Virtual Satellite 4 FDIR

User Manual

Version 4.10.0, 2020-06-02T11:39:25Z

Table of Contents

Chapter 1. What is Virtual Satellite 4 FDIR?

This user manual describes how to use the Software Virtual Satellite 4 FDIR (VirSat4 FDIR). The software is an extension to Virtual Satellite 4 CORE, for a guide on how to use Virtual Satellite in general, please refer to the Virtual Satellite 4 CORE Manual.

Chapter 2. Purpose

VirSat4 FDIR focuses on modeling Fault Detection, Isolation, and Recovery. This includes among others: Models on failure behavior, recovery behavior, detection behavior, analysis results, and more. The main purpose of VirSat4 FDIR is to enable the analysis of FDIR concepts by means of mathematically well founded evaluation.

The software primarily follows the ESA ECSS standards by means of the SAVOIR FDIR Handbook (http://savoir.estec.esa.int/SAVOIRDocuments.htm). For clarification on vocabulary, further details on FDIR analysis and process, please check the SAVOIR FDIR Handbook. Accessing the handbook requires registration / login. Please note that this in return is restricted to ESA member states.

Chapter 3. Getting Started

3.1. Modeling Workflow

Learn in this section about the recommended workflow for approaching FDIR modeling & analysis. The workflow for creating fault models is up to the preference of the users. Nevertheless, basing the workflow on the following steps is recommended:

- Create a system model using the product structures concepts (PS Concept). For details check the Virtual Satellite 4 CORE User Manual.

- Create a SubSystem dedicated to Risk / FDIR.

- Create a list of Feared Events in this sub system and assign severity categories to them.

- Create a list of faults, with their basic events, for each equipment.

- Perform a Fault Tree Analysis to determine the fault propagations.

- Create the FDIRParameters category at the top of your system model and configure it.

- Create FDIR analysis categories for the desired faults

- Refine the system model and the Fault Tree Analysis and update the FDIR analysis

In the following sections, the various categories, means of analysis, etc. will be elaborated.

Chapter 4. Fault Modeling

Fault modeling forms the core of VirSat4 FDIR and is the primary activity required to perform any FDIR analysis. Learn in this section how to use the graphical editor to perform the main analysis of VirSat4 FDIR, Fault Tree Analysis (FTA), and how to use it to build up fault models.

4.1. Modeling Fault Trees

Faults, their propagations, and inhibiting fault propagation through means of FDIR is modeled using fault trees. Fault trees are graphical models describing how faults combine with each other, propagate through the system, and eventually turn into a feared event. A fault tree is usually constructed top-down, but is read bottom-up, starting with initiating basic events. The recombination of faults is modeled via so-called gates, such as "AND" and "OR". In this section, you will learn the basics on fault tree analysis, which gates are supported by the software, and how to use the graphical editor to create a fault tree. For further in-depth information on how to perform an FTA, we refer to the standards.

4.1.1. Fault Tree Elements

The following contains a comprehensive list of fault tree elements supported by VirSat FDIR. The element descriptions are structured in the following format:

***[Name of the fault tree element]* [Icon]**

— **Graphical representation**

[Representation in the graphical editor]

— **Description**

[Purpose of the element, its propagation behavior, additional parameters, etc.]

In addition to its inherent parameters, each fault tree node also has a name, a list of inputs, and a list of outputs. For gates, the name is by default the type of the node.

Fault 🔧

— **Graphical representation**
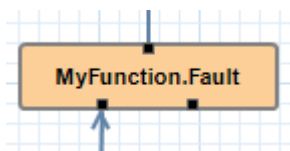
— **Description**

Faults represent logical, named events. They are used either to represent a *top-level event* of a fault tree or an *intermediate event*. Faults are also the logical containers for all other fault tree elements. As such, deleting a fault also deletes all contained elements such as gates, propagations, analysis information, etc. A fault always propagates if at least one input fails. It is, however, recommended to only have one input per fault (plus basic events).

BasicEvent ⬤

— **Graphical representation**
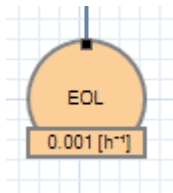
— **Description**

Basic events typically form the leaf elements of a fault tree. They represent basic anomalies that are not further broken down in the course of the fault tree analysis. In practice, basic events most commonly correspond to causes of equipment failure. A basic event is always directly associated to a fault. All fault propagations in a fault tree ultimately originate from basic events. A basic event supports the following additional properties:

- `failureRate`: At minimum, a basic event must have a failure rate for quantitative evaluation such as reliability analysis. The failure rate states how often the basic event is expected to occur within a time unit, quantifying its likelihood of occurrence over time.

- `repairRate`: Optionally, it may also be equipped with a repair rate, which conversely captures the likelihood of repair over time.

- `coldFailureRate`: A basic event may also be equipped with a cold failure rate, which comes into play when interacting together with the SPARE gate. It states the modified failure rate that is used when a basic event is dormant.

