Functional Safety Suite 7.1
User's Guide

Thomas Brunnengräber

March 2026

Email: fusasu@thomas-brunnengraeber.de

This product or its documentation has been created using

- Java™ Platform, Eclipse Temurin™(https://adoptium.net)
- OpenCL™ and SPIR-V™ (www.khronos.org/)
- CUDA by nVIDIA® (www.nvidia.com)
- NetBeans IDE (www.netbeans.org)
- LaTeX(www.tug.org/texlive)
- GIMP (www.gimp.org)
- Inkscape (www.inkscape.org)
- Notepad++ (www.notepad-plus-plus.org)
- docx4j (www.docx4java.org)

and other free software.

This product includes software developed by the JDOM Project (www.jdom.org).

This product includes software developed by Plutext Pty Ltd (https://www.docx4java.org/trac/docx4j).

This product makes use of the Liberation Font provided by Red Hat, Inc.

Revision history

Date	Changes
2020-04-12	First release for Functional Safety Suite release 5.0
2020-05-01	Update for Functional Safety Suite release 5.1.0: • figure 3 replaced and related text changed • figure 5 replaced, section 3.4.3.1 added, section 11.6.3 changed • figure 84 replaced, section B.1.1.2 added
2021-10-30	Extensions for Functional Safety Suite release 6.0: • section 2.2.2 added • section 3.4.1.4 incl. internationalization added • section 6 added • section 7.7.2 adapted • section 11.6.5 adapted to new functions • section 11.8.2 added • in section 11.3 commands related to *architectures* added • section 11.10 added
2026-01-06	Extensions for Functional Safety Suite release 7.0: • section 4.3.6 added due to new type *non-repairable spare* • section 7.4 changed due to new gate types • section 7.6 revised due to new gate types and Monte Carlo simulation

Contents

1 Overview

According to [Wikipedia],

> Functional Safety is the part of the overall safety of a system or piece of equipment that depends on the system or equipment operating correctly in response to its inputs, including the safe management of likely operator errors, hardware failures and environmental changes.

Functional Safety Suite is intended for modeling and calculus in the field of functional safety. It supports the following methods:

Event tree analysis: A flexible and therefore widely used method for quantitative **risk analysis**.

Functional architecture models: Graphical representation of the hardware of the safety function, including the logical and physical interaction between the components in a schematic using common symbols. Adding information related to the possible faults of each component and their propagation, a fault tree can automatically derived from the architecture model.

Fault tree analysis: A universal method for qualitative and quantitative **hazard analysis**. In particular suitable for calculation of failure rates and unavailabilities of systems, that are characterized by complex homogeneous or in-homogeneous multi-channel architectures. This software also supports correct calculation of failure rates of elements, that may fail multiple times during system life time. Monte-Carlo simulation allows new types of gates, suitable to model diagnosis, replacements, spares etc. in a convenient and precise manner. The unreliability can be calculated as well.

Reliability block diagrams: An alternative visualization of multi-channel structures, based on the same algorithms as used for fault trees.

Markov models: Flexible method for quantitative **hazard analysis** of some time-variant systems, that cannot be described by fault trees. Each fault tree can be represented by a Markov model, but the Markov model is typically much more complicated, since its complexity increases exponentially with the number of basic events.

Complex component models: Specific tool to calculate safety parameters of components with multiple time-variant failure modes, of whose some are safe, some dangerous. E. g. the typical "bath tub" curve of failure rates can be modeled.

Functional Safety Suite offers:

- A graphical user interface to create and edit models.
- Steady state and transient (time-dependent) evaluation.
- High performance algorithms for exact but nevertheless fast evaluation of even huge *fault trees*.

- Charts for unavailability, unreliability, occurrence rate and other values as function of time.
- Export of all graphics in bitmap (PNG) or vector graphic format (SVG).
- Export of all evaluation output data in text format.
- Calculation or Partial Derivatives (Birnbaum Importancy), Criticality Importance, Risk Reduction, Fussel-Vesely-Importance and Risk Achievement for both *basic events* and *generic basic events*.
- Readable file formats for all data (mostly XML files).
- Linking of *complex components*, *fault trees*, *reliability block diagrams*, *Markov models* and *event trees*.
- Support of modularization in *fault trees* and *reliability block diagrams*.
- Creation of reports in Microsoft Word format (OOXML, docx).
- Update of reports, even after they have been modified manually by using Microsoft Word.

Functional Safety Suite aims to provide the maximum possible symmetry between *fault trees* and *Markov models*. Thus most *fault trees* can be converted automatically to a *Markov model*, correctly considering common cause failures and condition events.

Functional Safety Suite further aims to support the user in correct modeling. This is achieved by

- automatic creation of *fault trees* based on *architecture schematics*,
- completely internal handling of common cause failures in *fault trees* (beta-model),
- simplified handling of common cause failures in *Markov models*,
- conversion of *fault trees* to *Markov models*, including common causes and conditions,
- many *generic basic event* models fitting to all typical events,
- extensions to standard Markov models, so that also condition events can be used,
- reasonable restrictions regarding modeling and configuration,
- reasonable modifiers for basic events,
- notes or warnings if suspicious data is encountered in evaluations,
- cancellation of calculations that don't make sense

1.1 Terms and Abbreviations

Table 1: *Terms and abbreviations*

Term	Meaning
Basic event	An event related to an →element. The basic events of a Markov model form the →edges, the basic events of a reliability block diagram are the →blocks, the basic events of an event tree are the →cases (of a →condition).
β	The common cause factor of an occurrence rate or probability.

Continued on next page

Term	Meaning
Branch	In a *fault tree*: The part of the tree that is below the event including the event itself (including the special case that the event is a *basic event* and therefore the branch is just the *basic event*).
Block	The representation of a *basic event* or a reference in a *reliability block diagram*.
Case	The representation of a *basic event* in an *event tree*. This is one out of one or multiple values that a →condition can take. Each case can be true or false, defined by a probability (typically an →unavailability).
Component	A technical unit that can have several failure modes.
Condition	In an *event tree*: A constraint that can take at least two →cases.
Condition event	A *basic event* that is characterized by a probability (typically an unavailability), but no occurrence rate.
D	duty cycle
Damage	In an *event tree*: The final state that can be reached in case of a hazard, characterized by its severity.
Edge	The representation of a *basic event* in a *Markov model*.
Element	Any →component, human behavior or environmental condition that influences the behavior of the system with respect to the →top event.
EUC	Equipment under Control, see definition in [EN 61508].
Event	A situation or a state that can occur related to an element, system or sub-system.
F(t)	The →unreliability.
FT	Fault tree
FTA	Fault tree analysis
Generic basic event	The probabilistic model that describes the occurrence or existence of a *basic event*. It is stored in a *library* and thus can be used in multiple models.
Generic damage	A possible damage, defined within one *event tree*, saved in the `.etf` file.
h	The occurrence rate (also: occurrence frequency) with unit 1/h. If *h* belongs to an event describing a failure, it is called (conditional) 'failure frequency' or (conditional) 'failure intensity' (CFI).
MRT	Mean repair time. If the overall system (i. e. the "EUC" in terms of [EN 61508]) is taken out of service immediately after detection of a failure, it is zero. If the overall system is still operated for a certain time (e. g. with only one channel instead of two) or if the overall system isn't shut down at all, it is to be considered.
MTTD	Mean time to detect. Necessary for all kinds of dormant failures.

Continued on next page

Term	Meaning
MTTF	Mean time to failure, and also the mean operation time between two failures.
MTTR	Mean time to restoration. Includes the →MTTD and the →MRT.
PFD	Probability of Failure on Demand, see definition in [EN 61508]. It is identical to the (mean) →unavailability $\overline{Q}$.
PFH	Probability of Failure per Hour, see definition in [EN 61508]. It is identical to the (mean) failure frequency, and thus the (mean) occurrence rate →$\overline{h}$.
PFTT	Short for 'Process Fault Tolerance Time', i. e. the time period for which the process (→EUC) can be operated with wrong control actions before it gets out-of-control ("point of no return").
PI	Short for 'Prime Implicant', the equivalent of a minimal cut-set for incoherent fault trees. For coherent fault trees, the prime implicants are identical to the minimal cut-sets. Please see relevant literature for more information.
Q	The →unavailability.
RBD	Reliability block diagram
State	In a *Markov model*: A state that a system can take.
Sub-tree	A *fault tree* or a branch of a *fault tree* that is referred by a TRANSFER-IN gate in a (higher-level) *fault tree*. A *sub-tree* may contain references to lower-level *sub-trees*. Dividing a *fault tree* in several *fault trees* is useful, when a *fault tree* is too large to be displayed on one page, or if a branch of a tree is needed more than once.
System lifetime	The (mean) lifetime of the system in scope. Needed to calculate some values of *complex components* (see 10), for some basic event models (see 4), and as stop time for transient evaluation.
TFFR	Tolerable Function Failure Rate, the result of a THR apportionment and thus the safety requirement for a high demand or continuous mode safety (sub-)function of a (sub-)system. Also called "Tolerable Probability of Failure per Hour" (TPFH).
THR	Tolerable Hazard Rate, the result of a risk analysis for each identified hazard. If given in 1/h, it is mathematically the same as the →TPFH, but not each failure is a hazard.
Top event	The topmost *gate* of a *fault tree*, describing the (undesired) state that the system can enter due to the occurrence of one or several *basic events*.
TPFD	Tolerable Probability of Failure on Demand, the result of a THR apportionment and thus the safety requirement for a low demand mode safety (sub-)function of a (sub-)system.

Continued on next page

Term	Meaning
TPFH	Tolerable Probability of Failure per Hour, the result of a THR apportionment and thus the safety requirement for a high demand or continuous mode safety (sub-)function of a (sub-)system. Also called "Tolerable Function Failure Rate" (TFFR).
Unavailability	The probability *Q(t)* that an →element or system wouldn't perform as intended, when it would be needed at time t ("on demand"). For non-repairable systems, the unavailability *Q(t)* is identical to the unreliability *F(t)*, and both are called "failure probability". For repairable systems (modeled e. g. by *basic events* of type *repairable* or *cyclic*), unavailability *Q(t)* and unreliability *F(t)* are completely different values, since the unavailability becomes zero with each (complete) test or repair, whereas the unreliability increases monotonously.
Unreliability	The probability F(t_1,t_2) that an →element or system doesn't perform as intended over a certain time interval $t_1 \ldots t_2$. Usually t_1=0 is assumed, thus $F(t_1,t_2)$ is shortened to *F(t)* with t being t_2-t_1=t.
w	The (unconditional) occurrence density considering restoration. In contrary to h it is unconditional with respect to whether the component is still available at time t. However it is not a 'probability density' (such as *f(t)*), since its integral over infinite time is greater than 1 in general. Its unit is 1/h. Note: Up to version 3.3 of this program, the symbol w has been used for the conditional occurrence frequency, which is now named h in coherence with most literature.

1.2 Conventions

- A term in *slanted letters* indicates a term with a certain meaning within Functional Safety Suite, e. g. a type of data objects.
- A term in **bold letters** indicates a menu, command or button name.
- A term in 'single quotation marks' indicates a fixed term not directly related to Functional Safety Suite.
- A term in "double quotation marks" indicates a name or a quote. It is also used to indicate, that a term or statement is not literally correct (e. g. a simplification or common but imprecise wording).

2 Introduction and user interface

When developing a new technical system, authorities, regulations, standards or just corporate rules due to manufacturers liability usually request a risk assessment (or risk evaluation) to be performed. A risk assessment typically includes three stages:

1. **Hazard identification:** Typical methods are to use existing lists, performing a system FMECA or just "brain storming". However, this is out of scope of Functional Safety Suite.

2. **Risk analysis:** Determine the possible consequences of each hazard, including the severity of each consequence and its occurrence probability (given the hazard exists). Depending on the risk acceptance criteria or principle, the risk analysis often includes the judgment, whether the residual risk is acceptable or not. Non-functional risk-reduction measures might be considered in the risk analysis already. Mitigation by safety related functions shall not yet be considered in the risk analysis step, but in step 3.

3. **Hazard analysis:** Identify the causes and conditions for the occurrence of each hazard. If several sub-systems are related to the occurrence of a hazard, this analysis must be performed on overall system level and for each of the sub-systems. Note that on some sub-system level, the failure of the sub-system might not lead to the hazard directly, thus 'hazard analysis' should be replaced by 'failure analysis' in that case.

Note that the terms risk analysis and hazard analysis might be used differently in some standards and norms, however in this guide they are used as defined above, which is consistent to [EN 50126].

2.1 Principles

With the aid of Functional Safety Suite risk assessment will look as follows:

1. Create project:
   Click **File – New Project**, select name and directory. Click **File – Project Properties**, set values according to the characteristics of the project. See section 3 for details.

2. Perform risk analysis for each identified hazard:
   A universal and therefore frequently used method for quantitative risk analysis is the event tree method. The event tree method is kind of a super-set of the risk graph method, i. e. each risk graph can be converted to an event tree, but not each event tree can be converted to a risk graph, since an event tree provides much more possibilities. If the risk acceptance criteria is defined as an explicit value (e. g. in terms of "damage equivalent per hour and unit"), the result of a risk analysis is a THR for each hazard. [1]
   See section 5 for details.

[1] This step can be omitted if the safety function(s) and related safety requirements are explicitly given already. This is typically the case for sub-systems (on lower level).

3. Define an architecture for the system part(s) related to each hazard or safety function:
   An architecture shall describe and define the design of the system (or the system parts) having a particular hazard (or safety function) in mind. Also non-technical elements such as a user or a physical value of a process should be mentioned in the architecture. Starting with version 6.0, Functional Safety Suite provides a graphical editor enabling you to draw architecture diagrams, which are explicitly focused on safety related function and failure analysis, see section 6. If some constraints are considered, the corresponding *fault tree* structure can automatically be derived from an *architecture*.

4. Perform preliminary hazard analysis and THR apportionment for each hazard:
   The preliminary hazard analysis (PHA) describes and defines the high level architecture of the system related to the hazard down to a level of still independent (or "autonomous") elements. Also human errors and technical failures outside the technical system to be developed need to be considered. The PHA is typically performed by a fault tree analysis (FTA), see section 7 for details. The output is a set of safety functions for each sub-system, including a TPFD and/or TPFH (or TFFR) for each function.

5. Perform hazard analysis (or failure analysis) for each safety sub-function of each sub-system:
   Analyze the technical sub-system that shall perform a safety sub-function with respect to why it could not be able to perform it as intended. This is usually done by a combination of a fault tree analysis (top-down method) and a FMEDA or similar (bottom-up method). The fault tree analysis is a quite simple method to model the architecture of the function in sufficient detail, whereas the FMEDA is a simple and fast method of ensuring completeness of all low-level failures.
   Note: It is often said, that a *fault tree* wouldn't be suitable for repairable systems. This is only partially correct. In fact, for repairable systems, the occurrence rate (also called failure frequency) of an event cannot be calculated based on the 'failure probability', even though many tools do this. However with some additional algorithms, a *fault tree* is very well suited for determination of failure frequencies of repairable systems. In Functional Safety Suite the calculation of occurrence rates is based on those algorithms and thus when using Functional Safety Suite, you are highly encouraged to use *fault trees* instead of e. g. (much more complicated) *Markov models* for repairable systems. Only in case of systems changing their architecture or major characteristics at certain instances of time (time-variant systems), a fault tree might not be suitable to model the behavior of the system in sufficient detail. In that case a Markov model might be suitable, see section 9.
   Finally each *basic event* of a fault tree, each block of a RBD, or *edge* of a Markov model is assigned a *generic basic event*, determined by its name and description.

6. Define generic basic events:
   In the next step, the *generic basic events* used in the fault trees, RBD's or Markov models are quantified. For most events, one of the standard models for *generic basic events* will be suitable. For basic events representing failures of components with several

failure modes (including time-variant failure rates), the *complex component* model (see section 10) can be used together with a *generic basic event* of type *link*.
Sometimes it is useful to describe a sub-sub-system by a fault tree or Markov model of its own. Also in this case, a *generic basic event* of type *link* can be used, referring to another *fault tree* or *Markov model*, see section 2.4.

7. Evaluate the models, e. g. calculate the value(s) of interest, investigate the importance of basic events in order to optimize the system etc.

8. Create a report, summarizing all inputs and all results of the risk evaluation, including assumptions and constraints.

Thus finally a Functional Safety Suite *project* is a collection of models — models of a system in its environment (*event trees*), models of a system with respect to a certain event or state (*fault trees*, *reliability block diagrams*, *Markov models*), and models of basic events (the *generic basic events* and *complex components*) — and algorithms needed for evaluation.

Often multiple elements of the same type are used in a system, so that the failure model of this type of element is needed several times in one or several models. Respecting this fact, in order to facilitate modeling, the probabilistic model of a *basic event* is given by a *generic basic event*. Each *basic event* used somewhere in a *fault tree*, *reliability block diagram*, *Markov model* or *event tree* is mainly a reference to a *generic basic event*, optionally extended by some proprietary data (see sections 7.3 for *fault trees* and 9.3 for *Markov models*). The reference is represented by the name of the *basic event*, which is in fact the name of the *generic basic event*. By this the creation and maintenance of models is significantly simplified, especially the handling of common cause failures and identical events (in [EN 61025] named 'repeated' or 'replicated' events). The *basic events* of *fault trees* are sometimes explicitly called *tree basic events*, whereas *basic events* of *Markov models* are typically called *edges*, and *basic events* of *reliability block diagrams* are called *blocks*.

In *fault trees* the combination of *basic events* is modeled by gates. A *fault tree* is in principle not suitable to describe the sequence of occurrence of events, but in fact this is not important for the big majority of systems. For the rare cases in which sequence matters, a *Priority-And gate* might be able to describe the behavior correctly. In *Markov models* the occurrence of one or multiple *basic events* results in different *states*, also depending on the sequence of occurrence.

In contrary to all other kinds of models, *complex components* are not based on *generic basic events*, since their failure modes are directly stated in the model, see section 10.

All *generic basic events* needed in the models of a *project* are collected in *libraries*. One *library* and any number of *event trees*, *architectures fault trees*, *reliability block diagrams*, *Markov models* and *complex components* form a *package*. Any number of *packages* and some common data valid for all packages build the *project*.

2.2 The Desktop

2.2.1 The Main Window

The desktop has seven areas:

- The menu bar.
- The tool bar.
- The project tree (shown only if no event or other element of a model is selected).
- A properties panel related to the currently active model, showing either the properties of the model or properties of some element of the model.
- The *model graphics tab pane*, with the active model presented in the active tab.
- A message output window displaying hints, warnings or errors occurring during file operations and calculations.
- A status bar displaying hints, if an action could not be performed.

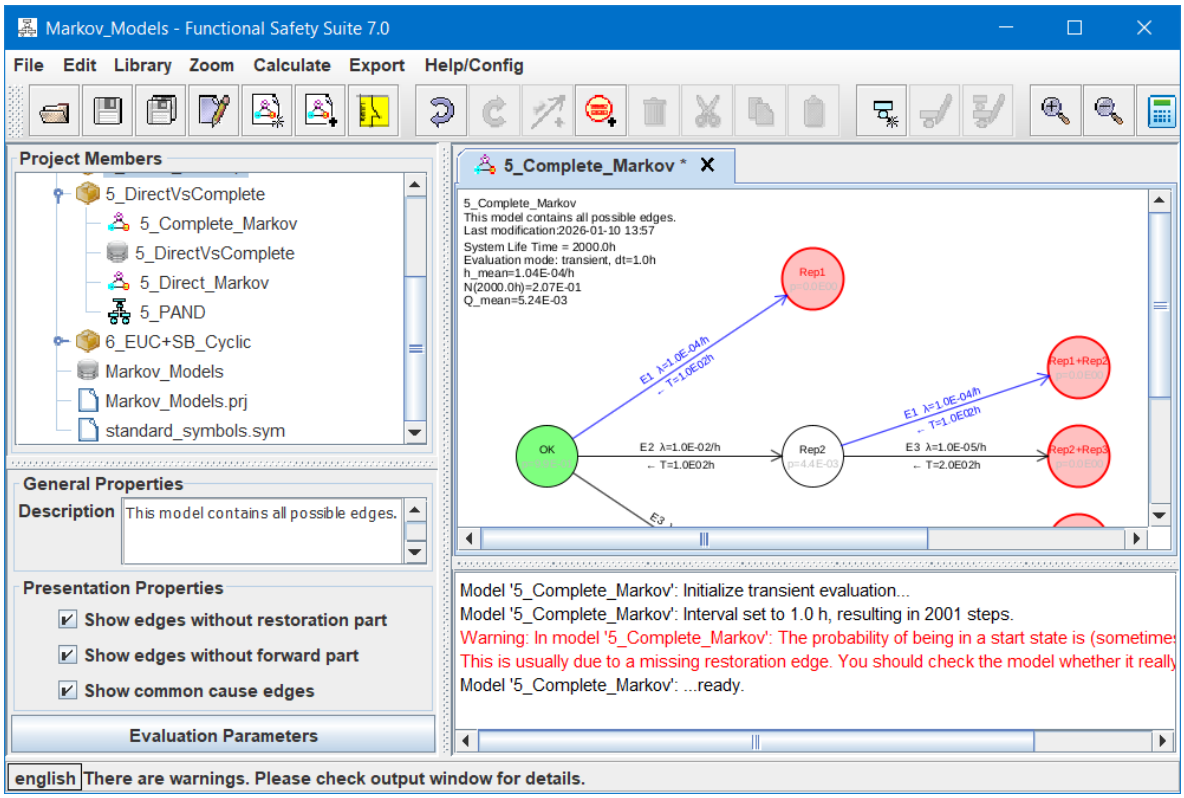


Figure 1: *The desktop*

The data displayed in the properties panel is related to the active *model graphics tab*.

- If no model is active, only the project tree presenting the members of the project is displayed.

- If a model is active, but no event selected (marked), in the upper section a tree presenting the members of the project is displayed, and in the lower section the model's properties are displayed, e. g. the description, some visualization related values, some evaluation related values.
- If a *architecture* is active and a *architecture component* is marked, the properties of the marked *architecture component* and its selected part are displayed.
- If a *complex component* is active and a *component event* is marked, the properties of the marked *component event* are displayed.
- If an *event tree* is active and a *case* is marked, the properties of the marked *case* are displayed, including the properties of the referred *generic basic event*.
- If an *event tree* is active and a *crotch* including its *condition* is marked (i. e. a *default case* or *always case*), the properties of the marked *crotch* and *condition* are displayed.
- If an *event tree* is active and a *damage* is marked, the properties of the marked *damage* are displayed, including the properties of the referred *generic damage*.
- If a *fault tree* or *reliability block diagram* is active and a *gate* is marked, the properties of the marked *gate* are displayed.
- If a *fault tree* or *reliability block diagram* is active and a *basic event* is marked, the properties of the marked *basic event* are displayed, including the properties of the referred *generic basic event*.
- If a *Markov model* is active and a *state* is marked, the properties of the marked *state* are displayed.
- If a *Markov model* is active and an *edge* is marked, the properties of the marked *edge* are displayed, including the properties of the referred *generic basic event*.

Further parameters are accessible via dialog frames.

Results such as lists of minimal cut sets/prime implicants, lists of importancies, or graphics of time-variant values will be shown in separate windows.

2.2.2 Floating Window

If you right-click in the heading of a *model graphics tab*, you can select to show the *model graphics tab* in a separate floating window. This is helpful to compare two models, in particular if you've got a second physical screen.

Multiple model tabs can be moved to the floating window and back again.

Note that in the floating window, not all editing functions might be available.

2.3 Evaluation Parameters

For each *complex component*, *fault tree*, *reliability block diagram* and *Markov model* you can select which system value shall be calculated:

- the mean unavailability $\overline{Q}_{\mathrm{sys}}$
- the mean occurrence rate $\overline{h}_{\mathrm{sys}}$ and the mean unavailability $\overline{Q}_{\mathrm{sys}}$

- the unreliability ("failure probability") after a defined system lifetime or mission time $F(t)_{\text{sys}}$

For each *complex component*, *fault tree*, *reliability block diagram* and *Markov model* you can select between steady-state evaluation and transient evaluation (time-dependent evaluation).

There are more parameters depending on the type of the model, see the related sections later in this user manual.

2.4 Hierarchy of Models: Links

Often it makes sense to split a large system into different modules. This is also possible for quantitative risk evaluation. Therefore each *complex component*, *fault tree*, *reliability block diagram* or *Markov model* can be used as a *basic event* in another *fault tree*, *reliability block diagram*, *Markov model* or in an *event tree*. The relation is created by a *generic basic event* of type *link*.

A linked model is evaluated before the higher level model makes use of it, according to its specific evaluation parameters. If the upper level model is evaluated in steady-state, a *generic basic event* of type *link* takes the unavailability $\overline{Q}$, the occurrence rate $\overline{h}$ or the unreliability *F(t)* of the referred model — even if the linked model is evaluated for its transients. If the upper level model is evaluated for its transients, it asks for *h(t)*, *Q(t)* or *F(t)*. The referred model will provide either these values or the mean values instead, depending on its evaluation mode.

3 Projects

The *project* organizes all data related to a problem. Therefore the first action after starting Functional Safety Suite is either opening an existing *project* or creating a new one. Only one *project* can be open at a time.

A *project* that has not been saved after the latest modification, is marked with an asterisk '*' in the window title.

3.1 Packages

Models are organized in *packages*. This concept has been introduced in order to simplify handling of files and to create some kind of information hiding in a similar manner as modern programming languages do. Each *package* consists of one *library* and optionally some models.

There is always one *global package*. The *global package* has the same name as the *project* and is located in the project directory, i. e. there is one `.lib` or `.gbes` file with the name of the project in the project directory, and all model files in the project directory are models of the global package. When creating a new *project*, the *global library* is created.

There might be any number of *local packages*. Each *local package* is located in an immediate sub-directory of the project directory. The name of a *local package* is given by its directory name. The *library* of the *local package* has the same name as the *package*. In other words, whenever a sub-directory of the project directory contains a `.lib` or `.gbes` file with the same name as the sub-directory, it is regarded as a *package*.

A model has access to the library and the models of its own *package* and the *global package*. A model of the *global package* has access to its own *package* only, accordingly.

In general, you should keep the *global package* as empty as possible, i. e. create most data in *local packages*. In particular, you should create a separate *local package* for each hazard or safety function you want to analyze. Only those models and *generic basic events*, that need to be referred by models of different *packages* should be contained in the *global package*.

A new package is created by **File – Create new Package**. You will be asked for the name of the new package. A sub-directory with the given name will be created in the project directory, and the local library file will be created.

3.2 Models

Functional Safety Suite supports models of the following types:

- Event Trees
- Architecture Diagrams
- Complex Components
- Fault Trees
- Reliability Block Diagrams

- Markov Models

A new model is created by **File – Create new Model**. The "Create New Model Dialog" will open, where you can select the *package* the new model shall belong to, and the name and type of the new model.

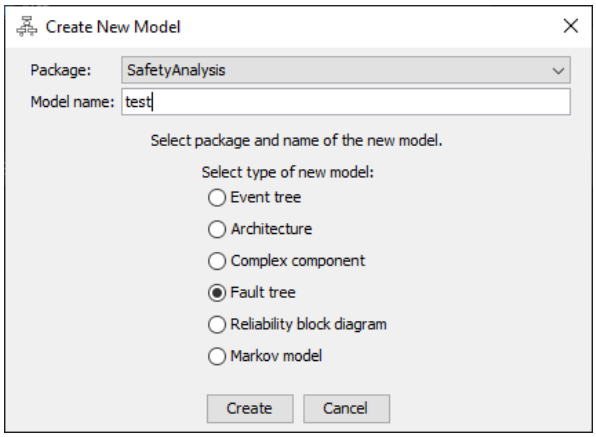


Figure 2: *The create new model dialog*

All model files found in a package directory will be loaded when opening a *project*.

3.3 Files

The following files are created and handled within a *project*:

- One project (`.prj`) file per *project*.
- At least one architecture symbol library (`.sym`) file per *project*.
- One generic basic event library (`.lib` or `.gbes`) file per *package*.
- Optionally one or multiple event tree (`.etf`) files per *package*.
- Optionally one or multiple architecture (`.arch`) files per *package*.
- Optionally one or multiple complex component (`.cmp`) files per *package*.
- Optionally one or multiple fault tree (`.ftl`) files per *package*.
- Optionally one or multiple reliability block diagram (`.rbd`) files per *package*.
- Optionally one or multiple Markov model (`.mdg`) files per *package*.

All files are text files in XML syntax. Therefore they can be read and their information can be interpreted and even changed manually (if someone considers this useful). XML schemes (`.xsd`) are available for all files.

In addition some evaluation results, intermediate results and graphics can be exported to files. Those files are described together with the related export command, see section 11.7.

Reports can be created in *Office Open XML Document* format (`.docx`), as used by Microsoft Office. [2]

[2]Please note that Libre Office or Open Office might not correctly work with the report files, unfortunately.

3.3.1 Project Files

The project properties as entered and shown in the *project properties dialog* are stored in the `.prj` file. The directory containing the project file is the project directory, and also contains the files of the *global package*.

3.3.2 Architecture Symbol Library Files

The graphic objects used to display *component parts* in *architectures* are stored in at least one symbol library file. A *default symbol library* file is shipped with Functional Safety Suite and located in the installation directory. Whenever a *project* is created, a copy of the *default standard symbol file* is created in the *project* directory.

You can create additional symbol files, in order to store your own symbols, created with the *architecture symbol editor* (see section 6.7). They will be located in the *project* directory as well. You can copy these files to other projects in order to re-use your symbols.

3.3.3 Generic Basic Event Library Files

The *library* file of the *global package* must have the name of the project. The *library* file of a *local package* must have the name of the package directory, extended by `.lib` or `.gbes`. [3] The library file only contains *generic basic event* data. The *generic basic events* contained in the *library* don't need to be actually used in the *project*. Each *generic basic event* data set includes the name and all parameters necessary to calculate the probabilistic values.

3.3.4 Model files

Each model has a name, that must be unique within the *package*. The name of the model is the same as its file name.

The *complex component* data is stored in one text file in XML format containing the following information:

- The component's description as indicated in the *properties window*.
- The component's maximum lifetime, test and repair times, evaluation mode and interval.
- Complete information of all *component events*.

Complex component file names must be extended by `.cmp`.

The *event tree* data is stored in one text file in XML format containing the following information:

- The event tree's description as indicated in the *properties window*.
- Information of all *conditions* including the names of the *cases*, so that their referred *generic basic events* can be found when loading.
- The *generic damages* used in this *event tree*, including name, description and severity.

[3] the extension has been changed to `.gbes` because `.lib` files are rejected by some email servers.

- Complete information of all *crotchs* including the structure of the *event tree*.

Event tree file names must be extended by `.etf`.

The *architecture* data is stored in one text file in XML format containing the following information:

- The architecture's description as indicated in the *properties window*.
- The *architecture components* and the *component parts* each *architecture component* consists of.
- The name of the *component part symbol* to be used for each *component part*.
- Optionally the name of the *generic basic event* representing the failure mode of each *component part*.
- The *nets* connecting the *architecture components*.

Architecture file names must be extended by `.arch`.

The *fault tree* data is stored in one text file in XML format containing the following information:

- The fault tree's description as indicated in the *properties window*.
- Some presentation related parameters.
- The mode how to convert branches to Markov models.
- The evaluation mode, algorithm and interval (in case of transient evaluation).
- Complete information of all *gates* including the structure of the *fault tree*.
- Names of the *basic events* so that their referred *generic basic events* can be found when loading.
- Optional suffixes and modifiers of the *basic events* (see section 7.3).

Fault tree file names must be extended by `.ftl`.

The data of *reliability block diagrams* is identical to the data of *fault trees*, and therefore the same file structure is used. The only difference is the extension `.rbd`.

The *Markov model* data is stored in one text file in XML format containing the following information:

- The diagram's description as indicated in the *properties window*.
- Some presentation related parameters.
- Whether and how the model shall be pre-processed before evaluation.
- The evaluation mode and interval.
- Complete information of all *states* including the structure of the *Markov model*.
- Names of the *edges* so that their referred *generic basic events* can be found when loading.
- Optional modifiers of the *edges* (see section 9.3).

Markov model file names must be extended by `.mdg`.

3.3.5 Model Files not belonging to the Project

If you want a certain model to be excluded from the project, but not delete it completely, you can remove it by **File – Remove active Model**. This will unload the model and add `_ignore` to its file extension.

You can add an existing model, whose file is extended by `_ignore`, to any *package* by **File – Add existing Model**.

Of course you can also add, rename or delete files using a file system tool, such as Windows Explorer, but you shouldn't do this while Functional Safety Suite is running in order to avoid inconsistencies.

3.4 The Project Properties Dialog

In the *project properties dialog* all options relating to all models of the project can be set. This information is stored in the project file in the project directory (extension `.prj`).

Note: There are many more parameters that can be set for each particular model and will be stored in each model's file, therefore. These parameters are described in the related model's sections later in this manual.

3.4.1 General tab

3.4.1.1 General Project Properties

Path: The path to the project directory.

Name: A user defined identifier of the *project*. The name is displayed in the title of the Functional Safety Suite window.

Description: An optional description of the *project*.

3.4.1.2 General Evaluation Parameters

System lifetime (Mission time): The system life time (in some literature called "mission time") in hours. It is used to determine the unreliabilities (occurrence probabilities) *F(T)* and occurrence numbers *N(T)* as well as to calculate some values for some *generic basic event* models (see section 4) and *complex components* (see section 10). It is also used for Monte Carlo simulation of *fault trees*, see section 7.6.2.

3.4.1.3 Model Header Display Properties

Select whether to show a header in the graphics or not. Select which values shall be shown in the header.

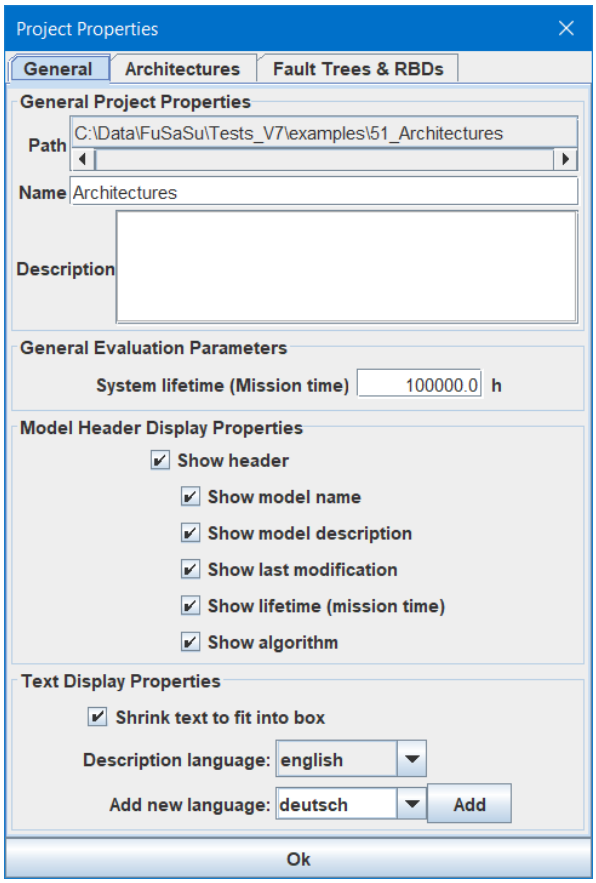


Figure 3: *The project properties dialog*

3.4.1.4 Text Display Properties

Select 'Shrink text to fit into box' if you want the text size to be adjusted so that the text fits into the event boxes of *fault trees* or *reliability block diagrams*.

From version 6.0 on, all descriptions can be entered in multiple languages:

- *project model* descriptions
- *architecture component* descriptions
- *architecture component part* descriptions
- *complex component* failure mode descriptions
- *generic basic event* descriptions
- *fault tree gate* descriptions
- *Markov model state* descriptions
- *event tree condition* descriptions
- *event tree generic damage* descriptions

The *active language* is selected by the field *Description language*. The list contains all languages already defined for this *project*. Only the descriptions in the active language will be shown and can be modified, descriptions in other languages are kept in the background and

will not be modified.

In order to add another language, select a language out of the list in *Add new language* and press **Add**. If the list of predefined languages doesn't contain the language of interest, you can type any other name. There is no difference between predefined languages and any other name, the list is only intended to guarantee that you use the same name for the same language in different project, so that you can import models or packages from other projects without problems.

If you didn't enter a description for the *active language* yet, the description of the first language will be shown instead.

3.4.2 Architectures tab

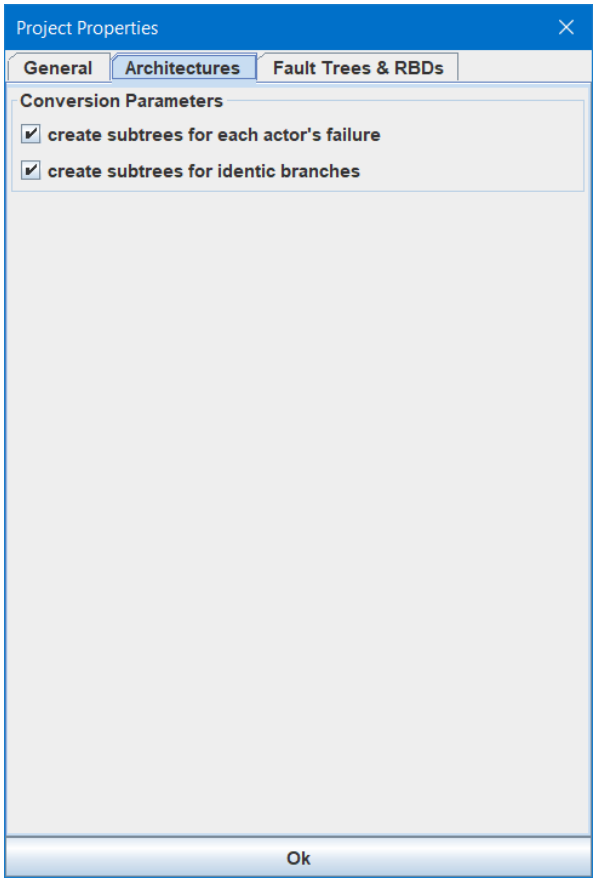


Figure 4: *The architecture properties tab*

Conversion Parameters

Select in which way a *fault tree* derived from the *architecture* is automatically split into sub-trees. See section 6.6.4 for more information.

3.4.3 Fault Trees & RBDs tab

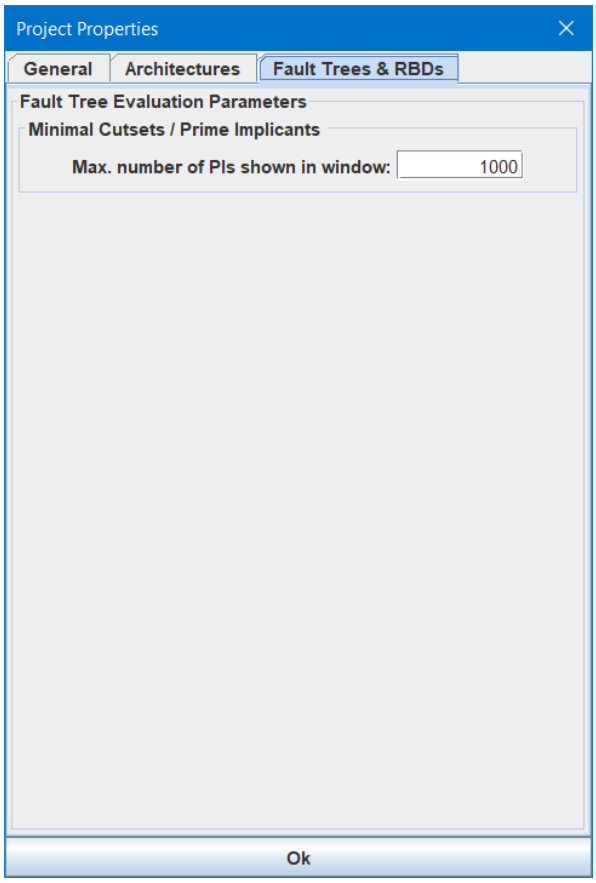


Figure 5: *The fault trees properties tab*

3.4.3.1 Minimal Cutsets / Prime Implicants

Max. number of PIs shown in window: This parameter is related to the Minimal Cutsets / Prime Implicants window, see section 11.6.3. In case of large fault trees with thousands of prime implicants, it might take quite some time to pop up the window. Since you'll usually be interested in the most critical prime implicants only, putting all prime implicants in the window is a waste of time and memory. In case of an export to a CSV file, all prime implicants will be exported.

4 Generic Basic Events and Libraries

Each *basic event* — a *basic event* of a *fault tree*, a *block* of a *reliability block diagram* as well as an *edge* of a *Markov model* — refers to a *generic basic event*. Same applies for failure modes of a part of an *architecture component*. A *generic basic event* has a unique name, a description, a type and several values needed to determine the actual values of e. g. occurrence rate *h(t)* or unavailability *Q(t)*. Also each *case* of an *event tree* refers to a *generic basic event*, even though this is not a real "basic event" logically but a constraint, characterized by its probability $p = \overline{Q}$.

Libraries are just collections of *generic basic events*. Each *package* has a *library* of its own. The name of the library is identical to the package and cannot be changed. You can import *generic basic events* from other libraries, see section 11.4.

The *generic basic events* in a *library* can be referred by all models in the *package*. The *generic basic events* in the *global library* can be referred by all models in all *packages*. In case there is a *generic basic event* in the *global library* with the same name as in the *local library*, a local model will use the *generic basic event* in the *local library*.

Due to the existence of *generic basic events* it is on the one hand safeguarded, that all models use the same data for identical or similar *basic events*, on the other hand you only have to change a basic event property at one position and all other usages are automatically updated, too. In addition, common cause factors can be handled very efficiently: In *fault trees* all *basic events* with the same names share the common cause factor (β-factor, see [EN 61508]) of the *generic basic event*— even if they are located in *sub-trees* referred by TRANSFER-IN gates.

Note that if you add an *event tree*, *fault tree* or *Markov model* from another *project* (by using **File – Add Fault Tree** etc.), the data of the related *generic basic events* is not copied since they are not stored in the `.etf`, `.ftl`, `.rbd` or `.mdg` file. Instead either a *generic basic event* of the local or global *library* of the new *project* is referenced (if there exists one with the given name stated in the model file), or a new *generic basic event* with this name is created in the *package* into which the existing model is imported (with default probabilistic data). Of course you can import the data of all *generic basic events* of the old *project* by using **Library – Import GBEs from other Library or Project**, see section 11.4. This avoids duplication of logically (but by accident not namely) identical *basic events*, resulting in wrong calculations due to lost common cause relations.

4.1 The Library View

The content of the *library* is displayed as a table by double-clicking on the library's name in the project member's tree, see figure 6.

You can sort the list of *generic basic events* in alphabetic order or by their model type. Unused values of each event are shown in light grey. All *generic basic events* not used in any model of the *project* are displayed with grey background, or not displayed at all if deselected in the control panel on the left.

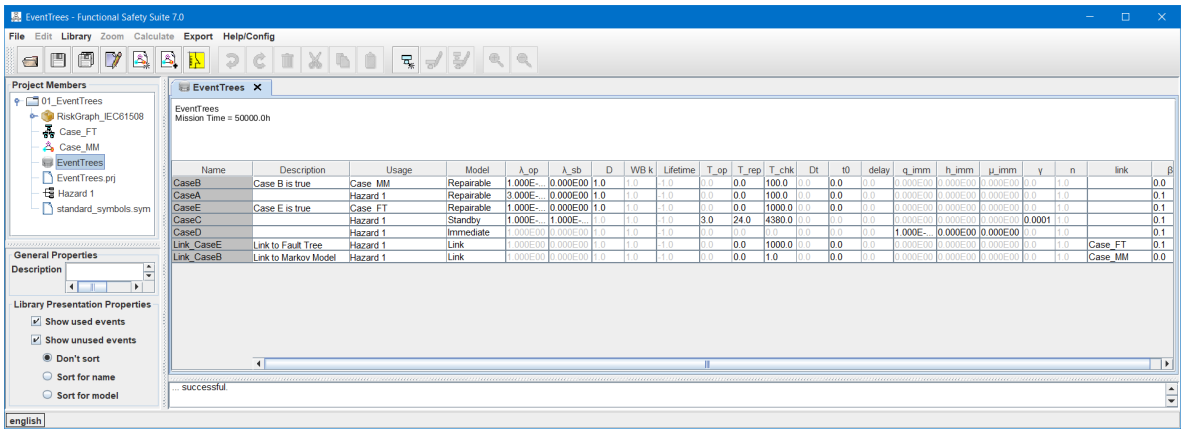


Figure 6: *Content of the library shown as table*

In the Library View, you can create new *generic basic events*, import *generic basic events* from another library or export all data to a `.csv` file, see section 11.4.

When a *generic basic event* is selected by clicking the table row, all its properties are displayed in the properties panel on the left, where they can be edited also. The selected *generic basic event* can be deleted, cut, copied, renamed, duplicated or moved to another *package*. Note that some actions can only be performed if some constraints are fulfilled, e. g. that the *generic basic event* is not used currently.

To show the project member's tree and the library properties again, deselect the *generic basic event* by clicking somewhere above the table.

4.2 General properties of Generic Basic Events

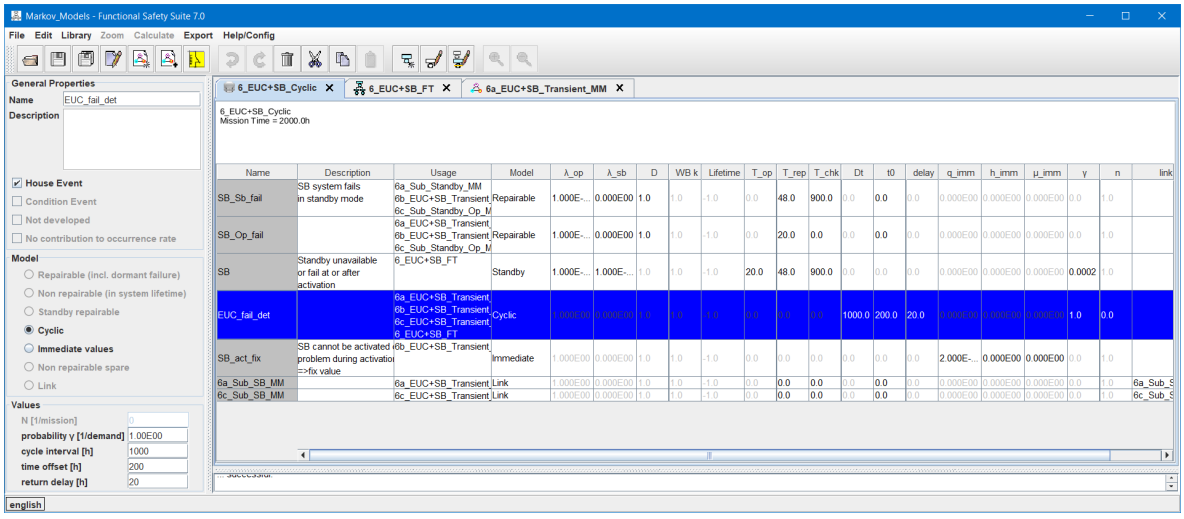


Figure 7: *Properties of a generic basic event*

Name:

A user defined identifier of the *generic basic event* and at the same time the name of all *basic events* (incl. *cases*) referring to it. The name must be unique within the *library*.

Description:

A user defined description of the *generic basic event* and therefore identical for all *basic events* referring to this *generic basic event*.

4.2.1 Modifiers

4.2.1.1 Condition

If the *condition event flag* is set, the occurrence rate h is set to 0. Thus only the unavailability $\overline{Q}$ or $Q(t)$ is considered in evaluations.

In *fault trees* a *condition event* is visualized by an ellipse instead of a circle. A *condition event* should always be connected to a condition input, e. g. of an INHIBIT-gate, except if the overall *fault tree* is evaluated for unavailability Q only.

In *Markov models* a *condition event* will create an *instantaneous transition*, see section 9.1.3.

A *case* of a *condition* in an *event tree* is always a probability (no rate), independent of the *condition event* flag being set or not. However it should be set.

There are two special cases related to the *condition event flag* in *fault trees* or *reliability block diagrams*:

1. If the *condition event* flag is set for a *generic basic event* of type *immediate*, and the *basic event* is used as condition input of a INHIBIT gate, and the parameter *probability* is set to 0, the branch topped by the INHIBIT gate is completely ignored.
2. If the *condition flag* is set for a *generic basic event* of type *immediate*, and the *basic event* is used as condition input of a INHIBIT gate, and the parameter *probability* is set to 1, the *condition event* is ignored.

These rules might be useful in order to adapt the structure of a (generic) *fault tree* to different specific applications, i. e. to simplify the re-use of a *fault tree* for different projects.

4.2.1.2 House

This modifier is only a marker with no effect to the evaluation. In *fault trees* the *basic event* is visualized by a "house" instead of a circle, this is where the name comes from. It indicates an event, that will occur in certain intervals for sure, not related to failures. The rate is often greater than once per hour.

House events must be described by the *immediate* model, requiring an occurrence rate h and optionally an unavailability Q, or by the *cyclic* model.

4.2.1.3 Non-developed

This modifier is only a marker with no effect to the evaluation. In *fault trees* the *basic event* is visualized by a rhombus instead of a circle. It is typically used to mark an event, that is out of scope or neglected.

4.2.1.4 No contribution to Occurrence Rate

If this modifier is selected, a *basic event* in a *fault tree* referring to this *generic basic event* will be ignored in the calculation of the occurrence rate. For *fault trees* this means, that the occurrence rate of each gate will be determined as if the *basic event* would be removed from the *fault tree*. It is also considered when a gate is converted to a *Markov model*.

The modifier might make sense if you want to describe a generic component, that is used in different systems, in some of which the fault modeled by the event effects the unavailability of 'low demand mode' safety functions, but is not dangerous for 'continuous mode' safety functions since it is detected within the 'process safety time' and leads to the EUC shutting down safely. This is the operation assumed in the formulas stated in [EN 61508-6].

However it is advised to use different models for different safety functions or operation, in particular if you want to convert it to a *Markov model*.

The unavailability Q is not affected by this tag.

4.3 Types and probabilistic Values of Generic Basic Events

The probabilistic failure characteristics of a component (such as the failure rate or the failure detection time) may significantly depend on the particular application. E. g. the failure rate of a relay will depend on the voltage and current (wear of contacts) as well as on the cycles per time and the overall system life time. The failure rate of a semiconductor will depend on the temperature etc. Therefore the failure rates of components of the same type may differ by multiple orders of magnitude depending on their function also within the same product. This is in particular true if the critical failure modes differ between the functions. If this is the case, different *generic basic events* should be created even for the same type of component, reflecting the different environments, operating conditions or critical failure modes of the components. The common cause factor β must be modeled manually in this case (explicit *basic events* referring to a *generic basic event* with the occurrence rate of the common cause). The same applies if components of the same type are checked in different intervals.

All data must be adopted to the defined safety function in a specific application, since the safe failure fraction and the fault detection time might be completely different. Imagine a computer: In one application a fault of the microprocessor might not be hazardous as long as it is detected, since the machinery (EUC) can enter a safe state. In another application there is no safe state (of the EUC), therefore a fault of the microprocessor is hazardous even if it was detected immediately. For this reason a THR or TFFR must always be defined in

combination with a well described hazard/failure — a component as such has no "hazard rate" [4].

The following models are available to describe the occurrence rate, the unavailability and unreliability of a *generic basic event*.

Important note: The naming of the failure models is not harmonized. Thus, there might be fault tree evaluation programs that define the "repairable" or "dormant failure" model and the "non-repairable" model in different ways, in particular with respect to when the overall "system lifetime" or "mission time" is used and when an explicit test interval t_{check} is used for evaluation. The following subsections provide all information that enables you to select the suitable model. Note that in Functional Safety Suite the system lifetime ("mission time") stated in the *project properties dialog* should always be set to the (maximum) system lifetime – any test intervals should be considered in the t_{check} parameter of the *Repairable* failure mode model, as described in section 4.3.1.

4.3.1 Repairable (incl. dormant Failure)

Using this model, the following failure scenarios can be modeled:

- failures directly leading to system failure (and therefore detected immediately) or
- failures leading to system failure if some other failures or events exist or occur (dormant failures), failure is detected by periodic tests.

These failure scenarios are the most common ones, and therefore this model is the one needed for most failures in machinery, as for example failures of an electric or electronic component, as well as failures of complex mechanical components such as pneumatic or hydraulic valves.

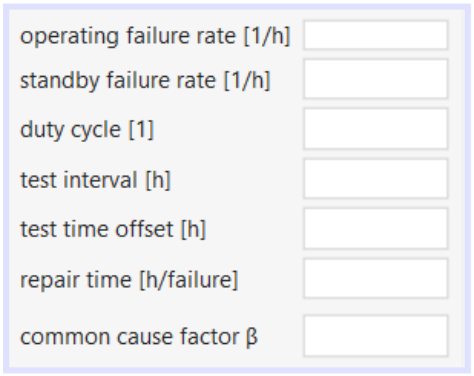


Figure 8: *Values of the repairable model*

In Functional Safety Suite the occurrence rate h consists of two parts for convenience

$$\overline{h} = h(t) = D \cdot \lambda_{\text{op}} + (1 - D) \cdot \lambda_{\text{sb}} \tag{1}$$

[4] also see [EN 50126]

where D is the duty cycle ($0 < D \leq 1$) [5], λ_{op} is the failure rate in operation ($0 < \lambda_{op}$ [1/h]) and λ_{sb} is the failure rate in standby ($0 \leq \lambda_{sb}$ [1/h]). If there is only a single mode of operation, set D=1.0 and λ_{sb}=0.0.

