SESAME
SECURE AND SAFE MULTI-ROBOT SYSTEMS

Project Number 101017258

D4.4 Tools for Automated Safety Analysis of MRS and for Production of EDDIs (Initial Version)

Version 1.0
29 June 2022
Final

Public Distribution

University of Hull and Fraunhofer IESE

Project Partners: **Aero41, ATB, AVL, Bonn-Rhein-Sieg University, Cyprus Civil Defence, Domaine Kox, FORTH, Fraunhofer IESE, KIOS, KUKA Assembly & Test, Locomotec, Luxsense, The Open Group, Technology Transfer Systems, University of Hull, University of Luxembourg, University of York**

PROJECT PARTNER CONTACT INFORMATION

Aero41 Frédéric Hemmeler Chemin de Mornex 3 1003 Lausanne Switzerland E-mail: frederic.hemmeler@aero41.ch	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany E-mail: scholze@atb-bremen.de
AVL Martin Weinzerl Hans-List-Platz 1 8020 Graz Austria E-mail: martin.weinzerl@avl.com	**Bonn-Rhein-Sieg University** Nico Hochgeschwender Grantham-Allee 20 53757 Sankt Augustin Germany E-mail: nico.hochgeschwender@h-brs.de
Cyprus Civil Defence Eftychia Stokkou Cyprus Ministry of Interior 1453 Lefkosia Cyprus E-mail: estokkou@cd.moi.gov.cy	**Domaine Kox** Corinne Kox 6 Rue des Prés 5561 Remich Luxembourg E-mail: corinne@domainekox.lu
FORTH Sotiris Ioannidis N Plastira Str 100 70013 Heraklion Greece E-mail: sotiris@ics.forth.gr	**Fraunhofer IESE** Daniel Schneider Fraunhofer-Platz 1 67663 Kaiserslautern Germany E-mail: daniel.schneider@iese.fraunhofer.de
FORTH Sotiris Ioannidis N Plastira Str 100 70013 Heraklion Greece E-mail: sotiris@ics.forth.gr	**KUKA Assembly & Test** Michael Laackmann Uhthoffstrasse 1 28757 Bremen Germany E-mail: michael.laackmann@kuka.com
Locomotec Sebastian Blumenthal Bergiusstrasse 15 86199 Augsburg Germany E-mail: blumenthal@locomotec.com	**Luxsense** Gilles Rock 85-87 Parc d'Activités 8303 Luxembourg Luxembourg E-mail: gilles.rock@luxsense.lu
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium E-mail: s.hansen@opengroup.org	**Technology Transfer Systems** Paolo Pedrazzoli Via Francesco d'Ovidio, 3 20131 Milano Italy E-mail: pedrazzoli@ttsnetwork.com
University of Hull Yiannis Papadopoulos Cottingham Road Hull HU6 7TQ United Kingdom E-mail: y.i.papadopoulos@hull.ac.uk	**University of Luxembourg** Miguel Olivares Mendez 2 Avenue de l'Universite 4365 Esch-sur-Alzette Luxembourg E-mail: miguel.olivaresmendez@uni.lu
University of York Simos Gerasimou & Nicholas Matragkas Deramore Lane York YO10 5GH United Kingdom E-mail: simos.gerasimou@york.ac.uk nicholas.matragkas@york.ac.uk	

DOCUMENT CONTROL

Version	Status	Date
0.1	Initial outline	10 June 2022
0.5	Hull input for HiP-HOPS, Dymodia, & ODE Converter etc done	20 June 2022
0.8	Integration with IESE input; ready for internal project review	23 June 2022
1.0	Final version incorporating feedback from TTS	29 June 2022

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

The Executable Digital Dependability Identity — or EDDI — is a composable model-based artefact that contains dependability information about a system. The EDDI concept is intended for dynamic dependability management at runtime. However, in order to generate runtime EDDIs, relevant information must be captured at design time.

It is these design-time activities that are covered in this deliverable. Existing safety analysis tools are targeted to create appropriate system models and safety artefacts, which can then be converted to ODE-compliant models via tool adapters and model converters. The tools, adapters, and converters are described within, along with information about how they can be used.

To demonstrate the use of the tools, the HiP-HOPS tool is applied to the Locomotec hospital disinfection robot case study.

LIST OF ABBREVIATIONS

ACCF	Actual Common Cause Failure
BE	Basic Event (i.e., root cause of a fault tree)
BN	Bayesian Network
CCF	Common Cause Failure
DDI	Digital Dependability Identity
DFT	Dynamic Fault Tree
DRA	Dynamic Risk Assessment
EDDI	Executable Digital Dependability Identity
FMEA	Failure Modes & Effects Analysis
FMEDA	Failure Modes, Effects, & Diagnostic Analysis
FTA	Fault Tree Analysis
HARA	Hazard Analysis & Risk Assessment (or Hazard And Risk Analysis)
MAS	Multi-Agent System
MBSA	Model-based Safety Analysis
ML	Machine Learning
MRS	Multi-Robot System
ODE	Open Dependability Exchange metamodel
PCCF	Potential Common Cause Failure
SACM	Structured Assurance Case Metamodel
SINADRA	Situation-Aware Dynamic Risk Assessment
TARA	Threat Analysis & Risk Assessment
UV-C	Shortwave ultraviolet light (200-280nm wavelength)
UW	Uncertainty Wrapper

1. INTRODUCTION

The Executable Digital Dependability Identity — or EDDI — is a composable model-based artefact that contains dependability information about a system. The EDDI concept is primarily aimed at runtime usage, where the EDDIs are intended to be executed onboard or alongside their target system to perform dynamic dependability management.

However, in order to generate runtime EDDIs, relevant information must be captured at design time. Furthermore, EDDIs can also serve as purely design-time artefacts, storing dependability information about a system and supporting design-time dependability analyses. It is these design-time activities that are covered in this deliverable; the runtime aspects are detailed further in **D7.1: Runtime Safety and Security Concept** and **D7.2: Tools for Generation of Runtime EDDIs**.

EDDIs are based on the Open Dependability Exchange (ODE) metamodel. The ODE is intended to be the common interchange format between the different EDDI-related tools, so that common models can be created and used to generate or interact with runtime EDDIs regardless of the original design-time tool. The updated ODE is defined in **D4.2/D5.2: Safety/Security ODE and EDDI Specification**.

To support design-time EDDI activities such as system modelling and safety analysis, three primary existing safety analysis tools are being targeted: HiP-HOPS, safeTbox, and Dymodia. These tools have been chosen for their good support for model-based safety analysis and, in the case of the first two, because we develop them ourselves and thus can more readily make required changes. All three are tools that allow the creation or import of architectural models of a system which can then be annotated with failure data to record the failure behaviour of the system. The data can then be analysed to produce safety analysis artefacts such as fault trees and FMEAs. More information about each tool can also be found in **D4.1: Safety Analysis Concept**.

Although not a safety analysis tool per se, we also target the GeNIe tool for Bayesian network analysis, as described in Section 6.

To ensure these tools can be integrated into the EDDI toolchain, their models need to be convertible to the ODE in order to facilitate exchange with other tools. This work is ongoing but the progress so far is described in this deliverable, along with an example of applying one of them to a SESAME use case.

2. HiP-HOPS

2.1 What is HiP-HOPS

Hierarchically Performed Hazard Origin & Propagation Studies, or HiP-HOPS[1] to use its more convenient name, is a comprehensive model-based safety analysis methodology with a tool of the same name.

Originally created in the late 1990s [1], it has been the focus of continuous development at the University of Hull over the ensuing 20+ years. Over that time it has evolved from a simple fault tree analysis tool to a powerful dependability engine with a wide range of capabilities and additional functionalities. These include multiple failure mode FMEAs, optimisation capabilities (for architecture, maintenance, and safety requirements) [2], dynamic analysis (using state machines and/or dynamic fault trees) [3], and automatic allocation of safety integrity levels according to safety requirements to support ISO 26262-style workflows [4].

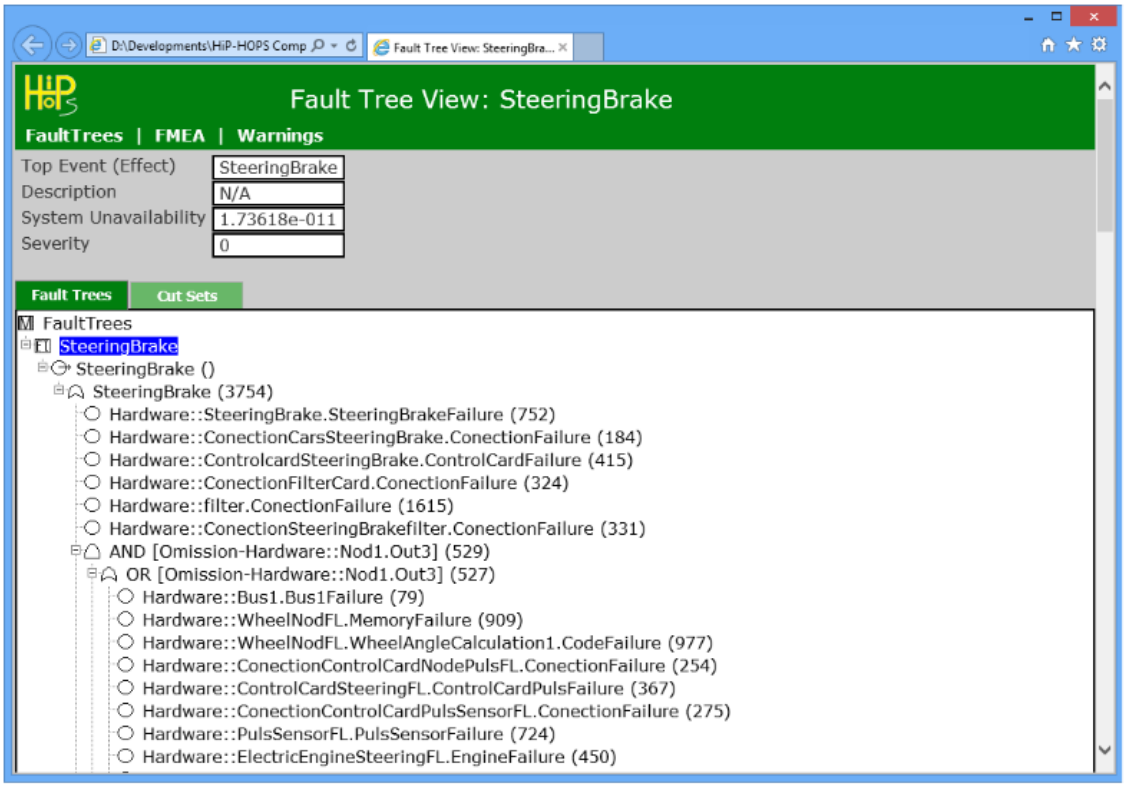


Figure 1 - HiP-HOPS analysis output

2.2 How it works

A full manual is available on the HiP-HOPS website. However, a shorter explanation on the core HiP-HOPS functionality is provided below.

A standard HiP-HOPS analysis consists of four main phases plus an extra optional step:

[1] https://hip-hops.co.uk

1. **System modelling**. In this phase, an external modelling package such as Matlab Simulink[2], SimulationX[3], or MetaEdit+ (with EAST-ADL)[4] is used to create a model of the system architecture — i.e., components and the connections between them. This provides the basis for the origin of failure modes and the channels along which they can propagate.

2. **Failure annotation**. Using a specific interface or HiP-HOPS plugin to one of the tools above, the system model is annotated with additional information to describe the system's failure behaviour. Components are annotated with potential failure modes — failures that originate within them, or which are first detectable via them — as well as how deviations of their outputs are caused either by these internal failure modes or deviations received at their inputs. Probabilistic information can also be added to these failure modes if available. Further logic can be added to describe the propagation of failures between components if required. Finally, system-level hazards are also identified and linked to the potential component output deviations that could cause them.

3. **Synthesis of fault propagation models**. With the model complete, it is passed to the HiP-HOPS engine itself. The tool begins by generating a fault propagation model: working backwards from each identified hazard, HiP-HOPS works its way through the system by following failure propagations and establishes all the possible component-level causes of that hazard. The result is a network of interconnected fault trees.

4. **Failure analysis**. Having generated fault trees in the previous step, HiP-HOPS can now analyse them. It does this by removing redundancies, resolving contradictions, and simplifying causes until it achieves the minimal set of possible causes for each hazard, along with an estimate for unavailability if the necessary data was provided. From these it can also generate a system-wide multiple failure mode FMEA, which indicates the potential hazards each failure mode can lead to.

5. **Optional extra steps.** If desired, and if appropriate data is present, these analysis results can also be used as the basis of other activities, e.g. architectural optimisation or allocation of safety requirements.

Each of these first four steps will be described further below.

2.2.1 System Modelling

A HiP-HOPS-compatible system model requires two main elements: components and connections. Components represent functional system elements. They do not necessarily have to be physical hardware components; software components and abstract functions are also possible components, and indeed it is not uncommon for different models to be created at different stages of the design lifecycle, e.g. beginning with a basic functional design architecture and then progressing through to a joint hardware/software architecture later. HiP-HOPS provides the concept of 'perspectives', which are essentially top-level subsystems, to separate out different forms of components if

[2] https://www.mathworks.com/products/simulink.html
[3] https://www.esi-group.com/products/system-simulation
[4] https://www.metacase.com/solution/east-adl.html

required. This allows, for instance, separate modelling of hardware and software components while still allowing software to be allocated to the hardware that executes it.

The other main elements are the connections between components. Depending on the modelling tool, these can vary from simple unidirectional point-to-point connections to more complicated undirected lines connecting multiple components with their own propagation logic. Connections connect to components via ports, which represent the interface between a component and the rest of the system.

As the name would suggest, hierarchical modelling is also possible. Any component can have a subsystem within it, which can host more components and more connections. HiP-HOPS can automatically combine failures that originate from a component's subsystem with failures that originate at the top level. This can be particularly useful for describing failures that affect the component subsystem as a whole; for example, individual failure modes can be modelled at the subsystem level, while generic issues like electromagnetic interference can be modelled once at the top component level.

Although different modelling tools handle it in different ways, in general a HiP-HOPS input file — containing the description of the system architecture and any failure annotations in XML form — is generated by the tool separately from its own model format. In this way, HiP-HOPS has a common input interface regardless of the originating modelling tool used.

2.2.2 Failure annotation

HiP-HOPS failure annotations come in three main forms:

- Failure modes / basic events, which are component-level failures;
- Input & Output deviations, which are propagated failures at inputs/outputs;
- Common Cause Failures, which are failures that exist at a system level rather than component level.

Failure modes (generally referred to as basic events in HiP-HOPS) are typically random hardware failures experienced by components, but in general they represent any failure that originates (or is initially detected) within a component. Software bugs, overheating, jamming, electromagnetic interference, water ingress etc are all possible basic events.

Output deviations represent the way these basic events (or input deviations) can cause unwanted deviations from nominal output at the output ports. Deviations all require a particular failure class, which indicates the general type of failure. Common classes include omission (lack of input/output when expected), commission (unintended/unwanted input/output), timing (e.g. late, early), and value (e.g. high, low) failures. Deviations are expressed as a combination of this failure class and the port name, e.g. `Omission-ImageOut` means an omission at the ImageOut port.

Output deviations specifically are also provided with logical expressions to describe their causes. This is how HiP-HOPS understands how failures propagate through

components. Most expressions consist of basic events, input deviations, and simple Boolean operators (AND/OR), but more complex logic is also possible. As an example:

```
Omission-ImageOut = HardwareDefect OR Omission-PowerIn
```

Here the omission at image out for an imaging component (e.g. a camera) is caused either by an internal hardware defect or a lack of power at the electrical input port.

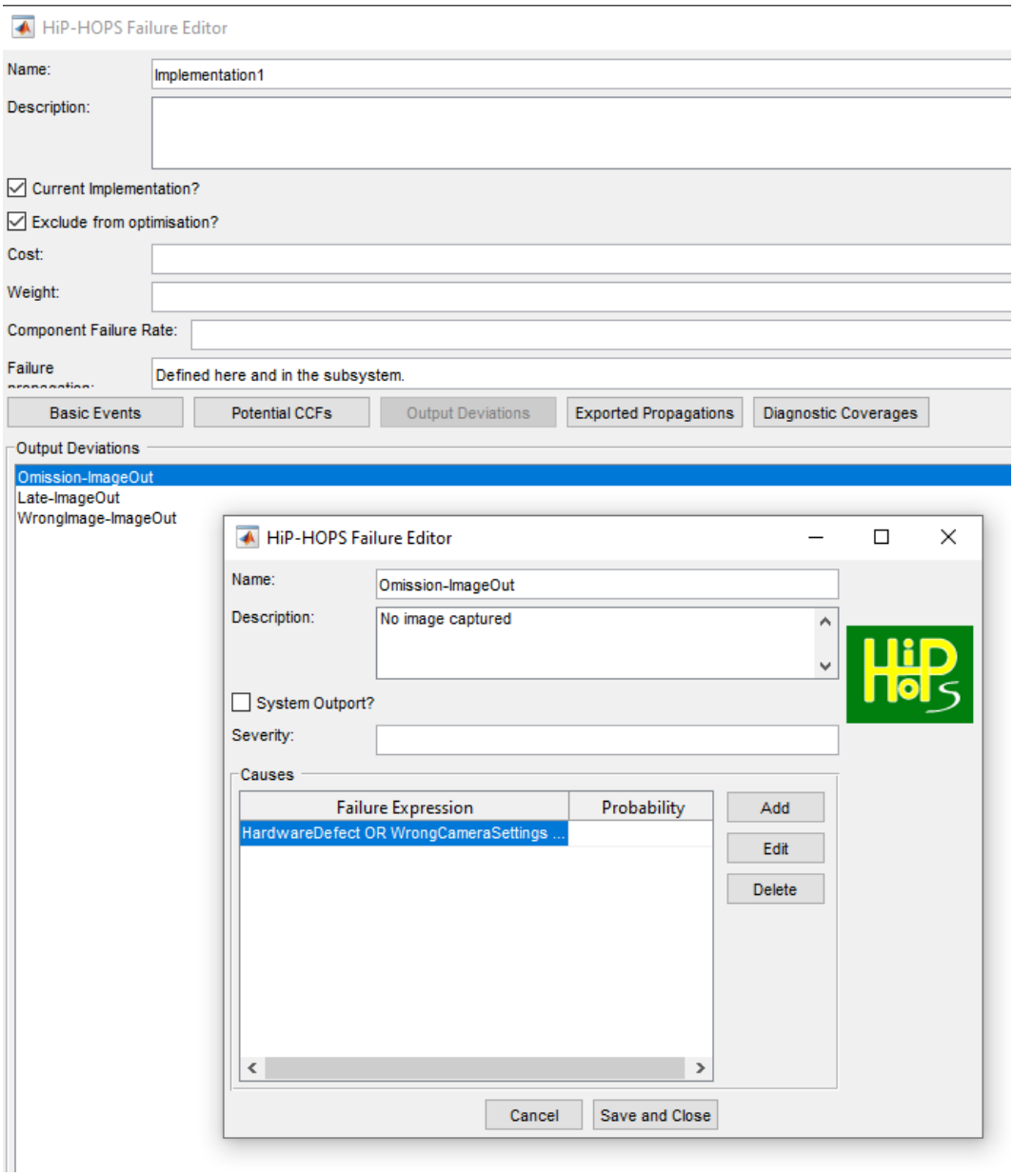


Figure 2 - HiP-HOPS failure annotation interface for Matlab Simulink

Output deviations can also refer to common cause failures. HiP-HOPS differentiates between what it calls *potential* common cause failures (PCCFs) and *actual* common cause failures (ACCFs). PCCFs are defined at the component level and represent effects from possible CCFs, should they exist; if the CCF does not exist in a particular operating context, then the PCCF is ignored.

To 'activate' a PCCF, a corresponding ACCF must be defined at the system level. These are common failures not specific to any particular component and typically represent environmental effects, e.g. flooding, fire etc.

In this way, the failure logic is more generic and can be stored in e.g. a component library without regard for whether or not the system environment features a particular CCF. For instance, components originally modelled for use aboard a ship may have a PCCF "water flooding" defined, but if re-used aboard an aircraft, the water flooding ACCF is (hopefully) absent and thus these PCCFs remain inert with no need to modify the logic of every output deviation.

2.2.3 Synthesis of propagation models

The internal workings of the tool are beyond the scope of this document, but different options are available when conducting the synthesis. The most important affect the structure of the resultant fault trees, which can either be left in their original form to better reflect the propagation of failures, or simplified and contracted to make them easier to read, albeit at the cost of most of the intermediate steps.

2.2.4 Failure Analysis

HiP-HOPS performs two main forms of analysis: fault tree analysis (FTA) and failure modes & effects analysis (FMEA). The former can involve both qualitative analysis (deduction of logical causes) and quantitative analysis (estimation of failure probability), while, as mentioned, FMEA can include both direct effects of failures and contributing effects that only occur in conjunction with other failures.

HiP-HOPS generates its analysis output in the form of multiple XML files which are by default accompanied with a HTML wrapper that enables them to be viewed in a web browser. Other forms of output are also possible, according to the options set, including output of a single XML file with no HTML (most useful for importing into other tools) and generation of an Excel spreadsheet.

2.3 SESAME-SPECIFIC EXTENSIONS AND FUNCTIONALITY

Both HiP-HOPS and the various compatible modelling tools are closed source tools. Therefore, in order to be able to use them effectively as part of the EDDI toolchain, it needs to be possible to convert from the native file formats (e.g. HiP-HOPS architecture or results XML files) to ODE files. This is done via one of the converters described in section 5.

Because HiP-HOPS served as one of the sources of input to the ODE design process, the two metamodels are broadly compatible. HiP-HOPS is fully capable of generating an ODE-compliant system architecture along with integrated ODE failure models like fault trees and FMEAs.

However, as a design time tool, HiP-HOPS lacks the capability to add runtime-specific information such as event monitoring or responses to fault detection/diagnosis. This must be added after the fact via other tools (e.g. one of the converters).

3. SAFETBOX

SafeTbox is a model-based safety modelling and analysis tool developed by and commercially available from Fraunhofer IESE. The tool supports the safety engineering lifecycle, focusing particularly in the stages before implementation, i.e. requirements analysis and system design. Thus, the tool allows modelling a system's architecture, analysing its risks and potential failures, and structuring assurance cases regarding the system's dependability.

3.1 WHAT IS SAFETBOX

SafeTbox is an add-in to the Enterprise Architect (EA) modelling framework. Thus, it extends EA's system modelling support for popular languages UML and SysML, featuring its own component-based system modelling variant. An example of how this variant appears graphically in the tool can be seen in Figure 3.

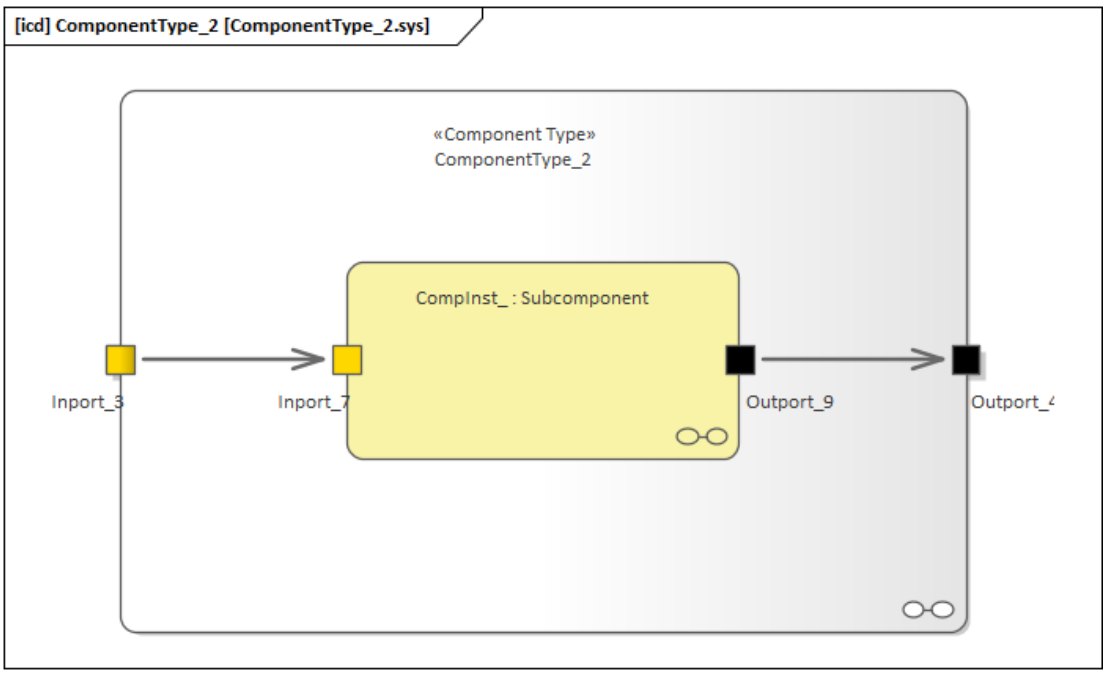


Figure 3 - Example of abstract component in safeTbox

In this figure, a host component (whose type is 'Component_Type2') contains input and output ports with which it connects to its external environment, and simultaneously contains an internal component (whose type is 'Subcomponent' and whose instance's name is 'CompInst'). The host component's ports are linked to the contained component's ports, allowing information, energy, or similar relations to flow from outside the host component, to the contained component, and vice versa.

