output and begins with an index sheet:



	A	B	C	D	E
1	FMEA tables				
2	Direct Effects				
3	Further Effects				
4					
5		Number of cut sets of order:			
6	**Fault Tree Cut Sets**	**1**	**2**	**3**	**Total**
7	Omission-StandbyRecovery.Out	1	1	1	3
8					

Figure 42: Excel index sheet

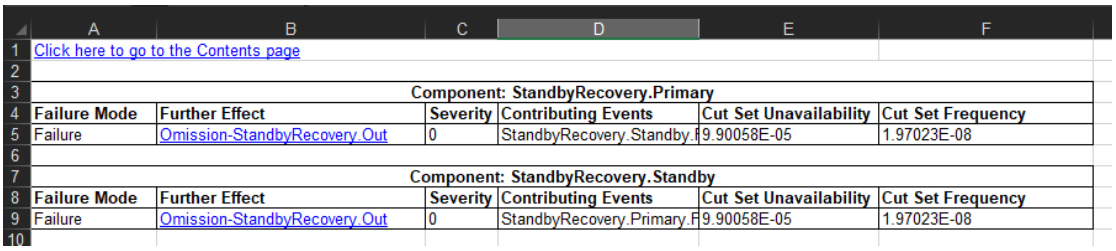
As usual, this displays the fault trees and the number of cut sets for each. The links are clickable and open either the FMEA tables or the fault tree sheets.



	A	B	C
1	Click here to go to the Contents page		
2			
3	**Component: SensorInput**		
4	**Failure Mode**	**Direct Effect**	**Severity**
5	Failure	Omission-StandbyRecovery.Out	0
6			
7	**Component: StandbyRecovery**		
8	**Failure Mode**	**Direct Effect**	**Severity**
9	EMI	Omission-StandbyRecovery.Out	0
10			

Figure 43: FMEA Direct Effects

The direct effects FMEA sheet shows all of the failure modes for each component and all the effects caused by that failure mode as a single point of failure, with severity if present.

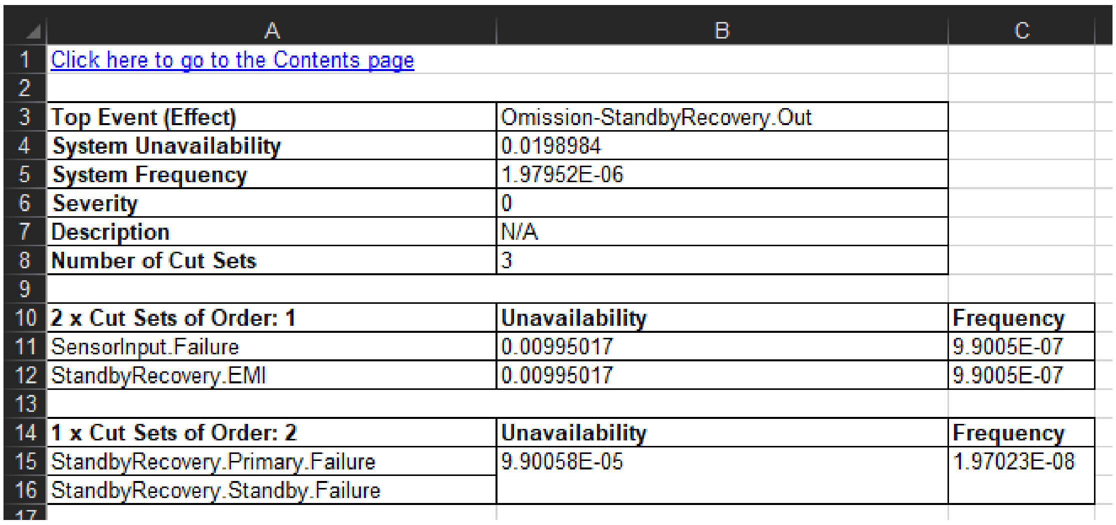


	A	B	C	D	E	F
1	Click here to go to the Contents page					
2						
3	**Component: StandbyRecovery.Primary**					
4	**Failure Mode**	**Further Effect**	**Severity**	**Contributing Events**	**Cut Set Unavailability**	**Cut Set Frequency**
5	Failure	Omission-StandbyRecovery.Out	0	StandbyRecovery.Standby.F	9.90058E-05	1.97023E-08
6						
7	**Component: StandbyRecovery.Standby**					
8	**Failure Mode**	**Further Effect**	**Severity**	**Contributing Events**	**Cut Set Unavailability**	**Cut Set Frequency**
9	Failure	Omission-StandbyRecovery.Out	0	StandbyRecovery.Primary.F	9.90058E-05	1.97023E-08
10						

Figure 44: And FMEA Further Effects

The further effects sheet is similar but also shows all of the contributing events that must also occur to acheve a given effect.

Clicking on either the sheet tab, the fault tree on the index page, or the effect on an FMEA page leads to the fault tree view:



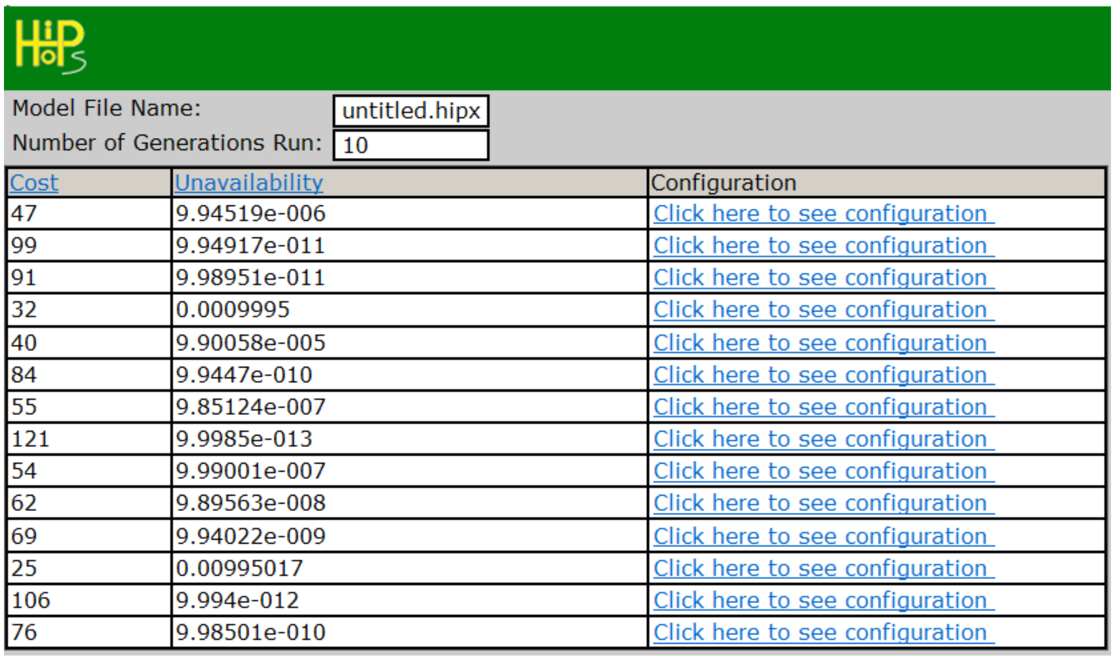
	A	B	C
1	Click here to go to the Contents page		
2			
3	**Top Event (Effect)**	Omission-StandbyRecovery.Out	
4	**System Unavailability**	0.0198984	
5	**System Frequency**	1.97952E-06	
6	**Severity**	0	
7	**Description**	N/A	
8	**Number of Cut Sets**	3	
9			
10	**2 x Cut Sets of Order: 1**	**Unavailability**	**Frequency**
11	SensorInput.Failure	0.00995017	9.9005E-07
12	StandbyRecovery.EMI	0.00995017	9.9005E-07
13			
14	**1 x Cut Sets of Order: 2**	**Unavailability**	**Frequency**
15	StandbyRecovery.Primary.Failure	9.90058E-05	1.97023E-08
16	StandbyRecovery.Standby.Failure		

Figure 45: Fault Tree view

There is no display of the fault tree itself, but its general information is displayed along with the list of the minimal cut sets, bunched by order.

5.3. Optimisation Output

Architectural optimisation results in browser output similar to the analysis output, except it shows the possible optimal system configurations that have been found. These are generated to a different directory, <*model name*>-OptimisationResults. The results show the various configurations along with their scores for each chosen objective (cost, weight, unavailability etc). Because it is a multi-objective optimisation, there is no single optimal solution, but all solutions should be pareto-optimal, i.e., they all represent trade-offs between the different objectives that are not worse than any of the other solutions generated.

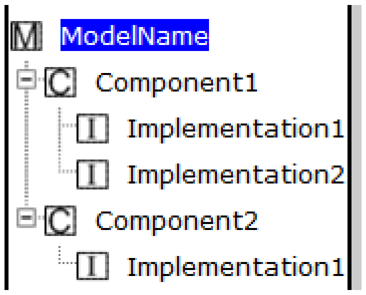


Model File Name: untitled.hipx
Number of Generations Run: 10

Cost	Unavailability	Configuration
47	9.94519e-006	Click here to see configuration
99	9.94917e-011	Click here to see configuration
91	9.98951e-011	Click here to see configuration
32	0.0009995	Click here to see configuration
40	9.90058e-005	Click here to see configuration
84	9.9447e-010	Click here to see configuration
55	9.85124e-007	Click here to see configuration
121	9.9985e-013	Click here to see configuration
54	9.99001e-007	Click here to see configuration
62	9.89563e-008	Click here to see configuration
69	9.94022e-009	Click here to see configuration
25	0.00995017	Click here to see configuration
106	9.994e-012	Click here to see configuration
76	9.98501e-010	Click here to see configuration

Figure 46: Optimisation output

In the final column is a hyperlink which shows the configuration of the model required to achieve the individual result. Following this hyperlink will display the model configuration in a tree format as in Figure 47.



```
M ModelName
⊟ C Component1
    I Implementation1
    I Implementation2
⊟ C Component2
    I Implementation1
```

Figure 47: Solution configuration

Each configurable component in the model is listed along with the implementations required.

6. HiP-HOPS Tutorials

In this section, we will explain how to perform various common tasks by working through some tutorials. These will cover:

- How to create, annotate, and analyse a simple Standby-Recovery model
- Updating the Standby-Recovery model to be hierarchical
- How to use common cause failures
- How to use multi-perspective modelling and allocation
- How to create a model for SIL decomposition
- How to create an optimisation model

Each of the tutorials has an accompanying model file showing the final result in the "Examples" subdirectory of your HiP-HOPS installation. You can either work alongside the tutorial to recreate the model or load up the finished example to compare against.

6.1. Tutorial 1: Analysing a simple model

For the first tutorial, we will create a model of a simple standby-recovery system like the one described in section 2.

6.1.1. Creating the model

First, create a new, blank Simulink model (select "New" from the Home menu, select New Simulink Model, then choose a Blank model). The first step is typically to create the components and their ports before doing any annotation of failure data. For this, we need three components like so (you can just use the standard subsystem block):

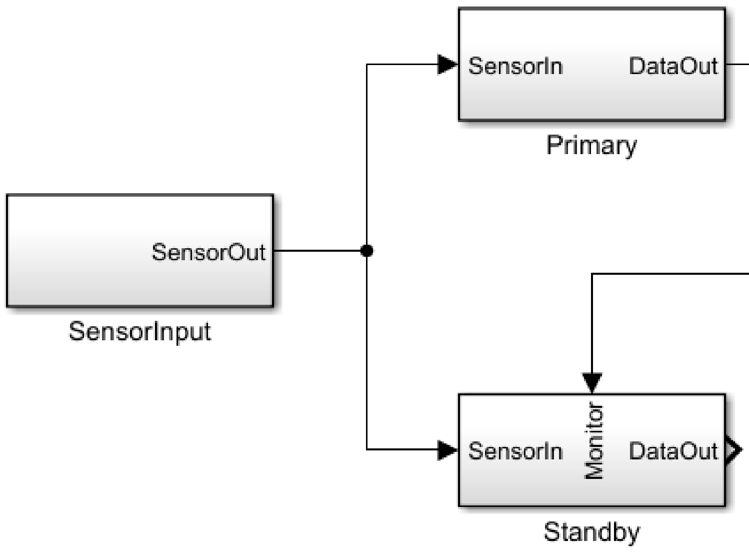


Figure 48: Model for Standby-Recovery tutorial

You will need to remove the input port from SensorInput and add a new one to Standby. To create the second branch from SensorInput to Standby, start from the SensorIn port and drag backwards.

Note that it is necessary to remove the automatic input-output connections that Simulink adds in the subsystem level for each level, otherwise this adds a new channel for propagation straight through each component.

The idea here is that we have a sensor that provides input to the system, then two redundant components that perform some calculation using that sensor input and provide output. By default, the system uses the Primary component, but a Standby component monitors the output of Primary and takes over if no output is detected. Note that other configurations are also possible, e.g. using a comparator or a voter, but here we will keep things simple.

6.1.2. Annotating the model

The next step is to begin annotating each component with failure data that describes how it fails and responds to failure. If you have not already opened the HiP-HOPS launcher, do so by typing `hiphops` into the command window. Then, select the SensorInput component and press the 'Edit component failure data' button on the Launcher.

The data we want to add to each component is listed in the tables below.

Component	Failure Modes	Failure Rate
SensorInput	sensorFailure	0.0005
Primary	calculationFailure	0.0002
Standby	calculationFailure	0.0002

Table 3: Failure modes for Standby-Recovery tutorial

Component	Output Deviations	Failure Expression
SensorInput	Omission-SensorOut	sensorFailure
Primary	Omission-DataOut	calculationFailure OR Omission-SensorIn
Standby	Omission-DataOut	(calculationFailure OR Omission-SensorIn) AND Omission-Monitor

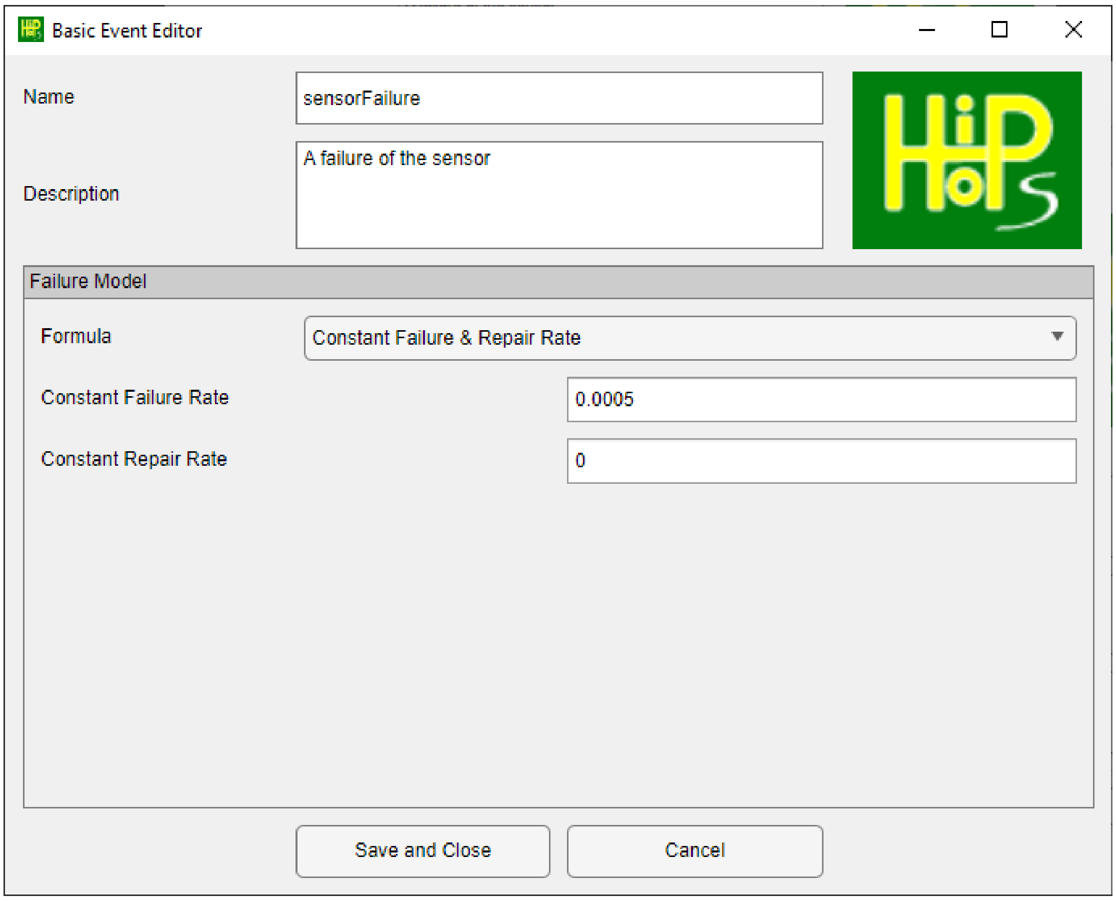
Table 4: Output Deviations for Standby-Recovery tutorial

The logic here is relatively straightforward:

- We get an omission of output from the sensor if the sensor suffers a failure;
- We get an omission of data from the Primary if either there is a calculation failure or if the Primary does not receive any input;

- We get an omission of data from the Standby if it has activated (i.e., no input received at the monitoring port) and if either there is no input or if the Standby also suffers from a calculation failure.

To begin with, we need to add a new basic event to the SensorInput component and add the information above. We use the "Constant Failure & Repair Rate" formula and set the failure rate to 0.05.