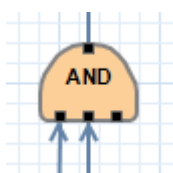## Propagation Ⅰ

— **Graphical representation**

— **Description**

Fault propagations are the edges of a fault tree, and connect the fault tree nodes. A fault propagation has a direction. It connects the output of a fault tree node with the input of another fault tree node. Since fault trees are acyclic graphs, fault propagations may not create any cycles.
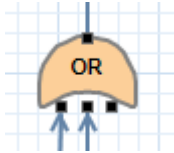
AND ⬟

— **Graphical representation**

— **Description**

A gate that propagates if all inputs have failed.

OR 🔶
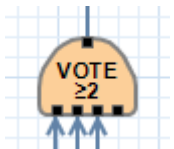
— **Graphical representation**

— **Description**

A gate that propagates if at least one input has failed.

VOTE 🔶

— **Graphical representation**

— **Description**

A gate that only propagates if a certain number of inputs have failed. A VOTE gate is equipped with a `votingThreshold` property, and propagates if at least `votingThreshold` many inputs have failed. The voting threshold has to be at least 1.

SPARE 🔶

— **Graphical representation**
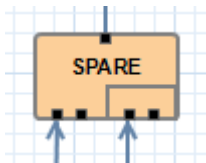
— **Description**

A gate with two types on inputs: Primaries and spares. If at least one primary input fails, the SPARE gate activates and claims one of the spares. Should no spares be available or failed, then the SPARE gate propagates. All spares are considered to be dormant. This means that contained basic events will use their cold failure rate, instead of their hot failure rate, as long as they are unclaimed. Once a spare is claimed, it is set to be activated and its hot failure rate is used again. Spares are claimed from left to right. In the case of a repair, the SPARE gate switches back. Spares may be shared between spare gates. However, there must not be common nodes between spares or between spares and primaries. The only exception of this rule, are functional dependency gates.

POR 🔶

— **Graphical representation**
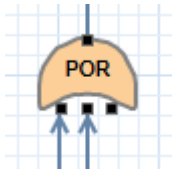
— **Description**

A Priority OR (POR) gate propagates if the left-most input occurs before any other input.

POR_I 🔶

— **Graphical representation**
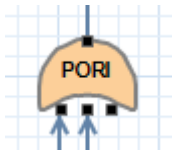
— **Description**

An Inclusive Priority OR (POR_I) gate propagates if the left-most input occurs before any other input, or at the same time as another input.

PAND 🔶

— **Graphical representation**
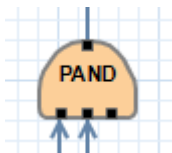
— **Description**

A Priority AND (PAND) gate propagates if the inputs fail exactly in sequence from left to right.

PAND_I 🔶

— **Graphical representation**
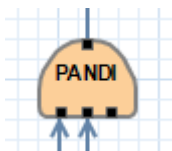
— **Description**

An Inclusive Priority AND (PAND_I) gate propagates if the inputs fail exactly in sequence from left to right, or at the same time.

SAND 🔶

— **Graphical representation**
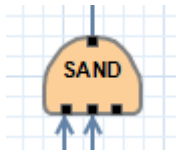
— **Description**

A Simultaneous AND (SAND) gate propagates if all inputs fail at the same time.

FDEP

— **Graphical representation**
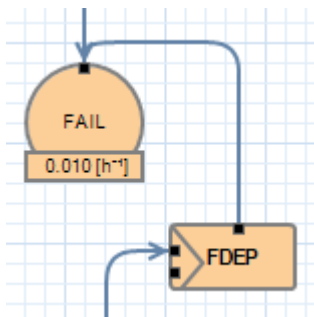
— **Description**

The functional dependency (FDEP) gate allows to trigger basic events. In the event of any input event occurring, all connected basic events get triggered.

RDEP

— **Graphical representation**
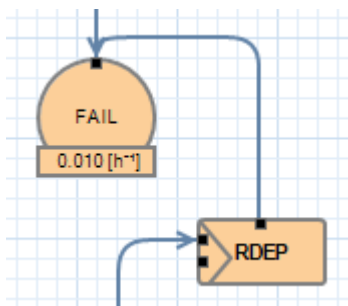
— **Description**

The rate dependency (RDEP) gate allows to increase the failure rate of a basic event. An RDEP is equipped with a rate change property `rateChange`. In the event of any input event occurring, the failure rate of all connected basic events is multiplied by `rateChange`.

PDEP

— **Graphical representation**
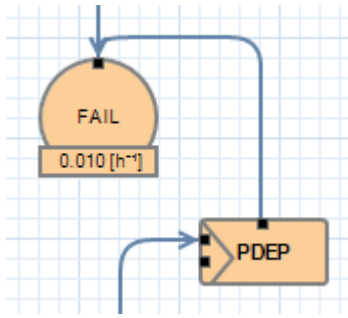
— **Description**

The probability dependency (PDEP) gate allows to trigger basic events. The PDEP gate is equipped with a trigger probability `probability`. In the event of any input event occurring, each connected basic event is triggered with probability `probability`. The PDEP propagation is checked every time an input fails.