While these are listed as 'components', they are abstract and flexible enough to also represent complex system architectures. At the system level, dependability analysis often begins with a Hazard Analysis and Risk Assessment (HARA), a part of which can be seen in Figure 4. In the figure, a spreadsheet view allows the user to navigate between sheets, each of which is dedicated to a different process step of the HARA. The initial sheets are mostly for process documentation and information; the first process step begins with the 'Functions' sheet, as seen in Figure 4. In this sheet, the user can

specify their own system functions manually, or add existing ones defined in the system model previously. For example, each of the components listed before can be considered as a function and included in the list.

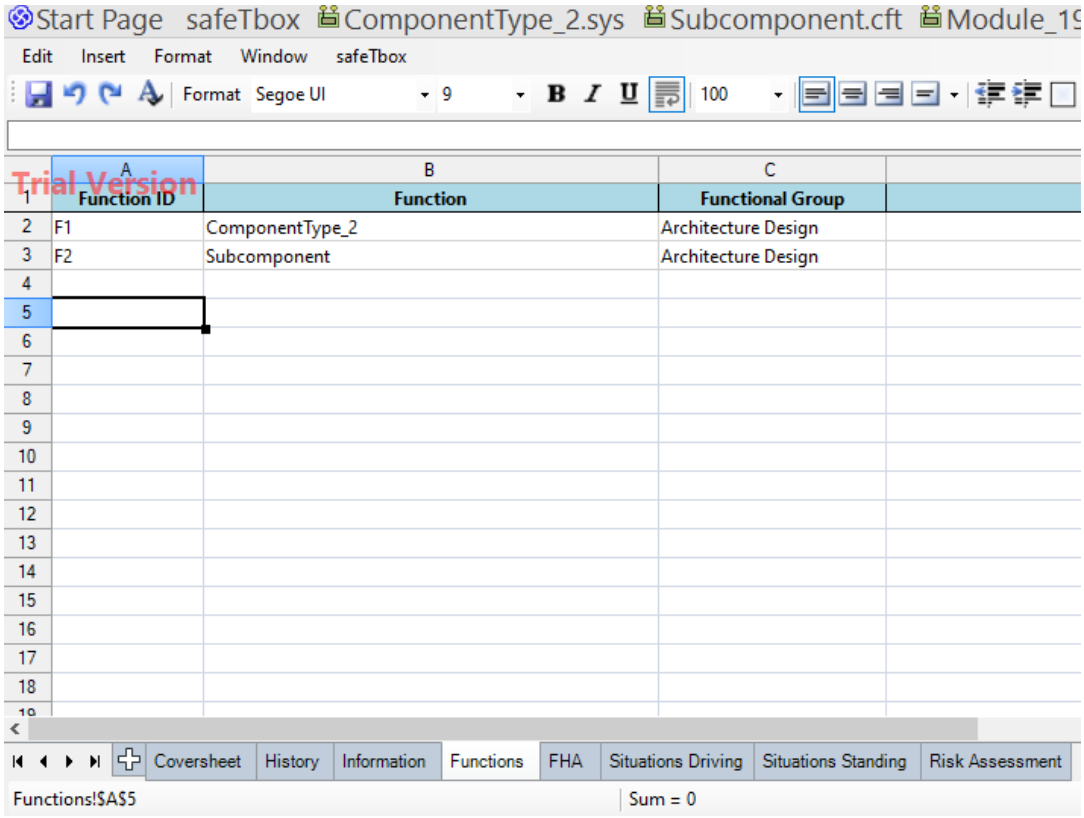
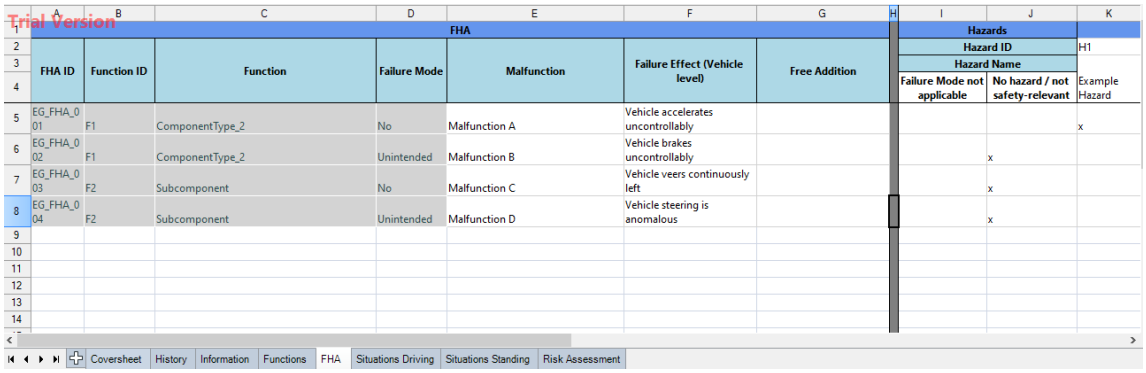


Figure 4 - Example of Functions sheet in safeTbox HARA

Once functions are included, the next step is identifying the different possible modes of functional failure. This is performed in the 'FHA' sheet, as seen in Figure 5. A standard list of keywords is typically used here, including 'Omission'/'Commission' (function was not provided when needed / provided when not needed), 'Too Early'/'Too Late', 'Too Low'/'Too High', etc. Each of the functions listed previously can be considered against each of the keywords (and beyond, if the analyst finds a unique failure). Failure modes that are considered potentially hazardous (could cause harm to people, environment, etc.) can be mapped to specific hazards. Each of the row entries corresponds to a functional failure, and each vertical entry on the rightmost columns represents a user-specified hazard that can be associated with those corresponding functional failures.

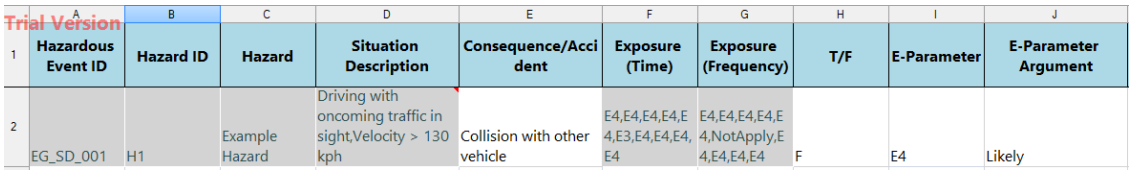


FHA							Hazards		
							Hazard ID		H1
FHA ID	Function ID	Function	Failure Mode	Malfunction	Failure Effect (Vehicle level)	Free Addition	Hazard Name		
							Failure Mode not applicable	No hazard / not safety-relevant	Example Hazard
EG_FHA_001	F1	ComponentType_2	No	Malfunction A	Vehicle accelerates uncontrollably				x
EG_FHA_002	F1	ComponentType_2	Unintended	Malfunction B	Vehicle brakes uncontrollably			x	
EG_FHA_003	F2	Subcomponent	No	Malfunction C	Vehicle veers continuously left			x	
EG_FHA_004	F2	Subcomponent	Unintended	Malfunction D	Vehicle steering is anomalous			x	

Coversheet | History | Information | Functions | FHA | Situations Driving | Situations Standing | Risk Assessment

Figure 5 - Example of FHA sheet in safeTbox HARA

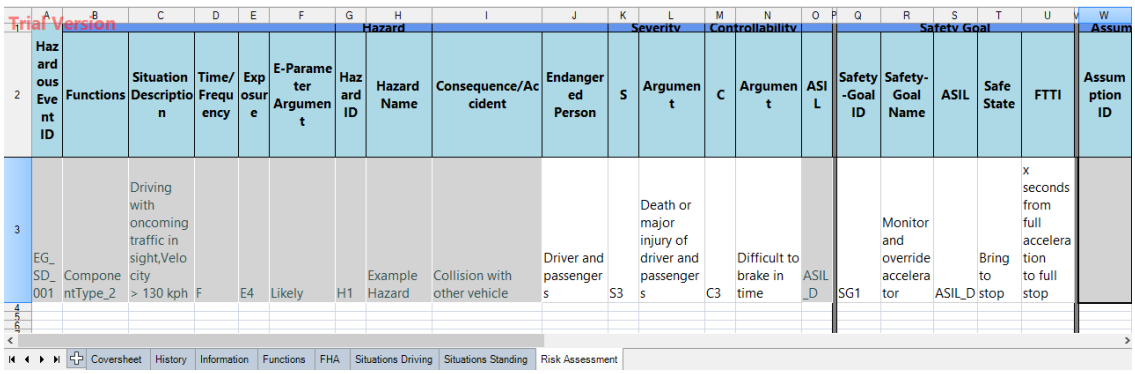
Once a set of hazards has been collected, the 'Situations Driving' and 'Situations Standing' sheets allow detailing of the exposure risk parameter for specific situations involving those hazards (while the vehicle is driving or standing still accordingly). See Figure 6 for an example. Exposure reflects how likely a given hazardous situation is expected to occur during the vehicle's operation. The combination of a hazard and such a situation is referred to as a hazardous event.



Hazardous Event ID	Hazard ID	Hazard	Situation Description	Consequence/Accident	Exposure (Time)	Exposure (Frequency)	T/F	E-Parameter	E-Parameter Argument
EG_SD_001	H1	Example Hazard	Driving with oncoming traffic in sight,Velocity > 130 kph	Collision with other vehicle	E4,E4,E4,E4,E4,E3,E4,E4,E4,E4	E4,E4,E4,E4,E4,NotApply,E4,E4,E4,E4	F	E4	Likely

Figure 6 - Partial example of situations driving sheet in safeTbox HARA

In the final sheet ('Risk Assessment'), the hazardous events from both situation sheets are collected, and each is further rated with respect to the severity of the hazard (e.g. death or severe injury of the driver) and with respect to its controllability by the driver (e.g. the driver can easily observe the problem and halt the vehicle). The more severe, frequently occurring, and difficult to control a hazard (-ous event), the higher the overall assessed risk rating it receives (represented as an ISO26262 Automotive Safety Integrity Level - ASIL). To mitigate one or more hazardous events, appropriate safety goals must be specified, and be assigned an ASIL corresponding to the highest ASIL of all the hazardous events they are intended to address.



							Hazard				Severity		Controllability			Safety Goal						Assum
Hazardous Event ID	Functions	Situation Description	Time/Frequency	Exposure	E-Parameter Argument	Hazard ID	Hazard Name	Consequence/Accident	Endangered Person	S	Argument	C	Argument	ASIL	Safety-Goal ID	Safety-Goal Name	ASIL	Safe State	FTTI	Assumption ID
EG_SD_001	ComponentType_2	Driving with oncoming traffic in sight,Velocity > 130 kph	F	E4	Likely	H1	Example Hazard	Collision with other vehicle	Driver and passengers	S3	Death or major injury of driver and passengers	C3	Difficult to brake in time	ASIL_D	SG1	Monitor and override accelerator	ASIL_D	Bring to stop	x seconds from full acceleration to full stop	

Coversheet | History | Information | Functions | FHA | Situations Driving | Situations Standing | Risk Assessment

Figure 7 - Example of risk assessment sheet in safeTbox HARA

Either during or after the HARA, more detailed failure analysis may be needed to dig deeper into underlying causes of system-level failures. Fault trees are one type of causal analysis that can be used in this way. Figure 8 shows how the failure logic of the subcomponent might look like. In the figure, the component's structure is represented by the outer box ('<<Component Type>>Subcomponent'), whereas the inner box ('<<CFT>>Subcomponent') represents its Component Fault Tree (CFT) [5] [6]. CFTs (described further in **D4.1: Safety Analysis Concept**) are a variant of traditional fault trees and allow us to modularize fault tree models and map them to their corresponding components.

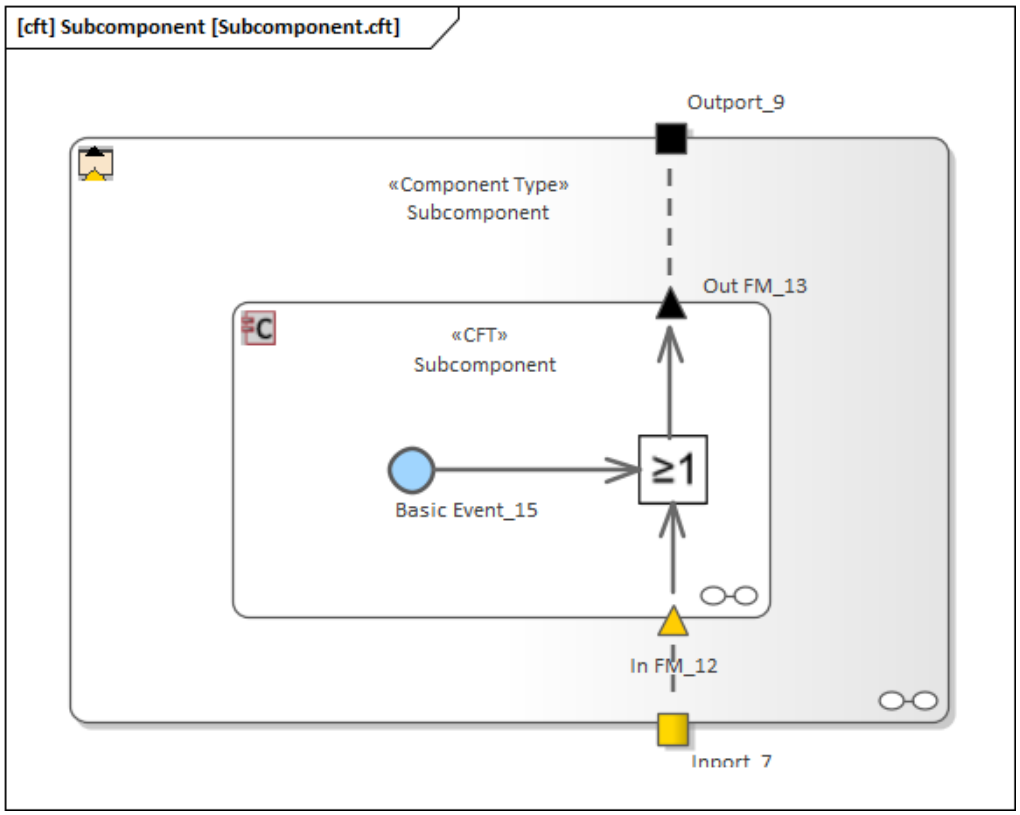


Figure 8 - Example of failure logic for subcomponent in safeTbox

The CFT contains an input failure mode (In FM_12) and output failure mode (Out FM_13). Each is associated with the outer box's component ports, indicating where that failure might occur and/or propagate to inside or outside the component. Inside the CFT, the failure modes are linked by a logical 'OR' gate ("at least 1" symbol) to a basic event (Basic Event_15). The basic event represents an internal component fault that does not need to be decomposed further. Automatic qualitative analysis of the CFT can reveal the minimal combinations of necessary and sufficient basic events and input failure modes needed to trigger a specific failure. The analysis can also be quantitative, evaluating the likelihood of the subject failure of occurring, based on assumed probability models of the basic events.

Finally, assurance cases can also be modeled, using the Goal Structuring Notation (GSN) [7]. An example of an abstract GSN structure in safeTbox can be seen in Figure 9, where the safety of 'ComponentType_2' is claimed to be acceptable. The claim is depicted using a rectangle (Goal_20), also known as a 'Goal' in GSN. To support the claim, a strategy of argument (represented as a parallelogram Strategy_21) is used,

arguing that acceptable safety is achieved by evaluating the system's residual risk after its safety goals have been validated. Another Goal (Goal_22) claims that the residual risk is acceptable; an associated Assumption (Assumption_25, represented with an ellipse) assumes that the safety goals contribute no risk of their own to the system. Finally, evidence supports the final claim, by referring to the CFT of the subcomponent seen previously in Figure 8, and notes that the risk of critical failures after the safety goals have been implemented is acceptably low. Of course, we should note that this is an oversimplified example, for illustration purposes.

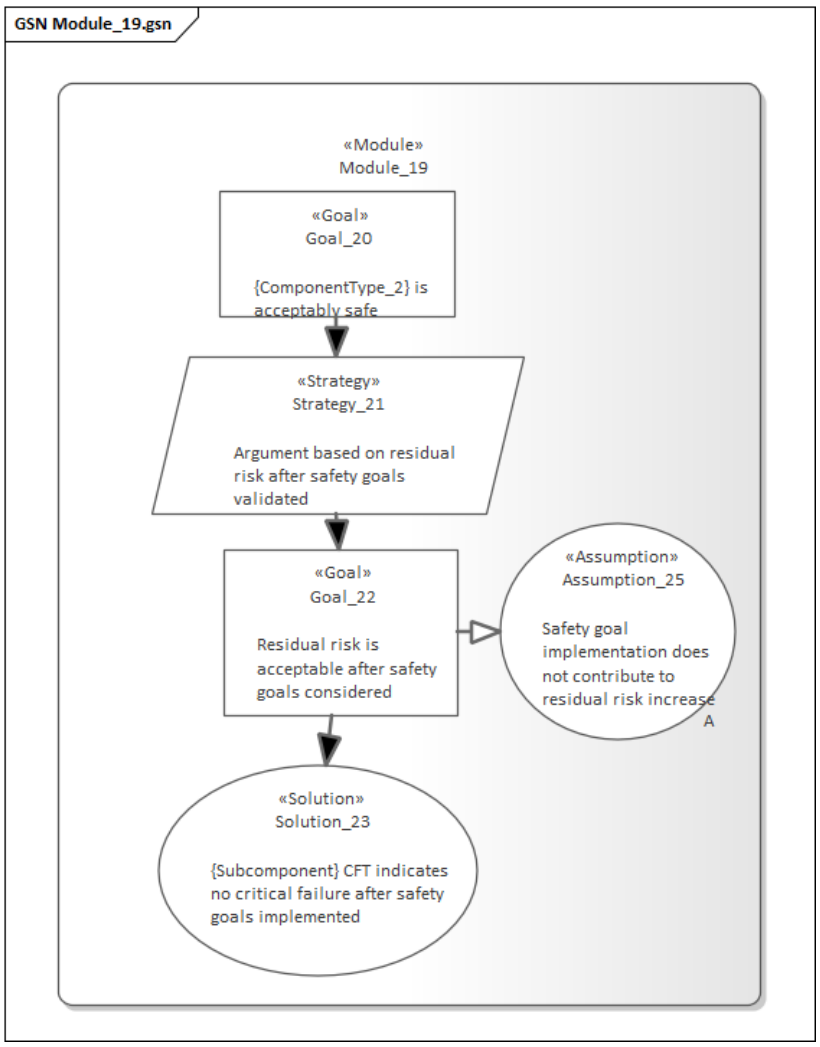


Figure 9 - Example of abstract GSN structure in safeTbox

3.2 HOW IT WORKS

For detailed guidance on using safeTbox, the user manual and provided example model should be useful references[5]. An abridged guide is included below for users in SESAME.

To use safeTbox, an existing installation of Enterprise Architect (EA) is required, up to version 15.1. Trial versions are also supported. SafeTbox can be downloaded and

[5] https://safetbox.de/docu-samples

installed for free from its dedicated website[6]; the user must register an account, as seen in Figure 10, and they can then download the installer executable and a trial license, as seen in Figure 11.

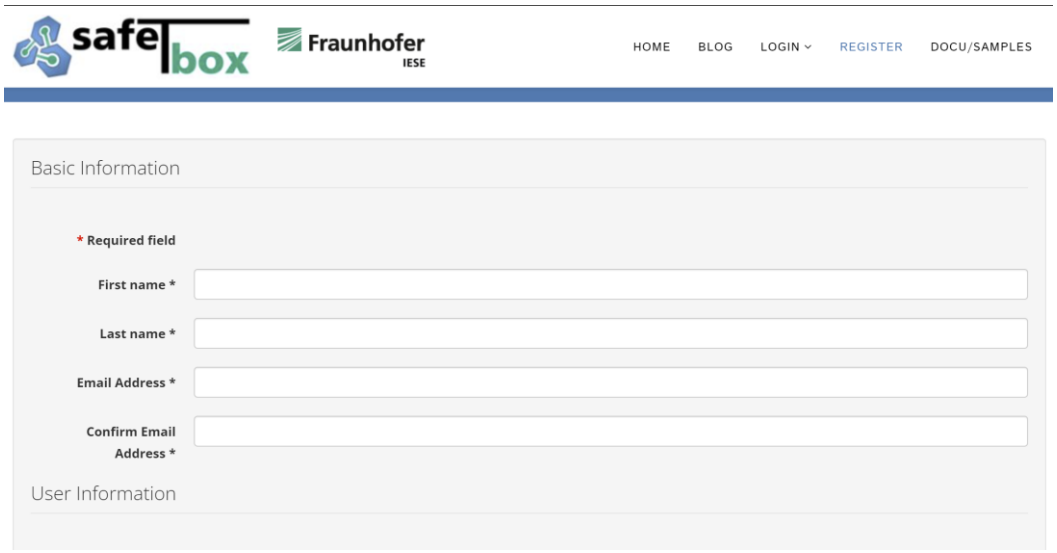


Figure 10 - safeTbox website registration page

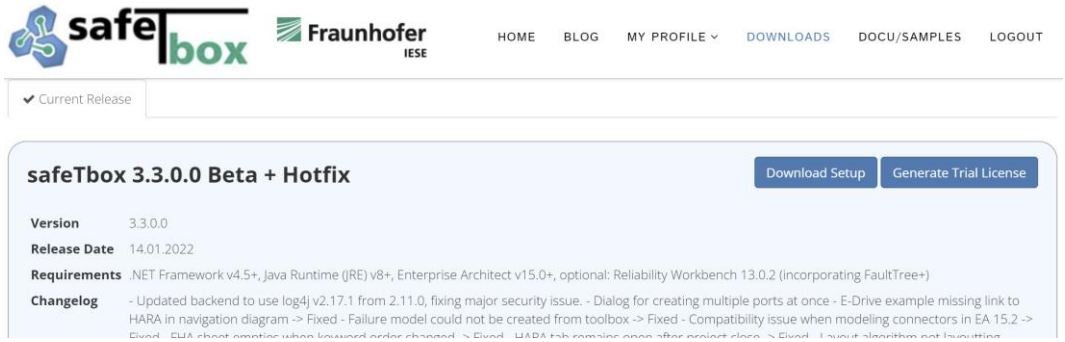


Figure 11 - safeTbox website downloads page

Once downloaded, run the installer executable and follow the instructions (installing any additional requirements that are needed as well).

Once installed, safeTbox should be usable the next time EA is launched.

The first time EA is launched, you will be requested to provide your downloaded license. Simply use the dialog to navigate to the license file and confirm. Upon starting EA from here onwards, you should be greeted by the safeTbox welcome screen, as seen in Figure 12.

[6] www.safetbox.de

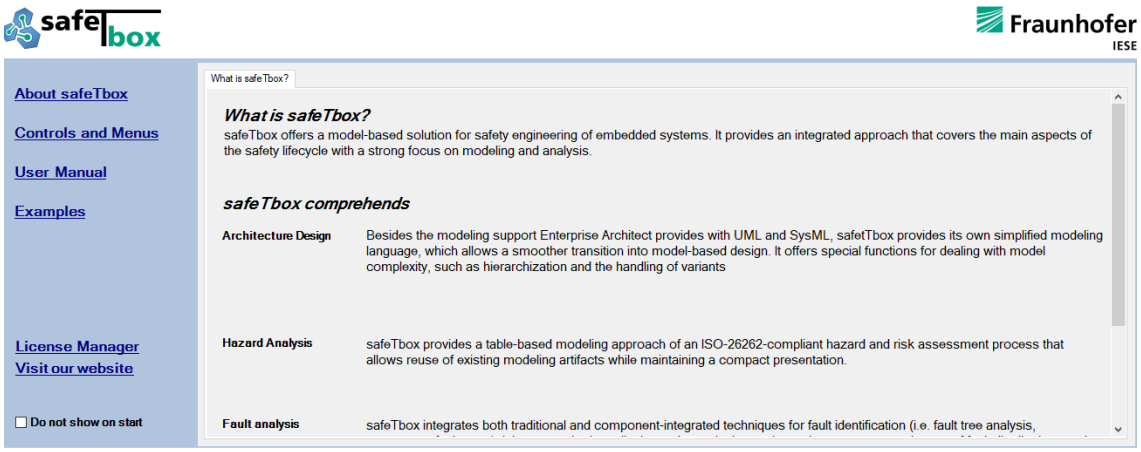


Figure 12 - safeTbox welcome screen

If this is not the case, review your installation steps, consult the user manual, or contact safeTbox user support via the website.