Name	sensorFailure
Description	A failure of the sensor

Failure Model

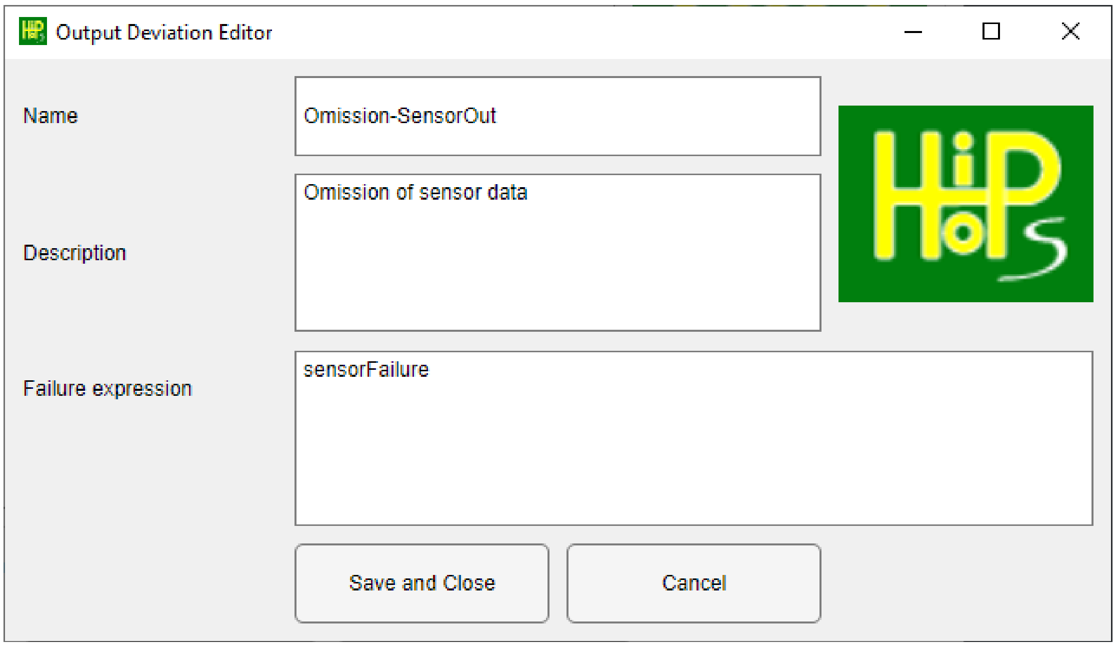
Formula	Constant Failure & Repair Rate
Constant Failure Rate	0.0005
Constant Repair Rate	0

Figure 49: SensorInput failure data

Remember to press Save and Close.

Next we add an Output Deviation (switch to the Output Deviations tab and press 'Add'). Here we specify the name, which consists of the failure class — `Omission` in this case — and the port name, `SensorOut`. Note that here we do not need to fully qualify the port name; HiP-HOPS understands that this port belongs to the current component.

The failure expression is very simple, since there is no input port and only one basic event.

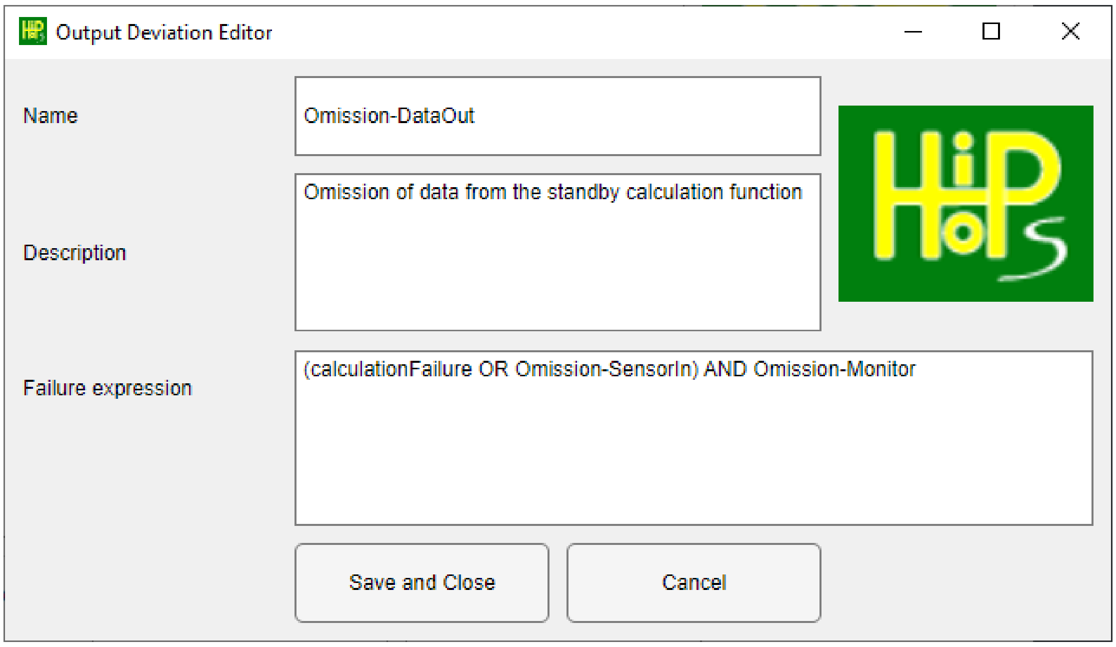


Output Deviation Editor

Name	Omission-SensorOut
Description	Omission of sensor data
Failure expression	sensorFailure

Figure 50: Output Deviation for SensorInput component

Repeat the process for the other two components, using the appropriate data from the tables above. Remember to select each component before pressing the 'Edit component's failure data' button. The output deviation for Standby should look like this:



Output Deviation Editor

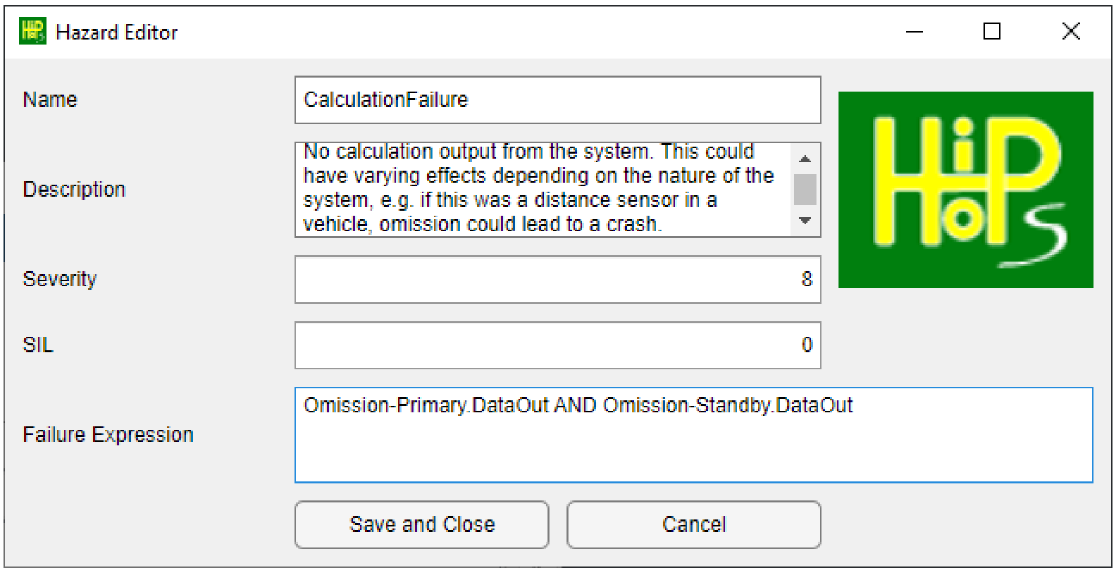
Name	Omission-DataOut
Description	Omission of data from the standby calculation function
Failure expression	(calculationFailure OR Omission-SensorIn) AND Omission-Monitor

Figure 51: And for the Standby component

Again, remember also to press 'Save and Close' each time.

The final step of the annotations is to edit the model-level failure data. From the Launcher, press the 'Edit model failure data' button.

First, add a new hazard and fill it out like so:



Name: CalculationFailure

Description: No calculation output from the system. This could have varying effects depending on the nature of the system, e.g. if this was a distance sensor in a vehicle, omission could lead to a crash.

Severity: 8

SIL: 0

Failure Expression: Omission-Primary.DataOut AND Omission-Standby.DataOut

Figure 52: Hazard for Standby-Recovery system

Here we define just one hazard, a failure to calculate a value from the sensor input. We've kept things abstract, so the effects of this are unclear, but e.g. if the system was meant to calculate a distance to an obstacle for a vehicle, failure could lead to the vehicle crashing. Hence we give it a relatively high severity value of 8.

The failure expression should be a conjunction of output from both Primary and Standby. Note that in this case the output deviations must be fully qualified, as we are working at the model level now, not at component level.

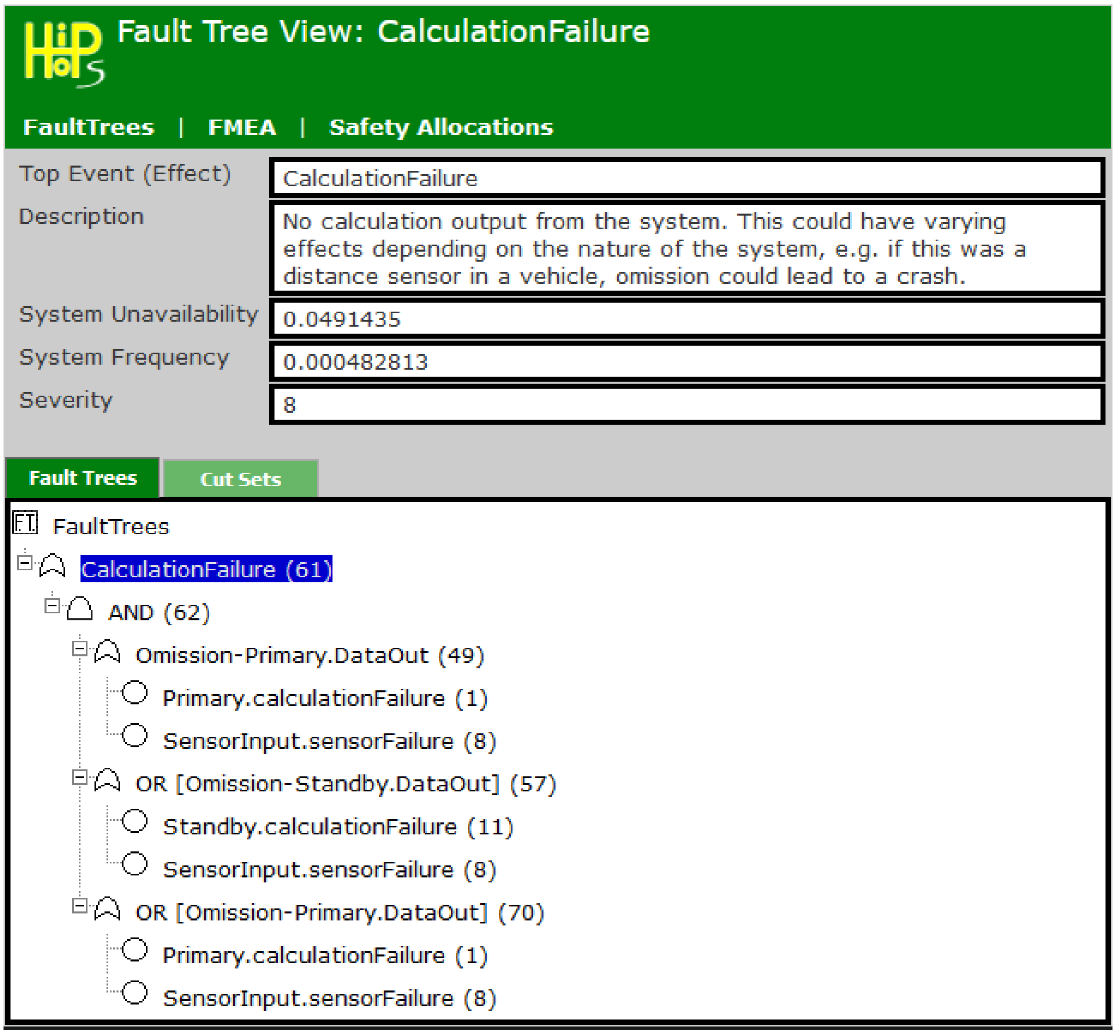
On the main model parameters window, set the risk time to 100 and add a description if you like. Then press Save and Close again.

6.1.3. Analysing the model and viewing the output

We should now be ready to analyse the model. Before doing so, **make sure to save the model in Simulink as well**. Press the save button (you will probably need to give it a filename too).

Once done, press 'Analyse model' on the Launcher.

If all went well, you should see some text in the Matlab Command Window indicating that HiP-HOPS was running and then the output should open up automatically in your browser. Since we only defined one hazard, we only get one fault tree:



Fault Tree View: CalculationFailure

FaultTrees | FMEA | Safety Allocations

Top Event (Effect)	CalculationFailure
Description	No calculation output from the system. This could have varying effects depending on the nature of the system, e.g. if this was a distance sensor in a vehicle, omission could lead to a crash.
System Unavailability	0.0491435
System Frequency	0.000482813
Severity	8

Fault Trees | Cut Sets

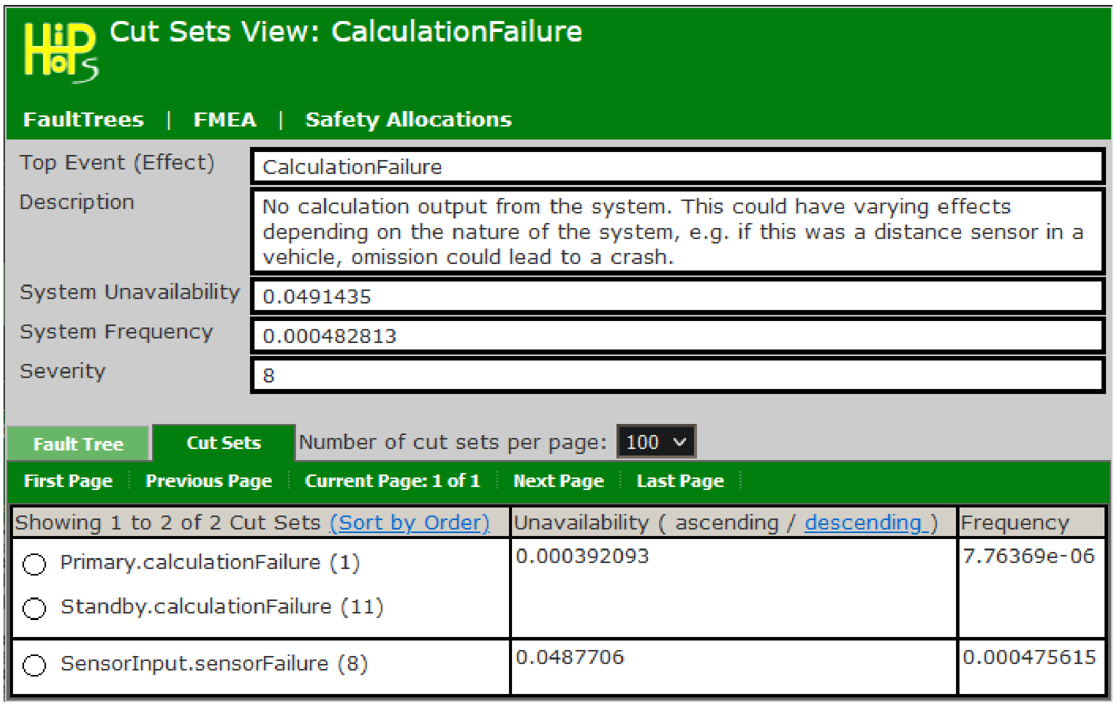
```
FaultTrees
CalculationFailure (61)
  AND (62)
    Omission-Primary.DataOut (49)
      Primary.calculationFailure (1)
      SensorInput.sensorFailure (8)
    OR [Omission-Standby.DataOut] (57)
      Standby.calculationFailure (11)
      SensorInput.sensorFailure (8)
    OR [Omission-Primary.DataOut] (70)
      Primary.calculationFailure (1)
      SensorInput.sensorFailure (8)
```

Figure 53: Output for tutorial 1

If we switch to the minimal cut sets, we see we only have two:

- A failure of the SensorInput (sensorFailure)
- A failure of both Primary and Standby (both calculationFailure)

This is what we would expect. The SensorInput is a single point of failure, since it is a common input to both Primary and Standby; without its output, the system cannot function. However, because we have redundancy in the calculation, it requires both Primary and Standby to fail to cause a calculation failure.



Top Event (Effect)	CalculationFailure
Description	No calculation output from the system. This could have varying effects depending on the nature of the system, e.g. if this was a distance sensor in a vehicle, omission could lead to a crash.
System Unavailability	0.0491435
System Frequency	0.000482813
Severity	8

Number of cut sets per page: 100

Showing 1 to 2 of 2 Cut Sets (Sort by Order)	Unavailability (ascending / descending)	Frequency
○ Primary.calculationFailure (1) ○ Standby.calculationFailure (11)	0.000392093	7.76369e-06
○ SensorInput.sensorFailure (8)	0.0487706	0.000475615

Figure 54: Minimal cut sets

Because we also provided quantitative failure data, we get estimates for the unavailability and frequency for both the cut sets and the fault tree as a whole. This follows the formulae set out earlier. For example, we calculate the unavailability of a basic event using the Constant Failure & Repair Rate formula:

$$u = \frac{\lambda}{\lambda + \mu} \times (1 - e^{-(\lambda+\mu)t})$$

Since the repair rate is 0, this is equal to 1 – e $^{-(\text{failureRate} * t)}$. *t* is the global risk time, since we did not specify a different risk time for any component. So for the sensor, this value is:

1 – e ^ -(0.0005 * 100) = 0.0487706

and for the Primary and Standby:

1 – e ^ -(0.0002 * 100) = 0.0198013

For the SensorInput cut set, since there is only one basic event, the unavailability is unchanged. For the other cut set, we multiply them together since they are connected via an AND relationship, which gives us 0.0003921.

Then to calculate the unavailability for the fault tree, we must apply the Esary-Proschan formula:

$$Q_S = 1 - \prod_{i=1}^{n} (1 - Q_{CS_i})$$

where Q_s is the unavailability of the system and Q_{cs} is the unavailability of each cut set. In this case, this is equivalent to:

$$
\begin{aligned}
& 1 - (1 - 0.0487706) * (1 - 0.0003921) \\
= \quad & 1 - (0.9512294 * 0.9996079) \\
= \quad & 1 - 0.9508565 \\
= \quad & 0.0491435
\end{aligned}
$$

Fortunately, HiP-HOPS does all of this for us. It also calculates the failure frequency as well.