The mean unavailability is calculated by

$$\overline{Q} = \frac{e^{-\overline{h} \cdot T_{\text{check}}} - 1}{\overline{h} \cdot T_{\text{check}} + \overline{h} \cdot T_{\text{repair}} \cdot (1 - e^{-\overline{h} \cdot T_{\text{check}}})} + 1 \tag{2}$$

with the test interval T_{check} ($0 \leq T_{\text{check}}$ [h]) and the mean repair time T_{repair} per failure ($0 \leq T_{\text{repair}}$ [h]).

For $T_{\text{check}} \to 0$, equation (2) tends to

$$\overline{Q} = \frac{\overline{h} \cdot T_{\text{repair}}}{\overline{h} \cdot T_{\text{repair}} + 1} \tag{3}$$

For $T_{\text{repair}} \to 0$, equation (2) tends to

$$\overline{Q} = \frac{e^{-\overline{h} \cdot T_{\text{check}}} - 1}{\overline{h} \cdot T_{\text{check}}} + 1 \tag{4}$$

The mean occurrence density $\overline{w}$ is given by

$$\overline{w} = \overline{h} \cdot (1 - \overline{Q}) \tag{5}$$

The restoration rate μ used in *Markov models* in steady-state evaluation is given by

$$\mu = \frac{\overline{h}}{\overline{Q}} - \overline{h} \tag{6}$$

In transient evaluation, if T_{check} is greater than 10 times the step time t_{step}, the current unavailability *Q(t)* is given by

$$Q(t) = 1 - (1 - Q_{\text{repair}}) \cdot e^{-(1-Q_{\text{repair}}) \cdot \overline{h} \cdot ((t-t_0) \bmod T_{\text{check}})} \tag{7}$$

with t_0 being the time to the first test (the "phase shift" of the test) and Q_{repair} the (mean) unavailability due to the repair time $Q_{\text{repair}} = \frac{\overline{h} \cdot T_{\text{repair}}}{\overline{h} \cdot T_{\text{repair}} + 1}$:

$$Q(t) = 1 - \frac{e^{-\frac{\overline{h} \cdot ((t - t_0) \bmod T_{\text{check}})}{\overline{h} \cdot T_{\text{repair}} + 1}}}{\overline{h} \cdot T_{\text{repair}} + 1} \tag{8}$$

The occurrence density w(t) is given by

$$w(t) = \overline{h} \cdot (1 - Q(t)) \tag{9}$$

[5] usually DC is used as symbol for the duty cycle, but this is already used for the 'diagnostic coverage' in the field of functional safety

In *Markov models* the restoration to the origin state is performed cyclically at times $t_i = n \cdot T_{\text{check}} + t_0 + T_{\text{repair}}$.

Note that t_0 is the same for all basic events referring to this *generic basic event*, it is not possible to assign different values to them. This is intended since in practice, all tests related to a sub-system will be performed at the same time. Of course there might be two sub-systems of the same type (and thus referring to the same *generic basic events*), but if they are in fact tested at different times, they are usually not related to each other, so that one can assume, that they won't be affected by common cause failures. In that case, the sub-system should be modeled by a completely separate model, that is linked to the overall system model by a *Link event*.

The maximum unavailability $Q_{\max}$ is given by equation (8) with t=T_{check}:

$$Q_{\max} = 1 - \frac{e^{-\frac{\overline{h} \cdot T_{\text{check}}}{\overline{h} \cdot T_{\text{repair}} + 1}}}{\overline{h} \cdot T_{\text{repair}} + 1} \approx 1 - e^{-\overline{h} \cdot (T_{\text{check}} + T_{\text{repair}})} \tag{10}$$

Often there is no defined test or inspection interval ("proof test interval"), but a component's fault will be detected during normal operation in an uncritical situation. In that case this time can be used as T_{check}. If the component is never tested (thus a fault will not be detected until the hazard occurs), the test interval T_{check} must be set to the component's mean lifetime or the *non-repairable* model must be used.

This failure model can be used as *condition event*, see section 4.2.1.1.

4.3.2 Non-repairable

Using this model, the following failure scenarios can be modeled:

- failures directly leading to system failure or
- failures leading to system failure if some other failures or events exist or occur, but not detected unless other failures occur (hidden failure)

In contrary to the "Repairable" model explained in the previous section, this model is only suitable, if the failure is very unlikely to occur at all during system lifetime. If the failure is likely to occur at least once in system lifetime, you should use the "Repairable" model in order to get realistic results.

Most non-repairable events can be modeled with sufficient accuracy by a Weibull distribution. The density function *f(t)* of a Weibull distribution is given as

$$f(t) = \lambda \cdot k \cdot (\lambda \cdot t)^{k-1} \mathrm{e}^{-(\lambda \cdot t)^k} \tag{11}$$

and the distribution function is given as

$$F(t) = 1 - \mathrm{e}^{-(\lambda \cdot t)^k} \tag{12}$$

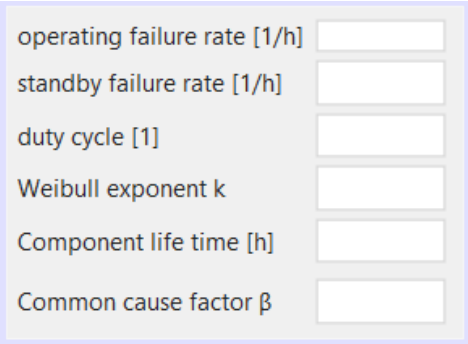


Figure 9: *Values of the non repairable model*

The exponential distribution (constant failure rate) is included as special case ($k = 1$).

In Functional Safety Suite the failure rate λ consists of two parts for convenience

$$\lambda = D \cdot \lambda_{op} + (1 - D) \cdot \lambda_{sb} \tag{13}$$

where D is the duty cycle ($0 < D \leq 1$), λ_{op} is the failure rate in operation ($0 < \lambda_{op}$ [1/h]) and λ_{sb} is the failure rate in standby ($0 \leq \lambda_{sb}$ [1/h]). Of course this splitting only makes sense for k=1 (constant failure rates), and only in this case λ is actually a 'failure rate'.

If $k \neq 1$, the occurrence rate is time variant, given by

$$h(t) = \lambda \cdot k \cdot (\lambda \cdot t)^{k-1} \tag{14}$$

The mean occurrence rate is a function of the lifetime T, given by

$$\overline{h(T)} = \frac{F(T)}{\int_0^T t \cdot f(t)\,dt + T \cdot R(T)} = \frac{F(T)}{\int_0^T t \cdot f(t)\,dt + T \cdot \big(1 - F(T)\big)} \tag{15}$$

with T being the *component lifetime*. If no *component lifetime* is stated in the properties explicitly, the global system lifetime (mission time) is used. The same applies if the stated component's lifetime is greater than the system lifetime (mission time). If the component is changed in certain intervals preventative, the component's lifetime is shorter.

For non-repairable, not preventative changed components the unavailability $Q(t)$ is identical to the unreliability $F(t)$:

$$Q(t) = 1 - e^{-(\lambda \cdot t)^k} \tag{16}$$

If the component is changed preventative after time T, the unreliability $F(t)$ is only identical to the unavailability $Q(t)$ for t<T.

The mean unavailability over lifetime $\overline{Q(T)}$ is given by

$$\overline{Q(T)} = \frac{\int_0^T F(t)\,dt}{T} \tag{17}$$

whereas the maximum unavailability $Q_{\max}$ is given by the unavailability at the end of the component's lifetime *Q(T)*

$$Q_{max} = Q(T) = 1 - e^{-(\lambda \cdot T)^k} \tag{18}$$

The mean occurrence density $\overline{w}$ is given by

$$\overline{w} = \overline{h} \cdot (1 - \overline{Q}) \tag{19}$$

and the actual occurrence density in transient evaluation *w(t)* is given by

$$w(t) = h(t) \cdot (1 - Q(t)) \tag{20}$$

This model is only applicable, if the assumption, that the component has no fault at t=0, is valid.

This failure model can be used as *condition event*, see section 4.2.1.1.

Specifics for Markov Models:

The steady-state of a *Markov model* is the state for which all transition rates are zero. This means that there must be an equilibrium between forward transition rates and return transition rates. A non-repairable element obviously has no return rate, thus the steady-state evaluation of a *Markov model* including one or several non-repairable events will always result in an accumulation of the probabilities in final states, what in turn will always result in $Q = \overline{Q} = 1$ and $w = \overline{w} = 0$. This is equivalent to $Q(t \to \infty)$ and $w(t \to \infty)$, what is in fact the only true steady state. Nevertheless what shall be calculated typically is a "pseudo steady-state", where all repairable events are in a steady state, but not the non-repairable events. Therefore for steady-state evaluation of *Markov models* an equivalent return rate μ is defined as

$$\mu = \frac{\overline{h}}{\overline{Q}} - \overline{h} \tag{21}$$

resulting in an unavailability at the end of the component's lifetime *Q(t)* (as if it would have been calculated by equation 17).

4.3.3 Standby repairable

As the name says, this model is suitable for elements, that are only needed rarely, i. e. in case of the failure of another element or the occurrence of a rare environmental condition. The interesting value is the probability, that the element is (not) available, when it is needed (on demand): the unavailability *Q*. Since its failure doesn't trigger a hazard, the occurrence rate *h* is zero. In other terms, this model is similar to the 'repairable' model, with the restriction to the unavailability as the interesting value. The overall unavailability of those elements can typically be modeled by four parts:

- A probability increasing by standby time, modeled by a failure rate in standby mode λ_{sb}. This probability is typically reduced by testing the element every T_{check} hours.

- A probability that the component is just being repaired due to a detected failure, given by the mean repair time T_{repair} per failure.
- A constant value γ representing the probability that the element fails when starting operation due to some failure occurring just when activating it. (This value can also be used to model unavailability due to maintenance or due to undetectable failures).
- The unreliability of the element in operation, i. e. the probability that the element doesn't perform its intended function during the (mean) operating time per demand T_{op}, given by the failure rate in operation λ_{op}.

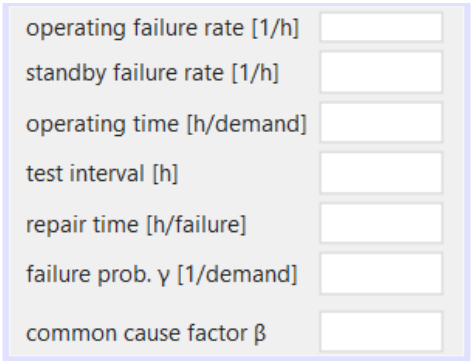


Figure 10: *Values of the standby repairable model*

The constant unavailability resulting from parts two to four is approximately given by

$$Q_{\text{const}} \approx e^{-\lambda_{\text{op}} \cdot T_{\text{op}}} \cdot \left(\frac{\lambda_{\text{sb}} \cdot T_{\text{repair}} \cdot (1-\gamma)}{\lambda_{\text{sb}} \cdot T_{\text{repair}} + 1} + \gamma - 1 \right) + 1 \tag{22}$$

All values may be zero, resulting in Q_{const}=0.

In transient evaluation, if T_{check} is greater than 10 times the iteration time T_{step}, the overall current unavailability $Q(t)$ is given by

$$Q(t) = (1 - Q_{\text{const}}) \cdot \left(1 - e^{-(1-Q_{\text{const}}) \cdot \lambda_{\text{sb}} \cdot (t \bmod T_{\text{check}})} \right) + Q_{\text{const}} \tag{23}$$

Both λ_{sb} and T_{check} must be positive values.

The overall mean unavailability is given by

$$\overline{Q} = \frac{e^{-(1-Q_{\text{const}}) \cdot \lambda_{\text{sb}} \cdot T_{\text{check}}} - 1}{\lambda_{\text{sb}} \cdot T_{\text{check}}} + 1 \tag{24}$$

This failure model always represents a *condition event*, see section 4.2.1.1.

Example

A (cold standby) emergency generator has a probability that it won't start when it is needed (given either by its failure rate in power-off state λ_{sb} and the test interval T_{check} or directly a probability γ), and a probability that it fails after successful start, but before the normal electricity is available again (given by λ_{op} and t_{op}.

4.3.4 Cyclic

If an element is needed in certain intervals over the system's lifetime, or a certain number of times (once, twice,...) within a mission independent of the length of the mission, this model might be useful.

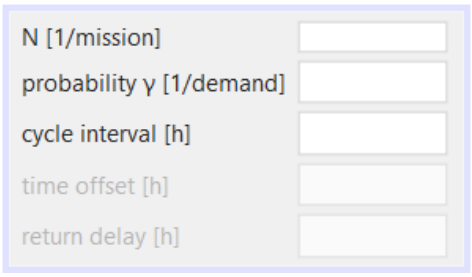


Figure 11: *Values of the cyclic model*

If N>0 the mean occurrence rate is given by

$$\overline{h} = \gamma \cdot N/T \tag{25}$$

with γ being the probability of failure per demand, N being the number of demands within the system's lifetime (or per mission) and T being the system's lifetime (or mission time). The cycle interval is $\Delta t = T/N$ in this case.

If $N \leq 0$ the mean occurrence rate is given by

$$\overline{h} = \gamma/\Delta t \tag{26}$$

with γ being the probability of failure on demand and Δt the cycle interval in hours.

If N>0 the maximum unavailability is given by

$$Q_{\max} = Q(t = T) = 1 - (1 - \gamma)^N \tag{27}$$

or by

$$Q_{\max} = Q(t = T) = 1 - (1 - \gamma)^{T/\Delta t} \tag{28}$$

if $N \leq 0$ respectively.

Note that always the maximum unavailability is used instead of a mean unavailability in steady-state calculation, in order to be on the safe side in any case. If this seems to be not adequate or too conservative, please use transient evaluation instead.

The mean occurrence density $\overline{w}$ is given by

$$\overline{w} = \overline{h} \cdot (1 - \overline{Q}) \tag{29}$$

whereas the occurrence density w(t) in transient evaluation is given by

$$w(t) = h(t) \cdot (1 - Q(t)) \tag{30}$$

respectively.

In case of transient evaluation, if the cycle interval Δt is greater than 10 times the iteration time, the transition(s) described by this *basic event* model are executed at discrete times $t_i = n \cdot \Delta t + t_0$ with $n \in \mathbb{N}_0$ and t_0 being an offset for the first transition. For transient evaluation, in addition a *return delay* can be modeled: If the *return delay* is greater than 0, the target state is left towards the source state at discrete times $t_i = n \cdot \Delta t + t_0 + T_{delay}$.

Note that the densities and rates *f(t)*, *w(t)* and *h(t)* are in fact Dirac impulses, which will for finite integration steps scatter to rectangular impulses, whose heights will depend on the integration step size. Therefore the densities and rates calculated at the steps where the cyclic events appear, are meaningless.

This failure model cannot be used as *condition event*, since this doesn't make sense.

Example

Imagine the events, that the brake pipe is closed when starting a train run, that the gear of a plane doesn't lower before landing, that the brake parachutes of the space shuttle don't open or the engine of a spacecraft doesn't restart on demand: In all these cases the mission will fail at a certain time independent of its (planned) duration.

It is also obvious, that defining THRs as safety targets doesn't make much sense for problems, that significantly depend on those events (the longer the mission the lower the failure rate gets!). Instead, the definition of a tolerable mission failure probability (mission unreliability) seems more adequate for those problems. This is also the reason why [NASA] always talks about probabilities instead of occurrence rates, in contrary to e. g. [NUREG].

4.3.5 Immediate

According to [NUREG] it's implicitly assumed, that an element is unavailable if it had a failure before. Therefore the unavailability *Q(t)* is linked to the occurrence rate *h(t)* and the test interval. For many *basic events* this assumption is not fulfilled, but the unavailability is independent of the occurrence of another event. To be able to model also those elements, the unavailability $Q(t) = \overline{Q}$, the occurrence rate $h(t) = \overline{h}$ and the restoration rate μ can be assigned explicitly ("immediately") to a *generic basic event*.

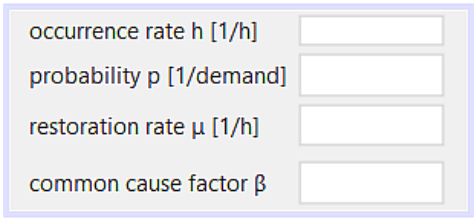


Figure 12: *Values of the immediate model*

The occurrence density is

$$w(t) = \overline{w} = h \cdot (1 - Q) \tag{31}$$

This failure model can be used as *condition event*, see section 4.2.1.1. In that case, the probability p=Q must be greater than 0, the occurrence rate h and the restoration rate μ are ignored.

4.3.6 Non-repairable spare

This model is defined for use in Monte-Carlo-Simulations. In Monte-Carlo-Simulation, a *basic event* of type *non-repairable spare* is not considered until it is activated due to a failure of some other branch. Used in other *Markov models* or *fault trees* evaluated based on minimal cut-sets, there is no difference compared to the *non-repairable* model described in section 4.3.2 above, except of the fewer parameters available.

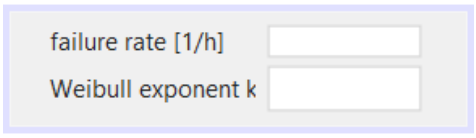


Figure 13: *Values of the non-repairable spare model*

4.3.7 Link

By *links* an *event tree*, *fault tree* or *Markov model* can use the result of the evaluation of another *fault tree*, *Markov model* or *complex component*. Thus several models of the *project* can be combined. This might be useful due to the following reasons:

- to split one large model in several small models ("modularization")
- to reuse the model of a module at multiple positions in one or several higher level models
- to "cut" common cause factors between elements of the lower level and optionally define new ones on a higher level (between the modules described by the *link*).

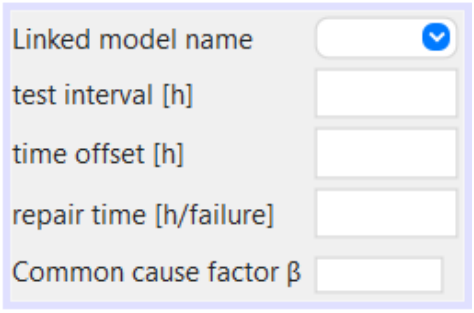


Figure 14: *Values of the link model*

The referred model is defined by its name.

A linked model is evaluated before the higher level model makes use of it, according to its specific evaluation parameters. If the upper level model is evaluated in steady-state, a *generic basic event* of type *link* takes the unavailability $\overline{Q}$, the occurrence rate $\overline{h}$ or the unreliability F(T) of the referred model — even if the linked model is evaluated for its transients. If the upper level model is evaluated for its transients, it asks for *h(t)*, *Q(t)* or *F(t)*. The referred model will provide either these values or the mean values instead, depending on its evaluation mode.

Obviously, the linked model must calculate the values required by the upper level model:

- If a *generic basic event* of type *link* is used in an *event tree*, $\overline{Q}$ of the referred model is transferred to the *event tree*.
- If a *generic basic event* of type *link* is used in a steady-state evaluation of a *fault tree*, *F(T)* or $\overline{Q}$ and $\overline{h}$ of the referred model are transferred to the *fault tree*. In a transient evaluation of a *fault tree*, *F(t)* or *Q(t)* and *h(t)* are requested from the referred model. If only the steady-state values of the referred model have been calculated, *F(T)* or $\overline{Q}$ and $\overline{h}$ are used instead. If the *condition event* flag is set (see section 4.2.1.1), only $\overline{Q}$ or *Q(t)* are transferred.
- If a *generic basic event* of type *link* is used in a *Markov model*, in steady-state evaluation either $\overline{Q}$ or $\overline{h}$ of the referred model are transferred to the *Markov model*, in transient evaluation either *Q(t)* or *h(t)* is requested respectively. Whether the unavailability or the occurrence rate is used, is defined by the *condition event* flag: If the link is marked as *condition event*, the unavailability is considered. If only the steady-state solution of the referred model has been calculated, the mean value $\overline{Q}$ or $\overline{h}$ is used instead.

Table 2 provides an overview about which values are required by which kind of link.

Table 2: *Values required by links*

Upper level model type	Calculation of[1]	Condition[2]	Required value(s)	Lower level calculation value(s)
Fault tree, RBD	*	yes	Q	Q or h and Q
Fault tree, RBD	Q	*	Q	Q or h and Q
Fault tree, RBD	h and Q	no	h and Q	h and Q
Fault tree, RBD	F (direct)[3]	no	F	F
Fault tree, RBD	F (via h and Q)[3]	no	h and Q	h and Q
Markov model	*	yes	Q	Q or h and Q
Markov model	*	no	h	h and Q
Event tree	n/a	(yes)	Q	Q or h and Q

[1] see section 7.6.3.1 for *fault trees* or section 9.5.1.1 for *Markov models*

[2] see section 4.2.1.1

[3] see section 7.6.3.7

An asterisk '*' in table 2 means any value ("don't care"). See section 7.6.3.1 for *fault trees* or section 9.5.1.1 for *Markov models* for how to set the correct calculation values.

Note that the higher level model gets no information about the structure of the linked model, thus if *basic events* in the higher level model and in the linked model refer to the same *generic basic event*, no common cause factor is considered. This behavior is typically useful, but sometimes not correct. Therefore also a *generic basic event* of type *link* can be assigned a common cause factor just as for all other types of *generic basic events*. This common cause factor is considered between all *basic events* referring to this *generic basic event* within a *fault tree* or *reliability block diagram*, and also within *fault trees* connected by TRANSFER-IN gates (since *fault trees* connected by TRANSFER-IN gates are treated as one *fault tree* in evaluation, see section 7.6).

In any case, the model using the link gets no information of the structure of the referred model(s). Thus common cause factors defined for *basic events* of the referred models are not considered in the higher level model. Nevertheless sometimes it is useful to define a common cause factor between modules of a system. This is possible by setting a common cause factor here.

Note that you can open/jump to the linked model by double-clicking the *case*, the *basic event* or the *edge*.

Specific for Markov models: Only the forward direction of the transition can be defined by a link. But in fact, sometimes also a transition back to the source state exists and thus needs to be modeled. For steady-state evaluation an equivalent restoration rate μ is calculated as

$$\mu = \frac{\overline{h}}{\overline{Q}} - \overline{h} \tag{32}$$

In transient evaluation this is not possible since *h(t)* has no relation to *Q(t)*. Therefore it is possible to define a restoration rate in the same manner as for a *repairable* element by setting a test interval $T_{check}>0$ and optionally in addition a repair time T_{repair} and the offset (phase shift) t_0 of the test.

5 Event trees

5.1 Introduction

Event tree analysis is a universal method of a quantitative **risk analysis**. Starting with the identified hazard, all possible direct consequences due to the possible *cases* of one *condition* are identified. Each possible *case* of the *condition* is assigned a probability. Only one *case* of the *condition* can be true at a particular time, thus one *case* excludes all other *cases* of this *condition*. A *condition* typically describes the existence of a certain constraint outside the technical system, e. g. the presence of people in the danger zone. For each *case* of the first *condition*, the subsequent consequences due to the *cases* of another *condition* are identified and so forth, up to a final consequence. This final consequence is the *damage*, characterized by its severity. If the consequence is "no damage", its severity is zero.

The risk R related to a certain hazard is the sum of all n severities s_i multiplied by the specific probability p_i of occurrence of this severity s_i if the hazard occurs and finally the hazard rate h_{hazard}:

$$R = h_{\text{hazard}} \sum_{i=1}^{n} \left(p_{s_i|\text{hazard}} \cdot s_i \right) \tag{33}$$

A risk graph is just a simple *event tree*, thus a risk graph can easily be replaced by an *event tree*.

Features of Functional Safety Suite related to *event trees*:

- Conditions with more than two cases.
- Several models for cases, including links to *fault trees*, *Markov models* and *complex components*.
- Two evaluation modes:
  - Calculation of the risk R, given a hazard rate h_{hazard}
  - Calculation of the tolerable hazard rate (THR), given a tolerable risk

5.2 The Event Tree Properties Panel

Properties of the overall *event tree* are displayed and edited in the *event tree properties panel*, see figure 15. All these values are stored in the event tree file (extension `.etf`).

5.2.1 General Properties

Description:

A user defined description of the *event tree*.

5.2.2 Presentation Properties

Note that in case the presentation related features don't fulfil your needs, you can export all graphics in SVG format for further processing by vector graphics tools.

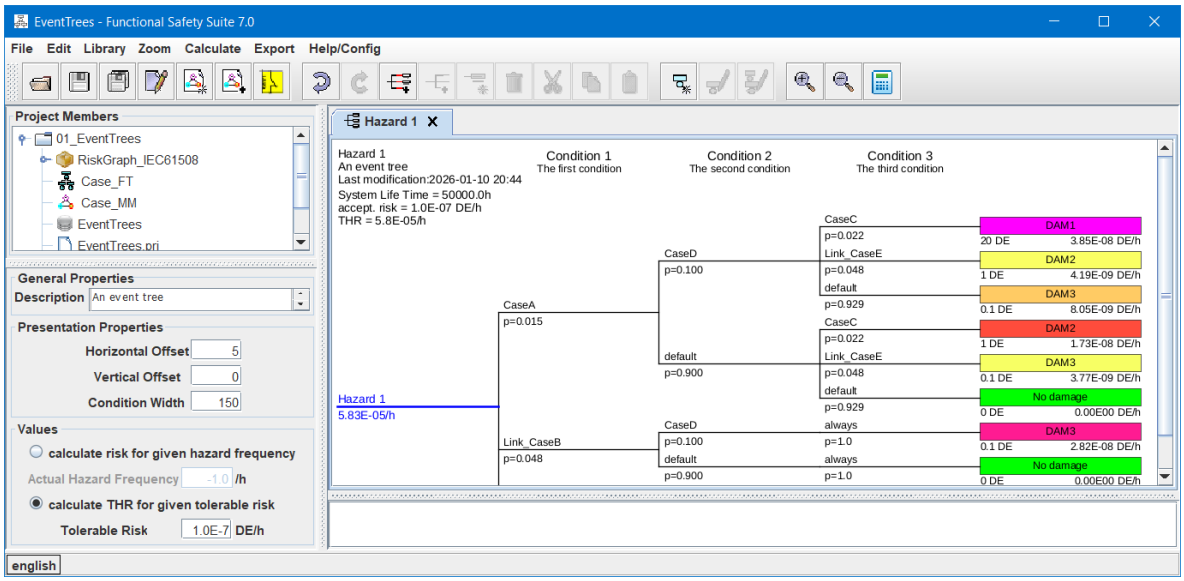


Figure 15: *The event tree properties panel*

Horizontal offset:

The margin between the window border and the left end of the hazard line.

Vertical offset:

The margin between the window border and the first damage.

Condition width:

The width of each condition column. Default is 150 pixel.

5.2.3 Values

Tolerable Risk:

If the THR shall be calculated, here the tolerable risk is to be entered. The unit is *Damage Equivalents per hour* (DE/h).

Actual Hazard Frequency:

If the risk shall be calculated for a given hazard frequency according to the selection in the *project properties dialog*, here the hazard frequency is to be entered.

5.3 The Condition and the Crotch Properties Panel

Each pass from one consequence to the consequence(s) of the next condition is a *crotch*. Each *condition* has as many *crotchs* as there are lines in the preceding *condition*: The first *condition* C1 has one *crotch*, since there is only one hazard. The second *condition* C2 has as

many *crotchs* as the first *condition* has *cases* n_{C1}, the third *condition* C3 has $n_{C1} \cdot n_{C2}$ *crotchs* and so on (except if the *always* flag is set, see below).

A *condition* is always selected together with one of its *crotchs* and vice versa, i. e. when you click on the *default case* or *always case*. Therefore the *condition properties panel* and the *crotch properties panel* are shown at the same time.

The parameters of *conditions* and *crotches* are stored in the event tree file (extension `.etf`).

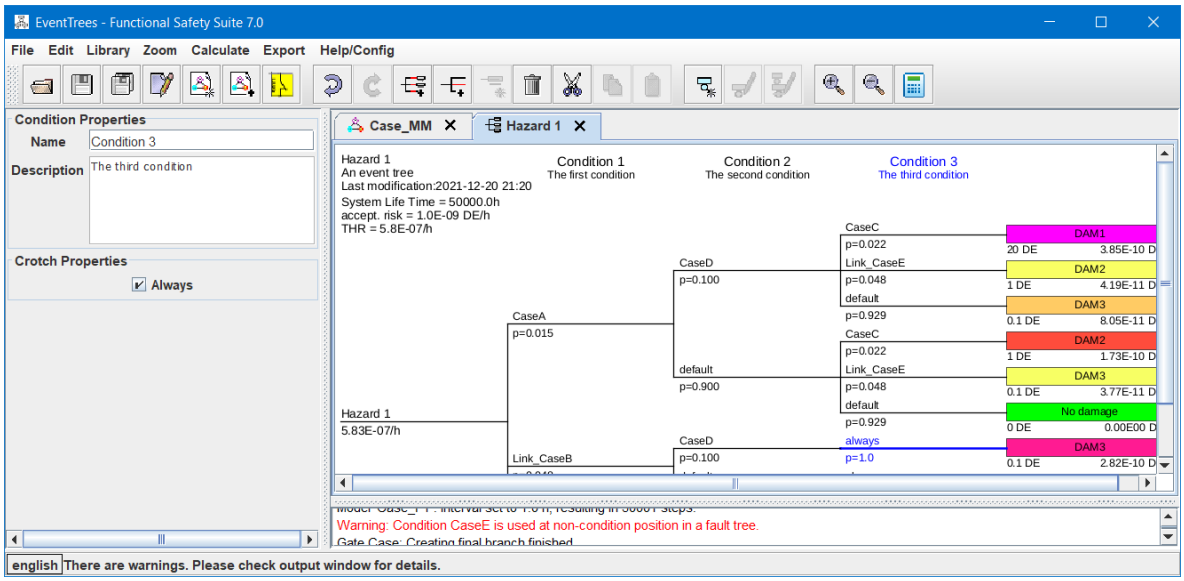


Figure 16: *The condition and the crotch properties panels*

5.3.1 Condition Properties

Name:

A user defined identifier of the *condition*.

Description:

A user defined description of the *condition*.

5.3.2 Crotch Properties

The lowest *case* of each *crotch* is usually the *default case*, which is the negated of all other cases of the *crotch*. However often due to the specific *case* of the previous condition, the cases of further conditions don't matter any more. If so, select the *crotch* by clicking on the *default case* and set the *always* flag in the *crotch properties panel* on the left – the *crotch* will be reduced to only one *case* with probability p=1.0. If you de-select the flag, the *crotch* will be expanded again, replicating the further branch for each *case* of the *condition*.

Note that selecting a *crotch* by clicking on the *default case* or the *always case* will also select the related *condition*.

5.4 The Case Properties Panel

A *case* of an *event tree* consists of the reference to the *generic basic event*, defined by its *package* and name, and the properties of this *generic basic event*.

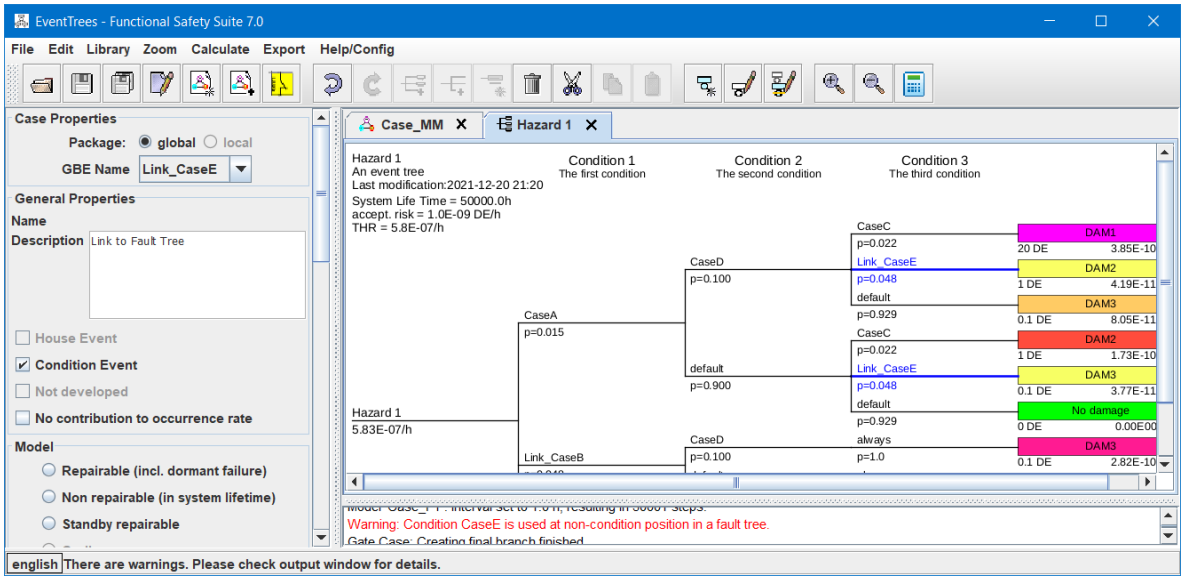


Figure 17: *The case properties panel*

5.4.1 Case Properties

Package:

Select whether the *generic basic event* is in the *library* of the *global package* or of the *local package*.

Name:

The identifier of the *generic basic event*. You can select a name (and by this the referred *generic basic event*) out of a list of the *generic basic events* belonging to the selected *package*.

5.4.2 Generic Basic Event

All parameters in this section belong to the *generic basic event*, thus they are stored in the *library*. Whenever one of these values is changed, this will effect also all other *basic events* and *cases* referring to this *generic basic event*, even in other models.

5.4.2.1 General Properties

Description:

A user defined description of the *generic basic event*.

5.4.2.2 Model

The probabilistic model of the *generic basic event*. See section 4.3 for details.

5.4.2.3 Values

The values needed by the model of the *generic basic event*. See section 4.3 for details.

5.5 The Damage Properties Panel

The final consequence is called *damage*, even if the severity is 0 ("no damage"). Typically several paths (via different cases) will lead to the same final scenario. Each possible scenario is called *generic damage*, describing its severity. The list of *generic damages* is part of the *event tree* and therefore stored in the `.etf` file. For each path, the severity can be selected out of the list of *generic damages*.

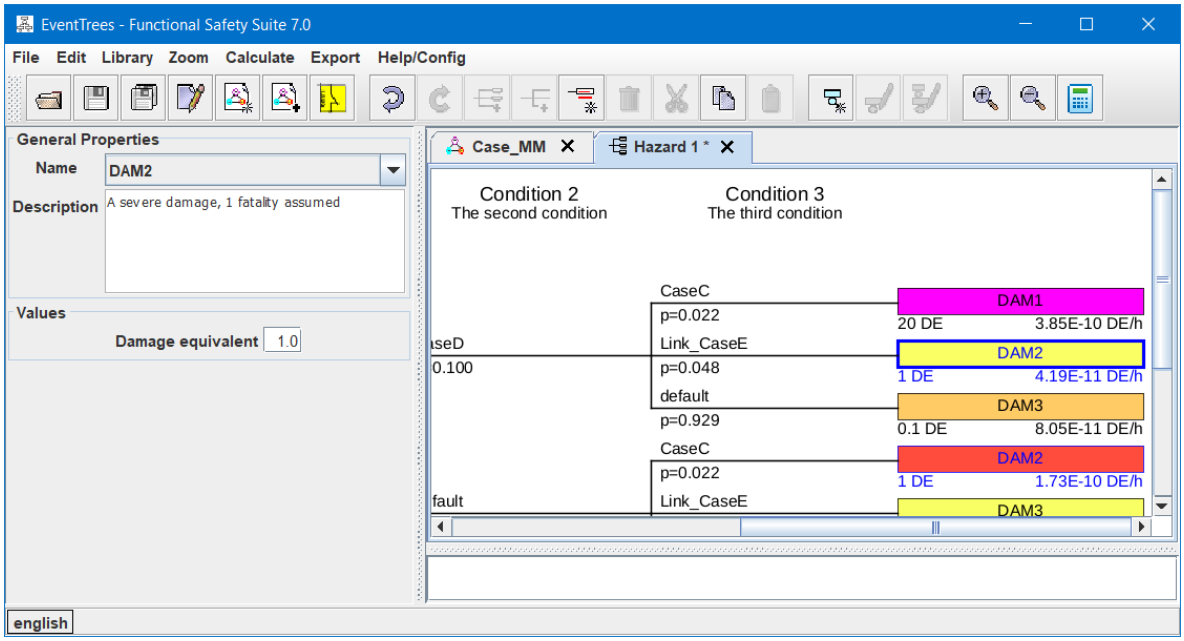


Figure 18: *The damage properties panels*

5.5.1 General Properties

Name:

A user defined identifier of the *generic damage*. Since *damages* with same name refer to the same *generic damage*, all other *damages* with the same name will be changed too whenever a

property belonging to the *generic damage* is changed in the *damage properties panel*.

You can select a name (and by this the referred *generic damage*) out of a list of the *generic damages* already created for this *event tree*.

Description:

A user defined description of the *generic damage*.

5.5.2 Values

The *damage equivalent*, indicating the severity of the damage.

5.6 Editing of Event Trees

After creation of a new *event tree* by **File – New Event Tree** a most simple event tree is shown.

The hazard is represented by the leftmost horizontal line. I can be selected by clicking on this line or somewhere above or below. If the hazard is selected, it is marked by a thicker, blue line and the *event tree properties panel* is shown on the left.

The damage is shown as rectangle on the right, including its name ("Default") and its severity in units of *Damage Equivalents* (DE). The name is given by the underlying *generic damage*. It is selected by clicking on it. If selected it is marked by a thick, blue border and the *damage properties panel* is shown on the left.

Usually there is not only one single, immediate consequence if the hazard occurs, but also other consequences leading to different *damages*, due to one or multiple *conditions*. A new *condition* is created and inserted by selecting the hazard (or a previous *condition*, see below) and **Edit – Add Condition**.

Each *condition* can take one or multiple *cases*. Each *case* has a certain probability that it is true when the hazard occurs. The sum of the probabilities of all *cases* of one *condition* must be equal to 1. Thus there is a *default case*, that applies if no other *case* is true. When a new *condition* is created, only the *default case* is created.

A *condition* is selected by clicking anywhere in the column belonging to the *condition* except on a line representing a specific *case*. If a *condition* is selected, its name and description is shown in blue text and the *condition properties panel* is shown on the left (see figure 16), so that name and description can be edited.

After selecting a *condition*, a *case* can be added to the condition by **Edit – Add Case**. Each *case* refers to a *generic basic event*, which determines its probability. Typically the *immediate* event model is used, allowing to directly enter the probability p. But also all other models that deliver a unavailability $\overline{Q}$ can be used, including *links*. When adding a *case*, the new *case* will refer to the last *generic basic event* in the local *library*.

A *case* is selected by clicking on one of the lines representing it. If selected it is marked by a thick, blue line and the *case properties panel* is shown on the left (see figure 17). Here you can change the *generic basic event* the *case* refers to. A new *generic basic event* is created by **Library – New Generic Basic Event**.

An *event tree* that has not been saved after the latest modification is marked with an asterisk '*' in its title.

5.7 Hints and Recommendations

If there are (multiple) conditions that must be fulfilled such that a (severe) damage can occur, the definition of the hazard is not unambiguous. For example think about a fire in a plane. You can define the burning paper in the toilet as a hazard, but you could also define the burning toilet room as a hazard, or you could consider all kinds of fire in other hazards, like loss of control equipment, loss of clean cabin air or loss of structural stability. Therefore you should clearly describe the relation between the identified hazards, thus that the sum of all hazard analyses has neither gaps nor significant overlaps. In particular, you should verify, that all identified hazards are "on the same level", i. e. that not one "hazard" is in fact just one basic event for another already defined "real" hazard.

In addition, depending on the definition of the hazard(s), the border between risk analysis and hazard analysis will shift significantly. As a general rule, you should never model barriers in an event tree, that rely on the correct function of technical elements ("active barriers", barriers that can fail on demand, i. e. PFD>0) that are part of the system under assessment, but only barriers that either exist independently of the system under assessment or that cannot fail on demand (PFD=0, i. e. "passive barriers"). That means, that the hazard should be defined in that way, that it includes the failures of all technical elements (as for example smoke detection and fire suppression systems).

If you — for whichever reason — consider active barriers in an event tree anyhow, it is mandatory to limit the PFD's to values, that reflect the systematic capabilities of the active barrier. E. g. for a barrier developed according to [EN 61508] for a safety integrity of SIL 2, its unavailability must not be set to values less than 1e-3 even its calculated PFD (by FTA or Markov model) is less. In addition, absence of common cause failures to the hazard or other barriers must be proven.

6 Architectures

6.1 Introduction

For each safety function, an architecture has to be created. If the architecture is simple, the standard electric (or pneumatic or hydraulic) schematic might be sufficient to describe the architecture. But in case of complex architectures, involving several sensors, relays, valves, computers etc. you should describe the architecture by means of block diagrams, supported by some verbal explanation. There are several norms related to electrical, pneumatic and hydraulic schematics for application in different areas such as machinery, automotive, process industry, railways or aerospace. The main purpose of all of these standards is to define the connections between components, i. e. which cable or pipe is necessary between which connector of a component or junction. None of these standards is focused on the functionality, or even functional safety. Therefore, according to the author's knowledge, whenever a functional block diagram is required, each company or even each engineer has created its own "block diagram language". Depending on effort and experience of the engineer or company, the quality of these block diagrams in terms of readability and level of detail differs quite a lot.

The *architectures* module of Functional Safety Suite aims to provide a functional block diagram language that is able to describe safety architectures in all fields of engineering. In a first step, you can use the module to draw functional block diagrams, requiring significantly less effort than using generic drawing tools. In a second step, an *architecture* can be automatically converted to a *fault tree*, given that some principles are respected and some additional information is provided.

6.2 Concepts

Each *architecture* consists of *architecture components*, see section 6.2.2. Each *architecture component* consists of one or multiple *component parts*, see section 6.2.1. Each *component part* is assigned a *component part symbol* used to visualize the *component part* and thus finally the component. Each *component part* has zero, one or two pins, which are necessary to connect the component to a *net*, see section 6.2.3.

Figure 19 shows a simple architecture.

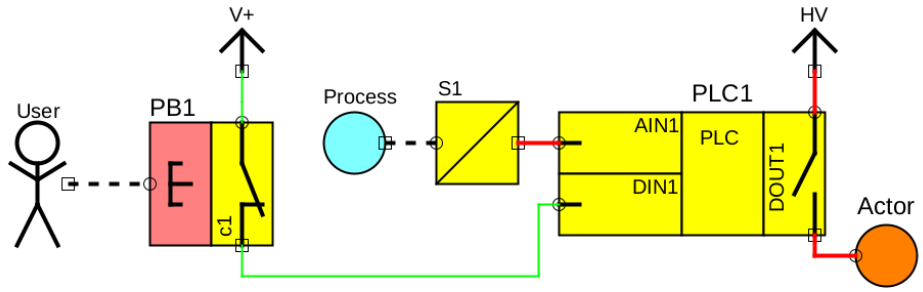


Figure 19: *Simple architecture*

6.2.1 Component Parts

There are four types of *component parts*:

SOURCE a *SOURCE* has one pin providing some information, energy or material. It can also be used to represent some process state, such as a physical entity like speed or temperature.

SINK a *SINK* has one pin receiving information, energy or material.

CONTACT a *CONTACT* has two pins. Depending on some other *component part*, there is an internal connection between the two pins or not, i. e. energy or material can pass from pin 1 to pin 2 or not. If the net represents an electric wire (and the contact represents an electric contact, therefore) the state representing connected pins is called "closed", the state representing disconnected pins is called "open". If the net represents a pneumatic or hydraulic pipe (and the contact represents a valve, therefore) the state representing connected pins is called "open", the state representing disconnected pins is called "blocked".

LOGIC a *LOGIC* part combines information, energy or material to new information, energy or material. It has no pins. A *architecture component* using a logic part must have at least one *component part* of type *SINK* and at least one *component part* of type *SOURCE* or *CONTACT* as well.

For each *component part* type, several common symbols are provided in the default symbol library. You can modify these symbols or create additional symbols in either this library or project specific libraries, see section 6.7.

6.2.2 Architecture Components

As written above, an *architecture component* consists of one or multiple *component parts*. There are five classes of *architecture components*:

SOURCE a component of class *SOURCE* consists of one *component part* of type *SOURCE*. It may be a source of information (e. g. a human), energy (e. g. a power line) or material, or just describe the state of a process such as speed or temperature. Each *architecture* shall have at least one component of class *SOURCE*.

ACTOR a component of class *ACTOR* consists of one *component part* of type *SINK*. Each *architecture* shall have at least one component of class *ACTOR*.

RELAY a component of class RELAY consists of one *component part* of type *SINK* and at least one *component part* of type *CONTACT*. It can be used to model switches and push-buttons, valves, or any other kind of component that has at least one output *CONTACT* (or valve way) which state depends on exactly one input.

LINE a component of class *LINE* consists of one *component part* of type *SINK* and one *component part* of type *SOURCE*. It is used to explicitly model transmissions, such as

repeaters, converters, sensors or couplers.

CONTROL a component of class *CONTROL* consists of one *component part* of type *LOGIC*, at least one *component part* of type *SINK* and at least one *component part* of type *SOURCE* or *CONTACT*. It is used to model more complex functional blocks, such as programmable logic controls, but can also be used for any other component with multiple inputs (e. g. a 3-position direction control valve).

Thinking about drawing block diagrams, the concept of components and component parts might look a bit complicated and over-engineered. The reason for having these definitions is in fact to enable Functional Safety Suite to automatically derive *fault trees* from *architectures*. But even if you don't use automatic *fault tree* creation, the concept will help you to create an architectural description that clearly shows the components that are relevant for safety and their interaction – not more and not less.

Let's go back to the architecture shown in figure 19. There are three *architecture components* of class *SOURCE*: 'User', 'V+' and Process'. The component 'Push-Button' is of class *RELAY*, using a part of type *SINK* (the button) and a part of type *CONTACT* (the contact named 'c1'). The component 'Sensor' is of class *LINE*, using a part of type *SINK* (including the left hand pin and some elements of the graphics) and a part of type *SOURCE* (including the right pin and some other elements of the graphics). The component 'control' is of class *CONTROL*, using two parts of type *SINK* ('AIN1' and 'D_IN1'), a part of type *LOGIC* ('PLC') and a part of type *CONTACT* ('DOUT1'). The component 'Actor' is of class *ACTOR* and uses one part of type *SINK*.

6.2.3 Nets and Pins

Each pin must be connected to at least one other pin by some net. A net represents a logical or physical connection, the user pressing a button, the temperature affecting the sensor, or of course the voltage of the source 'V+' connected to the push button contact and control output. In order to simplify reading of an *architecture*, different types of connections are distinguished by different line styles. The type is also considered when a *fault tree* is derived from the *architecture*, but only for some plausibility checks.

There are two types of pins: A square marks an output pin, a circle represents an input pin. If you want to derive a *fault tree* from your *architecture*, the rules stated in section 6.6.1 have to be fulfilled. In particular, each net needs exactly one output pin usually. Multiple outputs might be possible according to the rules stated in section 6.6.1.

6.3 The Architecture Properties

Figure 20 shows the architecture properties, including presentation related properties and some information for deriving *fault trees*.

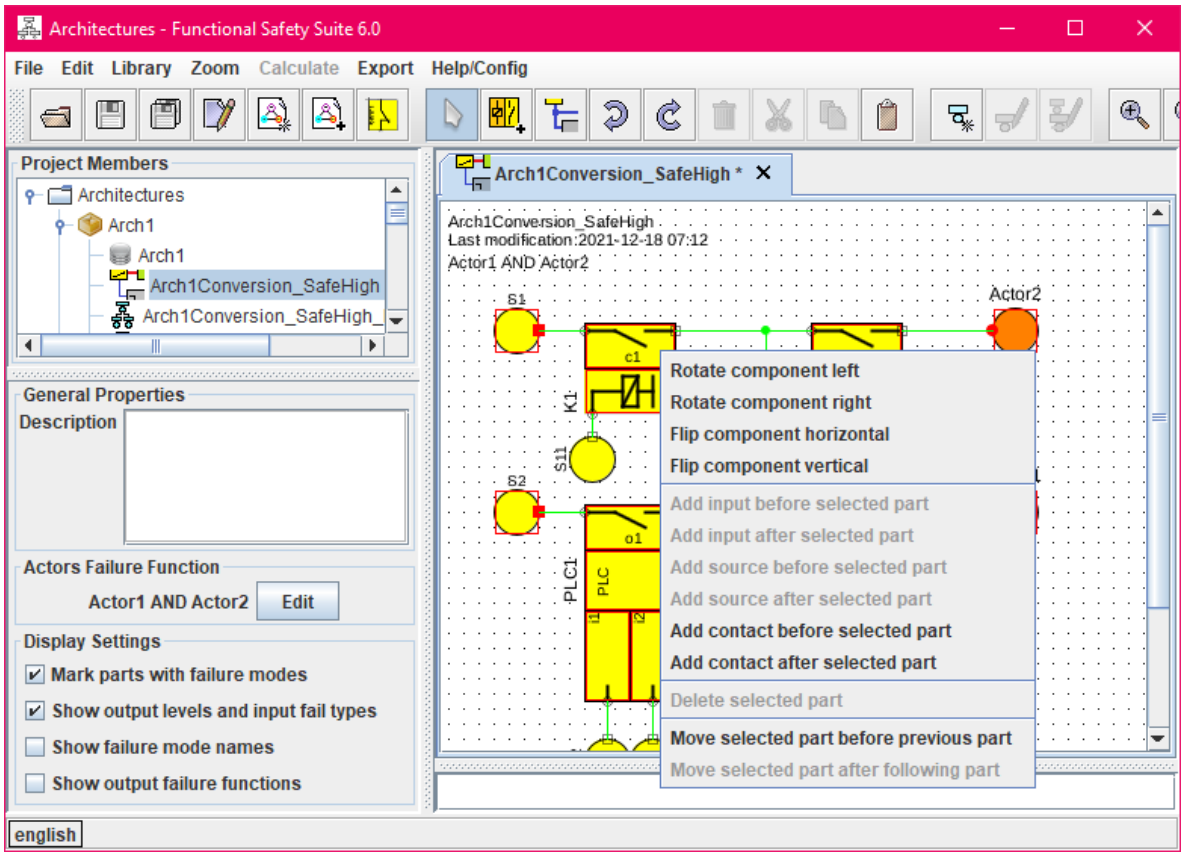


Figure 20: *The architecture properties panel and an open component pop-up menu*

6.3.1 General Properties

Description:

A user defined description of the *architecture*.

6.3.2 Output Failure Function

In case of multiple actors, you have to state which combination of actor failure is critical before you can derive the *fault tree*, see section 6.6.3.

6.3.3 Display Settings

Here you can define which of the information relevant for deriving a *fault tree* is shown or not. By default, neither the failure mode names (name of the *generic basic events* connected to a *component part* are not shown, nor the combinations of input failures in case of *architecture components* of type Logic. These checkbox values are not saved.

6.4 Component and Component Part Properties

If you click into a *component part*, both the properties of the *architecture component*, the selected *component part* belongs to, and the properties of the *component part* will be shown on the left for editing, see figures 21 and 25.

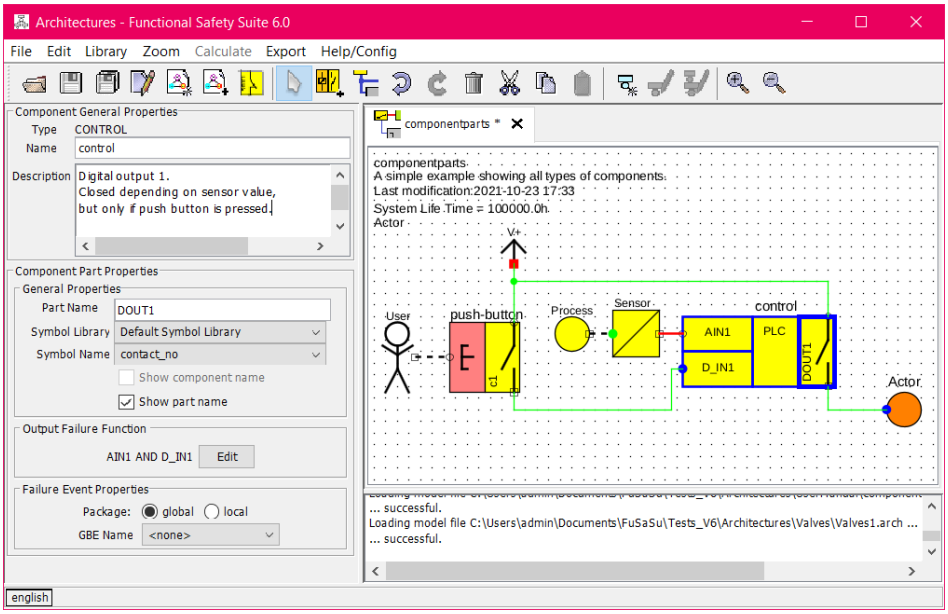


Figure 21: *The component and component part properties panel of a CONTACT*

6.4.1 Component General Properties

The type of the *architecture component* is stated, followed by a field for editing the name and an optional description. Note: You may assign the same name to multiple *architecture components* in order to state, that this is actually only one single unit, e. g. a PLC with is used at different locations in the *architecture*(with different inputs and outputs).

6.4.2 Component Part Properties

6.4.2.1 General Properties

Choose a name for the selected *component part* and select a *component part symbol* out of the available symbols for this *component part*'s type and the selected symbol library. If the selected *architecture component* consists of multiple *component parts*, you will switch off showing the *architecture component* name for all *component parts* except of one. Sometimes you might also want to switch off showing the *component part*'s name.