To create a model, open the (project) 'Browser' pane and click to open or create a new project, as per Figure 13.

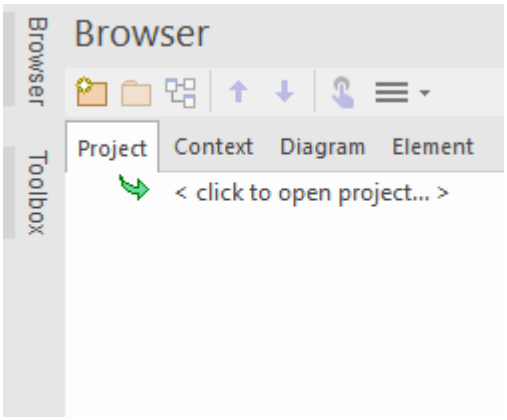


Figure 13 - Creating a new project in safeTbox

Use the file dialog to either select the file path of the new project, or the file path of an existing one. After successful creation/loading, the project's 'Model' should be visible in the Browser pane, as in Figure 14. By selecting and pressing Ctrl+Space, the safeTbox 'Smart Menu' should appear, allowing access to options such as Create, as seen in Figure 15.

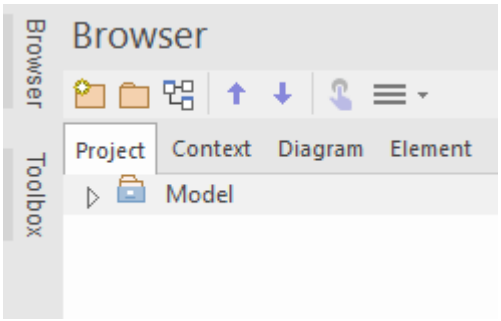


Figure 14 - A project with a loaded model in safeTbox

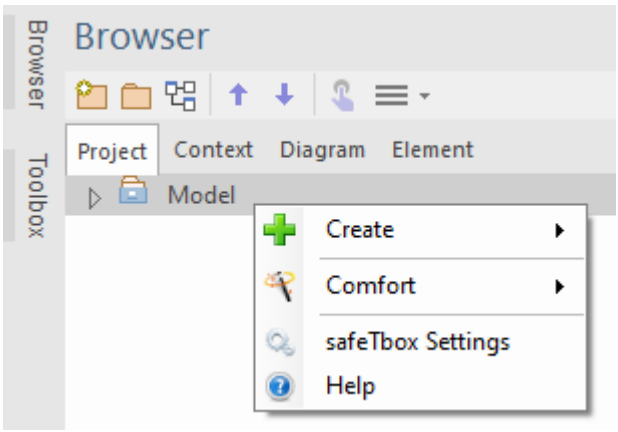


Figure 15 - safeTbox Smart Menu

The Create option is likely the most useful, as it lets the user add new system, HARA, CFT, and GSN models to the project model.

3.2.1 Using SafeTbox for Systems Modelling

To model a system e.g. an individual robot or even MRS, create a new Architecture Model via the Smart Menu.

A new tab should appear, where a randomized name has been given to the new component, seen at the center of the view. To rename the component and set its properties, either double-click on it, or select and use the Smart Menu to access its safeTbox Properties. In the dialog that appears, you can set various options, importantly its name and description.

Upon confirming your changes, the dialog should close and the component should reflect its new name and other visual properties that changed.

New ports and sub-components can also be added via the Smart Menu. Ports can reflect communication or interaction points between the subject system and the external environment.

Sub-components can represent participating elements of their host system e.g. individual robots within an MRS. Components can be re-used by selecting the corresponding option in the component creation dialog, as seen Figure 16.

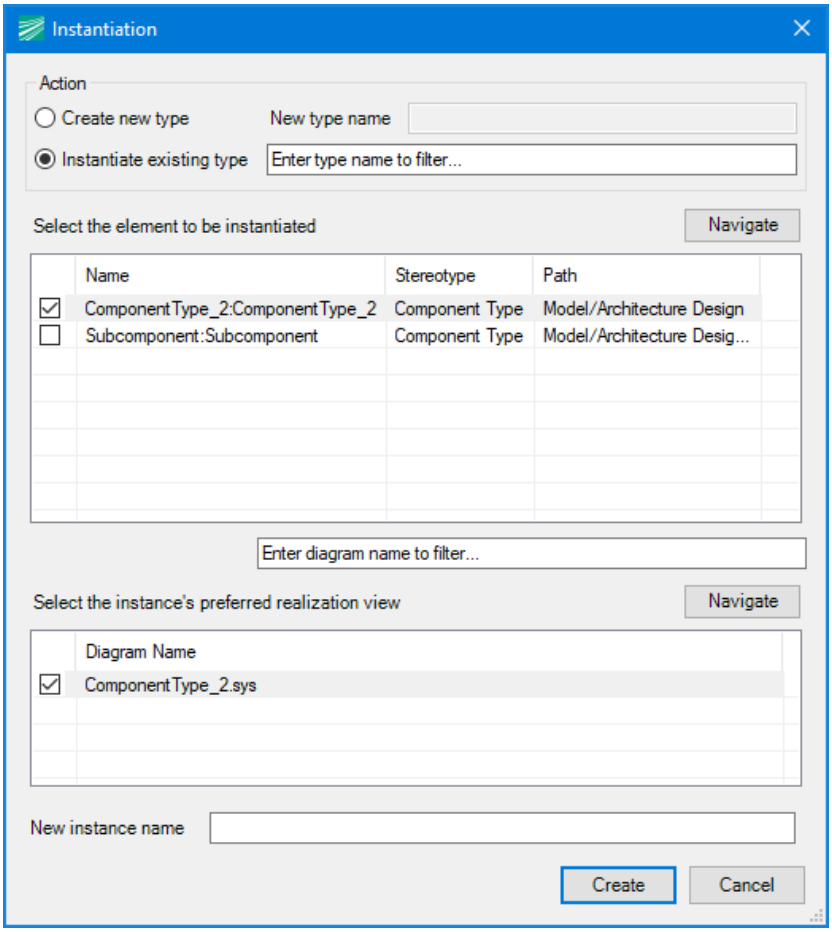


Figure 16 - Instantiating existing component types in safeTbox

The first list in the dialog allows the user to select which existing component to instantiate. The second list refers to a more advanced feature that is explained in the user manual. At the end of the dialog, an optional name for the new instance of the component can be provided directly.

3.2.2 Using SafeTbox for CFT Modelling

Once a new component type has been created, a CFT can be added to it using the Smart Menu => Create => Add Failure Model. A new tab should appear, depicting an emptier but otherwise similar view to Figure 8. Once again, the Smart Menu should allow access to creating all elements relevant to the CFT diagram.

3.2.3 Using SafeTbox for HARA

A HARA can be created from the (project) Browser pane via the Smart Menu. Note that the HARA model requires user-triggered synchronization with the rest of the model; this option is available via the tab menu => safeTbox => Synchronization. Synchronization allows updates in both Model-HARA directions to occur.

As you progress through the sheets, you can often add safeTbox-related content to the rows and columns by right-clicking on the margin. For example, this applies in the Functions sheet, where the user can add Functions by right-clicking on the row margin and selecting the Add Function option in the menu that appears, as seen in Figure 17.

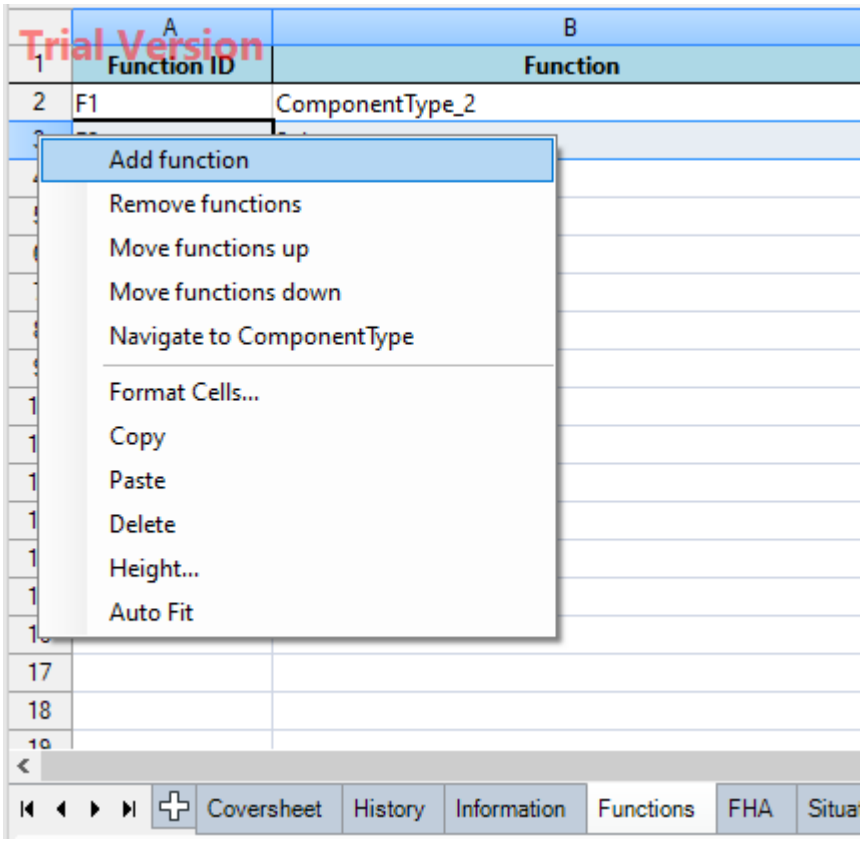


Figure 17 - Adding Functions to the HARA in safeTbox

Similar functionality can be found throughout the HARA spreadsheet, including:

- In the FHA sheet, new Hazard entries can be added by right-clicking on the rightmost columns' margin
- In the Situations sheets, new entries can be added by right-clicking on the row margin
- In the Risk Assessment sheet, new Safety Goals can be added by right-clicking on cells under the columns in the Safety Goal section.

The embedded safeTbox menu is also important for some of the sheets:

- In the Functions sheet, safeTbox => "Load existing component types" opens a dialog to conveniently select which system model elements shall be imported into the sheet
- In the FHA sheet, safeTbox => "Permutate functions with failure modes" automatically generates entries based on combinations of the chosen guidewords and the Functions sheet's functions.
- In the Situations sheets, safeTbox => "Update situations sheet" automatically generates situations based on the hazards specified in the FHA sheet.

- In the Risk Assessment sheet, safeTbox => "Update risk assessment sheet" automatically collects hazardous events from the Situations sheets.

3.2.4 Using SafeTbox for GSN Assurance Cases

To create a new GSN case, use the Smart Menu in the (project) Browser. A new GSN Module should appear in the modeling view. A GSN Module represents a part of an assurance case, and its contained elements can be referenced by other Modules.

The Smart Menu allows the user to create new Goals, Strategies, Solutions, and other GSN elements. For referencing other Modules, use the Toolbox pane to access Away elements (e.g. Away Goals), which can be dragged onto the module to add them. Adding a new Away element to a Module will prompt the user with a dialog to specify which GSN element should be referenced. Note that only GSN elements with the property 'Public' can be referenced, so make sure to set that property on the elements you wish to be referenceable (via the safeTbox properties dialog).

3.2.5 Using SafeTbox for ConSerts

Creating ConSerts involves adding at least two models: a Collaborative System Group (CSG), and a Collaborative System (CS). This can be done via the Smart Menu, as seen in Figure 18.

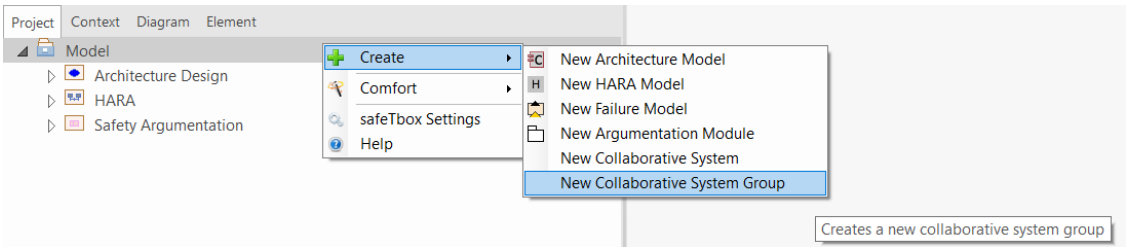


Figure 18 - Adding ConSerts in safetbox

CSGs represent sets of systems that collaborate to deliver an application-level service e.g. hospital disinfection for the corresponding SESAME use case. To implement this service, each constituent CS provides some of its own services as the application service itself, or as a supporting service to other CS.

The CSG acts as a specification of service types and operational modes for the CS that are associated to it. Each CS must offer services compatible with those types.

To begin the process, a CSG should be created first; to edit the service types and operational modes of the CSG, use the Smart Menu => Specify Services option. A form should appear in the main view, as partially seen in Figure 19.

The Operational Modes tab allows adding, editing and removing operational modes. In the figure, an example based on the hospital disinfection case of 3 operational modes is shown. "Setting-up" refers to preparing the disinfection mission e.g. by initializing the robot positions and orientations, "Disinfecting" refers to the mode where the robots are performing the disinfection, and "Ending" refers to the robots returning to their mission completion position.

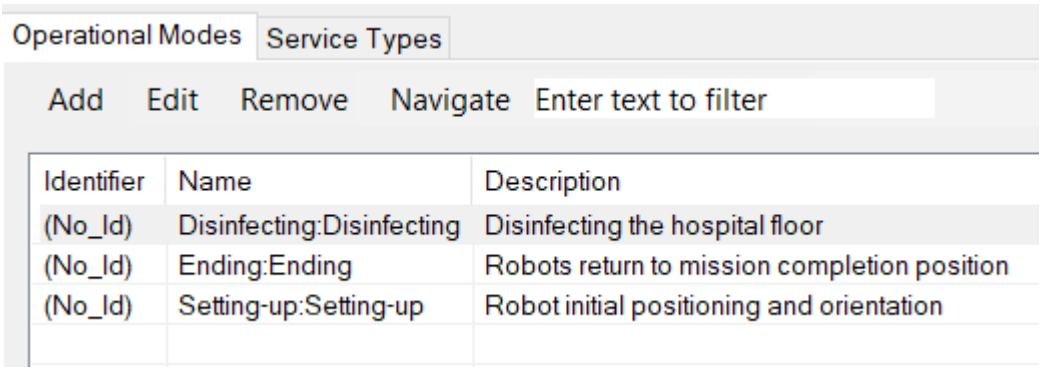


Figure 19 - Editing a Collaborative System Group's Operational Modes in safeTbox

The Service Types tab allows specification of different types of services provided by the constituents to the CSG i.e. the CSs. The Functional Properties sub-tab allows each service type to be specified and be associated with a set of operational modes it is applicable for. In the example seen in Figure 20, an application-level service type, 'Disinfection', is defined in the first list, associated with the 'Disinfecting' operational mode. In the second list, CS-level services are specified; 'NavigateToTarget' refers to the robot navigating to a destination, 'TurnOffLampX' refers to the robot deactivating a specific UV-C lamp and 'TurnOnLampX' the opposite.

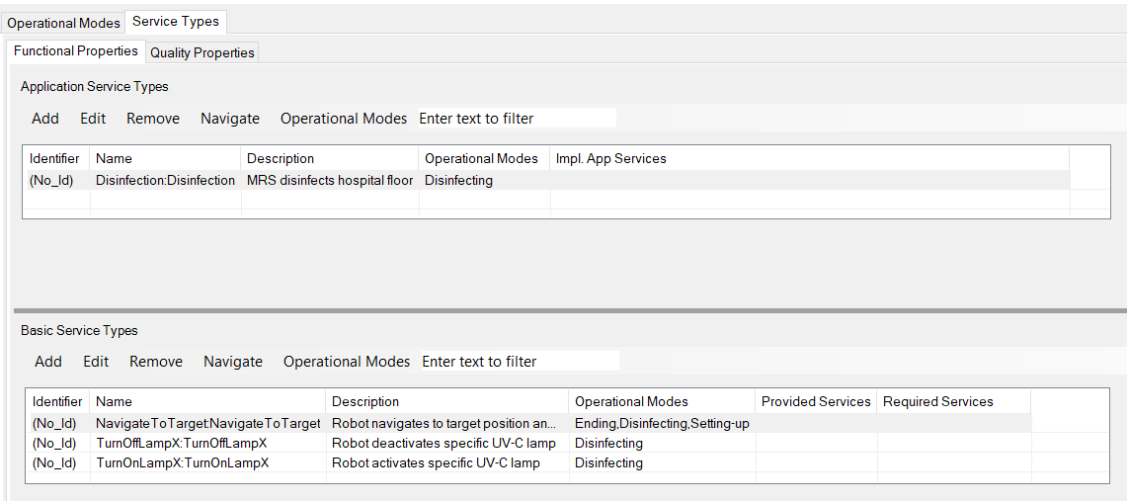


Figure 20 - Specifying CSG Service Types in safeTbox

In the Quality Properties sub-tab, each (application or basic) service's quality properties can be specified. For now, only 'safety' properties are supported semantically by the interface, but the interface can be used to define non-safety properties as well, as there is no semantical constraint.

To specify quality properties for a service, select it in the first list. The second list should update to reflect its current quality properties, if it has any. Use the buttons on top of the second list to add/edit/remove or set the operational modes a quality is relevant in. In the example seen in Figure 21, the application-level service 'Disinfection' has one safety quality property 'UV-C Overexposure', reflecting the situation where one or more people have been overexposed by the one or more of the MRS robots' lamps during the disinfection. The safety quality property is assigned a risk rating (based on the automotive standard ISO26262 currently, this will be adapted

to the use cases), and can also be given a ‘failure mode’ type (‘Commission’ reflects that the overexposure happened due to the UV-C lamps being on when not intended) and assigned to an operational mode that applies to its service type owner.

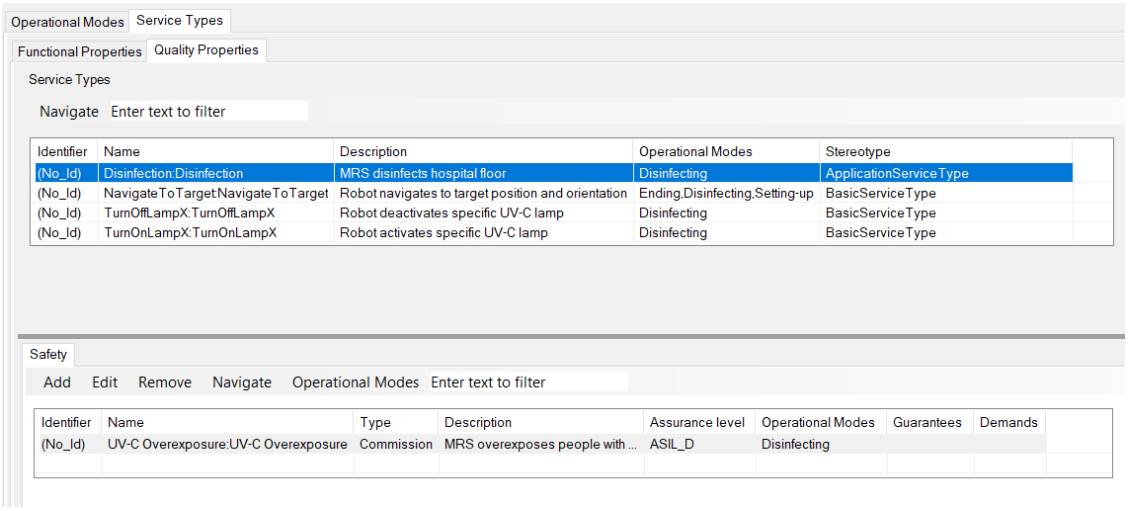
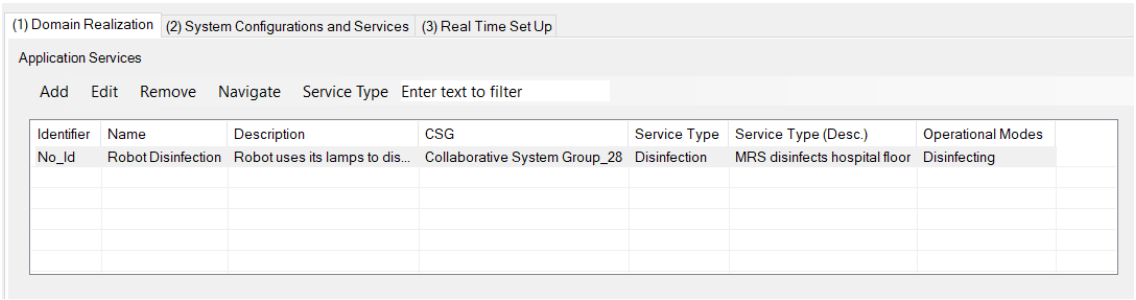


Figure 21 - Specifying CSG Service Type Quality Properties in safeTbox

For each robot modeled, a CS can be specified, using the Smart Menu in the (project) Browser. Once created, use the Smart Menu => Specify Configurations and Services actions to edit the properties of the CS.

The first tab in the CS editing form is the Domain Realization, seen in .



The first section of the tab allows us to specify which, if any, of the application-level services are supported by the given CS. In the case of our example, each robot contributes to the MRS’ application-level service of ‘Disinfection’. Therefore, an application-level service is added here, and its service type is set to match be the Disinfection service, defined previously in the CSG tab (see Figure 20). Use the buttons above the list area to add/edit/remove and set the service type.

The second and third sections of the tab allows us to specify which Provided and Required Services are associated with this CS. In our example, this refers to the individual services a given robot can perform. As seen in Figure 22, a given robot can navigate to a target and turn its lamps on and off. The required services can be specified similarly, and reflect which services a robot depends on to yield its provided services. For example, a robot must perceive its environment to navigate and detect people, therefore its required service is ‘Perception’, as seen in Figure 23.

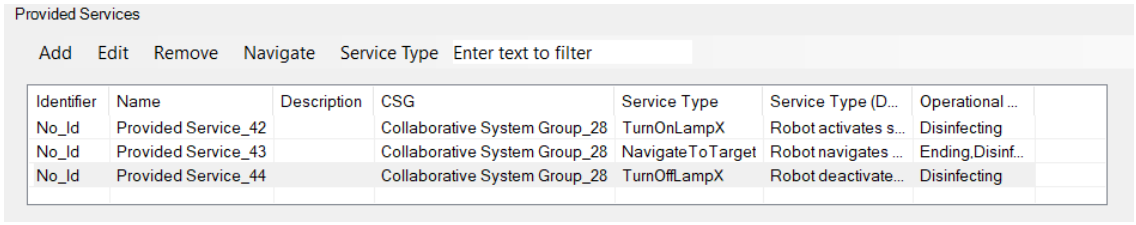


Figure 22 - Specifying CS Provided Services in safeTbox

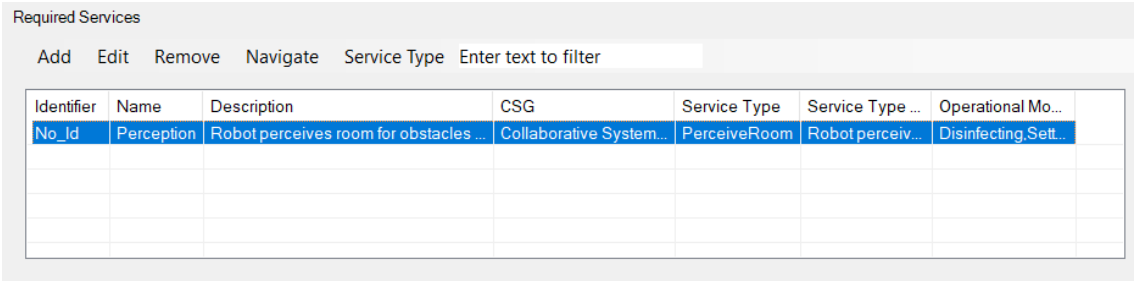


Figure 23 - Specifying CS Required Services in safeTbox

In the next tab, as seen in Figure 24, two configurations are specified for the given CS. In our minimal example, a robot can be considered to either be detecting a person around them, or not.

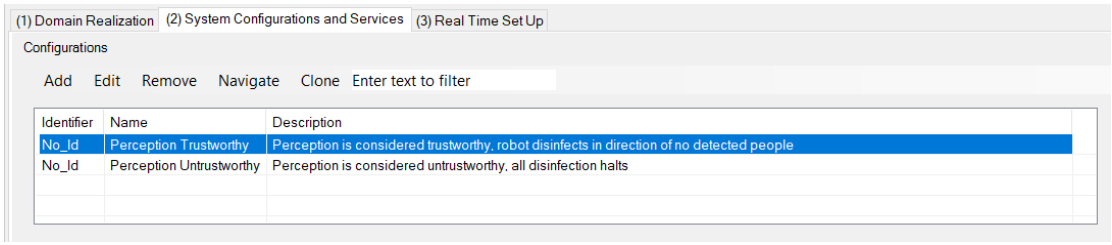


Figure 24 - Specifying CS System Configurations in safeTbox

3.3 SESAME-SPECIFIC EXTENSIONS AND FUNCTIONALITY

Building on prior work with DDIs, safeTbox has also been extended as part of joint work for WP4 and WP7 to import/export EDDIs of its contained models. Simply use the Smart Menu => Import/Export to EDDI File… to parse or serialize an EDDI. This feature uses the Tool Adapter concept, explained later in Section 5.1. (Note that the interface lists DDIs instead of EDDIs, but the models currently supported are indeed updated to the new EDDI metamodels, as described in **D4.2/D5.2: Safety/Security ODE Specification**).