Hopefully this first tutorial explains the basics of how to create and annotate a system model for analysis with HiP-HOPS.

6.2. Tutorial 2: Hierarchical models

For the second tutorial, we will modify the standby-recovery system from Tutorial 1 to be hierarchical and introduce failures at the subsystem level.

6.2.1. Creating the model

Create a new, blank Simulink model and add a new component at the top level. Then copy the SensorInput component from the first Tutorial and connect them like so:

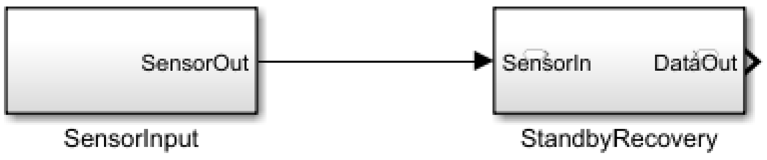


Figure 55: Making it hierarchical

Because the failure annotations are stored as extra data within each component (or model), copying a component also copies the failure data, saving us from having to annotate it again.

In this case, we want to make the StandbyRecovery function its own subsystem. By doing so, we can more easily reuse it elsewhere, e.g. in another model, if we wanted to. So far we have added the top-level subsystem, so the next step is to copy the Primary and Standby components in as subcomponents of this block (double click on StandbyRecovery to open the subsystem level).

However, we have a slight problem: we can connect the SensorIn port to the two subcomponent inputs, but how do we connect the two subcomponent outputs to the parent DataOut output? Matlab allows an output port to connect to multiple inputs, but not multiple outputs to a single input (or output in this case).

The workaround is to add a dummy "Joiner" component with two inputs and one output. This is not a "real" component and does not have its own basic events; it serves only to host the logic that joins the outputs together.

When you are finished, it should look something like this:

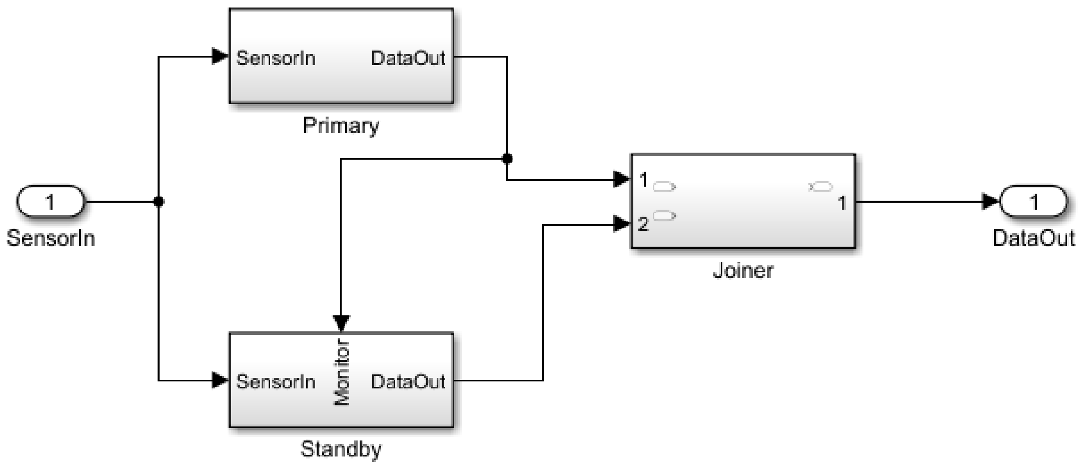


Figure 56: The StandbyRecovery subsystem

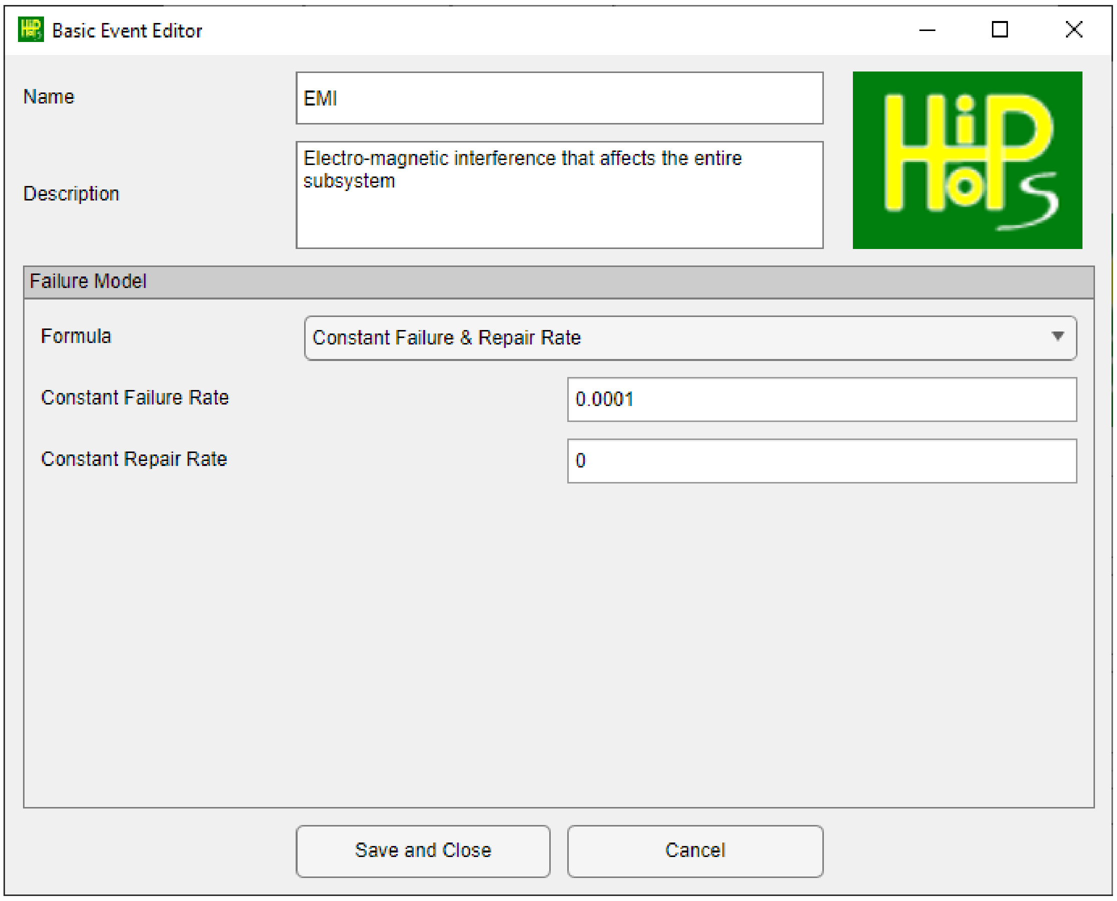
You will need to define a new output deviation in Joiner like so:

- Omission-Out1 = Omission-In1 AND Omission-In2

6.2.2. Adding failures at the subsystem level

Before we move on to analysis, we will add something else new. Exit the subsystem level and return to the top level and select the StandbyRecovery subsystem block. Here we can choose to select how it propagates failure; we could e.g. specify that it only propagates deviations from its subcomponents by selecting "Defined in the subsystem only" from the drop-down menu at the bottom, but for now we will leave it as "Defined here and in the subsystem".

This is because we want to add a new basic event that applies to the subsystem as a whole: EMI.


Basic Event Editor

Name	EMI
Description	Electro-magnetic interference that affects the entire subsystem

Failure Model

Formula	Constant Failure & Repair Rate
Constant Failure Rate	0.0001
Constant Repair Rate	0

Figure 57: The EMI basic event

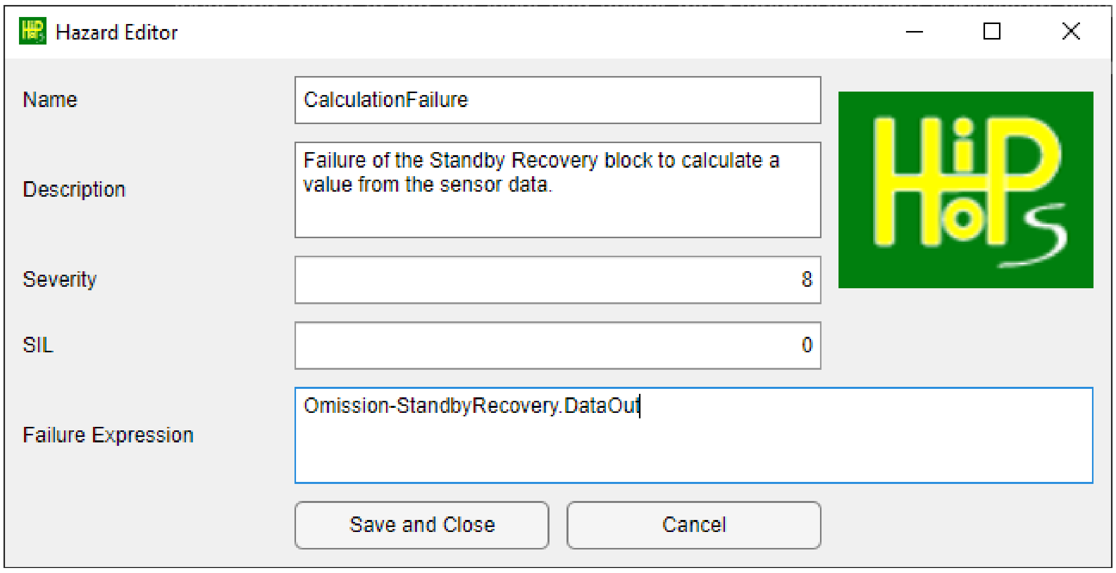
This functions like a limited common cause failure. We could achieve the same effect by adding an EMI basic event to all subcomponents (Primary and Standby), but it is more expedient to add it once to the subsystem instead. We then define a new output deviation that uses it:

- Omission-DataOut = EMI

and then save and close.

Note that we now have two separate output deviations that both propagate to StandbyRecovery.DataOut: the one in StandbyRecovery.Joiner and the one in StandbyRecovery itself. The drop-down menu mentioned earlier tells HiP-HOPS which of these output deviations to use. Since we chose to use both in this case, it will join them together using an implicit OR.

The last step is to add the hazard and risk time in the Model Parameters editor. Unfortunately these cannot simply be copied across but we can reuse the same data. Since we have added the AND logic to the Joiner, we do not need it for the hazard and can just refer to the output port of the StandbyRecovery subsystem:



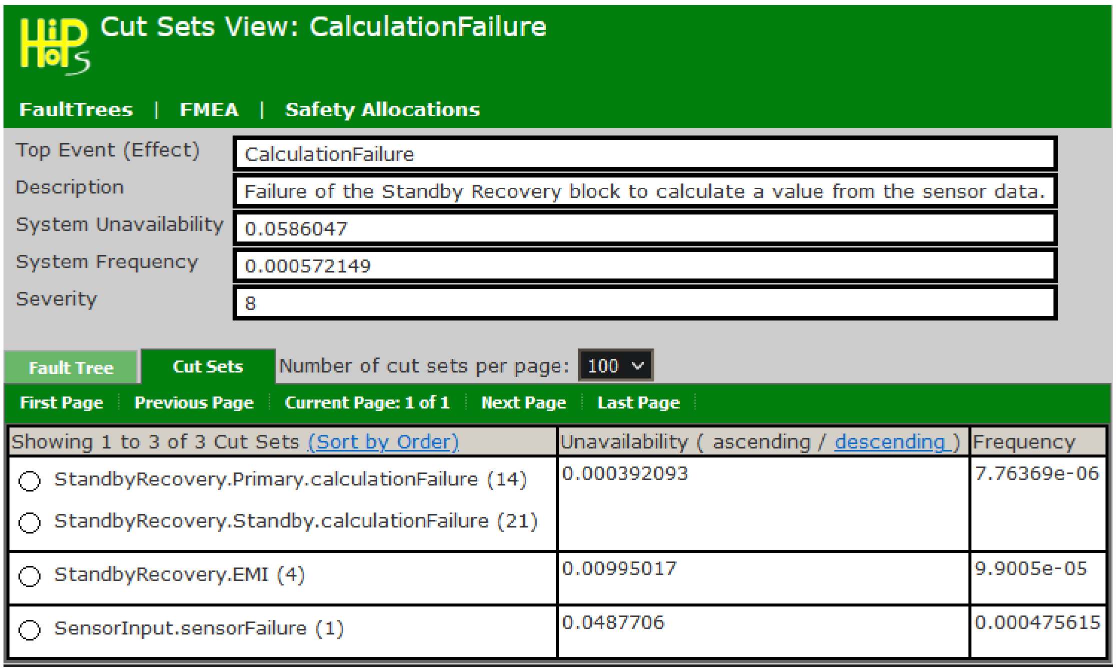
Hazard Editor

Name	CalculationFailure
Description	Failure of the Standby Recovery block to calculate a value from the sensor data.
Severity	8
SIL	0
Failure Expression	Omission-StandbyRecovery.DataOut

Figure 58: Hazard for tutorial 2

6.2.3. Analysis Results

Once done, make sure to save the model in Simulink and press Analyse model. If all went well, the results will appear.



Cut Sets View: CalculationFailure

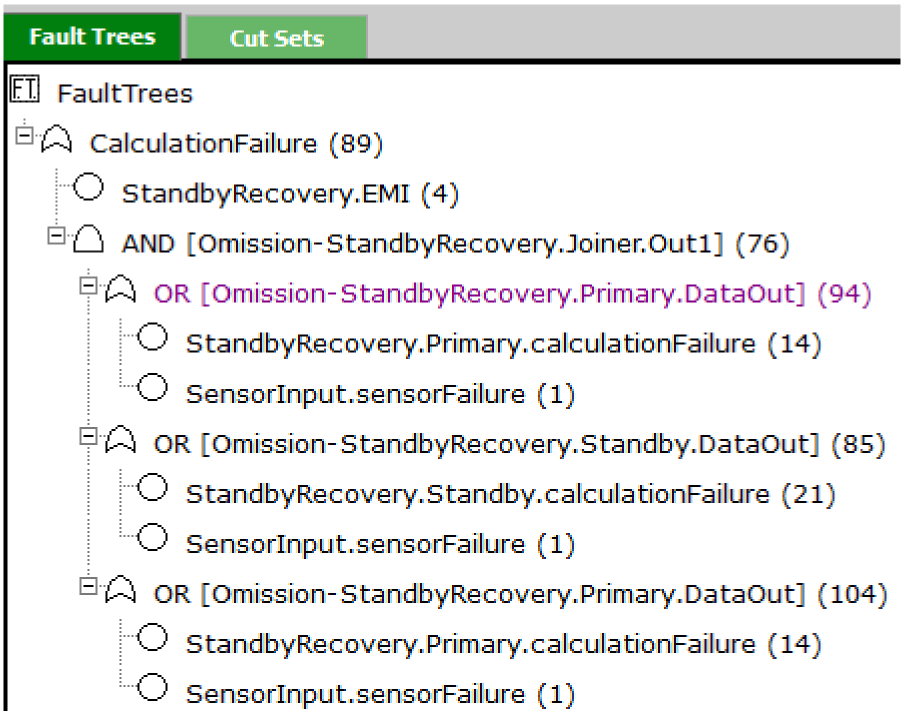
Top Event (Effect)	CalculationFailure
Description	Failure of the Standby Recovery block to calculate a value from the sensor data.
System Unavailability	0.0586047
System Frequency	0.000572149
Severity	8

Showing 1 to 3 of 3 Cut Sets (Sort by Order)	Unavailability (ascending / descending)	Frequency
StandbyRecovery.Primary.calculationFailure (14) StandbyRecovery.Standby.calculationFailure (21)	0.000392093	7.76369e-06
StandbyRecovery.EMI (4)	0.00995017	9.9005e-05
SensorInput.sensorFailure (1)	0.0487706	0.000475615

Figure 59: Tutorial 2 results

As we would expect, the results are similar to before. The SensorInput failure is exactly the same and the Primary + Standby calculation failure is also the same, except now the names indicate these are subcomponents of the StandbyRecovery block. The difference is that we now have a third minimal cut set for EMI, which also increases the overall system unavailability.

You can see how HiP-HOPS has joined the output deviations at different hierarchical levels by looking at the fault tree structure:



```
FaultTrees
  CalculationFailure (89)
    StandbyRecovery.EMI (4)
    AND [Omission-StandbyRecovery.Joiner.Out1] (76)
      OR [Omission-StandbyRecovery.Primary.DataOut] (94)
        StandbyRecovery.Primary.calculationFailure (14)
        SensorInput.sensorFailure (1)
      OR [Omission-StandbyRecovery.Standby.DataOut] (85)
        StandbyRecovery.Standby.calculationFailure (21)
        SensorInput.sensorFailure (1)
      OR [Omission-StandbyRecovery.Primary.DataOut] (104)
        StandbyRecovery.Primary.calculationFailure (14)
        SensorInput.sensorFailure (1)
```

Figure 60: The fault tree

Because EMI is a direct cause of Omission-DataOut (the cause of the hazard), it appears right under the top event. SensorInput.sensorFailure is also a single point of failure but occurs three times in the tree: once as a cause of Omission-Standby.SensorIn, once as a cause of Omission-Primary.SensorIn, and once as an indirect cause of Omission-Standby.Monitor (via Primary). These three events all have the same ID, indicating they're the same, so HiP-HOPS minimises them accordingly.

Hopefully this tutorial has demonstrated how subsystems can be used to create hierarchical models that can be used with HiP-HOPS. Not only does this reduce clutter by separating out related components into logical subsystems, it also allows us to define basic events at the subsystem level that effectively apply as local common cause failures to all subcomponents in that subsystem.

6.3. Tutorial 3: Common Cause Failures

For the third tutorial, we will again modify the standby-recovery system — this time to include a perspective and a common cause failure. A perspective is like a modelling view or layer, a way of grouping together components on a more abstract level. Perspectives also provide a home for common cause failures, which is what we will achieve in this tutorial.