6.4.2.2 Failure Event Properties

Each *component part* can be assigned exactly one failure mode, which is a reference to some *generic basic event* in the *library* of this *package* or the global *package*. When creating a *fault*

tree for this *architecture* automatically, a *basic event* will be created for this *component part*, referring to the given *generic basic event*. The suffix of the *basic event* will be derived from the name of the *architecture component* and the name of the *component part*. See section 4 for details of the failure modes, and section 6.6 for how to derive a fault tree.

Further properties of the *component part* will depend on the types of the selected *architecture component* and *component part*:

6.4.2.3 Output Level

If the selected *component part* is of type *SOURCE*, the output level has to be selected. See section 6.6.1 for how to use output levels and input fail safe types.

6.4.2.4 Input Fail Safe Type

If the selected *component part* is of type *SINK*, you can state whether there is some safe side input. See section 6.6.1 for how to use output levels and input fail safe types.

6.4.2.5 Output Failure Function

If the selected *component part* is of type *SOURCE* or *CONTACT* and is part of a *architecture component* of class *CONTROL*, and the *architecture component* has multiple inputs, you have to state, which combination of faulty inputs will lead to a failure of the output, see section 6.6.2.

6.4.3 Changing the Component's Structure

If you right-click on a *architecture component*, a menu will pop-up, see figure 20. You can in particular add another *component part* to a component, change the sequence of the *component parts* within a *architecture component*, or flip or rotate the complete *architecture component*. Deleting a *component part* is only possible if it is not connected to a *net*.

The possibilities to add or delete *component parts* depends on the class of the *architecture component* and its limitations, of course.

6.5 Net Properties

Pins of *component parts*are connected by *nets*.

6.5.1 Net General Properties

You can give a label to a *net*, which will be shown beside of the longest section of the *net*. If you don't, a default label will be used. Labels starting with 'net' won't be shown in the graphics.

6.5.2 Net Type

The *net* type will change color and width of the lines of the *net*. In addition, the available safe side options of inputs connected to the *net* might be reduced for some types. The gate descriptions of a derived *fault tree* will also depend on the *net* type.

6.6 Deriving a Fault Tree

Functional Safety Suite can automatically derive a *fault tree* for a given *architecture*, if certain structural requirements are fulfilled and some additional information is given:

- each pin must be connected to a *net*,
- each *net* must be connected to at least one source, which might be a *component part* of type *SOURCE* or the output pin of a *component part* of type *CONTACT*,
- each *net* must be connected to at least one *component part* of type *SINK*.

Figure 22 shows a typical control loop: An oven burning some liquid fuel is used to heat something to a given temperature. The temperature set-point is entered by a used via a terminal, connected to a PLC 'ControlPLC'. The actual temperature is measured by a sensor, which is connected to a PLC as well. The PLC controls a proportional valve, controlling the fuel flow to the oven. If the oven temperature exceeds some maximum, a hazard occurs.

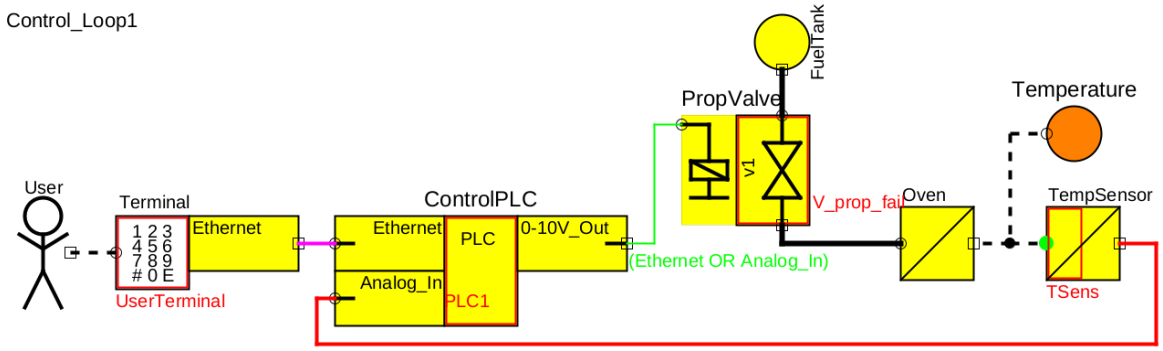


Figure 22: *Converting an architecture to a fault tree.*

Given a suitable control algorithm in the PLC, this system might work fine, as long as none of the components fail. But in reality, some of the components can fail: The input terminal can falsify the entered value, the PLC can output a wrong analogue value to the valve due to an internal fault, the valve can be stuck, the temperature sensor can output a value not reflecting the actual temperature. These failures are considered by assigning a *generic basic event* to one of the *component parts* of each of these *architecture components*– visualized by a red rectangle around the symbol of the *component part*. Many more failure modes can be distinguished, of course, and in fact you could assign a failure mode to each of the *component parts*. However, usually it is possible to consider all failure modes of a component within one single failure event. Only in case a *architecture component* of class *CONTROL* or a *RELAY* has multiple outputs, you might usually want to assign separate failure modes (*generic basic*

events) to each of the output *component parts*. The oven itself cannot fail dangerously, it just burns the fuel that it gets. Therefore, no failure mode is assigned to any of the *component parts* of the oven.

The name of the referred *generic basic event* is not shown by default, but it can be shown by selecting 'Show failure mode names' in the 'Display Settings' in the properties panel.

If you click **Edit – Convert to Fault Tree**, Functional Safety Suite will try to create a *fault tree* representing the failure characteristic of this architecture. The algorithm will parse the architecture starting with the actor(s). Here, we have only one *component part* of class *ACTOR*, the 'Temperature'. The 'Temperature' fails, if the oven doesn't work as expected. As long as the oven gets the correct amount of fuel, it is assumed to behave as expected (no failure mode assumed). Thus, the 'Temperature' can only fail, if the fuel flow to the oven doesn't fit. The fuel flow is determined by the proportional valve – if the valve doesn't close correctly, the fuel flow will be too high and the temperature will fail (become too high). The valve fails, if it is either defect, OR if it gets a wrong voltage by the control PLC. The PLC output fails, if either the PLC itself is defect, OR if a wrong set-point is sent by the terminal, OR if the temperature value used by the control algorithm is not correct (see section 6.6.2 for how to tell the algorithm which gate(s) to create for an output of a *CONTROL*). The temperature value is not correct, if the temperature sensor is defect.

In the next recursion, the algorithm would follow the input of the sensor via the oven back to the valve again – and also again to the PLC, and so on. In fact, the algorithm would detect that there is a loop and abort conversion, but it will not automatically resolve this situation. You have to tell the algorithm explicitly, that it shall not trace down further. This is achieved by setting the *Input Fail Safe Type* of the temperature sensor to 'no dangerous' in the properties of the input *component part*, see figure 25. The input pin will now be presented in green color. In figure 23 the resulting fault tree is shown.

Obviously, there are only OR gates created in this conversion – what is correct in this case. There are three principles how to define redundancies and fail-safe behavior, finally leading to AND gates in the conversion, where appropriate:

A. By stating levels of sources and fail safe characteristics of inputs, see section 6.6.1.

B. By defining failure functions for outputs of *architecture components* of class *CONTROL*, see section 6.6.2.

C. In case of multiple actors: by defining a failure function for the actors, see section 6.6.3.

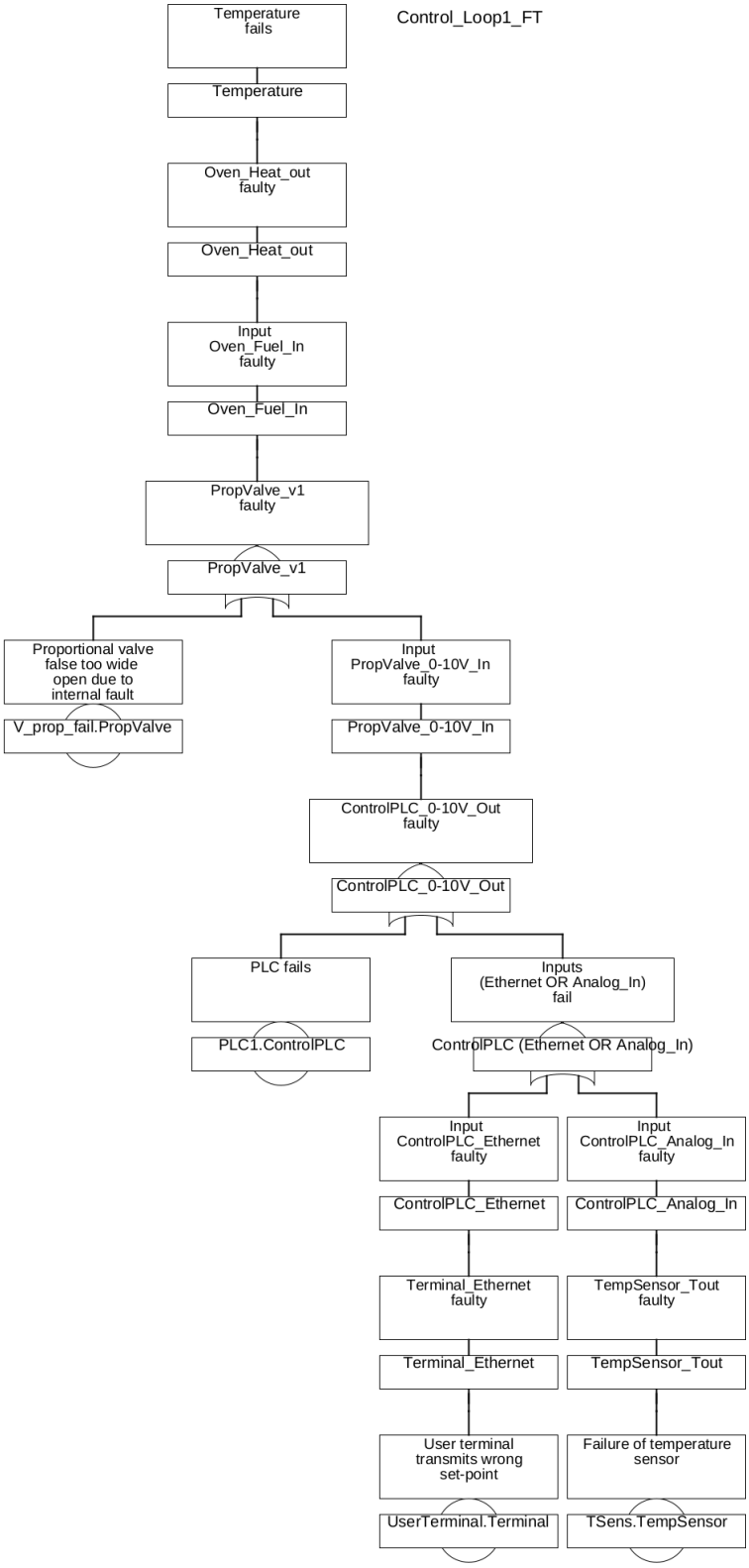


Figure 23: *Fault tree for architecture shown in figure 22.*

6.6.1 Input Fail Safe Types and Source Output Levels

In order to derive a fault tree from a given block diagram, the logical architecture in terms of redundancies or monitoring channels must be known. For this task, additional information is necessary in general.

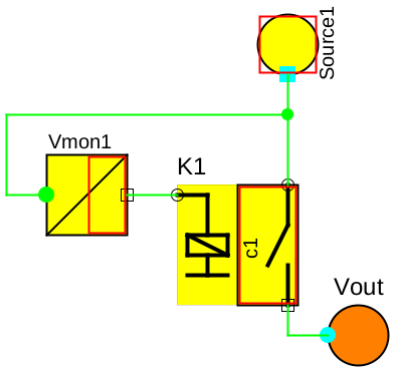


Figure 24: *Usage of safe low or open input type.*

Without further information, the diagram shown in figure 24 could be interpreted as

- The output voltage fails (is not available), if the source fails OR the contact is open.
- The output voltage is too high, if the source creates too high voltage AND the contact is closed.

There is no way how the algorithm could determine by itself, which of both interpretations is meant by the creator (i. e. for which reason the relay has been added). But this information is mandatory, since in the first case, an OR gate has to be created, whereas in the second case an AND gate has to be created.

The necessary information is stated in terms of the *Input Fail Safe Type* of an input in combination with the *Output Level(s)* of the source(s) connected to a path. The term path denotes the overall connection from the output pin of a *component part* of type *SOURCE* to the input pin of a *component part* of type *SINK*. A path may contain multiple *nets*, which are separated by *component parts* of type *CONTACT*.

There are five *Output Levels* defined for sources (refer to figure 25):

- (weak) unknown
- weak low
- weak high
- strong low
- strong high

Together with the *Input Fail Safe Types* all typical architectures can be described as defined below. The following definitions are based on the assumption, that a strong source can fail only open (i. e. doesn't provide anything, but a strong high source cannot turn into a low

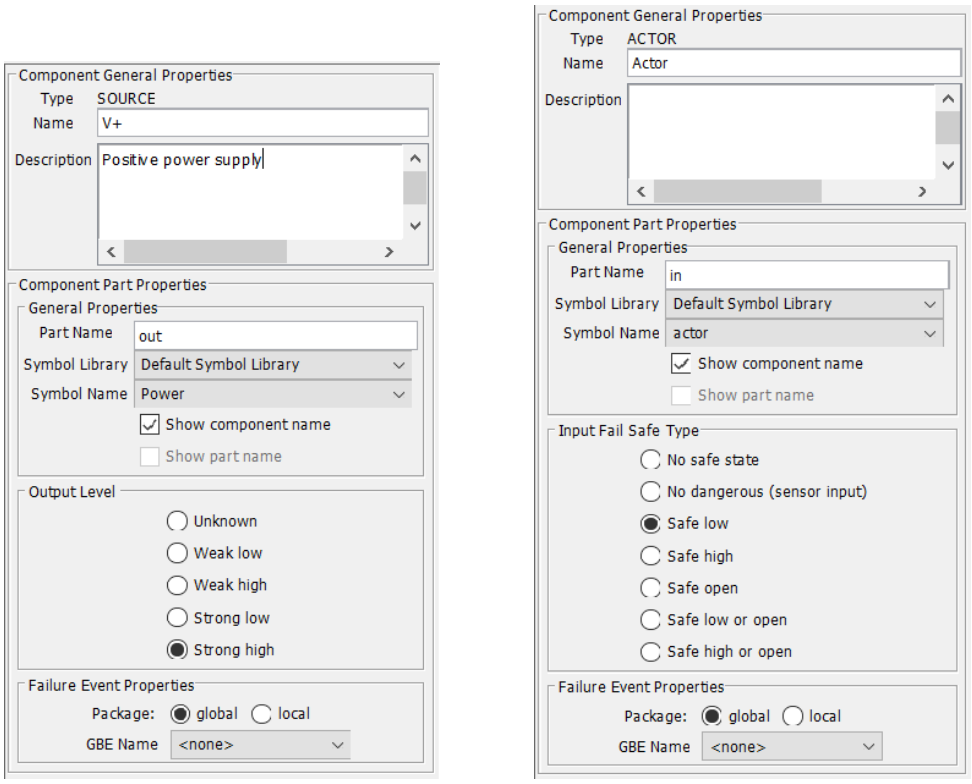


Figure 25: *The component and component part properties panel of a SOURCE and a SINK (being part of a architecture component of class ACTOR)*

source by fault and vice verse), whereas a weak source can fail to the opposite (i. e. a weak high source can turn into a weak low source by fault and vice verse):

- The default *Input Fail Safe Type* is No safe state, which leads to an OR gate independent of the source(s).
- The No dangerous option is used, if no event shall be created for the signal connected to this input, i. e. in order to explicitly stop the algorithm from following the signal further down. It is typically used for sensors, as shown in figure 22 and in figure 24.
- Safe low states that the input must be connected to a low source, in order not to fail. Therefore, the following rules apply:
  - there must be a path to at least one *strong low* source or to a *weak low* source.
  - there must be at most one weak source (*weak low* source, *weak high* source or *(weak) unknown* source)
  - if the net is connected to a *strong low* source and doesn't contain a *weak low* source: – The input fails if all paths to *strong low* source(s) fail (any contact of each path fails) or the source(s) fail.
  - if the net is connected to a *strong low* source and contains a *weak low* source:

  - The input fails if all paths to *strong low* source(s) or the source(s) fail AND the path to the *weak low* source fails (any contact of the path fails) or the *weak low* source fails.
- if the net is not connected to a *strong low* source (but a *weak low* source, see first rule):
  - The input fails if the path to the *weak low* source or the *weak low* source fails.

- Safe high is the opposite: it states that the input must be connected to a high source, in order not to fail. Therefore, the following rules apply:
  - there must be a path to at least one *strong high* source or to a *weak high* source.
  - there must be at most one weak source (*weak low* source, *weak high* source or *(weak) unknown* source)
  - if the net is connected to a *strong high* source and doesn't contain a *weak high* source:
    - The input fails if all paths to *strong high* source(s) fail (any contact of each path fails) or the source(s) fail.
  - if the net is connected to a *strong high* source and contains a *weak high* source:
    - The input fails if all paths to *strong low* source(s) or the source(s) fail AND the path to the *weak high* source fails (any contact of the path fails) or the *weak high* source fails.
  - if the net is not connected to a *strong high* source (but a *weak high* source, see first rule):
    - The input fails if the path to the *weak high* source or the *weak high* source fails.
- The Safe open option indicates an input that is safe, if not connected to a source. Therefore, the following rules apply:
  - the input fails if any of the paths to any sources fails (OR gate for parallel paths).
  - a path fails, if all contacts fail AND the source fails (if no failure event is assigned to the source, the source is ignored).
- the Safe low or open option states, that the input must either be connected to a low source or not connected to any source in order not to fail. Therefore, the following rules apply:
  - there must be a path to at least one *strong low* source or to a *weak low* source.
  - there must be at most one weak source (*weak low* source, *weak high* source or *(weak) unknown* source)
  - if the net is connected to a *strong low* source and doesn't contain a weak source:
    - The input cannot fail, because it is always low or open.

  - if the net is connected to a *strong low* source and there is a *weak high* source or *(weak) unknown* source:
    – The input fails if the path to the weak source fails (all contacts fail) AND all paths to *strong low* source(s) fail (any contact in each path fails) or the *strong low* source(s) fail.
  - if the net is connected to a *strong low* source and contains a *weak low* source:
    – The input fails if all paths to *strong low* source(s) fail (any contact in each path fails) or the *strong low* source(s) fail AND the path to the *weak low* source fails (all contacts fail) AND the *weak low* source fails (if it can fail, else the *weak low* source is ignored).
  - if the net is not connected to a *strong low* source (but a *weak low* source, see first rule):
    – The input fails if the path to the *weak low* source fails (all contacts fail) AND the *weak low* source fails (if it can fail, else the *weak low* source is ignored).

- the Safe high or open option is just the opposite: it states, that the input must either be connected to a high source or not connected to any source in order not to fail. Therefore, the following rules apply:
  - there must be a path to at least one *strong high* source or to a *weak high* source.
  - there must be at most one weak source (*weak low* source, *weak high* source or *(weak) unknown* source)
  - if the net is connected to a *strong high* source and doesn't contain a weak source:
    – The input cannot fail, because it is always high or open.
  - if the net is connected to a *strong high* source and there is a *weak low* source or *(weak) unknown* source:
    – The input fails if the path to the weak source fails (all contacts fail) AND all paths to *strong high* source(s) fail (any contact in each path fails) or the *strong high* source(s) fail.
  - if the net is connected to a *strong high* source and contains a *weak high* source:
    – The input fails if all paths to *strong high* source(s) fail (any contact in each path fails) or the *strong high* source(s) fail AND the path to the *weak high* source fails (all contacts fail) AND the *weak high* source fails (if it can fail, else the *weak high* source is ignored).
  - if the net is not connected to a *strong high* source (but a *weak high* source, see first rule):
    – The input fails if the path to the *weak high* source fails (all contacts fail) AND the *weak high* source fails (if it can fail, else the *weak low* source is ignored).

In the architecture shown in figure 24, the safe state safe low or open is used, in combination with a *weak low* source, telling the conversion algorithm that a loss of the output voltage

'Vout' doesn't matter, but only a (too) high 'Vout'. The source 'Source1' is usually 'low', i. e. correct in the view of the algorithm. Only if the 'Source1' is faulty (provides too high voltage) AND the contact 'c1' of relay 'K1' doesn't open, the output voltage 'Vout' will fail (be too high). The resulting fault tree is shown in figure 26.

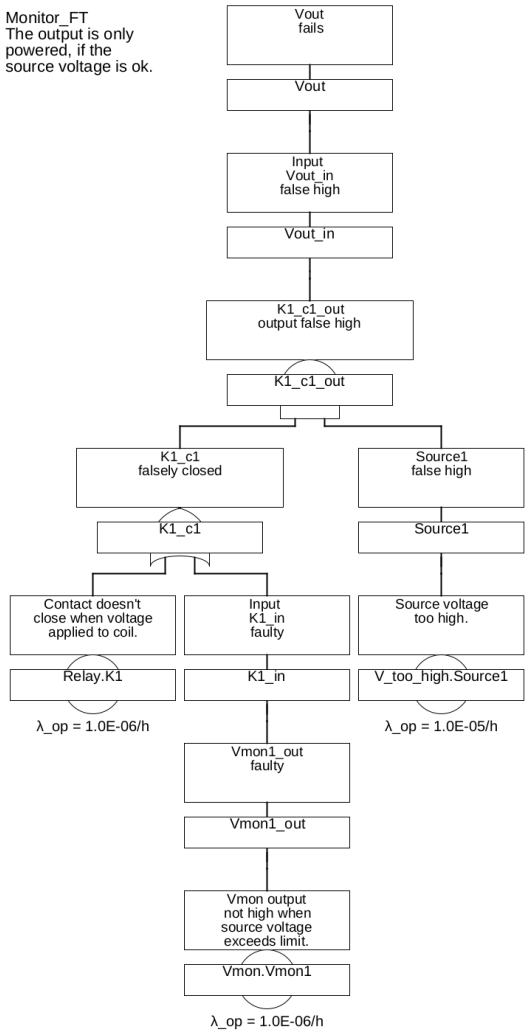


Figure 26: *Fault tree for architecture shown in figure 24.*

In figure 27 an example using the safe low and the safe open input types is shown:

Standard railway brakes are applied, if the brake pipe pressure is below 3.5 bar. If the brake pipe pressure is 5.0 bar, the brakes are released. Between 3.5 bar and 5.0 bar, the brakes are partially applied. The brake pipe pressure is controlled by some electro-pneumatic equipment, here shown as a brake pipe control unit 'BPC', based on the position of the 'Brake Handle' and information from other systems, e. g. automatic speed controls or safety systems. The automatic train protection system 'ATP' shown in figure 27 commands an emergency brake by cutting the voltage of the coil of two emergency brake valves 'EBV1' and 'EBV2', which in turn open a wide section towards the open air each, so that the brake pipe pressure will

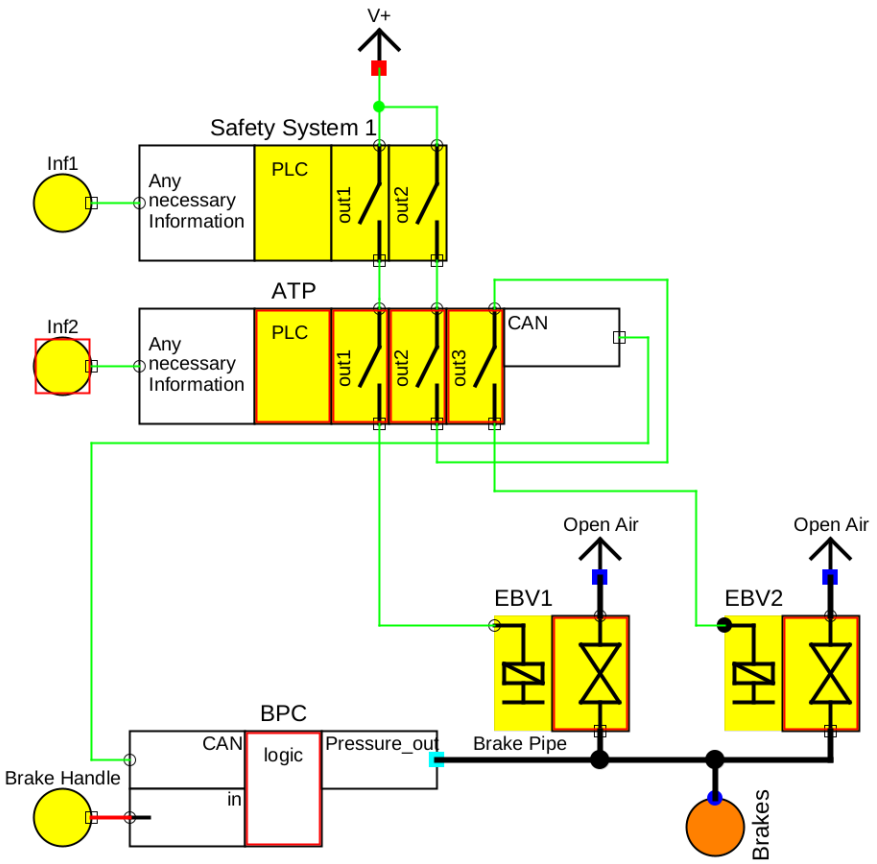


Figure 27: *Usage of safe open and safe low input types.*

drop to some very low value, even if the *weak low* source 'Pressure_out' of the brake pipe control unit 'BPC' provides a high pressure (>3.5 bar). In addition to this hard-wired cut-off, it is assumed that the ATP provides the brake command via CAN bus to the 'BPC' in addition, and that the 'BPC' creates a brake pipe pressure according to the most restrictive input information (in absence of faults).

The safe low actor 'brakes' only fails (to create brake effort), if there is neither a connection to any of the *strong low* sources 'Open Air' by fault AND the 'BPC' provides high pressure by fault. The resulting fault tree is shown in figure 28.

The coil of 'EBV2' in figure 27 is marked as safe open in order to indicate, that it is sufficient if any of the two contacts 'out2' or 'out3' opens (doesn't fail), i. e. that an AND gate is created above 'out2' and 'out3'. Without the safe open marking, an OR gate would be created above the 'out2' and 'out3' contacts. The contacts of 'Safety System 1' are not considered in the fault tree at all, since no failure event is referred by any part of the 'Safety System 1' and its inputs. This system is not considered for this function, therefore. In fact you shouldn't mention any components not related to the safety function, in order to avoid any wrong gates in the fault tree.

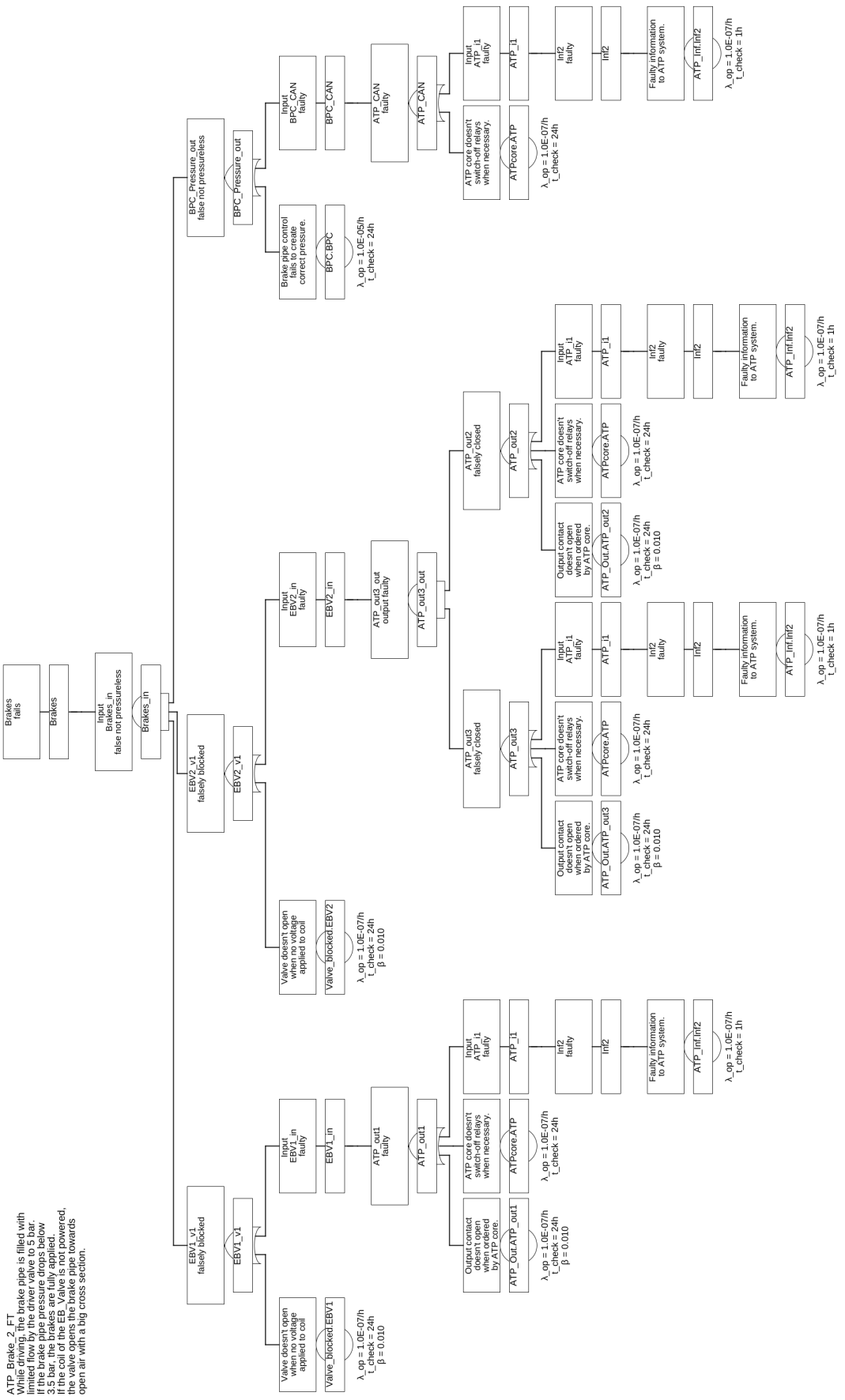


Figure 28: *Fault tree for the architecture shown in figure 27*

6.6.2 Output Failure Functions

A *architecture component* of class *CONTROL* can have multiple inputs (*component parts* of type *SINK*). In that case, you've to tell the conversion algorithm which input or which combination of inputs has to fail, in order to make the output fail, i. e. in which way the output depends on the input(s). This is done by stating a boolean *output failure function* for each output by using the *output failure function input window*. The window will open when you press **Edit** in the *Output Failure Function* panel of the selected output (see figure 21).

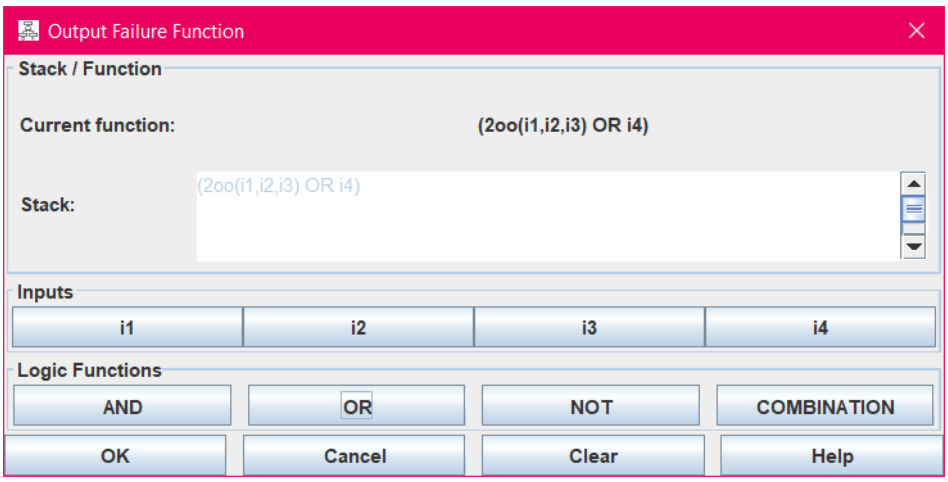


Figure 29: *Output failure function input window.*

The formula is entered in reverse Polish notation (RPN), because no parentheses () are needed in RPN. In reverse Polish notation, the operators follow their operands; for instance, to combine A and B by the boolean OR operator, one would enter A B OR rather than A OR B. If there are multiple operations, operators are given immediately after their last operand; so the expression written A AND (B OR C) in conventional notation would be entered A B C OR AND in reverse Polish notation. Since (B OR C) AND A is equivalent, one could also enter B C OR A AND.

In functional safety, an 'M-out-of-N' operation is needed frequently (the so-called COMBINATION gate). This is no boolean operator, but a combination of boolean operations in fact, with unknown number of operands. In the failure functions dialog, a COMBINATION gate is entered by pressing COMBINATION after the second operand and each additional operand. When you press COMBINATION after the second operand, you'll be asked to enter the minimum number of critical failures M for this combination.

The RPN is implemented by a stack. The operator will always be applied to the top one or two operands on the stack. The stack is shown in the stack field in the dialog (growing downwards, i.e. the "top" is the lowest line).

Examples Note: The '!' indicates the NOT operation, '*' the AND operation, '+' the OR operation.
Note: The NOT operator has highest priority, followed by the AND operator, and finally the

OR operator. I. e. ((!A) * B * C) + D is equivalent to !A * B * C + D.

Table 3: *Examples for how to enter a boolean formula*

Formula	**Input**
(i1 + i2 + i3) * i4	i1 i2 OR i3 OR i4 AND
i1 * i2 * i3 + i4	i1 i2 AND i3 AND i4 OR
i1 * i2 * (i3 + i4)	i1 i2 AND i3 i4 OR AND (or i1 i2 i3 i4 OR AND AND)
(i1 + i2) * (i3 + i4)	i1 i2 OR i3 i4 OR AND
(i1 + i2 + i3) * i4 + i2 * (i1 + !i3 * !(i1 + i4))	i1 i2 OR i3 OR i4 AND i2 i1 i3 NOT i1 i4 OR NOT AND OR AND OR (or i1 i2 OR i3 OR i4 AND i1 i4 OR NOT i3 NOT AND i1 OR i2 AND OR)
i1 * 2-out-of(i2,i3,i4,i5)	i1 i2 i3 COMBINATION (2) i4 COMBINATION i5 COMBINATION AND
i1 * 2oo(i2 + i3, i4 + i5, i6 + i7)	i1 i2 i3 OR i4 i5 OR COMBINATION (2) i6 i7 OR COMBINATION AND

6.6.3 Actor Failure Functions

If there are multiple *architecture components* of class *ACTOR*, you have to tell the algorithm which combination of failures of actors makes the overall system fail. This is done in the same way as for the output of a *CONTROL*. The window will open when you press **Edit** in the *Actor Failure Function* panel of the overall *architecture*, which is shown when nothing is selected (see figure 20).

6.6.4 Options

In the *project properties dialog* you can select whether the derived *fault tree* is automatically split into sub-trees at certain levels:

- In case of multiple actors, the top tree will refer to individual sub-trees for the failure of each actor by default.
- Often identical branches are needed at several locations in the fault tree. Branches used multiple times in the overall tree will be separated as sub-trees by default.

6.7 Symbol Libraries and the Symbol Editor

The symbols available to present a *component part* are collected in *symbol libraries*. There is a *default symbol library* delivered with Functional Safety Suite (file `standard_symbols.sym` in the `XML` directory below the installation directory). When you create a new project, a copy of the *default symbol library* will be created in the project directory.

The symbols can be edited and new symbols can be created using the *Symbol Editor*, see figure 30.

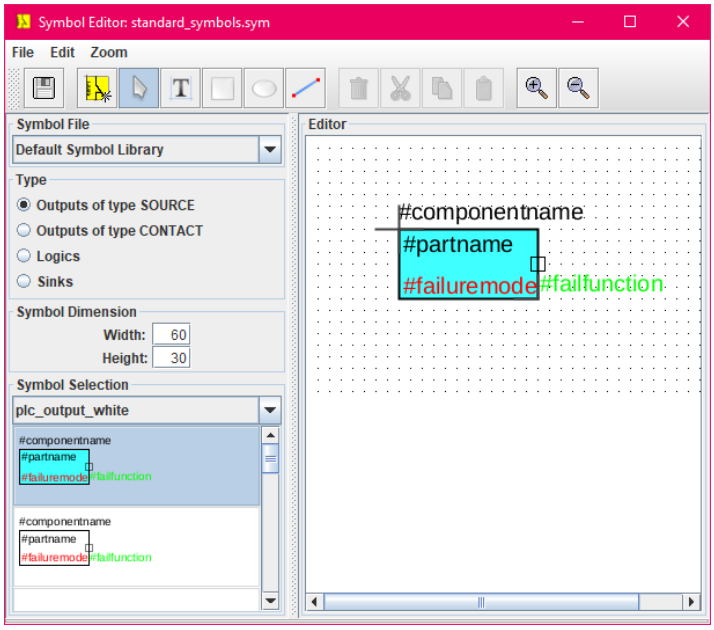


Figure 30: *The Symbol Editor*

As an alternative, the symbol library files `.sym` can be edited with any text editor (Notepad++, www.notepad-plus-plus.org is highly recommended). This might be necessary if you want to copy symbols between different libraries or delete certain symbols you don't need any more.

First of all you have to select the library that contains the symbol that you want to edit or where you want to create the new symbol.

You can create a new *project specific symbol library* by **File – Create new Symbol File** in the menu of the *Symbol Editor*. The new file will be stored in the project directory and only be available in this project. You can copy the file manually to another project, it will be available after (re-)loading the other project without further action.

Then select the type of the symbol you want to change or created in the *Type* panel. Finally either select **Edit – Create Symbol** or select the existing symbol out of the list shown below.

If you create a new symbol, you'll be asked for its name in a dialog window. A new symbol will be created with default size and all available placeholder text fields (see section 6.7.3 below).

6.7.1 Symbol Dimension

Select the symbol size by entering width and height. The dimension will be used to arrange the symbols or the *component parts* within a *complex component*, in combination with the location of the pin(s). Note that the graphic elements (i. e. text, rectangle, ellipse, polygon lines) are not considered for the arrangement.

6.7.2 Pins

Pins cannot be created or deleted, since they are defined by the symbol type. Thus, they can only be moved with **Shift** – **Up/Down/Left/Right** and only on the grid.

6.7.3 Fix Text and Text Placeholders

A symbol can contain fix texts as well as text fields that will be replaced in the actual architecture by the name of the *architecture component*, the *component part*, the name of the referred *generic basic event* (if assigned) or the failure function (only used for outputs). The names of the placeholders are shown in figure 31.

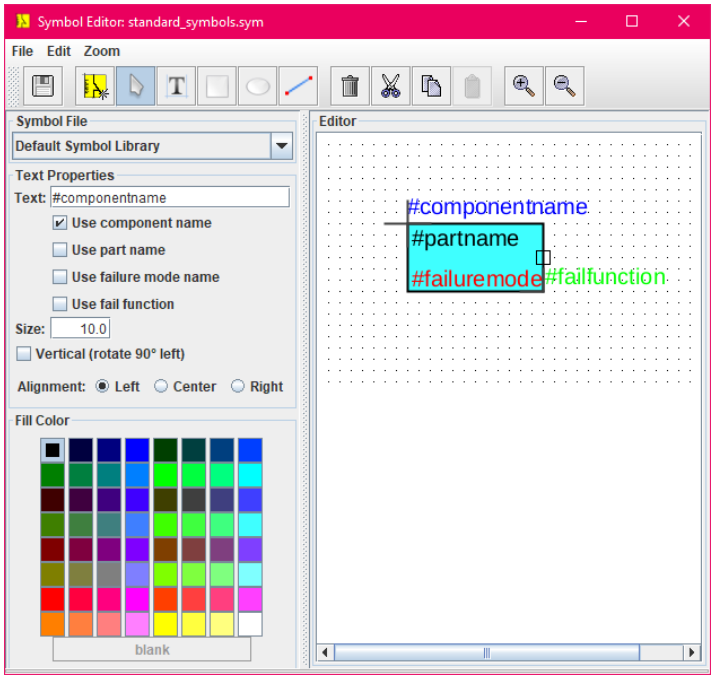


Figure 31: *The Symbol Editor if a text is selected*

A new text is added by **Edit** – **Add Text Mode** and click where you want to place it. Change mode to **Edit** – **Select/Modify Mode** in order to move the text. If you select one of the checkboxes, the related placeholder will be assigned to the selected text. Finally select text size, orientation and color.

6.7.4 Rectangles and Ellipses

A new rectangle is added by **Edit** – **Add Rectangle Mode**, a new ellipse is added by **Edit** – **Add Ellipse Mode**, then press the mouse button at one corner and drag the mouse to the diagonally opposite corner (in case of an ellipse, it will be fitted inside this virtual rectangle). Finally select stroke width, stroke color and fill color.

6.7.5 Polylines

A new polyline is added by **Edit − Add Polyline Mode**. Each click will create a new corner of the line. The line is finished by pressing **Cancel** or selecting another Edit Mode. Finally select stroke width and stroke color.

6.7.6 Move and Resize

You can move and resize graphic elements and pins with the mouse or with the arrow keys while either **Shift** or **Alt** is pressed. If **Shift** is pressed, the selected item is moved by 5 pixels, if **Alt** is pressed by 1 pixel. Pins will always be placed on the grid (if you edit the library XML file with a text editor, make sure that only multiples of 5 are used for pin coordinates).

6.7.7 Saving the Changes

After you've modified a symbol or created a new symbol, you'll want to save it.

If you've changed a symbol of the *default symbol library*, you'll be asked if you want to copy all symbols in the (project specific) *default symbol library* to the overall *default symbol library* in the Functional Safety Suite directory (which is used when a new project is created, see above). If you confirm, all symbols with the same names in the overall *default symbol library* will be replaced by those of the project specific *default symbol library*. If you don't confirm, you'll be asked if you want to copy the currently active symbol the overall *default symbol library*.

7 Fault Trees

7.1 Introduction and Overview

Fault tree analysis is the most common method for hazard analysis. The algorithms provided by Functional Safety Suite allow calculations of all values necessary for safety analysis or reliability analysis. In particular, adequate algorithms for calculation of occurrence rates related to repairable systems are implemented, therefore *fault trees* can also be used for occurrence rate (PFH, failure rate, hazard rate) calculations in accordance to [EN 61508], [EN 50126], [EN 50129], [ISO 13849], [ISO 26262-5] or similar.

Probably the most cited book related to fault tree analysis is the "Fault Tree Handbook" [NUREG], published in 1981 by the US Nuclear Regulatory Commission following the *Three Mile Island* accident in 1979. In 2002 NASA published the "Fault tree handbook with aerospace applications" [NASA]. Even though this book refers to [NUREG], its focus is different and just its existence already shows, that the spectrum of problems is too large to be explained in one book.

The most remarkable difference is, that in [NASA] the class of technical processes to be analyzed is a mission characterized by

- a defined start and end time
- no components serviceable or repairable during system lifetime
- no (safety related) undetected faults assumed at t=0
- no possibility to enter a safe state in case of a failure
- a probability p that the mission fails (equivalent to the unreliability $F(0,T_{\text{mission}})$

whereas in [NUREG] as well as in all machinery or transportation related standards such as [EN 61508], [ISO 13849], [EN 50126], [ISO 26262-5] the problems can be characterized by

- a continuous process without defined start and end time
- maintenance including inspections and tests in certain intervals
- optionally a defined safe state of the overall system ([EN 61508]: 'equipment under control', EUC) that can be taken up in case of a detected failure
- repair possible either during operation or after (save) shut-down of the equipment under control (EUC)
- either a mean probability $\overline{Q}$ that all safety systems and (active) safety barriers fail when required (equivalent to their unavailability) — in [EN 61508] called "probability of failure on demand" (PFD)
- or a mean occurrence rate $\overline{h}$ of dangerous system failures — in [EN 61508] called "probability of failure per hour" (PFH).

Both [NUREG] and [NASA] are available on the Internet for free and describe in detail the method of fault tree analysis. In addition they provide detailed information of how to construct a fault tree correctly. Therefore this documentation focuses on the specific characteristics and the usage of Functional Safety Suite. Note that [EN 61025] does not cover repairable systems

and hence is of very limited use (in particular, it doesn't cover calculation of a system failure rate $\overline{h}$).

According to [NUREG] a

> fault tree is a graphic representation of the various parallel and sequential combinations of faults, that will result in the occurrence of the predefined undesired event.

In Functional Safety Suite it is even more: From version 7.0 on, *fault trees* can be simulated by Monte-Carlo-Simulation as an alternative to the "classic" evaluation via prime implicants (minimal cut-sets). Monte-Carlo-simulation permits a much more detailed and thus more realistic modeling and evaluation. In particular, *gates* and *basic events* have a state that changes over time both randomly and according to predetermined rules:

- randomly occurring faults let *basic events* change their state,
- determined test intervals let *basic events* change their state,
- states of some *gates* may affect the detection of faults of lower level *basic events*,
- the behavior of *gates* may depend on the sequence of input failures etc.

See section 7.4 for the available gate types. Using these new gate types, *fault trees* are better be described as

> fault tree is a graphic representation of the various faults of components (or occurrence of other unwanted events), their detection, their technical and operational effects and reactions, and their parallel and sequential combinations that will result in the occurrence of the predefined undesired event.

Typically a fault tree analysis is used on a high level. A fault tree analysis is a deductive method, thus fault trees are always developed top-down (already having basic events in mind when starting to create a tree is one of the most common mistakes).

A *basic event* of a *fault tree* can describe the status of an element of the system (a situation or condition lasting for a while) or the occurrence of something in just a moment (a failure, or an action of the operator for instance).

Each *basic event* is assigned a failure or occurrence model with a specific set of parameters. Based on these parameters, the (conditional) occurrence rate h (unit 1/h), the unconditional occurrence rate for repairable elements w (unit 1/h), the (unconditional) failure density F (unit 1/h), the unavailability Q, and the unreliability F can be calculated for each *basic event*. The occurrence rates h or w and the unavailabilities Q of the *basic events* are needed to calculate both occurrence rates and unavailabilities of higher level *gates*. The mean unavailability $\overline{Q}$ of the top event is the PFD, its mean occurrence rate $\overline{h}$ is the PFH. For many systems, the system unreliability $F_{\text{sys}}(0,T_{\text{mission}})$, i. e. the probability of mission failure can directly be calculated based on the unreliabilities F of the *basic events*. If there are conditions in the *fault tree*, i. e. elements that are described by their unavailability Q instead of F, the system unreliability must be calculated based on the mean occurrence rate $\overline{h}_{\text{sys}}$ or by the time-dependent failure

density $f_{\mathrm{sys}}(t) = h_{\mathrm{sys}}(t) \cdot (1 - F_{\mathrm{sys}}(t))$.

If used for THR apportionment (often named preliminary FTA), the values for the *basic events* are defined based on what seems realistic and achievable with reasonable effort. Thus, experience is necessary to perform a preliminary FTA. The parameters assigned to each *basic event* serve as the tolerable probability of failure on demand (TPFD) or tolerable failure rate (TPFH, TFFR), that has to be achieved by the responsible system element.

Features of Functional Safety Suite related to *fault trees*:

- correct calculation of occurrence rates also for repairable systems,
- many occurrence/failure models for basic events, including links to other *fault trees*, *reliability block diagrams*, *Markov models* and *complex component*,
- gates of many types, including several types suitable for Monte-Carlo simulation only (non-boolean gates, sometimes called "dynamic gates"),
- Combination gates with an arbitrary number of different inputs,
- Determination of the minimal cut-sets or prime implicants, even for huge trees,
- modularization using special TRANSFER-IN gates,
- the β-model for common cause factors,
- conversion of architecture diagrams to *fault trees*,
- conversion of *fault trees* to (extended) *reliability block diagrams*,
- conversion of *fault trees* to (extended) *Markov models*,
- check of fault trees according to [SiRF] rules,
- steady state evaluation,
- transient (time-dependent) evaluation,
- evaluation by Monte-Carlo-simulation.

Note: The structure of a fault tree is often more important with respect to the correctness of the derived conclusions than the actual quantitative values. It is absolutely necessary that the structure of a fault tree reflects reality and that no important events are omitted because of rules or "political" reasons. All relevant conditions (e. g. responsibilities, maintenance cycles etc.) must be known in order to enable the safety engineer to develop a correct fault tree. Where the relevant conditions are not known, assumptions can be used, but these must be mentioned explicitly. In fact for most systems a small number of critical elements (*basic event*) can clearly be determined, e. g. by a *importance analysis* (see section 11.6.2). It makes sense to concentrate on elements with high impact and to validate that they are correctly modeled.

7.2 The Fault Tree Properties

Presentation related properties of the *fault tree* are edited in the *fault tree properties panel* directly, see below. Evaluation related properties are set in the *fault tree evaluation properties* dialog, see section 7.6. All properties of the *fault tree* are stored in the fault tree file (extension `.ftl`).

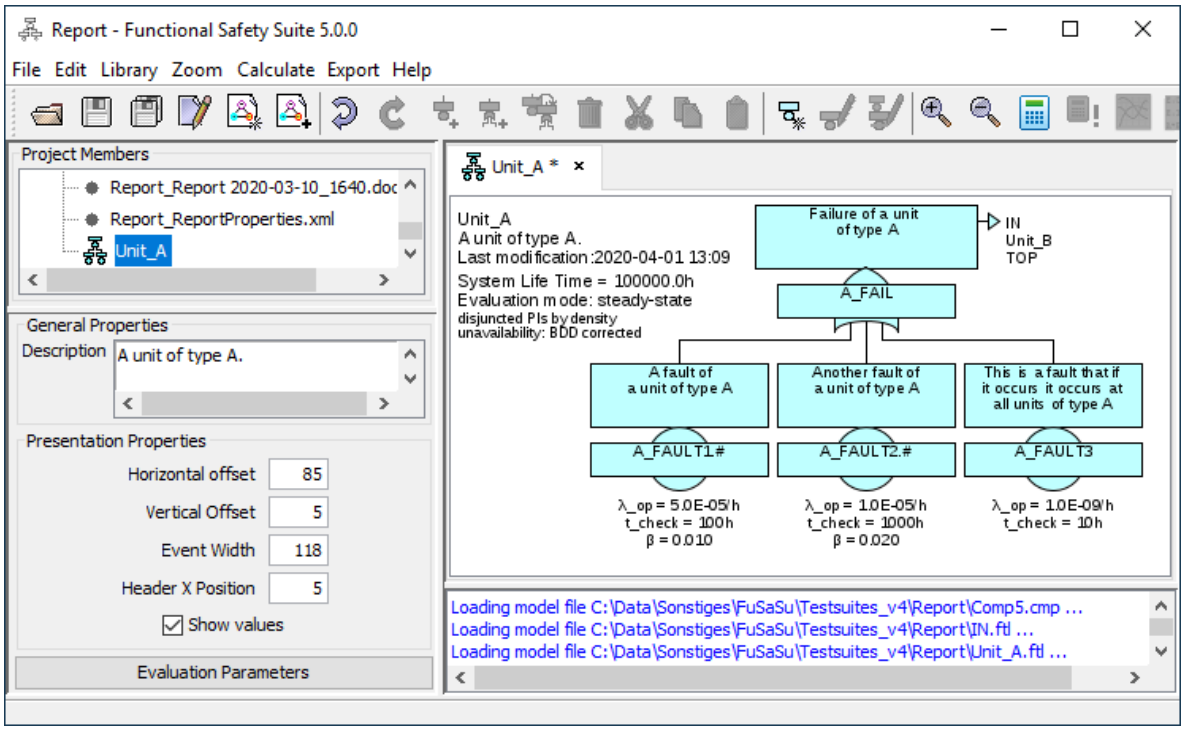


Figure 32: *The fault tree properties panel*

7.2.1 General Properties

Description:

A user defined description of the *fault tree*.

7.2.2 Presentation Properties

Note that in case the presentation related features don't fulfil your needs, you can export all graphics in SVG format for further processing by vector graphics tools.

Horizontal offset:

The margin between the window border and the left edge of the leftmost *basic event*. Standard is 5 [pixel] (multiplied by the zoom-factor). A bigger value makes sense for trees with few basic events in order to create space for the tree description (avoid overlap of description and top event).

Vertical offset:

The margin between the window border and the upper edge of the top event. Standard is 5 [pixel] (multiplied by the zoom-factor).

Event width:

Select the width of the name boxes in *fault trees*. The description boxes of *basic events* have the same width, description boxes of *gates* are about 20% wider.

Standard is 118 pixel, allowing to display both occurrence rate and unavailability in one line. If you only want to display unavailabilities or no values at all, you can enter a smaller width. If you need more space especially in description boxes, you can enter a larger width.

Header X Position:

The margin between the window border and the left side of the header. Standard is 5 [pixel] (multiplied by the zoom-factor). You can shift the header to the right by setting to a higher value.

Show values:

If values shall not be shown, you can switch them off here.

7.3 Tree Basic Events

A *basic event* of a *fault tree* consists of the reference to the *generic basic event*, an optional suffix, the 'second text line' (for qualitative *fault trees* only), and the background color.

The parameters in the 'Tree Basic Event' section belong to the specific *basic event*, which is part of a *fault tree*, and thus are stored in the fault tree file (extension `.ftl`).

The parameters in the 'Generic Basic Event' section belong to the *generic basic event* as selected by the field 'GBE name', and thus are stored in the *library*.

Remember: Changing parameters in the 'Generic Basic Event' section will change the properties of all other *basic events* referring to the same *generic basic event* too.

7.3.1 Tree Basic Event – General Properties

7.3.1.1 Package

Select whether the *generic basic event* is in the *library* of the *global package* or of the *local package*.