DELAY 🔲

— **Graphical representation**
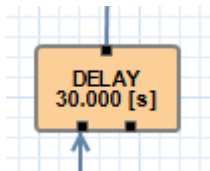
— **Description**

The DELAY gate can be used to describe time delays in propagation. The gate is equipped with a delay parameter `delay`. Propagation occurs if any input fails and remains failed for a duration of `delay` time units. If the failed inputs are repaired before the DELAY gate performs a propagation, then the propagation process is stopped.

MONITOR 🔲

— **Graphical representation**
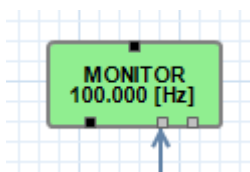
— **Description**

The MONITOR gate is used in fault trees where not all events are observable. If a monitor gate is used, the semantics of fault occurrence are changed as follows: By default, the observation of a basic event or any further propagated gate, is not guaranteed. In order for it to be observed it must propagate to an observation input of a MONITOR gate. A MONITOR gate has two types of inputs: Fail inputs and observation inputs. Fail inputs work as usual, in the event of any fail input occurring, the MONITOR gate fails as well and propagates. Failed MONITOR gates may no longer perform any observations. In the case of an observation input occurring, the event is marked as observed. Only then can reactive gates such as SPARE gates react and claim a spare. The recovery actions in partial observable fault trees are managed by recovery

automata. A MONITOR gate is also equipped with an `observationRate` property. In the event of it being 0, all observation events are observed immediately. If `observationRate` is non-zero, all observation events are observed with the time delay given by the inverse of `observationRate`.

4.1.2. The Graphical Fault Tree Editor

Besides the usual table based user interface, VirSat FDIR offers a graphical diagram editor interface. Learn in this section how to create new fault tree diagrams and how to use them for building fault tree models. Fault tree diagrams can also be used to simply visualize existing fault tree models.

Creating a New Fault Tree Diagram

Each fault tree diagram is associated with a fault. The fault locally constitutes a top-level event. Note that on a system wide level, the fault might be just an intermediate event. A diagram may also contain multiple top-level events, but it is not recommended, as all elements inserted into a fault tree diagram are automatically associated with the corresponding fault corresponding to the diagram.

A new diagram can be created by selecting a fault in the navigator and then choosing **Context › Open Diagram Editor**. A new editor window named according to the UUID of the selected fault will pop up and also automatically contain the selected fault. The diagram editor can be opened again using the same process. The file of the newly created diagram can be found in the *documents* folder of the structural element instance the fault is attached to.
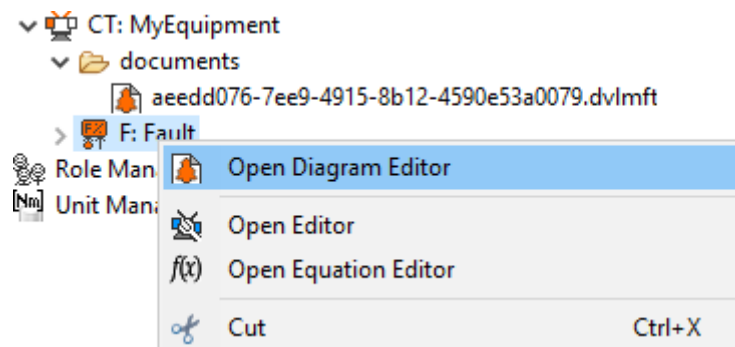
Basic Usage