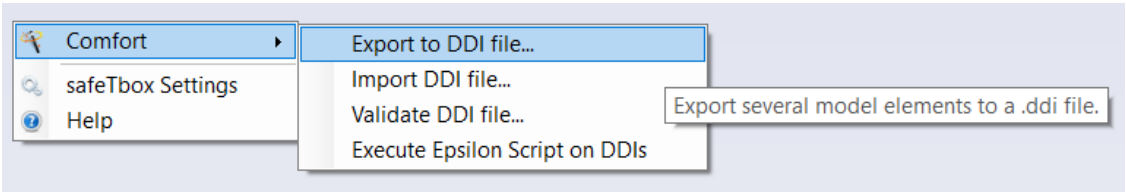


Figure 25 - safeTbox (E)DDI I/O

4. DYMODIA

4.1 WHAT IS DYMODIA

Like HiP-HOPS and safeTbox, the Dymodia tool[7] is a model-based safety analysis tool that supports common analyses like FTA and FMEA. However, unlike the other two tools, Dymodia integrates architectural, behavioural, and failure modelling and analysis as part of a single platform. In this way, it is not reliant on an external modelling package like Matlab Simulink or Enterprise Architect, which limits the other tools to a uni-directional flow of separate steps: model, then analyse. By combining the model and the failure analysis as part of the same package, Dymodia enables tighter integration and allows a more rapid iterative process where design changes can quickly be reflected in the analysis results

Another difference is that unlike the primarily static analyses of HiP-HOPS and safeTbox, Dymodia also supports the use of state machines to model dynamic system behaviour, which can then be linked directly to the architectural and failure models. Standalone dynamic fault trees can also be defined to model failure-related behaviour that does not correspond directly to system architecture elements or system states, e.g. when modelling more complex hazardous scenarios.

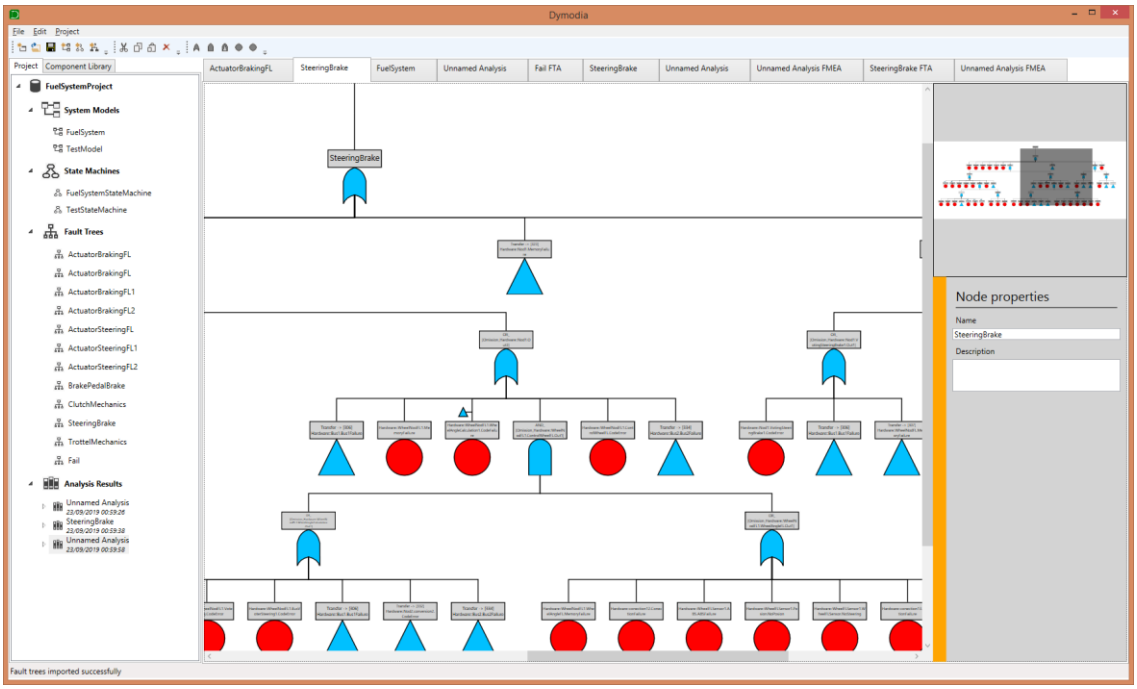


Figure 26 - Dymodia UI

Both FTA and FMEA are supported. FTA results track changes in state, so that not only combinations but also sequences of events that cause failures can be included. The analysis results themselves are also integrated as part of the package, allowing e.g. users to click on a result and be taken to the part of the system model that generated it.

However, Dymodia is still under development and is not as mature as the other tools. It also features a much more complex input format, since it combines several elements in

[7] https://dymodiansystems.com/dymodia/

a package more like an IDE rather than a simple modelling tool. This makes it more difficult to adapt to the ODE.

4.2 HOW IT WORKS

Dymodia follows roughly the same process as the preceding tools; the primary difference is that it all takes place within the same interface. Three types of models are possible to create:

- System models, to describe system architecture;
- State machines, to describe dynamic behaviour;
- Standalone fault trees, to describe failure logic that is not system-specific.

System models consist of components and connections, with ports serving as the interface:

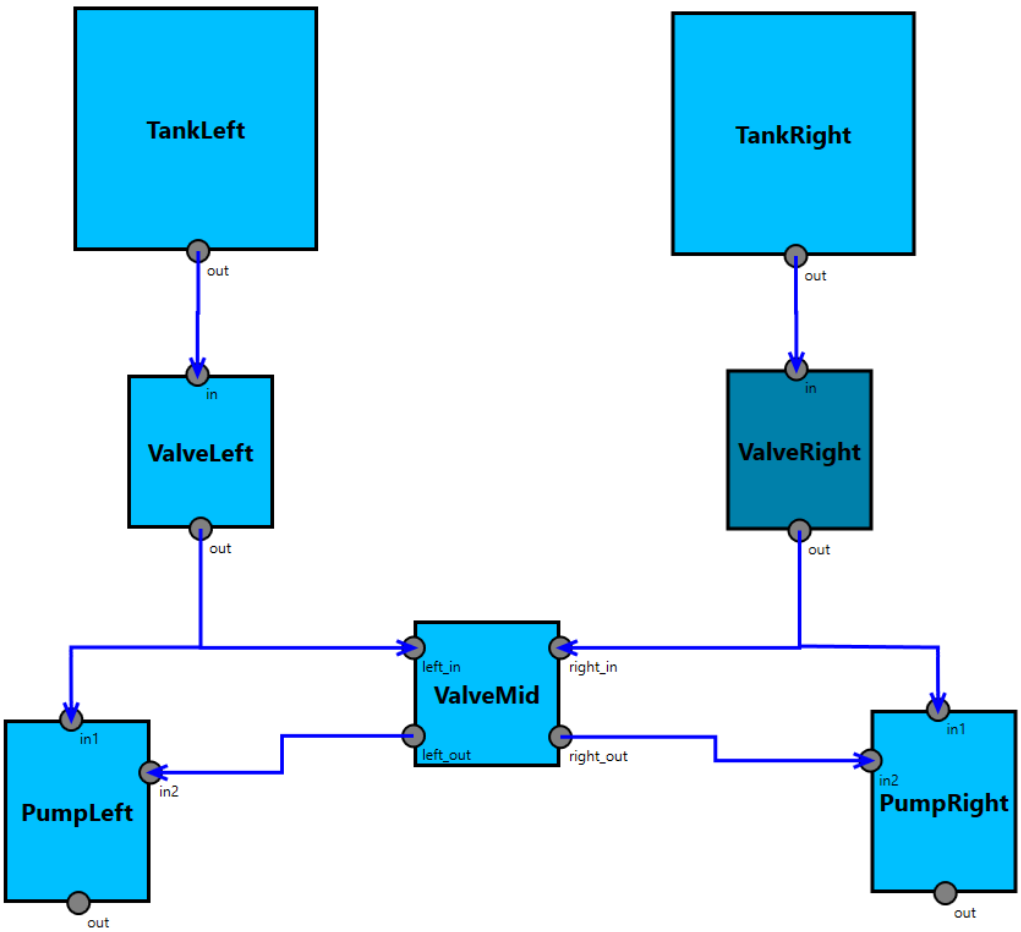


Figure 27 - Example Dymodia system model

As with the other tools, components can be annotated with failure data to describe internal failure modes and deviations at their outputs. A particular grammar is used for the logic, effectively forming a kind of simple domain-specific language.

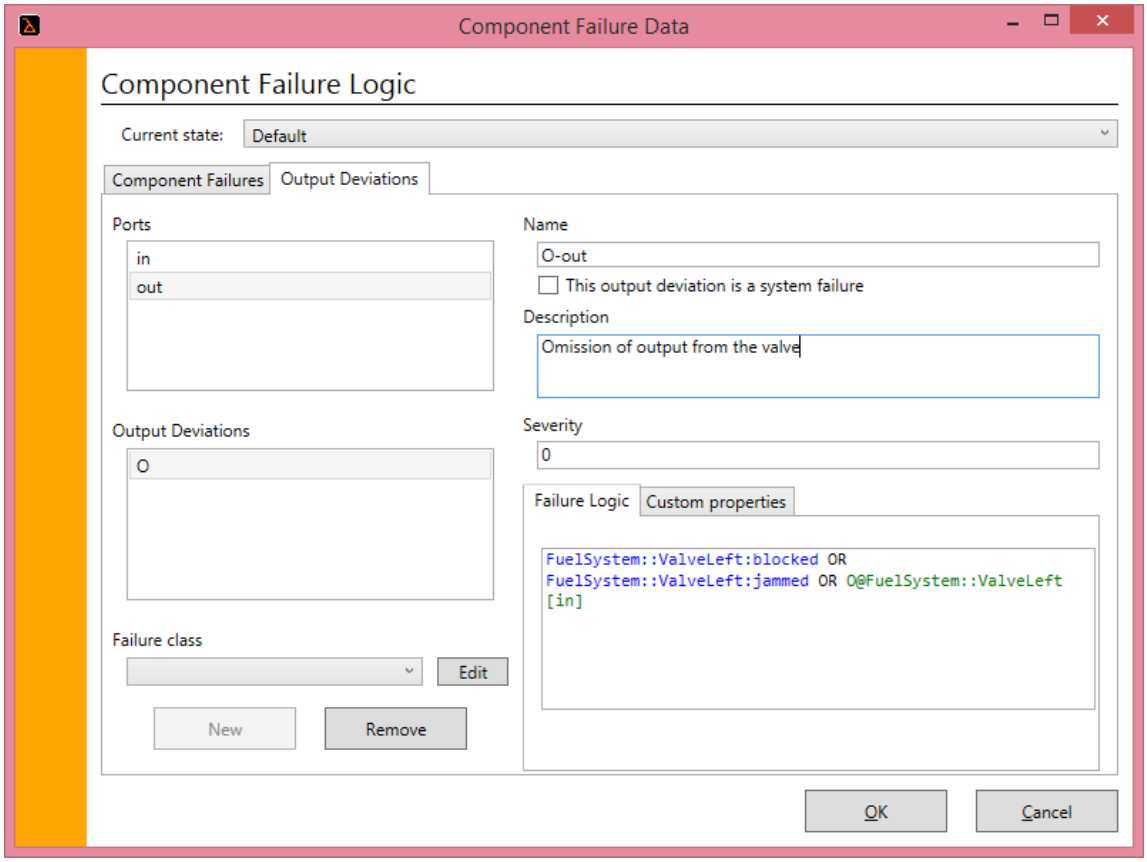


Figure 28 - Failure data in Dymodia

State machines can also be created and linked to system models. When linked in this way, the states defined by the state machine are assumed to be the different modes of operation for the system architecture, thus allowing different failure logic to be defined for each state per component.

Transitions in the state machine can have a variety of triggers, including failure modes and output deviations defined within the system architecture model. They can also be triggered by standalone fault trees or even by transitions in other state machines, thus allowing a form of hierarchical state machine to be created.

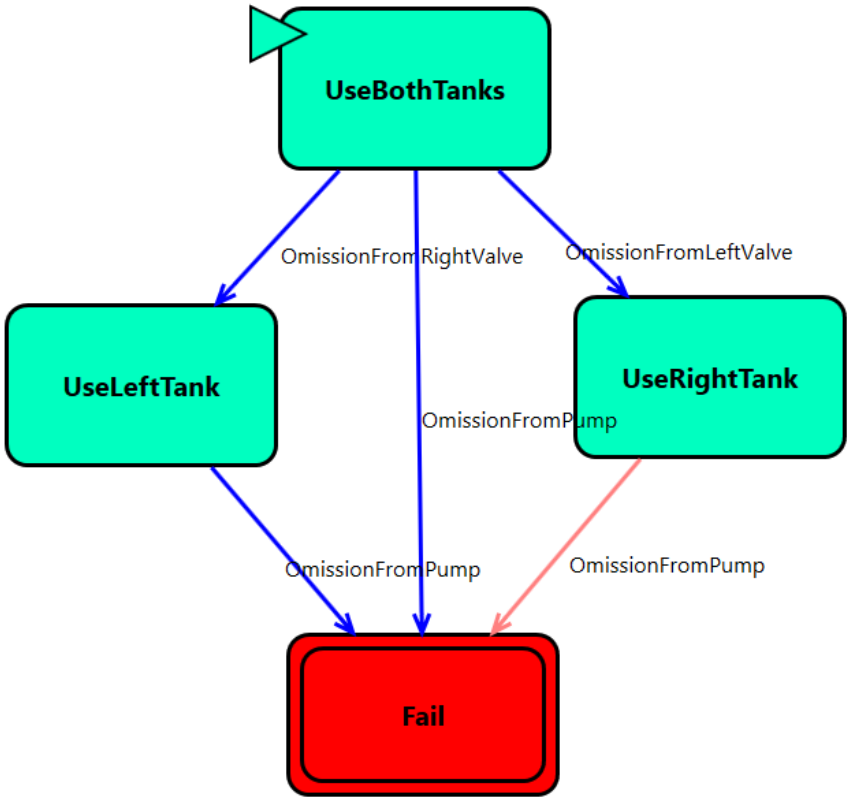
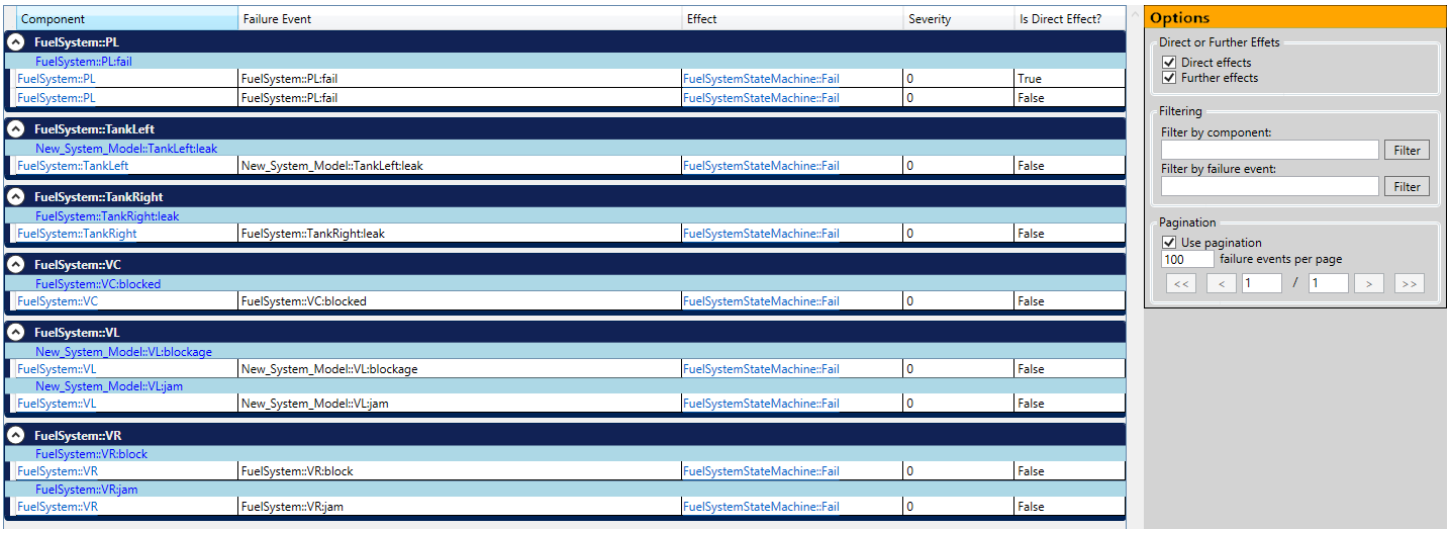


Figure 29 - Dymodia state machine

Once created, the model(s) can be analysed. As with HiP-HOPS, both FTA and FMEA is conducted. Unlike HiP-HOPS, these analyses can incorporate elements from multiple models, including more than one model of each type.



Component	Failure Event	Effect	Severity	Is Direct Effect?
FuelSystem::PL				
FuelSystem::PL:fail				
FuelSystem::PL	FuelSystem::PL:fail	FuelSystemStateMachine::Fail	0	True
FuelSystem::PL	FuelSystem::PL:fail	FuelSystemStateMachine::Fail	0	False
FuelSystem::TankLeft				
New_System_Model::TankLeft:leak				
FuelSystem::TankLeft	New_System_Model::TankLeft:leak	FuelSystemStateMachine::Fail	0	False
FuelSystem::TankRight				
FuelSystem::TankRight:leak				
FuelSystem::TankRight	FuelSystem::TankRight:leak	FuelSystemStateMachine::Fail	0	False
FuelSystem::VC				
FuelSystem::VC:blocked				
FuelSystem::VC	FuelSystem::VC:blocked	FuelSystemStateMachine::Fail	0	False
FuelSystem::VL				
New_System_Model::VL:blockage				
FuelSystem::VL	New_System_Model::VL:blockage	FuelSystemStateMachine::Fail	0	False
New_System_Model::VL:jam				
FuelSystem::VL	New_System_Model::VL:jam	FuelSystemStateMachine::Fail	0	False
FuelSystem::VR				
FuelSystem::VR:block				
FuelSystem::VR	FuelSystem::VR:block	FuelSystemStateMachine::Fail	0	False
FuelSystem::VR:jam				
FuelSystem::VR	FuelSystem::VR:jam	FuelSystemStateMachine::Fail	0	False

Figure 30 - Dymodia FMEA output

4.3 SESAME-SPECIFIC EXTENSIONS AND FUNCTIONALITY

As it is the least mature tool and still in development, it is harder to use Dymodia for the purpose of creating ODE-compliant EDDI models. However, its capabilities — particularly for dynamic analysis with integrated state machines — makes it an attractive prospect.

Furthermore, because Dymodia is inspired by similar preceding tools like HiP-HOPS, it too uses a metamodel that is broadly compatible with ODE. The intention is therefore to try to generate a converter for Dymodia models, allowing conversion of a Dymodia file to an ODE model.

The reverse is unlikely to be possible, however, because of the fact that Dymodia is an integrated modelling and analysis tool. The models imported by HiP-HOPS and safeTbox do not need to know about e.g. the position of each component or other display information (like colours or images); this is instead handled by the external modelling tool.

This same extra detail makes it more challenging to extract the relevant information out of a Dymodia file to generate a corresponding ODE model.

5. MODEL CONVERTERS

Two model converters are available for conversion of other models/file formats to ODE models.

5.1 COMMON TOOL ADAPTER

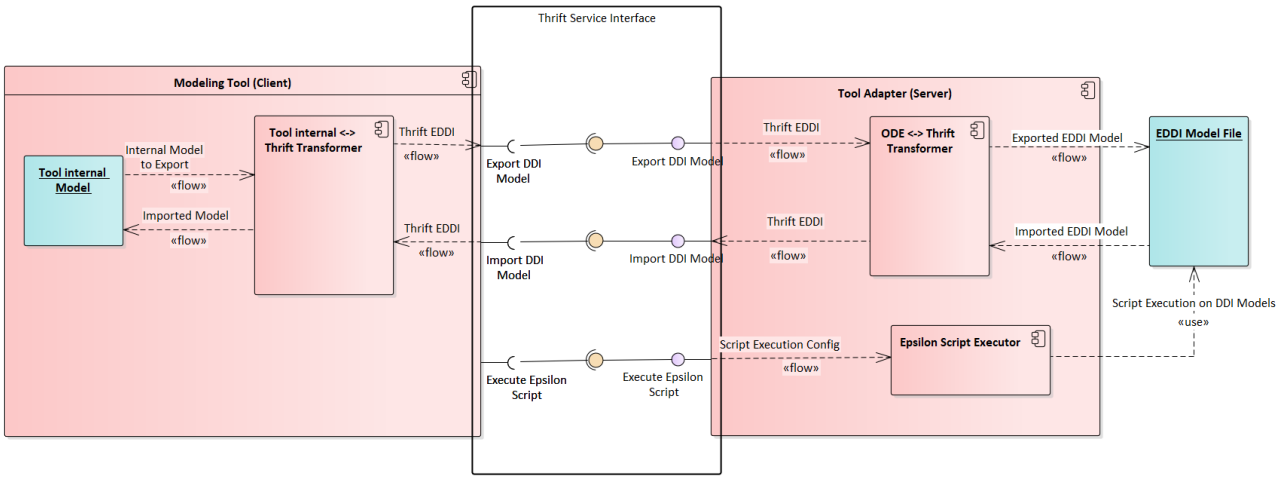


Figure 31 - Tool Adapter Service Interface

The Tool Adapter is a service providing software component that allows importing and exporting DDI models that conform to the ODE metamodel into or from an arbitrary software (e.g. modelling software). Additionally, it provides a service for executing Epsilon scripts on existing DDI models. The standardized, language-agnostic and generic service interface is realized using Apache Thrift. Figure 31 shows the service interface provided by the Tool adapter and how it is used by modelling tools. This chapter describes the service interface definition and usage in a modelling tool. For further information about the Tool Adapter and its architecture, please refer to SESAME project deliverable **D5.4: Tailorability of EDDIs** (Section 3.1).

Apache Thrift is used as an intermediate service interface provider. The open-source framework comes with its own Interface Definition Language (IDL). Using the IDL allows us to define service interfaces and the data types that shall be exchanged through the services in a language independent manner. The Thrift compiler is then used to generate language-specific source code from the IDL definitions. This source code (e.g. Java, C#, C++, etc.) can then be used in server- or client-side applications. For instance, the Tool Adapter (Server) and modelling tools (Client) use Thrift to establish a standardized service provision and consumption. Sending and receiving models to and from the Tool Adapter needs a transformation between the tool internal model representation and the Thrift data types on the client side (see Section 5.1.2 for further information).

Figure 32 provides a brief overview on how generated code is integrated into client and server software solutions. The ServiceInterface.thrift element represents the service and data type definition using IDL. On each side, the generated Code is integrated and the

language-specific Thrift library is used to invoke the client or server-side specific request and response procedures.