6.3.1. Creating the model

Create a new, blank Simulink model and add a new component at the top level called 'Hardware'. Make sure it has no input or output ports. Then copy both the SensorInput and StandbyRecovery components from Tutorial 2 inside it as subcomponents.

It should result in a very simple topmost layer:

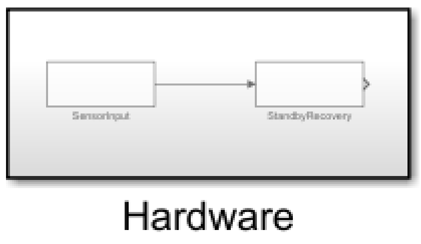


Figure 61: Introducing perspectives

As far as Simulink is concerned, this is simply a subsystem component with two subcomponents and no inputs or outputs — and at first the same is true for HiP-HOPS. To get HiP-HOPS to treat it as a perspective, we need to use the second button on the launcher to open the Component Editor. Then in the Type drop-down menu at the top, select 'Perspective'. This changes it from a normal component to a perspective as far as HiP-HOPS is concerned.

Perspectives are merely containers. They do not have any failure logic of their own except for common cause failures.

6.3.2. Adding a common cause failure

When we changed the component to a perspective, the previous fields were replaced by a list of Actual Common Cause Failures.

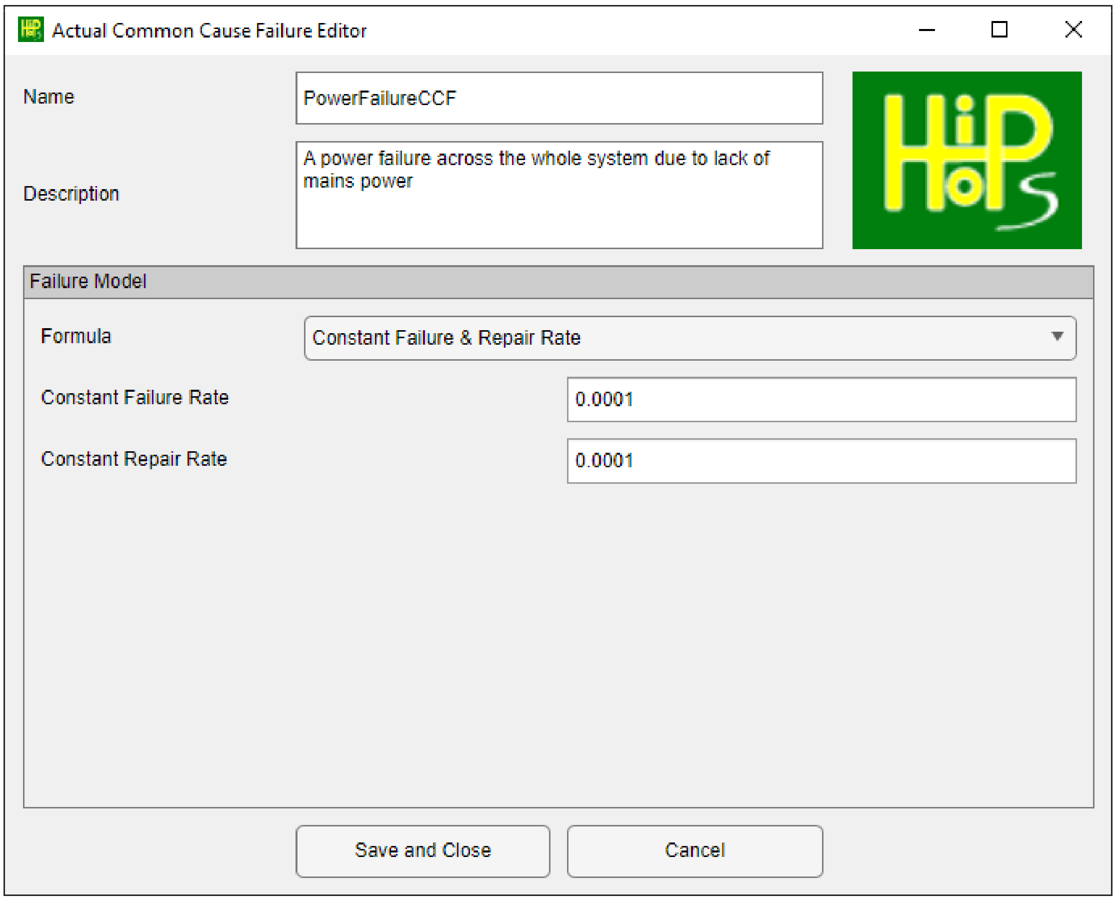
HiP-HOPS splits common cause failures (CCFs) into two types: Actual CCFs (or ACCFs) and Potential CCFs (or PCCFs). The idea is that PCCFs are defined in component failure logic as proxies or placeholders. They become part of the logic but are effectively dormant unless paired with a corresponding ACCF. The main reason for this is reuse: we can reuse (i.e., copy and paste) a component with a PCCF defined without any problem, since the PCCF is defined as part of the component, whereas if we had a direct reference to an ACCF, it would break every time we tried to use it elsewhere, since the ACCF is defined in the perspective.

The solution is therefore to have PCCFs defined in the component failure data and match them with an Actual CCF when we want them to become 'active'.

This approach to CCFs has several benefits. For example, we can define a "waterIngress" PCCF for various components we expect to be used aboard a ship. Then when creating a model of the ship systems, we can define ACCFs that indicate flooding of different compartments — engine room, generator room, control room etc.

Then we connect up the component PCCFs to ACCFs depending on which room they're in. If we "move" a component to a different room, all we need to do is change which ACCF it references.

If that sounds complicated, fear not: using CCFs in HiP-HOPS is quite easy. First, we can add the ACCF to the perspective. In this case, we will make it a power failure CCF. It is often convenient to make power failure a CCF as this saves us connecting everything with power lines and propagating power failures throughout the entire model.


Actual Common Cause Failure Editor

Name: PowerFailureCCF

Description: A power failure across the whole system due to lack of mains power

Failure Model

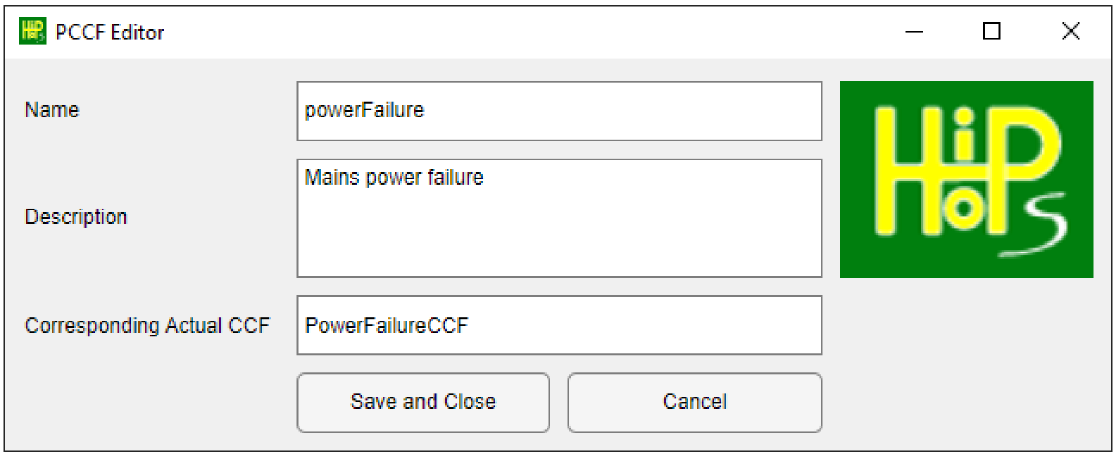
Formula: Constant Failure & Repair Rate

Constant Failure Rate: 0.0001

Constant Repair Rate: 0.0001

Figure 62: Adding the ACCF

The next step is to add PCCFs to our subcomponents, SensorInput and StandbyRecovery. PCCFs are defined in the Component Failure Data Editor like Basic Events (2nd tab).


PCCF Editor

Name	powerFailure
Description	Mains power failure
Corresponding Actual CCF	PowerFailureCCF

Figure 63: Defining the PCCFs

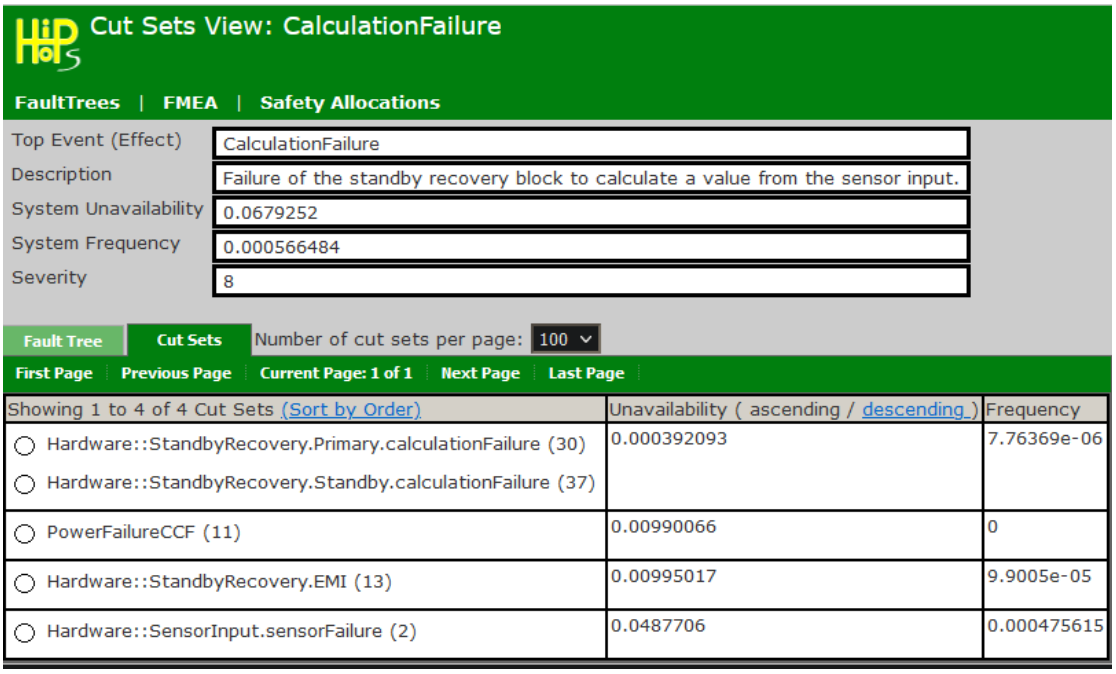
We can use the same name for both (`powerFailure`). For the corresponding ACCF, we specify `powerFailureCCF`. We also need to update the output deviations to include them:

- For SensorInput: `SensorFailure OR powerFailure`
- For StandbyRecovery: `EMI OR powerFailure`

After that, set the model parameters (risk time = 100, CalculationFailure hazard = Omission-Hardware::StandbyRecovery.DataOut), then save and analyse the model.

6.3.3. Analysis Results

With luck, the analysis should have succeeded and you will see results similar to those below:

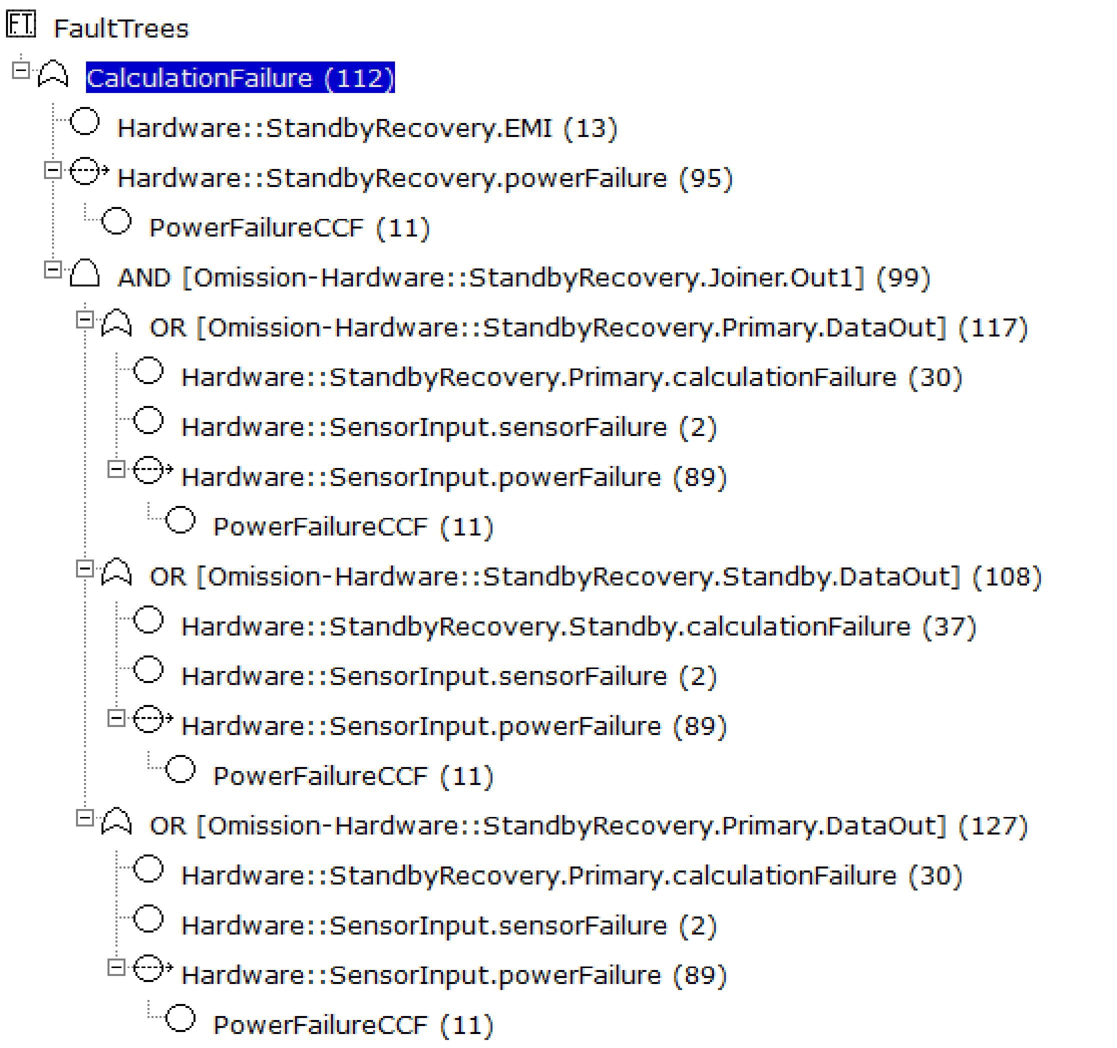

Cut Sets View: CalculationFailure

Top Event (Effect)	CalculationFailure
Description	Failure of the standby recovery block to calculate a value from the sensor input.
System Unavailability	0.0679252
System Frequency	0.000566484
Severity	8

Showing 1 to 4 of 4 Cut Sets (Sort by Order)	Unavailability (ascending / descending)	Frequency
Hardware::StandbyRecovery.Primary.calculationFailure (30) Hardware::StandbyRecovery.Standby.calculationFailure (37)	0.000392093	7.76369e-06
PowerFailureCCF (11)	0.00990066	0
Hardware::StandbyRecovery.EMI (13)	0.00995017	9.9005e-05
Hardware::SensorInput.sensorFailure (2)	0.0487706	0.000475615

Figure 64: CCF analysis results

This time we get another new basic event, the PowerFailureCCF common cause failure. It acts just like any other basic event here, but if we view the fault tree, we can see how it appears:



```
FaultTrees
CalculationFailure (112)
  Hardware::StandbyRecovery.EMI (13)
  Hardware::StandbyRecovery.powerFailure (95)
    PowerFailureCCF (11)
  AND [Omission-Hardware::StandbyRecovery.Joiner.Out1] (99)
    OR [Omission-Hardware::StandbyRecovery.Primary.DataOut] (117)
      Hardware::StandbyRecovery.Primary.calculationFailure (30)
      Hardware::SensorInput.sensorFailure (2)
      Hardware::SensorInput.powerFailure (89)
        PowerFailureCCF (11)
    OR [Omission-Hardware::StandbyRecovery.Standby.DataOut] (108)
      Hardware::StandbyRecovery.Standby.calculationFailure (37)
      Hardware::SensorInput.sensorFailure (2)
      Hardware::SensorInput.powerFailure (89)
        PowerFailureCCF (11)
    OR [Omission-Hardware::StandbyRecovery.Primary.DataOut] (127)
      Hardware::StandbyRecovery.Primary.calculationFailure (30)
      Hardware::SensorInput.sensorFailure (2)
      Hardware::SensorInput.powerFailure (89)
        PowerFailureCCF (11)
```

Figure 65: CCF fault tree

Here we can see the PCCFs appearing as proxy nodes (a circle with a dotted arrow through it), each with the ACCF (ID# 11) as the sole cause.

Hopefully this tutorial has shown you how you can create a perspective at the top level of your model and use it to specify common cause failures that can be referenced from any component in that perspective.

In the next tutorial, we will expand on the perspective concept to perform multi-perspective modelling and analysis.

6.4. Tutorial 4: NOT gates and non-coherent fault trees

For the fourth tutorial, we will look at how to use NOT gates and what that means for the fault tree analysis. We will also make use of multiple types of deviations this time.

6.4.1. Creating the system model

The system is going to be a very simple one. We need three components like so:

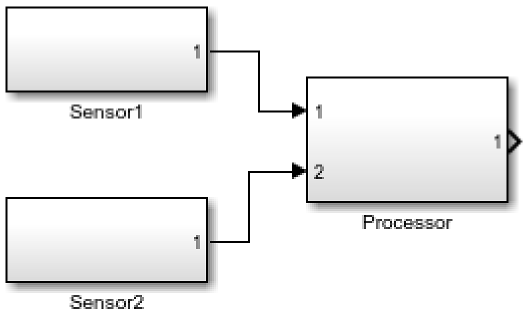


Figure 66: Tutorial 4 system model

The two sensors are identical and each have two basic events that lead to one of two classes of output deviation:

- An *omission* failure, i.e. an omission of expected output, is caused by an internal `sensorFailure`
- A *value* failure, i.e. output is present but incorrect in some way, is caused by an internal `sensorError`

The processor is meant to take the data from the two sensors and perform some calculation with it. It is intended to be resilient to value deviations and fail silent where possible, thereby transforming value failures (which are harder to detect and deal with) into omission failures (which are generally less severe). For example, the processor may be intended to send instructions to a hazardous piece of machinery, where no action is less severe than incorrect action (e.g. doing something before people have moved to a safe distance, outside the safety envelope).