This section introduces the basic concepts needed to operate the diagram editor. The diagram editor is based on the same technology as other Virtual Satellite diagrams. Previous experience with Virtual Satellite diagram editors should at least partially translate. The diagram editor consists of two main areas shown in the figure below: The actual modeling canvas (left-hand side) and the modeling palette (right-hand side).
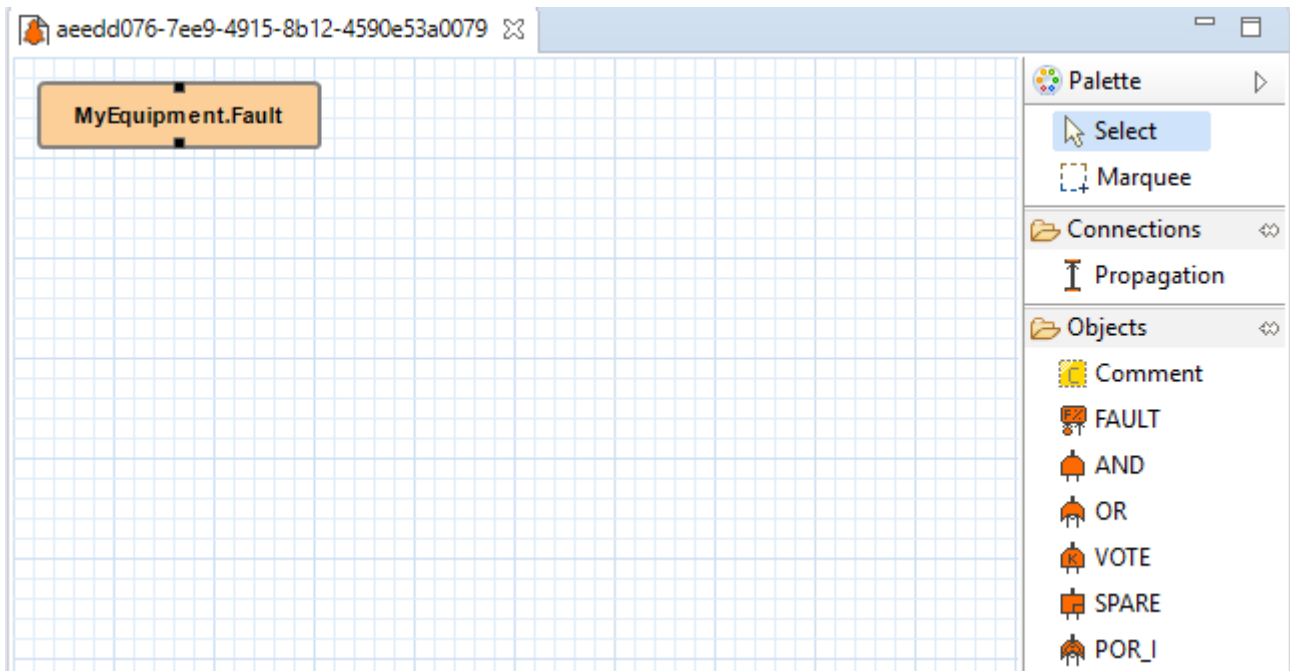
Elements can be placed in the canvas by selecting an element type in the palette, and then left-clicking on the canvas. Alternatively, existing elements can be added to the canvas by drag & dropping them from the navigator view. This is useful for example, for referencing other, already existing faults. For further documentation on the editoring capabilities provided by the underlying technology, please consult the official documentation available at https://www.eclipse.org/graphiti/documentation/.

Connecting elements

Each element in the canvas is equipped with input and output ports. These can be connected via the propagation element. The editor only allows creating propagations that start in an input port and end in an output port. Propagations can also be reconnected by selecting the desired port and moving the propagation end to the new desired port. However, the same restrictions apply as when creating new propagations. If a port is used, a new port of the respective type will be generated. Likewise, if a used port becomes unused, other free ports of the type are automatically deleted. Deleting an element with incoming and outgoing propagations also deletes those propagations, if the user has the necessary rights to delete them.

Finally, it is also possible to insert into a propagation. This redirects the propagation to end in the input port of the newly added element. Additionally, a new propagation is created connecting the output of the newly created element to the input of the old end of the propagation.

Diagram Specific Features

In addition to the normal diagram and modeling functionalities, the fault tree diagram offers some additional features. Their usage and functions are explained in this section.

Comment

— **Description**

A comment contains a simple multi-line text. Line breaking can be achieved by pressing `SHIFT`+`ENTER`. It can be used to add clarity to the model, describe the purpose behind potentially difficult to understand fault tree constructs, etc. Unlike other elements in the

palette, comments do not have a model representation in Virtual Satellite. They only exist in the diagram. This also means, that if a diagram is deleted, then so are the contained comments.

— **Graphical representation**
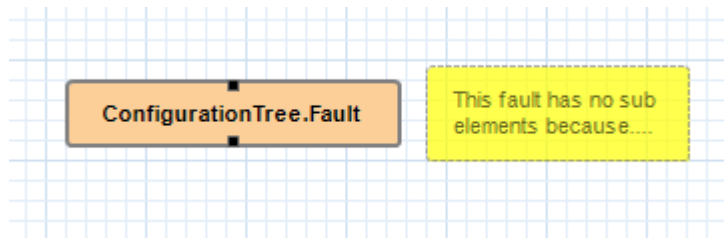
Collapse and Expand

— **Description**

The collapse and expand operations aim to simplify the management of the logical level of detail in a fault tree diagram. They are available only for faults. Gates cannot be expanded or collapsed. Expanding a fault means to add all directly contained fault tree elements into the diagram. Likewise, collapsing a fault removes all elements directly contained in the fault from the diagram. If a fault is not fully expanded, then the expand operation is shown in the user interface. If a fault is fully expanded, the collapse operation is shown. Performing either of the operations leads to a automatic layouting of the fault tree diagram.

— **Graphical representation**
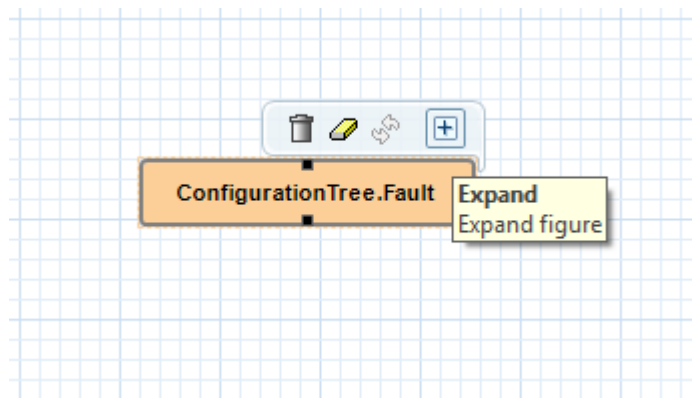
Using the Auto Layout functionality