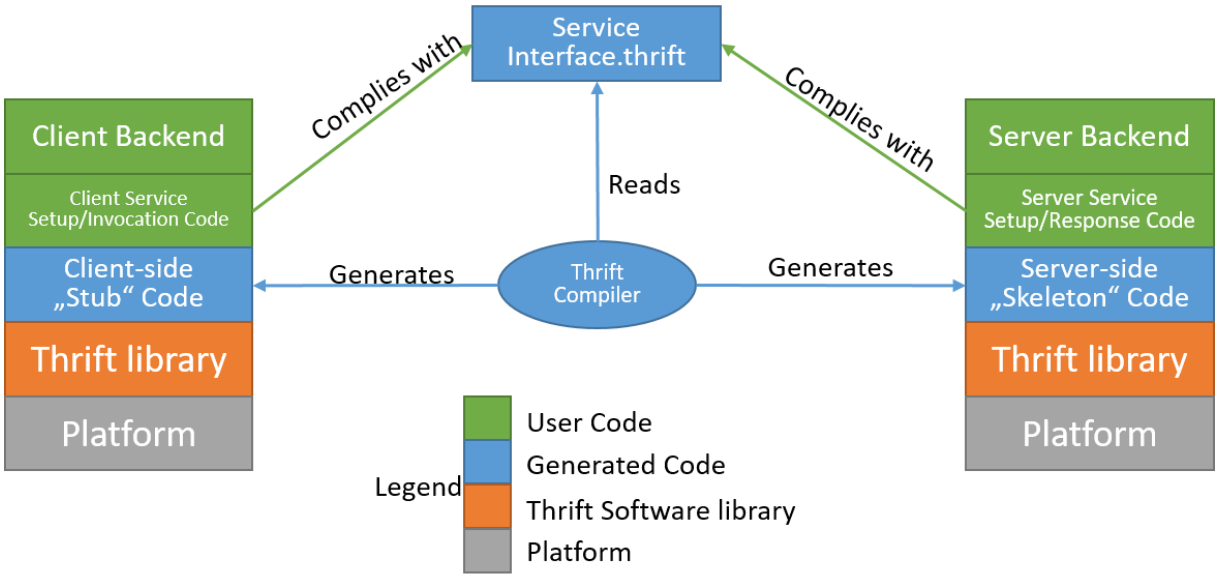


Figure 32 - Overview of Apache Thrift (DEIS D4.2)

As mentioned above, within the Tool Adapter, Apache Thrift is used to define the service interface for importing/exporting DDI models into/from modelling tools and for executing Epsilon scripts on DDI models. Furthermore, all ODE metamodel elements and their relationships are defined as Apache Thrift datatypes. Thus, generating language specific code out of the Thrift contract defined in IDL means the ODE data structure including the server and client code for providing and consuming mentioned services is available in the specific language and is ready to be used in software that shall communicate with each other.

5.1.1 Preparation of Tool Adapter (server side)

This chapter provides a step-by-step guide to prepare and configure the Tool Adapter, and describes how to launch the adapter. There are two possibilities to start the Tool Adapter. It can be either launched from the command line or within the Eclipse/Epsilon IDE. Both ways are explained in the following sections.

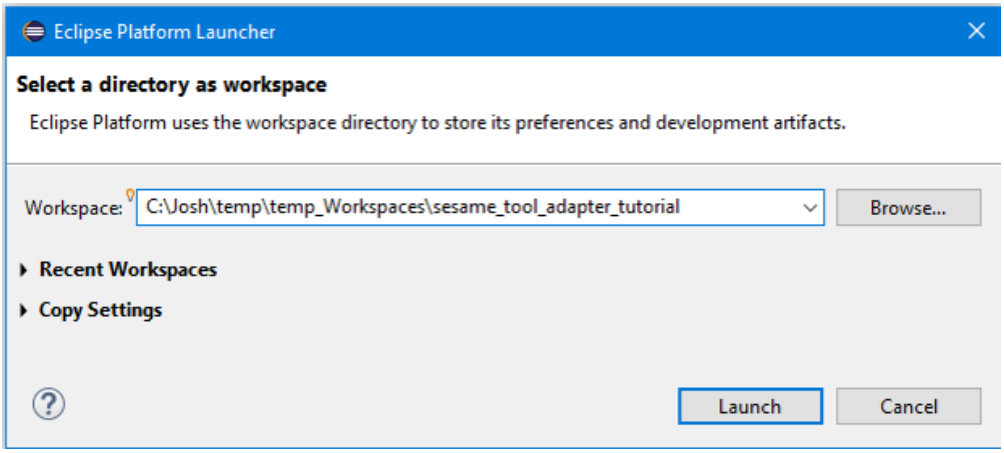
IMPORTANT NOTE: Currently there is a strict dependency to an older insecure version of Apache Log4j given by the older Thrift version we also depend on. Please make sure you do not use the Tool Adapter in an insecure environment (e.g. providing the services to the public internet).

5.1.1.1 Downloading IDE and importing project

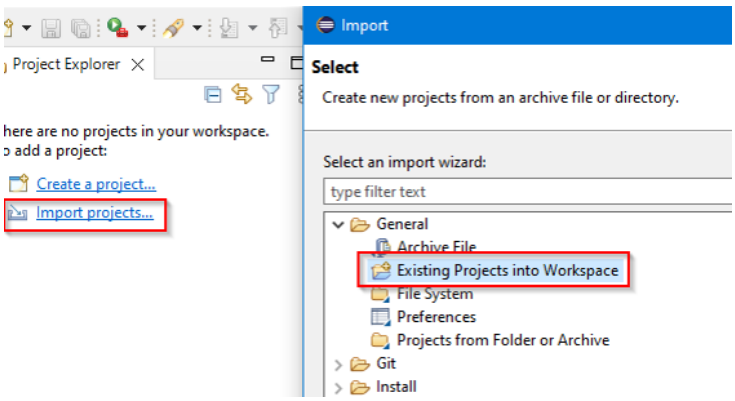
The current ODE and Tool Adapter were developed using the Eclipse Epsilon IDE version 2.4 (current version as of writing this document). To download and install the correct version click here and follow instructions on the Eclipse homepage.

After having Eclipse installed, download the projects of the ODE metamodel and Tool Adapter.

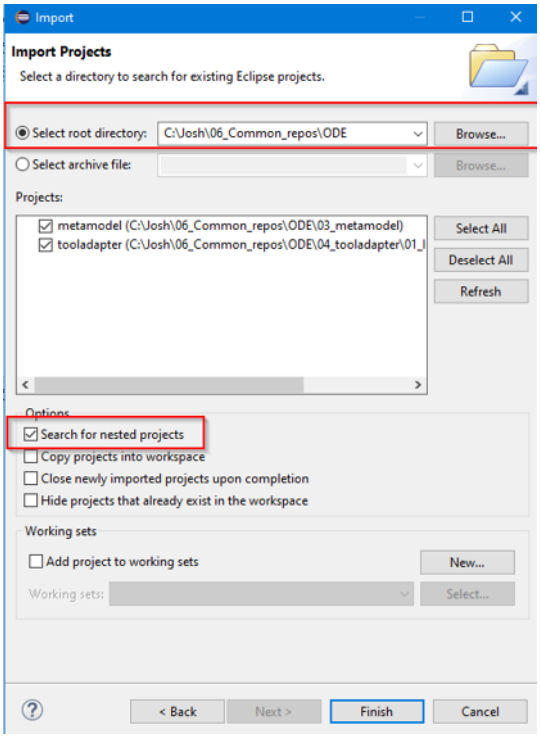
Create a new workspace, which shall contain both projects.



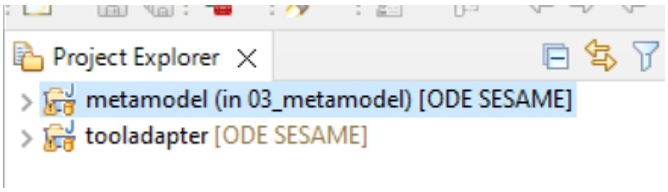
Within the empty workspace, import the ODE metamodel and Tool Adapter projects. Click on “Import projects” and choose “Existing Projects into Workspace”.



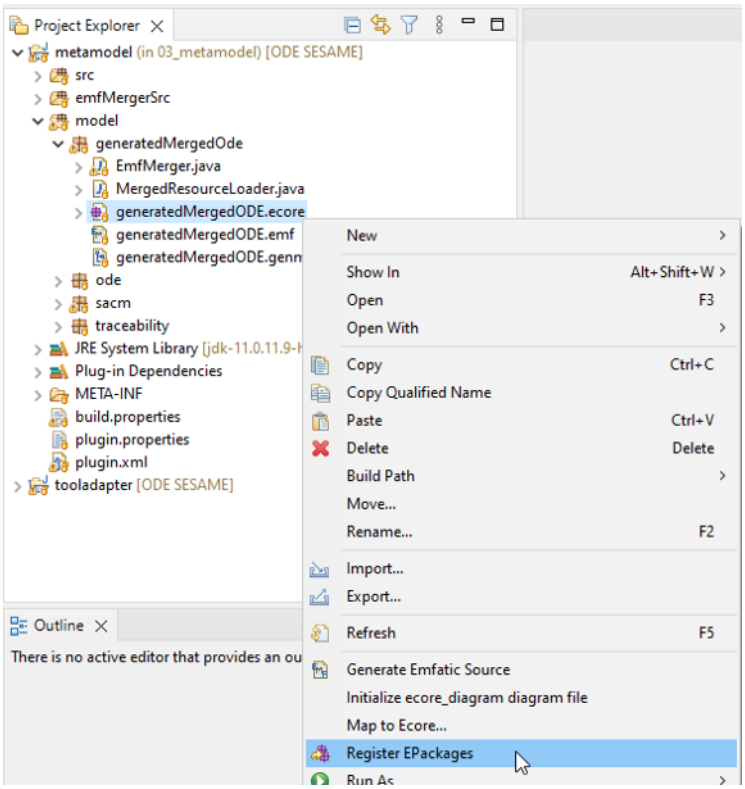
In the next dialog window, choose the root directory where you have downloaded the projects and check the check box “Search for nested projects”. Make sure you have selected the projects you want to import (here metamodel and tooladapter) and click ‘Finish’.



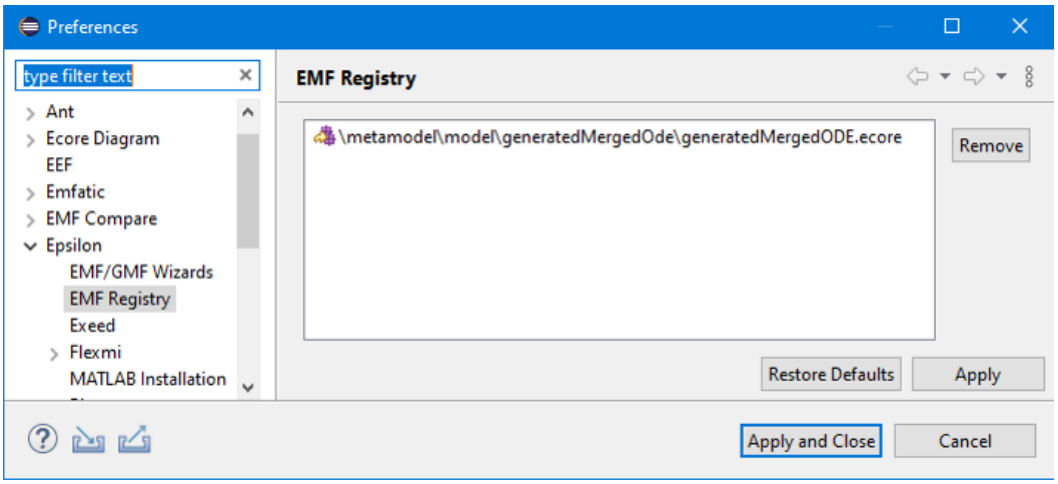
Afterwards the projects should be imported and visible within the Project Explorer:



HINT: If Epsilon scripts shall be executed within Eclipse manually or the Tool Adapter shall be started from within Eclipse, the Ecore package of the metamodel has to be registered. Therefore, go to "metamodel → model → generatedMergedOde" and right-click on *generatedMergedODE.ecore* file and choose *Register EPackages*.



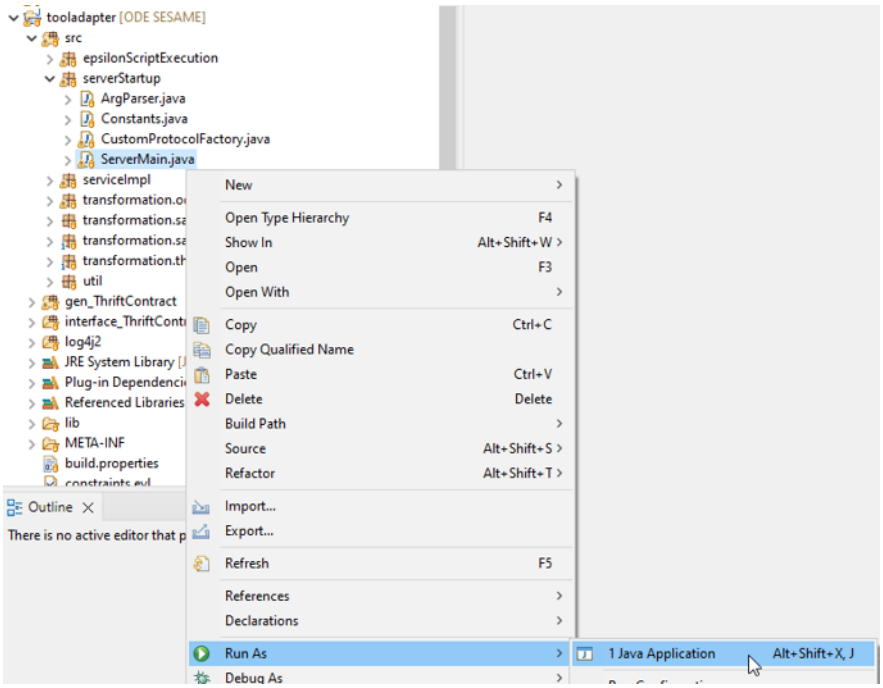
Verify the registration by going into the preferences dialog (Window → Preferences) and check under Epsilon → EMF Registry that your packages are registered.



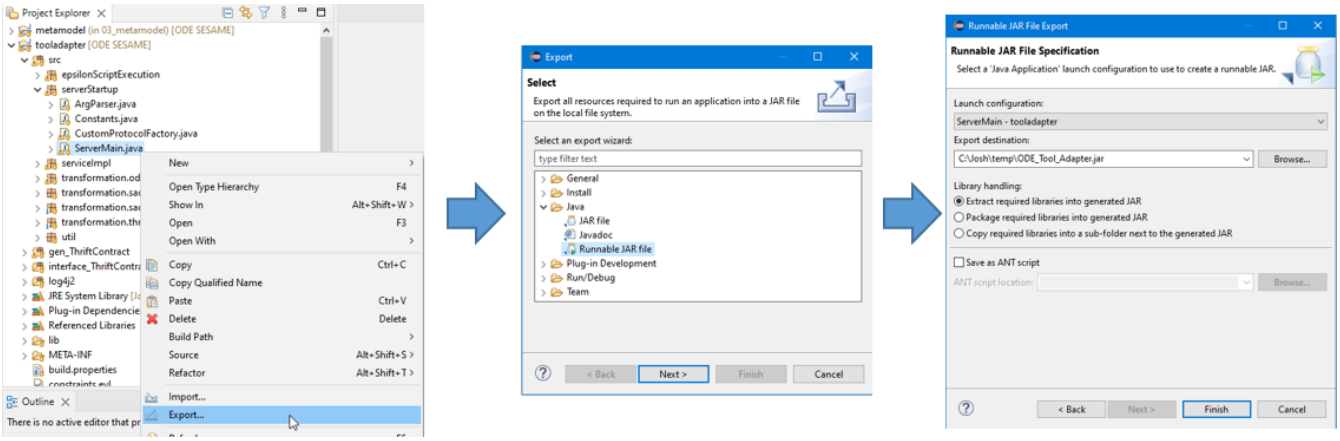
5.1.1.2 Creating an executable .jar file for starting Tool Adapter from command line

In order to start the Tool Adapter server independently from the Eclipse IDE, an executable .jar file has to be created. Therefore the ServerMain.java has to be started once to have the start configuration defined within Eclipse. Therefore, go to

"tooladapter → src → serverStartup" and right-click the *ServerMain.java* file and choose "Run As → Java Application".



Afterwards, the .jar can be built by right-clicking the *ServerMain.java* again and choose "Export … → Runnable JAR file". In the Runnable JAR File Export dialog, choose the launch configuration and set an export destination, where the .jar file shall be saved to.



5.1.1.3 Starting Tool Adapter with arguments

In the current version of the Tool Adapter, only the port for providing the services can be configured, when starting the Tool Adapter with command line.

For starting the Tool Adapter from command line, use the java command with the –jar option.

```
java -jar <path to Tool Adapter jar>
```

By default, this will start the Tool Adapter listening on port **1339**.

If you want to change the port the server shall listen on, provide the port number with the option "–server-port":

```
java -jar <path to Tool Adapter jar> -server-port <port number>
```

Here is an example that launches the ODE_Tool_Adapter.jar located in the current directory, which listens on port 1234:

```
java -jar .\ODE_Tool_Adapter.jar -server-port 1234
```

When starting the Tool Adapter within Eclipse, the default port is always used. To change the default port, you can adapt the value within the "tooladapter → src → serverStartup → ArgParser.java" file:

```
@Option(names = {"-server-port"}, description = "Port of the ODE Tool Adapter Server. Default: 1339")
private int serverPort = 1339;
```

5.1.2 Preparation of modelling tool (-adapter) (client side)

This chapter provides step by step instructions on how to:

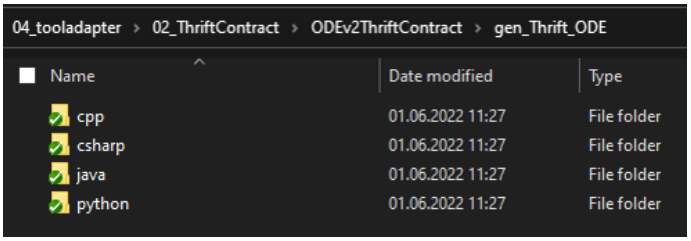
- generate code with Thrift for service infrastructure and model data types defined in IDL;
- configure code generation for new language or alternative namespaces;
- integrate generated code and Thrift library into client application;
- use the Thrift library to call the service provided by the Tool Adapter.

IMPORTANT NOTE: Currently, the Tool Adapter only supports Thrift in its older version **0.11.0**.

5.1.2.1 Generating service and model code

We currently provide generated code to integrate into Java, C#, C++ and Python projects. You can find these code files within the Tool Adapter directory:

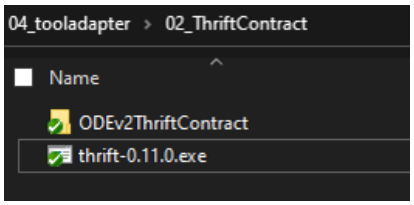
04_tooladapter\02_ThriftContract\ODEv2ThriftContract\gen_Thrift_ODE



NOTE: These generated source code files come with predefined namespaces. The namespace can either be applied in your project and/or manually be adapted to your namespaces or you can follow the next steps to generate the files from scratch with customized namespaces.

The Thrift compiler for Windows (*thrift-0.11.0.exe*), used for generating the code, is located under:

04_tooladapter\02_ThriftContract



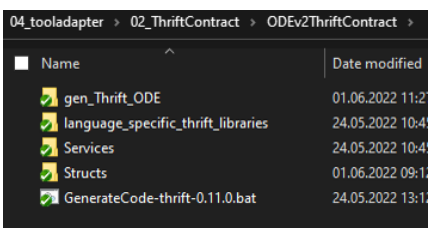
For usage on other operating systems or in general for compiling the Thrift compiler, you can find the repository and its version 0.11.0 branch under https://archive.apache.org/dist/thrift/0.11.0/

Following the link below, there is a table listing all coding languages and the features supported by Thrift v. 0.11.0.

https://github.com/apache/thrift/blob/0.11.0/LANGUAGES.md

In the *ODEv2ThriftContract* directory lies a batch file (*GenerateCode-thrift-0.11.0.bat*) for configuring and executing the code generation process. **Running this script will:**

- fetch all service and data type definition files located within the *Services* and *Structs* directories in the *ODEv2ThriftContract* directory:



- merge all definitions into one file
- add namespace declarations for each coding language the script if configured for
- generate code for each configured language using the *thrift-0.11.0.exe*

5.1.3 Configuring code generation

As mentioned before, the code generation can be further configured, for instance to change namespace definition for already supported languages or adding a new language the Thrift shall generate code for. This section explains how the batch file is structured and what has to be done to adapt it.

Open *\04_tooladapter\02_ThriftContract\ODEv2ThriftContract\GenerateCode-thrift-0.11.0.bat* in a text editor. In the following all sections within the batch file and the optional configuration steps are explained.

5.1.3.1 Namespace definition section

```
1  @REM namespace definition
2  @set csharpNamespace=namespace csharp STB_Modeling_Techniques.DEISProject.ODEDataModel.ThriftContract
3  @set javaNamespace=namespace java thriftContract
4  @set cppNamespace=namespace cpp thriftContract
5  @set pyNamespace=namespace py thriftContract
```

The first section defines variables that contain the Thrift IDL namespace expression for each configured language, where namespaces are supported.

The pattern for those definitions is as follows:

```
namespace <Thrift language keyword> <name of namespace>
```

For instance: If you want change the namespace for your C# application, you can change the value behind the *csharp* keyword.

If you want to generate code for a different language, which supports namespaces, have a look at the documentation of Thrift IDL to find out the respective keyword and add a namespace definition statement as shown in the screenshot.

5.1.3.2 Environmental variable definition and merging section

```
7  @REM environmental variable definition
8  @set mergedFileName=mergedDDIThriftContract.thrift
9  @set mergedFilePath=%basePath%%mergedFileName%
10 @set basePath=%~dp0
11 @set thriftCompiler=..\thrift-0.11.0.exe
12
13 @echo.
14
15 @echo Merge all files into one thrift file.
16 @copy %basePath%\Structs\*.thrift_enums+%basePath%\Structs\*.thrift_refs+%basePath%\Structs\*.thrift %mergedFileName%
17
```

This section defines further variables for execution and merges the content of the Thrift data type definitions into one file (defined in *mergedFileName* variable).

No further changes have to be applied for correct usage.

5.1.3.3 Adding namespaces to merged file section

```
18  @REM add namespaces to merged file
19  @set tempFile=%basePath%tempFile.txt
20  @echo %csharpNamespace% > %tempFile%
21  @echo %javaNamespace% >> %tempFile%
22  @echo %cppNamespace% >> %tempFile%
23  @echo %pyNamespace% >> %tempFile%
24  @type %mergedFilePath% >> %tempFile%
25  @type %tempFile% > %mergedFilePath%
26  @del %tempFile%
```

This section first collects each Thrift IDL namespace statement within a temporary file and afterwards puts those namespace statements at the beginning of the file that contained the merged data type definition.

If you have defined a new namespace statement for a new language you want to generate code for, please make sure to add an additional line here, which adds the new namespace to the merged file.

5.1.3.4 Append service definitions to merged file section

```
28  @REM append services to mergedFilePath
29  for %%f in (%basePath%\Services\*.thrift) do (
30      TYPE %%f >> %mergedFilePath%
31  )
```

This section simply appends the service Thrift definitions to the merged file.

No further changes have to be applied for correct usage.

5.1.3.5 Prepare folder structure section

```
33  @REM prepare folder structure
34  @rmdir "./gen_Thrift_ODE" /s /q
35  @mkdir "gen_Thrift_ODE/csharp"
36  @mkdir "gen_Thrift_ODE/java"
37  @mkdir "gen_Thrift_ODE/cpp"
38  @mkdir "gen_Thrift_ODE/python"
39
```

This section re-creates the necessary folder structure, where the generated code files will be saved to. If you added an additional supported language, please make sure you also add a new line for creating a new directory here respectively.

5.1.3.6 Generate code section

```
@echo.
@echo Generate C# files
%thriftCompiler% -r -out gen_Thrift_ODE/csharp --gen csharp %mergedFilePath%

@echo.
@echo Generate Java files
%thriftCompiler% -r -out gen_Thrift_ODE/java --gen java %mergedFilePath%

@echo.
@echo Generate C++ files
%thriftCompiler% -r -out gen_Thrift_ODE/cpp --gen cpp:pure_enums %mergedFilePath%

@echo.
@echo Generate Python files
%thriftCompiler% -r -out gen_Thrift_ODE/python --gen py %mergedFilePath%

@echo.
@echo Delete merged thrift file.
@del %mergedFilePath%

@pause
```

In the last section, for each configured language, the Thrift compiler is called with the options

- where to save generated files (*-out gen_Thrift_ODE/<language>*)
- what language the code shall be generated for (*--gen <language keyword>*)
- where the thrift definition is defined (*%mergedFilePath%*)

Add a new compiler call here for each new language.

5.1.3.7 Include necessary files in client application

Next, import generated files into the client application. Generated files can be found in the directory:

\04_tooladapter\02_ThriftContract\ODEv2ThriftContract\gen_Thrift_ODE.

After generated files are imported the Thrift library has to be imported as well. Libraries for Java C# (.Net Framework 4.5) and Python the libraries can be found under *\04_tooladapter\02_ThriftContract\ODEv2ThriftContract\language_specific_thrift_libraries.*

For all other languages, you may find an entry on the Thrift library webpage[8] for integrating the library. Make sure that you get the library in the older version **0.11.0**.

[8] https://thrift.apache.org/lib/

Alternatively, you can build the library from source code by following the link below, navigate into directory of the coding language and follow instructions in the readme file on how to build/install the necessary library:

https://github.com/apache/thrift/tree/0.11.0/lib

5.1.4 Use Thrift library for invoking provided services of Tool Adapter

The following example shows how to set up the connection to the Tool Adapter and invoke the services in C#. For other languages, please refer to the tutorials provided within the Thrift repository for version 0.11.0: https://github.com/apache/thrift/tree/0.11.0/tutorial. This section is divided into parts, where each part describes the essential implementation tasks.