However, the processor is also meant to continue operation in the case where one sensor has failed, and so there is a limit to how resilient it can be in the face of an omission of one input.

6.4.2. Failure logic

To model the processor logic correctly, we need to look at the possible combinations of input deviations and their effect on the output, where O = omission, V = value, and N = normal (i.e., no failure).

In1	In2	Out1	Notes
N	N	N	All inputs nominal
N	V	O	Inputs differ, so output nothing
V	N	O	Inputs differ, so output nothing
V	V	O	Inputs differ, so output nothing
O	N	N	Ignore one missing input and use the other
N	O	N	Ignore one missing input and use the other
O	O	O	Both inputs are missing, so no output possible
O	V	V	The only available input is incorrect = value deviation
V	O	V	The only available input is incorrect = value deviation

Here we can see the logic more clearly. If the inputs differ (due to value deviations), then the processor will choose to output nothing rather than risk outputting an incorrect value. However, if one input is missing (due to an omission), it will be forced to use the sole remaining input, even if it is incorrect.

The difficulty here is in distinguishing between different cases. For example, where there is an omission of input 1, the outcome depends on the state of input 2:

- Value-In2 results in a value deviation (the only input is in error)
- Omission-In2 results in an omission (no inputs available)
- No deviation of In2 results in no deviation of the output (the sole input is correct and can be used normally)

To describe this in logical expressions, NOT gates are required:

- Omission-In1 AND Value-In2 AND ~Omission-In2
- Omission-In1 AND ~Value-In2 AND Omission-In2
- Omission-In1 AND ~Value-In2 AND ~Omission-In2

The failure logic for the processor is therefore as follows:

Value-Out1
`(Value-In1 AND Omission-In2) OR (Omission-In1 AND Value-In2) OR processorError`

Omission-Out1
`(NOT Omission-In1 AND Value-In2) OR (Value-In1 AND NOT Omission-In2) OR (Value-In1 AND Value-In2) OR (Omission-In1 AND Omission-In2) OR processorFailure`

In other words, if there is one omission and one value deviation at the inputs, or `processorError` occurs, we get a value deviation of the output. If one input suffers a value deviation and the other is *not* omitted, we get an omission — as we do if both inputs suffer value deviations, both inputs are omitted, or if the processor itself fails.

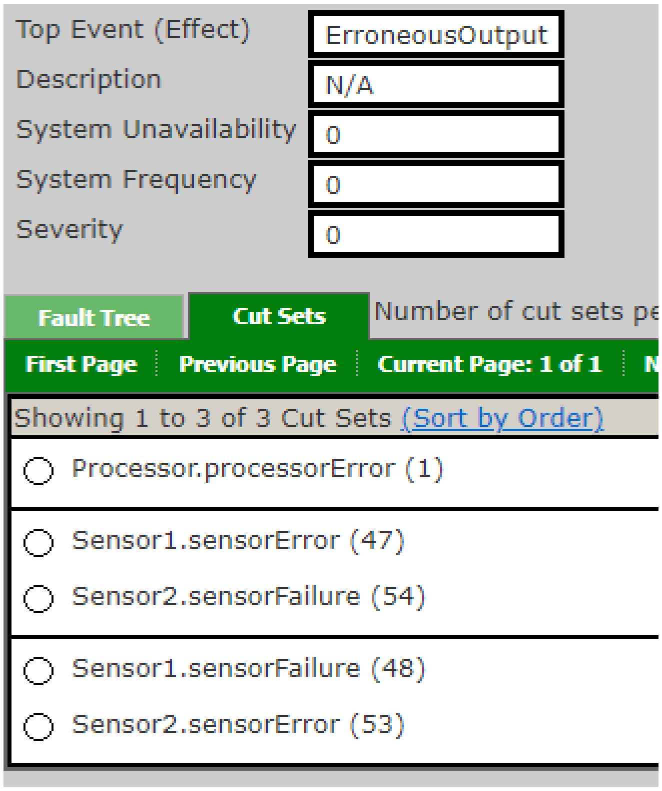
The hazards are then very simple:

- NoOutput = Omission-Processor.Out1

- ErroneousOutput = Value-Processor.Out1

6.4.3. Analysis & Results

For this model, we get 3 minimal cut sets for ErroneousOutput:



Top Event (Effect)	ErroneousOutput
Description	N/A
System Unavailability	0
System Frequency	0
Severity	0

Showing 1 to 3 of 3 Cut Sets (Sort by Order)

- Processor.processorError (1)

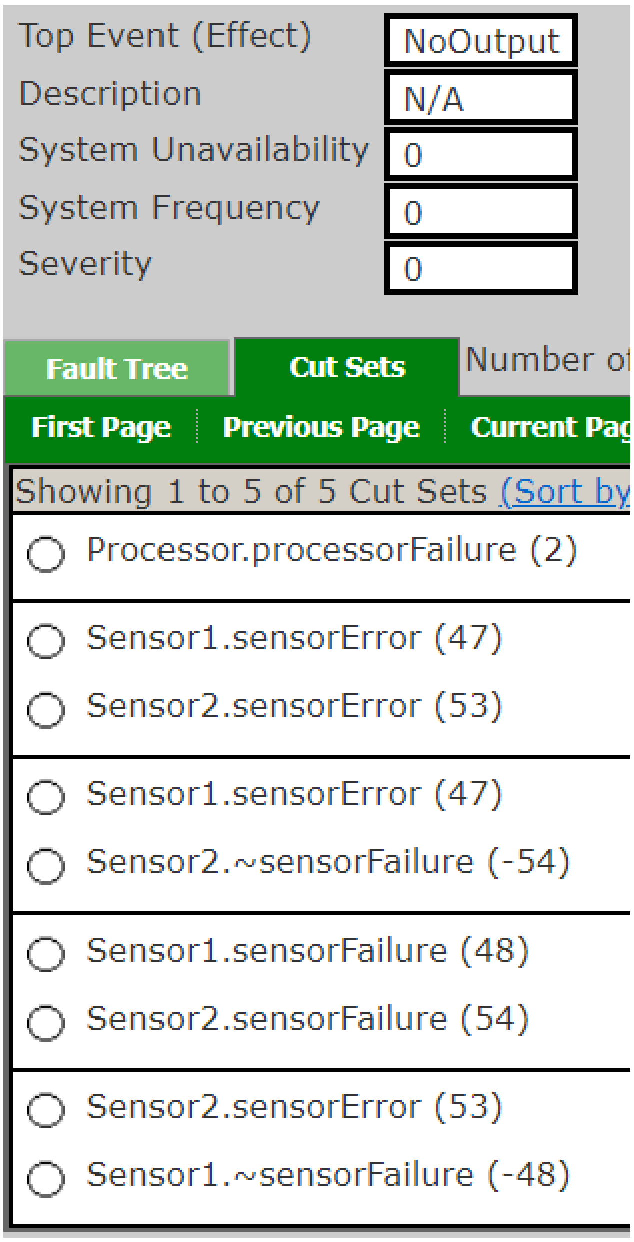
- Sensor1.sensorError (47)
- Sensor2.sensorFailure (54)

- Sensor1.sensorFailure (48)
- Sensor2.sensorError (53)

Figure 67: Results for ErroneousOutput

which fits with what we would expect, namely that a single omission (caused by sensorFailure) and a single value failure (caused by sensorError) will lead to the processor being unable to compare inputs and being forced to trust the remaining input. Processor error is also a single point of failure for this hazard.

For NoOutput, we get 5 cut sets:



Top Event (Effect)	NoOutput
Description	N/A
System Unavailability	0
System Frequency	0
Severity	0

```
Showing 1 to 5 of 5 Cut Sets (Sort by

○ Processor.processorFailure (2)

○ Sensor1.sensorError (47)
○ Sensor2.sensorError (53)

○ Sensor1.sensorError (47)
○ Sensor2.~sensorFailure (-54)

○ Sensor1.sensorFailure (48)
○ Sensor2.sensorFailure (54)

○ Sensor2.sensorError (53)
○ Sensor1.~sensorFailure (-48)
```

Figure 68: And for NoOutput

Here the results are more complex because we also have *complement events*, i.e., the negation of a failure mode. Again, processor failure is a single point of failure (if it stops operating, there is no output). The remaining four cases are as follows:

- Value failures from both sensors means both inputs are different and the processor cannot trust either, so it outputs nothing.
- A value failure from sensor 1 and normal (non-deviated) input from sensor 2 also means both inputs are different; the processor does not know which is correct, so it outputs nothing.
- An omission from both sensors due to simultaneous sensor failure means the processor has no input to work with.

- Again, a single value failure and normal (non-deviated) input means the processor cannot distinguish correct from incorrect and it outputs nothing.

The presence of NOT gates and complement events complicates the analysis because it results in a *non-coherent fault tree*, where both the presence and absence of basic events must be considered. As explained previously, this manifests in the Consensus Law:

- (X . Y) + (~X . Z) = (X . Y) + (~X . Z) + (Y . Z)

Although this example is relatively simple, meaning the analysis is very fast, in general the application of the Consensus Law slows down the HiP-HOPS analysis dramatically because it must generate new cut sets to test for redundancy. For example, consider a case where we initially have three cut sets like so:

- X . Y
- ~X . Z
- Y . W . Z

At first glance, there is no redundancy and so we might be tempted to assume these are all minimal cut sets. However, application of the Consensus Law reveals the fourth implicit cut set:

- X . Y
- ~X . Z
- Y . W . Z
- → Y . Z

which in turn means the third cut set is redundant. The newly generated cut set is also redundant (since it is implicit in the first two) and so our final minimal cut sets are just:

- **X . Y**
- **~X . Z**
- ~~Y . W . Z~~
- ~~→ Y . Z~~

The difficulty comes from the fact that implicit cut sets generated by the Consensus Law may themselves generate *more* cut sets using the Consensus Law, e.g.:

- X . Y
- ~X . Z
- ~Y . W
- → W . X   (From X.Y + ~Y.W)
- → Y . Z   (From X.Y + ~X.Z)
- → → W . Z   (From Y.Z + ~Y.W or ~X.Z + W.X)

Each of the resultant cut sets must then be checked against every other cut set for potential redundancy, which is slow.

Consequently, it is recommended to be sparing with NOT gates and complement events and use them only where necessary to avoid the significant performance penalty of non-coherent fault tree analysis.

6.5. Tutorial 5: SIL Decomposition

For the fifth tutorial, we will take a more substantial pre-existing model and look at how it can be used to perform automatic SIL decomposition. This model is a simplified version of the one described on the HiP-HOPS website at https://hip-hops.co.uk/page/how_it_works/.

6.5.1. The system model

The system model (`tutorial5_silDecomposition.slx`) is for a hybrid braking system (HBS) for an electric vehicle, which combines both a frictional electromechanical brake (EMB) and an electric in-wheel motor (IWM). The model is shown below.

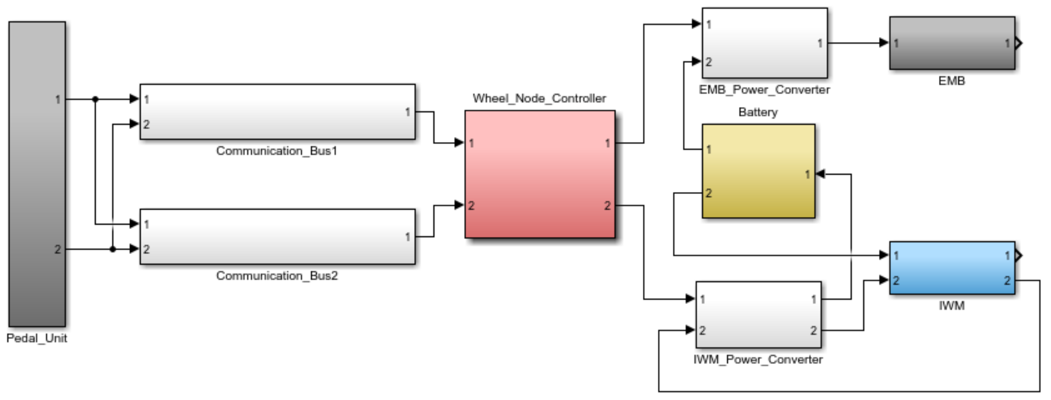


Figure 69: Tutorial 5 model — the hybrid braking system

This system is a brake-by-wire system, meaning there is no direct hydraulic link between the brake pedal and the wheel brake. Instead, a redundant system of two electronic buses and a controller is used to pass instructions from the pedal unit to the two brake controllers. Because the in-wheel motors work as generators during braking, converting kinetic energy into electrical energy, the brakes can also be used to help recharge the battery and increase the range of the vehicle.

Although only one wheel braking subsystem is used in the model, in the full version, all four wheels and their braking units would be represented. In the full system, the driver's action on the mechanical pedal is sensed and processed at an electronic pedal unit. The latter acts as a central control unit that coordinates the four local wheel controllers according to the driver's braking intentions. All the wheel node braking units are interconnected through a digital communication network with two parallel buses. After receiving the braking force demand for the wheel it is responsible for, every local wheel controller calculates the amount of torque to be developed by each actuator and then controls them accordingly through power electronics. When braking is taking place, power flows from the low voltage battery (auxiliary battery) to the EMB and from the IWM to the high voltage battery (powertrain battery). The haptic feedback that should be provided to the driver according the braking action is neglected in this model.

6.5.2. Failure data annotations

The failure behaviour of the HBS architecture is described next.

At the model level, for the purposes of this example only two hazards are considered:

- No braking after command (H1)
- Wrong value braking (H2)

An omission of braking hazard (H1) occurs when both the in-wheel motor and the electromechanical brake fail to produce any braking force. A value braking hazard (H2) occurs when either of the two braking devices brakes with an incorrect value (e.g. too high or too low). This results in the following failure expressions:

Hazard	Causes
H1	Omission-EMB.out1 AND Omission-IWM.out1
H2	Value-EMB.out1 OR Value-IWM.out1

Next, we look at the local failure annotations for each component. For the electronic pedal, there are two outputs (each connected to a different bus) and each one may suffer from either an omission failure or a value failure. We do not refine the causes of these in depth at this stage, assuming that the detailed knowledge of the unit's implementation is not available (e.g. because it is still early in the design process). Instead, we model this as four basic events, each causing one output deviation.

Output Deviation	Causes
Omission-Out1	OFailure1
Omission-Out2	OFailure2
Value-Out1	VFailure1
Value-Out2	VFailure2

The communication buses are considered to have only internal failures of the omission type due to the use of the TTP/C protocol: in the presence of other types of faults, such as a value failure caused by EMI interference, a Cyclic Redundancy Check (CRC) mechanism is responsible for discarding the message, ensuring fail-silent behaviour. Further, omission of the output of each bus can also be caused by the omission of its two inputs.

Failures may also be propagated to the bus as input as well as originating internally. In such cases, the bus would typically propagate the failure from its output. Here, however, two communications controllers are used, which provides a measure of redundancy and makes more checking possible. Therefore the buses' outputs will only suffer a value deviation if the output of the leading controller is deviated or if the leading controller outputs nothing and the replica controller outputs a deviated signal. The leading controller in this case is the element whose information is considered first (and thus takes priority). Thus for each bus we have the following expressions:

Output Deviation	Causes
Omission-Out1	OFailure1 OR (Omission-In1 AND Omission-In2)
Value-Out1	Value-In1 OR (Omission-In1 AND Value-In2)

Both types of failure are also possible deviations of the wheel node controller outputs. As mentioned, we do not investigate the internal faults in depth and define one basic event for each output deviation. Furthermore, wrong value outputs can also be triggered by value deviation transmitted by the leading bus or by the omission of that same output and value deviation of the output of the replica bus. Omission of both outputs can also be caused by omission of both buses' outputs.

Output Deviation	Causes
Omission-Out1	OFailure1 or (Omission-In1 and Omission-In2)
Omission-Out2	OFailure2 or (Omission-In1 and Omission-In2)
Value-Out1	VFailure1 or Value-In1 or (Omission-In1 and Value-In2)
Value-Out2	VFailure2 or Value-In1 or (Omission-In1 and Value-In2)

The battery is relatively simple. Here we do not consider value failures at all, only an omission of power, which is caused solely by a fault of the battery itself.

For the EMB power converter, deviations of either type are propagated from either input or caused by corresponding internal failure modes. However, since there are no value failures from the battery power connection, this input is irrelevant for value deviations.

Output Deviation	Causes
Omission-Out1	OFailure1 or Omission-In1 or Omission-In2
Value-Out1	VFailure1 or Value-In1

The IWM power converter is somewhat more complicated because it has both a regenerative braking power connection to the battery for recharging (Out1) and the output to the IWM itself (Out2). In both cases, there are internal failure modes that can cause deviations. Omission of power to the battery (Out1) can be caused by the omission failure mode, omission of input signal, or omission of regenerative power from the IWM; omission of signal to the IWM is caused only by omission of input signal or internal failure.

For value deviations, the situation is similar. Value failures at either input are propagated from Out1, but only a deviated input signal from the wheel controller is propagated to Out2.

Output Deviation	Causes
Omission-Out1	Omission-In1 or OFailure1 or Omission-In2
Omission-Out2	Omission-In1 or OFailure1
Value-Out1	Value-In1 or Value-In2 or VFailure1
Value-Out2	Value-In1 or VFailure1

The two braking units are straightforward. The EMB propagates any input deviation to its output and may also suffer from internal failure modes. The IWM likewise propagates any input deviation to its outputs, with the exception that value failures are not propagated from the regenerative power connection to the IWM power converter.