The fault tree editor provides the functionality to auto layout diagrams. Auto layouting them makes it so that the top-level elements are located at the top, and lower level elements are located further to the bottom. The auto layout can be invoked by pressing the auto layout bottom located at the top-right above the navigator.
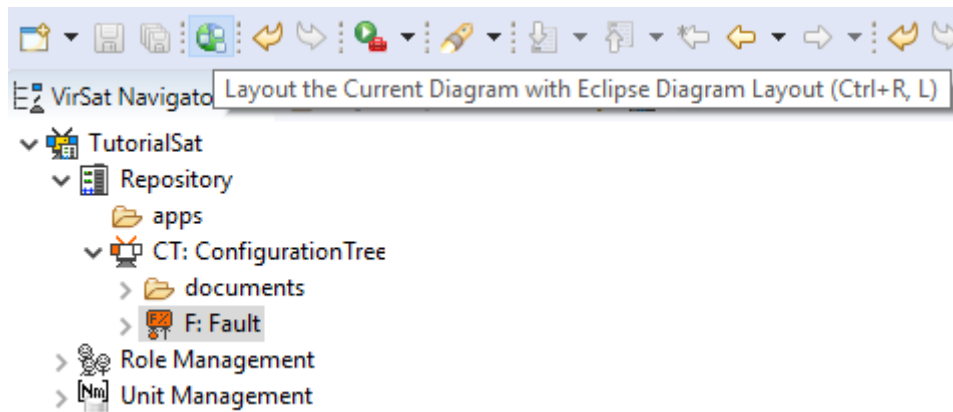
For advanced users, it is possible to customize the layouting. This can be done by editing the layouting properties accessible via the layout view. The layout view is opened by right-clicking on the diagram canvas and selecting **Show Layout View**.
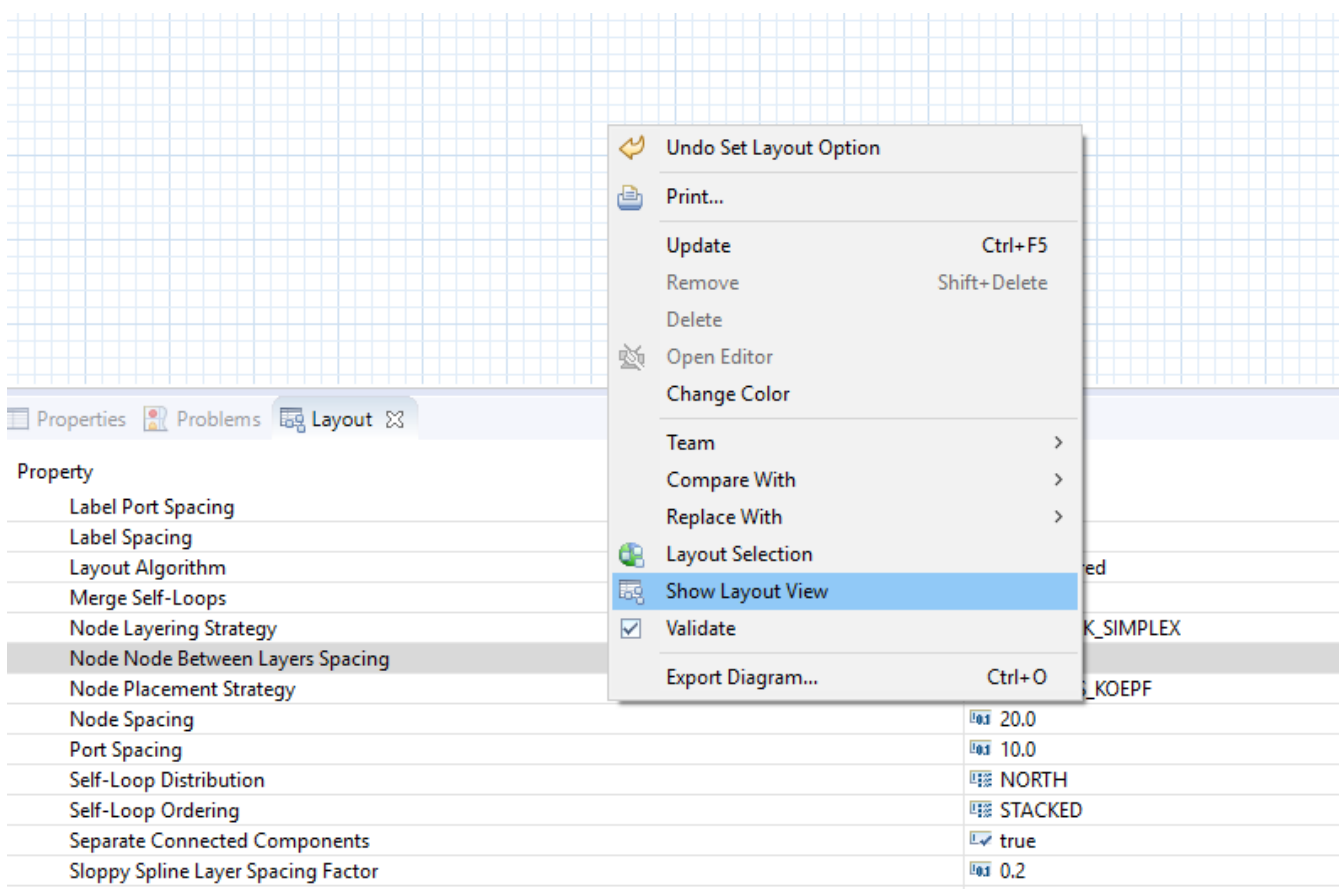
For example, the default minimum distance between two node levels can be adjusted via the **Node Node Between Layers Spacing** property. For further documentation regarding the auto layout functionality in general, and the available customization options, please refer to the official documentation available at https://www.eclipse.org/elk/reference.html.