7.3.1.2 GBE Name

The identifier of the *generic basic event*, also serving as the name of the *basic event*. You can select a name (and by this the referred *generic basic event*) out of a list of the *generic basic events* belonging to the selected *package*.

7.3.1.3 Suffix

Two possibilities exist with respect to common causes in general (also see section 7.7.1):

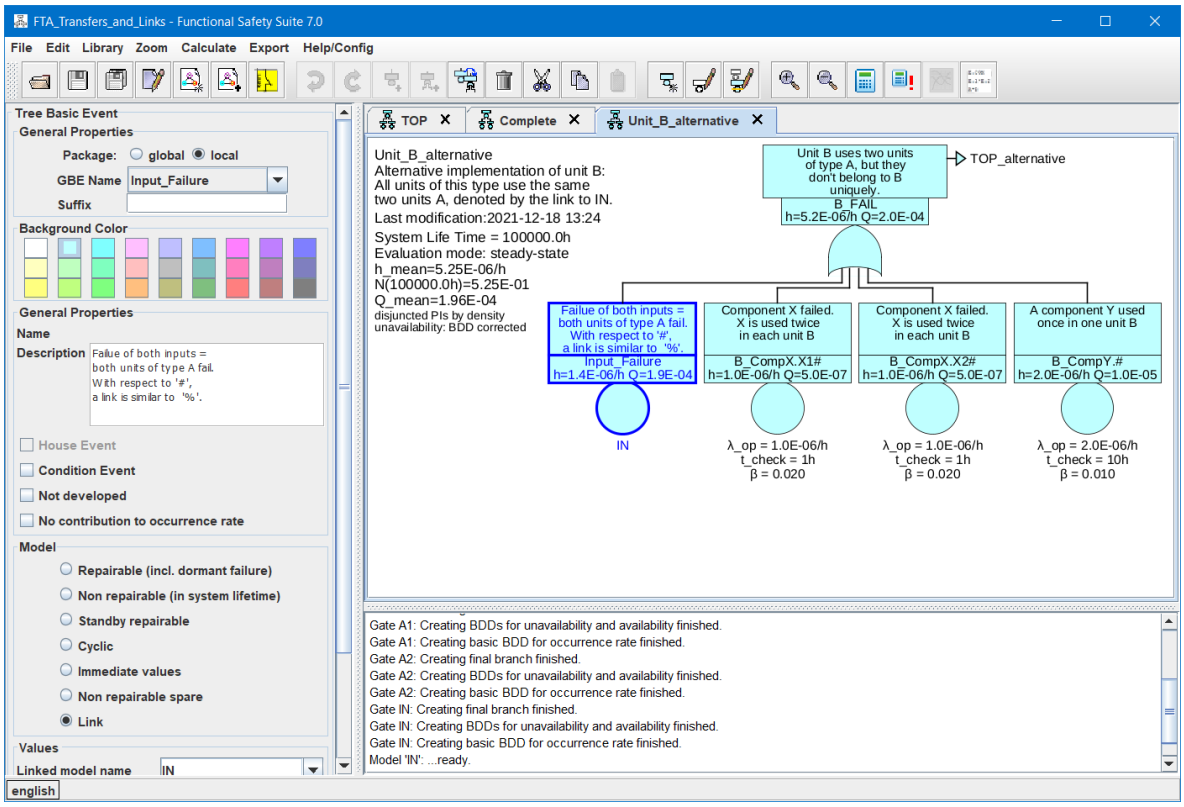


Figure 33: *The fault tree basic event properties panel*

1. The same event is needed at multiple positions in the tree(s) (e. g. the system-wide loss of the main power supply): In this case either none or all of these events are true at a given time t. In EN 61025 this is called 'repeated events' or 'replicated events', within Functional Safety Suite it is called 'identical events'. To tell the evaluation algorithm that you mean the same event everywhere, you must provide the same *generic basic event* and the same suffix to all of these events (or an empty suffix for all of them).

2. Multiple components of the same type are used in the system in a similar way (for instance in a 1oo2 or 2oo3-(sub)system). In that case these components will typically share some common cause failures. Their contribution can be set in the properties of the selected *basic event*. To tell the evaluation algorithm that it shall consider the given common cause factor between these components, you must use the same *generic basic event* (the same name) for all of them, but with different suffixes (e. g. '1', '2' and '3' or 'FRONT' and 'REAR').

Thus the suffix is used to distinguish multiple *basic events* referring to the same *generic basic event*, that are not identical, but share a common cause factor $\beta > 0$.

Since dots '.' are used automatically to delimit the name from the suffix and also to indicate multiple parts of the suffix in prime implicants (cut-sets), they should not be used in the name or suffix (even it would be technically possible).

7.3.2 Tree Basic Event – Qualitative Properties

For qualitative *fault trees* a 'second text line' is displayed instead of the numeric values in the name field of each event. It doesn't get lost when changing the project type in the *project properties dialog* to quantitative evaluation. Since this text is specific to each *basic event*, it is a property of the *basic event*, not the underlying *generic basic event*. Therefore it is stored in the fault tree file.

If you want to check the *fault tree* according to [SiRF]-rules, the 'second text line' must begin with either 'SAS' or 'SL', followed by one optional space, followed by a number 0 to 4. After that arbitrary text is allowed. Also see section 7.8 and the example provided with Functional Safety Suite.

7.3.3 Tree Basic Event – Background Color

The background color can be selected separately for each event.

7.3.4 Generic Basic Event – General Properties

7.3.4.1 Description

A user defined description of the *generic basic event* and therefore identical for all *basic events* referring to this *generic basic event*.

7.3.4.2 House event, Condition event, Not developed

See section 4.2.1.

7.3.5 Generic Basic Event – Model

The probabilistic model of the *generic basic event*. See section 4.3 for details.

7.3.6 Generic Basic Event – Values

The values needed by the model of the *generic basic event*. See section 4.3 for details.

7.4 Gates

For mathematical evaluation based on prime implicants (minimal cut-sets) the following *gate* types are supported:

- OR
- AND
- COMBINATION (M-out-of-N)
- NOT
- INHIBIT
- IF-THEN-ELSE
- REDUCED-COMBINATION
- TRANSFER-IN

For Monte-Carlo-Simulation the *gate* types indicated in table 4 are available in addition. If a *fault tree* containing those *gates* is evaluated based on prime implicants instead of Monte-Carlo-Simulation, they are replaced by either the first input or by another *gate* type as indicated in column *Replacement*. The result will typically be conservative, but in some cases it might be optimistic (e. g. if basic events with decreasing failure rates are used and in case of a VOTING-AND). Some of these "gates" cannot be used in condition or replacement branches,

Table 4: *Non-boolean gates for Monte-Carlo-Simulation*

Type	**Restrictions**	**Replacement**
DIAGNOSIS	not in condition	first input only
DIAGNOSIS-INHIBIT	not in condition	AND
DIAGNOSIS-AND		AND
VOTING-AND		COMBINATION
PRIORITY-AND		AND
REPLACEMENT	not in replacement	first input only
DIAGNOSIS-AND-REPLACEMENT	not in condition or replacement	first input only
SEQUENTIAL	not in spare or sequential	AND
SPARE	not in spare or sequential	input only
RESTORATION-ON-SAFE-FAILURE		first input only

see column *Restrictions* in table 4. Some of these "gates" influence the basic events in their branches. In that case, you might need to specify which basic events shall be affected by adding those basic events to a list, see the detailed description of each *gate* below and section 7.5.

Actually, only OR, AND, COMBINATION, NOT and IF-THEN-ELSE represent boolean combinations. The TRANSFER-IN-gate is just a structuring element not modeling any behavior of the system, see section 7.4.18 and section 7.7.2.1. All other "gates" are more semantic elements which make it possible to describe the behavior of this branch in the particular system precisely, considering operational aspects in a particular application, such as diagnosis, replacement and exact type of redundancy. These "gates" should be understood as tools to

model the behavior in case of faults in the real application accurately. They will provide less conservative results compared to using boolean gates only and prevent using boolean gates in a wrong, non-conservative manner (such as misusing an AND-gate for diagnosis). In case of generic systems (element/system out of context), the assumptions leading to the usage of a specific gate type should be documented and exported as safety related application condition (SRAC).

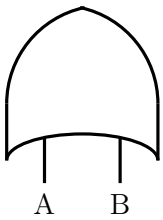
Events connected to a *gate* are called 'inputs'. The *gate* an event serves as input is called 'parent'. The topmost event of a *fault tree* is called *top event*, in case of the highest level fault tree also 'top hazard'.

If only one input is connected to a *gate* of type AND, *DIAGNOSIS-AND*, *VOTING-AND* or OR, the symbol is not displayed, since the gate has no effect.

Each input can be a *basic event* or another *gate*.

Common cause contributions are considered when calculating a *gate*, see section 7.7.1.

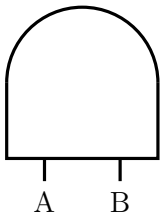
7.4.1 OR



Characteristics:

- An OR-gate fails whenever one input fails (in Monte-Carlo-Simulation: only if no other input is failed already),
- is unavailable whenever any input is failed.

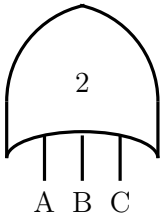
7.4.2 AND



Characteristics:

- An AND-gate fails when all inputs are failed,
- is unavailable while all inputs are unavailable (i. e. until at least one is restored).

7.4.3 COMBINATION (M-out-of-N)



Characteristics:

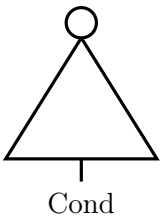
- A COMBINATION-gate fails if at least m inputs fail,
- is unavailable as long as at least m inputs are unavailable.

In fact the COMBINATION-gate is just an abbreviation of OR-ed AND-gates according to m and the number of inputs. Therefore calculation is done as with standard OR and AND-gates. You can check that by calculating and exporting the prime implicants or by exporting

the resulting fault tree by **Export – Final Fault Tree**. Identical events or common cause factors between *basic events* contained in the branches below the COMBINATION gate will be considered as if it would be one single *fault tree*.

7.4.4 NOT

Characteristics:

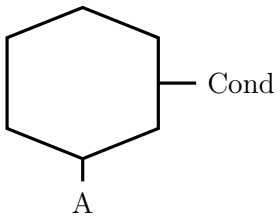


- A NOT-gate has one input,
- is unavailable while the input is ok (false),
- in Monte-Carlo-Simulation: a failure is counted when the input becomes false (ok), i.e. the occurrence rate is the same as the input occurrence rate, but the failure is set when the input is restored (transition from failed to ok).

Since the complement is defined for probabilities only, but not for rates or densities, a NOT-gate cannot deliver an occurrence rate to a higher level gate. Therefore it only makes sense in the condition branch of e. g. INHIBIT-gates or the diagnosis branch of e. g. DIAGNOSIS-gates, or if the unreliability *F(t)* shall be calculated.

7.4.5 INHIBIT

Characteristics:



- An INHIBIT-gate has one input A and one condition input Cond,
- fails if the condition is true (failed) when the input A fails,
- is unavailable if it has failed until the input A is restored, (even if the condition Cond is not fulfilled any more already before).
- if the condition changes its state while the input A it still failed, the output does not change any more (i. e. if condition becomes true after the input A failed), the output will stay false (OK), and if the condition is not fulfilled any more (false, OK) while the input A is still failed, the output will stay failed until the input is restored.

Thus the INHIBIT-gate is similar to an AND-gate, except that the occurrence rate of the second input is ignored. Since the input A often describes a standard situation but not a failure, it is often a *basic event* marked as *House Event*.

Since the condition is quantified by a probability only (no occurrence rate), the *basic events* of the branch connected to the condition input should be marked with the modifier *Condition Event*, see section 4.2.1.1.

There are two special cases related to the *condition input* of an INHIBIT-gate:

1. If a *generic basic event* of type *immediate* is connected to the *condition input* and the parameter *probability* is set to **zero**, the branch topped by the INHIBIT-gate is completely ignored.
2. If a *generic basic event* of type *immediate* is connected to the *condition input* and the parameter *probability* is set to **one**, the *condition input* is ignored.

These rules might be useful in order to adapt the structure of a (generic) *fault tree* to different specific applications, i. e. to simplify the re-use of a *fault tree* for different projects. You can check the effect if you export the final fault tree by **Export − Final Fault Tree**.

When converting a fault tree to a Markov model, an INHIBIT-Gate will lead to instantaneous transitions.

An INHIBIT-gate is not allowed in the condition branch of a higher level gate.

Use Case

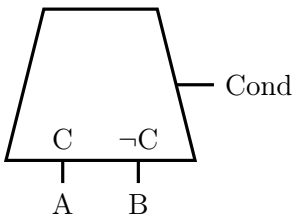
An INHIBIT-gate is appropriate

- when the failure of the input A only matters if a (typically external) condition is (fulfilled)
- to reduce the effective failure rate and the unavailability by the probability of the condition being true (failed)

7.4.6 IF-THEN-ELSE (MUX)

Characteristics:

- An IF-THEN-ELSE-gate has two inputs A and B and one condition input Cond,
- if the condition is true (failed) input A is considered for occurrence rate and unavailability,
- if the condition is false (OK) input B is considered for occurrence rate and unavailability,
- the condition is only considered at the time the input fails (i. e. a later change of the condition has no effect)
- only the input selected by the condition is considered, i. e. a change of the other input has no effect (even it is the one that has led to the failure of this gate before).



The gate's occurrence rate and unavailability are input A times the probability Q of Cond, plus those of input B times the complementary probability 1–Q of Cond.

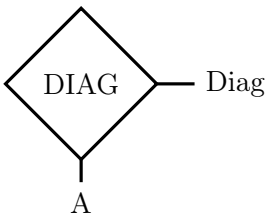
Note that due to the intended purpose of the gate, the two inputs A and B are mutually exclusive per definition, i. e. the combination of the two inputs is not included in the list of prime implicants.

Note that the use of this gate will form a non-coherent fault tree, including all the incon-

veniences non-coherent fault trees bring along (such as non-unique prime implicants or non-conservativeness even all failure rates are conservative).

7.4.7 DIAGNOSIS

Characteristics:



- A DIAGNOSIS-gate has one input A and one diagnosis input Diag,
- fails when the input A fails,
- is unavailable as long as the input A is failed (detected or undetected),
- if the input A fails while the diagnosis input is OK (false), some *basic events* will immediately set to repair state (or OK if t_{rep}=0), see below.

The failures of the following *basic events* will detected and the *basic event* sent to repair state (or to OK state directly if t_{rep}=0):

- if the input A is a *basic event* of type *repairable* or *non-repairable*, it will go to repair state, i. e. after t_{rep} it will be in OK state again,
- if the input A is a *gate* the user has to select the *generic basic events* whose failure will be detected out of a list containing all *generic basic events* of type *repairable* or *non-repairable* in branch A, see section 7.5.
- if the diagnosis is failed (diagnosis input true), the gate has no effect, i. e. the failures below input A will remain undetected until end of lifetime (for *non-repairable*) or until detected by the next periodic test (for *repairable*).

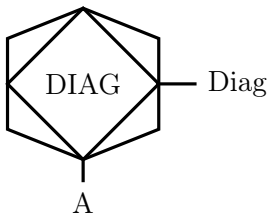
Use Case

- independent and immediate diagnosis when input A fails
- reduce unavailability due to sleeping faults
- diagnostic coverage may be less than 100%, i. e. not all *basic events* below input A can be detected

Note: This gate has no effect on failure frequency, i. e. the gate will fail with the same frequency as the input A. Only the unavailability of input A will be reduced due to the detection. If the failure of input A is not dangerous at all if if is detected, the DIAGNOSIS-INHIBIT-gate might be the model of choice, see next section.

7.4.8 DIAGNOSIS INHIBIT

Characteristics:



- A DIAGNOSIS-INHIBIT-gate has one input A and one diagnosis input Diag,
- fails if the diagnosis is failed when the input A fails (as for INHIBIT gate),
- is unavailable if it has failed until the input A is restored (as for INHIBIT gate),
- if the input A fails while the diagnosis input is false (OK), some *basic events* will immediately set to repair state (as for DIAGNOSIS gate, see there).

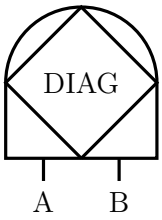
Use Case

- independent immediate (=within PFTT) diagnosis when input A fails
- reduce unavailability due to sleeping faults
- only if the failure of the input A is safe if diagnosis is OK, i. e. system is forced to a safe state if detection is OK

Note: All failures of branch A must be detectable by the diagnosis if the diagnosis part is OK (false)! Failures not detectable by the diagnosis must be modeled in a parallel branch, connected by an OR-gate above this *gate*.

7.4.9 DIAGNOSIS-AND (AND with cross-wise diagnosis)

Characteristics:



- A DIAGNOSIS-AND gate has an arbitrary number of inputs A, B etc.,
- fails if all inputs fail at the same time (either by the same *basic event* or by different *basic events* sharing a common cause) or when the last remaining input fails while the other inputs are still unavailable (i. e. the failure cannot be detected by another channel or still being repaired),
- is unavailable while all inputs are failed (i. e. until at least one is repaired),
- some *basic events* will immediately set to repair state if at least one input is OK (see below).

If at least one input is OK (false), any failures of the following *basic events* below failed inputs will detected and the *basic event* sent to repair state (or to OK state directly if t_{rep}=0):

- if a failed input is a *basic event* of type *repairable* or *non-repairable*, it will go to repair state, i. e. after t_{rep} it will be in OK state again,

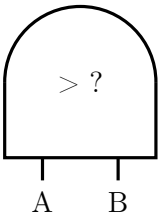
- if a failed input is a *gate* the user has to select the *generic basic events* whose failure will be detected out of a list containing all *generic basic events* of type *repairable* or *non-repairable* in all branches, see section 7.5. Note: even no specific *basic events* but only *generic basic events* are selected for detection in the list, only failed *basic events* below a failed input of the DIAGNOSIS-AND gate will be detected, not below any (non-failed) input of the DIAGNOSIS-AND gate or somewhere else in the overall *fault tree*.

Use Case

- multiple channels with cross-wise immediate (=within PFTT) diagnosis (with a $DC \leq 100\%$)
- reduce unavailability due to sleeping faults

7.4.10 VOTING AND

Characteristics:



- A VOTING-AND gate has an arbitrary number of inputs,
- fails if at least 50% of the available inputs fail at the same time,
- is unavailable after failure until more than 50% inputs are available again,
- some *basic events* might immediately be set to repair state if at least one input is OK (as for DIAGNOSIS-AND gate, see there).

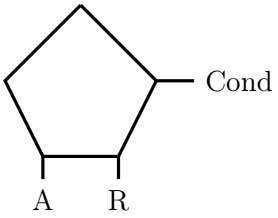
Use Case

This gate can be used if

- the majority of channels decide what is correct in case of discrepancy between channels,
- there is no safe state of the system (fail active operational design),
- there is no specific diagnosis in each channel, i. e. "diagnosis" is only comparison of the results and assumption that majority of outputs will be correct,
- optional: each channel can raise a failure information in some way in order to start the restoration.

7.4.11 REPLACEMENT

Characteristics:



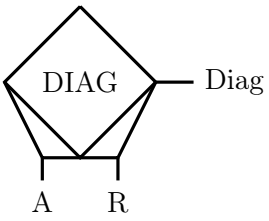
- A REPLACEMENT-gate has one input A and one replacement input R,
- the *basic events* of type *standby repairable* in the replacement input R are activated when input A fails,
- the *basic events* of type *standby repairable* in the replacement input R are deactivated when input A is OK again,
- the gate fails (gets true), if the replacement R fails at activation or while the input A is still failed,
- the gate is unavailable after failure until input A changes to OK (false).

Use Case

- for component(s) in cold standby, that can fail at activation or during operation
- no diagnosis/complex activation to be modeled (e. g. detection of failure of A is guaranteed)

7.4.12 DIAGNOSIS REPLACEMENT

Characteristics:



- A DIAGNOSIS-REPLACEMENT-gate is a combination of DIAGNOSIS / DIAGNOSIS-INHIBIT and REPLACEMENT,
- has one input A, one replacement input R and one diagnosis input Diag,
- fails if the diagnosis or the replacement R is failed when input A fails,
- fails if the replacement R fails at activation or while input A is still unavailable,
- is unavailable if it has failed until input A is restored,
- the *basic events* of type *standby repairable* in the replacement input R are activated when input A fails,
- the *basic events* of type *standby repairable* in the replacement input R are deactivated when input A is OK again,
- if input A fails while diagnosis is OK, some *basic events* of input A are immediately set to repair state (or OK if t_{rep}=0).

Use Case

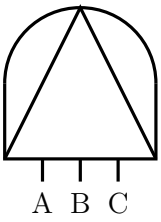
This *gate* describes a system with

- component(s) in cold standby, that can fail at activation or during operation,
- where the component(s) in cold standby are only activated if the diagnosis is working,
- with complex diagnosis containing many components that can fail as well or depend on further conditions.

7.4.13 PRIORITY AND

Characteristics:

- A PRIORITY-AND-gate fails if the first input A failed before the second input B, the second before the third input C etc. until all inputs have failed (and not been restored before failure of the last input),
- is unavailable as long as no input is restored,
- if one input is OK (false) – no matter whether it has never failed yet or has been restored –, some *basic events* of all failed inputs on the right of this input are immediately set to repair state (or OK if t_{rep}=0), see below.



Example: Assume input A failed first, then input B. After that input A has been restored (due to some periodic test and restoration or exchange). Now that input A is OK while input B is failed, the failure of input B will be detected as for a DIAGNOSIS-AND gate:

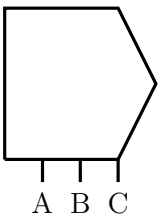
- if the input B is a *basic event* of type *repairable* or *non-repairable*, it will go to repair state, i. e. after t_{rep} it will be in OK state again,
- if the input B is a *gate* the user has to select the *generic basic events* whose failure will be detected out of a list containing all *generic basic events* of type *repairable* or *non-repairable* in any branch (A, B and C), see section 7.5. Note: even no specific *basic events* but only *generic basic events* are selected for detection in the list, only failed *basic events* below a failed input of the PRIORITY-AND gate will be detected, not below any (non-failed) inputs of the PRIORITY-AND gate or somewhere else in the overall *fault tree*.

Use Case

- If the top failure only occurs if the inputs fail in a specific sequence.

7.4.14 SEQUENTIAL

Characteristics:



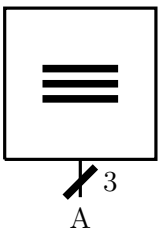
- A SEQUENTIAL-gate has one active input at any time,
- in the beginning the first input A is considered only, it is the active input,
- when the active input fails, the next input is activated and considered from now on,
- when the last (the rightmost) input fails, the gate fails,
- basic events of type *non-repairable* will be activated at t=0,
- basic events of type *non-repairable spare* and *standby repairable* will be activated when the particular input becomes the active one (i. e. cannot fail before the input is the active one).

Use Case

- If there are alternative (typically degraded) operating modes in case the main component (main mode) fails.
- Similar to SPARE, but with different spare components.
- Typically the unreliability F is to be quantified for these kind of systems/applications.

7.4.15 SPARE

Characteristics:



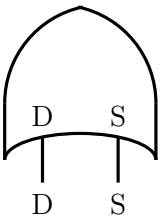
- A SPARE-gate has one input A, which indicates m elements of the same type,
- fails if its input failed more than m-spare times,
  - Note: m=1 means, there is one spare, i. e. the gate fails when the input fails for the 2nd time,
- remains failed after it fails until end of lifetime,
- basic events of type *non-repairable* will be activated at t=0,
- basic events of type *non-repairable spare* and *standby repairable* will be (re-)activated after each failure.

Use Case

- If there is only a limited number of spare parts available during the mission.
- Similar to SEQUENTIAL, but with identical spare components.
- Typically the unreliability F is to be quantified for these kind of systems/applications.

7.4.16 RESTORATION ON SAFE FAILURE

Characteristics:



- A RESTORATION-ON-SAFE-FAILURE-gate has two inputs: input D contains the dangerous failures, input S contains the safe failures S,
- fails when input D fails,
- is unavailable as long as the input D is failed,
- certain *basic events* are restored when the safe input S fails.

When the safe input S fails, any failures of the following *basic events* below failed inputs will detected and the *basic event* sent to OK state (t_{rep}=0 assumed):

- if the input (D or S) is a *basic event* of type *repairable* or *non-repairable*, it will go to OK state,
- if the input (D or S) is a *gate* the user has to select the *generic basic events* whose failure will be restored out of a list containing all *generic basic events* of type *repairable* or *non-repairable* in both input branches, see section 7.5.

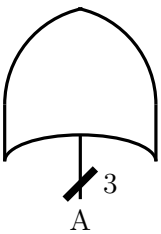
Use Case

Most components or modules include many failure modes. Many of them directly lead to a safe state of the EUC, and need to be repaired before the EUC can be operated at all, therefore. Those failure modes (and only those!) are called 'safe' failures. The repair is often a replacement of a bigger module or assembly, thus undetected dangerous ('dormant') faults will be eliminated as well, even they haven't been detected at all. Other dangerous ('dormant') faults might be detected (and corrected) in a mandatory commissioning test after repair. In effect, safe failures often lead to (known or unknown) elimination of dangerous faults in particular for components, that typically failing safely before or more often than failing dangerously. This effect can be modeled with a RESTORATION-ON-SAFE-FAILURE gate.

Note: In contrary to dangerous *basic events*, the rates of safe failures should be intentionally selected too low in order to get a conservative overall model.

7.4.17 REDUCED COMBINATION

Characteristics:



- A REDUCED-COMBINATION-gate has one graphically displayed input, which symbolizes n elements of the same type,
- no common cause failure considered between the n elements,
- number of elements n is limited to 170.

The output of a REDUCED-COMBINATION gate is true (faulty), if at least m out of the n inputs are true (faulty).

In contrary to all other *gates*, REDUCED-COMBINATION gates numerically evaluate the input event (they calculate the unavailability Q_{in} and occurrence rate h_{in} or unreliability F_{in} of the connected input before evaluation of the *fault tree* containing the REDUCED-COMBINATION gate). Unavailability Q_{g} and occurrence rate h_{g} or unreliability F_{g} of the gate are then calculated as function of n, m, Q_{in} and occurrence rate h_{in} or unreliability F_{in} and stored in a new temporary *generic basic event*, referred by a new temporary (internal) *basic event* of type *Link*. The formulas used to calculate higher gates only consider this *basic event* and thus do not get information of the structure below the REDUCED COMBINATION gate.

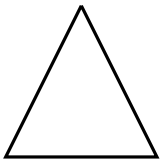
The suffix of the temporary (internal) *basic event* can be set to '#' in order to create multiple instances of the element described by the REDUCED COMBINATION gate in higher level *fault trees*. This is achieved by adding an '#' at the end of the name of the REDUCED COMBINATION gate, also see section 7.7.1.

7.4.17.1 Use Case

- When there is a large number of similar items, that fail independent of each other.

7.4.18 TRANSFER-IN

A TRANSFER-IN gate is a reference to another event defined by the name of the referred *fault tree* and the name of an event in the referred *fault tree*. The referred *fault tree* must either be a member of the same *package* or of the *global package*. The referred event can be any *basic event* or *gate* in the tree, not only the *top event*. The referred *fault tree* cannot be the tree the TRANSFER-IN gate belongs to. Regarding calculations, *sub-trees* referred by TRANSFER-IN gates are treated as if they would be stated directly in the higher level tree. Therefore you can split trees wherever you want.



Fault Tree Reference
Gate Reference
Transfer Extension

Please note the possibilities for handling of *basic event* names and common cause factors as described in section 7.7.1.

Note: Circular references are forbidden for obvious reasons. They are detected in the evaluation, the evaluation will be aborted and the error indicated in the status bar.

You can open the referred *fault tree* by double-clicking the gate.

7.5 The Gate Event Properties Panel

All properties of the *gate* are stored in the fault tree file (extension `.ftl`).

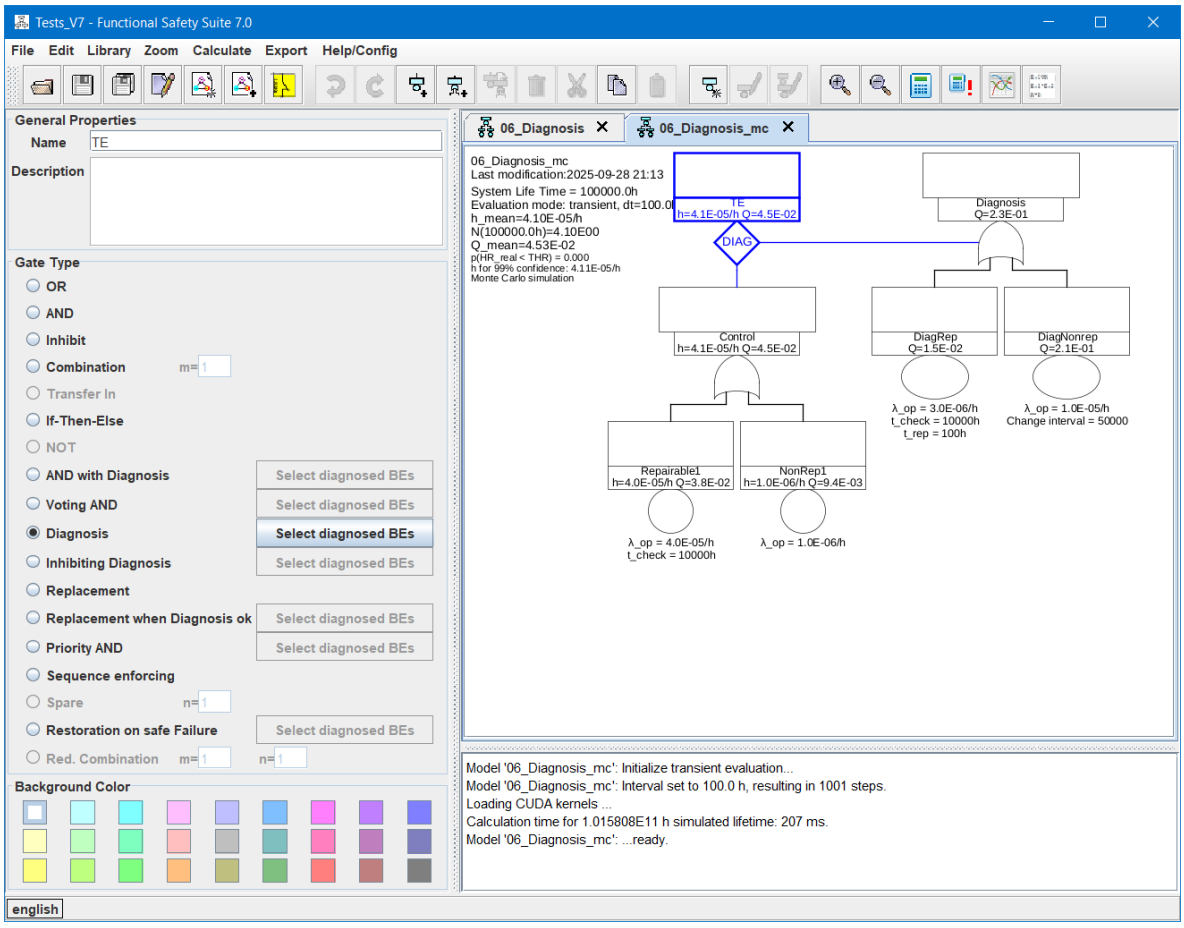


Figure 34: *The gate event properties panel*

7.5.1 General Properties

Name:

A user defined identifier of the *gate*. Every gate should have another identifier although this is not required by Functional Safety Suite.

Description:

A user defined description of the *gate*. The description of TRANSFER-IN gates is copied from the referred event, whenever you change the fields for the referred tree name or the referred event name.

7.5.2 Gate Type

For a description of each particular type see section 7.4 above. Depending on the type of the *gate* you might have to define which failure modes (i. e. which *generic basic events*) can be detected or restored. If you click the **Select diagnosed BEs** button, a dialogue will open

showing all *generic basic events* of types *repairable* and *non-repairable* below the relevant inputs of the particular *gate* and a checkbox for each of them.

7.5.3 Qualitative Properties

For qualitative *fault trees* a 'second text line' is displayed instead of the numeric values in the name field. It is stored in parallel to the numeric values and therefore doesn't get lost when changing the evaluation mode in the *project properties dialog* (tab *Fault Trees & RBDs*) and vice versa.

If you want to check the *fault tree* according to [SiRF]-rules, the 'second text line' must begin with either 'SAS' or 'SL', followed by one optional space, followed by a number 0 to 4. After that arbitrary text is allowed. Also see the example in the `doc`-directory.

7.5.4 Background Color

The background color can be selected separately for each *gate*.

7.6 Evaluation of Fault Trees

In case of quantitative evaluation the value of interest for each safety function is either

1. the mean unavailability on demand $\overline{Q}$ (PFD),
2. the mean occurrence rate $\overline{h}$ (PFH),
3. or the probability of failure *F(T)* after system lifetime (or mission time) T.

A *fault tree* can be evaluated in a qualitative way as well, see section 7.6.3.1 and 7.8.

The value of interest and several parameters related to quantitative evaluation of *fault trees* are set in the *fault tree evaluation properties* dialog, see section 7.6.3 below.

To evaluate a *fault tree*, select **Calculate – Calculate Model Values**. First the *final tree* is determined, this is the *fault tree* in which all TRANSFER-IN gates have been replaced by the adequate branch and all REDUCED-COMBINATION gates have been replaced by a link to another model. Also INHIBIT gates might have been eliminated, see section 7.4.5. Circular references are detected and indicated in the message window before the evaluation starts. In case of conventional evaluation (no Monte-Carlo-simulation), some gates will be replaced in the *final tree* according to table 4 above. The *final tree* can be exported to a new *fault tree* by **Export – Final Fault Tree**, e. g. in order to check if all modules referred by TRANSFER-IN gates have been considered as intended, see section 11.7.9.

Results of the evaluation are displayed in the header of the *fault tree*. In case of quantitative evaluation, results are displayed in each event's symbol as well, see section 7.6.3.1.

The quantitative evaluation is performed either based on prime implicants (identical to minimal cut-sets in case of coherent fault trees), or it is performed by Monte-Carlo simulation. Since the algorithms are completely different, they are described in different sections 7.6.1 and 7.6.2.

If a value of a *generic basic event*, the suffix of a *basic event* or the structure or evaluation parameters of the *fault tree* is changed, all values that might be affected by this change are automatically marked invalid and not displayed anymore, so that no inconsistent values are displayed.

7.6.1 Evaluation via Prime Implicants or BDDs

For evaluation via prime implicants and/or BDDs some gates will be replaced in the *final tree* according to table 4 above. After creating the *final tree*, prime implicants and/or binary decision diagrams (BDDs) describing the unavailability Q, the unreliability F and/or the occurrence rates h or densities w are created. Then all lower level models connected by *links* and all *generic basic events* are evaluated, so that finally the prime implicants and/or BDDs can be quantified. Depending on your choice in the *Gate Calculation Mode* panel, either only the *top event* is calculated or all *gates*.

7.6.1.1 Quantitative Steady-state Evaluation

A steady-state analysis is appropriate for all systems that are supposed to operate for many years, with certain test intervals and optionally some down-times for maintenance and repairs. Several parts of the system might be replaced or repaired during the system's lifetime. In case that all failures are detected in adequate time (either by continuous diagnosis, by periodic tests or by malfunction of the system), both the failure rate h (PFH) and the unavailability Q (PFD) of the system don't depend on its actual age, but will reach some pseudo-stationary state where both values will oscillate around a mean value. The frequency of this oscillation is equal to the longest detection interval or a multiple of it. This is even correct in case the failure rates of some particular components depend on their specific age, if the lifetimes of these components are shorter than the system life time. The value of interest for each safety function performed by such a system is either the mean unavailability on demand $\overline{Q}$ (PFD), or the mean occurrence rate $\overline{h}$ (PFH). The related standard is mainly [EN 61508] and the derived standards. Examples for those systems are machines, cars, trains, air-crafts, chemical plants, power plants, etc. and their control systems.

A steady-state analysis is very fast, because all values have to be calculated only once (compared to transient analysis, where all values have to be calculated many times, see below).

Unfortunately, there is one major issue related to steady-state calculation of unavailabilities: The mean value of the product of two (or more) time-variant values is in general not equal to the product of the mean values but greater:

$$\overline{Q_1(t) \cdot Q_2(t)} > \overline{Q_1(t)} \cdot \overline{Q_2(t)}$$

The unavailability function *Q(t)* of a repairable component is a periodic function: It becomes *Q(t)=0* after each (complete) test at time $t_n = n \cdot t_{\text{check}} + t_0$ and increases until the next test at time $t_{n+1} = (n+1) \cdot t_{\text{check}} + t_0$.

The following examples show the unavailability as function of time for a system of two repairable (and therefore periodically tested) components. The first example considers non-redundant components (connected by an OR gate), the second considers redundant components (connected by an AND gate).

Example 1:

A safety system consists of two components (both needed, not redundant) with two failure events E1 and E2.

E1 has a failure rate of $\lambda_1 = 10^{-4}$/h and is tested every 150 h.

E2 has a failure rate of $\lambda_2 = 10^{-3}$/h and is tested every 30 h.

Figure 35 shows the unavailability function Q(t) if at t_0=0 both components are tested. Due to the given test intervals, also at $t = n \cdot 150$ h both components will be tested in parallel. The dotted lines are the single event unavailabilities, the solid line shows the overall unavailability, which is approximately the sum of both single event unavailabilities.

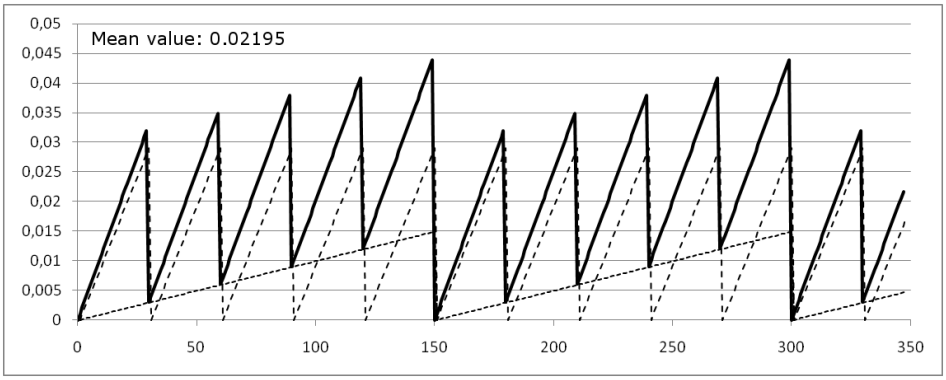


Figure 35: *The unavailability functions of two components, first test at the same time*

Figure 36 shows the unavailability function *Q(t)* if the test for E1 is executed at t_1=t_0, whereas the test for E2 is executed at t_2=t_0+15 h. Since there is no complete test, the overall unavailability never evaluates to 0 (at least not for t>0).

Note that the mean value is the same, independent of the relation of the test times.

The lifetime is usually much longer than a period of *Q(t)*, because the component is tested or maintained several times during the lifetime.

Another special case is that the component/function is used only for one mission, e. g. it is created/tested before or at the start of the mission and becomes invalid after the end of the mission. In that case the interval is identical to the mission time.

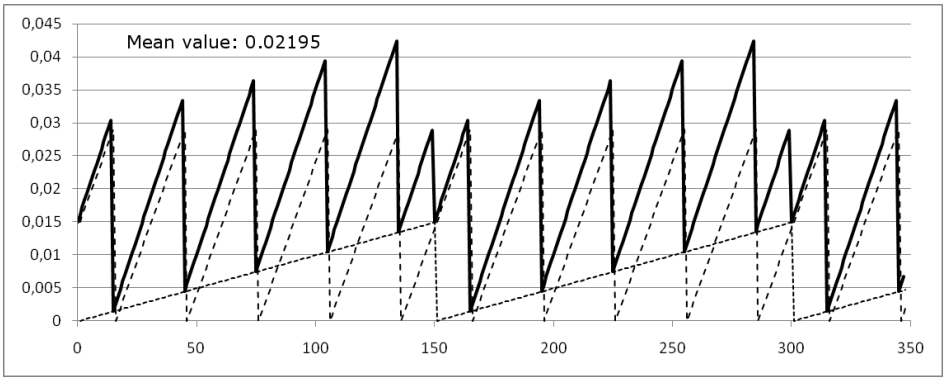


Figure 36: *The unavailability functions of two components, staggered tests*

Example 2:

A safety system consists of two redundant subsystems S1 and S2. S1 has a failure rate of $\lambda_1 = 10^{-4}$/h and is tested every 150 h. S2 has a failure rate of $\lambda_2 = 10^{-3}$/h and is tested every 50 h. With these values the mean unavailability of S1 is $\overline{Q_1} = 7.45 \cdot 10^{-3}$, of S2 it is $\overline{Q_2} = 2.45 \cdot 10^{-2}$. Multiplication of both values gives $1.83 \cdot 10^{-4}$.

Figure 37 shows the unavailability function *Q(t)* if at t=0 h both subsystems are tested. The

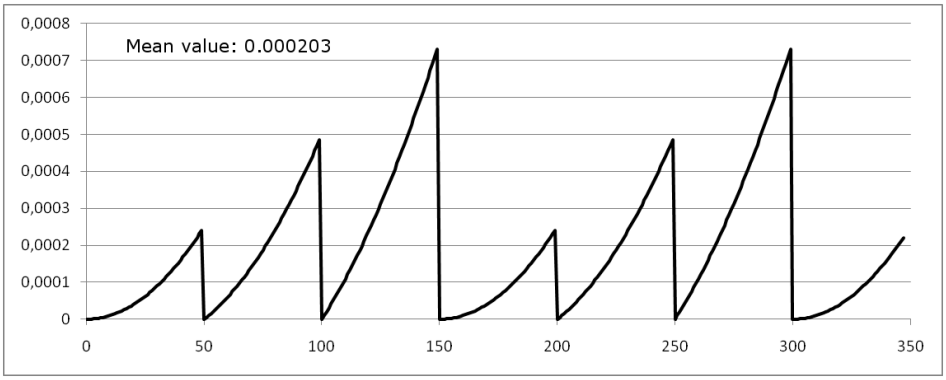


Figure 37: *The unavailability functions of redundant systems, parallel tests*

mean unavailability is $\overline{Q} = 2.03 \cdot 10^{-4}$ but not $1.83 \cdot 10^{-4}$! Simple multiplication of the values is obviously not correct and not even conservative.

Figure 38 shows the unavailability function *Q(t)* if the test for S1 is executed at t_1=0 h, whereas the test for S2 is executed at t_2=15 h. With this shift the mean unavailability is $\overline{Q} = 1.77 \cdot 10^{-4}$, with a shift of t_2=25 h it would be $\overline{Q} = 1.72 \cdot 10^{-4}$.

As shown in example 2, when performing the conjunction of events (using an AND-gate) the mean overall unavailability is not given by the product of the mean unavailabilities of each event. For synchronized tests without shift, the mean unreliability of the system is always higher than the product of the mean unavailabilities of its events. Also refer to [EN 61508-6], section B.2.2.

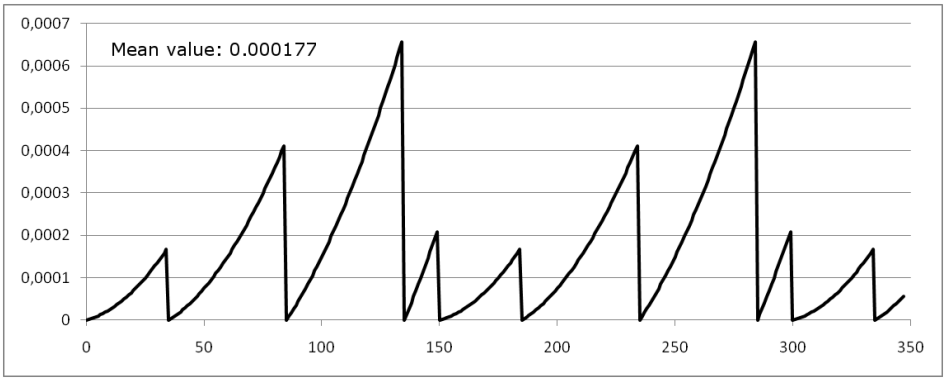


Figure 38: *The unavailability functions of redundant systems, staggered tests*

Thus, using mean values of the components unavailabilities is too optimistic in general. One option is to use maximum values of the components (*basic events*) unavailabilities, i. e. the unavailability just before the next test, but this is quite pessimistic. Functional Safety Suite provides three options on how to deal with this issue in steady-state evaluations, see section 7.6.3.4. The only way how to calculate the exact values is using a transient evaluation, but this needs much more computing time, unfortunately.

7.6.1.2 Quantitative transient Evaluation

A transient (or time-variant) analysis typically produces more precise results for

- *fault trees* or *Markov models* containing *basic events* with time-variant parameters, e. g. *basic events* of type *non-repairable*,
- *fault trees* or *Markov models* including *basic events* of type *cyclic*.
- *fault trees* resulting in minimal cut-sets that are longer than 2 events.

Also if the 'mission failure probability', namely the system's unreliability $F(0,T_{\text{mission}})$ is the value of interest, a transient evaluation usually makes sense, even though many systems of this kind can also be modeled by steady-state *fault trees* or *Markov models*, using *generic basic events* of type *non-repairable*, see section 4.3.2.

Time interval: The step size for transient evaluation in hours. A smaller step size means more steps for the given system lifetime and thus takes more time in calculation. This might be an issue in case of large systems. The step size must be less than a 10th of the smallest periodic (cyclic) event you want to evaluate for discrete times. Time constants less than 10 times the step size are handled as rates. However step size should be even smaller in order to reduce the computational errors.

7.6.2 Evaluation via Monte Carlo Simulation

In contrary to all other algorithms, the Monte-Carlo simulation is not implemented in Java code but in C++, CUDA® and OpenCL™ and compiled to separate libraries in order to

speed up simulation. It requires either

1. a CUDA capable Nvidia® GPU or
2. a GPU or CPU with a driver supporting OpenCL™ 2.1 or higher and SPIR-V™ 1.4 or
3. MS Windows.

If you have a look to the message output window just after starting Functional Safety Suite, you'll see information whether a CUDA device and/or OpenCL device with SPIR-V was found.

Minimal cut-sets (or prime implicants in case of incoherent fault trees) only contain *basic events* but no *gates*, thus they can only describe boolean combinations of *basic events*. They cannot describe sequences of effects that depend on the order of events, nor can they distinguish effects to basic events that depend on the state of other events. Those effects can nevertheless be described using the fault tree notation by introducing non-boolean "gates", see section 7.4. Quantification of those (extended) fault trees however requires a Monte-Carlo simulation. Since a Monte-Carlo simulation is no mathematical calculation but a simulation of a random experiment, the more random experiments are simulated the more precise and reliable the result will be. One experiment is the simulation of one system over its lifetime. The number of experiments to be performed depends on which tolerable unavailability Q_{tol}, tolerable failure frequency h_{tol} or tolerable unreliability F_{tol} you want to prove, and how close to this tolerable value the actual value of your system is. Thus, it is not possible to define the necessary number of experiments prior to doing the simulation, but only to estimate the statistic confidence of the result.

See section 7.6.4 for explanation of the parameters.

In addition to the simulation result, two values are stated in the model header:

1. the confidence that the simulation result is in fact less than the tolerable value entered in the parameters (see paragraph 7.6.4)
2. the value that can be guaranteed with a confidence of 99%

For Monte Carlo simulation, there is basically no difference between steady-state and transient evaluation — the simulation is identical. The difference is, that in transient evaluation, it will be recorded when a failure or restoration occurs. This will slightly increase simulation time and is only beneficial if you want to analyze the results in a chart, see section 11.6.5. Note that you'll need a huge number of simulations in order to get a somewhat smooth chart, many more than necessary to get the overall result.

7.6.3 Evaluation Parameters

Note: If the *fault tree* is referred in another *fault tree* by a TRANSFER-IN gate, these parameters are irrelevant, since this *fault tree* will just be a branch of the *final tree* of the higher *fault tree* during evaluation.

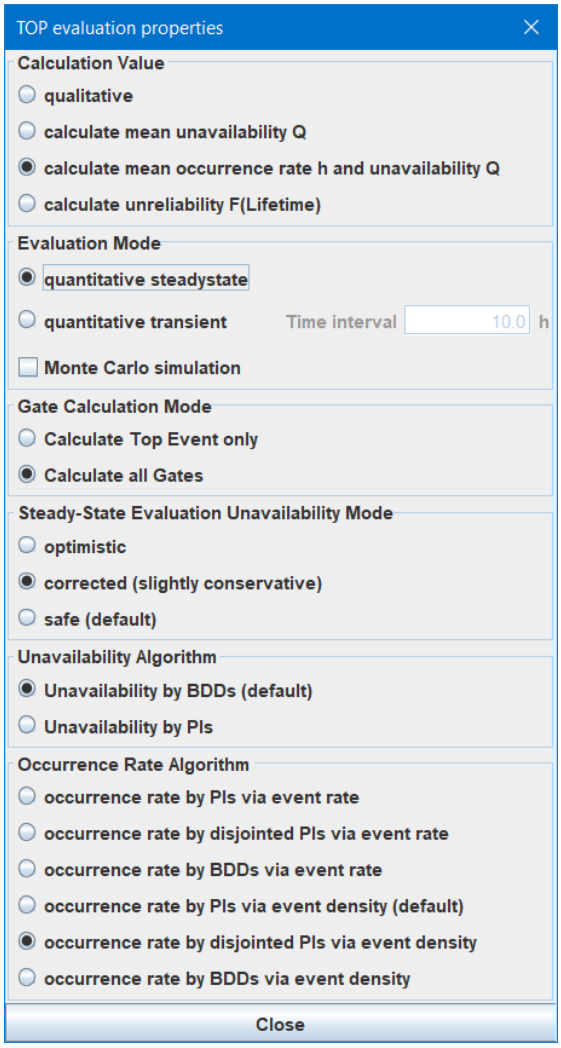


Figure 39: *Parameters for evaluation of occurrence rate and unavailability*

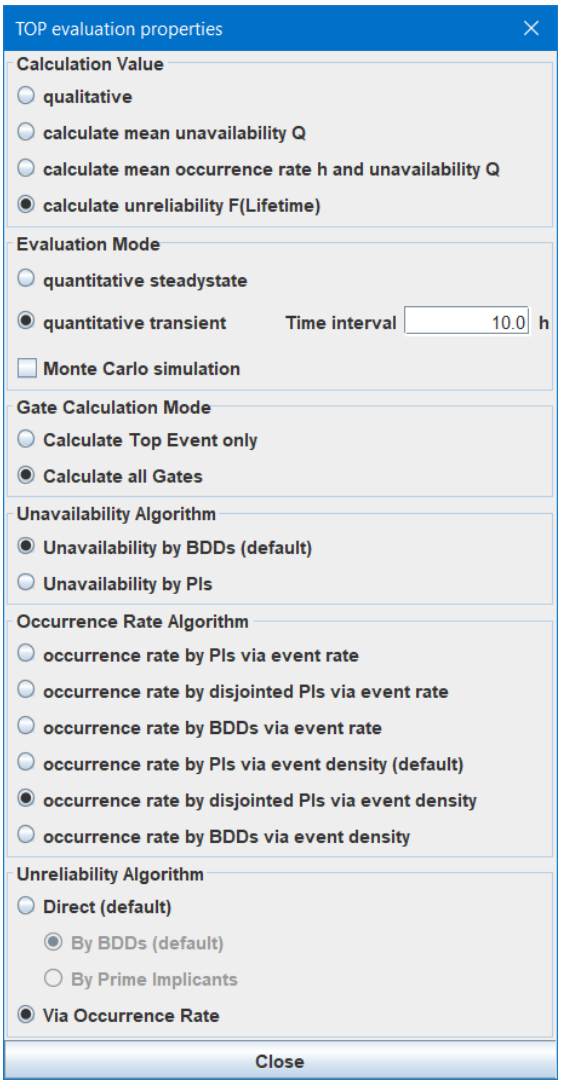


Figure 40: *Parameters for evaluation of unreliability*

7.6.3.1 Calculation Value

Select which value(s) to calculate:

- If **qualitative** is selected, the following operations are performed:
  - determination of minimal cut-sets of a *fault tree* or *reliability block diagram*.
  - optionally check a *fault tree* for consistency based on rules (version 7.1 includes an algorithm to check the rules stated in the [SiRF]).

  Events of *fault trees* provide a second text line in the name field below the name, intended to be used to indicate some "safety level" or another qualitative specifier. The content of the second text line can be entered separately for each event and belongs to this event, not to the *generic basic event* (as the quantity related values do). This is

necessary, because the "safety levels" of events (=component failures) of qualitative *fault trees* (maybe better called "occurrence rate levels" or "unavailability levels") are often determined top-down (similar to a THR apportionment) and therefore different levels might be applied to the same (generic) event at different positions. The *second text line* is stored together with the *gate* or *basic event* in the fault tree file — even if the type is set to *quantitative*. See section 7.8 for more information.