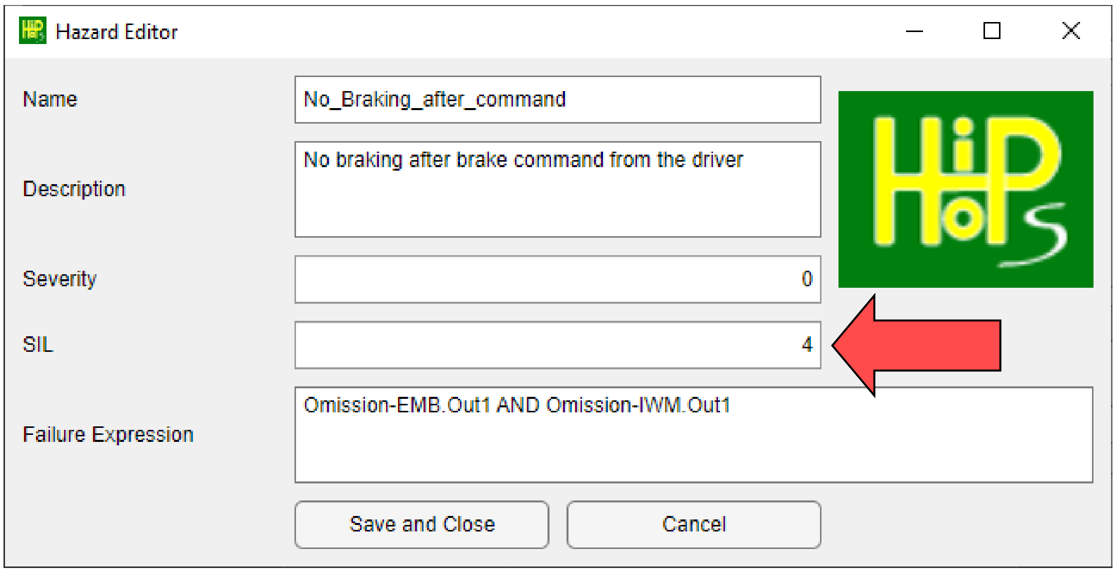
6.5.3. SIL Decomposition

Setting up the model for SIL decomposition is surprisingly easy. The first step is simply to assign appropriate SILs to each hazard. The type of SIL and its corresponding semantics will depend on the domain; in this case, as an electric car, ASILs (Automotive Safety Integrity Levels) would be used. For aircraft, DALs would be used instead. For other systems, standard SILs or some other measure may be used.

For ASILs, there are 4 levels of stringency, ranging from A (least strict) to D (most strict), with an additional level of QM meaning "Quality Management only", i.e., no special safety requirement applies. We can translate this into integers for HiP-HOPS quite simply:

ASIL	QM	A	B	C	D
Value	0	1	2	3	4

Here we assume that an omission of braking (Hazard H1) is much more severe and assign it ASIL D (i.e., a value of 4).



Name	No_Braking_after_command
Description	No braking after brake command from the driver
Severity	0
SIL	4
Failure Expression	Omission-EMB.Out1 AND Omission-IWM.Out1

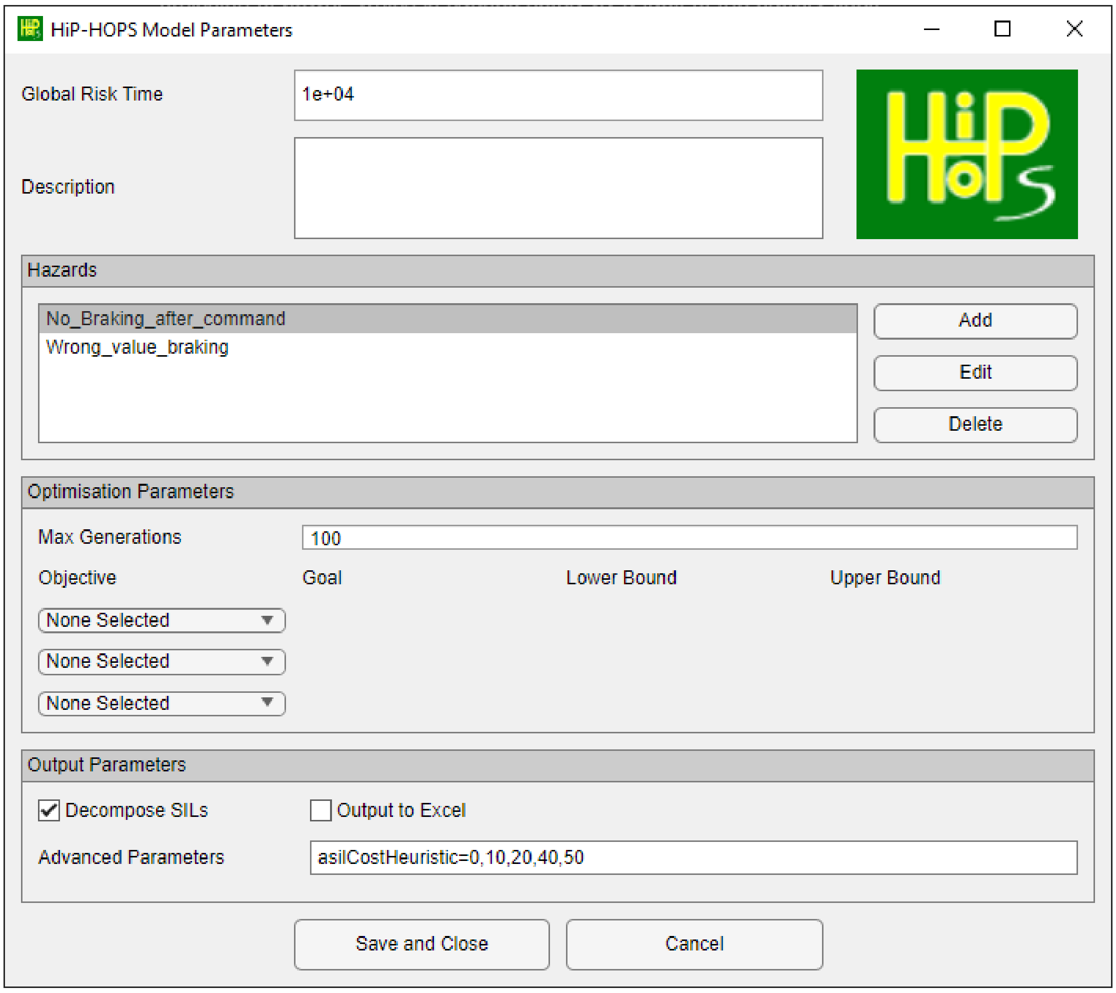
Figure 70: Assigning an ASIL to a hazard

For hazard H2, we assume that a deviation in braking value is less severe than no braking at all, and assign ASIL A (i.e., a value of 1). Naturally, for a real system, a full ISO 26262 compliant risk assessment would be undertaken to derive appropriate ASIL values for each hazard.

The last step is to provide parameters for the decomposition process, namely a cost heuristic. HiP-HOPS uses a default cost heuristic that increases by 10 for each level of severity (i.e., SIL value * 10). However, for ASILs, it is more common that there is a cost jump between ASIL B and ASIL C, so we add the following heuristic using the "asilCostHeuristic" parameter:

ASIL	QM	A	B	C	D
Cost	0	10	20	40	50

We also need to tick the 'Decompose SILs' box to instruct HiP-HOPS to perform decomposition.


HiP-HOPS Model Parameters

Global Risk Time: 1e+04

Description:

Hazards
- No_Braking_after_command
- Wrong_value_braking

Optimisation Parameters

Max Generations: 100

Objective	Goal	Lower Bound	Upper Bound
None Selected			
None Selected			
None Selected			

Output Parameters

[✓] Decompose SILs [ ] Output to Excel

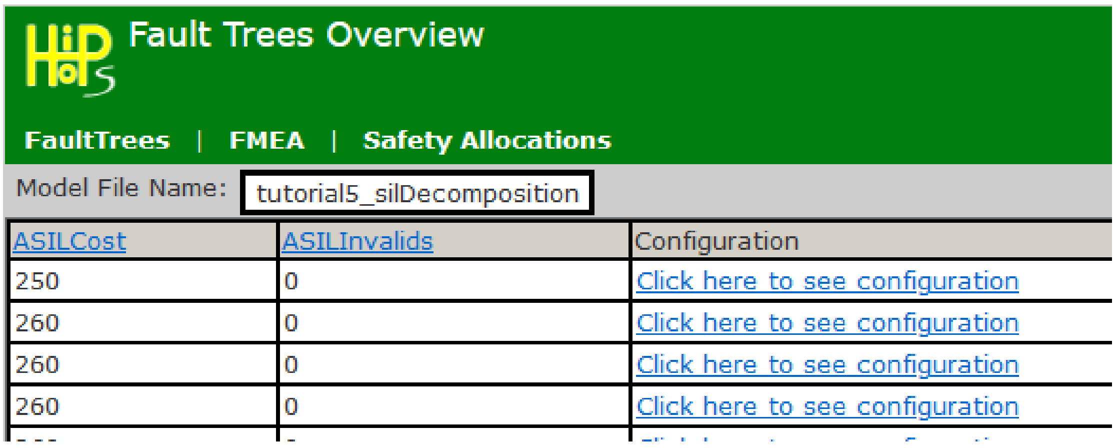
Advanced Parameters: asilCostHeuristic=0,10,20,40,50

Figure 71: Model parameters for ASIL decomposition

6.5.4. Results

This time, when we press 'Analyse model', in addition to the normal analysis results we also get the results of the SIL decomposition optimisation process. This can be seen in two places: in the Matlab command window, where the ASIL (or DAL) decomposition process will be shown, and also in the browser results (SIL decomposition is not available in spreadsheet form at this time).

The SIL decomposition results are visible by selecting "Safety Allocations" from the top of the HiP-HOPS output. This displays a list of all allocations found, along with their total cost and a link to view the individual allocation.


Fault Trees Overview

FaultTrees | FMEA | Safety Allocations

Model File Name: tutorial5_silDecomposition

ASILCost	ASILInvalids	Configuration
250	0	Click here to see configuration
260	0	Click here to see configuration
260	0	Click here to see configuration
260	0	Click here to see configuration

Figure 72: Safety Allocations

HiP-HOPS will attempt to allocate an appropriate SIL to each basic event that causes each hazard. Thus if there is only a single cause of a hazard, that basic event receives the full SIL value assigned to that hazard. If there are multiple causes linked in an independent AND configuration, then a range of values becomes possible depending on the semantics of the SIL, e.g. for ASILs, the ASILs of the constituent causes simply need to add up to the hazard ASIL:

Cause 1	Cause 2	Total (= Hazard)
QM (0)	ASIL D(4)	0 + 4 = 4
ASIL A (1)	ASIL C (3)	1 + 3 = 4
ASIL B (2)	ASIL B (2)	2 + 2 = 4
ASIL C (3)	ASIL A (1)	3 + 1 = 4
ASIL D (4)	QM (0)	4 + 0 = 4

The situation is complicated where a given basic event contributes to more than one hazard. Decomposition may reveal that the event receives e.g. ASIL C from one hazard but ASIL A from another; in such a case, the highest value is used, but may mean that this particular configuration is strictly worse than some other configuration. For example, consider a scenario with two events, E1 and E2, which contribute to two hazards, H1 (ASIL D) and H2 (ASIL B) like so:

- H1 is caused by E1 AND E2
- H2 is caused only by E2

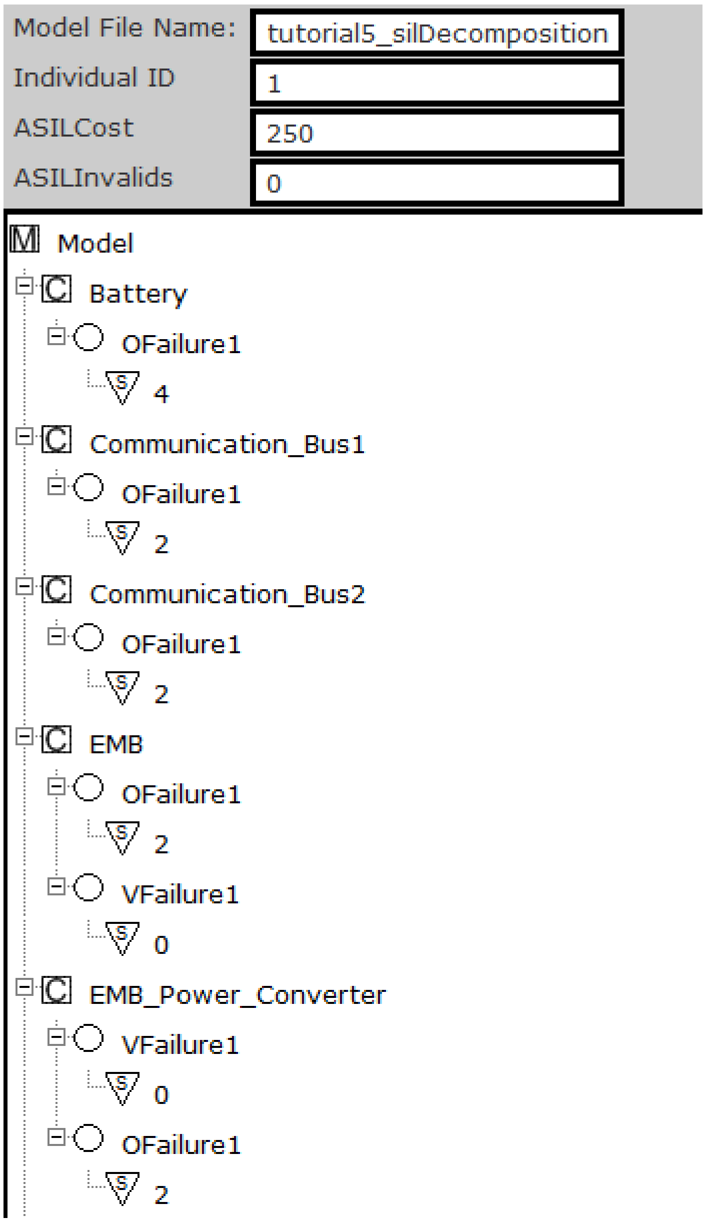
From the table above, we can see that there are five possible allocations for two events causing a hazard with ASIL D, each adding up to 4 in total. However, because E2 is the sole cause of H2, it can never be allocated an ASIL less than that of H2, i.e., E2 must have a minimum ASIL of B (i.e., 2). This yields the following:

E1	E2	Total (= Hazard)
QM (0)	ASIL D(4)	0 + 4 = 4
ASIL A (1)	ASIL C (3)	1 + 3 = 4
ASIL B (2)	ASIL B (2)	2 + 2 = 4
ASIL C (3)	ASIL B (2)	3 + 2 = 5
ASIL D (4)	ASIL B (2)	4 + 2 = 6

Here we can see that the last two possible configurations have a higher cost than the others for no benefit. The more combinations of causes there are in the results, the more potential configurations there are, and thus an exhaustive generation of all possible allocations is not always possible.

Instead, optimisation using Tabu search is used to identify optimal or near optimal configurations with low costs. Each configuration is evaluated according to total cost (the sum of all ASIL cost values of all basic events leading to the hazards) using the cost heuristic defined.

Going back to the model results, we can open up the configurations to see which (A)SILs have been assigned to each basic event:



Model File Name:	tutorial5_silDecomposition
Individual ID	1
ASILCost	250
ASILInvalids	0

```
Model
  Battery
    OFailure1
      4
  Communication_Bus1
    OFailure1
      2
  Communication_Bus2
    OFailure1
      2
  EMB
    OFailure1
      2
    VFailure1
      0
  EMB_Power_Converter
    VFailure1
      0
    OFailure1
      2
```

Figure 73: Allocations to each event

The total cost is the sum of the cost heuristic applied to each of these allocations. These configurations then serve as potential allocations that should be evaluated separately to ensure that all safety requirements are properly met, since some constraints may not be captured by the model or by HiP-HOPS.

6.6. Tutorial 6: Architectural Optimisation

For the final tutorial, we will look at architectural optimisation. As with the last tutorial, we will start with a more substantial pre-made model — in this case, a fuel oil service system for a cargo ship (`tutorial6_optimisation.slx`). By defining alternative implementations for the various components of the system, we generate a design space of possible architectural configurations that we can then explore via an optimisation process.

6.6.1. <u>The system model</u>

Figure 79 shows the fuel oil service system. The goal is to ensure fuel flows to the main engine; lack of fuel results in loss of engine propulsion (`OmissionEnergy-mainEngine.mech`), which is a serious issue that can lead to the ship becoming grounded as a result of unpowered drifting.

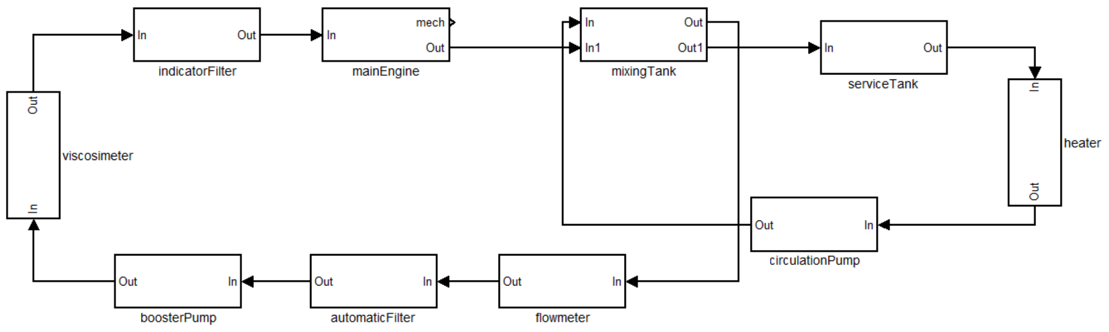


Figure 74: Fuel oil service system for a cargo ship.