5.1.4.1 Implement transformation routines

The Tool Adapter services for importing and exporting a DDI model require the respective model be in the ODE format using the generated Thrift representation. Therefore, exporting a model as a DDI model requires the model to be transformed from modeling tool internal representation into the ODE Thrift representation. Thus, when importing a DDI model, the Thrift representation also has to be transformed back into the modeling tool representation.

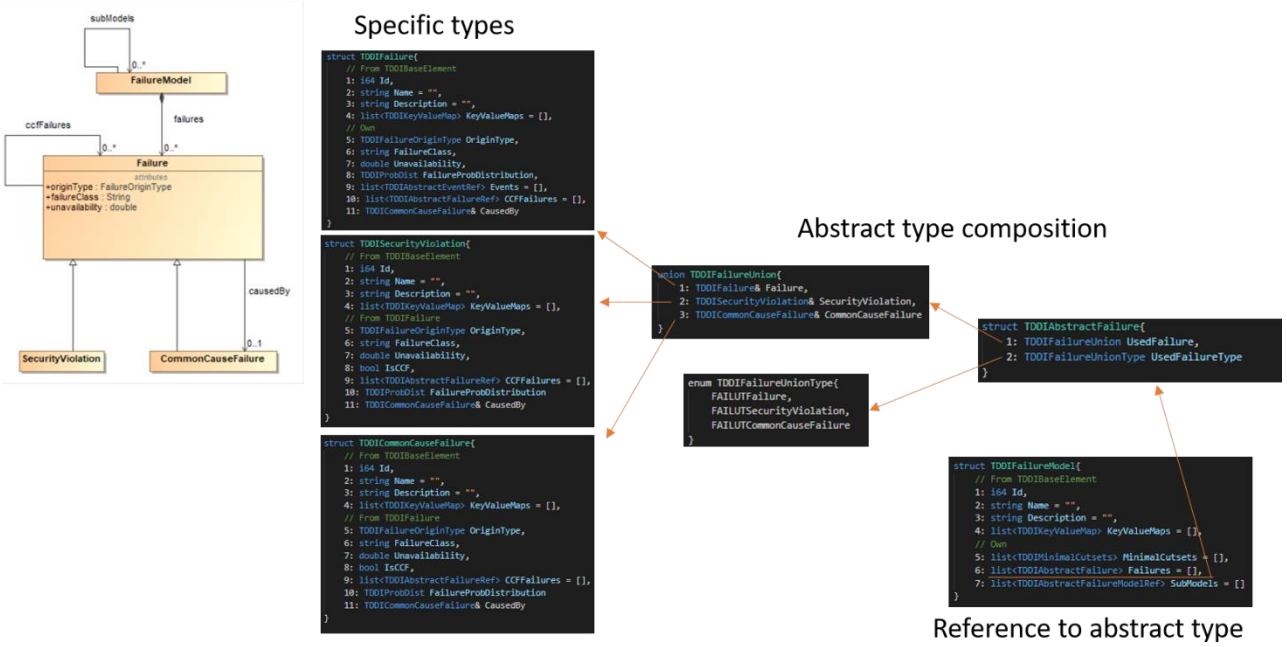
Unfortunately, Thrift (at least version 0.11.0) does not allow to easily define inheritance between data types within IDL. This prevents one from implementing a simple one-to-one mapping from modeling tool internal representation to the Thrift representation. As a result of this, a workaround is used to specify inheritance by using Thrift's *union* types together with enums to hold a more specific type within an abstract representation of the parent data type. A union type is like a struct and can have multiple fields, but only one field can be set at a time.

On the left side of the picture below, an excerpt of the ODE metamodel is depicted, which shows the *Failure* type with its sub types and the relation between *FailureModel* and *Failure*. On the right side, the specification and relation between those elements is defined in Thrift IDL.

The specification of each instantiable *Failure* type (Failure, SecurityViolation and CommonCauseFailure) together with all inherited and own fields are defined in its own *struct* within Thrift IDL; namely *TDDIFailure*, *TDDISecurityViolation* and *TDDICommonCauseFailure*.

A *union* type (*TDDIFailureUnion*) is defined, which has a field for each concrete *Failure* type. Additionally, an enum is defined, which is used for identifying which of the union's fields is currently set.

A *FailureModel* can refer to multiple *Failures*, given by the inheritance, where a failure could be of all mentioned concrete types. Therefore, within the Thrift definition the *TDDIFailureModel* has a field defined as a list of *TDDIAbstractFailure* elements.



For example, to transform a security violation from the modeling tool internal representation to the Thrift representation, the following steps have to be taken, when implementing the transformation routine.

1. Instantiate a new object of type *TDDISecurityViolation* and fill the fields.

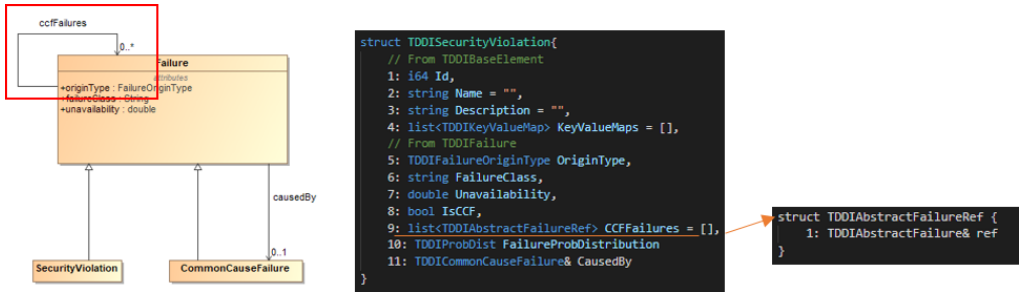
2. Instantiate a new object of type *TDDIFailureUnion* and set the field *SecurityViolation* to the previous instantiated *TDDISecurityViolation* object

3. Create an instance of *TDDIAbstractFailure* and fill the *UsedFailure* field with the previously created *TDDIFailureUnion* object and set the field *UsedFailureType* to the enum value *TDDIFailureUnionType.FAILUTSecurityViolation*

4. Finally create the *TDDIFailureModel* object and add the *TDDIAbstractFailure* object to the *Failures* list field.

In some languages such as C and C++, it is explicitly differentiated between compositions and simple association relationships between types. Thrift also provides the option to declare whether a relationship is of the type of a simple association or a composition. To let Thrift generate C++ code of a struct that shall have only a reference pointer to a single element of another type, within the IDL definition, the ampersand (&) character is used as suffix for the field type. In the above overview, the *TDDIFailureUnion* union type makes use of this suffix for each field.

However, this annotation can only be added to field types which represent a single element. Thus, for lists that should contain reference pointers there is a workaround, in that a separate wrapper struct is defined, which wraps the reference and can be instantiated to fill the list. The picture below shows the Thrift representation of the self-reference of *Failure* as *ccfFailures*. These compositions for realizing lists of reference pointers have also been considered during the implementation of the transformation routines.



5.1.4.2 Invoking Tool Adapter services with Thrift client

For invoking the services, first a Thrift client has to be set up. Please refer to the tutorial section of the Thrift v.0.11.0 repository branch, to find out how to configure the Thrift client in your language.

An example of how a Thrift client can be instantiated in C# is shown in the picture below:

```
private static TTransport transportSocket;
public static DDIService.Client DDIService { get; private set; }
private const string serverHost = "localhost";
private const int serverPortDefault = 1339;

private static void InitializeThriftClient(int serverPort)
{
    if (transportSocket != null)
        transportSocket.Close();
    int elapsedTries = 0;
    while (elapsedTries <= SERVER_TIMEOUT_SECONDS)
    {
        try
        {
            transportSocket = new TSocket(serverHost, serverPort);
            transportSocket.Open();

            TProtocol protocol = new TBinaryProtocol(transportSocket);
            protocol.RecursionLimit = 99999;
            DDIService = new DDIService.Client(protocol);

            return;
        }
        catch (SocketException)
        {
            if (elapsedTries < SERVER_TIMEOUT_SECONDS)
            {
                WaitNSeconds(1);
                elapsedTries++;
            }
            else
            {
                throw;
            }
        }
    }
}
```

Afterwards you can invoke the server call by calling the method representing the service on the created thrift client. The picture below depicts the Thrift service declarations:

```
service DDIService{
    void ExportModelToDDIFile(1: string DDIFilePath,
                              2: TDDIDDIPackage DDIPackage) throws (1: TDDIAbstractEpsilonScriptExecutionException EpsilonScriptExecutionException),

    TDDIDDIPackage ImportDDIModel(1: string DDIFilePath) throws (1: TDDIAbstractEpsilonScriptExecutionException EpsilonScriptExecutionException),

    TDDIValidationResult ValidateDDI(1: string DDIFilePath, 2: string EvlFilePath),

    TDDIDDIPackage ExecuteEpsilonSriptsOnDDIFile(1: list<TDDIEpsilonScriptExecutionConfig> EpsilonScriptExecutionConfigs,
                                                 2: bool BackupDDIFile,
                                                 3: bool ReturnDDIPackage) throws (1: TDDIAbstractEpsilonScriptExecutionException EpsilonScriptExecutionException)
}
```

ExportModelToDDIFile: Is used to export a model from within a modeling tool to a DDI model file

- Parameter 1: The file path where the model file shall be saved to (string)
- Parameter 2: The model to be exported in the Thrift DDI representation, where TDDIDDIPackage is the root element of the whole model.

ImportDDIModel: Is used to import a DDI model file into modeling tool

- Parameter 1: The file path pointing to the DDI model file (string)
- Return value: The DDI model in its Thrift representation (TDDIDDIPackage)

ValidateDDI: Is used to run an Epsilon model validation script on existing DDI model

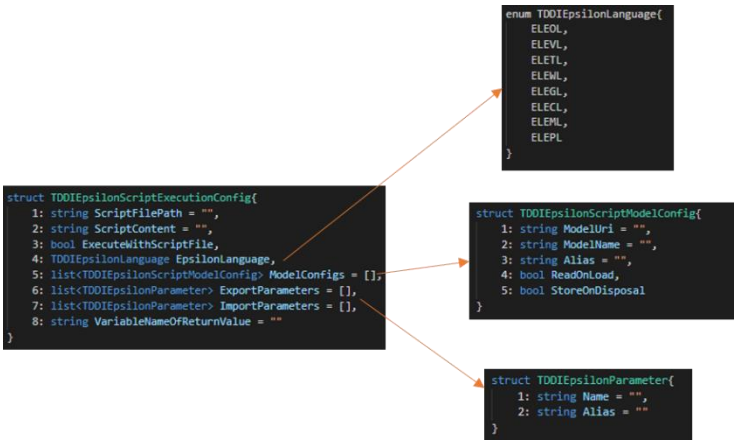
- Parameter 1: File path to existing DDI model (string)
- Parameter 2: File path to validation script (string)
- Return value: The validation result which contains information about which validation rules have not been fulfilled by the model (TDDIValidationResult

```
struct TDDIValidationViolationMessage{
    1: string Context = "",
    2: string Message = ""
}

struct TDDIValidationResult{
    1: bool ValidationViolationOccurred = false,
    2: list<TDDIValidationViolationMessage> ValidationViolationMessages = []
}
```

ExecuteEpsilonScriptOnDDIFile: Is used to run an arbitrary epsilon script on one or multiple DDI models.

- Parameter 1: The configuration for executing the script on the model



  - ScriptFilePath: Path to script file as string. This is set, when script is saved as file

- ScriptContent: Content of actual script as string. This is set, when script content is passed as parameter
- ExecuteWithScriptFile: Boolean value to indicate if script is provided as a file or not (True, when 1. field is set and False, when 2. field is set).
- EpsilonLanguage: Epsilon language of script that shall be executed as enum (TDDIEpsilonLanguage) value.
- ModelConfigs: Configurations of DDI models, the script shall be executed on.
  - ModelUri: Path to DDI model
  - ModelName: Identifier on how to access model within script
  - Alias: Alternative identifier for model within script
  - ReadOnLoad: If set to True, when the script is executed, the current values of the model are loaded into memory and can be read in the script. If set to False, then the script considers the model to be empty when it first executes.
  - StoreOnDisposal: If set to True, when the script completes execution, any changes to the model are written back to the model file. If set to False, any changes to the model are discarded when the script completes execution.
- ExportParameters: Parameters/variables names, which shall be exported from the executed script.
- ImportParameters: Parameters/variable names, which shall be imported to the executed script.
- VariableNameOfReturnValue: Name of the variable that shall be returned by the script.

- Parameter 2: Boolean value to indicate whether DDI model shall be backed up before executing the script on it.
- Parameter 3: Boolean value to indicate whether DDI model shall be returned by service or not.
- Return value: The TDDIDDIPackage returned by the script. If value of parameter 3 *ReturnDDIPackage* is *false* an empty *TDDIDDIPackage* is returned.

5.2 ODE MODEL CONVERTER

For the tools described in sections 2-4 to support ODE models, converters are required. In addition to the common tool adapter, a standalone model converter has been developed to support conversion from HiP-HOPS (with the intention of expanding it to

Dymodia later). This converter imports a HiP-HOPS file (input, output, or both) and generates an ODE model from it, which can be saved in XML format. As development continues, it may also be possible to allow conversion in the other direction, i.e. from ODE to HiP-HOPS.

The ODE Model Converter is a standalone executable and so does not require any associated infrastructure; to launch, simply run the executable.

The main interface is displayed in the figure below:

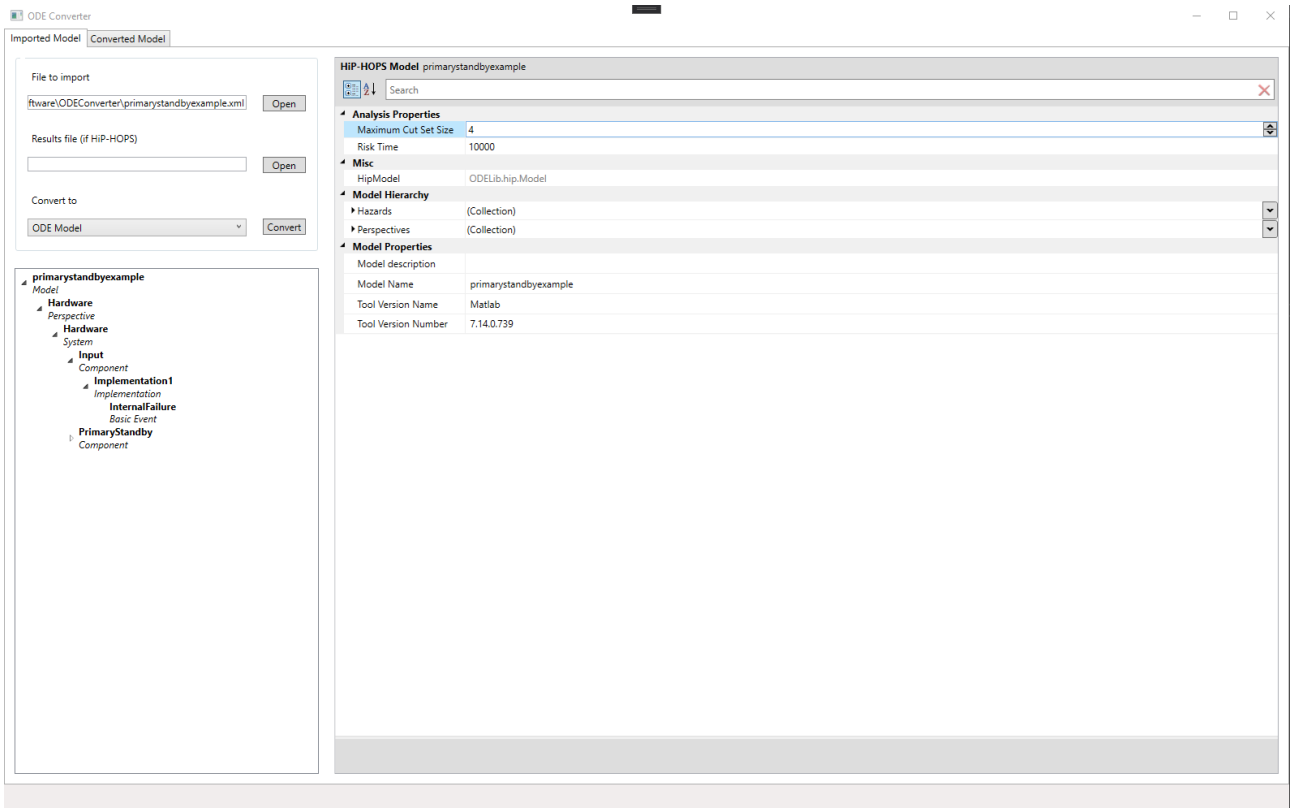


Figure 33 - ODE Converter - imported model

The ODE converter has two tabs. The first displays the imported model (i.e., from HiP-HOPS) that serves as the source model for the conversion. Because HiP-HOPS actually uses two separate files, one containing the system architecture and the other containing the failure propagation model and analysis results, the converter allows both to be imported and merged to create a combined HiP-HOPS model. Equally, it also allows just one file to be imported, though this limits the usefulness of the converted model in turn.

The imported/merged model hierarchy is viewable in the tree view on the left. Different elements can be selected from this hierarchy, allowing their properties to be viewed and edited in the grid on the right. This also allows rudimentary editing of the model, e.g. to tweak any necessary properties before conversion to an ODE model.

The second tab displays the converted model (i.e., the ODE model) in a similar fashion. On the left is the model hierarchy and on the right are the properties of the selected system element.

Here, it is also possible to save the converted model to a file. The result is an ODE-compliant XML file that can then be imported by other EDDI-related tools, thus enabling it to be used to e.g. create runtime EDDIs.

Note that the model converter is still a work in progress and features may change as development continues.

6. OTHER TOOLS: GENIE BAYESIAN NETWORK TOOL

6.1 WHAT IS GENIE MODELER

GeNIe Modeler[9] is a commercial modelling and analysis tool by BayesFusion[10]. It allows graphical modelling of Bayesian networks, and supports Bayesian inference at design time to get fast feedback during the modelling phase. Further, a plethora of machine learning algorithms are supported for either learning the structure, i.e., the causal connections, of a Bayesian network or for parametrization of the network's conditional probability distributions with the help of data. Diverse analysis tools, for instance sensitivity analysis, are available in the tool as well to deepen the understanding of a specific Bayesian network and its respective inference process.

In addition to Bayesian network modelling and inference, GeNIe Modeler supports the following machine learning algorithms:

- Algorithms for learning the network's structure:
    - PC (Peter Clark)
    - Bayesian Search
    - Greedy Thick Thinning
    - Tree Augmented Naïve Bayes
    - Augmented Naïve Bayes
    - Naïve Bayes
- Algorithms for learning the network's parameters:
    - EM (Expectation Maximization

Further, the application supports the following methods for evaluation and analysis:

- Evaluation using normal testing
- Evaluation using a k-fold cross validation
- Strength of influence analysis
- Sensitivity analysis

Notice that GeNIe Modeler is not only used for Bayesian networks, but also supports influence diagrams, dynamic Bayesian networks, equation-based models, and hybrid models.

[9] https://www.bayesfusion.com/genie/
[10] https://www.bayesfusion.com/

For a more in-depth perspective on GeNIe Modeler, we refer to the official BayesFusion website[11] and their support page[12].

6.2 HOW IT WORKS

GeNIe Modeler is a commercial software. So, beforehand a license must be acquired and the software must be installed. In addition to their commercial business license, BayesFusion distributes a free academic license as well. The following steps show the process of getting a license, installing, and setting up GeNIe Modeler:

To use GeNIe Modeler, the user must:

1. Install the respective version of GeNIe Modeler as described on the official website[13]. Notice that you must decide between the business[14] and the academic[15] version. So, navigate to https://www.bayesfusion.com/downloads/.
    a. If an academic version is desired, navigate to https://download.bayesfusion.com/files.html?category=Academia and download Genie academic version (compatible with Windows, can be used in MacOS and Linux with emulation software e.g. Wine)
    b. The business version requires purchasing a license by contacting Bayesfusion. 30-day trial versions are available at https://download.bayesfusion.com/files.html?category=Business
2. Execute the installer (in the emulated environment, if applicable) and follow the instructions to install Genie Modeller.
3. Launch Genie Modeller for the first time.
4. Get a license for your GeNIe Modeler version after launching the application for the first time. For that, you just need to follow the dialog box that opened automatically.
    a. In case of a business license, you must have the individual license file provided by BayesFusion on hand.

Now, GeNIe Modeler can be used. In addition to the following tutorial sections, we refer to the official user handbook[16] for GeNIe Modeler for specific questions and more details in general.

6.2.1 Using GeNIe Modeler for Bayesian Network Modelling

1. Launch GeNIe Modeler.
2. Now, GeNIe Modeler starts with an empty network like shown in Figure 34.

[11] https://www.bayesfusion.com/
[12] https://www.bayesfusion.com/resources/
[13] https://www.bayesfusion.com/downloads/
[14] https://download.bayesfusion.com/files.html?category=Business
[15] https://download.bayesfusion.com/files.html?category=Academia
[16] https://support.bayesfusion.com/docs/GeNIe/

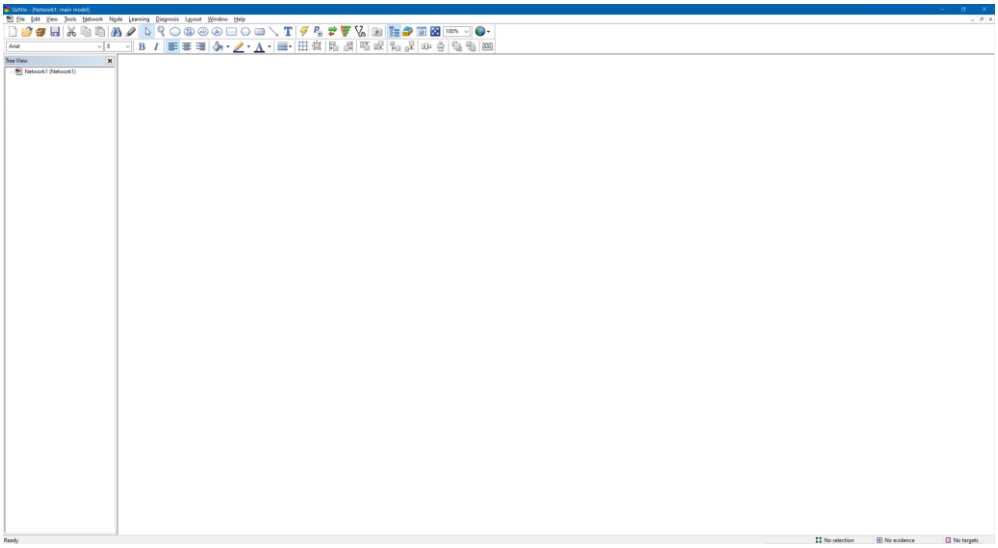


Figure 34 - Initial screen of GeNIe Modeler

3. Now, you can start modeling a Bayesian network by adding nodes via the button in the topbar that symbolizes a network node (as shown in Figure 35). Hint: You have to click on the canvas after selecting the node button to create new nodes.

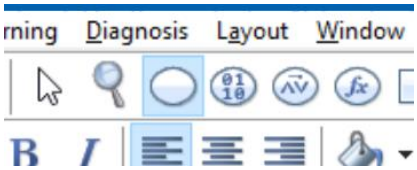


Figure 35 - Button to select for creating new Bayesian network nodes

4. Similar, with the arc button (as shown in Figure 36), you can connect two nodes in the canvas to model a causal relationship.

Figure 36 - Button to select for creating new causal relationships between two nodes

5. In the "Node properties" you can specify each node. You can open the "Node properties" dialog window by double clicking on a node on the canvas.
    a. In the "Definition" tab, you can define the concrete states of that node. You can rename them by double clicking on the node's states (left part in Figure 38). Via the buttons in the bar above (as shown in Figure 37), you can add or remove states.
    b. In the same "Definition" tab, you can define the conditional probability distribution in form of a table for that specific node. This is the concrete parameterization for this specific node, thus, it decides the outcomes when the Bayesian network is inferred given evidence. This table gives the probabilities for each state of the node given a state permutation over all parents of this node (as shown in Figure 38).

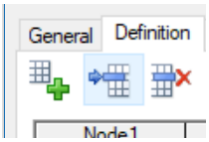


Figure 37 - Buttons for adapting the number of states of a node

Node1	State0		State1	
Node3	State0	State1	State0	State1
State0	0.5	0.5	0.5	0.5
State1	0.5	0.5	0.5	0.5

Figure 38 - Conditional probability table for a node that has two parents in an example Bayesian network

6. Finally, for saving the modelled Bayesian network, you can use the standard save functionality. This option can be reached via the "File" menu as seen in Figure 39 - Save option via the "File" menu in GeNIe Modeler. Accordingly to Figure 40, assure that you save the file as a .xdsl format.