- If **calculate mean unavailability Q** is selected, only $\overline{Q}$ is calculated. In case of a transient evaluation, *Q(t)* is available for display in a *chart frame*.
- If **calculate mean occurrence rate h and unavailability Q** is selected, in addition to $\overline{h}$ and $\overline{Q}$ the estimated number of occurrences of the top event *N(T)* is calculated based on $\overline{h}$ or *h(t)*, see below. In case of a transient evaluation, also the unconditional failure density *f(t)*, the semi-conditional failure rate *w(t)* and the unreliability *F(t)* are calculated and available for display in a *chart frame*, in addition to *h(t)*, *Q(t)*, *N(t)*.
- If **calculate unreliability F(Lifetime)** is selected, at least *F(t)* is calculated. If the unreliability is calculated directly, i. e. based on the unreliabilities of the *basic events* (see section 7.6.3.7), in case of the transient evaluation, also the unconditional failure density *f(t)* is calculated and thus can be displayed in a *chart frame* in addition to *F(t)*. If the unreliability is calculated via occurrence rate, the same algorithms are used as if **calculate mean occurrence rate h and unavailability Q** would be selected, but instead of $\overline{Q}$ and $\overline{h}$ the unreliability *F(t)* is displayed in the graphics. Consequently, in case of a transient evaluation *h(t)*, *f(t)*, *w(t)*, *F(t)*, *Q(t)* and *N(t)* are available in the *chart frame*.

Depending on the selection, the parameters dialog will be adapted in order to show only the relevant parameters, see figure 39 and figure 40 for examples.

7.6.3.2 Evaluation Mode

Select whether the *fault tree* shall be evaluated in steady-state mode or in transient (time-variant) mode.

Also select whether Monte Carlo simulation shall be used.

In case of transient evaluation, the time interval must be set as well. For transient evaluation based on prime implicants or BDDs, you can use a quite small step size, such as 10 h or even 1 h. For transient evaluation by Monte Carlo simulation, at most 10000 steps are allowed, but you should go for about 1000 steps.

7.6.3.3 Gate Calculation Mode

Usually you should calculate all gates, because the intermediate gate values typically help understanding the *fault tree* and the critical paths. However, in order to save calculation time, you might want to calculate the top event only.

Calculate top event only: Only the top event of the *fault tree* is calculated, but no lower gates. This option just saves evaluation time, the top results will be the same.

Calculate all gates: All gates of the active *fault tree* are calculated. Select this option if you want to analyze where the top event's results come from.

7.6.3.4 Steady-State Evaluation Unavailability Mode

Optimistic

In 'optimistic' mode, the mean unavailabilities of *generic basic events* are used, as calculated according to section 4. As explained here before, this is usually optimistic.

Corrected

In 'corrected' mode, also mean unavailabilities of *generic basic events* are used, but combinations of unavailabilities are multiplied by a factor greater than 1 (depending on the length of the cut-set), so that the result is for sure not too optimistic (however it might be pessimistic). Unfortunately this correction cannot be performed on BDDs directly, and thus if unavailability is calculated by BDDs (see section 7.6.3.5) the BDDs need to be converted to a sum of products of events, what requires quite some calculation effort for large fault trees.

Safe

In 'safe' mode, the maximum unavailabilities of *generic basic events* are used, as calculated according to section 4. Obviously this is pessimistic for most basic event types, and the longer the cut-sets, the more pessimistic is the result.

Hint: If a *fault tree* contains components modeled by *basic events* of type *repairable*, *standby* or *link*, that are tested at the same time and only rarely (e. g. in a preventive maintenance once per month or year), so that their unavailabilities are not obviously negligible, you should perform a transient evaluation instead of a steady-state evaluation to get precise results. If a *fault tree* contains *non-repairable* events, you should always go for transient evaluation.

7.6.3.5 Unavailability Algorithm

Calculation by BDDs

Calculations based on BDDs are both accurate and very fast even for huge *fault trees*. The only reason not to select this option is in steady-state evaluation with unavailability mode 'corrected' (see section 7.6.3.4) for big *fault trees*.

Calculation by PIs

Calculations based on PIs are more or less pessimistic due to missing disjointedness between cut-sets (or prime implicants). The only reason to select this option is in steady-state evaluation with unavailability mode 'corrected' (see section 7.6.3.4) for big *fault trees*.

7.6.3.6 Occurrence Rate Algorithm

If the minimal cut-sets (or prime implicants, PIs) of a *fault tree* are known, the occurrence rate of the top event h_{sys} can be estimated based on the occurrence rates and unavailabilities of the basic events by

$$h_{\mathrm{sys}} \approx \sum_{i=1}^{n_{\mathrm{PI}}} \left(\sum_{j=1}^{n_{\mathrm{Lit,PI_i}}} \left(h_j \cdot \prod_{k=1,k\neq j}^{n_{\mathrm{Lit,PI_i}}} q_{i,k} \right) \right) \tag{34}$$

The PIs are determined by an algorithm using modified BDDs (in fact ternary decision diagrams, TDDs), which is very fast. The evaluation of the PIs doesn't need much memory, but some computing time. In case of high unavailabilities ($Q > 0.5$) this estimation can be too conservative. However, since such high unavailabilities shouldn't occur in a safety related system, the estimation is sufficiently precise for most problems.

In case of high unavailabilities, a more precise estimation can be calculated based on the (unconditional) occurrence density w_{sys} and the unavailability Q_{sys}:

$$w_{\mathrm{sys}} = \sum_{i=1}^{n_{\mathrm{PI}}} \left(\left[\sum_{j=1}^{n_{\mathrm{Lit,PI_i}}} \left(w_j \cdot \prod_{k=1,k\neq j}^{n_{\mathrm{Lit,PI_i}}} q_{i,k} \right) \right] \cdot \prod_{j=1,j\neq i}^{n_{\mathrm{PI_j}}} \left(1 - \prod_{k=1}^{n_{\mathrm{Lit,PI_j}}} q_{j,k} \right) \right) \tag{35}$$

$$h_{\mathrm{sys}} = \frac{w_{\mathrm{sys}}}{1 - Q_{\mathrm{sys}}} \tag{36}$$

This algorithm obviously needs some more calculations and is therefore slower.

Both algorithms are conservative estimations also due to the fact, that the prime implicants are not disjointed. The exact calculation requires disjointed prime implicants, which need to be determined by converting the PIs to BDDs again, one BDD per literal. This is a resource intense operation and thus is only possible for small to medium size *fault trees*. But once the BDDs have been created (if they can be created with given memory resources), numerical evaluation will be very fast.

In addition to these algorithms, an algorithm not using PIs at all is implemented, i. e. the occurrence rate is directly calculated by BDDs. This is the fasted algorithm. Unfortunately there is no formal proof of the correctness (or at least conservativeness), therefore you shouldn't rely on it before having it crosschecked by another algorithm.

All possible combinations of algorithms are available for selection. Algorithms using the occurrence rates of the basic events are always somewhat faster than their counterparts using unconditional occurrence frequencies (by a factor of 2 approximately).

occurrence rate by PIs via rate The occurrence rate is calculated based on the occurrence rates and unavailabilities of the basic events contained in the PIs. Doesn't need much memory, but high computing effort for transient evaluation. Result is conservative, in particular for high unavailabilities or large *fault trees*.

occurrence rate by disjointed PIs via rate PIs are sorted for literals and then being disjointed by BDDs. The occurrence rate is then calculated based on the occurrence rates and unavailabilities of the basic events. Needs much memory, but only medium computing effort for transient evaluation. Result is slightly conservative, in particular for high unavailabilities or large *fault trees*.

occurrence rate by BDDs via rate The occurrence rate is directly calculated based on BDDs, using occurrence rates and unavailabilities of the basic events. No PIs are determined. Needs few memory and only small computing effort for transient evaluation. Result might not be correct for some trees (deviates from PI based algorithms in both directions).

occurrence rate by PIs via density The occurrence rate is calculated based on the occurrence densities and unavailabilities of the basic events contained in the PIs. Doesn't need much memory, but high computing effort for transient evaluation. Result is conservative, in particular for large *fault trees*.

occurrence rate by disjointed PIs via density PIs are sorted for literals and then being disjointed by BDDs. The occurrence rate is then calculated based on the occurrence densities and unavailabilities of the basic events. Needs much memory, but only medium computing effort for transient evaluation. Gives the correct result.

occurrence rate by BDDs via density The occurrence rate is directly calculated based on BDDs, using occurrence densities and unavailabilities of the basic events. No PIs are determined. Needs few memory and only small computing effort for transient evaluation.

7.6.3.7 Unreliability Algorithm

Up to version 4 of Functional Safety Suite, the unreliability has been calculated based on the occurrence rate $\overline{h}$ or h(t). This method is suitable for all kinds of systems, but a little conservative and quite slow. Version 5 provides an algorithm to calculate the unreliability directly. This is quite simple and fast, and in fact this is how all "traditional" fault tree tools calculate unreliability, and what is explained in all books including [EN 61025]. Unfortunately, this is wrong in case the fault tree contains conditions, as demonstrated in the following example.

Example 3:

A top event 'TE' occurs, whenever a periodic event 'H' appears and a condition 'Cond' is fulfilled when 'H' appears. This is modeled in the fault trees shown in figure 41. Event h appears every 1000 hours with a probability of 0.1. Thus, given a system lifetime of 200 000 hours, it will appear about 20 times (of course this is no fix value, but the expected value). In the left fault tree, the top event's unreliability is calculated "directly", i. e. by $F_{\mathrm{sys}} = F_{\mathrm{H}} \cdot Q_{\mathrm{Cond}} = 0.1$. Even though both F_{H} and Q_{Cond} are probabilities, they must not be multiplied, because they

are different quantities. The correct result has to be calculated based on the occurrence rate by $F_{\text{sys}} = 1 - \exp(-\overline{h}_{\text{sys}} \cdot T)$ with $\overline{h}_{\text{sys}} = \overline{h}_{\text{H}} \cdot Q_{\text{Cond}} = 1\text{E}{-}5\,/\text{h}$, as shown on the right.

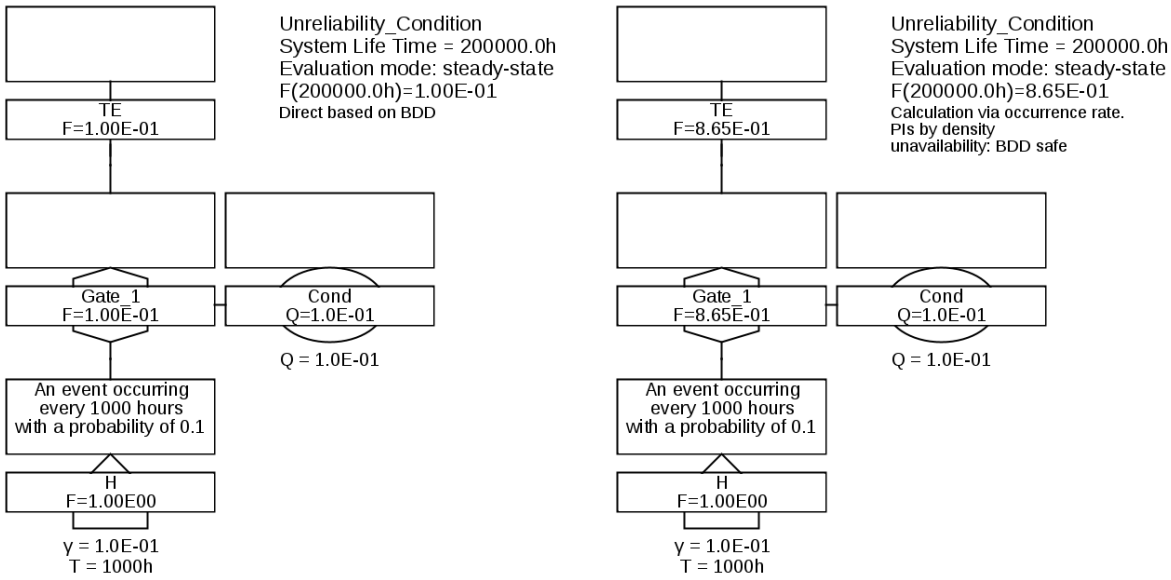


Figure 41: *Unreliability for a system with conditions – left: directly calculated (wrong), right: via occurrence rate (correct)*

Thus if the fault tree doesn't contain conditions, select **Direct**, if it contains conditions select **Via Occurrence Rate**.

In case of direct calculation of the unreliability, you can select whether it shall be calculated **By BDDs** or **By Prime Implicants** (minimal cut-sets). In fact there is no reason for using prime implicants — it is much slower than via BDDs and the result is pessimistic (by BDDs the result is exact).

If the unreliability *F(t)* is calculated via occurrence rate ($\overline{h}$), in steady-state evaluation it is calculated by

$$F(T) = 1 - \mathrm{e}^{-\overline{h} \cdot T} \tag{37}$$

and in transient evaluation it is calculated by

$$F(t) = 1 - \mathrm{e}^{-\int_0^t h(\tau)\, d\tau} \tag{38}$$

For small values of N ($N \ll 1$, as fulfilled typically for higher level events), $F \approx N$ is valid. For lower level events h often does not have the meaning of a failure rate, but determines usually occurring events, so that often $N \gg 1$ applies for a longer system lifetime, and $F \approx 1$ accordingly.

7.6.4 Evaluation Parameters – Monte-Carlo Simulation

When the **Monte Carlo simulation** checkbox is selected, the *Monte Carlo Simulation Parameters* panel (figure 42) is shown instead of the panels described above.

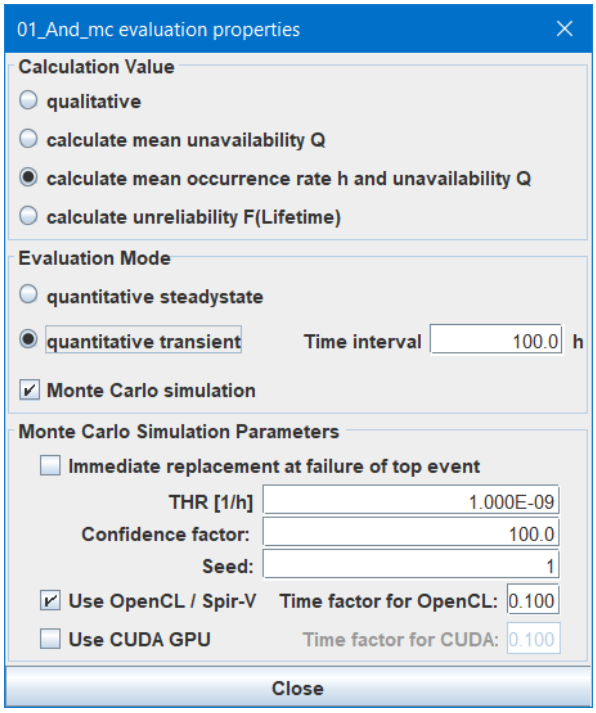


Figure 42: *The fault tree's evaluation parameters panel for Monte-Carlo simulation*

7.6.4.1 Immediate Replacement

Set checkbox **Immediate replacement at failure of top event** if the system described by the *fault tree* is completely replaced when the top even occurs. This will lead to reset of all time variant *generic basic events* in the *fault tree* in case of the top failure. The effect will only be visible in case of a quite high top event failure rate.

7.6.4.2 Tolerable Unavailability, Tolerable Hazard Rate, Tolerable Unreliability

Depending on the selected calculation value (see section 7.6.3.1) the tolerable unavailability Q_{tol}, the tolerable hazard rate THR or the tolerable unreliability F_{tol} can be entered. The smaller the tolerable value, the more simulations will be performed, see next paragraph.

In addition, this value is used for calculation of the achieved probabilistic confidence, see section 7.6.2.

7.6.4.3 Confidence Factor

Since it is not possible to calculate the necessary number of simulations for a given tolerable value or required confidence prior to the simulation, the number of simulations to be performed

needs to be selected by the user. One simulation means the simulation of one instance of the system over its lifetime as defined in the *project properties dialog*.

The number of simulations n_{sim} is calculated as follows, depending on the calculation value, the tolerable value, the system lifetime T and the confidence factor c:

Q: $n_{sim} = \frac{c}{T \cdot Q_{tol} \cdot 1e{-}4/\text{h}}$

h: $n_{sim} = \frac{c}{T \cdot h_{tol}}$

F: $n_{sim} = \frac{c}{F_{tol}}$

7.6.4.4 Seed

The Monte Carlo simulation uses pseudo random numbers during evaluation. With the same seed you'll get the same result whenever you perform a calculation (given all other parameters are the same as well, of course). [6] If you for some reason want to perform a different random experiment, set another value here.

7.6.4.5 Use OpenCL / Spir-V

If your computer is equipped with a device (GPU or CPU) supporting OpenCL™ with SPIR-V™ you can use the Spir-V library provided with Functional Safety Suite. This will typically speed up simulation by 5 to 1000 times — depending on the exact CPU or GPU.

7.6.4.6 Time Factor for OpenCL

The OpenCL driver will cancel the calculation after a few seconds. Functional Safety Suite tries to estimate the complexity of the simulation and will split the simulation of one system lifetime into shorter parts when necessary. In order to avoid those failures, you might need to adjust this value to smaller values. Larger values typically don't make sense since only simple simulations might speed up a little bit, but they are quite fast anyhow.

7.6.4.7 Use CUDA GPU

If your computer is equipped with a Nvidia® GPU with compute capability 5.0 or higher, you can use the CUDA libraries provided with Functional Safety Suite. This will typically speed up simulation by 20 to 1000 times — depending on the exact GPU.

7.6.4.8 Time Factor for CUDA

The GPU driver will cancel the calculation after a few seconds. (leading to a restart of the GPU driver that might come along with some short flickering of your monitor). Functional Safety Suite tries to estimate the complexity of the simulation and will split the simulation of one system lifetime into shorter parts when necessary. In order to avoid those failures, you

[6] This only applies as long as the same `dll` or CUDA driver is used.

might need to adjust this value to smaller values. Larger values typically don't make sense since only simple simulations might speed up a little bit, but they are quite fast anyhow.

7.7 Modularization and Common Causes

7.7.1 Handling of Common Causes

Two common cause scenarios are to be considered:

1. Multiple events in the *fault tree*(s) depict the identical event and have therefore the same name and suffix. If those events are conjuncted via AND or COMBINATION gates, all duplicates are deleted in the minimal cut-set, thus the common cause factor stated in the *basic event* properties is meaningless.

2. Multiple events in the *fault tree*(s) refer to different components in the system and therefore have the same name `X` (pointing to the same *generic basic event* `X`) but different suffix. In that case, when calculating the unavailability and occurrence rate of a minimal cut-set, the common cause factor β stated in the *basic event* properties is considered. Therefore the *generic basic event* `X` is split into two *generic basic events* internally: One containing the single-cause part $(1 - \beta) \cdot Q$ and $(1 - \beta) \cdot h$, the other containing the common-cause part $\beta \cdot Q$ and $\beta \cdot w$. The *generic basic event* of the common cause part has the same name `X`, but a suffix `COM`, as you can see in the minimal cut-set list (see section 11.6.3).

See figure 43 for an example. Given two components `X.1` and `X.2` of same type `X`. A common cause factor β of 1% is assumed between components of this type due to their construction, manufacturing and environment. For one function 'A' at least one of the two components `X.1` and `X.2` is needed, in another function 'B' component `X.1` and another component `Y` is needed.

As expected, a common cause factor of 0.01 is used between the two components `X.1` and `X.2` of the same type `X` (see the values of 'FKT_A_FAILS') whereas events `X.1` are treated as the identical event: 'FKT_B_FAILS' has no impact to the top event since event `X.1` in 'FKT_B_FAILS' will always occur whenever event 'FKT_A_FAILS' occurs (obviously such a system architecture makes no sense). 'FKT_B_FAILS' is not even calculated, since component `Y` doesn't occur in the cut-sets at all and therefore is not initialized. You can check the cut-sets by clicking **Calculate – Show Minimal Cut-sets**.

Note that if a *basic event* is selected, all *basic events* referring to the same *generic basic event* as the selected one are highlighted in blue color as well (see figure 43). This applies also to *basic events* in other models, thus when you switch to another model, you can also directly see the related *basic events*.

7.7.2 Modularization of Fault Trees

For the same reasons as in other fields of engineering, it usually makes sense to split a complex system into several 'modules' or 'units'. This is the same for *fault trees*. In fact one will aim

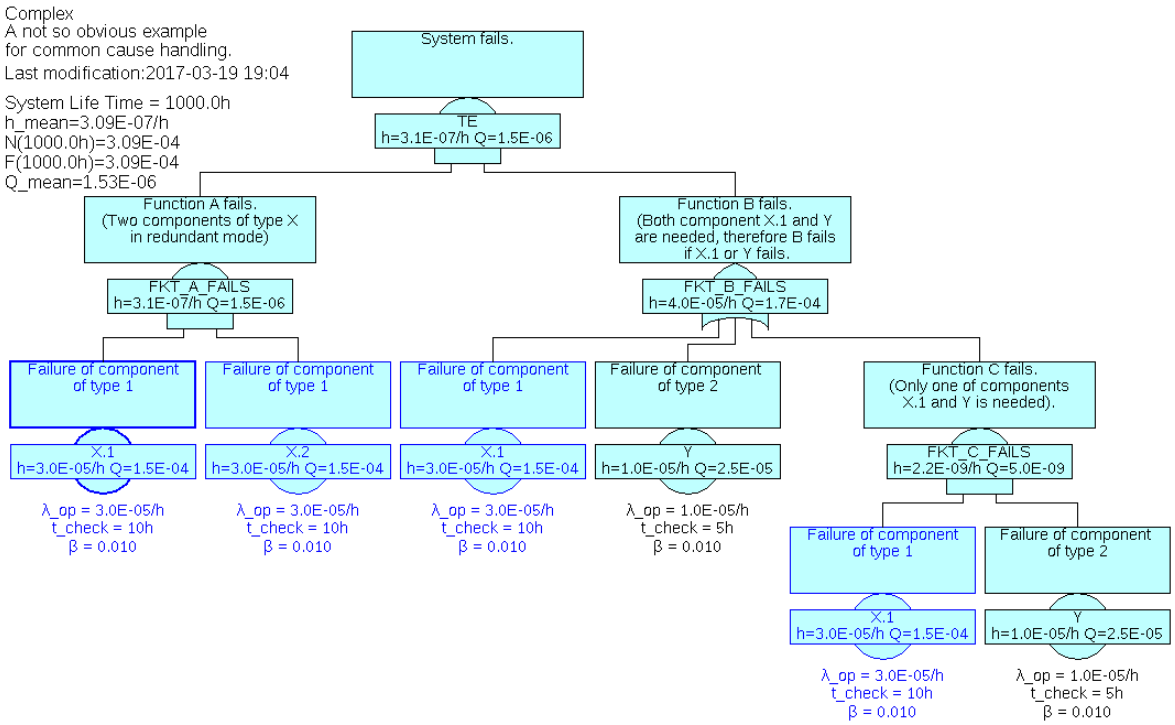


Figure 43: *Example for common cause handling by suffixes*

to reflect the architecture of the system in the *fault trees*.

See figure 44 for an example of a system, consisting of four modules, characterized as follows:

- The overall system consists or two units of type B, named B1 and B2.
- The overall system fails, when both units B1 and B2 fail.
- Each unit of type B includes two components of the same type X, named X1 and X2 per unit B.
- Each unit of type B includes one component of type Y.
- Each unit of type B uses the same two external units of type A, named A1 and A2.
- A unit of type B fails, when both units A1 and A2 fail, or any of its own components X1, X2 or Y.
- Each unit of type A can fail due to three faults, FAULT1, FAULT2, FAULT3.
- FAULT3 will always let all units of type A fail.

Units A could be sensors, whereas B1 and B2 could be computers working with the sensor data.

The *fault tree* of this system is shown in figure 45.

At least if units A and B are even more complex, you'll feel a need to split this tree into several *sub-trees*. In principle Functional Safety Suite provides two possibilities: TRANSFER-IN gates and *links*.

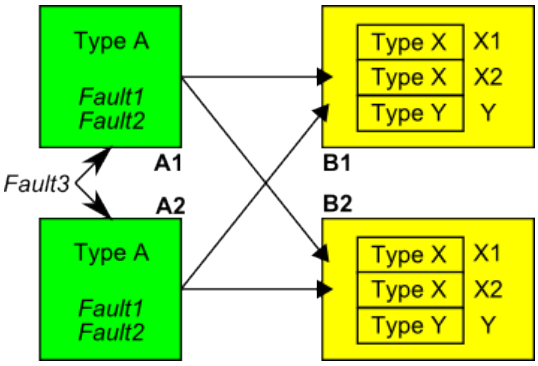


Figure 44: *Block diagram of a complex system*

7.7.2.1 Modularization by TRANSFER-IN gates

Apart from dividing large *fault trees* to several pages, gates of type TRANSFER-IN are used in two cases:

1. the same unit is needed for multiple higher level units or (sub-)functions. In that case, the TRANSFER-IN gates referring to the *sub-tree* shall represent the same unit and thus the identical event.
2. a unit is existing twice or more in the system. In that case, the *sub-tree* describes one of these units. The *fault tree* uses multiple TRANSFER-IN gates referring to this *sub-tree*, whereas each TRANSFER-IN gate shall represent a different unit and thus different events.

The two cases are handled in a simple way in Functional Safety Suite:

- If the names of TRANSFER-IN gates are equal, the referred *sub-trees* are considered to relate to the same unit (case 1).
- If the names of TRANSFER-IN gates differ, the referred *sub-trees* are considered to relate to different units (case 2).

However imagine that in case 2, the *sub-tree* refers to some events, that are not specific to each unit described by this *sub-tree*, but shared by all units of this kind (and maybe even other units). In order to describe also this case correctly, the following rule applies:

Rule 1: If the suffix of a *basic event* ends with '#', a new *basic event* will be created internally when constructing the cut-sets during calculation. The suffix of the new *basic event* will be the name of the TRANSFER-IN gate plus '.' plus the original suffix (excluding the '#'). [7]

In case of multiple levels of TRANSFER-IN gates, the names of all TRANSFER-IN gates will be stringed together, separated by '.' and starting with the lowest level.

By this mechanism, the overall example system could now be split as shown in figures 46 and

[7] Prior to version 6.0, the complete original suffix including the '#' has been appended. From version 6.0 on, the '#' is omitted in the final fault tree, because it's not needed anymore.

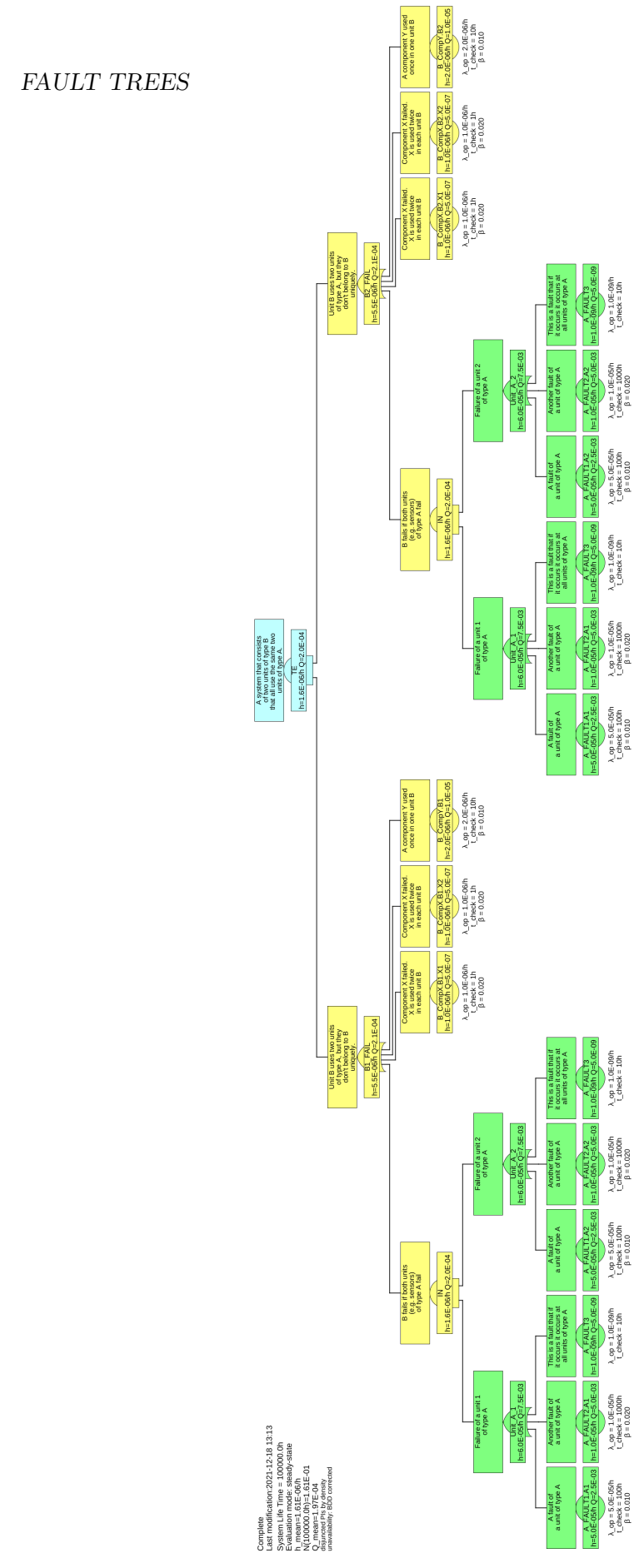


Figure 45: *Example for a complex system*

47.

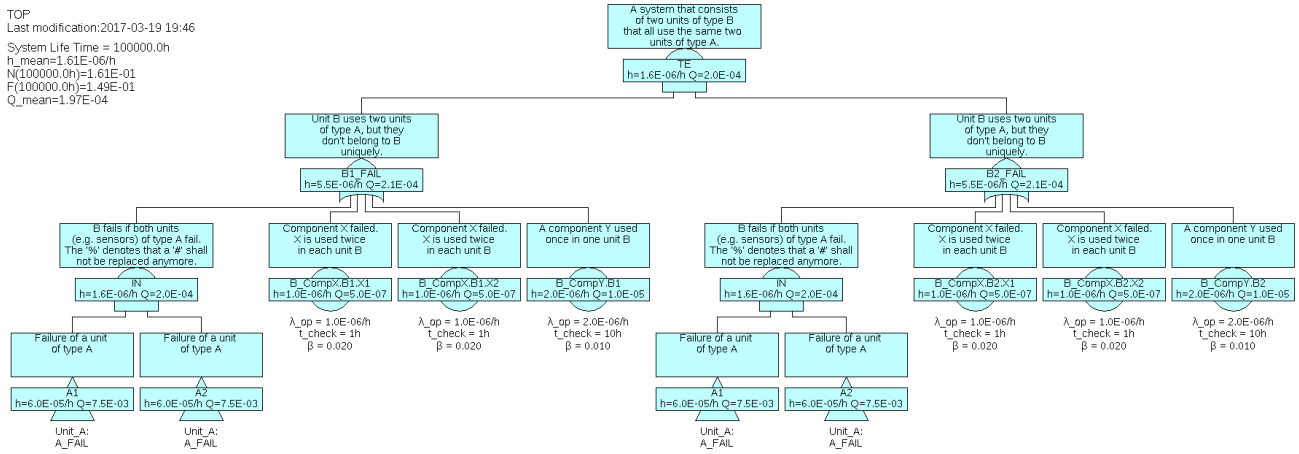


Figure 46: *Example: Separation of units of type A (1)*

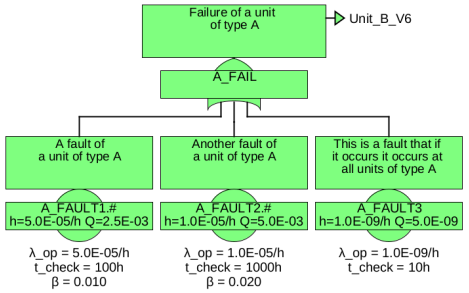


Figure 47: *Example: Separation of units of type A (2)*

The suffixes '#' of 'A_Fault1' and 'A_Fault1' will be extended to 'A1' and 'A2'. Now you might also want to model units of type B with one generic *fault tree* B. If you'd just replace the gates 'B1_Fail' and 'B2_Fail' by TRANSFER-IN gates, rule 1 would apply again. Thus the suffixes '#' of 'A_Fault1' and 'A_Fault1' would be extended to 'A1.B1_Fail', 'A1.B2_Fail', 'A2.B1_Fail' and 'A2.B2_Fail', i. e. you'd model independent units A1 and A2 for each of the units B1 and B2. But remember, that the same units (*sub-tree*) A1 and A2 are needed in all units of type B. So you'll have to tell in tree B, that you mean identical A's in all instances of B.

Therefore the following rule applies:

Rule 2: If the name of the TRANSFER-IN gate ends with '%' [8], or if a *transfer extension* is stated, the names of higher level TRANSFER-IN gate names will be ignored.

Given this, it is possible to split the top tree as shown in figure 48.

In any case you can check the result by having a look at the list of minimal cut-sets (see section 11.6.3). It should always be as shown in table 5.

[8]Note: The name of the TRANSFER-IN gate must not be '%' only.

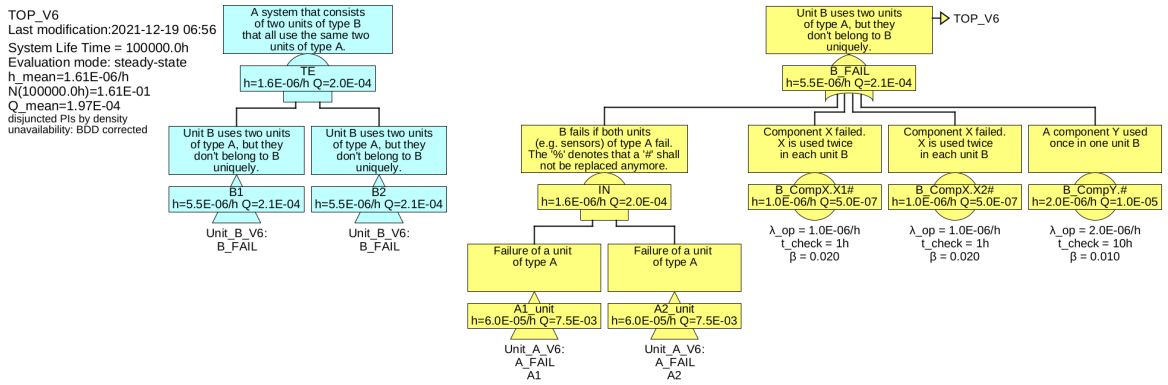


Figure 48: *Example: Top tree (left) referring to two generic units of type B (right), each referring to the same two generic units of type A.*

Table 5: *Minimal cut-sets for the example*

Cut-set
A_FAULT1.COM
A_FAULT2.COM
A_FAULT3
B_CompX.COM
B_CompY.COM
B_CompX.B1.X1 * B_CompX.B2.X1
B_CompX.B1.X2 * B_CompX.B2.X1
B_CompX.B2.X1 * B_CompY.B1
B_CompX.B1.X1 * B_CompX.B2.X2
B_CompX.B1.X2 * B_CompX.B2.X2
B_CompX.B2.X2 * B_CompY.B1
B_CompX.B1.X1 * B_CompY.B2
B_CompX.B1.X2 * B_CompY.B2
B_CompY.B1 * B_CompY.B2
A_FAULT1.A1 * A_FAULT1.A2
A_FAULT1.A2 * A_FAULT2.A1
A_FAULT1.A1 * A_FAULT2.A2
A_FAULT2.A1 * A_FAULT2.A2

7.7.2.2 Modularization by Links

The linking mechanism (see sections 2.4 and 4.3.7) provides another possibility to split *fault trees*. In the given example, it can be used as shown in figure 49. The difference to using a TRANSFER-IN gate is, that the link hides all internal details of 'IN' from *fault trees* using this link.

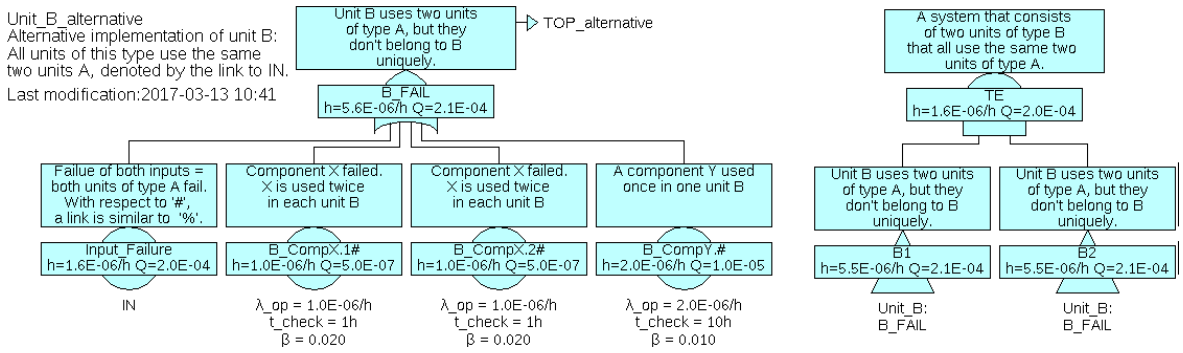


Figure 49: *Example: Separation of units A by the use of a link*

7.7.3 REDUCED-COMBINATION Gates and deleting Common Cause Contributions

In contrary to all other gates, REDUCED-COMBINATION gates numerically evaluate the list of minimal cut-sets of the input event (they calculate q_i and h_i of the input). Unavailability q_g and occurrence rate h_g of the gate are then calculated as function of n, m, q_i and h_i and stored in a new temporary *generic basic event*, referred by a new temporary *basic event*. Minimal cut-sets of higher level events only contain this *basic event* and thus do not contain lower level information including any common cause factor.

Since they reduce information, they are called *'Reduced' Combination* gates. This reduction has three reasons:

1. Implicit conversion to AND and OR gates as done in standard COMBINATION gates is not feasible any more for very large number of inputs (n).

2. The calculation of Q and h based on minimal cut-sets is not accurate if the same events occur in many cut-sets. The calculation error is typically $\ll 1\%$ and therefore negligible, but when the same events occur in hundreds or thousands of cut-sets, it is not negligible any more. Correct calculation by disjointing the cut-sets on the other hand is by far too expensive especially for those cut-sets, where it would be necessary.

3. For many problems, a common cause between similar components is not assumed any more on a certain higher level (e. g. common causes have to be considered for components of the same type in one box, but several boxes of this type don't share all those common causes any more since they are located far away from each other). Therefore a mechanism is needed to cut these low-level common causes in higher levels — which is provided by this kind of gate.

The suffix of the temporary (internal) *basic event* can be set to '#', in order to create multiple instances of the element described by the REDUCED-COMBINATION gate in higher level trees. This is achieved by adding an '#' at the end of the name of the REDUCED-COMBINATION gate.

7.8 Specifics of qualitative Fault Trees

The decision, whether *fault trees* are qualitative or quantitative is defined in the *Evaluation properties* of each *fault tree*, see section 7.6.3. [9] No data gets lost if the type is changed between qualitative and quantitative (steady-state or transient). Thus it is possible to use the same *project* and the same *fault tree* for qualitative and quantitative evaluation.

Events of qualitative fault trees are sometimes classified in some way. For example a 'safety level' (SL) is assigned to each of them. Therefore qualitative *fault trees* provide a *second text line*, displayed in the name field below the name, intended to be used to indicate some "safety level" or another qualitative specifier. The content of the second text line can be entered separately for each event and belongs to this event, not to the *generic basic event* (as the quantity related values do). This is intended due to the fact, that the "safety levels" of events of qualitative *fault trees* are often determined top-down and therefore different safety levels might be assigned to the same event in different trees — or even within different branches of the same tree.

The *second text line* is stored together with the *gate* or *basic event* in the fault tree file (`.ftl`).

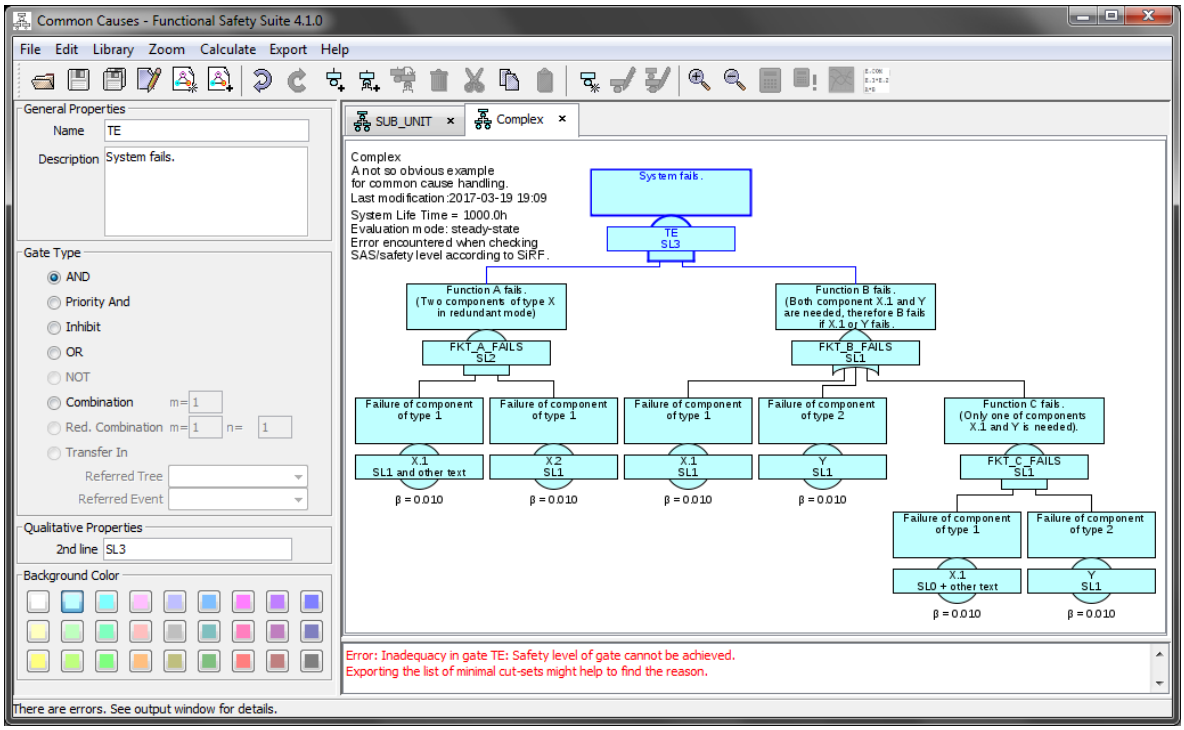


Figure 50: *A qualitative fault tree*

Since there is no mathematical rule how to "calculate" the SL of a *gate* based on the SL's of the input *gates* or *basic events*, this task has to be performed manually. Therefore when evaluation type 'qualitative' is selected, the *second line* can be filled with an arbitrary text — as for example the assigned SL.

[9] Up to version 6.0 it was a project parameter, from version 7.0 you can select for each *fault tree* separately.

Nevertheless there might be rules of how to assign or apportion classes or SL's. One example are the rules defined for the authorization of railway vehicles in Germany ([SiRF]). Functional Safety Suite includes an algorithm to check qualitative fault trees according to the [SiRF] rules.

The qualitative evaluation is always based on minimal cut-sets.

7.9 Editing of Fault Trees

Create a *fault tree* by **File – New Model**. Select a name and *package* for the new tree that will be created. Finally a simple *fault tree*, only consisting of the top gate, is created.

Select the top gate by clicking on it. Add a *gate* below by **Edit – Add Gate**.

Add a *basic event* based on a new *generic basic event* below the selected *gate* by **Library – New Generic Basic Event**. After entering a name, a new *generic basic event* will be created, and a *basic event* referring to this *generic basic event* will be added below the selected *gate*. Note that the new *generic basic event* is created in the *local package* of the *fault tree* by default, but you can also choose to create it the *global package*.

Add a *basic event* referring to an existing *generic basic event* below the selected *gate* by **Edit – Add Tree Basic Event**. The new *basic event* will refer to the latest *generic basic event* by default. You can select any other existing *generic basic event* by selecting it via its name and *package* in the *Tree Basic Event Properties Panel*, see section 7.3.

The sequence of events below a *gate* can be changed by 'Shift-→' and 'Shift-←'.

A *gate* or a *basic event* can be deleted with the 'Delete' key. In case of a *gate*, the inputs of the deleted *gate* will be added to the parent *gate*.

A branch can be deleted by 'Shift+Delete'.

If you want to assign a new (not yet existing) *generic basic event* to an existing *basic event*, select **Library – New Generic Basic Event**. The name of the *basic event* will change to the name of the new *generic basic event*, showing that it now refers to the new *generic basic event*.

You can copy or cut branches by selecting the top *gate* of the branch and pressing 'Ctrl+C' or 'Ctrl-X'. The branch saved like this can be pasted below any *gate* of the same or another *fault tree* by 'Ctrl+V'. Neither the names of the *gates* nor the suffixes of the *basic events* of the pasted branch will be changed automatically, so it's up to you to change them according to their new meaning and relation to other events.

Changing properties of *gates* or *basic events* is done in the properties window. The only exception is the change of the name of a *generic basic event*, for which a special command **Library – Rename Generic Basic Event** is foreseen. The properties of the *generic basic event* referred by a *basic event* can be edited in the library view as well, see section 4.1.

A *fault tree* that has not been saved after the latest modification is marked with an asterisk '*' in its title.

8 Reliability Block Diagrams

A reliability block diagram (RBD) is a diagrammatic method for showing how component reliability contributes to the success or failure of a complex system. A RBD is drawn as a series of blocks connected in parallel or series configuration. Each block represents a component of the system with a failure rate. Parallel paths are redundant, meaning that all of the parallel paths must fail for the parallel network to fail. (Compare [Wikipedia])

Mathematically, a (standard) *reliability block diagram* is just a negated *fault tree*, only consisting of AND and OR gates: Parallel paths in a RBD represent redundancies, i. e. they are equivalent to conjunctions (AND-gates) in a fault tree, whereas serial connections are represented by OR-gates.

Consequently in Functional Safety Suite a *reliability block diagram* is internally represented by *fault tree* data, and evaluated according to *fault tree* rules. Thus all characteristics and options described in section 7 apply for *reliability block diagrams* as well, except of a few deviations explained hereafter.

8.1 Conjunctions Properties Panel

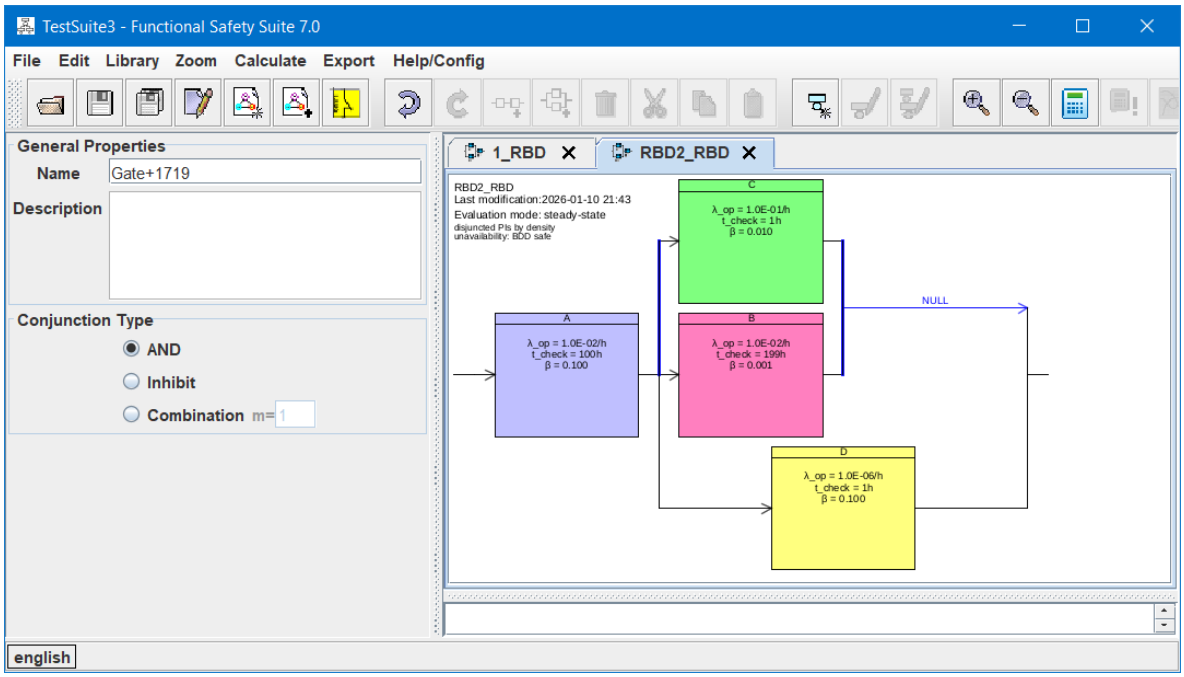


Figure 51: *Changing the type of a conjunction in a reliability block diagram*

Since in Functional Safety Suite a *reliability block diagram* is internally represented by a *fault tree*, even the additional conjunction types INHIBIT and COMBINATION can be used in an *reliability block diagram*. These types of conjunctions are indicated by *INH* or the number of paths that must fail on top of the vertical line on the right side of the block(s) that are children of this gate, see figure 51.

Only AND, OR, INHIBIT, COMBINATION and TRANSFER-IN gates are allowed in *reliability block diagrams*, since graphical representation in terms of parallel and serial blocks is not adequate for other types and would be potentially misleading. If you need other types, you should use a *fault tree* instead.

Unfortunately providing different types of conjunctions goes along with a graphical problem in RBDs, see section 8.4.

8.2 TRANSFER-IN Gates

Also TRANSFER-IN gates can be used in *reliability block diagrams*. However since gates are only indicated as lines in reliability block diagrams, TRANSFER-IN gates have to be shown as blocks, i. e. similar to basic events. Therefore in order to create a TRANSFER-IN gate, first add a new block, then convert the block to a TRANSFER-IN gate by **Edit – Convert to Transfer-In**. The properties panel will change so that you can select the referred model, see figure 52. The reference of the TRANSFER-IN gate can either be another *reliability block diagram* or another *fault tree*.

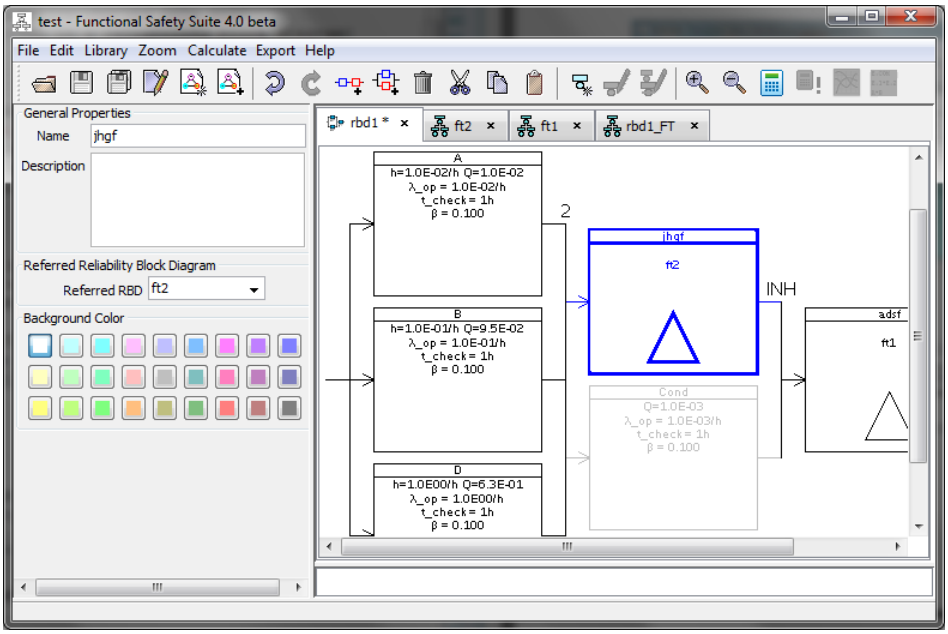


Figure 52: *Representation and properties of a Transfer-In gate*

Note that in contrary to *fault trees*, the reference will always be the top event of the referred *reliability block diagram* or *fault tree*.

8.3 Editing of Reliability Block Diagrams

Create a *reliability block diagram* by **File – New Model**. Select a name and *package* for the new *reliability block diagram* that will be created. Finally a simple *reliability block diagram*, consisting of one block only, is created.

Create a new *generic basic event* by **Library – New Generic Basic Event**. Usually you should create the *generic basic event* in the *local package*, which hence is the default location.

Click on the block to select it.

Add a new block referring to the latest *generic basic event* by **Edit – Add Block Serial** or **Edit – Add Block Parallel**. You can select any other existing *generic basic event* by selecting it via its name and *package*.

Multiple blocks can be selected by drawing up a selection rectangle by pressing the left mouse button and pulling the mouse, or by clicking on multiple blocks while pressing the left mouse button.

The sequence of blocks in a series can be changed by 'Shift+→' and 'Shift+←', the sequence of parallel blocks can be changed by 'Shift+↑' and 'Shift+↓'. You can select one or multiple blocks for moving.

A single block or a selection of multiple blocks can be deleted with the 'Delete' key.

If you want to assign a new (not yet existing) *generic basic event* to an existing block, select **Library – New Generic Basic Event**. The name of the block will change to the name of the new *generic basic event*, showing that it now refers to the new *generic basic event*.

You can copy or cut selections of one or multiple blocks by pressing 'Ctrl+C' or 'Ctrl+X'. The selection saved like this can be pasted by **Edit – Paste Serial** or **Edit – Paste Parallel**. 'Ctrl+V' is equivalent to **Edit – Paste Serial**. Don't forget to change the suffixes when coping blocks if necessary.

8.4 NULL Blocks

In a standard RBD, it is impossible that two "AND-gates" are directly linked, i. e. that one is the parent gate of the other, because the two "And-gates" would just be merged to one.