The following tables contain the implementation failure data for the components in the example. The main engine provides the system output (i.e., propulsion) and thus also the system-level hazard (`NoEnginePower`, an omission of energy at the mechanical output of the engine) which is caused by an omission of flow of oil at the input. It has no internal failures.

mainEngine		
Output Deviation	**Description**	**Failure logic (Propagation)**
OmissionEnergy-mech (System output)	Omission of energy at the mechanical output of the mainEngine caused by an omission of flow of oil at the input	OmissionFlow-In

The other components in the system (indicator filter, viscosimeter, pre-heater, circulation pump, mixing tank, flow meter, automatic filter, booster pump, and service tank) each contain 3 alternative subsystems that define different levels of parallel redundancy (shown in the 3 figures below). In each case the propagation of the omission of flow of oil is combined in the 'AND' block so that both (or all three) redundant components must fail to cause failure of the subsystem.

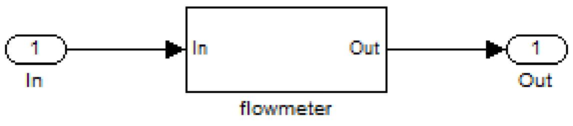


Figure 75: Alternative 1: no redundancy

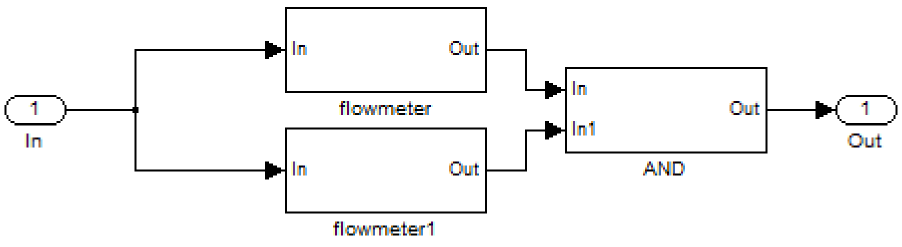


Figure 76: Alternative 2: one parallel redundancy

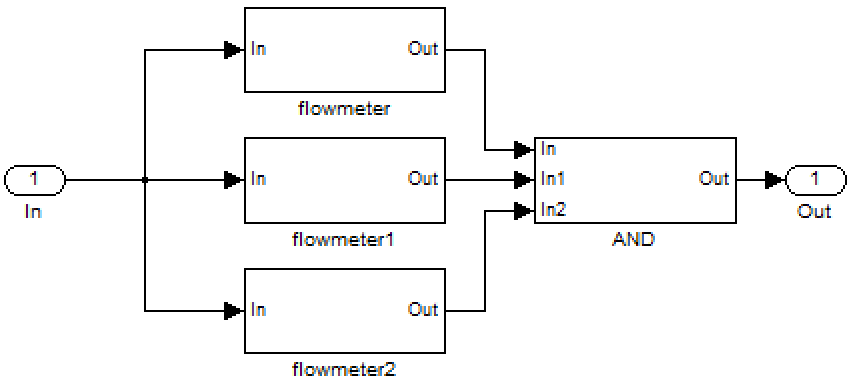
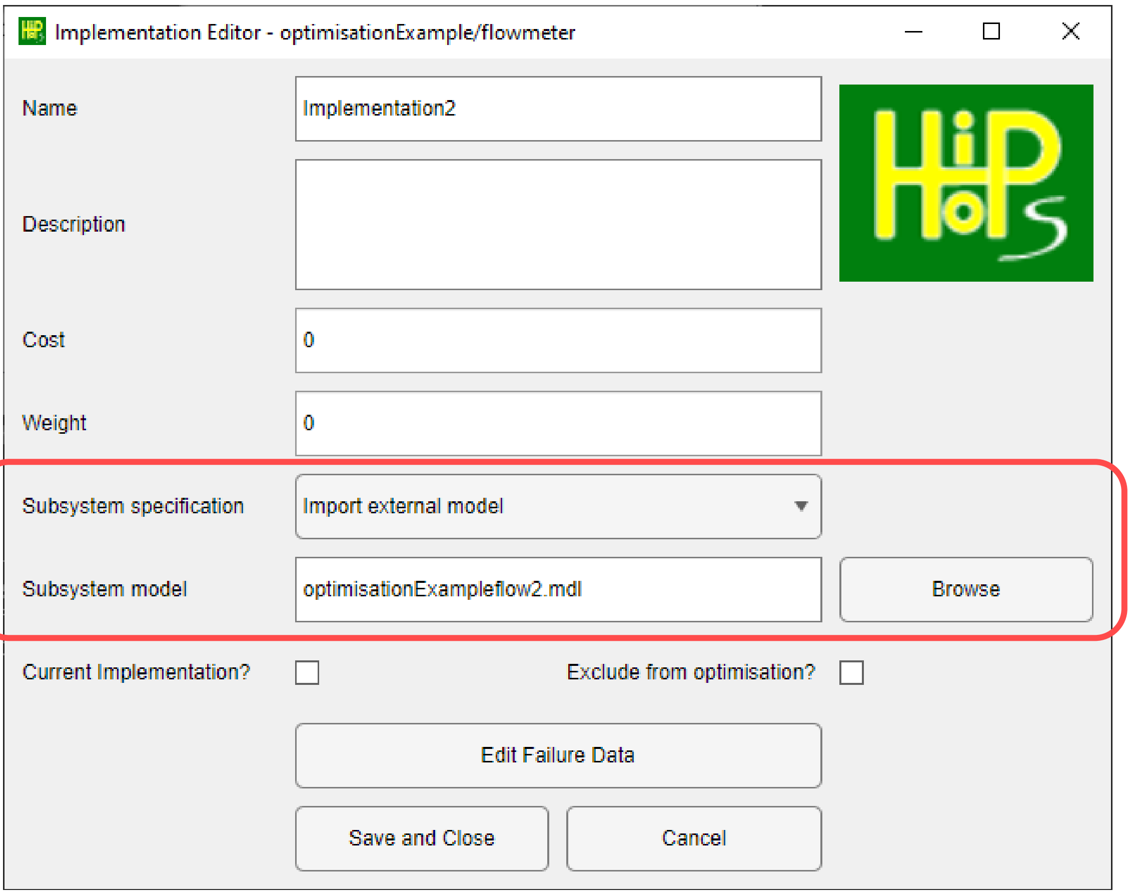


Figure 77: Alternative 3: two parallel redundancies

Other than the default single component subsystem, all of these alternative subsystems are modelled as separate files and referenced from within the Implementation Editor:


Implementation Editor - optimisationExample/flowmeter

Name: Implementation2

Description:

Cost: 0

Weight: 0

Subsystem specification: Import external model

Subsystem model: optimisationExampleflow2.mdl

Current Implementation? ☐ Exclude from optimisation? ☐

Figure 78: Setting an alternative subsystem model in the Implementation Editor

For this potential substitution to be valid, the subsystem model must have top-level ports that match (both in number, direction, and name) the ports in the parent component. For the flowmeter, there is a single input (In) and a single output (Out), so there needs to be a corresponding In and Out in the subsystem.

All of the subcomponents have the same failure propagation logic. The omission of flow of oil at the output is caused by the omission of flow of oil at the input, or an internal failure of the component.

All other system subcomponents		
Output Deviation	**Description**	**Failure logic (Propagation)**
OmissionFlow-Out	Omission of flow of oil at the output can be caused either by an omission of flow of oil at the input or an internal failure mode of the component	OmissionFlow-In or [component]Failure

Each of the subcomponents has 3 alternative implementations with different costs and the internal failure modes have different failure rates.

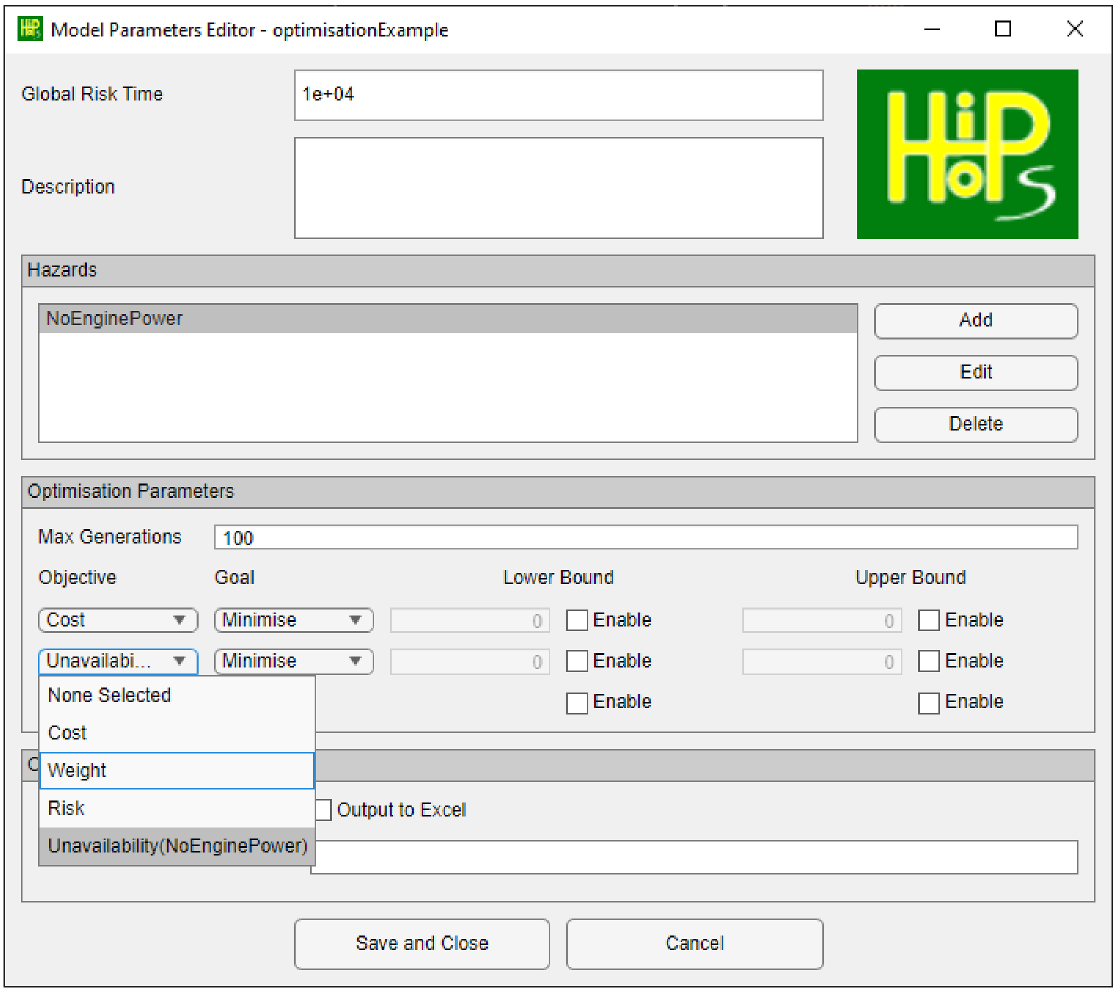
	Alternative 1		**Alternative 2**		**Alternative 3**	
Components	**Cost**	**Failure Rate**	**Cost**	**Failure Rate**	**Cost**	**Failure Rate**
Indicator filter	1500	5.0E-7	2500	2.0E-7	3222	1.0E-7
Viscosimeter	2500	2.5E-6	3178	1.0E-6	3814	5.0E-7
Pre-heater	2000	6.7E-6	2505	5.0E-6	3956	1.0E-6
Circulation pump	6000	3.2E-5	13380	2.0E-5	18000	7.0E-6
Mixing tank	2000	1.6E-5	2963	8.0E-6	4444	2.0E-6
Flow meter	2000	1.0E-5	3000	1.0E-6	4444	5.0E-7
Automatic filter	2000	1.0E-5	2647	5.0E-6	3529	1.0E-6
Booster pump	5000	3.2E-5	10682	2.0E-5	12500	5.0E-6
Service tank	1500	1.6E-5	1957	5.0E-6	2739	1.0E-6

This means that for a single subcomponent subsystem there are three possible configurations, for a dual subsystem there are nine (3 x 3), and for a triple subsystem there are 27 (3 x 3 x 3), for a total of 39 possible configurations for every top-level component (aside from the mainEngine).

The cost is set in the same Implementation Editor interface; as mentioned previously, no units are defined so care must be taken to ensure that the numbers have a consistent meaning across the model. A value for weight can also be added for situation where this is particularly important, e.g. in aircraft design.

6.6.2. Configuring optimisation options

Once all of the model failure data and implementation options have been defined, the next step is to set up the optimisation process itself.


Model Parameters Editor - optimisationExample

Global Risk Time: 1e+04

Description:

Hazards: NoEnginePower

Optimisation Parameters

Max Generations: 100

Objective	Goal	Lower Bound		Upper Bound	
Cost	Minimise	0	Enable	0	Enable
Unavailabi...	Minimise	0	Enable	0	Enable
			Enable		Enable

Figure 79: Optimisation parameters

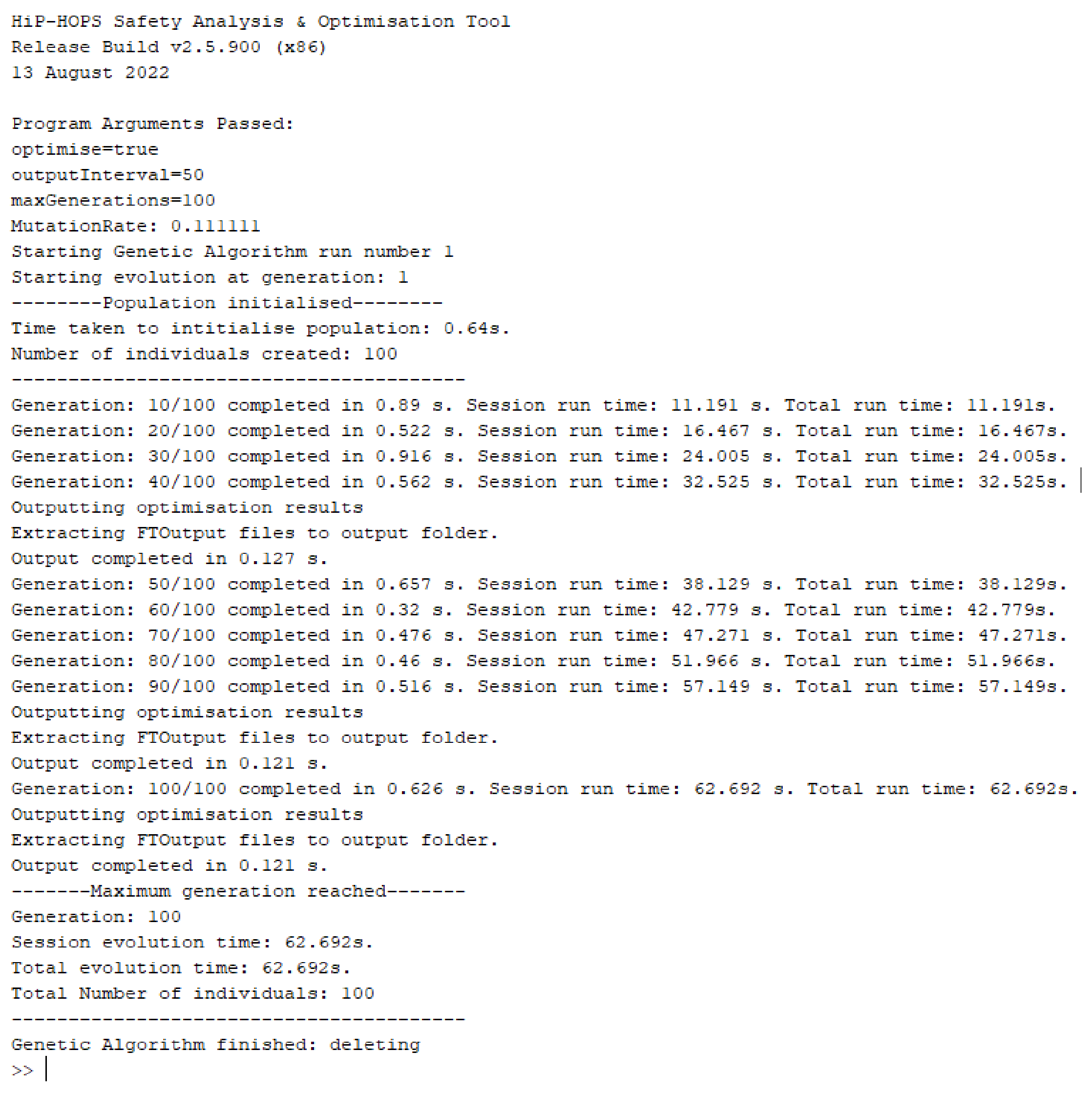
All of these configuration parameters can be found in the Model Parameters Editor under the optimisation parameters section.

The first option is to set the number of generations the underlying genetic algorithm will run for. Each generation will evolve the solution set, so in general more generations will result in better solutions. However, given that HiP-HOPS must analyse and evaluate each possible solution, setting a high value here could result in a very long runtime. It is recommended to use a lower value first (e.g. the default value of 100) and then adjust depending on the quality of the results.

Underneath the generations box is the objectives section. Here, the objectives of the optimisation can be configured. Up to three objectives can be used. Cost and weight optimise according to those two properties. Additionally, one may optimise the unavailability for a specific hazard/fault tree (like `NoEnginePower` in the example above), or select "Risk" to minimise the overall risk (i.e., the sum of unavailability * severity for each hazard) across the whole system.

Each objective can be set to either minimise or maximise (minimise is usually recommended!) and both lower and upper bounds can be optionally used to constrain the solution set, e.g. if the total cost must be below a given value. Note however that these bounds are used as guidelines for the optimisation: HiP-HOPS will attempt to favour solutions that fall within the bounds but cannot guarantee that all final solutions will be within them.