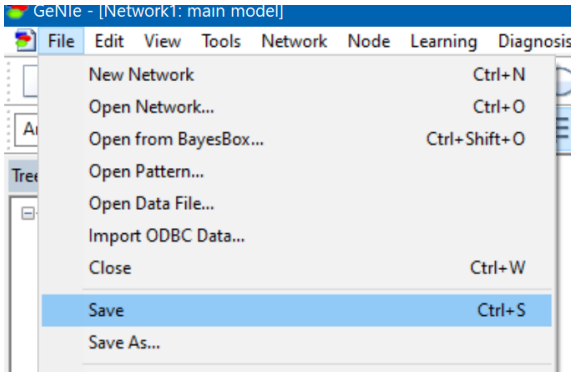


Figure 39 - Save option via the "File" menu in GeNIe Modeler

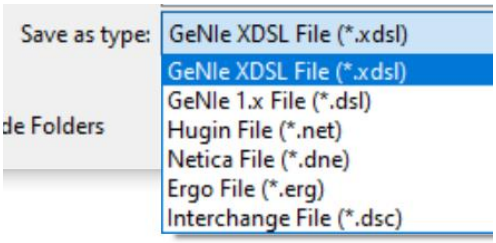


Figure 40 - File format options provided by GeNIe Modeler

6.2.2 Using GeNIe Modeler for Bayesian Network Inference

1. To test your Bayesian network, you can run inferences in GeNIe Modeler. The lightning button in the top bar updates all nodes in the network at once (as shown in Figure 41).

    a. To easier observe the outcomes of the nodes, you can change the view option on the canvas to include the states with their probability distribution. For this, mark all nodes and right click on them. Then, select the "View As" – "Bar Chart" option.

b. To set specific evidence for one node, you can simply double click on the respective states when the "Bar Chart" view option is active.

Figure 41 - Update button to run an inference over the network

2. After setting an evidence but before updating the network (= inference), the Bayesian network still has nodes without probability distributions like in the example (Figure 42).

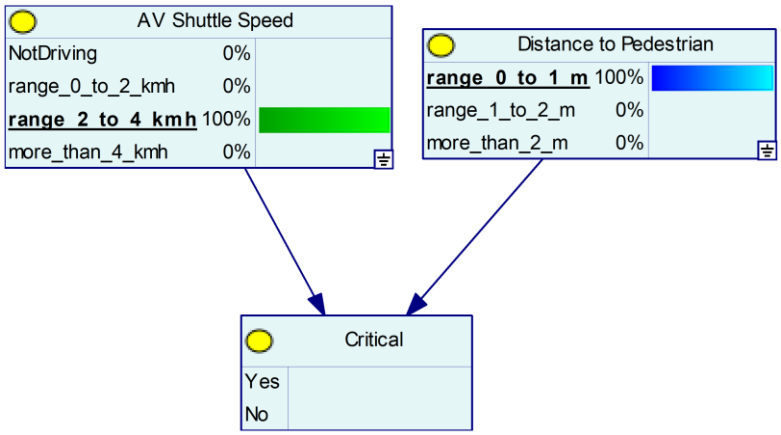


Figure 42 - Simplified example Bayesian network which is computing the criticality of a situation (node "Critical") for a autonomous shuttle given the shuttle's speed (node "AV Shuttle Speed") and the distance to the closest pedestrian (node "Distance to Pedestrian"). Two evidence (bold and underlined states) are manually set.

3. Now, after the network is updated, there is a probability distribution of the node states assigned to, respectively, computed for, all the nodes in the network. So, the node of interest can be observed that way, like the "Critical" node in the example (Figure 43).

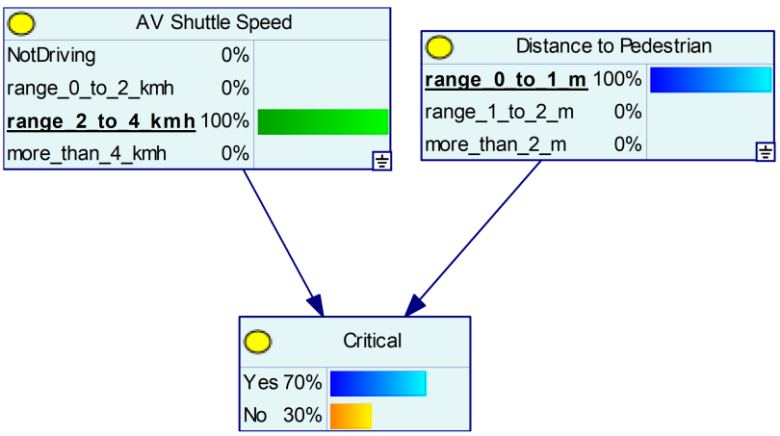


Figure 43 - Simplified example Bayesian network from Figure 42 after the inference

6.2.3 Using GeNIe Modeler for Machine Learning Tasks

1. Open the dataset that shall be used for the evaluation in GeNIe Modeler. This can be achieved via drag-and-drop or via the "File" menu using the "Open Data File…" option. Typical dataset formats like .csv, .txt, or GeNIe's proprietary .gdat format are supported.

    a. Notice that the dataset must have a data column for each evidence and output node with the data values being the node's states each.

2. Open the ML dialog box.

    a. In case of parameter learning:

        i. Simply open the "Learn Parameters…" option in the "Learning" menu like shown in Figure 44.

        ii. Confirm the "March Network and Data" window. There you can assure that all the data entries are linked to the correct network nodes and states. In case that the dataset columns and values align with the network's nodes and states identifiers, GeNIe Modeler automatically matches everything correctly. Conflicts are highlighted in this window as well and you can manually link entries by drag-and-drop.

    b. In case of structure learning:

        i. Assure that you have the dataset that you want to use actively selected as the open window in GeNIe Modeler. For that you have to select the dataset in the "Window" menu like shown in Figure 45.

        ii. Now, you have different menu options available. Select the option "Learn New Network…" in the "Data" menu like shown in Figure 46.

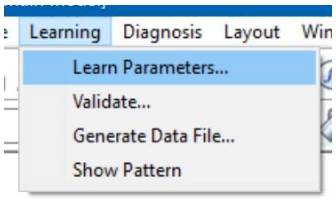


Figure 44 - Option to open the parameter learning dialog window

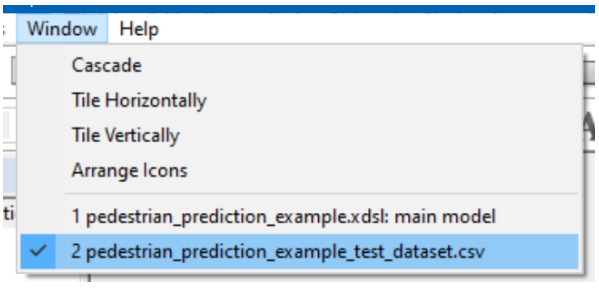


Figure 45 - Option to change the active file in GeNIe Modeler

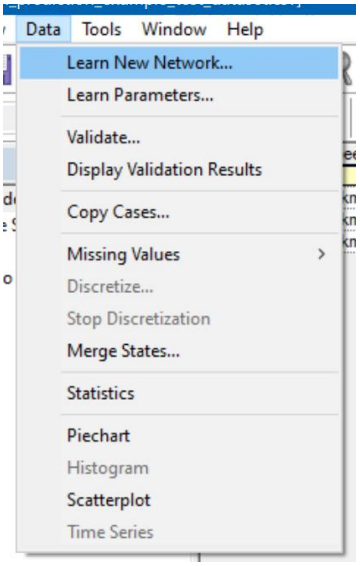


Figure 46 - Option to open the structure learning dialog window

3. Select the ML algorithm of choice, set up the respective parameters and run the ML task. For the parameter learning with the EM algorithm you can skip this step because EM is the only available option.

    a. Details on the supported datafiles, ML algorithms and their parameters etc. can be found in the official documentation[17] in the section "Using GeNIe" → "Learning".

4. After the ML task finished, the created, respectively adapted, Bayesian network will be automatically shown in GeNIe Modeler. Be aware that you must save the newly created network manually again in case of structure learning.

6.2.4 Using GeNIe Modeler for Bayesian Network Validation

Using GeNIe Modeler for Bayesian Network Evaluation

1. Open the dataset that shall be used for the evaluation in GeNIe Modeler. This can be achieved via drag-and-drop or via the "File" menu using the "Open Data File…" option. Typical dataset formats like .csv, .txt, or GeNIe's proprietary .gdat format are supported.

    a. Notice that the dataset must have a data column for each evidence and output node with the data values being the node's states each. So, an example dataset for evaluating the previous example Bayesian network (Figure 42) may look like Figure 47.

[17] https://support.bayesfusion.com/docs/GeNIe/

```
AV_Shuttle_Speed, Distance_to_Pedestrian, Critical
NotDriving, range_0_to_1_m, No
range_0_to_2_kmh, range_0_to_1_m, No
range_0_to_2_kmh, range_1_to_2_m, No
range_2_to_4_kmh, range_1_to_2_m, Yes
```

Figure 47 - Example dataset (.csv file) to evaluate the example Bayesian network from Figure 42

2. Now, the validation window must be opened like shown in Figure 48.

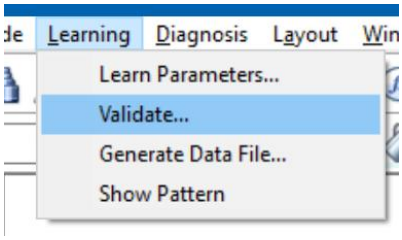


Figure 48 - Option to open the validation window in GeNIe Modeler

3. Confirm the "March Network and Data" window. There you can assure that all the data entries are linked to the correct network nodes and states. In case that the dataset columns and values align with the network's nodes and states identifiers, GeNIe Modeler automatically matches everything correctly. Conflicts are highlighted in this window as well and you can manually link entries by drag-and-drop.

4. Select the "Validation method" that you want to perform and set up the respective parameters. It is important that you chose at least one node of interest ("Class nodes") for which you want to see the evaluation results. Finally, run the evaluation.

    a. Details on the supported evaluation types, their parameter, and the evaluation result can be found in the official documentation[18] in the section "Using GeNIe" → "Learning" → "Validation".

5. After the evaluation finished, a pop-up window with the evaluation results is shown (see Figure 49). This window includes, for instance, the accuracy and the confusion matrix for each selected class node. Details on the different types of results can be found in the official documentation[19] under "Using GeNIe" → "Learning" → "Validation".

[18] https://support.bayesfusion.com/docs/GeNIe/
[19] https://support.bayesfusion.com/docs/GeNIe/

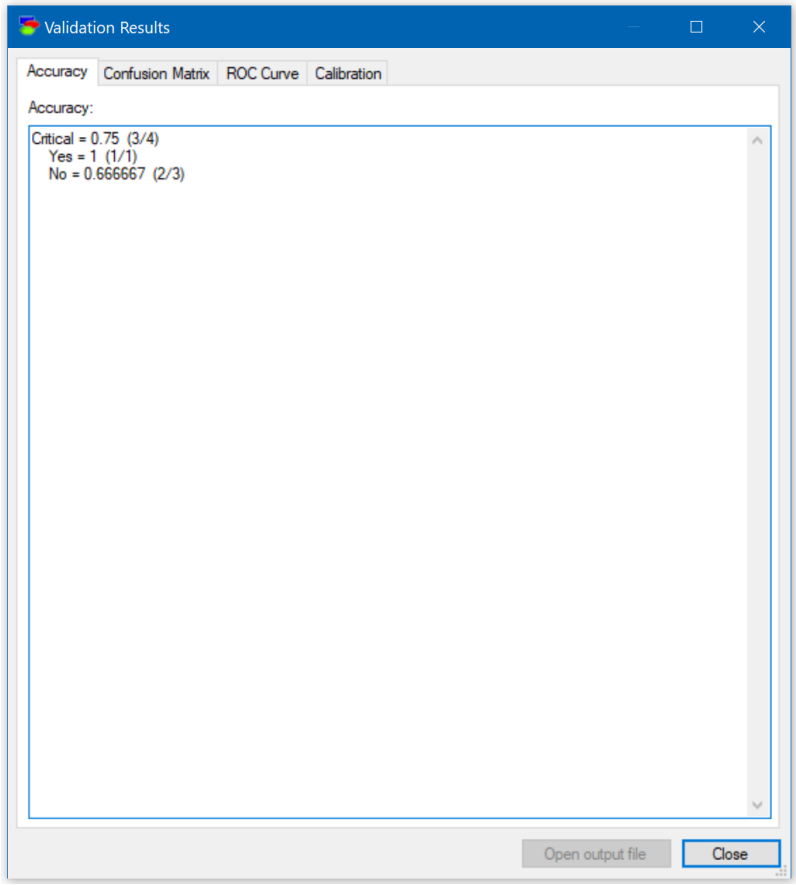


Figure 49 - Validation results for the node "Critical" from the example Bayesian network given in Figure 42 validated with the example dataset given in Figure 47

Using GeNIe Modeler for Bayesian Network Analysis

GeNIe Modeler provides tools for performing "Strength of Influence" and "Sensitivity" analyses. For details on the advantages of those analyses, the required steps to perform such analyses, as well as, information about the expected results for each type of analysis, we refer to the official documentation which goes into great detail here.

The documentation for the "Strength of Influence" analysis can be found in the official documentation[20] under "Using GeNIe" → "Bayesian networks" → "Strength of influences".

The documentation for the "Sensitivity" analysis can be found in the official documentation[21] under "Using GeNIe" → "Bayesian networks" → "Sensitivity analysis in Bayesian networks".

6.3 SESAME-SPECIFIC EXTENSIONS AND FUNCTIONALITY

To translate the Bayesian networks modelled with GeNIe Modeler to EDDIs, the tool adapter introduced in Section 5.1 is used. The tool adapter allows via remote procedure calls using Apache Thrift to send the content of an EDDI as argument and then creates and stores the actual EDDI file for the user.

[20] https://support.bayesfusion.com/docs/GeNIe/
[21] https://support.bayesfusion.com/docs/GeNIe/

The connection and integration for Bayesian networks is implemented in a Python script that connects to the Apache Thrift[22] server that the tool adapter is running. The Python script takes an .xdsl file as an input argument and triggers the tool adapter after extracting all relevant information from the .xdsl Bayesian network model. For the extraction of the Bayesian network itself, the SMILE Engine[23] from BayesFusion is used. This library is used as backend for GeNIe Modeler as well and supports the .xdsl format out of the box. The engine is written in C++ but BayesFusion provides official wrapper for a plethora of programming languages as well, i.e., Python, Java, R, and .NET. A connection to MATLAB is established too. For the SMILE Engine to work you need a license. There are free academic licenses and commercial business licenses.

So, for running the Python script three things are mandatory beforehand:

1. The tool adapter must be set up properly and the respective Apache Thrift server must be running.

2. The transformation Python script must be set up so that the user's SMILE Engine license is used.

3. The Python environment must be set up properly for running the file format transformation. This includes installing several Python packages, e.g., Apache Thrift and PySMILE (the Python wrapper for the SMILE engine).

In the following, the three points are covered. Afterwards, it is shown how to run the actual Python script to generate the EDDI Bayesian network model out of the .xdsl Bayesian network model.

6.3.1 Setting up & Running the Tool Adapter

Information on this topic can be found in Section 5.1.1 and 5.1.1.3.

6.3.2 Setting up the Transformation Python Script with the User's SMILE License

First, a license must be granted by BayesFusion to the user. For this the user must go to the download page[24] and select between the free academic page[25] or the commercial business page[26]. On their the user shall follow the steps to issue a license for the SMILE engine. After receiving and downloading the SMILE Engine license, the user must unzip the archive and locate the "pysmile_license.py" file.

After that, the "pysmile_license.py" file must be copied to the code base so that the transformation Python script is able to validate the installed PySMILE version when running the script. The user must copy the file into the following directory: "./xdsl_to_ddi/license/".

[22] https://thrift.apache.org/
[23] https://www.bayesfusion.com/smile/
[24] https://www.bayesfusion.com/downloads/
[25] https://download.bayesfusion.com/files.html?category=Academia
[26] https://download.bayesfusion.com/files.html?category=Business

6.3.3 Setting up the Python environment to run the Transformation Python Script

The user must install Apache Thrift and PySMILE in the Python environment that shall be used for running the Python script. To achieve this, the following two commands are sufficient:

1. Apache Thrift Python installation: `python3 -m pip install thrift==0.11.0`
2. PySMILE Python installation (the required command depends on the type of license that was issued):
    a. In case of a free academic license: `python3 -m pip install --index-url https://support.bayesfusion.com/pysmile-A/ pysmile`
    b. In case of a commercial business license: `python3 -m pip install --index-url https://support.bayesfusion.com/pysmile-B/ pysmile`

6.3.4 Run the Transformation Python Script to get an EDDI model

You can run the transformation Python script with the following command: `python3 xdsl_to_ddi.py –bayesianNetwork <BayesianNetwork.xdsl>`

With “BayesianNetwork.xdsl” being the name of the concrete input Bayesian network .xdsl file that shall be transformed into an EDDI file.

The generated output EDDI file can be found in the “./xdsl_to_ddi/out/” directory.

7. USE CASE APPLICATIONS

To illustrate the use of one of the tools to a SESAME use case, HiP-HOPS was used to model and analysis the Locomotec use case. Note that due to confidentiality concerns, the full results are not displayed and some elements are redacted.

7.1 LOCOMOTEC DISINFECTION ROBOT

7.1.1 Use case description

The Locomotec use case involves a UV-C disinfection robot which is capable of disinfecting surfaces in public infrastructure like hospitals, universities, or airports. Contactless disinfection using UV-C light offers advantages over traditional chemical-based disinfection in terms of time savings, coverage, and efficiency.

Figure 50 - Locomotec ARODIS KELO disinfection robot

However, the use of UV-C light also poses new hazards: over-exposure to UV-C light can be hazardous to health. As such, the robot is fitted with cameras and a machine learning-based person detection module to automatically switch off the lamps when a nearby person is detected.

The overall architecture of the system is as follows:

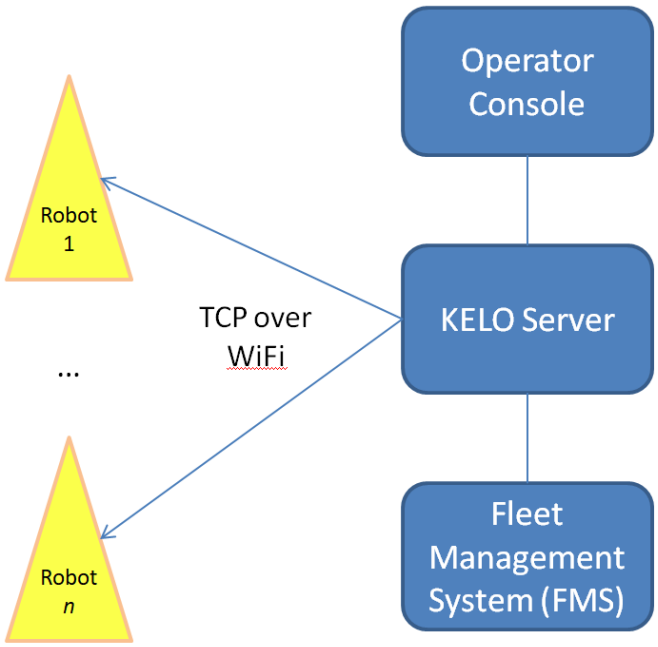


Figure 51 - Overall architecture

- **KELO Server**: The backend server for communication with the robots.
- **Fleet Management System (FMS)**: Software to manage the task planning, allocation, scheduling, and monitoring of the robots.
- **Operator Console**: Graphical interface that communicates with the FMS to show the schedule and progress of the disinfection tasks.
- **Robots**: The robots that perform the disinfection tasks. They communicate with the server via WiFi.

The robots themselves further consist of:

- **Person Detection System**: Uses cameras and a neural network machine learning system to detect nearby people. Four cameras are used, each with two concurrent ML models; a detection from a single model is sufficient to detect a person. Along with the lamps, this forms the UV-C tower on top of the robot.
- **Task Execution component**: Responsible for executing the disinfection tasks.
- **Navigation component**: Enables the robot to navigate from point to point to achieve its tasks.
- **Lamp Manager**: Turns the lamps on or off via the **Lamp Driver** depending on whether nearby people are detected.

7.1.2 FMEA for the Person Detection System

Locomotec provided a hierarchical FMEA for the person detection system and related subsystems (e.g. cameras, lamp manager software) in the form of a spreadsheet, which

showed failure modes and their effects as well as parameters like severity and likelihood, and a mind map, which showed the hierarchical composition of the various failures.

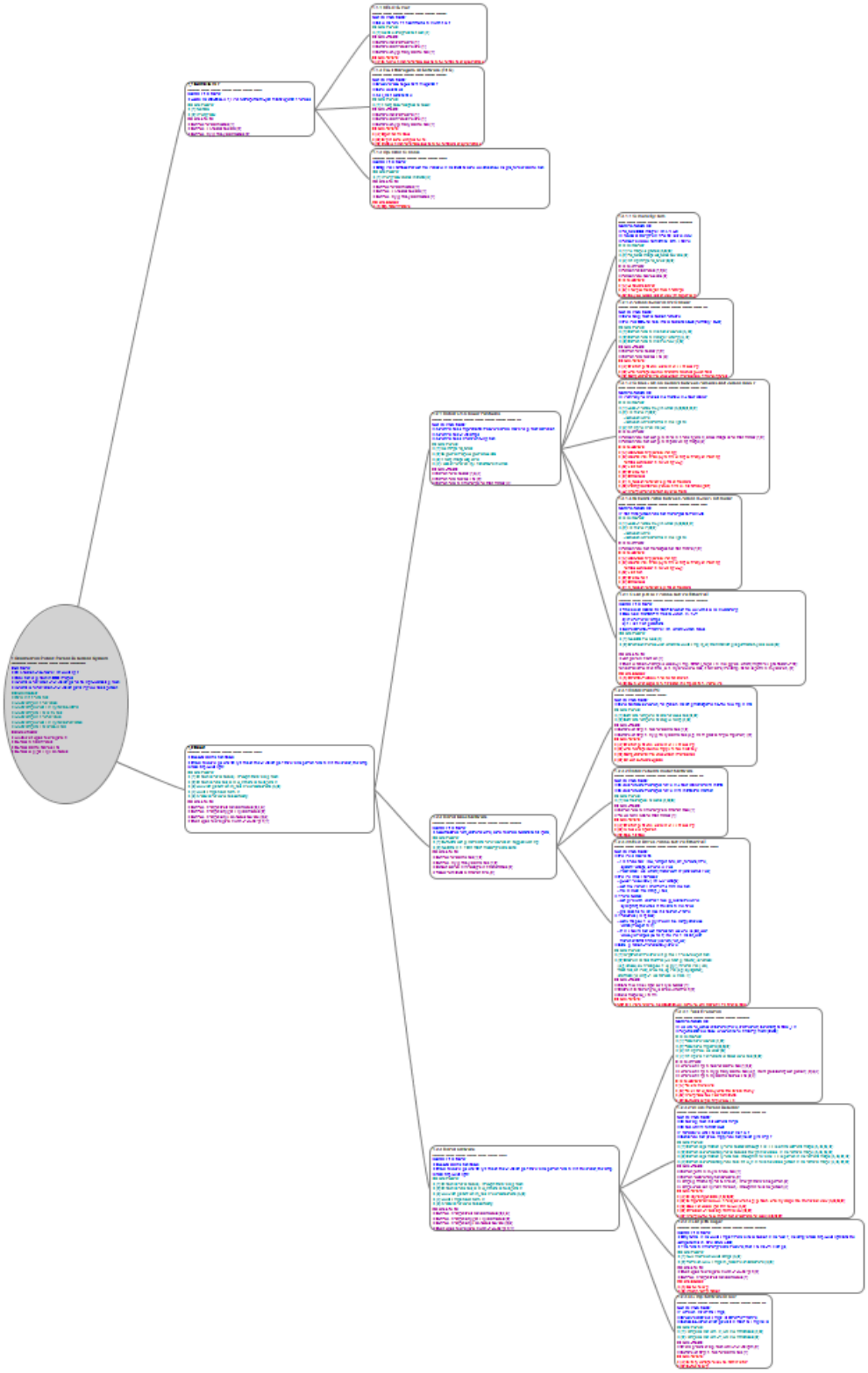


Figure 52 - Mind map of the FMEA for the Person Detection system

Each section of the mind map describes the nominal functions of a particular component along with the possible failure modes, effects, and causes, along with basic causal relationships. For example:

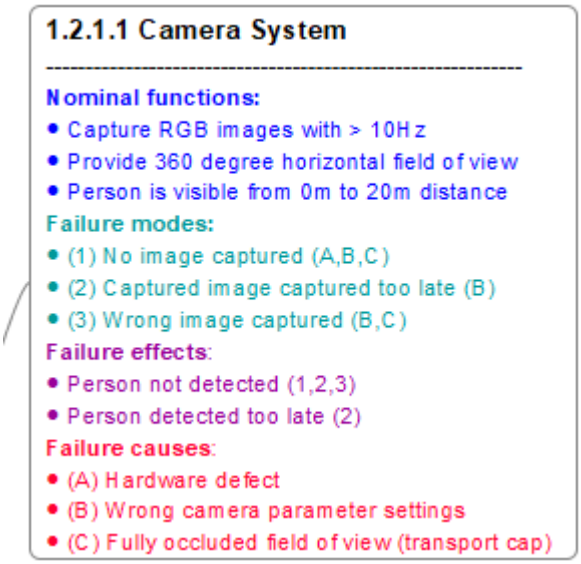


Figure 53 - Excerpt of FMEA of the Camera System

Here, the camera system functionality is described along with possible failures. Three failure modes of the camera system are listed. For each failure mode, possible causes are included (referenced by letter) and likely effects are specified (referenced by number). Thus e.g. a hardware defect (A) can lead to no image being captured (1) which in turn may cause a person to go undetected.