However if we have different types of conjunctions following each other, e. g. an INHIBIT and an AND, we have to distinguish the "gates" graphically. This is done by introducing *NULL* blocks, see figure 53. A *NULL* block is a *generic basic event* with no failure rate ($h = 0$) and no unavailability ($Q = 0$).

In figure 53 the INHIBIT conjunction is OR'ed with the *NULL* block just in order to put it below an AND conjunction. Mathematically the *NULL* block has no effect of course.

If you need to add a *NULL* block, you'll find it in the *global library*.

When deleting blocks in a serial structure, a *NULL* block is automatically inserted where necessary. When converting a *fault tree* to a *reliability block diagram*, *NULL* blocks are automatically inserted where necessary as well. However even this automatism doesn't guarantee that you will always be able to visually distinguish all different combination gates in some structures. Therefore you should be careful using combination gates different from AND, or prefer using a *fault tree* instead.

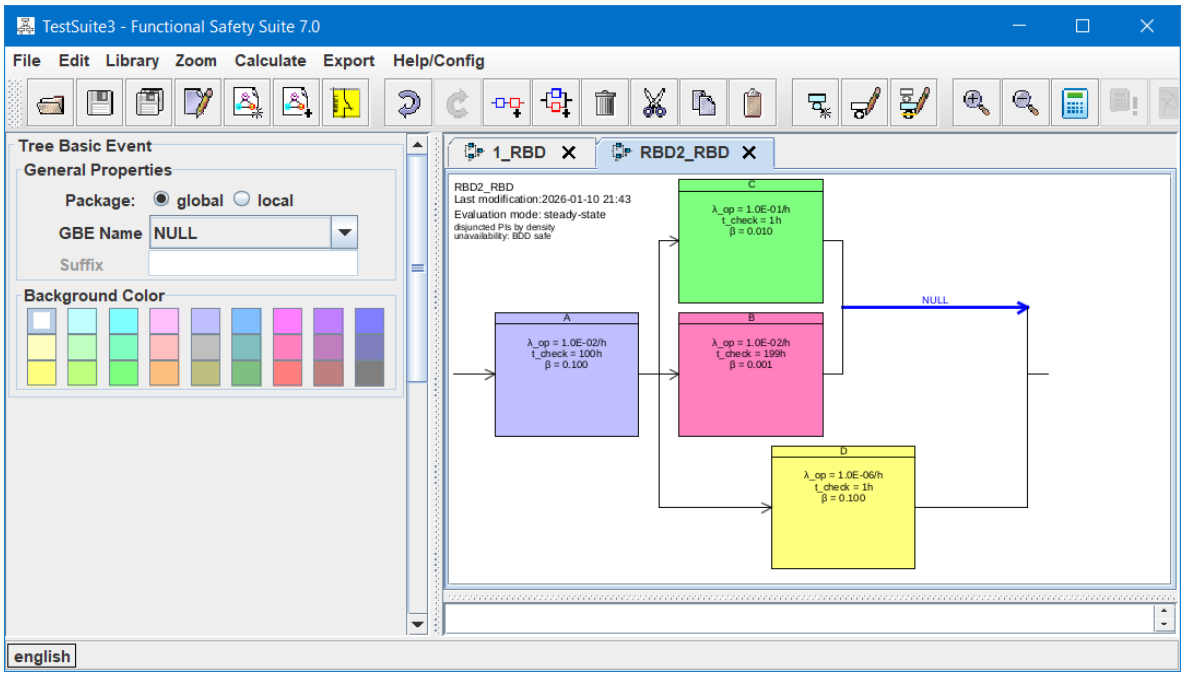


Figure 53: *The NULL block*

8.5 Pro's and Con's of RBDs versus FTAs

Pro's RBD:

- Block diagrams are obviously a good method to present the structure of a system. This is why they are used in thousands of variants, in all fields of engineering and even other sciences.

Con's RBD:

- A technical system doesn't consist of "series" and "parallel" structures only, but other structures (such as enabling/disabling components) are not supported by (standard) RBD's.
- The RBD only shows the logical structure (and even this only as long as it is similar to the physical structure) on global level. There is no way of describing the behavior or functionality of a group of components in the RBD, because the "gates" are not explicitly shown and therefore no descriptions can be added to the "gates".
- When using different types of conjunction gates, it isn't automatically safeguarded that you will actually see the different types. Therefore if you want to use conjunction gates different from AND, you should use a *fault tree*.

Pro's FTA:

- The "top-down" approach allows to analyze even very complex systems, without increasing the risk of introducing errors in the structure. This approach is supported (or even only enabled) by the explicit presentation of gates, allowing for additional intermediate

descriptions.
- The explicit presentation of gates allows nice graphical representation of additional gate types.

Con's FTA:

- More abstract than a block diagram, thus the structure is not immediately visible.

9 Markov Models

9.1 Introduction and Overview

Markov models are another commonly used method for hazard analysis and reliability analysis in general. The values describing the "quality" of a system,

- the mean occurrence rate $\overline{h}$ (PFH),
- the mean unavailability on demand $\overline{Q}$ (PFD),
- or the probability of mission failure $\mathrm{F}(T_{\mathrm{mission}})$

can be calculated for Markov models as well. For more details see section 9.5.

9.1.1 States and Edges

In contrast to fault trees, Markov models are created inductively (bottom-up): Starting with the up-state (indicating that the system is completely ok), events that can occur in this *state* are identified, and the resulting system states respectively. Then for each new identified state, the possible events and consecutive states are identified — and so on, up to when no further change of the system's state is possible (except due to restoration, see below). This final state is typically a fail state of the system, whereas the preceding states are intermediate states. In an intermediate state, the system described by the model still performs its function, but it has detected or hidden faults.

The *edge* between two *states* represents a transition from one state of the system (the *source state* of the *edge*) to another state (the *target state* of the *edge*). In standard Markov models this transition takes place due to the occurrence of an event, thus an *edge* is characterized by the occurrence rate of this event (compare [EN 61165]).

Fail state for a low demand function means, that the function is unavailable in this state, namely the probability of this state contributes to the unavailability Q (PFD) of the function.

Fail state for a high demand or continuous demand mode function means, that the transition rate(s) towards this state contribute to the failure rate h (PFH) of the function.

Fail state for a non-repairable function means, that the probability of being in this state contributes to the unreliability *F(t)*.

Note: If you want to calculate the unreliability *F(t)*, there should be no restorations from any of the fail states, because in that case it would be a restorable system, but the unreliability *F(t)* is not relevant for restorable systems. If there are restorations anyhow, the unreliability cannot be calculated based by just summing up the probabilities of the fail states at time t. Therefore, the unreliability will be calculated based on the failure density $f_{\mathrm{sys}}(\mathrm{t})$ instead, which is derived from the failure rate $h_{\mathrm{sys}}(\mathrm{t})$.

9.1.2 Restoration

If no restoration would take place, after some time the system would be in one of the final states (*fail states*), thus the sum of the probabilities of these states would be 1, no events and thus no transitions could occur anymore. All systems that shall work for a long time usually need some kind of restoration, based on preventive of corrective maintenance. In Markov models, restorations are also events, represented by *edges*, and thus characterized by transition rates. Even they are usually called *restoration rate*, mathematically they are transitions, just as those representing failures or any other event.

If an *edge* represents the correction of exactly one failure, this *restoration edge* is exactly anti-parallel to the related *failure edge* (the 'forward' part), it 'returns' to the *source state*. In *fault trees* both failure and restoration are defined in each *basic event* (if there is a restoration) — and since Functional Safety Suite aims for best equivalency between *fault trees* and *Markov models*, this principle is used for *edges* as well. Therefore instead of defining two independent *edges*, the restoration is inherent to the *edge* describing the failure. Thus the same *generic basic events* can be used for *edges* as for *basic events* of *fault trees*. The *restoration rate* μ is automatically calculated and written below the *edge*.

Important note: Usually inspections and tests are executed in predefined intervals. Thus in redundant architectures, all channels are checked and repaired at the same time. Thus all restorations related to these inspections and tests will take place at the same time, not independent of each other. For correct modeling, you should separate the restoration from the failure part of the event, see section 9.3.1.6.

The direction of the restoration is indicated by a small text arrow left of the restoration parameter text below the *edge*, see figure 54.

9.1.3 Extensions of Functional Safety Suite related to Markov Models

9.1.3.1 Conditions and Instantaneous transitions

Functional Safety Suite provides some extensions to standard Markov models, named *instantaneous transitions* and *cyclic transitions*, in addition to *transition rates*.

An *instantaneous transition* is not a defined by a transition rate, but a transition probability. That means, the *source state* is immediately left with the probability represented by the *edge* towards the *target state*. The sum of the probabilities of the *instantaneous transitions* leaving a state must be 1. Thus the probability of finding the system in the *source state* of such *edges* is 0, and the *state* is called *virtual state* accordingly. Since the probability of a *virtual state* is 0, *edges* of another type (not instantaneous) starting in a *virtual state* don't make sense and are therefore forbidden.

Instantaneous transitions are necessary to model *conditions*. Conditions are necessary in two scenarios:

Condition events: In a *fault tree*, some *basic events* describe a probability instead of an

occurrence rate. Those *basic events* usually serve as *condition* for a *inhibit gate*. For the same reasons as for fault trees, also in Markov models sometimes a condition needs to be modeled.

Modularization: Condition events are also a basis to make modularization and encapsulation possible: If a higher level Markov model shall reuse results of a lower level model, the lower level often also describes an unavailability, and thus a probability.

If the *edges* leaving a *state* are determined by probabilities (namely *Q(t)* and *A(t)=1-Q(t)*), the *edges* represent *instantaneous transitions*. That means, that this *state* is immediately left with the probability represented by each *edge* towards the next *target state*. The sum of the probabilities of the *instantaneous transitions* must be 1, and there must be no *edges* of another type starting in this *state*. Thus the probability of finding the system in the *source state* of such *edges* is zero, the *state* is called *virtual state* accordingly. *Virtual states* are displayed with gray text and circle.

9.1.3.2 Cyclic Events

An event that periodically appears, can be modeled by a *generic basic event* of type *cyclic*. This leads to an edge describing a *cyclic transition*. This edge is described by a deterministic period and a probability, that it actually appears after each period. In contrary to *source states* of instantaneous transitions, the *source state* of a *cyclic transition* is not "virtual", since its probability is not (always) 0. Thus also other transitions are allowed with this *state* as source, e. g. continuous transitions, and the sum of the probabilities of *cyclic transitions* doesn't need to be 1.

9.1.4 Summary of Features of Markov Models in Functional Safety Suite

- Many occurrence/failure models for *edges*, including links to other *fault trees*, *Markov models* and *complex components*.
- Same *generic basic event* can be used for single cause and common cause part of each failure.
- Instantaneous transitions: Transitions can be determined not only by rates, but also probabilities.
- Transitions at certain discrete times.
- Allows states with constant probability.
- Internal creation of common-cause chains.
- Internal creation of completed Markov model.
- Two evaluation modes:
  - Steady state evaluation
  - Transient evaluation

9.2 The Markov Model Properties

Presentation related properties of the *Markov model* are edited in the *Markov model properties panel* directly, see below. Evaluation related properties are set in the *Markov model evaluation properties* dialog, see section 9.5. All properties of the *Markov model* are stored in the Markov model file (extension `.mdg`).

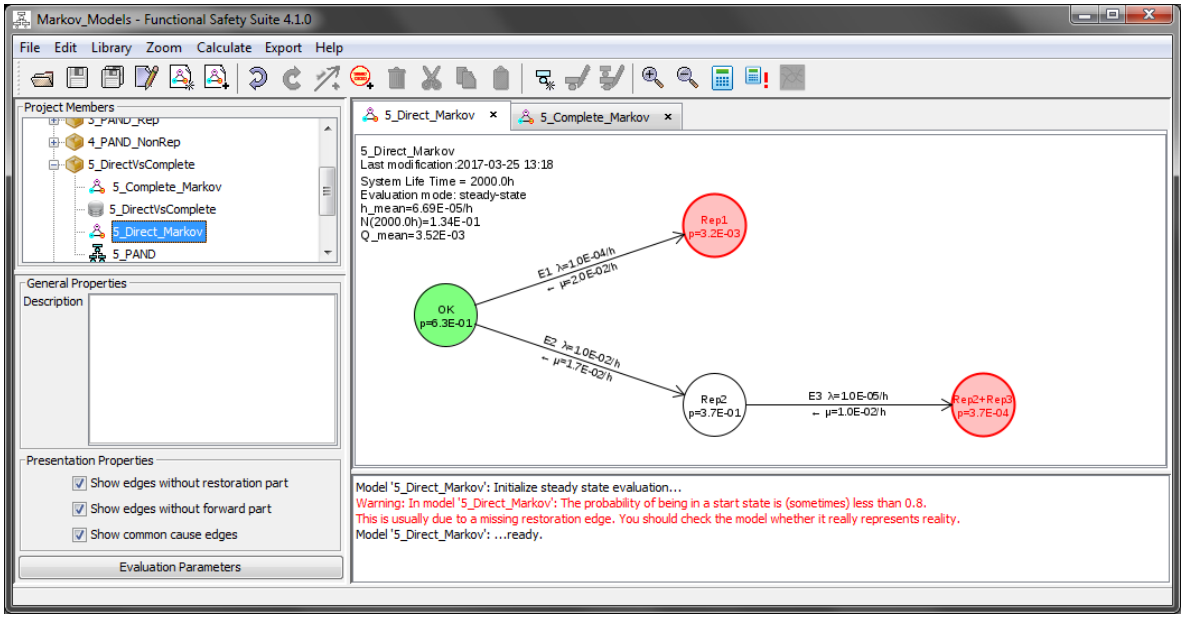


Figure 54: *The Markov model properties panel*

9.2.1 General Properties

Description:

A user defined description of the *Markov model*.

9.2.2 Presentation Properties

Note that in case the presentation related features don't fulfill your needs, you can export all graphics in SVG format for further processing by vector graphics tools.

Show Edges without restoration Part:

In order to get a better overview of restorations, you can disable the display of edges without restoration. Evaluation is not affected.

Show Edges without forward Part:

In order to get a better overview of forward edges, you can disable the display of edges only describing a restoration. Evaluation is not affected.

Show Common Cause Edges:

In order to get a basic view of the forward edges, you can disable the display of common cause edges. Evaluation is not affected.

9.3 The Edge Properties Panel

An *edge* models a transition from a *source state* to a *target state* and optionally vice verse (restoration). The target state is the *state* the *edge* points to. Each *edge* of a *Markov model* consists of the reference to the *generic basic event* and several *modifiers*.

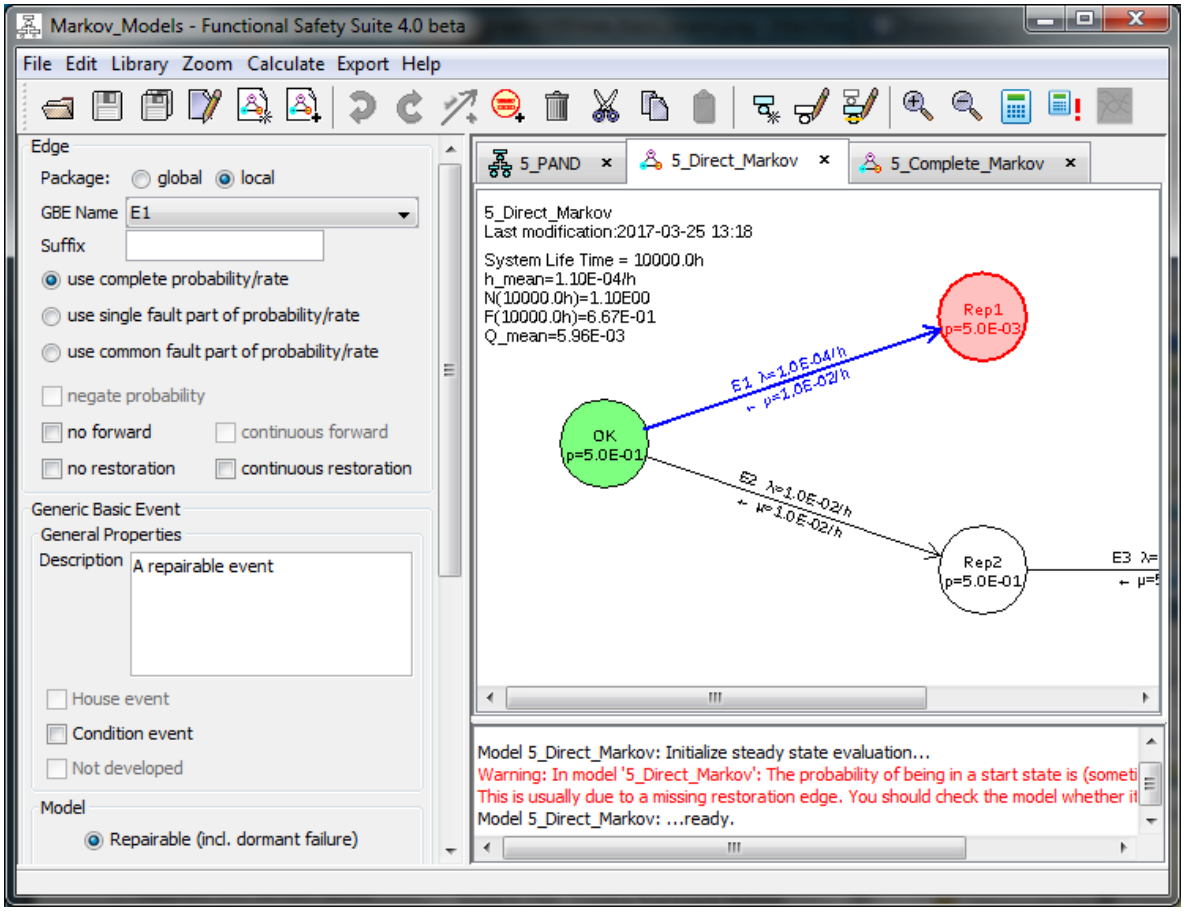


Figure 55: *The edge properties panel*

The parameters in the 'Edge' section belong to the specific *edge*, which is part of the *Markov model*, and thus are stored in the Markov model file (extension `.mdg`).

The parameters in the 'Generic Basic Event' section belong to the *generic basic event* as selected by the field 'GBE name', and thus are stored in the *library*.

Remember: Changing parameters in the 'Generic Basic Event' section will change the properties of all other *basic events* referring to the same *generic basic event* too.

9.3.1 Edge Properties

9.3.1.1 Package

Select whether the *generic basic event* is in the *library* of the *global package* or of the *local package*.

9.3.1.2 GBE Name

The identifier of the *generic basic event*, also serving as the name of the *edge*. You can select a name (and by this the referred *generic basic event*) out of a list of the *generic basic events* belonging to the selected *package*.

9.3.1.3 Suffix

A user defined identifier of the specific instance or the event described by the *generic basic event*. In contrary to the suffix of *basic events* of *fault trees*, the suffix of an *edge* has no effect on the evaluation. It is only for easier creation and better readability of the *Markov model*.

9.3.1.4 Selection of partial Rates or Probabilities

Whereas in *fault trees* the common cause factors are considered completely internally during evaluation, in *Markov models* a common cause factor results in multiple *edges* with different occurrence rate. In order not to require several *generic basic events* for the same failure — one for the single failure, one for the common cause part — you can select which part of the occurrence rate to use in each *edge*.

Also for conditions the non-negated probability can be split into the single cause probability and the common cause probability. Of course the negation of a condition is always the complete (negated) condition probability $\neg p = 1 - p_{\text{complete}} = 1 - (p_{\text{single}} + p_{\text{common}})$.

9.3.1.5 Negate Probability Checkbox

For virtual *states* the sum of probabilities of the leaving *edges* must be 1. Typically one leaving *edge* models the unavailability of an element or sub-system, the second leaving *edge* the availability *A(t)=1-Q(t)* of an element. The unavailability is often modeled by an appropriate *generic basic event* or a link to another model, and not given immediately. In that case the *edge* for the availability shall be set to the same *generic basic event*, but with the 'negate probability' checkbox set. By this the value *A(t)=1-Q(t)* will be assigned to it. An edge with negated probability is displayed in dark green color.

9.3.1.6 *No forward* Checkbox

Depending on the maintenance and repair strategy, the detection and repair of a fault modeled by an *edge* doesn't necessarily lead back to the *source state* of the *edge*. This is the case e. g. if a larger unit with multiple potential faults is exchanged if at least one error is detected, or in case of common cause failures. Also when using *instantaneous transitions*, the restoration

path is different from the forward path. In these cases the restoration must be modeled by a separate *edge*. However in order not to need a separate *generic basic event* only for the restoration, this separate *restoration edge* can refer to the same *generic basic event* as the *edge* representing the failure(s), as long as all failures are restored at the same time. In that case set the *no forward* flag in order to indicate to the program, that only the restoration part of the *generic basic event* shall be used for this *edge*.

However the direction of the *edge* is still defined by the *generic basic event*, thus the *edge* must point towards the failure's *target state*. See figure 56: All states will directly being restored to the OK state – either after check every 1000 h (restoration part of edges "X" or "Y" or separate edges "Rest_1000h") or immediately when the hazard occurs (edge "Rest_imm"). The restoration edges "Rest_1000h" and "Rest_imm" are modeled by *generic basic events* of type *Repairable*, just as "X" and "Y". Since their forward part is not used, the failure rates don't matter.

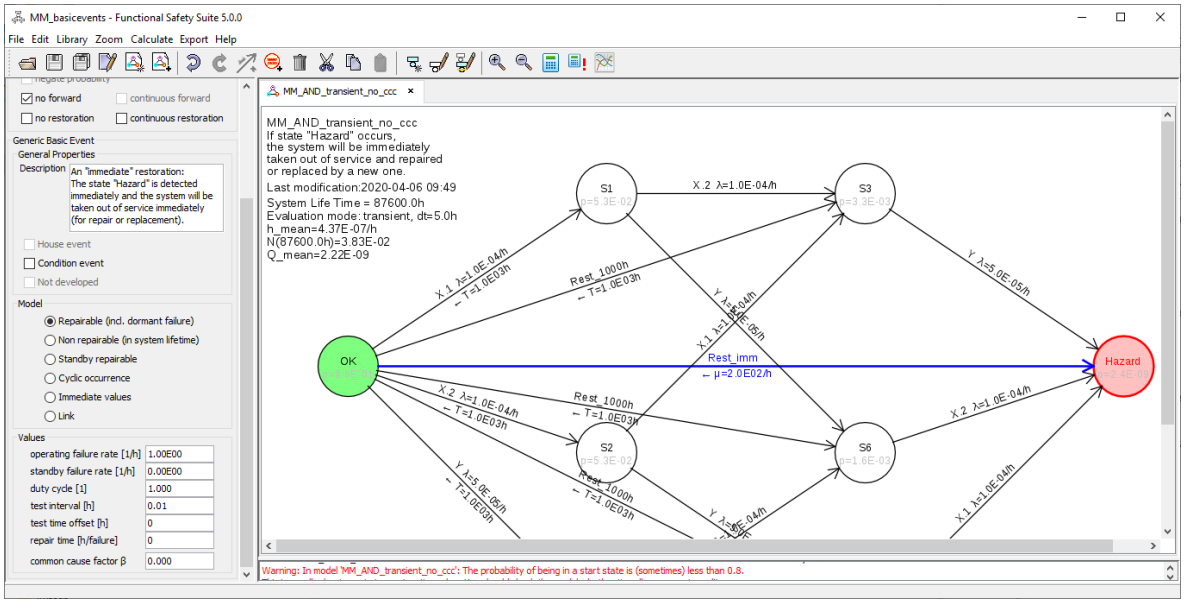


Figure 56: *A pure restoration edge*

9.3.1.7 No Restoration Checkbox

This is just the opposite of the *No forward* flag, see there for explanation.

9.3.1.8 Continuous forward Checkbox

This option makes sense for *edges* of *generic basic event* type *Cyclic* only. There must be at most one cyclic *edge* leaving any *state*, because in case of multiple cyclic *edges*, ambiguity could appear during evaluation. If there is more than one cyclic *edge* leaving a *state*, select this checkbox for those *edges* that shall be treated as normal transitions using their equivalent constant transition rate, see section 4.3.4.

9.3.1.9 Continuous Restoration Checkbox

There must be at most one cyclic restoration leaving any *state*, because in case of multiple cyclic restorations, ambiguity could appear during evaluation. If there is more than one cyclic restoration of a *state*, select this checkbox for those restorations that shall be treated as normal transitions using their equivalent constant restoration rate.

9.3.2 Generic Basic Event – General Properties

9.3.2.1 Description

A user defined description of the *generic basic event* and therefore identical for all *basic events* referring to this *generic basic event*.

9.3.2.2 House Event, Condition Event, Not developed

The *house event* and *not developed* modifiers have no effect in *Markov models*. They are for information only and cannot be changed here.

The effect and usage of the *condition event* modifier in *Markov models* is explained in sections 9.1.3.1 and 9.5. Also see section 4.2.1.

9.3.3 Generic Basic Event – Model

The probabilistic model of the *generic basic event*. See section 4.3 for details.

9.3.4 Generic Basic Event – Values

The values needed by the model of the *generic basic event*. See section 4.3 for details.

9.4 The State Properties Panel

A *state* of a *Markov model* refers to a state that a system can enter physically, e. g. due to a failure, a maintenance action, an action of the operator, the mission profile, etc.

A *state* can be target of multiple *edges* and source of multiple *edges*. Two *states* can be connected by one *edge* only (this *edge* may have a forward part and a restoration part).

All properties of the *state* are stored in the Markov model file (extension `.mdg`).

9.4.1 General Properties

Name:

A user defined identifier of the *state*. Every *state* must have a different name, since the name is used to determine the sources and targets of *edges*.

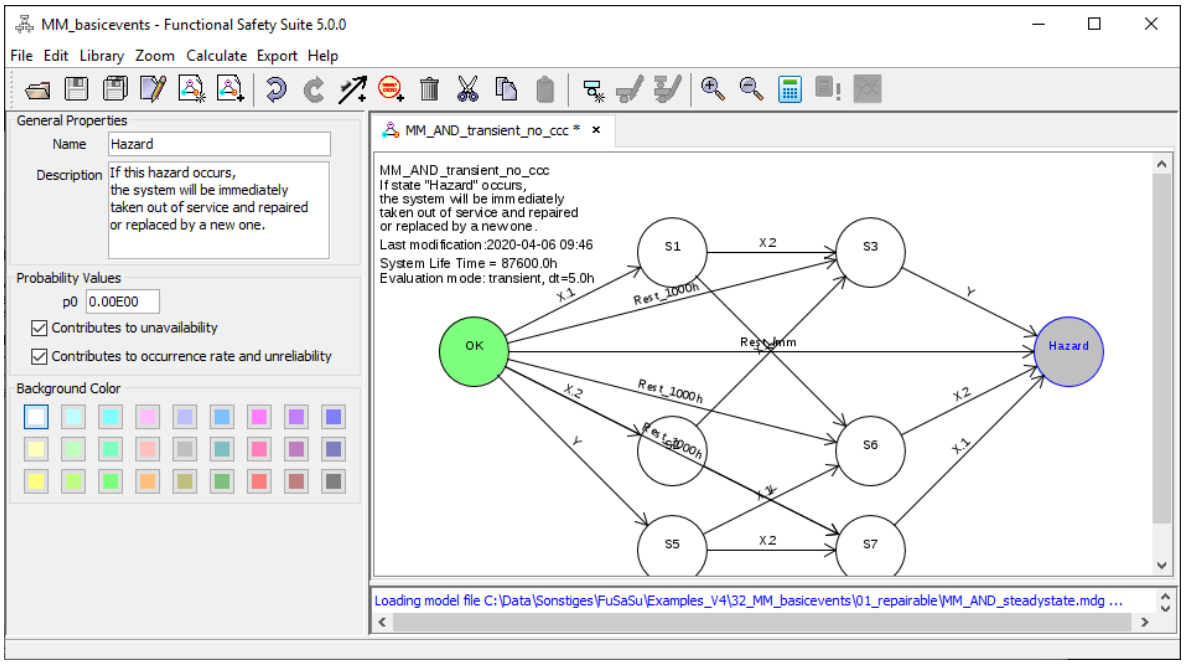


Figure 57: *The state properties panel*

Description:

A user defined description of the *state*.

9.4.2 Probability Values

9.4.2.1 Start Probability p0

States can be assigned a start probability p_0=$p(t=0)$, except of *virtual states*. For states being source or target of *edges*, the start probability is only used in transient evaluation. The sum of all start probabilities of the *Markov model* must be 1.

States with fix Probability

States with no *edges* keep their start probability p_0 forever. By this, it is possible to define states with a fix probability, e. g. to model a constant basic unavailability.

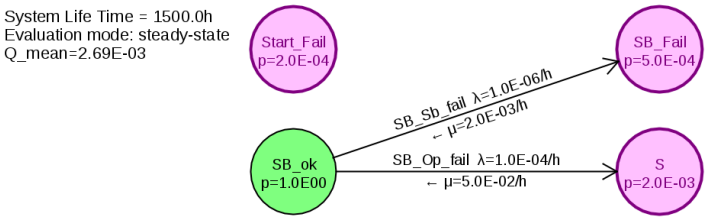


Figure 58: *A state with a fixed probability*

9.4.2.2 Contributes to Unavailability

Each state can contribute to the unavailability *Q(t)* of the element modeled by the model. *States* contributing to the unavailability are marked with a violet circle. *States* that also contribute to the occurrence rate are marked with a red circle, see below.

9.4.2.3 Contributes to Occurrence Rate and Unreliability

Each state can contribute to the occurrence rate *h(t)* and unreliability *F(t)* of the top event modeled by the *Markov model*. *States* contributing to the occurrence rate and unreliability are marked with a red circle. They also contribute to the unavailability, since an element in this state cannot perform any function anymore.

9.4.3 Background Color

The background color can be selected separately for each *state*.

9.5 Evaluation of Markov models

The value of interest for each safety function is either

- the mean unavailability on demand $\overline{Q}$ (PFD),
- the mean occurrence rate $\overline{h}$ (PFH),
- or the probability of failure *F(T)* after system lifetime (or mission time) T.

The value of interest and several parameters related to quantitative evaluation of *Markov models* are set in the *Markov model evaluation properties* dialog, see below.

To calculate these values for a given *Markov model*, select **Calculate – Calculate Model Values**. The mean unavailability $\overline{Q}$ and mean occurrence rate $\overline{h}$ can be calculated by steady-state or transient evaluation, the unreliability *F(t)* can only be determined by transient evaluation. Some modeling features are only available for transient evaluation, allowing a more realistic description of some systems. As for *fault trees*, a transient evaluation is more precise but takes much more computing time.

If the *Markov model* refers to other models by *edges* of type *link*, these models are evaluated before. Circular references are recognized and indicated in the message window before the evaluation starts.

The values of the *event* modeled by the *Markov model* are displayed in the upper left corner in the graphics tab. The probability of each state is displayed in each state's symbol, depending on the method of evaluation (steady-state or transient), see below.

If a parameter of a *state* or an *edge* is changed, all values that might be affected by this change are automatically marked as invalid and not displayed anymore.

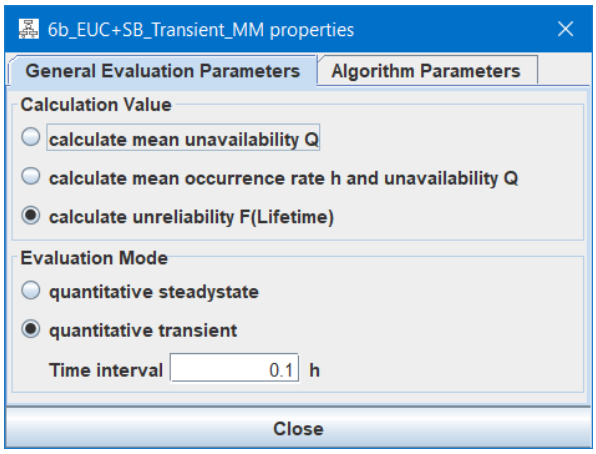


Figure 59: *The Markov model's general evaluation parameters panel*

9.5.1 Evaluation Parameters – General Evaluation Parameters

9.5.1.1 Calculation Value

Select which value(s) to calculate. In contrary to *fault trees*, there aren't different algorithms for the different values. Therefore, in fact in steady-state evaluation the mean unavailability $\overline{Q}$ and the mean occurrence rate $\overline{h}$ will be calculated independent of your selection (calculation of the unreliability *F(T)* is not possible in steady-state mode). In transient evaluation mode, *h(t)*, *f(t)*, *w(t)*, *F(t)*, *Q(t)* and *N(t)* are calculated and available for display in the *chart frame*. Depending on the selected value, $\overline{Q}$, $\overline{Q}$ and $\overline{h}$ and the unreliability *F(T)* is displayed in the header.

The unavailability Q is the sum of the probabilities of all states marked as contributing to the unavailability:

$$Q_{\text{sys}} = \sum_{i=0}^{n} p_{\text{final,Q}_i} \tag{39}$$

The occurrence density of a *state* j w_{j} is the sum of the transition rates of all *edges* pointing to it, multiplied by the actual probability of the *source state* of each *edge*

$$w_j = \sum_{i=0}^{n} h_{\text{in}_i} \cdot p_{\text{source}_i} \tag{40}$$

If the *edge* represents an *instantaneous transition*, the transition rate of this *edge* is the sum of the transition rates to the *source state* of this *edge*, (i. e. the *virtual state*) multiplied by the probability of the *instantaneous transition*.

The occurrence density of the event modeled by the *Markov model* w_{sys} is the sum of the occurrence densities of all states marked as contributing to the occurrence rate of the model (see section 9.4.2):

$$w_{\text{sys}} = \sum_{j=0}^{n} w_j \tag{41}$$

Note: Typically there will be no restorations leading to a state contributing to the occurrence rate of the model, but if there were any, they would not be considered in calculation of w_{sys}, since a restoration obviously doesn't contribute to the occurrence of an undesired state of the system.

For calculus of h_{sys} (PFH) it is presumed, that the system is always in an up-state when the (last, dangerous) failure occurs. The higher the probability of the final state(s) (indicating the unavailability of the function), the lower the occurrence density of the event modeled by the *Markov model*. Therefore, if *divide density by availability* is selected in the *project properties dialog*(tab *Markov Models*), the occurrence rate (usually a failure rate) of the *Markov model* is calculated by

$$h_{sys} = \frac{w_{sys}}{A_{sys}} = \frac{w_{sys}}{1 - Q_{sys}} \tag{42}$$

If *divide density by start state probability* is selected in the *project properties dialog* the occurrence rate is calculated by

$$h_{sys} = \frac{w_{sys}}{p_{OK}} \tag{43}$$

which is more conservative.

Note that the failure rate h can be calculated only for $Q \not\approx 1$, since the difference cannot be calculated with sufficient accuracy for values close to 1. If this happens, h is set to 'NaN' and not displayed, thus it is ensured that only values with sufficient numeric accuracy are displayed. For (correctly built and modeled) safety systems this is never an issue, since $Q \ll 1$ is always fulfilled. In transient evaluation mode, this must be fulfilled for each time step, i. e. not only the final step or an average value.

9.5.1.2 Evaluation Mode

Select whether the *Markov model* shall be evaluated in steady-state mode or in transient (time-variant) mode. In case of transient evaluation, the time interval must be set as well.

Quantitative Steady-State Evaluation

A steady-state analysis is appropriate for all systems that are supposed to operate for many years, with certain test intervals and optionally some down-times for maintenance and repairs. Several parts of the system might be replaced or repaired during the system's lifetime. In case that all failures are detected in adequate time (either by continuous diagnosis, by periodic tests or by malfunction of the system), both the failure rate h (PFH) and the unavailability Q (PFD) of the system don't depend on its actual age, but will reach some pseudo-stationary state where both values will oscillate around a mean value. The frequency of this oscillation is equal to the longest detection interval or a multiple of it. This is even correct in case the failure rates of some particular components depend on their specific age, if the lifetimes of these components are shorter than the system life time. The value of interest for each safety function performed by such a system is either the mean unavailability on demand $\overline{Q}$ (PFD), or the mean occurrence rate $\overline{h}$ (PFH). The related standard is mainly [EN 61508] and the

derived standards. Examples for those systems are machines, cars, trains, air-crafts, chemical plants, power plants, etc. and their control systems.

In steady-state evaluation cyclic *edges* due to periodic tests of *generic basic events* of type *repairable* will be replaced by restoration rates μ, cyclic *edges* due to periodically occurring events (*edges* referring to *generic basic events* of type *cyclic*) will be replaced by forward transition rates λ. Note that these conversions do not correctly reflect reality, but are only approximations.

In principle a *Markov model* is evaluated for a steady-state by solving a linear equation system, whose variables are the state probabilities and whose coefficients are the transition rates. Probabilities of instantaneous edges are multiplied with the transition rate of the preceding continuous edge.

The steady-state probability of each state is displayed inside the state's symbol. Based on the state probabilities in steady-state, the mean unavailability $\overline{Q}$ and the mean occurrence rate $\overline{h}$ are calculated, see section 9.5 above.

A steady-state analysis is very fast, because all values have to be calculated only once (compared to time-variant analysis, where all values have to be calculated many times, see below).

Quantitative transient Evaluation

A transient analysis is mandatory, if the 'mission failure probability', namely the system's unreliability $\mathrm{F}(0, T_{\mathrm{mission}})$, is the value of interest. This is typical for non-restorable systems, that are supposed to perform their function in a pre-defined way for a pre-defined lifetime (e. g. a certain mission), such as a rocket or spacecraft. These systems are characterized by final states without restoration paths.

A transient evaluation is also necessary for time-variant systems, that shall be examined in detail. The *Markov models* describing those systems might include *edges* of type *cyclic*, and the restoration might need to be modeled by discrete restoration times. Also if time variant occurrence rates are used, e. g. due to *links* to other models or due to *generic basic events* of type *non-restorable* with increasing or decreasing failure rates, a transient analysis is recommended. In general, a transient (or time-variant) analysis produces more precise results, but is much slower.

From the mathematical point of view, the model is in principle a linear differential equation system. Thus for transient evaluation, a differential equation system must be integrated. For most practical problems the rates differ for multiple orders of magnitude, thus the differential equation system is typically stiff. Therefore an implicit integration algorithm is used. If rates are constant over the lifetime, the differential equation system is time in-variant and therefore doesn't need to be created in each step. In any case the system is linear, so the Jacobian matrix is directly given. In case of non-constant failure rates, the Jacobian must be calculated for each step, which is quite a time-consuming operation. You should avoid time variant failure rates, therefore, but it makes no difference whether you've got one or many.

The differential equation system only describes the continuous transitions. Due to the extensions of Functional Safety Suite, also instantaneous transitions and cyclic (periodic) transitions occur. Therefore in each step the following is done:

1. calculate all *generic basic events*, including linked models for the next step
2. create the differential equation system for the next step according to current continuous transition rates *h(t)* and μ*(t)* and current instantaneous transition probabilities and integrate it for one step
3. determine the changes of all *states* due to cyclic (periodic) forward transitions
4. determine the changes of all *states* due to periodic restorations

The unreliability at a given time t is typically just the sum of the probabilities of the final states marked as contributing to the unreliability:

$$F_{\text{sys}}(t) = \sum_{i=0}^{n} p_{\text{final,F}_i}(t) \tag{44}$$

If there is at least one restoration from any final state contributing to the unreliability, the unreliability is automatically calculated via the system occurrence rate h_{sys}:

$$F(t) = 1 - \mathrm{e}^{-\int_0^t h(\tau)\,d\tau} \tag{45}$$

The mean values for unavailability $\overline{Q}$ is calculated by

$$\overline{Q} = \frac{1}{T}\int_0^T Q(t)\,dt \tag{46}$$

The state's probabilities after the last calculation step $p(T_{\text{mission}})$ are displayed in each state's symbol. Note that these are displayed for better understanding only, but are in general not equivalent to the system's safety values. Therefore they are displayed in light gray.

Time interval: The step size for transient evaluation in hours. A smaller step size means more steps for the given system lifetime and thus takes more time in calculation. This might be an issue in case of large systems. The step size must be less than a 10th of the smallest periodic (cyclic) event you want to evaluate for discrete times. Time constants less than 10 times the step size are handled as rates. However step size should be even smaller in order to reduce the computational errors.

Example: Given a redundant control system, consisting of two similar channels 1 and 2. Each channel has two failure modes A and B, optionally with some common cause factor. The proof test intervals are 24 h and 168 h. Its Markov model is presented in figure 60.

With a step time of 0.5 h, the evaluation considers the tests at multiples of 24 hours and 168 hours, resulting in a *h(t)* and *Q(t)* displayed in figure 61.

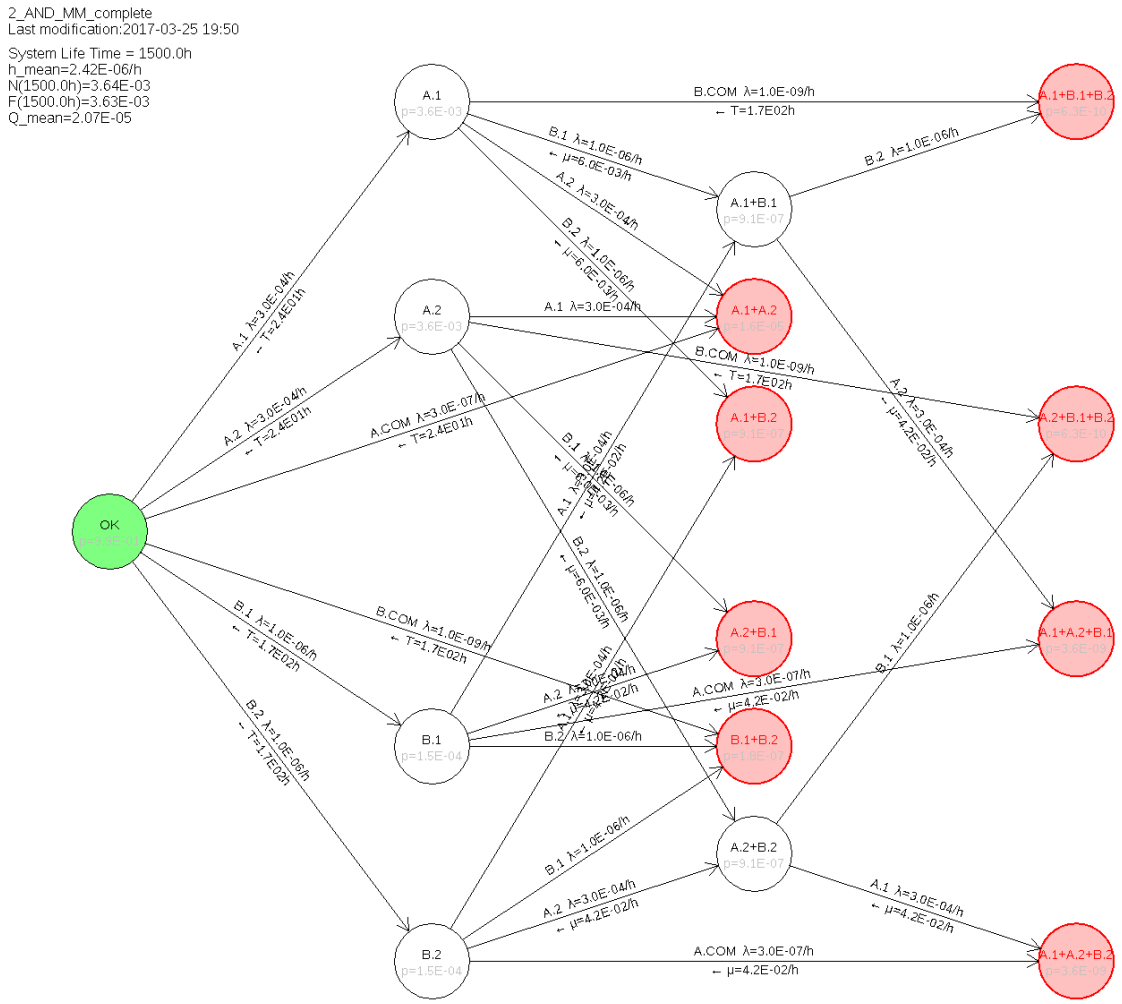


Figure 60: *Markov model of a 1-out-of-2 system*

With a step time of 20 h, the proof test intervals are less than 10 times the step time, so that a continuous restoration rate of μ is considered instead of a cyclic transition. The resulting $h(t)$ and $Q(t)$ is displayed in figure 62.

You might wonder why the occurrence rate $h(t)$ is never zero, even in figure 61. This is due to the common cause factor between both channels, modeled by direct edges from state "OK" to some failure states. If you set the common cause factor of both *generic basic events* to zero, the result will look completely different, compare figure 63.

9.5.2 Evaluation Parameters – Algorithm Parameters

9.5.2.1 Pre-Processing Mode

Starting with version 3.2 of Functional Safety Suite, the common cause factors between *edges* of *Markov models* can be handled automatically for most *Markov models*. In addition, most

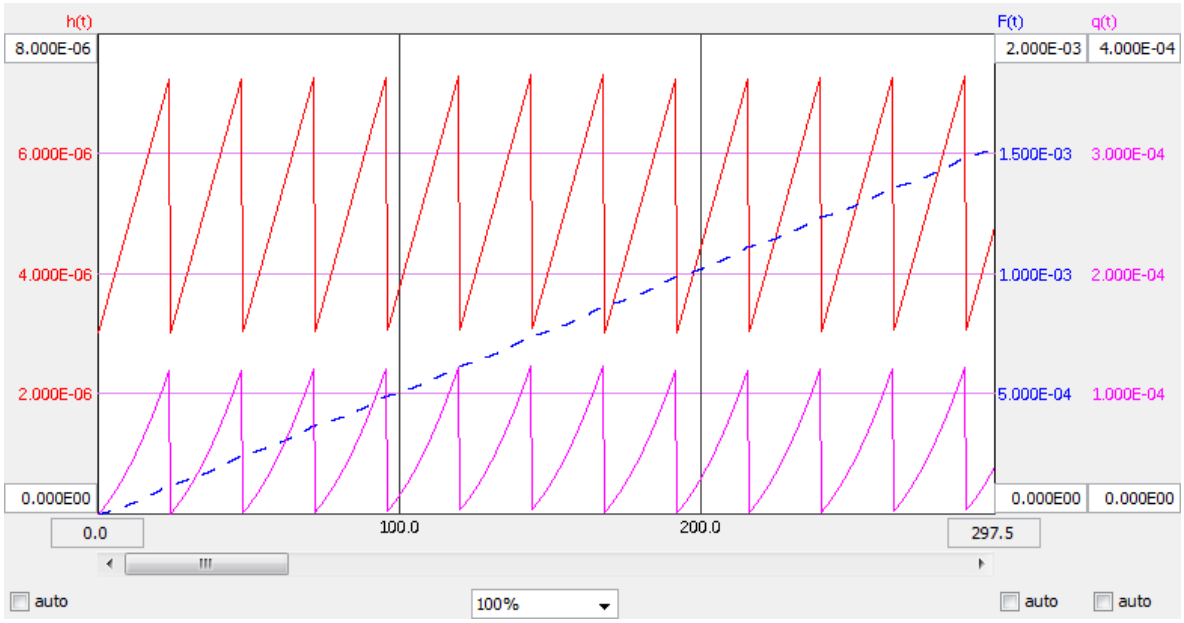


Figure 61: *textslh(t) and Q(t) for a 1-out-of-2 system with small step size*

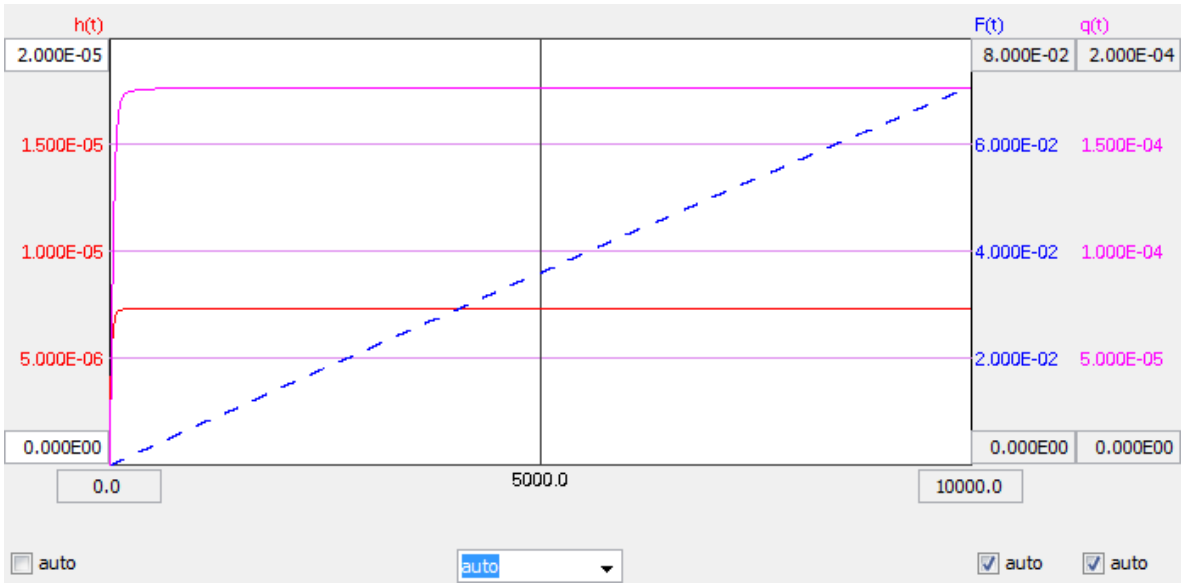


Figure 62: *h(t) and Q(t) for a 1-out-of-2 system with large step size*

Markov models can be completed automatically. Automatic completion always includes automatic creation of common cause chains. In both cases, the *Markov model* finally used for evaluation will be created internally.

To understand the difference between “direct” chains and “complete” chains, have a look at the *fault tree* shown in figure 65.

The *Markov model* representing the minimal cut-sets of this *fault tree* is presented in figure 66.

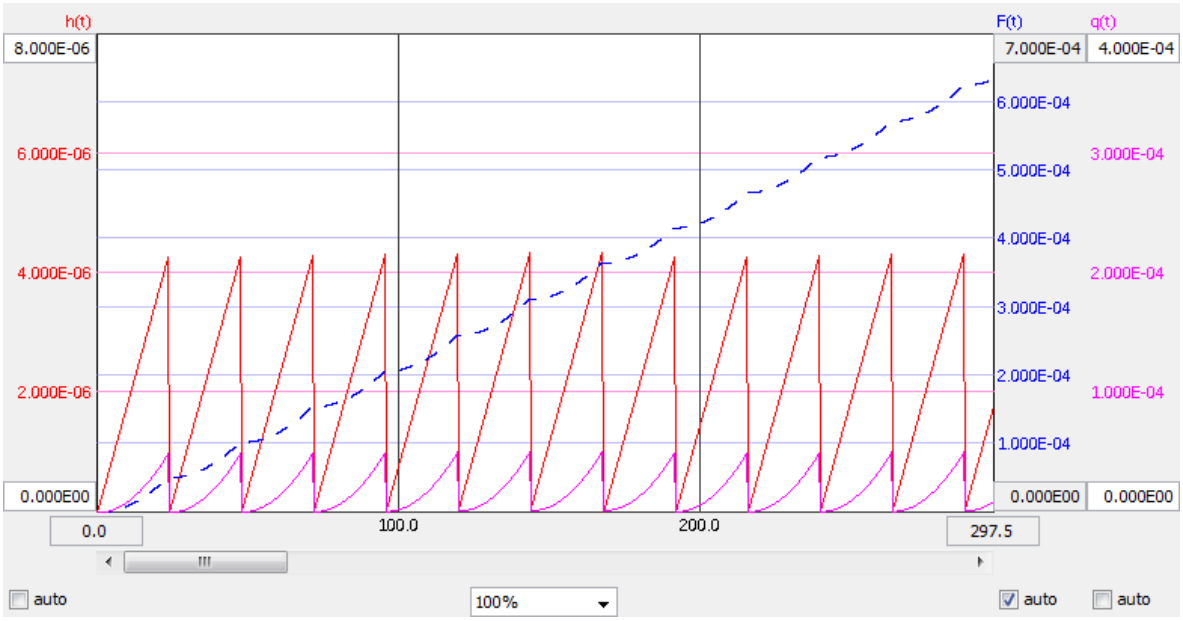


Figure 63: *h(t) and Q(t) for a 1-out-of-2 system with small step size and small β*

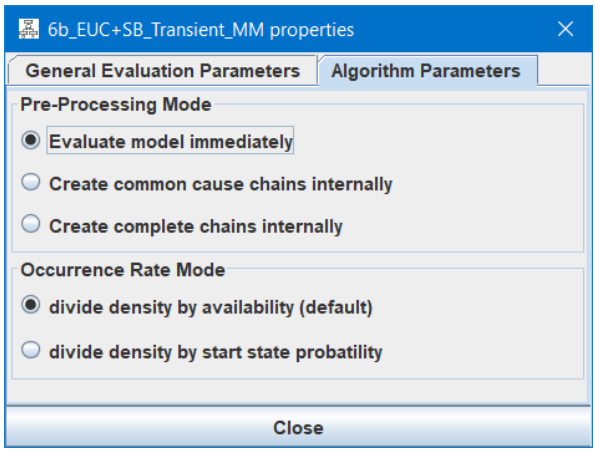


Figure 64: *The Markov model's algorithm parameters panel*

In fact, event "Rep1" can occur also in states "Rep2" and "Rep3". The complete *Markov model* also considering these degrees of freedom is shown in figure 67. The completed *Markov model* also considers, that the system's state can "jump" from one (yet incomplete) chain to another chain, until a final state is reached — even multiple times. This option includes the internal creation of common cause chains.