The final step is to launch the optimisation using the 'Optimise model' button on the Launcher. Status updates will begin to be written to the Matlab command window:



```
HiP-HOPS Safety Analysis & Optimisation Tool
Release Build v2.5.900 (x86)
13 August 2022

Program Arguments Passed:
optimise=true
outputInterval=50
maxGenerations=100
MutationRate: 0.111111
Starting Genetic Algorithm run number 1
Starting evolution at generation: 1
--------Population initialised--------
Time taken to intitialise population: 0.64s.
Number of individuals created: 100
----------------------------------------
Generation: 10/100 completed in 0.89 s. Session run time: 11.191 s. Total run time: 11.191s.
Generation: 20/100 completed in 0.522 s. Session run time: 16.467 s. Total run time: 16.467s.
Generation: 30/100 completed in 0.916 s. Session run time: 24.005 s. Total run time: 24.005s.
Generation: 40/100 completed in 0.562 s. Session run time: 32.525 s. Total run time: 32.525s.
Outputting optimisation results
Extracting FTOutput files to output folder.
Output completed in 0.127 s.
Generation: 50/100 completed in 0.657 s. Session run time: 38.129 s. Total run time: 38.129s.
Generation: 60/100 completed in 0.32 s. Session run time: 42.779 s. Total run time: 42.779s.
Generation: 70/100 completed in 0.476 s. Session run time: 47.271 s. Total run time: 47.271s.
Generation: 80/100 completed in 0.46 s. Session run time: 51.966 s. Total run time: 51.966s.
Generation: 90/100 completed in 0.516 s. Session run time: 57.149 s. Total run time: 57.149s.
Outputting optimisation results
Extracting FTOutput files to output folder.
Output completed in 0.121 s.
Generation: 100/100 completed in 0.626 s. Session run time: 62.692 s. Total run time: 62.692s.
Outputting optimisation results
Extracting FTOutput files to output folder.
Output completed in 0.121 s.
-------Maximum generation reached-------
Generation: 100
Session evolution time: 62.692s.
Total evolution time: 62.692s.
Total Number of individuals: 100
----------------------------------------
Genetic Algorithm finished: deleting
>>
```

Figure 80: Optimisation progress

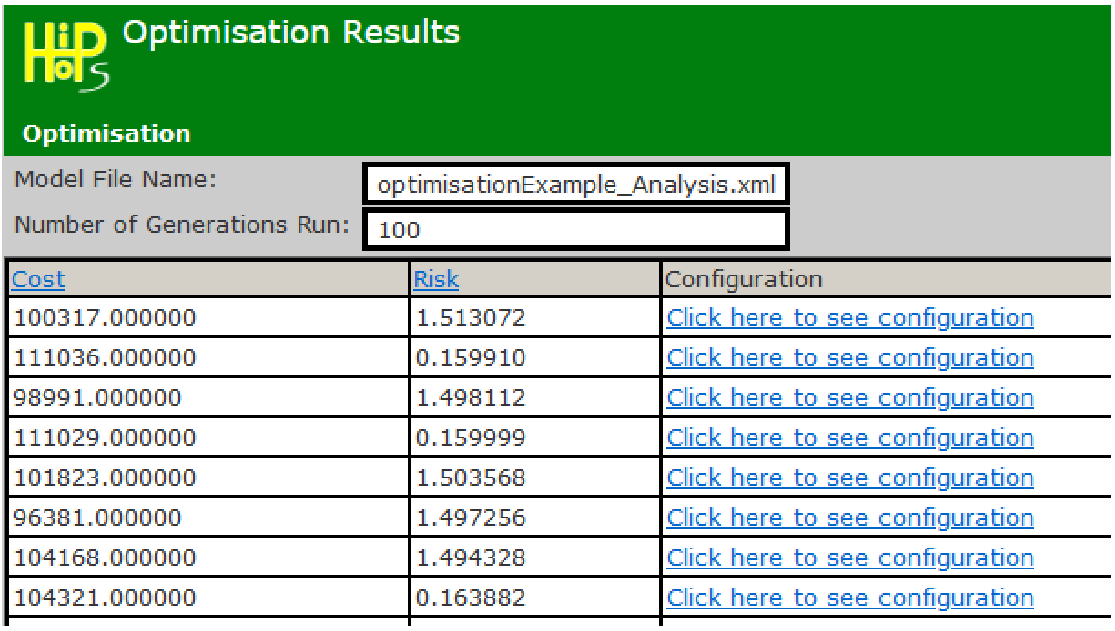
Note that the results are not opened until the process completes, which may take some time, though intermediate results are output at regular intervals in case of interruptions (usually every 50 generations).

Unlike normal analysis output, optimisation results are output to the <modelname>-OptimisationResults directory.

6.6.3. Viewing the optimisation results

Once complete, the results should open automatically. You will be presented with a list of all configurations found, along with their scores for each of the objectives set (e.g. total cost and unavailability in this case). Because this was a multi-objective optimisation, the results all represent Pareto-optimal trade-offs, i.e. while one solution might be worse in one objective, it should be better in another, and so there should never be a solution in the results which is strictly worse in all objectives than another.

As usual, you can sort the results by clicking on the appropriate headings.



Optimisation Results

Optimisation

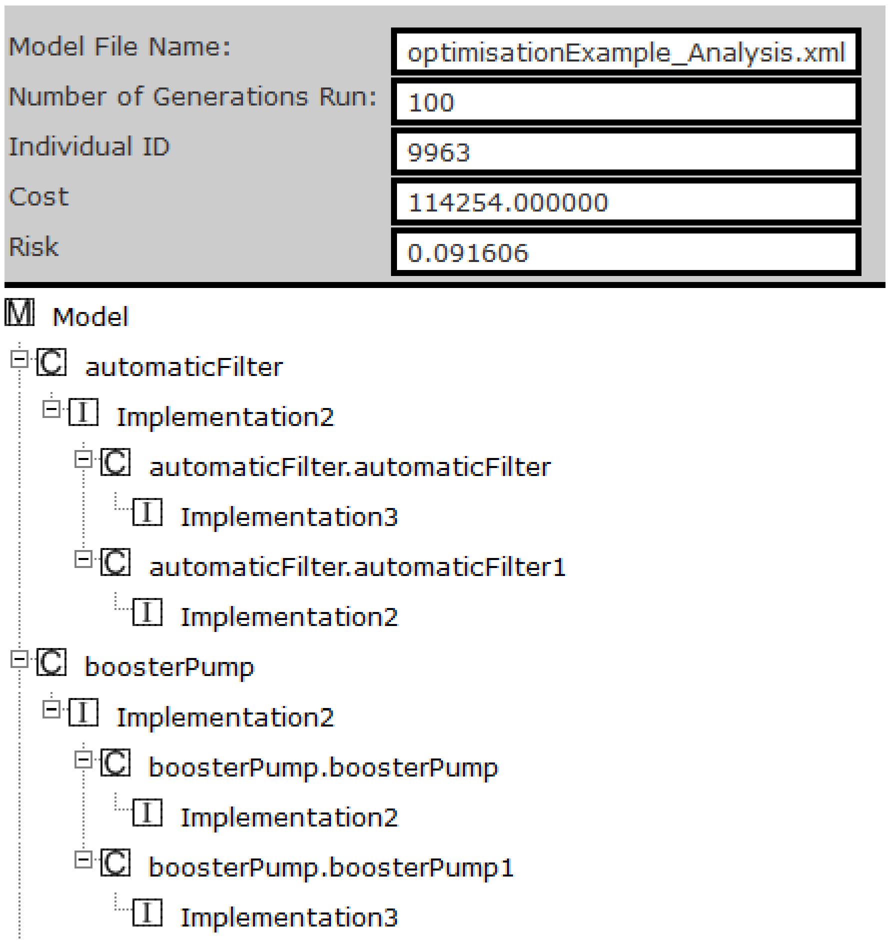
Model File Name:	optimisationExample_Analysis.xml
Number of Generations Run:	100

Cost	Risk	Configuration
100317.000000	1.513072	Click here to see configuration
111036.000000	0.159910	Click here to see configuration
98991.000000	1.498112	Click here to see configuration
111029.000000	0.159999	Click here to see configuration
101823.000000	1.503568	Click here to see configuration
96381.000000	1.497256	Click here to see configuration
104168.000000	1.494328	Click here to see configuration
104321.000000	0.163882	Click here to see configuration

Figure 81: Optimisation output

Clicking on the links in the last column will display the corresponding model configuration in a tree format. Each configurable component in the model is listed along with the implementations used. In our example here, for example, we would expect the more reliable solutions to be those which used multiple redundancy (i.e., subsystem implementations with 2 or 3 subcomponents in parallel) and high quality implementations, which would also be the most expensive; vice versa, the cheapest solutions would be those with mostly individual components, which would be least reliable.

Clicking on one of the more reliable configurations reveals this to be the case:



Model File Name:	optimisationExample_Analysis.xml
Number of Generations Run:	100
Individual ID	9963
Cost	114254.000000
Risk	0.091606

```
Model
  automaticFilter
    Implementation2
      automaticFilter.automaticFilter
        Implementation3
      automaticFilter.automaticFilter1
        Implementation2
  boosterPump
    Implementation2
      boosterPump.boosterPump
        Implementation2
      boosterPump.boosterPump1
        Implementation3
```

Figure 82: One of the optimisation configurations

The information obtained by the optimisation is meant to be a guide for continued architectural development rather than a complete solution in itself. It is intended to help rapidly evaluate large potential design spaces to narrow down the directions for future design work, e.g. by showing which areas of the system would benefit most from being replicated or being replaced with different implementations.

7. Running HiP-HOPS as a standalone engine

The HiP-HOPS tool itself is separate from the Matlab Simulink interface and can be executed independently. The necessary files are all inside the HiP-HOPS_FailureEditor subdirectory of your HiP-HOPS install and are:

• hipop.exe	The primary HiP-HOPS executable
• dal.exe	The DAL allocation program (launched by hipop.exe)
• asildecomp.exe	The ASIL allocation program (launched by hipop.exe)
• FTOutput.zip	Contains the files necessary to display the output

As input, HiP-HOPS (i.e., hipop.exe) requires a model file in XML format. The path to this file is always the first program argument (to avoid errors, ensure the path does not contain spaces). The HiP-HOPS Matlab Simulink interface automatically transforms and exports this file when you press 'Analyse model' in the Launcher, but the file can be generated by other tools or even by hand if required. In addition, various arguments may be provided as further instructions for HiP-HOPS (see section 7.1).

Just as HiP-HOPS is independent of any particular modelling tool, its output can also be viewed or processed independently. By default, HiP-HOPS outputs a series of XML files (using the outputtype=HTML or XML options); these files can also be consolidated into a single file using (using the outputtype=RESULTS option) for ease of post-processing in other tools. The dal.exe and asildecomp.exe programs load these XML files in just such a manner.

Further documentation regarding the input and output XML formats is available upon request.

7.1. Command Line Parameters

The tables below describe the currently supported command line parameters in HiP-HOPS. There are parameters that configure standard analysis, parameters for architectural optimisation, and parameters for safety requirement (SIL) allocation and decomposition.

In addition to controlling HiP-HOPS when launched as a standalone executable, these parameters can also be used in the 'Advanced Parameters' field of the Model Parameters window in Simulink, as described in section 4.10.

Most parameters are of the form `<parameter>=<value>`. Possible values are also included in the table.

Table 5: Basic analysis parameters

Parameter	Values	Effect
analyse	true/false	If false, HiP-HOPS will generate fault trees but will skip its fault tree analysis phase. Depending on other options set, this will output the fault trees but not the cut sets or FMEA. Defaults to **true**.
circles	true/false	Determines whether HiP-HOPS will remove cut sets with circle nodes where it has detected circular logic or whether it will omit these cut sets entirely. A circle node represents a contradiction in the logic and thus any cut set containing one can never be true. However, it can be useful to know that circular logic exists. As such, it defaults to **false**.
contract	true/false	As part of its normal fault tree analysis process, HiP-HOPS will contract the fault trees by removing redundant nodes. This results in better performance as well as more compact and readable fault trees, but can be disabled if so desired. Defaults to **true**.
countNormalEvents	true/false	When counting the number of basic events in a cut set to determine order, this parameter will include or exclude normal events in the count. Defaults to **true**.
decomposeDALs	none	Enables DAL decomposition mode. See SIL decomposition parameters table below for further options.
decomposeSILs	none	Enables ASIL decomposition mode. Equivalent to ticking the "Decompose SILs" box. See SIL decomposition parameters table below for further options.
displayOutput	true/false	If false, HiP-HOPS will not automatically open its results in the default system browser. Defaults to **true**.
idevns	true/false	Determines whether HiP-HOPS will remove cut sets with hanging input deviation nodes or whether it will omit these cut sets entirely. A hanging input deviation represents a contradiction in the logic and thus any cut set containing one can never be true. Although they indicate dead ends in the propagation, hanging input deviations are not necessarily indicative of errors in the

		failure logic. It defaults to **true**.
maxCutSetSize	integer	As above.
modularise	true/false	As part of its normal fault tree analysis process, HiP-HOPS will attempt to identify independent sub-modules within the fault trees and analyse these separately. This results in significant performance benefits but can be disabled if required. Defaults to **true**.
optimise	true/false	Enables architectural optimisation mode. See Optimisation parameters table for further options.
outputContractedTrees	true/false	Assuming the *contract* parameter is set to true, by default HiP-HOPS will output the contracted fault trees rather than the originally synthesised fault trees. This can be overridden by setting this parameter to false. Note that the value of *contract* overrides this: HiP-HOPS cannot output contracted trees if they are not generated in the first place. Defaults to **true**.
outputdir	directory	Tells HiP-HOPS to output to a different location. Must be a valid directory (HiP-HOPS will not create the path).
outputtype	XML, HTML, EXCEL, RESULTS	This parameter tells HiP-HOPS what form of output to generate: **HTML** = standard browser output (a mixture of XML and JS files) **XML** = XML only output **EXCEL** = generates an .xlsx file that can be opened in Microsoft Excel 2007 or better **RESULTS** = generates a single XML file to <modelname>-Results that contains all the results all together, rather than in different files (as is the case with the **XML** setting). Intended as raw input data for other tools and cannot be opened in a browser. These values can be concatenated together in a comma-separated list, e.g. "outputtype=HTML,EXCEL". By default, when executed directly HiP-HOPS will not generate **any** output files and will only output results to the console. When used with the Simulink

		interface, HTML output is used by default and the Excel output can be added with the appropriate tick box. Using the outputtype parameter in the Advanced Parameters will replace these settings.
quantanalysis	true/false	If set to false, this will tell HiP-HOPS not to perform quantitative analysis. This can be useful for achieving small performance increases when only qualitative (cut set) results are required. Defaults to **true**.
runSilent	true/false	HiP-HOPS will output a lot of information to the console (stdout) by default. Setting this option to **true** will disable this output. Defaults to **false**.
warnings	true/false	Enables/disables generation of warnings in HiP-HOPS. Defaults to **true**.

As well as fault tree analysis, HiP-HOPS is capable of additional operations, including architectural optimisation and SIL decomposition and allocation. Both operations require a specially configured model and have their own additional sets of parameters, described below.

Table 6: Optimisation parameters (NB: Only applicable in optimisation)

Parameter	Values	Effect
allowInfeasibleResults	true/false	Determines whether solutions that exceed upper and lower bounds on the optimisation objectives can be output to the results. Note the distinction between this and strictBoundaries as this allows them to exist in the population during optimisation but exclude them from the results. Defaults to **false**.
childPopulationSize	integer	How many new individual candidate solutions are created for evaluation each generation. These are then combined with the existing population before the numbers are culled to meet the populationSize limit. Defaults to **100**.
crossoverRate	float	The probability (0.0 to 1.0) that recombination will occur between two individual ‘parent’ candidate solutions when production a new candidate solution. Defaults to **0.9**.
forceRestart	true/false	Overrides the inclusion of a previousStateFileName to start without loading a previous state.

		Defaults to **false**.
includeArchive	true/false	When the size of the Pareto front is greater than the population size, then feasible candidate solutions can get lost. The archive will maintain all found feasible undominated solutions found by the algorithm. Defaults to **true**.
maxGenerations	integer	How many generations the optimisation algorithm will be run for. Note that this acts as an upper limit when used in conjunction with a previous state (see previousStateFileName) rather than being an additional number of generations. Defaults to **1000**.
mutationRate	float	The probability (0.0 to 1.0) that an encoding site will be randomly mutated. Defaults to **0.05**.
numberOfRepeatRuns	integer	How many times to repeat the optimisation. Usually 0 to 10 to account for the stochastic nature of the optimisation algorithms. For each repeat run, the complete number of generations specified by maxGenerations will be executed. Defaults to **1**.
outputInterval	integer	In order to be able to monitor the state of the optimisation, whilst minimizing the impact on performance, you can use this parameter to set a number of generations to wait before outputting the current results state of the optimisation. This should usually be some factor of maxGenerations. Set to zero if you don't want to activate this feature. Defaults to **0**.
populationSize	integer	The number of individual solution candidates that are maintained each generation. Defaults to **100**.
previousStateFileName	filepath	You may wish to further explore a population for more generations than you previously set without starting over from scratch. You can use this parameter to specify a previous results xml file which will load that as the starting population for the optimisation. Note that you will need to increase the maxGenerations parameter.
pureElitist	true/false	Only undominated solutions on the Pareto front are allowed in the

		population. Defaults to **false**.
strictBoundaries	true/false	When upper and lower bounds are specified for the optimisation objectives then setting strictBoundaries=true will cause all solutions that exceed the boundaries to be cut from the population. Otherwise they are allowed in the population, but selective pressures should tend towards their removal over the course of the optimisation. Defaults to **false**.

Table 7: SIL allocation parameters (NB: Requires that either decomposeSILs=true or decomposeDALs=true)

Parameter	Values	Effect
asilCostHeuristic	comma separated numbers	The cost associated with ASILs QM and A to D in that order. Defaults to **0,10,20,30,40**.
dalCostHeuristic	comma separated numbers	The cost associated with DALs A to E in that order. Defaults to **40,30,20,10,0**.
tabuNumOutputSolutions	integer	Maximum number of undominated decomposition alternatives to be output to the results. Defaults to **50**.
tabuPascLimit	float	Maximum value for the tabu period multiplier for ascending moves. Defaults to **0.4**.
tabuPdesLimit	float	Maximum value for the tabu period multiplier for descending moves. Defaults to **1**.
tabuRepetitionLimit	integer	Analogous to maxGenerations, how many iterations to run the tabu search optimisation for. Default to **1000**.
tabuUpdatePascFrequency	integer	Number of iterations that pass before updating the value of the tabu period for ascending moves. Defaults to **4**.
tabuUpdatePdesFrequency	integer	Number of iterations that pass before updating the value of the tabu period for descending moves. Defaults to **3**.