4.2. Modeling Detection

Fault trees by default are considered to be fully observable. This means, that e.g. SPARE gates can immediately react to the occurrence of basic events. Adding a MONITOR gate to a fault tree turns the model into a partial observable fault tree. In this model, only events that are directly linked to the observation input for a MONITOR gate, or that can be logically derived, can be observed. For

example, if all inputs to an AND gate are observed, then the AND gate is observable as well. Using MONITOR gates is absolutely necessary, if the user wishes to perform analysis on the observation behavior and times.

Partial observable fault trees with SPARE gates require a recovery automaton to be defined, otherwise their behavior is undefined. On the other hand, partial observable fault trees with static gates only (e.g. AND, OR, VOTE, etc.) do not require a recovery automaton to be specified.

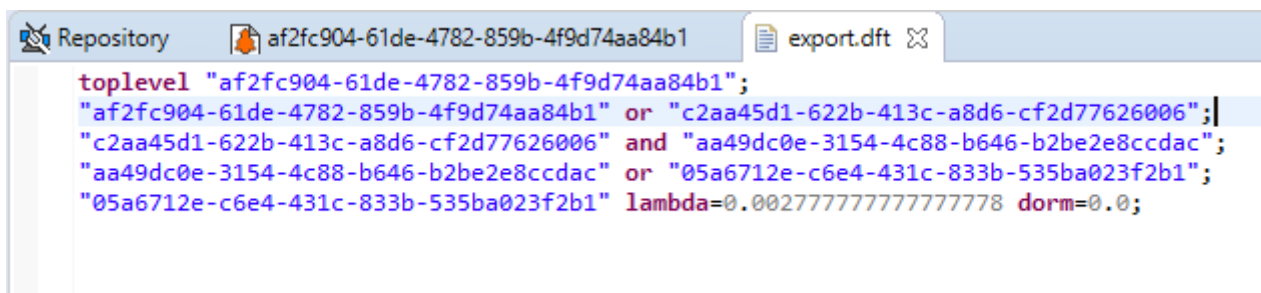# 4.3. Data Exchange With the GALILEO Format

Connecting external fault trees with Virtual Satellite trees is possible is possible with the GALILEO file format. Virtual Satellite can import and export fault trees into this textual format, extended by the node types supported in Virtual Satellite. The GALILEO file format is a simple fault tree format, and further descriptions on its syntax can be found at https://www.cse.msu.edu/~cse870/Materials/FaultTolerant/manual-galileo.htm. Exporting and importing a fault tree can be done using the **Galileo DFT Export** and **Galileo DFT Import** wizard, respectively. The wizards are available under **File › Export › FDIR** and **File › Import › FDIR**.

For identification, the exchange uses the UUIDs as identifiers. This means that importing a GALILEO fault tree with some specified names will create appropriately named fault trees in Virtual Satellite. However, when re-exporting the fault tree to the GALILEO format, Virtual Satellite will use the UUIDs, giving a different output. Should the user decide to modify the fault tree and re-import, Virtual Satellite can identify existing fault tree elements via the UUIDs. VirSat FDIR also ships with a very simple GALILEO file format textual editor meant for simple viewing and editing of GALILEO fault trees. It is automatically used when opening a file with the *.dft* extension. And example of the export output created by VirSat and viewed with the textual editor is given below.

Chapter 5. Recovery Modeling

Recovery behavior can be modeled using **recovery automata**. They specify what recovery actions should be executed upon occurrence of a fault. Recovery automata are necessary when dealing with fault trees with complex recovery behavior. This includes especially partial observable fault trees with MONITOR gates. Learn in this section how the recovery automaton model is structured, and how to manage it.

5.1. Modeling Recovery Automata

A recovery automaton (RA) is a finite, deterministic state machine without timed transitions. An RA contains is constituted of the following properties:

- `states`: States represent a recovery internal state of knowledge.
- `initial`: Each RA must have an initial state specified.
- `transitions`: Transitions state the actual recovery behavior. They are labeled with a list of **guards** that states the condition that triggers the transition and a list of recovery actions, which are then executed. RAs operate under maximum progress assumption, i.e., whenever a transition is enabled it must be taken.