However in most practical systems, the difference is negligible. Here in fact the values have been selected explicitly in order to show a difference.

Note: If the *Markov model* is pre-processed, no state probabilities will be shown, since only the final model is evaluated but not the model displayed in the graphics tab.

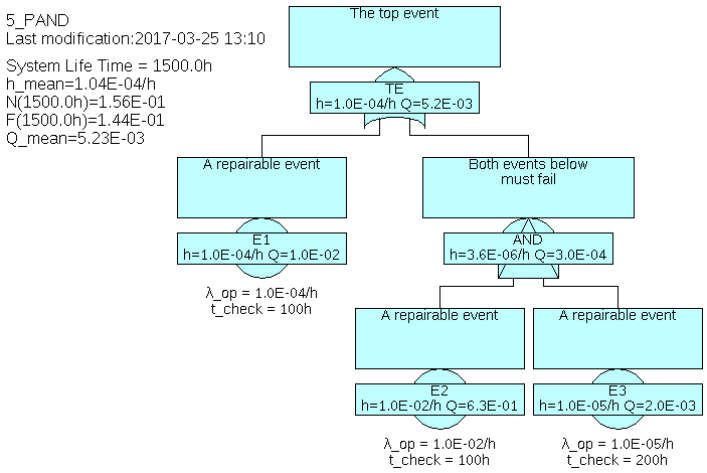


Figure 65: *A simple fault tree.*

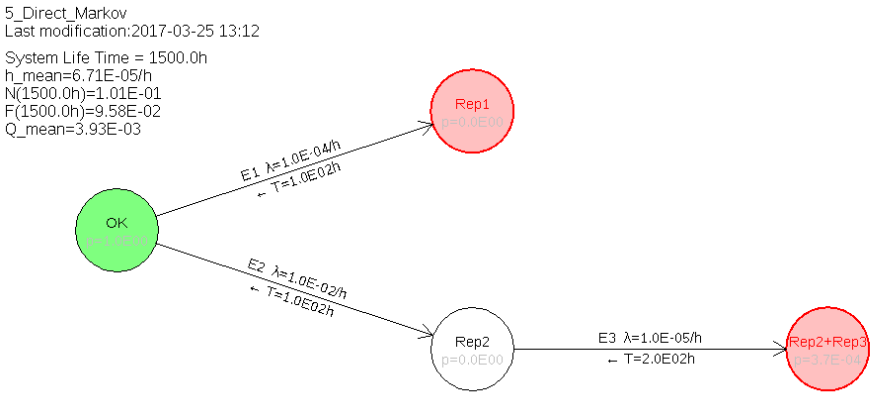


Figure 66: *The corresponding Markov model, considering only direct chains.*

The final *Markov model* used for evaluation can be exported by **Export – Export Final Markov Model**. It is recommended to check the correct pre-processing and apply manual corrections or adaptations if necessary.

9.5.2.2 Occurrence Rate Mode

A Markov model directly models the (unconditional) occurrence density w. The occurrence rate h can be derived from w in two ways:

- By dividing w by the probability not being in a final state or
- by dividing w by the probability being in a start state.

For systems with good detection and repair rates, there will be no significant difference, however for systems usually not being in a start state, the first alternative can be too optimistic.

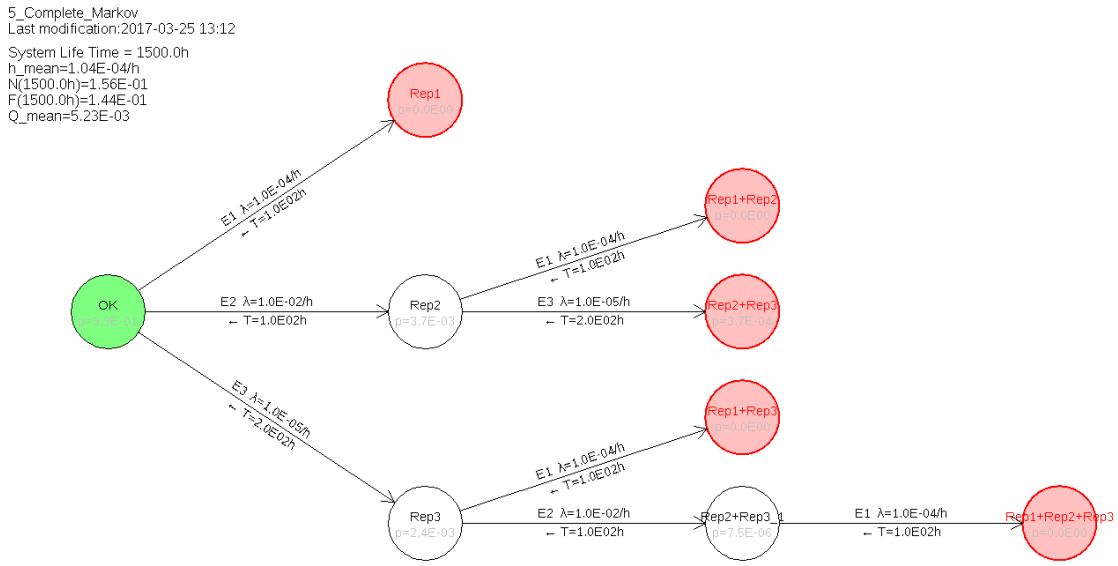


Figure 67: *The corresponding Markov model, considering only direct chains.*

9.6 Editing of Markov Models

Create a *Markov model* by **File – New Markov Diagram**.

You can add *states* by **Edit – Add State** and clicking the mouse at the position you want to set the *state*. A unique name will be automatically assigned to the new *state*. At each grid position, only one *state* can be placed.

In order to add an *edge*, select the *source state*, then select **Edit – Add Edge**, then click on the *target state*. An *edge* referring to the last *generic basic event* in the *library* will be created. You can select any other existing *generic basic event* by selecting it via its name and *package* in the *Edge Properties Panel*, see section 9.3.

You can also cut (or copy) and paste *states* and *edges*: Select the *state* by **Edit – Cut** (or 'Ctrl+X') or **Edit – Copy** (or 'Ctrl+C') , then press **Edit – Paste** (or 'Ctrl+V'), finally click on the position where you want to paste it. For an *edge*, cut or copy it, then select the *source state*, press **Edit – Paste** (or 'Ctrl+V'), finally click on the *target state*.

States are moved by 'Shift+Cursor', connected *edges* will stay connected. If a *state* is deleted, all connected *edges* will be deleted too (the *generic basic event* will not be deleted of course).

Multiple *states* and *edges* can be selected with the mouse, either by clicking the left mouse key together with 'Shift', or by pressing the left mouse key and pulling the selection rectangle around several *states*. All *edges* between selected *states* will be selected also. A selection can be moved by holding the left mouse key while dragging, or you can cut or copy and paste it as single events. All *states* pasted into the same or another *Markov model* will get a unique name, consisting of the original one, appended by a number if necessary to become unique.

Changing properties of *states* or *edges* is done in the properties window. The only exception is the change of the name of an *edge* (⇔*generic basic event*), for which a special command **Library – Rename Generic Basic Event** is foreseen. The properties of the *generic basic event* referred by an *edge* can be edited in the library view as well, see section 4.1.

A *Markov model* that has not been saved after the latest modification is marked with an asterisk '*' in its title.

10 Complex Components

10.1 Introduction

The *complex component* model has been developed in order to support the calculation of basic parameters for components characterized by multiple, time-variant failure modes (the *component events*). Thus the "bath tub" curve of failure rates can be modeled. Each failure mode can be defined to be either safe or dangerous.

In the previous sections related to fault trees or Markov models, the mean value $\overline{h}$ has been calculated as the arithmetic mean value $\overline{h} = h_{\text{avg}} = \frac{1}{T}\int_0^T h(t)dt$ for the given system lifetime t. This arithmetic mean value can be used only if the unavailability of the system is small. For all practical system for which the mean failure rate (or hazard rate) $\overline{h}$ is interesting at all, this condition is fulfilled, because these systems will be restored (repaired or replaced) if they are defect. Even if the system is not continuously used, and its failure might not be immediately detected, therefore, the arithmetic mean value is a good approximation of the mean hazard frequency over system lifetime. The arithmetic mean value can also be used if the system lifetime t is short compared to the mean lifetime of each component, i. e. if the MTTF of each component is greater than the system lifetime.

In order to understand the necessity of the *complex component* model, have a look at figure 68. This figure shows typical variations of failure rate h(t), failure density *f(t)* and unreliability

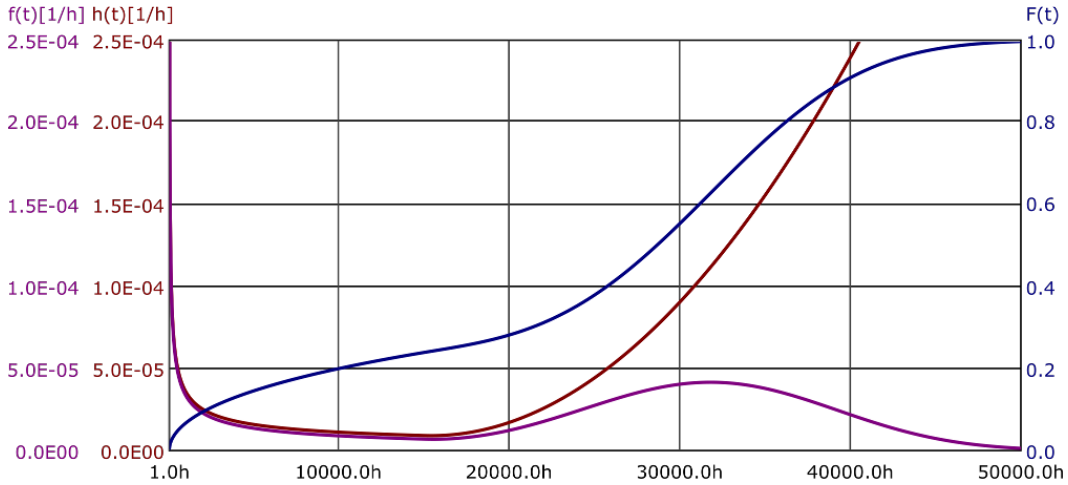


Figure 68: *A component with several failure modes.*

F(t) over time. If you have such components in your system, you'll have to determine a mean failure rate $\lambda = \overline{h}$ that you can use in a *fault tree* or *Markov model* in steady-state evaluation, or a time dependent failure rate *h(t)* that can be used for transient evaluation. If the component is very "good", i. e. if it will probably not fail during system lifetime, you can use the arithmetic mean failure rate $\overline{h}$ for use in other models. For transient evaluation, you can directly use the time dependent failure rate *h(t)* in that case.

If the component is likely to fail during system lifetime, its mean failure rate $\overline{h}$ has to be

calculated via its MTTF [10]. For all failure distributions, the MTTF is given by

$$\mathrm{MTTF} = \int_0^\infty t \cdot f(t)\,dt \tag{47}$$

With the general relationship between any failure density and failure rate function

$$f(t) = h(t) \cdot R(t) = h(t) \cdot \mathrm{e}^{-\int_0^t h(\tau)\,d\tau} \tag{48}$$

the MTTF can be calculated based on the failure rate function *h(t)* by

$$\mathrm{MTTF} = \int_0^\infty t \cdot h(t) \cdot \mathrm{e}^{-\int_0^t h(\tau)\,d\tau}\,dt \tag{49}$$

This is the complete MTTF or natural MTTF, that can be found by experiment if you operate many of these components until they fail. The arithmetic mean value of all times until failure will be the MTTF. For the component shown in figure 68, the natural MTTF is about 25 000 h. If you use this component in a system that you intend to use for 200 000 h, you'll need 7 (maybe 8) spare components per system throughout its lifetime. The mean failure rate is $\overline{h} = \frac{1}{\mathrm{MTTF}} \approx 4\mathrm{E}{-5}\,/\mathrm{h}$. [11]

If a failure of this component is associated with a significant damage of other parts of the system (as for example the rupture of the timing belt of a Diesel engine), or if the failure is even safety critical, and the failure rate function has a considerable increasing part (similar to that shown in figure 68), preventive change makes sense. In case of preventive change at time T=T_{change}, the incomplete MTTF(T) or effective MTTF(T) is given by

$$\begin{aligned}\mathrm{MTTF}(T) &= \frac{\int_0^T t \cdot f(t)\,dt + T \cdot R(T)}{F(T)} = \frac{\int_0^T t \cdot f(t)\,dt + T \cdot \big(1 - F(T)\big)}{F(T)} = \frac{\int_0^T t \cdot f(t)\,dt + T}{F(T)} - T \\ &= \frac{\int_0^T t \cdot h(t) \cdot \mathrm{e}^{-\int_0^t h(\tau)\,d\tau}\,dt + T}{1 - \mathrm{e}^{-\int_0^T h(t)\,dt}} - T\end{aligned} \tag{50}$$

For $T \to \infty$ the incomplete MTTF(T) migrates to the complete MTTF.

Finally imagine some kind of a normally closed electromagnetic valve with the following main failure modes:

- fail to open due to a coil failure

[10] If you try to calculate a mean failure rate via arithmetic mean value, you'll get a too high value — if you don't fail due to numeric calculation issues.

[11] The arithmetic mean value is $h_{\mathrm{avg}}(200\,000\,\mathrm{h}) \approx 4\mathrm{E}{-3}\,/\mathrm{h}$, i. e. 100 times too high.

- fail to open due to blockage of the gasket
- fail to open due to corrosion
- fail to close due to a broken spring
- fail to close due to blockage of the gasket
- fail to close due to mechanical obstruction
- fail to close due to corrosion

Let's assume that failures to open are safe, failures to close are dangerous in a given application. Let's further assume, that some failures are early failures (production faults), some are late failures (due to wear), some might have a nearly constant failure rate. Obviously you have to distinguish between safe and dangerous failure modes in some way, and thus an overall component failure rate and a dangerous component failure rate $\overline{h_d}$ or $h_{\mathrm{d}}(\mathrm{t})$. Let's finally assume, that the valve is supposed to be replaced in certain intervals — how will this action affect the overall dangerous failure rate, and which is the optimal interval to replace the component?

The effective dangerous MTTF(T) is given by

$$\begin{aligned}\mathrm{MTTF_d}(T) &= \frac{\int_0^T t \cdot f(t)\,dt + T \cdot R(T)}{\int_0^T \varphi_d(t)\,dt} = \frac{\int_0^T t \cdot h(t) \cdot R(t)\,dt + T \cdot R(T)}{\int_0^T h_d(t) \cdot R(t)\,dt} \\ &= \frac{\int_0^T t \cdot h(t) \cdot \mathrm{e}^{-\int_0^t h(\tau)\,d\tau}\,dt + T \cdot \mathrm{e}^{-\int_0^T h(t)\,dt}}{\int_0^T h_d(t) \cdot \mathrm{e}^{-\int_0^t h(\tau)\,d\tau}\,dt}\end{aligned} \tag{51}$$

where $h_{\mathrm{d}}(\mathrm{t}$ denotes the dangerous failure rate and $h(t)$ the overall failure rate. Obviously, this formula cannot be calculated with universal spreadsheet tools or calculators due to the complicated double integrals, that cannot be expressed in closed-form.

In contrary to all other kinds of models, *complex components* make no use of *generic basic events*, since their failure modes are directly stated in the model. Each *component event* (failure mode) can be specified to be safe or dangerous, see section 10.4 below for details.

Summary

The *complex component* model will calculate the following values according to the *component events* and the component properties: [12]

Mean time to failure: The (natural, complete) MTTF is the mean life time of the component. It is relevant if it is short compared to the (intended) system lifetime and the component is not exchanged in certain intervals, but operated until it fails.

[12] In principle, the algorithm is able to calculate these values for any kind of failure distribution function. By default, only Weibull distribution is available, other distributions can be added on demand.

Mean time to dangerous failure: The (natural, complete) $\mathrm{MTTF}_{\mathrm{dang}}$ is relevant if the components overall MTTF is short compared to the (intended) system lifetime and the component is not exchanged in certain intervals, but operated until it fails.

Mean time to dangerous failure with preventive exchange: If the actual component life time t in the application is much shorter than the MTTF, the failure modes with increasing failure rates will occur rarely, thus the reliability and safety of the component tends to be determined by the failure modes with decreasing or constant failure rates. Therefore in that case the $\mathrm{MTTF}_{\mathrm{dang}}$ is calculated for the actual mean component life time t as well. Often t is defined by a preventive exchange interval, so this value is named $\mathrm{MTTF}_{\mathrm{dang,prev}}(\mathrm{T})$. Nevertheless t might also be given by the overall system life time (stated in the *project properties dialog*) or by the change interval or the MTTF of the assembly group that contains the component.

Mean failure rate: The mean failure rate λ is the reciprocal of the MTTF.

Mean dangerous failure rate: The mean dangerous failure rate λ_{dang} is the reciprocal of the $\mathrm{MTTF}_{\mathrm{dang}}$.

Mean dangerous failure rate with preventive exchange: The mean dangerous failure rate for shortened life time $\lambda_{\mathrm{prev,dang}}$ is the reciprocal of the $\mathrm{MTTF}_{\mathrm{dang,prev}}(\mathrm{T})$ (if it exists).

Time to 10% failed components: The time at which 10% of a large number of components have failed either safely or dangerously T(B10). This is equivalent to the time at which the overall reliability $R(\mathrm{T(B10)})=0.9$ or the overall unreliability $F(\mathrm{T(B10)})=0.1$.

Time to 10% dangerously failed components: The time at which 10% of a large number of components have failed dangerously *T(B10d)*. This is equivalent to the time at which the dangerous unreliability $F_{\mathrm{d}}(\mathrm{T(B10d)})=0.1$. Note that this value only exists, if at least 10% of the components can fail dangerously at all, that is not more than 90% of the components already failed safely before.

The *complex component* model can be linked to all other models, such as *fault trees* and *Markov models*. If the *complex component* is referred in other models by links, the mean occurrence rate $\overline{h}$, the mean unavailability $\overline{Q}$, the unreliability *F(T)* or the time variant values *h(t)*, *Q(t)* or *F(t)* are transferred as described in section 2.4 and section 10.3 below.

10.2 The Component Properties Panel

The component properties are presented, if no event is selected (see figure 69).

All properties of the *complex component* are stored in the component file (extension `.cmp`). A *complex component* that has not been saved after the latest modification is marked with an asterisk '*' in its title.

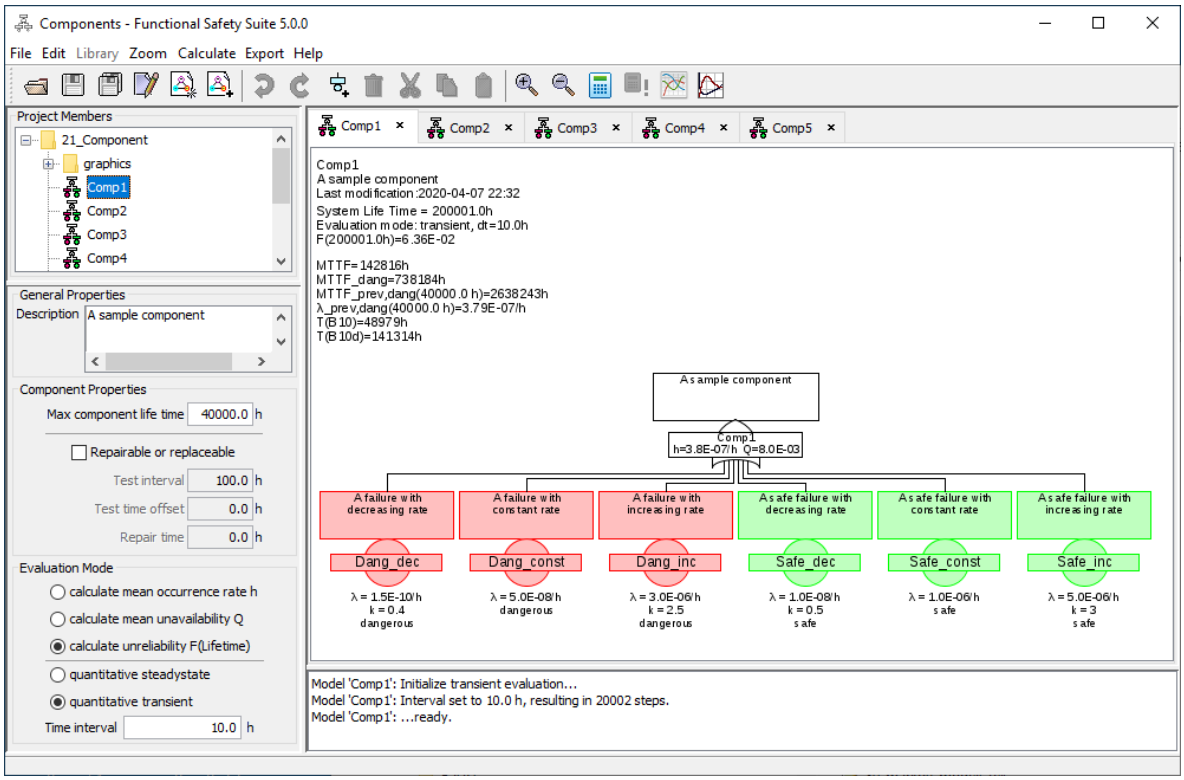


Figure 69: *A component with several failure modes (component events) with time-variant failure rates.*

10.2.1 General Properties

Description:

A user defined description of the *complex component*.

10.2.2 Component Properties

10.2.2.1 Max Component Life Time

If the component life time is defined by preventive exchange in certain intervals or by the life time of the assembly group of which it is part, this can be stated here.

If the stated value is greater than zero, less than the system life time defined in the *project properties dialog*, and less than the complete MTTF of the component, the stated value will be used for the calculation of the effective MTTF and some other values (see below). Otherwise if the complete MTTF is greater than the system life time defined in the *project properties dialog*, the system life time is used for the calculation of these values. If both conditions are false, the MTTF is used for further calculations.

10.2.2.2 Repairable or replaceable

If the component is tested in certain intervals (and replaced if a fault is detected), select this checkbox and set the following values accordingly.

Test Interval: The interval in which the component is tested for hidden ("sleeping") faults.

Test Time Offset: If tests of several components are performed with a defined time difference, this offset (related to system start time) can be stated here. Overall system unavailability will become less if tests are performed at different times. This value is only used in transient evaluation.

Repair Time: The time needed to repair the component in case a fault has been detected in the test.

10.2.3 Evaluation Mode

Note: In this section, whenever $\overline{h}$ or *h(t)* is written, the (dangerous) failure rates handed over to any higher level model are meant.

Select which value is of interest. In fact there is no difference in the algorithm, only some warnings might differ, and the value(s) displayed in the graphics tab.

As for *fault trees* and *Markov models* you can select between steady-state or transient evaluation mode. In case of transient evaluation, the time interval must be set as well.

In case of steady-state evaluation, the mean values $\overline{h}$ or $\overline{Q}$ or the maximum unavailability $Q_{\max}$ or the final unreliability *F(T)* will be handed over to any higher level model (*fault tree* or *Markov model*). In case of transient evaluation, if the higher level model is evaluated in transient mode as well, the values *h(t)*, *Q(t)* or *F(t)* will be handed over to the higher level model.

10.3 Values handed over to higher Level Models

The calculation of unavailability and unreliability depends on the expected event that most probably determines the end of life of the component. In reality, the life of a component will end when one of the following events occurs:

Case 1: The life of the overall system ends (as defined in the *project properties dialog*).

Case 2: The life of the subsystem, which contains the component, ends, or the component is replaced in predefined intervals (defined by the parameter *component life time*, see section 10.2.2.1 above).

Case 3: The component fails safely (so that the system goes to a safe state). The component must be replaced by a new component before the system can be used again.

Case 4: The component fails dangerously. This case has three sub-cases:

Case 4a: The component is part of a safety barrier or a redundancy. Thus its failure doesn't directly lead to an accident. Those components are normally checked in certain intervals. If a fault is detected, the component is replaced by a new one.

Case 4b: The dangerous failure of the component directly leads to an accident (or at least to a hazard, that is immediately detected), but not to the loss of the overall system. The component will be replaced after all and the system will continue operation.

Case 4c: The dangerous failure of the component directly leads to a loss of the system (e. g. a severe accident, that destroys the system).

Depending on the design of the component and the system, either one of the cases might be the by far most probable case, or it might not be clear, which case will usually end the component's life. In Functional Safety Suite the following definition applies:

corresponding to case 1: If the (natural) MTTF is greater than the system lifetime T_{sys} and there is no lifetime limitation set (see section 10.2.2.1 above), it is assumed that the component's life usually ends with the overall system life. Thus the mean (dangerous) occurrence rate is given by

$$\overline{h} = 1/\text{MTTF}_{\text{dang}}(T_{\text{sys}}) \tag{52}$$

The dangerous occurrence rate $h_{\text{dang}}(\text{t})$ is equal to the sum of the time variant failure rates of the dangerous failure modes

$$h(t) = h_{\text{dang}}(t) = \sum_{i=1}^{n_{\text{dang}}} h_{i,\text{dang}}(t) \tag{53}$$

corresponding to case 2: Else if there is a lifetime limitation $T_{\text{life,max}}$ (see section 10.2.2.1 above) and the (natural) MTTF is greater than this limitation, it is assumed that the component's life ends at $T_{\text{life,max}}$. Thus the mean (dangerous) occurrence rate is given by

$$\overline{h} = 1/\text{MTTF}_{\text{dang}}(T_{\text{life,max}}) \tag{54}$$

The dangerous occurrence rate $h_{\text{dang}}(\text{t})$ is equal to the sum of the time variant failure rates of the dangerous failure modes

$$h(t) = h_{\text{dang}}(t) = \sum_{i=1}^{n_{\text{dang}}} h_{i,\text{dang}}(t \bmod T_{\text{life,max}}) \tag{55}$$

corresponding to cases 3 and 4: Else it is assumed that the component's life usually ends with component failure. Thus the mean (dangerous) occurrence rate is given by

$$\overline{h} = 1/\text{MTTF}_{\text{dang}} \tag{56}$$

Since in these cases the exchange times cannot be predicted, the time-variant (dangerous) occurrence rate $h_{\text{dang}}(\text{t})$ cannot be predicted. Therefore the mean occurrence rate $\overline{h}$ is used also in transient evaluation.

Regarding unavailability $\overline{Q}$ and *Q(t)*, it is assumed that safe failures don't contribute to (safety related) unavailability. This is reasonable, since "safe failure" means, that the overall system goes to a "safe state", and thus it doesn't matter for safety, how frequently this state is entered and for how long the system stays in this state. Therefore $\overline{Q}$ and *Q(t)* depend on the frequency of dangerous failures and the mean time to detect them. If the dangerous failure cannot be detected (what includes, that it doesn't directly lead to an accident), it will remain in the component and thus in the system, until the component is exchanged due to other events (safe failure of the component, preventive exchange, life end of the system or assembly group).

10.3.1 Unavailability of periodically tested components

Note: In this section, whenever $\overline{h}$ or *h(t)* is written, the (dangerous) failure rates calculated as described in section 10.2.3 are meant.

Components that can fail dangerously without immediately causing an accident are usually periodically tested.

For components that are tested periodically, $\overline{Q}$, $Q_{\max}$ and *Q(t)* are calculated based on the mean occurrence rate similar to *generic basic events* of type *repairable*

$$\overline{Q} = \frac{e^{-\overline{h}\cdot T_{\text{check}}} - 1}{\overline{h}\cdot T_{\text{check}} + \overline{h}\cdot T_{\text{repair}}\cdot(1 - e^{-\overline{h}\cdot T_{\text{check}}})} + 1 \tag{57}$$

In transient evaluation, if T_{check} is greater than 10 times the step time t_{step}, the current unavailability *Q(t)* is given by

$$Q(t) = 1 - (1 - Q_{\text{repair}})\cdot e^{-(1-Q_{\text{repair}})\cdot\overline{h}\cdot((t-t_0) \bmod T_{\text{check}})} \tag{58}$$

with t_0 being the time to the first test (the "phase shift" of the test) and Q_{repair} the (mean) unavailability due to the repair time $Q_{\text{repair}} = \frac{\overline{h}\cdot T_{\text{repair}}}{\overline{h}\cdot T_{\text{repair}}+1}$:

$$Q(t) = 1 - \frac{e^{-\frac{\overline{h}\cdot((t-t_0) \bmod T_{\text{check}})}{\overline{h}\cdot T_{\text{repair}}+1}}}{\overline{h}\cdot T_{\text{repair}} + 1} \tag{59}$$

The maximum unavailability $Q_{\max}$ is given by equation (59) with t=T_{check}:

$$Q_{\max} = 1 - \frac{e^{-\frac{\overline{h}\cdot T_{\text{check}}}{\overline{h}\cdot T_{\text{repair}}+1}}}{\overline{h}\cdot T_{\text{repair}} + 1} \approx 1 - e^{-\overline{h}\cdot(T_{\text{check}}+T_{\text{repair}})} \tag{60}$$

The return rate used in *Markov models* in steady-state evaluation is given by

$$\mu = \frac{\overline{h}}{\overline{Q}} - \overline{h} \tag{61}$$

In transient evaluation, if t_{check} is greater than 10 times the step time t_{step}, the return to the origin state is performed cyclically at times $t_i = n\cdot T_{\text{check}} + T_0 + T_{\text{repair}}$.

10.3.2 Unavailability for non-tested components

Note: In this section, whenever $\overline{h}$ or *h(t)* is written, the (dangerous) failure rates calculated as described in section 10.2.3 are meant.

If the maximum lifetime of the component is limited by the preventive exchange interval $T_{\text{life,max}}$, the (dangerous) unavailability *Q(t)* is given by the (dangerous) unreliability considering the replacements:

$$Q(t) = F_{\text{dang}}(t \bmod T_{\text{life,max}}) \tag{62}$$

The mean unavailability $\overline{Q}$ is conservatively defined as the maximum unavailability which is equal to the unreliability at the end of the exchange interval $T_{\text{life,max}}$:

$$\overline{Q} \stackrel{!}{=} Q_{\text{max}} = F_{\text{dang}}(T_{\text{life,max}}) \tag{63}$$

The same applies if the lifetime is limited by the assembly group (in case the lifetime of the assembly group is not clearly defined, enter a time significantly bigger than the mean lifetime).

If $T_{\text{life,max}}$ is not given, the maximum lifetime of the component is given by the overall system lifetime T_{sys}. In that case the dangerous unavailability *Q(t)* is equal to the unreliability due to the dangerous failure modes:

$$Q(t) = F_{\text{dang}}(t) \tag{64}$$

The mean unavailability $\overline{Q}$ is conservatively defined as the maximum unavailability which is equal to the unreliability at the end of the system's lifetime T_{sys}:

$$\overline{Q} \stackrel{!}{=} Q_{\text{max}} = F_{\text{dang}}(T_{\text{sys}}) \tag{65}$$

10.4 The Component Failure Mode Properties Panel

Figure 69 shows a *complex component* with six *component events* (different failure modes of the component). If a *component event* is selected, it's properties are shown on the left, see figure 70.

10.4.1 General Properties

Name:

A user defined identifier of the *component event*. Every *component event* should have a different name.

Description:

A user defined description of the *component event*.

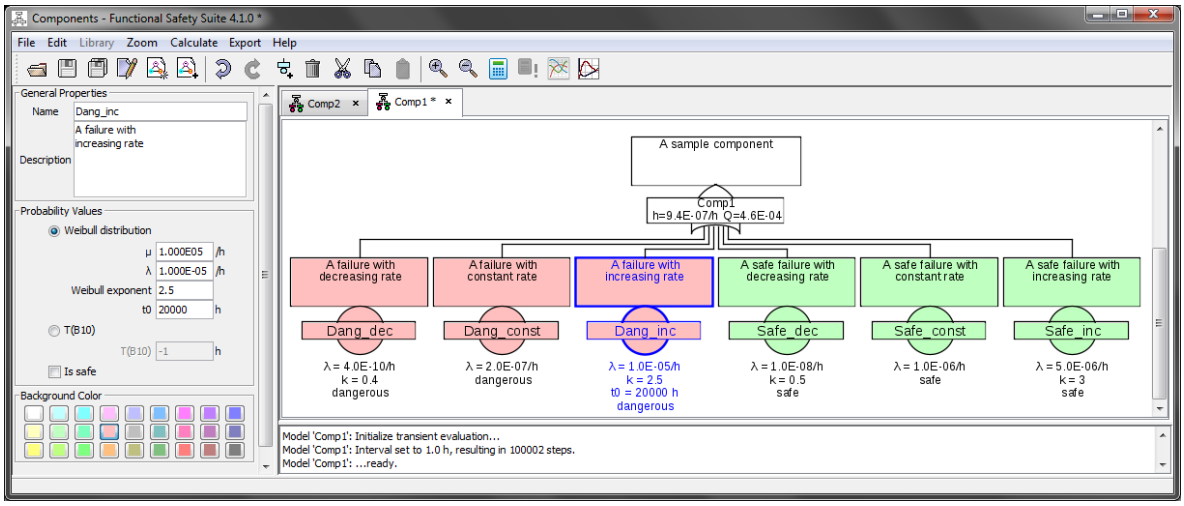


Figure 70: *One out of several failure modes of the component.*

10.4.2 Probability Values

Weibull Distribution Parameters

In Functional Safety Suite version 7.1 each *component event* is assumed to be Weibull distributed. Thus the failure density function *f(t)* is given as

$$f(t) = \lambda \cdot k \cdot (\lambda \cdot (t - t_0))^{k-1} \mathrm{e}^{-(\lambda \cdot (t-t_0))^k} \tag{66}$$

with k being the Weibull exponent.

The failure rate $h(t)$ of the single *component event* is therefore given by

$$h(t) = \lambda \cdot k \cdot \big(\lambda \cdot (t - t_0)\big)^{k-1} \tag{67}$$

Note that a delay time t_0>0 only makes sense for decreasing failure rates and thus is allowed only if $k \geq 1$.

T(B10)

For convenience you can also specify the b10-time T(B10) for this failure mode, if you've got this value. Nevertheless the Weibull exponent k must be defined as well, with 3.0 being the default.

Specification by T(B10) is only supported for t_0=0.

Is safe

Each *component event* can be either safe or dangerous. Note that safe failures need to be modeled as well (if they exist), since they significantly affect the calculated values, including $\mathrm{MTTF}_{\mathrm{dang}}$ etc.

Note: If you're performing non-safety related calculations (e. g. operational reliability or availability calculations), you'll feel no need to distinguish between "safe" and "dangerous" – it's just a failure mode. In that case, you'll have to define all failure modes to be "dangerous", because only the "dangerous" values will be handed over to any higher level model.

10.4.3 Background Color

The background color can be selected separately for each *component event*. By default, safe failure modes have light green background, dangerous failure modes have light red background. If you select white background, these default values will be used instead.

10.5 Editing of Complex Components

Create a *complex component* by **File – New Component**.

In order to add or delete events, select an event with the mouse, then press a button in the tool bar or select a command in the menu bar.

Changing properties of *component events* is done in the properties window.

10.6 The Component Chart Window

After performing a calculation of a *complex component*, the temporal variation of the calculated values can be visualized in a separate window.

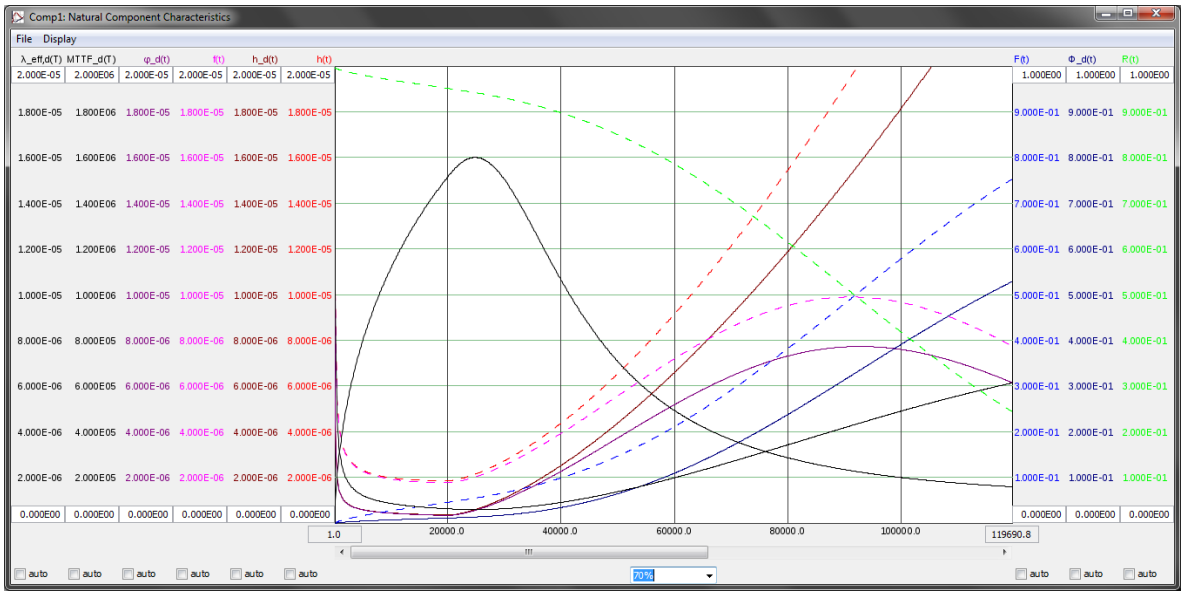


Figure 71: *A component chart window*

By default, the dangerous values are displayed only: The unreliability $F_{\mathrm{d}}(\mathrm{t})$, the failure rate $h_{\mathrm{d}}(\mathrm{t})$, the failure density $f_{\mathrm{d}}(\mathrm{t})$ and the effective mean time to dangerous failure $\mathrm{MTTF}_{\mathrm{dang}}(\mathrm{T})$

for a component life time t. Optionally the overall values can be displayed as well, just select the corresponding check-boxes in the 'Display' menu. They will be presented by dashed lines.

All axis can be scaled and zoomed.

The presented graphics can be exported to a vector graphic (`.svg`) or a bitmap (`.png`). Note that in vector graphics format, the graph data is exported with original resolution, so a later printout will have a very high quality (if not reduced by the later processing).

11 Menus and Commands

11.1 Generals

The descriptions of models or events are immediately changed whenever you type a key in the description field. All other numerical or text values are changed after pressing 'Enter' only.

Note that not all commands or properties will be available in a specific situation. Typically only those possibilities, that make sense and result in a valid model are offered. In the case that an error occurs when executing a command, an error message is displayed in the status bar or in the message window.

11.2 The File Menu

This menu contains all commands related to the *project*, its packages, models and libraries. Most commands are also available in pop-up menus that open when pressing the right mouse button in the *project members tree*.

11.2.1 New Project

If there is an open *project* this will be closed. If necessary you are asked to save data. After that a dialog will appear where you are asked for a name of the new *project*. Finally an empty *project* will be created.

11.2.2 Open Project

If there is an open *project* this will be closed. If necessary you are asked to save data. After that a dialog will appear where you can select the *project* to be opened. All libraries and models referred in the project file will be opened and indicated in the 'Project Members' tree.

11.2.3 Close Project

The current *project* is closed. If necessary you are asked to save changes in the *project*, libraries or models.

11.2.4 Project Properties

A *project properties dialog* window will open where you can set the project properties. Refer to section 3.4 for details.

11.2.5 Create new Package

A new package is created by **File – Create new Package**. You will be asked for the name of the new package. A sub-directory with the given name will be created in the project directory, and the *local library* file will be created.

11.2.6 Import Package

A package created in one *project* can be imported to the currently open *project*. When selecting **File – Import Package**, a dialog will open up where you can select the package directory, and enter a new package name, see figure 72.

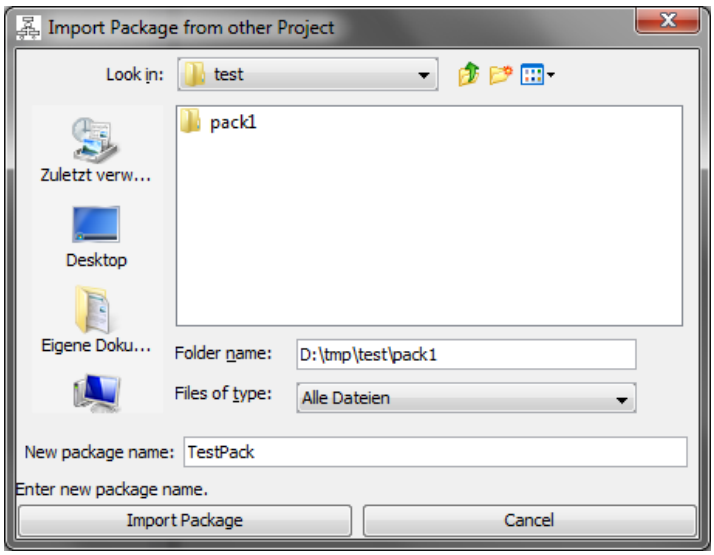


Figure 72: *Import package dialog*

The package will be copied to the open *project*, using the entered name.

11.2.7 Create new Model

A new model is created by **File – Create new Model**. The *Create New Model Dialog* will open, where you can select the *package* the new model shall belong to, and the name and type of the new model, see figure 73.

11.2.8 Add existing Model

When selecting **File – Add existing Model**, a dialog will appear where you can select the model file to be added to the *project*. Per default, only model files with extension `.ignore` will be displayed. Anyhow you can also select other model files, including models already belonging to the *project*.

You can select the *package* to which the model shall be added, and enter a name for it.

If not all *generic basic events* referred by the imported model's *basic events* are available in the open *project*, new *generic basic events* with the referred names are created with default data. You will have to add correct data to these cases before the model can be evaluated. If you forget to do so, the next calculation will show wrong results.

If the model to be added refers *generic basic events* from another *package*, you should import the *generic basic events* before via **Library – Import from other library or project**, see

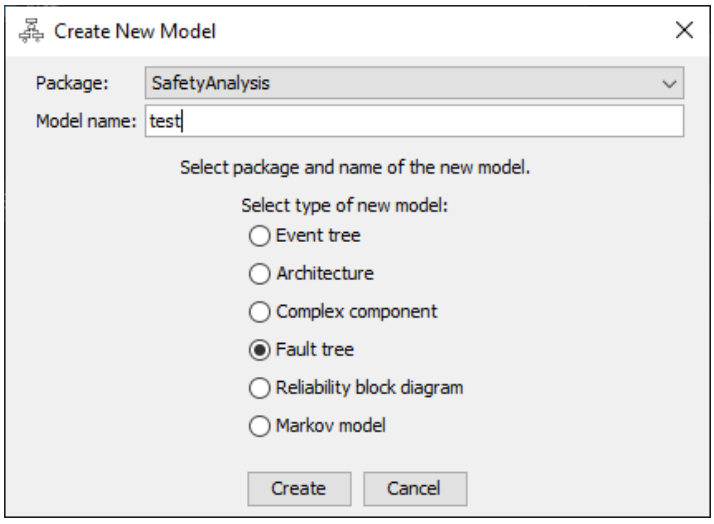


Figure 73: *The create new model dialog*

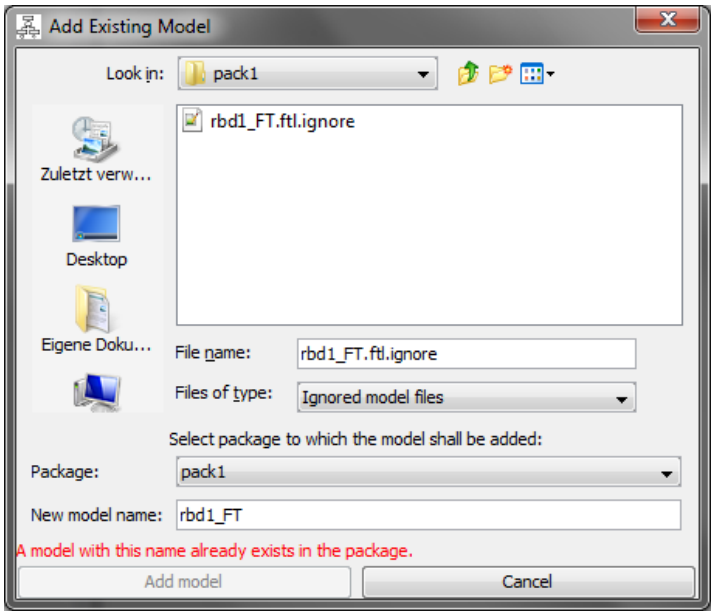


Figure 74: *Add existing model dialog*

section 11.4.6. This command should be used for the *local library*, since the missing *generic basic events* should be added to the *local library*. Since the import function for *generic basic events* will not override *generic basic events* already existing in the *library*, the import must be executed before adding the model (before creation of default *generic basic events*).

11.2.9 Remove active Member

The reference to the model presented in the active tab will be removed from the *project*, the tab will be closed. If necessary you are asked to save data of this model.

If there are references from other models to the one to be removed, you are asked for confirmation, since these model cannot be evaluated anymore afterwards.

11.2.10 Rename active Model

The model presented in the active tab can be renamed by **File – Rename active Model**. A dialog will appear asking you for a new name.

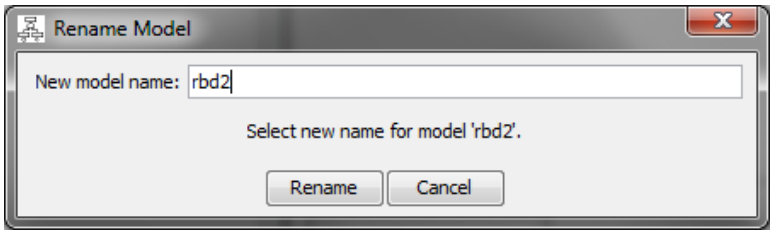


Figure 75: *Rename model dialog*

If there are *links* to this model, you'll have to update the *links* (i. e. the *generic basic event*) manually.

11.2.11 Move active Model

The model presented in the active tab can be moved to another *package* by **File – Move active Model**. A dialog will appear asking you for a new package.

11.2.12 Duplicate active Model

The model presented in the active tab can be duplicated by **File – Duplicate active Model**. A dialog will appear asking you for the *package* and the name of the duplicate, see figure 76.

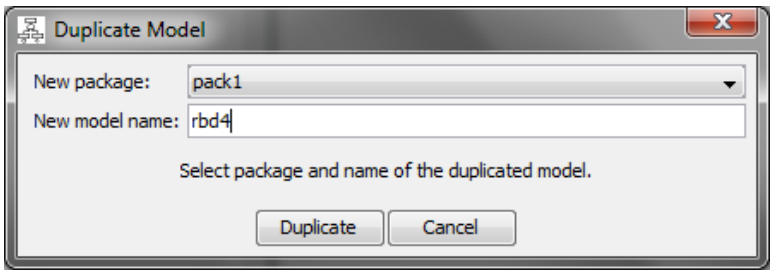


Figure 76: *Duplicate model dialog*

Note that if you duplicate the model to another *package*, the *generic basic events* of the original *local package* won't be reachable anymore.

11.2.13 Save active member

Saves the model currently displayed in the graphics tab. If this is for the first time after creation, you'll be asked for a location and file name. The file extension is automatically appended.

11.2.14 Save All

All models, the *library* and the *project* are saved if changed. Note that a project, that has not been saved after the latest modification, is marked with an asterisk '*' in the window title. An unsaved model is marked with an asterisk '*' in the title of its graphics frame.

11.2.15 Save As

The model in the active tab is saved as a new file. A dialog will appear asking you to select a location and file name. The reference in the *project* is replaced by the new file. The old file will remain unchanged.

11.2.16 List of recently used projects

A list of recently used projects is presented. Selecting one is similar to **File – Open Project**, only that no dialog will appear.

11.2.17 Exit

If necessary you are asked to save changes in the *project* the library or in models. After that the application is terminated.

11.3 The Edit Menu

The **Edit** menu contains all commands related to changing the structure of a model.

The most often needed actions are directly available as button in the menu bar. For some actions keyboard commands (short-cuts) exist, see the entries in the **Edit** menu.

11.3.1 Undo last change

The last ten actions can be withdrawn. Here an action can be either an *edit*-action as stated above or a change of a model or *generic basic event* property in the properties window on the left. So this command is not only related to the structure of a model. The tool-tip text always informs about the next action of the *undo*-action.

11.3.2 Redo last undo

All undo actions can be withdrawn.

11.3.3 Add condition

To add a new *condition* in an *event tree*, select the hazard or the previous *condition*and press **Edit – Add Condition**. A new condition will be created and inserted after the marked *condition*.

11.3.4 Add case

After selecting a *condition* in an *event tree*, a *case* can be added to the condition by **Edit – Add Case**. Each *case* refers to a *generic basic event*, which determines its probability. Typically the *immediate* event model is used, allowing to directly enter the probability p. But also all other models that deliver a unavailability $\overline{Q}$ can be used, including *links*. When adding a case, the new *case* will refer to the last *generic basic event* in the *library*.

11.3.5 New Damage

After selecting a *damage* in an *event tree*, its *generic damage* can be replaced by a new *generic damage*. The user is asked for a name of the new *generic damage*.

Note: To use an already existing *generic damage*, just select it in the *General Properties* panel.

11.3.6 Set Select Mode

The default mode of the *architecture* editor. You can select one *component part*, or any combination of *complex components* and corners and lines of *nets*. Note that Cut/Copy and Paste is only possible if exactly one *complex component* is selected.

11.3.7 Add Component Mode

Select *Add Component Mode* and click to a position in the *architecture*. A dialog will open, see figure 77.

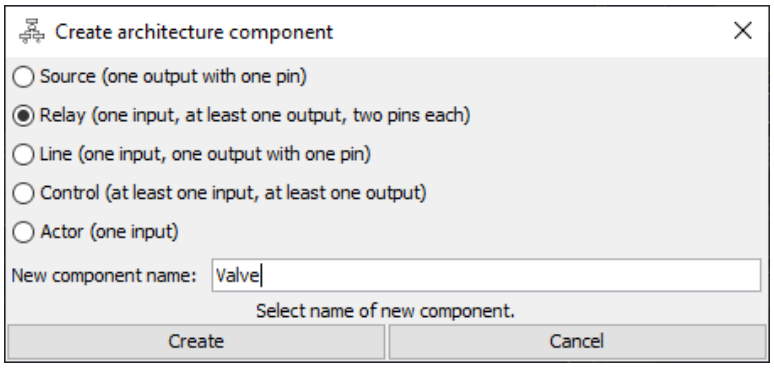


Figure 77: *Create architecture component dialog*

Select class and name of the new component.

11.3.8 Select Draw Net Mode

In order to connect pins of *architecture components*, select this mode. Start drawing by click on a pin or an existing *net* line or corner, and click for a new corner or at another pin, line or corner. A line can be finished by double-click at any point on the grid.

11.3.9 Convert to Fault Tree

Derive a *fault tree* for the current *architecture*. See section 6.6 for details.

11.3.10 Add Failure

In a *complex component* model, a new failure mode (*component event*) is created and added to the right.

11.3.11 Add Block Serial

In a *reliability block diagram* a new block is created and added in series to the selected block(s).

11.3.12 Add Block Parallel

In a *reliability block diagram* a new block is created and added in parallel to the selected block(s).

11.3.13 Add Tree Basic Event

A new *basic event* is created below the currently selected *gate* in a *fault tree*. By default the newest *generic basic event* is used therefore, hence the *generic basic event* created with the last call of **Library – New Generic Basic Event** (see section 11.4.1) or **Library – Duplicate Generic Basic Event** (see section 11.4.4).

11.3.14 Add Gate

A new default *gate* is created below the currently selected *gate* in a *fault tree*. You can add *basic events* to this gate by **Edit – Add Basic Event**, **Library – Create Generic Basic Event** or **Edit –Paste**.

11.3.15 Convert to TRANSFER-IN

The selected block of a *reliability block diagram* is converted to a TRANSFER-IN block.

11.3.16 Convert to Gate

The selected *basic event* in a *fault tree* is converted to a *gate* (maintaining the name and description).

Note: You should change the name of the gate to avoid multiple different events (here a *basic event* and a *gate*) having identical names.

11.3.17 Convert to Subtree

A new *fault tree* is created whose *top event* is the marked *gate*. The marked *gate* is replaced by a new *gate* of type TRANSFER-IN. The name and the description of the former *gate* are preserved. You can change the names of the *gates* and the description of the *top event* of the

new *fault tree* manually. Remember to change the reference in the new TRANSFER-IN gate if you change the name of the *top event* of the new *fault tree*.

11.3.18 Convert Fault Tree to Reliability Block Diagram

A new *reliability block diagram* is created for the active *fault tree*. The new *reliability block diagram* is added to the same package. The name of the *reliability block diagram* will be that of the *fault tree*, extended by `_RBD`.

11.3.19 Convert Reliability Block Diagram to Fault Tree

A new *fault tree* is created based on the active *reliability block diagram*. The new *fault tree* is added to the same package. The name of the *fault tree* will be that of the *reliability block diagram*, extended by `_FT`.

11.3.20 Convert Branch to Markov Model

A *Markov model* is generated for the branch, topped by the selected *gate*. The *fault tree* itself remains unchanged.

The new *Markov model* gets the same name as the fault tree plus the name of the gate. It is directly added to the project, thus you can open it by clicking on its name in the 'Project Members' tree.

You will be asked whether the Markov model shall contain 'direct chains only' or 'complete chains'. A complete conversion to a Markov model considers, that the system's state can in fact "jump" from one (yet incomplete) chain to another chain, until a final state is reached. Please see section 9.5.2 for an example. The internal conversion can get quite complex even for not very large fault trees. Therefore you should consider creating the *direct Markov chains* only.