The causal flow can be illustrated like this:

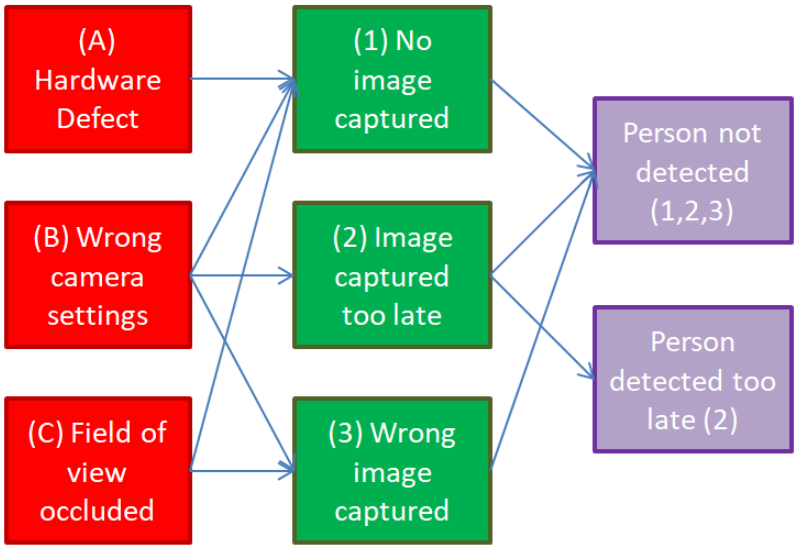


Figure 54 - Camera failure causal flow

In tabular form, the FMEA spreadsheet describes the camera system as follows:

CAMERA SYSTEM							
Nominal functions	**Failure effects**	**Severity**	**Failure modes**	**Failure causes**	**Likelihood**	**Detection**	**RPN**
- Capture RGB images - Provide 360 degree field of view - Person is visible from up to 20m	Person not detected (1,2,3)	9	(1) No image captured (A,B,C)	(A) Hardware defect	2	2	36
	- Person not detected (1,2,3) - Person detected too late (2)	9	(1) No image captured (A,B,C) (2) Image captured too late (B) (3) Wrong image captured (B,C)	(B) Wrong camera parameter settings	3	2	54
	Person not detected (1,2,3)	9	(1) No image captured (A,B,C) (3) Wrong image captured (B,C)	(C) Fully occluded field of view (transport cap)	4	1	36

7.1.3 HiP-HOPS model

Using HiP-HOPS and Matlab Simulink, a HiP-HOPS model of the system was created based on this FMEA. The model consists of various components and subcomponents along with associated failure logic.

Although HiP-HOPS makes use of FMEAs, it does so as an output of its analysis rather than an input. Therefore, the FMEA above along with the system description was used to generate a system architecture model which could then be annotated with component-level failure data and logic derived from the FMEA.

Broadly speaking, Failure Causes in the FMEA were converted to basic events (i.e., component failures), Failure Modes to output deviations (i.e., deviations of a component's expected output), and the Failure Effects to top-level hazards (which also form the top events of fault trees). For example:

- `Omission-ImageOut = HardwareDefect OR WrongCameraSettings OR OccludedCamera`
- `Late-ImageOut = WrongCameraSettings`
- `WrongImage-ImageOut = WrongCameraSettings OR OccludedCamera`

where e.g. Omission-ImageOut means omission of the image at the ImageOut port of the camera, eventually leading to Person Not Detected later in the system.

Illustrations of the model are shown below.

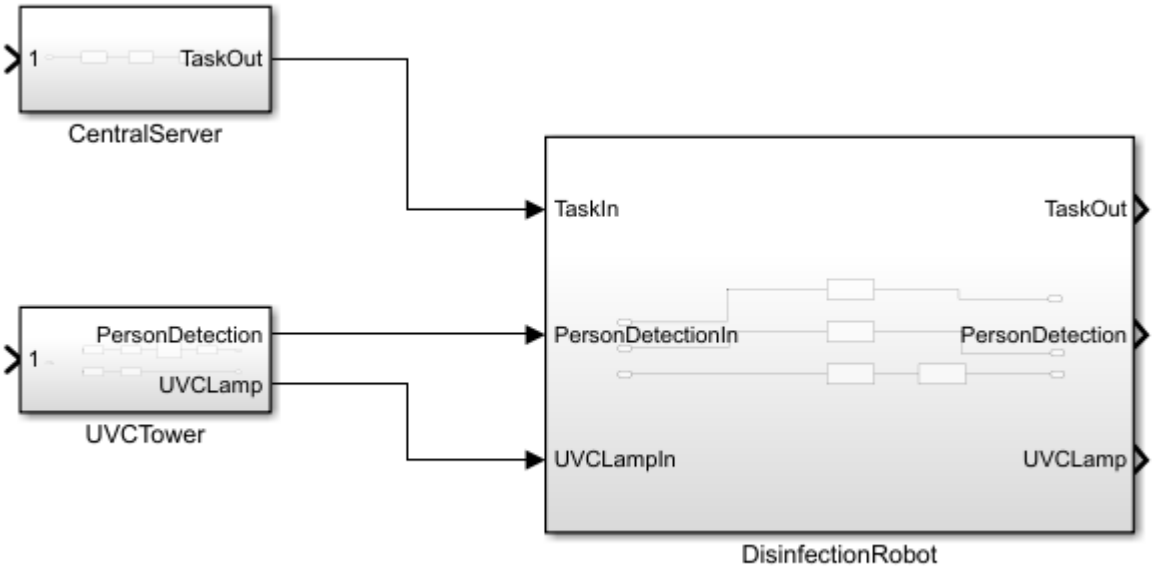


Figure 55 - HiP-HOPS model - top layer

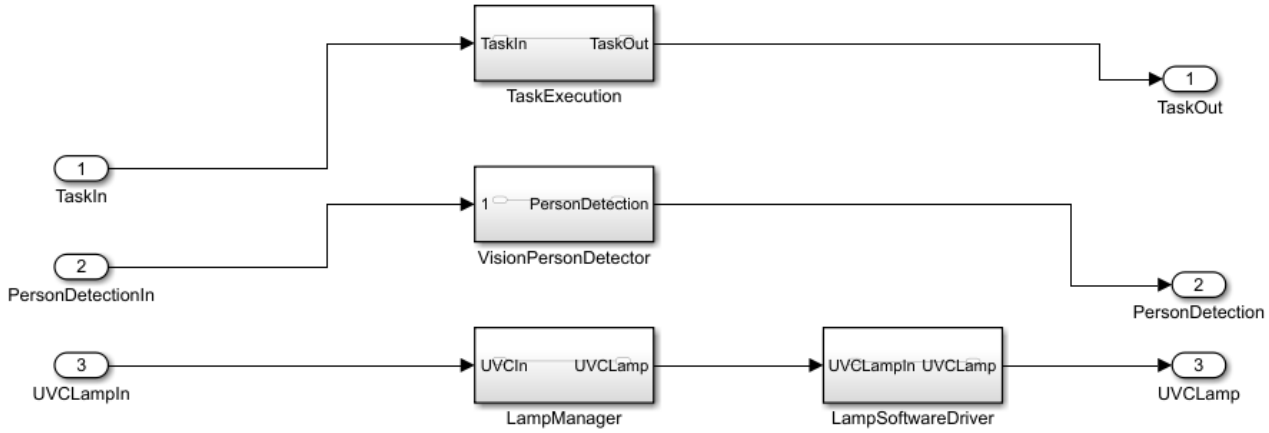


Figure 56 - HiP-HOPS model - inside DisinfectionRobot

Below is an example of the failure logic captured by HiP-HOPS for the camera system:

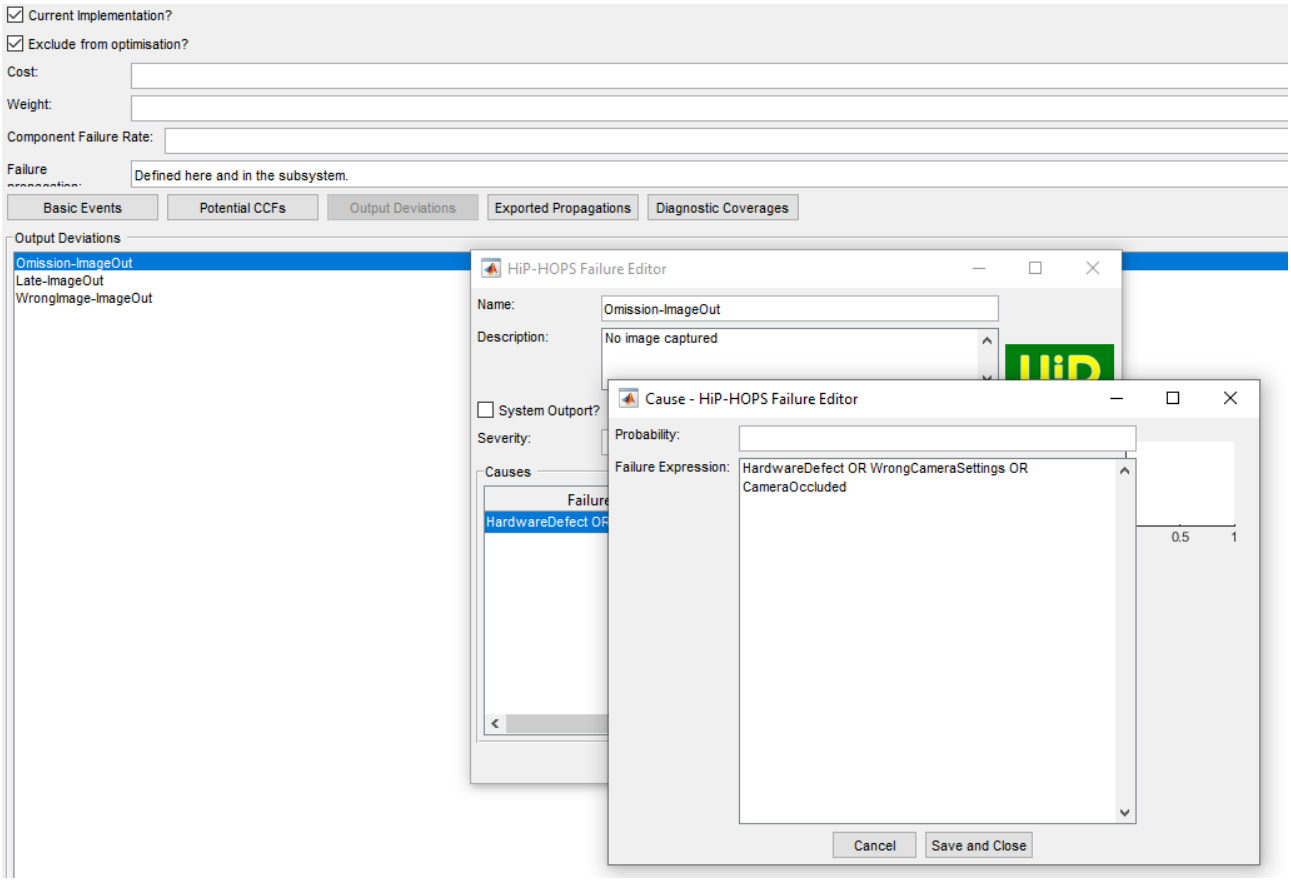


Figure 57 - Failure logic for the camera

With the model complete, a HiP-HOPS analysis can take place to produce fault trees and an FMEA. The HiP-HOPS FMEA differs slightly from the original in that it deals only with failure modes (component-level failures) and top-level system effects (i.e., hazards), omitting the middle layer of the original Locomotec FMEA but providing a more direct view of global cause/effect relationships.

The intermediate information is instead captured by the fault trees, which also show the propagation of failures through the system:

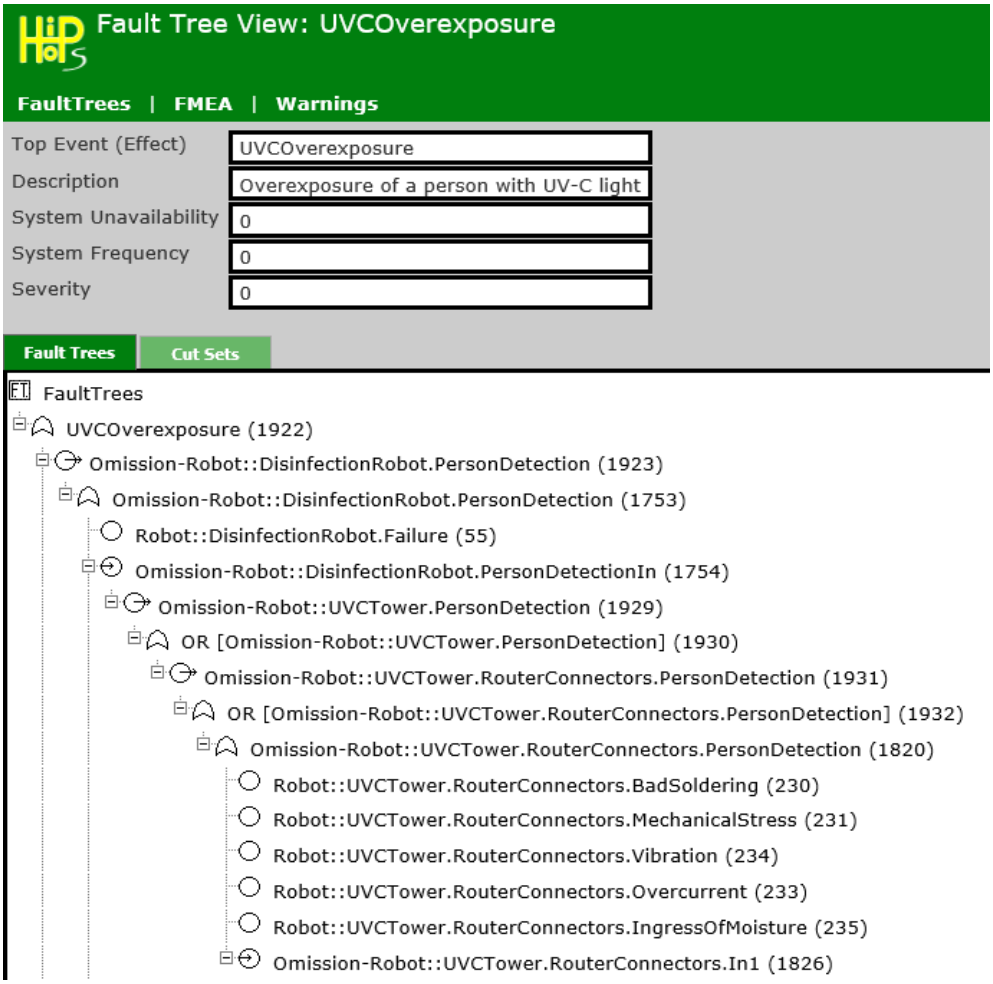


Figure 58 - Portion of the fault tree of UV-C overexposure

Because the data for the fault trees originated in an FMEA, there are only OR gates in the fault tree, no AND gates (i.e., no hazards caused only by combinations of failures). This means the cut sets are relatively simplistic but still show how component-level failures from around the system can cause the hazard.

7.1.4 Conversion to an ODE Model

Two HiP-HOPS files are used: the output from the Matlab Simulink interface, containing the system architecture description and associated failure logic, and the analysis results file. Both were imported into the ODE Model Converter to obtain a merged model containing the system architecture and the analysis information (i.e., fault trees and FMEA).

From this combined model, an ODE model was then generated. A portion of this can be seen in the following figure:

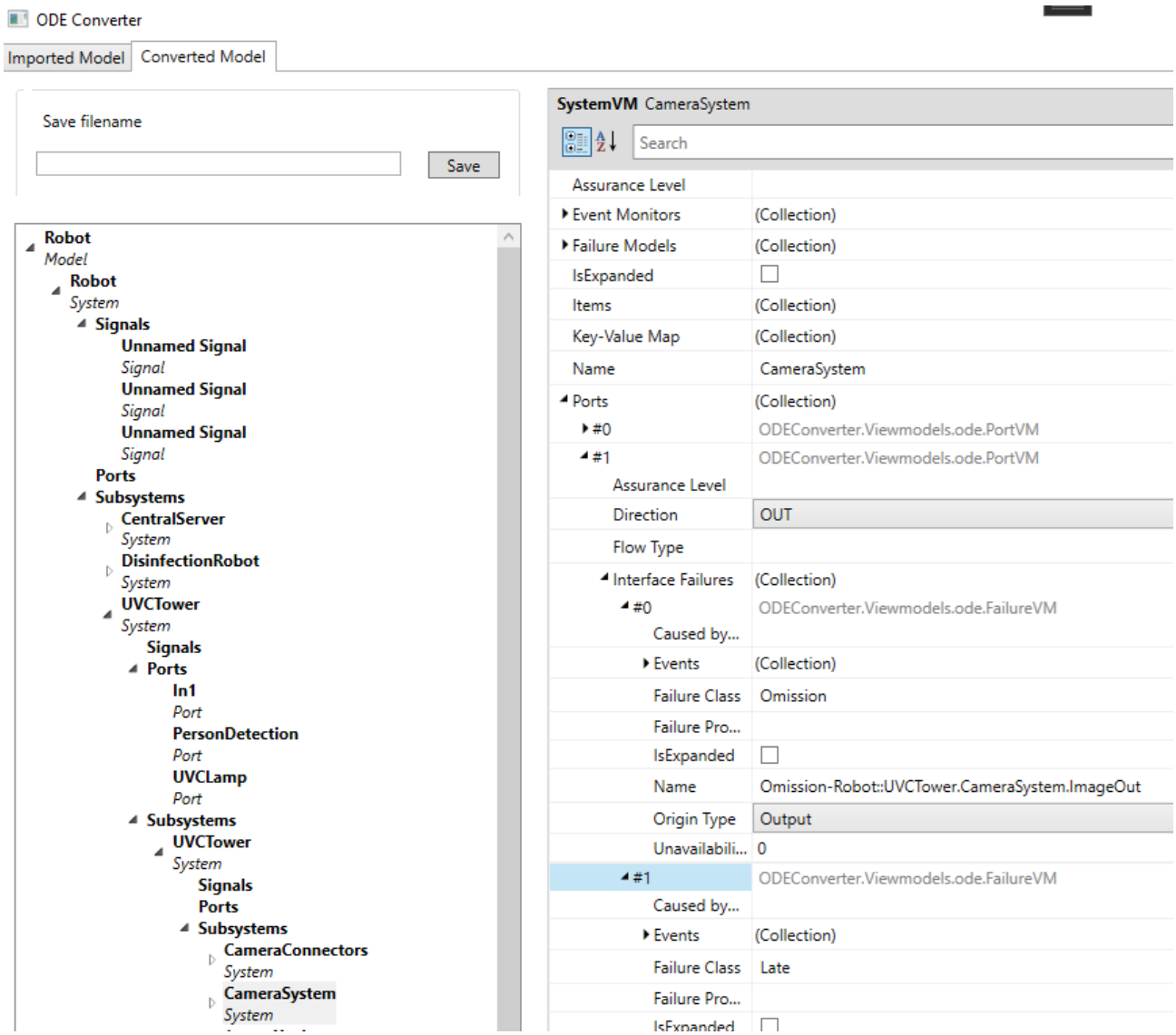


Figure 59 - Converted ODE Model

Finally, this was then saved to file to obtain an ODE-compliant XML file. The section corresponding to the camera example above is shown below:

```
<ID>120</ID>
<Name>CameraSystem</Name>
<Signals />
<Ports>
  <Port>
    <KeyValueMap />
    <ID>121</ID>
    <Name>In1</Name>
    <Direction>IN</Direction>
    <InterfaceFailures />
    <RefinedPorts />
  </Port>
  <Port>
    <KeyValueMap />
    <ID>122</ID>
    <Name>ImageOut</Name>
    <Direction>OUT</Direction>
    <InterfaceFailures>
      <Failure>
        <KeyValueMap>
          <item>
            <key>
              <string>FailureLogic</string>
            </key>
            <value>
              <dictionary>
                <item>
                  <key>
                    <string>value</string>
                  </key>
                  <value>
                    <string>HardwareDefect OR WrongCameraSettings OR CameraOccluded</string>
                  </value>
                </item>
              </dictionary>
            </value>
          </item>
        </KeyValueMap>
        <ID>124</ID>
        <Name>Omission-Robot::UVCTower.CameraSystem.ImageOut</Name>
        <OriginType>Output</OriginType>
        <FailureClass>Omission</FailureClass>
        <Unavailability>0</Unavailability>
        <Events />
      </Failure>
```

Figure 60 - Sample of the XML output for the camera subsystem

Although this is where the process stopped for now, this XML file can in future be passed to tools developed as part of WP7, where runtime-specific information such as event monitoring, actions, and ConSerts can be defined so it can serve as the foundation for the generation of runtime EDDI artefacts.

8. CONCLUSION

Within this document, three safety analysis tools have been described: HiP-HOPS, safeTbox, and Dymodia, plus the Bayesian network tool GeNIe. Information about what they are and how they work has been presented, showing how these tools can be used to create system models and safety artefacts at design time — artefacts that capture the type of information needed to produce EDDIs, either for runtime usage or as design time artefacts themselves.

The ODE metamodel serves as the foundation for EDDIs, representing the necessary information. Translation from the tools to ODE-compliant models is possible via a common tool adapter or model converter. These allow conversion of tool-specific models and file formats to ODE models that can then be further processed for generation of runtime EDDIs or used in other parts of the EDDI toolchain.

Finally, to demonstrate how this pipeline looks in action, the HiP-HOPS tool was applied to the Locomotec case study, following the steps from informal safety analysis (FMEA) to a structured system model (in Matlab Simulink) with failure data annotations and analysis results (via HiP-HOPS) and finally to an ODE XML file (via the standalone model converter).

This process is not specific to any one type of robot or indeed any particular type of system, and can equally be applied to other case studies or contexts. However, the tools and converters are still under development, and further features are forthcoming. In particular, ways of capturing further runtime-specific information such as event monitoring would be useful in facilitating the move to runtime executable EDDIs.

9. REFERENCES

[1] Y. I. Papadopoulos and J. A. McDermid, "Hierarchically performed hazard origin and propagation studies," in *Proceedings of the 18th International Conference in Computer Safety, Reliability, and Security; collected in LNCS by Springer, vol 1698 (p139-152)*, Toulouse, France, 1999.

[2] Y. I. Papadopoulos, M. D. Walker, D. J. Parker, E. Rude, R. Hamann, A. Uhlig, U. Gratz and R. Lien, "Engineering Failure Analysis & Design Optimisation with HiP-HOPS," *Journal of Engineering Failure Analysis*, vol. 18, no. 2, pp. 590-608, 2011.

[3] M. D. Walker, L. Bottaci and Y. I. Papadopoulos, "Compositional Temporal Fault Tree Analysis," *Computer Safety, Reliability, and Security. SAFECOMP 2007. Lecture Notes in Computer Science*, vol. 4680, pp. 106-119, 2007.

[4] D. J. Parker, M. D. Walker, L. S. Azevedo, Y. I. Papadopoulos and R. E. Araujo, "Automatic Decomposition and Allocation of Safety Integrity Levels Using a Penalty-Based Genetic Algorithm," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, Amsterdam, Netherlands, 2013.

[5] B. Kaiser, P. Liggesmeyer and O. Mäckel, "A New Component Concept for Fault Trees.," in *Safety Critical Systems and Software 2003, Eighth Australian Workshop on Safety-Related Programmable Systems, (SCS2003)*, Canberra, Australia, 2003.

[6] B. Kaiser, D. Schneider, R. Adler, D. Domis, F. Mohrle, A. Berres, M. Zeller, K. Hofig and M. Rothfelder, "Advances in component fault trees," in *Safety and Reliability - Safe Societies in a Changing World*, London, UK, CRC Press, Taylor Francis, 2018, p. 9.

[7] A. C. W. Group, "Goal Structuring Notation Community Standard (Version 2)," January 2018. [Online]. Available: https://scsc.uk/scsc-141B. [Accessed 12 11 2021].