5.1.1. Recovery Automata Elements

The following section gives a more in-depth description of the available elements for creating recovery models. It is structured similarly to the description of the fault tree elements using the format:

[Name of the recovery automaton element] **[Icon]**

 — **Graphical representation**

  [Representation in the graphical editor]

 — **Description**

  [Purpose of the element, its behavior, additional parameters, etc.]

If there is no applicable graphic representation, the item is left out. Like all Virtual Satellite elements, in addition to its parameters, every recovery element also has a name.

State ![icon]
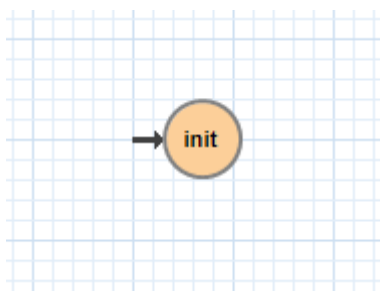
 — **Graphical representation**

A state is recovery internal information. Changing a state changes the recovery behavior. If possible, states should be given meaningful names. Every recovery automaton has an initialState property.

FaultEventTransition ⊏↓

— **Graphical representation**
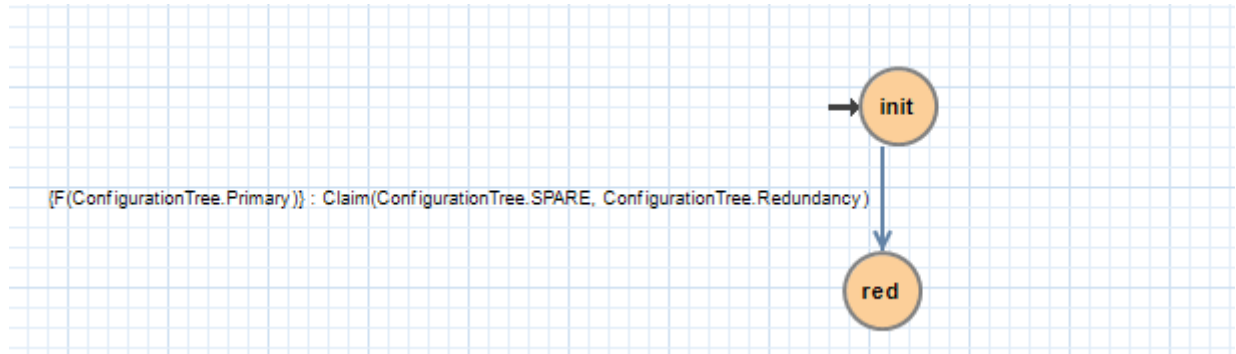
— **Description**

A fault event transition is a transition that has fault tree events listed in the guards. That means, it is triggered if exactly the specified nodes in the fault tree fail at the same time. As a transition, is is also equipped with a list of recovery actions, executed upon triggering the transition.

TimeoutTransition ⊤↓

— **Graphical representation**
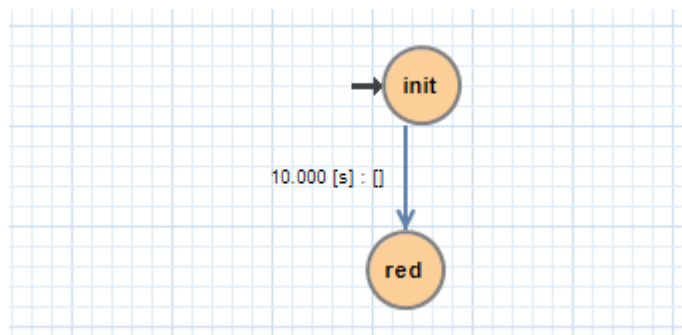
— **Description**

A timeout transition has a `time` property. It is used as the guard for the transition. As a transition, is is also equipped with a list of recovery actions, executed upon triggering the transition. Every state may be equipped with at most **one** timeout transition. If multiple are declared, only the one with the smallest time property is used due to the maximum progress assumption.

FreeAction F■

— **Description**

The free action is a recovery action, describing that all claims on a specified spare should be cleared.

ClaimAction ⬛

— **Description**

The claim action is a recovery action, describing that some SPARE gate should claim the specified spare.

5.1.2. The Graphical Recovery Automaton Editor

Recovery automata can be visualized and graphically modified using the recovery automaton diagram editor. The editor operates similarly to the fault tree diagram editor. The following sections thus refer to the fault tree diagram sections, if applicable, and otherwise focuses on the aspects where the diagram editor usage differs.

Each RA diagram is associated to exactly one recovery automaton, identified by the UUID in the diagram. The recovery automaton object itself is not represented in the diagram. Instead, it only contains the items contained by the RA, i.e., states and transitions.

Creating a New Recovery Automaton Diagram

The process to creating a new recovery automaton diagram is analogous to creating a fault tree diagram. However, here, the user selects a RecoveryAutomaton object and then proceeds as usual by choosing **Context › Open Diagram Editor**.

Basic Usage

The basic usage of the recovery automaton diagram editor is similar to the fault tree diagram editor. States can be directly connected via Transitions. To set or unset the initial state, select a state and after opening the context menu via right click, tick or untick the box in **Context › initial state**. An initial state is marked by an incoming arrow from the left (see the graphical representation of states for reference).