Converting a *fault tree* to a *Markov model* is a very complex task. For most *fault trees*, this function will do all the work perfectly. However it is generally not possible to model all restorations correctly, because a fault tree doesn't provide information about the restoration strategy. This function will create restorations that usually fit to most systems, however you should check the restoration edges manually.

11.3.21 Convert Reliability Block Diagram to Markov Model

A new *Markov model* is created based on the active *reliability block diagram*. The new *Markov model* is added to the same package. The name of the *Markov model* will be that of the *reliability block diagram*, extended by `_MM`.

See section 11.3.20 for more information.

11.3.22 Import Markov Chains

It is possible to import previously created Markov chains. Since all *states* and *edges* of the *Markov model* will be deleted when importing Markov chains, this command is only available, if the active *Markov model* contains maximum 2 states.

The file containing the Markov chains can be created by **Export – Export Markov Chains** if a *gate* is selected in a *fault tree*. Since it is an XML file, it can also be created manually, or based on the export of another program.

11.3.23 Add Edge

In order to add an *edge* in a *Markov model*, select the *source state*, then select **Edit – Add Edge**, then click on the *target state*. An *edge* referring to the last *generic basic event* in the *library* will be created.

11.3.24 Add State

You can add *states* to a *Markov model* by **Edit – Add State** and clicking the mouse at the position you want to set the *state*. A *state* will be placed to the nearest grid point. A unique name will be automatically assigned to the new *state*. At each grid position, only one *state* can be placed, so if there is a *state* at the nearest position on grid already, no *state* will be added.

11.3.25 Adjust State Names and Positions to other Model

Sometimes you might want to adapt the state names and/or state positions to those of another *Markov model* with similar structure. This is achieved with this command: In the first step, the structure is analyzed in order to identify similar *states*. Two *states* of two *Markov models* are similar, if there is at least one Markov chain, that leads to both of them. A *state* identified as being similar gets the name of its counterpart in the reference model. However since other chains leading to these two *states* might differ, there might be multiple similar *states*, and thus multiple *states* that would be renamed to the same name. In this case, the new name is appended automatically by a number, so that it becomes unique. In the second step, each *state* having the same name as a *state* in the reference model is set to the position of its counterpart in the reference model.

11.3.26 Delete

If a *case* of an *event tree* is selected, this *case* will be deleted from the *condition*.

If a *component part* is selected, which is not mandatory for the class of the *architecture component*, and which is not connected to a *net*, it will be deleted.

If a s

If a *condition* of an *event tree* is selected, the *condition* will be deleted, but only if it doesn't contain any *cases*.

In an *architecture*, the selected elements will be deleted. If a *component part* is selected, it will only be deleted if it is not connected to any net and if this is allowed according to the class of the *architecture component*.

If a failure mode (*component event*) of a *complex component* is selected, it will be deleted.

If a *basic event* of a *fault tree* or a block of a *reliability block diagram* is selected, it will be deleted if there remain enough inputs of the parent gate (see section 7.4 for how many inputs a *gate* of a certain type needs).

If a *gate* is selected, it will be deleted and its input events will be shifted to its parent gate.

If an *edge* is selected, it will be deleted.

If a *state* is selected, it will be deleted including all *edges* connected to it.

11.3.27 Delete Component or Selection

A selection of *architecture components* and/or *nets* can be deleted with **Del**.

If a *component part* is selected, the complete *architecture component* can be deleted by **Ctrl**+**Del**.

11.3.28 Delete Branch

The branch topped by the selected *gate* will be deleted.

11.3.29 Cut

The selected *case*, *condition*, *component event*, *basic event*, *state* or the branch topped by the selected *gate* will be deleted.

A deleted *component event*, *basic event*, *state* or branch is stored in the background so that it can be pasted somewhere later on. A *case* or *condition* cannot be pasted, therefore the 'cut' command is the same as a 'delete' for these events.

11.3.30 Copy

The selected *damage*, *component event*, *basic event*, *state* or the branch topped by the selected *gate* will be copied to a background memory. It can be pasted somewhere later on, see below.

11.3.31 Paste

If a *event tree* is active and a *damage* is selected, and a *damage* has been copied before, the *generic damage* of the selected *damage* is replaced by that of the copied *damage*.

If a *complex component* is active and a *component event* is selected, the *component event* is pasted as new event to the component.

If a *fault tree* is active and a *gate* is selected, a *basic event* or branch copied or cut before is pasted as new input to the selected *gate*.

If a *Markov model* is active and a *state* is selected, an *edge* copied or cut before will be added starting in the selected *state*. Click to another *state* in order to set the target of the *edge* just as for the **Edit − Add Edge** command.

If a *Markov model* is active, no *state* is selected, and a single *state* has been copied or cut before, it will be added after clicking to an empty position.

If a *Markov model* is active, no *state* is selected, and multiple states and edges have been cut or copied before, they will be added to the active *Markov model* below or at the right of the existing states.

11.3.32 Paste Serial

The saved block(s) will be added in series to the selected block(s) of the *reliability block diagram*.

11.3.33 Paste Parallel

The saved block(s) will be added in parallel to the selected block(s) of the *reliability block diagram*.

11.3.34 Move Left/Move Right

In an *event tree* the selected *condition* is moved one position right or left.

In a *fault tree* the selected *basic event* or the branch topped by the selected *gate* is moved one input to the left or to the right.

Note: This command is not available for children of *inhibit gates*.

In a *Markov model* the selected *state* is moved to the next free position left or right of its current position.

In a *reliability block diagram* the selected blocks are moved left or right. This is equivalent to changing the sequence of the inputs of an OR-gate.

11.3.35 Move Up

In a *fault tree* the selected *basic event* or branch topped by the selected *gate* is moved one level up. If the parent of the selected event is an *inhibit gate* or will get too few inputs the command is ignored.

In a *Markov model* the selected *state* is moved to the next free position over of its current position.

In a *reliability block diagram* the selected blocks are moved up. This is equivalent to changing the sequence of the inputs of an AND-gate (or another conjunction gate).

11.3.36 Move Down

In a *Markov model* the selected *state* is moved to the next free position under its current position.

In a *reliability block diagram* the selected blocks are moved down. This is equivalent to changing the sequence of the inputs of an AND-gate (or another conjunction gate).

11.4 The Library Menu

11.4.1 New Generic Basic Event

A dialog will appear, where you can enter the name of the new *generic basic event*. If no *generic basic event* with the entered name already exists, a new *generic basic event* will be created with default data.

If a *basic event* of a *fault tree* or an *edge* is marked, or a *case* of an *event tree*, the new *generic basic event* will be assigned to it automatically.

If a *gate* is marked, a new *basic event* referring to the new *generic basic event* will be created and added to the gate if possible.

11.4.2 Rename Generic Basic Event

If a *basic event* or an *edge* is marked, or a *case* of an *event tree*, the name of the referred *generic basic event* can be altered. A dialog will appear, where you can enter a new name.

If a *generic basic event* with the new name already exists, you are asked for another name.

The operation is executed on the existing *generic basic event*, no new *generic basic event* is created. Therefore the names of all *basic events* referring to this *generic basic event* in all models will change.

11.4.3 Move Generic Basic Event

Move the selected *generic basic event* to another *package*. A dialog will open where you can select the new *package*.

11.4.4 Duplicate Generic Basic Event

Duplicate the selected *generic basic event* to another *package*. A dialog will open where you can select the new *package* and the new name.

11.4.5 Remove unused Generic Basic Event (GBEs)

Generic basic events are not deleted automatically (you can see this by checking the name list in the properties panel of a *basic event* of a *fault tree* or an *edge*). Thus if you remove a model from a *project* or delete *basic events* from a model, it will happen, that the *library* contains *generic basic events* not used anymore in any model of the *project*. When this menu action is selected in the *library* view, these unused *generic basic events* are removed from the *library*.

11.4.6 Import from other library or project

If you select this action in the *library* view, a dialog will appear, where you can select a *library* file (`.lib` or `.gbes`). The *generic basic events* defined in the selected file will be copied to the currently shown *library*.

Those *generic basic events* with names already existing in the new *library* will not be copied. The messages in the output window will provide detailed information.

11.4.7 Export to CSV file

If you select this action in the *library* view, a dialog will appear, where you can enter a file name. All *generic basic events* of the library will be saved in a text file, one event per line. Line breaks in the description will be replaced by spaces.

11.5 The Zoom Menu

Selects the zoom factor of the models and exported bitmap graphics (`.png`) of the models.

11.6 The Calculate Menu

11.6.1 Calculate Model Values

The probabilistic values of the model in the active tab are calculated. This includes the calculation of all referred *sub-trees* or linked models. For details and parameters related to calculation, see the model type specific sections in this guide.

11.6.2 Calculate importances

This action is only available after successful calculation. A dialog will appear, where you can select which importances shall be calculated, see figure 78.

For explanations of the available importances, see appendix A.

The results are shown in a separate window, see figure 79. The values can be exported as (complete) HTML file, as a HTML snippet (to be included in another HTML file) or as CSV file (in order to import it further evaluate them e. g. in a spreadsheet program).

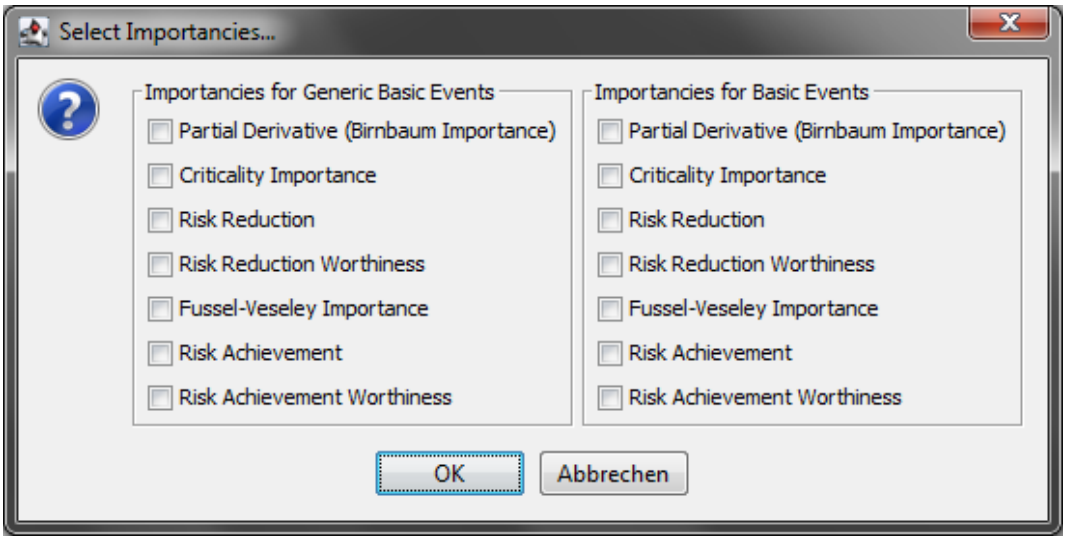


Figure 78: *The importances dialog*

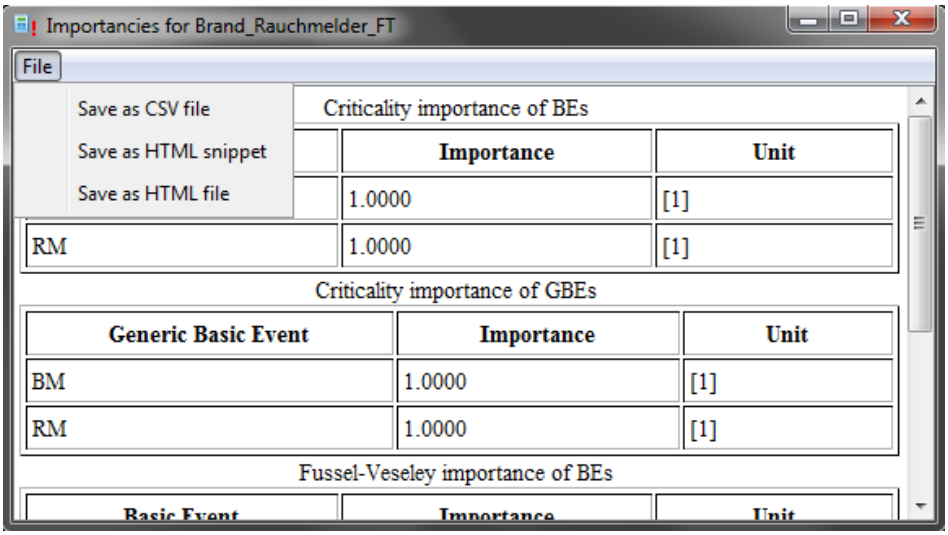


Figure 79: *The importances window*

11.6.3 Determine and show Prime Implicants (Minimal Cut-Sets)

This action is only available after successful calculation. The prime implicants for unavailability, occurrence rate or unreliability (depending on the selected value in the *fault tree evaluation properties* dialog) of the *top event* of the *fault tree* or *reliability block diagram* presented in the active tab are determined. After the prime implicants have been determined, they will be shown in a new window.

A prime implicant consists of one or several conjuncted *basic events*. The *basic events* are separated by asterisks '*'. Each implicant is stated in a separate line. For coherent fault trees, the prime implicants are identical to the minimal cut-sets. For non-coherent fault trees, prime implicants are not canonical, i. e. there are multiple equivalent sets of prime implicants in

general.

As in the name fields of *basic events*, the first dot ('.') separates name and suffix. Multiple parts of the suffix created by TRANSFER-IN gates are also separated by dots. The part after the first dot is the name of the highest level TRANSFER-IN gate, followed by lower TRANSFER-IN gate names down to the original suffix of the *basic event*.

The values can be exported as (complete) HTML file, as a HTML snippet (to be included in another HTML file) or as CSV file (in order to import it further evaluate them e. g. in a spreadsheet program).

Note: The number of lines shown in the window and exported to a HTML file is limited to the number stated in the *project properties dialog*, see section 3.4.3.1. The CSV file export will include all prime implicants.

11.6.4 Check to SIRF Rules

The qualitative *fault tree* is checked against [SiRF] rules.

11.6.5 Show Chart

After performing a transient (time-variant) evaluation, the temporal variation of the calculated values can be visualized in a separate window.

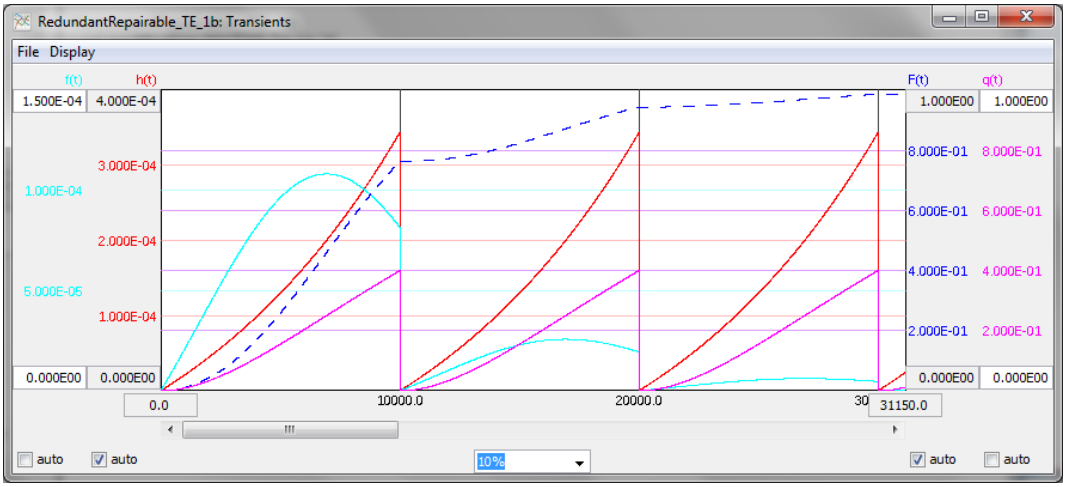


Figure 80: *A chart window*

By default, the values selected in the model-specific dialog are displayed. Optionally, additional values might be available for display, depending on the type of the model and the evaluation parameters.

All axis can be scaled and zoomed.

The presented graphics can be exported to a vector graphic (`.svg`) or a bitmap (`.png`) file, select **File – Export ...** in the menu of the chart. Note that in vector graphics format,

the graph data is exported with original resolution, so a later printout will have a very high quality (if not reduced by the later processing).

11.6.6 Show Component Chart

After performing the evaluation of a *complex component*, the temporal variation of the calculated values can be visualized in a separate window. See section 10.6 for more details.

11.7 The Export Menu

11.7.1 Create Report

Create a report, see appendix B.

11.7.2 Update Report

Update an existing report, see appendix B.

11.7.3 Export Graphic as PNG

The graphic of the currently shown model is saved as portable network graphic (`.png`) in the `graphics` sub-directory below the *package* directory. The selected zoom factor is applied as well as the current markings, but the output is not limited to the visible part. The resolution will be twice the resolution of the display.

11.7.4 Export All Graphics as PNG

The graphics of all models, for which a tab exists in the *model graphics tab pane*, are saved as portable network graphic files (`.png`) in the `graphics` sub-directory below the *package* directory.

11.7.5 Export Graphic as SVG

The graphic of the currently shown model is saved as scalable vector graphic (`.svg`) in the `graphics` sub-directory below the *package* directory. Markings are not preserved. This format can be imported by most vector graphic programs. It can also be displayed by most browsers.

11.7.6 Export All Graphics as PNG

The graphics of all models, for which a tab exists in the *model graphics tab pane*, are saved as scalable vector graphic files (`.svg`) in the `graphics` sub-directory below the *package* directory.

11.7.7 Export Basic Events List

Saves a list of all *generic basic events* of the currently displayed *library* in a text file (without parameters).

The name of each *generic basic event* is followed by a list of models in which it is used.

After that for each model it is stated, which of the *generic basic events* contained in this *library* it uses.

11.7.8 Export Transient Values

After performing a transient evaluation, the calculated data can be saved to a text file. Its extension is `.tdf`. Each time step is a line, values are separated by ';'. The first line in the file indicates the model, the second line the meaning of each column.

11.7.9 Export Final Tree

Save the *final tree* of the active *fault tree* as new model.

The *final tree* is the *fault tree*, in which all COMBINATION gates and TRANSFER-IN gates have been replaced by the adequate branch, and all REDUCED-COMBINATION gates and PRIORITY-AND gates have been replaced by a link to another (temporary) model.

Note: If the *fault tree* includes REDUCED-COMBINATION gates or PRIORITY-AND gates, the exported *final tree* cannot be evaluated, since the temporary models created for these gates are not exported.

11.7.10 Export Markov Chains

This function will create the Markov chains, that are equivalent to the *fault tree* branch topped by the selected *gate*. Creation of these chains is done in multiple steps. You may select, which step's result shall be exported.

You will be asked whether the Markov chains shall contain 'direct chains only' or 'complete chains'. A complete conversion to a Markov model considers, that the system's state can in fact "jump" from one (yet incomplete) chain to another chain, until a final state is reached. Please see section 9.5.2 for an example. The internal conversion can get quite complex even for not very large fault trees.

11.7.11 Export States and Edges List

Saves a list of all *states* and *edges* of the *Markov model* in a text file, including their descriptions.

11.7.12 Export Final Markov Model

Exports the *Markov model*, that has been created internally for the last evaluation of the active *Markov model*. You may add the exported *Markov model* to the project manually by **File – Add Markov Model**. In that case, you might want to adjust the state names and positions to the original model by **Edit – Adjust State Names and Positions to other Model**, see section 11.3.25.

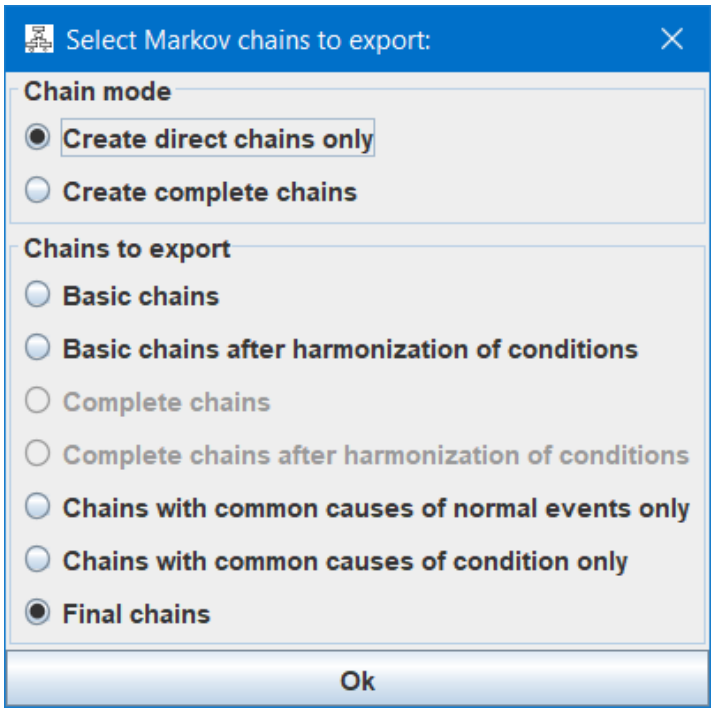


Figure 81: *The dialog to select, which chains to save.*

11.8 The Help and Configuration Menu

11.8.1 Help

The content of this document is presented in HTML format.

11.8.2 Set User Interface Look&Feel

Select your preferred Look&Feel, depending on which User Interface Manager are installed on your machine. Restart of Functional Safety Suite will be necessary in order to activate the

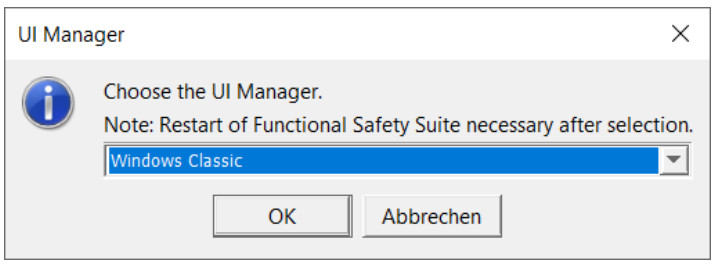


Figure 82: *The dialog to select the user interface Look&Feel.*

new Look&Feel.

11.8.3 Set License File

Specify the path to the license file here.

Note: You must restart Functional Safety Suite to load the new license file.

11.8.4 About

A window opens, indicating the version of Functional Safety Suite and some parameters of the license.

11.9 The tool bar

All frequently used commands are also available as buttons in the tool bar. The tool bar is context sensitive.

Table 6: *Toolbar buttons*

Icon	Command
	Close project and open another project
	Save active model
	Save all
	Project properties
	Create new model
	Add existing model
	Open the architecture symbol editor
	Architectures: Set select mode
	Architectures: Add a new *architecture component*
	Architectures: Select draw net mode
	Undo last change
	Redo last undo
	Event trees: Add condition to event tree
	Event trees: Add case to event tree
	Event trees: Create new damage for event tree

Continued on next page

Icon	Command
	Fault trees: Add tree basic event or component event
	Fault trees: Add gate event
	Fault trees: Convert to gate
	RBDs: Add Block Serial
	RBDs: Add Block Parallel
	Markov models: Add edge
	Markov models: Add state
	Delete event, branch or state
	Cut marked event, branch or state
	Copy marked event, branch or state
	Paste event, branch or state
	Create new generic basic event
	Rename generic basic event
	Duplicate generic basic event
	Zoom in
	Zoom out
	Calculate model values
	Calculate importance of basic events
	Show chart with transient values
	Show chart with component values
	Determine and show prime implicants (minimal cut-sets)

11.10 Menus and Commands of the Symbol Editor

Table 7: *Toolbar buttons*

Icon	Command
	Create a new symbol of the selected type in the selected library
	Create a new text in the symbol
	Create a new rectangle in the symbol
	Create a new ellipse or circle in the symbol
	Create a new line or polygon in the symbol

APPENDIX

A Importances

Importances indicate the influence of each basic event on a system parameter. There is a whole range of importances in the literature, which are often defined differently, and almost always without mentioning the system parameter for which they were defined. Usually it is the unreliability $F_{\mathrm{sys}}(T_{\mathrm{mission}})$.

Importance for the mean system failure rate $\overline{h_{\mathrm{sys}}}$ is practically not mentioned in the literature. This is understandable because importances are almost always defined in connection with fault trees, and the calculation of the system failure rate with fault trees is also rarely dealt with in literature. Some importances can be transferred directly to the failure rate, some analogous, and some importances cannot be meaningfully defined for the failure rate.

Although importances were mostly defined for use with fault trees, some can also be applied to other models, such as Markov models.

A.1 Partial Derivative (PD) and Birnbaum-Importance (BI)

The partial derivative is an obvious measure of the importance of individual base events of the system value $F_{\mathrm{sys}}(\mathrm{T})$, $\overline{Q}_{\mathrm{sys}}$ or $\overline{h}_{\mathrm{sys}}$. The partial derivatives of the system unreliability F_{sys} to the unreliability of each basic event F_{x} are also called Birnbaum-Importances.

A.1.1 Partial derivative of the system unreliability

For fault trees, the derivative of the system unreliability F_{sys} to the unreliability of each basic event F_{x} is given by:

$$\mathrm{I}^{\mathrm{PD}}_{\mathrm{F,x}} = \frac{\partial F_{\mathrm{sys}}(T)}{\partial F_x(T)} = \frac{F_{\mathrm{sys}}(T, \mathbf{F} + \partial F_x) - F_{\mathrm{sys}}(T, \mathbf{F})}{\partial F_x(T)} \tag{68}$$

Here, $\mathbf{F}$ denotes the vector of the unreliabilities of the basic events – either as time variant functions or at the end of system lifetime. If the fault tree contains conditions, i. e. basic events described by their unavailability Q instead of F, the derivative of the system unreliability F_{sys} to the unreliability of these basic events is not defined. Instead, the derivative of the system unreliability to the condition's unavailability Q_{x} may be defined:

$$\mathrm{I}^{\mathrm{PD}}_{\mathrm{F,x}} = \frac{\partial F_{\mathrm{sys}}(T)}{\partial \overline{Q}_x} = \frac{F_{\mathrm{sys}}(T, \mathbf{F}, \mathbf{Q} + \partial Q_x) - F_{\mathrm{sys}}(T, \mathbf{F}, \mathbf{Q})}{\partial \overline{Q}_x} \tag{69}$$

In case of Markov models, where the basic events are described by their failure rates $\mathbf{h}$ instead of F, the partial derivatives can be defined by:

$$\mathrm{I}^{\mathrm{PD}}_{\mathrm{F,x}} = \frac{\partial F_{\mathrm{sys}}(T)}{\partial \overline{h}_x} = \frac{F_{\mathrm{sys}}(T, \mathbf{h} + \partial h_x) - F_{\mathrm{sys}}(T, \mathbf{h})}{\partial \overline{h}_x} \tag{70}$$

If the Markov model contains conditions, the derivative to their unavailability can be defined in the same way as for fault trees, see formula (69).

A.1.2 Partial derivative of the system unavailability

For fault trees, the derivative of the system unavailability Q_{sys} to the unavailability of each basic event Q_{x} is given by:

$$\mathrm{I}^{\mathrm{PD}}_{\mathrm{Q,x}} = \frac{\partial \overline{Q}_{\mathrm{sys}}}{\partial \overline{Q}_x} = \frac{\overline{Q}_{\mathrm{sys}}(\mathbf{Q} + \partial Q_x) - \overline{Q}_{\mathrm{sys}}(\mathbf{Q})}{\partial \overline{Q}_x} \tag{71}$$

Here, $\mathbf{Q}$ denotes the vector of the unavailabilities of the basic events – either as functions of time or as mean values.

In case of Markov models, where the basic events are described by their failure rates h instead of Q, the partial derivative can be defined by:

$$\mathrm{I}^{\mathrm{PD}}_{\mathrm{Q,x}} = \frac{\partial \overline{Q}_{\mathrm{sys}}}{\partial \overline{h}_x} = \frac{\overline{Q}_{\mathrm{sys}}(\mathbf{h} + \partial h_x) - \overline{Q}_{\mathrm{sys}}(\mathbf{h})}{\partial \overline{h}_x} \tag{72}$$

If the Markov model contains conditions, the partial derivative can be defined by formula (71).

A.1.3 Partial derivative for the system failure rate

For fault trees, the system failure rate h_{sys} is a function of both the failure rate h_{x} and the unavailability Q_{x} of each basic event, in general. Therefore, a partial derivative to the failure rate h_{x} only ($\frac{\partial h_{\mathrm{sys}}}{\partial h_x}$) doesn't make much sense. You could of course define two derivatives $\mathrm{I}^{\mathrm{PD}}_{\mathrm{h_h,x}} = \frac{\partial h_{\mathrm{sys}}}{\partial h_x}$ and $\mathrm{I}^{\mathrm{PD}}_{\mathrm{h_Q,x}} = \frac{\partial h_{\mathrm{sys}}}{\partial Q_x}$, but Q_{x} depends on the failure h_{x} for most basic event models:

$$h_{\mathrm{sys}} = \mathrm{fct}(h_x, Q_x = \mathrm{fct}(h_x)) \tag{73}$$

Thus, it makes more sense to define $\mathrm{I}^{\mathrm{PD}}_{\mathrm{h,x}}$ as derivative to the (mean) failure rate of the basic event λ_{i}:

$$\mathrm{I}^{\mathrm{PD}}_{\mathrm{h,x}} = \frac{\partial \overline{h}_{\mathrm{sys}}}{\partial \lambda_x} = \frac{\overline{h}_{\mathrm{sys}}(\boldsymbol{\lambda} + \partial \lambda_x) - \overline{h}_{\mathrm{sys}}(\boldsymbol{\lambda})}{\partial \lambda_x} \approx \frac{\partial \left(\sum_{i=1}^{n_{\mathrm{MCS}}} h_{\mathrm{MCS,i}} \right)}{\partial \lambda_x} = \sum_{i=1}^{n_{\mathrm{MCS}}} \frac{\partial h_{\mathrm{MCS,i}}}{\partial \lambda_x} \tag{74}$$

If you calculate the occurrence rate $h_{\mathrm{MCS,i}}$ of each minimal cut-set MCS by

$$\begin{aligned} h_{\mathrm{MCS}} \lessapprox\ & h_1 \cdot Q_2 \cdot Q_3 \cdot \ldots \cdot Q_m \\ & + h_2 \cdot Q_1 \cdot Q_3 \cdot \ldots \cdot Q_m \\ & + \ldots \\ & + h_m \cdot Q_1 \cdot Q_2 \cdot \ldots \cdot Q_{m-1} \end{aligned} \tag{75}$$

you'll get

$$\begin{aligned}\frac{\partial h_{\mathrm{MCS,i}}}{\partial \lambda_x} &\approx \frac{\partial (h_1 \cdot Q_2 \cdot Q_3 \cdot \ldots \cdot Q_m)}{\partial \lambda_x} \\ &+ \frac{\partial (h_2 \cdot Q_1 \cdot Q_3 \cdot \ldots \cdot Q_m)}{\partial \lambda_x} \\ &+ \ldots \\ &+ \frac{\partial (h_m \cdot Q_1 \cdot Q_2 \cdot \ldots \cdot Q_{m-1}}{\partial \lambda_x} \\ &= \sum_{j=1}^{m} \frac{\partial \left(h_j \cdot \prod_{k=1, k\neq j}^{m} Q_k \right)}{\partial \lambda_x}\end{aligned} \tag{76}$$

If basic event x is not included in $\mathrm{MCS_i}$ the derivative is zero. If it is included, the summand with j=x is equal to $\prod_{k=1,k\neq j}^{m} Q_k$ (where all unavailabilities of this product are independent of basic event x), and all summands with $j \neq x$ are equal to $h_j \frac{\partial Q_x}{\partial \lambda_x} \prod_{k=1,k\neq j,k\neq x}^{m} Q_k$.

Thus we get

$$\mathrm{I}_{\mathrm{h,x}}^{\mathrm{PD}} \approx \sum_{i=1}^{n_{\mathrm{MCS}}} \begin{cases} 0 & \text{if } \mathrm{BE}_x \notin \mathrm{MCS_i} \\ \prod_{k=1,k\neq x}^{m} Q_k + \frac{\partial Q_x}{\partial \lambda_x} \cdot \sum_{j=1,j\neq x}^{m} \left(h_j \cdot \prod_{k=1,k\neq j,k\neq x}^{m} Q_k \right) & \text{if } \mathrm{BE}_x \in \mathrm{MCS_i} \end{cases} \tag{77}$$

For Markov models, the derivative of the system failure rate to the event's failure rate is just given by

$$\mathrm{I}_{\mathrm{h,x}}^{\mathrm{PD}} = \frac{\partial \overline{h}_{\mathrm{sys}}}{\partial \overline{h}_x} = \frac{\overline{h}_{\mathrm{sys}}(\mathbf{h} + \partial h_x) - \overline{h}_{\mathrm{sys}}(\mathbf{h})}{\partial \overline{h}_x} \tag{78}$$

In Functional Safety Suite the derivatives are calculated numerically. The principle described by formula (74) is implemented in a simple way: In order to calculate $\mathrm{I}_{\mathrm{h,x}}^{\mathrm{PD}}$, all basic event values are altered in parallel, i. e. each basic event value given to the model for calculating the system value will be varied, let it be h_{x}, Q_{x} or F_{x}.

A.2 Criticality Importance (CRI) and statistical confidence

The criticality importance is defined as the ratio of the relative change in system quantity Ψ for the relative change of basic event quantity χ:

$$I_{\Psi,x}^{\mathrm{CRI}} = \frac{\frac{\partial \Psi_{\mathrm{sys}}}{\Psi_{\mathrm{sys}}}}{\frac{\partial \chi_x}{\chi_x}} \tag{79}$$

The criticality importance is the most interesting importance at all, because it gives a direct answer to the question of how much a (relative) uncertainty in the statistical value of a base event affects the overall result: A CRI of e. g. 0.1 means, that the system quantity Ψ will increase by 10% if the basic event's failure rate is in fact twice as high as assumed (+100%).

Or in other words: The greater the criticality importance, the greater the impact that a relative improvement of the component has. It is therefore sometimes called Upgrading Importance.

In addition, the criticality importance is equal to the probability that component x is in failure, if the system has failed. Hence it gives a hint where to look for the failure first, if the system has failed.

If the system unreliability is given by $F_{\mathrm{sys}}(\mathrm{T})$=fct(F) (e. g. by a fault tree without conditions), the criticality importance can be calculated by:

$$I_{F,x}^{\mathrm{CRI}} = \frac{\frac{\partial F_{\mathrm{sys}}(T)}{F_{\mathrm{sys}}(T)}}{\frac{\partial F_x}{F_x}} = \frac{F_{\mathrm{sys}}(\mathbf{F}+\partial F_x) - F_{\mathrm{sys}}(\mathbf{F})}{F_{\mathrm{sys}}(\mathbf{F})} \cdot \frac{F_x}{\partial F_x} = I_{F,x}^{\mathrm{PD}} \cdot \frac{F_x}{F_{\mathrm{sys}}(\mathbf{F})} \tag{80}$$

A.3 Risk Reduction (RR)

The risk reduction is the difference of the system value $\overline{Q}$, $F(T)$ or $\overline{h}$, given component **x** would never fail. For a fault tree, the risk reduction can be calculated by

$$I_{F,x}^{\mathrm{RR}} = F_{\mathrm{sys}}(T,\mathbf{F}) - F_{\mathrm{sys}}\left(T,\mathbf{F}\big|_{F_x:=0}\right) \tag{81}$$

where $F_{\mathrm{sys}}\left(T,\mathbf{F}\big|_{F_x:=0}\right)$ denotes the vector of the component unreliabilities, in which the unreliability F_{x} of component **x** is set to zero.

If the fault tree contains conditions, and component **x** describes such a condition, the formula can be replace by

$$I_{F,x}^{\mathrm{RR}} = F_{\mathrm{sys}}(T,\mathbf{F},\mathbf{Q}) - F_{\mathrm{sys}}\left(T,\mathbf{F},\mathbf{Q}\big|_{Q_x:=0}\right) \tag{82}$$

where $F_{\mathrm{sys}}\left(T,\mathbf{F},\mathbf{Q}\big|_{Q_x:=0}\right)$ denotes the vector of the component unavailabilities (in general time dependent unavailability functions, in fact), in which the unavailability Q_{x} of component x is set to zero.

Equivalent formula can be used for unavailabilities:

$$I_{Q,x}^{\mathrm{RR}} = \overline{Q}_{\mathrm{sys}}(\mathbf{Q}) - \overline{Q}_{\mathrm{sys}}\left(\mathbf{Q}\big|_{Q_x:=0}\right) \tag{83}$$

The risk reduction can directly be applied to the system failure rate, but in case of fault trees, both h_{x} and Q_{x} must be set to zero:

$$I_{h,x}^{\mathrm{RR}} = \overline{h}_{\mathrm{sys}}(\mathbf{h},\mathbf{Q}) - \overline{h}_{\mathrm{sys}}\left(\mathbf{h}\big|_{h_x:=0},\mathbf{Q}\big|_{Q_x:=0}\right) \tag{84}$$

A.4 Risk Reduction Worth (RRW)

The Risk Reduction Worth states the relative reduction of the system value $F(T)$, $\overline{Q}$ or $\overline{h}$ if component $\mathbf{x}$ wouldn't fail:

$$I_{F,x}^{\mathrm{RRW}} = \frac{F_{\mathrm{sys}}(T,\mathbf{F}) - F_{\mathrm{sys}}\left(T,\mathbf{F}\big|_{F_x:=0}\right)}{F_{\mathrm{sys}}\left(T,\mathbf{F}\big|_{F_x:=0}\right)} = \frac{F_{\mathrm{sys}}(T,\mathbf{F})}{F_{\mathrm{sys}}\left(T,\mathbf{F}\big|_{F_x:=0}\right)} - 1 \tag{85}$$

$$I_{Q,x}^{\mathrm{RRW}} = \frac{\overline{Q}_{\mathrm{sys}}(\mathbf{Q}) - \overline{Q}_{\mathrm{sys}}\left(\mathbf{Q}\big|_{Q_x:=0}\right)}{\overline{Q}_{\mathrm{sys}}\left(\mathbf{Q}\big|_{Q_x:=0}\right)} = \frac{\overline{Q}_{\mathrm{sys}}(\mathbf{Q})}{\overline{Q}_{\mathrm{sys}}\left(\mathbf{Q}\big|_{Q_x:=0}\right)} - 1 \tag{86}$$

$$I_{h,x}^{\mathrm{RRW}} = \frac{\overline{h}_{\mathrm{sys}}(\mathbf{h},\mathbf{Q}) - \overline{h}_{\mathrm{sys}}\left(\mathbf{h}\big|_{h_x:=0},\mathbf{Q}\big|_{Q_x:=0}\right)}{\overline{h}_{\mathrm{sys}}\left(\mathbf{h}\big|_{h_x:=0},\mathbf{Q}\big|_{Q_x:=0}\right)} = \frac{\overline{h}_{\mathrm{sys}}(\mathbf{h},\mathbf{Q})}{\overline{h}_{\mathrm{sys}}\left(\mathbf{h}\big|_{h_x:=0},\mathbf{Q}\big|_{Q_x:=0}\right)} - 1 \tag{87}$$

Obviously the Risk-Reduction-Worth can take any value from 0 to infinity. The higher the RRW, the higher the effect of an enhancement of component $\mathbf{x}$. A value close to zero means that component $\mathbf{x}$ has no significant effect.

Note: In some other definitions, the summand -1 is omitted.

A.5 Fussel-Vesely-Importance (FV)

Even though the Fussel-Vesely importance has been defined based on minimal cut-sets of fault trees originally, it can be defined in a general manner as the quotient of the risk reduction (RR) and the the original system value:

$$I_{F,x}^{\mathrm{FV}} = \frac{I_{F,x}^{\mathrm{RR}}}{F_{\mathrm{sys}}(T,\mathbf{F})} = \frac{F_{\mathrm{sys}}(T,\mathbf{F}) - F_{\mathrm{sys}}\left(T,\mathbf{F}\big|_{F_x:=0}\right)}{F_{\mathrm{sys}}(T,\mathbf{F})} \tag{88}$$

$$I_{Q,x}^{\mathrm{FV}} = \frac{I_{Q,x}^{\mathrm{RR}}}{\overline{Q}_{\mathrm{sys}}(\mathbf{Q})} = \frac{\overline{Q}_{\mathrm{sys}}(\mathbf{Q}) - \overline{Q}_{\mathrm{sys}}\left(\mathbf{Q}\big|_{Q_x:=0}\right)}{\overline{Q}_{\mathrm{sys}}(\mathbf{Q})} \tag{89}$$

$$I_{h,x}^{\mathrm{FV}} = \frac{I_{h,x}^{\mathrm{RR}}}{\overline{h}_{\mathrm{sys}}(\mathbf{h},\mathbf{Q})} = \frac{\overline{h}_{\mathrm{sys}}(\mathbf{h},\mathbf{Q}) - \overline{h}_{\mathrm{sys}}\left(\mathbf{h}\big|_{h_x:=0},\mathbf{Q}\big|_{Q_x:=0}\right)}{\overline{h}_{\mathrm{sys}}(\mathbf{h},\mathbf{Q})} \tag{90}$$

A.6 Risk Achievement (RA)

The Risk Achievement value is defined as the difference of the system value with an extremely bad component, i. e. Q_{x}:=1 (component never available) or F_{x}:=1 (component for sure fails until end of mission/end of system lifetime), and the estimated system value:

$$I_{F,x}^{\mathrm{RA}} = F_{\mathrm{sys}}\left(T,\mathbf{F}\big|_{F_x:=1}\right) - F_{\mathrm{sys}}(T,\mathbf{F}) \tag{91}$$

or

$$I_{Q,x}^{\mathrm{RA}} = \overline{Q}_{\mathrm{sys}}(\mathbf{Q}|_{Q_x:=1}) - \overline{Q}_{\mathrm{sys}}(\mathbf{Q}) \tag{92}$$

The component failure rate is not limited to a certain maximum value, and thus also the system failure rate is not limited (see formulas for fault trees in section 7). Hence, no Risk-Achievement can be defined for the system failure rate.

A.7 Risk Achievement Worth (RAW)

If the Risk Achievement is divided by the original system value, you get the factor, by which the system value would increase if the component will fail for sure:

$$I_{F,x}^{\mathrm{RAW}} = \frac{F_{\mathrm{sys}}\left(T, \mathbf{F}|_{F_x:=1}\right) - F_{\mathrm{sys}}(T, \mathbf{F})}{F_{\mathrm{sys}}(T, \mathbf{F})} = \frac{F_{\mathrm{sys}}\left(T, \mathbf{F}|_{F_x:=1}\right)}{F_{\mathrm{sys}}(T, \mathbf{F})} - 1 \tag{93}$$

or is always unavailable:

$$I_{Q,x}^{\mathrm{RAW}} = \frac{\overline{Q}_{\mathrm{sys}}\left(\mathbf{Q}|_{Q_x:=1}\right) - \overline{Q}_{\mathrm{sys}}(\mathbf{Q})}{\overline{Q}_{\mathrm{sys}}(\mathbf{Q})} = \frac{\overline{Q}_{\mathrm{sys}}\left(\mathbf{Q}|_{Q_x:=1}\right)}{\overline{Q}_{\mathrm{sys}}(\mathbf{Q})} - 1 \tag{94}$$

Note: In some definitions, the summand -1 is omitted.

As for the Risk Achievement, the RAW is not defined for failure rates.

A.8 Importancies for generic basic events

The above mentioned definitions are related to single *basic events*. If a system includes several components of the same type, a modification of the component (or an uncertainty of the statistic values) will influence several *basic events* simultaneously, but not only one. Therefore, Functional Safety Suite offers the calculation of the above mentioned importancies also for *generic basic events*: All *basic events* referring to the same *generic basic event* will be modified simultaneously.

The CRI related to *generic basic events* can be greater than 1.0, if the *generic basic event* is used in multiple conjuncted paths. For example for the simple fault tree shown in figure 83, the CRI for the *generic basic event* A $I_{F,A}^{\mathrm{CRI}}$ is 2.0.

If the failure rate of A increases by 1%, the system failure rate $\overline{h}_{\mathrm{sys}}$ will increase by 2%. But be careful: Whenever you've got redundant structures, the values of the*generic basic events* are non-linear in the system values. The CRI $I_{F,A}^{\mathrm{CRI}}$ is valid only for small changes, therefore. If you double the failure rate of *generic basic event* A, the system failure rate $\overline{h}_{\mathrm{sys}}$ will be four times as high.

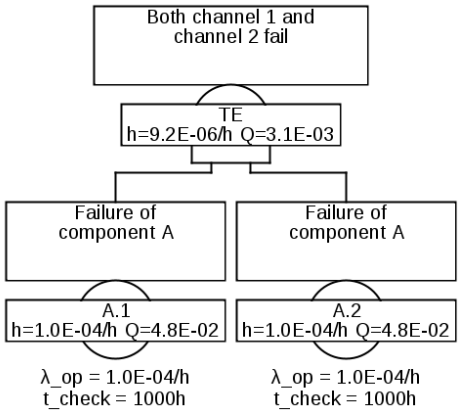


Figure 83: *Importance of generic basic events*

B Reports

With Functional Safety Suite it is possible to create professional reports within seconds (ok, the actual creation of the report might take some minutes or even hours if you have large models with transient evaluation and importancies calculation, but you can drink some coffee or watch TV or talk to your boss in the meantime).

Note: All described functionality has been tested with Microsoft Office 365, with German language settings. If you encounter any problems with other Microsoft Office/Microsoft Word versions or language settings, don't hesitate to send some problem report, including the .docx file that doesn't work as intended. If you want some company specific modification of the report template or some additional feature, also write to the email stated on the front page of this manual.

B.1 Create a Report

B.1.1 Automatic work

Click **Export – Create Report** directly after loading a project.

A dialog will appear, providing several options in order to adjust the content of the report to your needs, see figure 84.

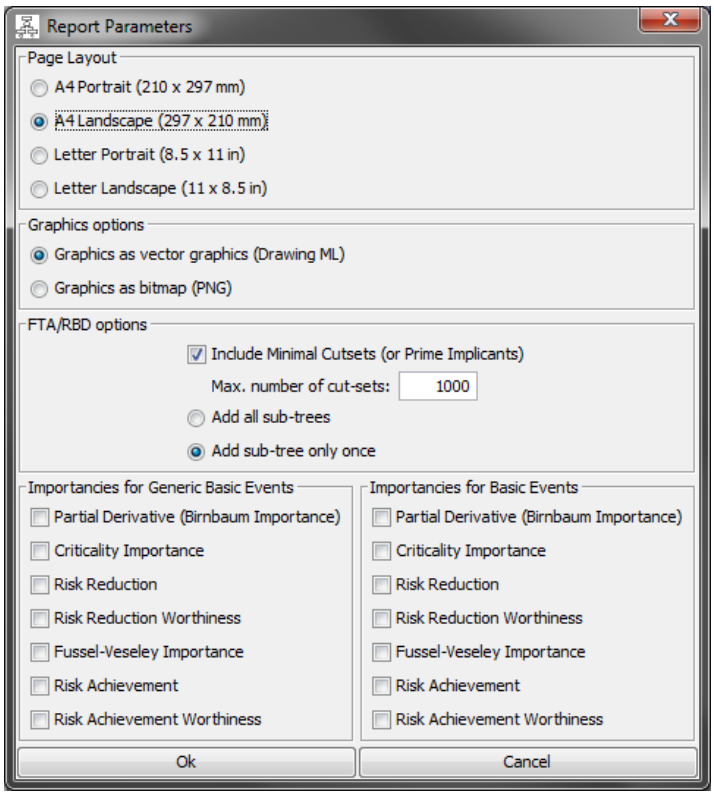


Figure 84: *The create report dialog*

B.1.1.1 Page Layout

Set the page layout. The corresponding template in the templates folder will be selected.

Note: You can modify the templates according to your needs to some extend. In particular, you can change the text formats, the header and footer, the page size and page margins, add some text etc.

B.1.1.2 Graphics options

Select if the graphics in the report shall be vector graphics (Drawing ML format) or pictures (bitmaps in PNG format). Vector graphics can be edited, but they are displayed correctly only in newer versions of Microsoft Word (Microsoft Office). Pictures cannot be edited, but they will be correctly shown even in older versions of Microsoft Word and some other programs.

B.1.1.3 FTA/RBD options

Select whether lists of the minimal cut-sets / prime implicants shall be included or not. The list will be sorted for importance, i. e. the minimal cut-sets with highest value will be the first in the list. The number of lines of the list will be limited to the number stated here, since it typically doesn't make sense to have list hundreds of pages.

Select whether a sub-tree, that is used by several top-level fault trees (or RBDs) is added only once at the end of the report, or if it shall be added to the section of each top-level fault tree that uses it. Note: If you decide to add sub-trees only once in the report, each sub-tree will be calculated separately using the evaluation parameters assigned to the sub-tree. If you decide to add separate sub-trees for each top-level fault tree section, the sub-trees will be evaluated using the evaluation parameters assignd to the top-tree.

B.1.1.4 Importancies

Select the importancies that shall be calculated for each (top) model. Note that importancies calculation might take quite some while, in particular in case of transient evaluation.

B.1.2 Post-work

After successful creation of the report (see status line and text output), you'll find the report file in the project directory. Its name consists of the project name, extended by date and time of its creation. Open the report file in Microsoft Word (Microsoft Office 365). Mark all via Ctrl+A and then press F9 in order to update all fields. Now you can complete the report. In particular, you might want to adapt the front page, the headers and footers, add references to the input documents that have been used for performing the analysis, add descriptions for each model, add a summary section etc.

Feel free to add text, figures or tables wherever you want, but in order to be able to update the report automatically (see next subsection), you should not do the following:

- Do not change the names of packages or models in the project.
- Do not change the predefined section numbering or section headings. You may add new sub-sections at the end of each section, and you may add new sections et the end of the report.
- Do not change the figure or table captions. In particular, the 'alternative text' defined for each figure and table must not be changed, because this is used as the unique ID to find the correct figure or table in case of an update. In order to prevent any duplicates of these IDs, do not copy and paste the automatically created figures or tables, not even as a template for manually added figures or tables.
- Do not change the content of a table. (You can, but it will get lost in case of an update, because the complete table will be replaced.)
- Do not delete any automatically created figures or tables, except if you never want to have them updated anymore.
- Do not change the page number format.

B.2 Update a Report

There is no project without changes. Thus, also the quantitative risk evaluation report will be subject to change. Often only minor changes need to be introduced, such as a corrected failure rate or a corrected fault tree.

Therefore Functional Safety Suite provides the function to update an existing report in case of typical changes by **Export – Update Report**. A dialog will ask you to select the existing report file that shall be used as the template for the update. A new report will be created, where all automatically created figures and tables will be replaced by new ones. The existing file will not be changed.

B.3 Important notes related to reports

1. If you define a large header or footer, don't forget to adjust the page margins in Microsoft Word accordingly. (depending on your particular version, you might find that under "Layout – Page Margins – User defined page margins – Page Margins – Top/Bottom"). Select "Apply to overall document", not only current section. If you don't increase the page margins sufficiently, the size of figures will not be correctly calculated so that you'd have to adjust them manually.
2. You can modify or even delete the section "Warning, Hints, Information". If you do an update, a new section will be added at the end of the document in any case.
3. The page number format 'section - page in section' has been selected in order to reliably predict the page number of the refered model while creating the figures, and to allow you to add text of any length or figures or tables at (more or less) any place in the document. Unfortunately, the page number format is not compatible with Libre Office.

4. You should create or update a report before any model has been calculated. If you don't do so, the warnings, hints and information section might not be complete (it will only contain the information of those models that haven't been calculated previously).

5. If you mark some event, state, condition etc. the marking will also be visible in the related figure(s) in the report. This might be intended or not.

6. If you've modified the report by other tools than Microsoft Word, the update will probably not work. In particular, Libre Office deletes the file "report_properties.xml" which is included in the '.docx' file while the report is created and which is necessary to update the report.

7. If you change the structure of the project, i. e. if you add or delete models, change the name of models or packages etc., the report might not be updated correctly. In that case, it might be necessary to create a new report as explained in section B.1, and copy all manually added text/graphics/tables to the new report manually. The same applies if you want to change any of the report options.

References

[NUREG] : *The Fault Tree Handbook*, NUREG-0492, US Nuclear Regulatory Commission (1981)

[NASA] : *Fault tree handbook with aerospace applications*, NASA (2002)

[SiRF] : *Schienenfahrzeuge, Sicherheits-Richtlinie Fahrzeuge*, Eisenbahn-Bundesamt, VDV, VDB, DB, Germany (2020)

[EN 50126] *Railway Applications – The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*, CENELEC (2017)

[EN 50128] : *Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*, CENELEC (2010)

[EN 50129] *Railway Applications – Communication, signalling and processing systems – Safety related electronic systems for signalling*, CENELEC (2017)

[EN 61025] : *Fault tree analysis*, IEC (2006)

[EN 61165] : *Application of Markov techniques*, IEC (2006)

[EN 61508] : *Functional safety of electrical/electronic/programmable electronic safety-related systems*, IEC (2010)

[EN 61508-1] : *Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 1: General requirements*, IEC (2010)

[EN 61508-2] : *Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 2: Requirements for electrical/electronic/probrammable electronic safety-related systems*, IEC (2010)

[EN 61508-3] : *Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 3: Software requirements*, IEC (2010)

[EN 61508-6] : *Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3*, IEC (2010)

[ISO 13849] : *Safety of machinery — Safety-related parts of control systems*, ISO (2015)

[ISO 26262-5] : *Road vehicles — Functional Safety — Part 5: Product development at the hardware level*, ISO (2011)

[Wikipedia] : *http://en.wikipedia.org/wiki/**