Using the Auto Layout functionality

Auto layouting is also enabled for recovery automata diagrams.

Chapter 6. FDIR Analysis

Once faults model and optionally recovery models have been created, they can be directly evaluated within VirSat FDIR using the available analysis procedures. There a main two types for analysis procedures.

Those which primarily focus on non-numerical information such as: * Are all single points of failures covered? * Which combinations of faults lead to a failure? are referred to as *qualitatie analysis*. In VirSat FDIR most qualitative analysis procedures also supply additional useful information. For example, for fault combinations, VirSat FDIR also computes the mean time to failure (MTTF). Qualitative analysis procedures are typically added to structural element instances.

The second type focuses on numerical information such as: * What is the reliability after time t? * What is the mean time to failure? * What is the expected time until observation of a failure? This type of analysis is referred to as *quantitative analysis*. Quantitative analysis procedures can be added to faults.

Learn in this section how to setup analysis procedures, configure them, and how read their results.

6.1. Configuring Analysis information

Contextual information required by analysis procedures must be provided on the root level of the model. The **FDIRParameters** category allows the user to define the necessary contextual parameters. It provides the following properties:

- `missionTime` gives the time frame of analysis. All analysis procedures requiring this parameter have by default a link to this property.

- `timestep` determines the time granularity of the analysis output. E.g. is it desired to know the current reliability every day, month, year, etc. All analysis procedures requiring this parameter have by default a link to this property. Note that this value should be chosen sensibly large. Choosing a `timestep` of a second for a multi year mission means having several millions points of analysis in the model. This might not be desireable.

- `probabilityLevels` define what is considered a probable or unlikely event. The default levels are taken from the standard ECSS-Q-ST-30-02C.

- `detectionLevels` define what is considered to be a likely detected event or an unlikely detected event. The default levels are taken from the standard ECSS-Q-ST-30-02C.

- `criticalityMatrices` define whether triples consisting of probability level, a detection level, and a severity level is considered critical. The default criticality matrices are taken from the standard ECSS-Q-ST-30-02C. A criticality matrix is defined for each detection level. An entry in a criticality matrix is called a criticality level. It is determined by the product `probabilityLevel * severityLevel * detectionLevel`. Each criticality level can be edited using a provided table. Select an entry in the table and select `true` if it defined to be critical. Set it to `false` if the entry is not critical. Critical triples are marked with orange, non-critical triples are marked with green.

Severity	1 - ExtremelyRemote	2 - Remote	3 - Occasional	4 - Probable
4 - Catastrophic	4	8	12	16
3 - Critical	3	6	9	12
2 - Major	2	false ⌄	6	8
1 - Minor	1	false / true	3	4

i Some analysis information such as `missionTime` and `timestep` can be later overridden at the individual analysis level.

i Criticality matrices for fault trees without partial observability only require the criticality matrix for the detection level `VeryLikely`. The remaining criticality matrices can then be ignored.

6.2. Qualitative Analysis

VirSat FDIR provides the qualitative analysis procedures described in the following. They can be attached to any structural element instance.

FMECA

Failure Modes and Effects Analysis (FMECA) interprets the fault model in the way that any basic event can cause the top level event. AND, SPARE, etc. gates, locally defined repair actions are interpreted as compensation. In order to generate an FMECA, attach the FMECA category to the desired structural element instance (e.g. the FDIR subsystem), and press the **Perform Analysis** button. A list of FMECAEntries will then be generated. An FMECAEntry has the following properties:

- `failure` is the local top-level failure event.
- `failureMode` is the event that can cause the failure.
- `failureCause` is the event that can cause the `failureMode`. Might not be applicable if `failureMode` is a basic event without functional dependencies.
- `failureEffects` are the events that can be caused as a consequence of the occurrence o f the `failure` event. Might not be applicable if `failure` is globally a top-level event.
- `severity` is the severity level as defined in the `failure` event.
- `probability` is the probability level as defined in the FDIRParameters.
- `criticality` is the criticality level defined by the product of `severity * probability`.
- `meanTimeToFailure` is the MTTF of the `failure` event when only considering events in the sub-trees of to the `failureCause` (or `failureMode` if there is no `failureCause`). The inverse of this property is used for classification of the probability level.
- `compensation` are any defined fault tree nodes or properties that inhibit or fix the fault propagation.

MCSAnalysis

A cut set is a set of basic events, that causes a top-level event. A minimum cut set (MCS) is a cut set, where removing any element does not cause the top-level element to fail. An MCSAnalysis can be attached to a structural element instance, and computes all MCS that cause any of the top-level events attached to the same structural element instance. The analysis is executed by pressing the **Perform Analysis** button. An MCSAnalysis has the following properties:

- `maxMinimumCutSetSize` defines the maximum MCS size that should be considered. If 0 or no value is defined, then the maximum size is not restricted.

- `faultTolerance` is the number of basic events that need to occur for any failure to occur. It is determined by the size of the smallest MCS determined by the analysis minus one. This field is filled by the analysis.

- `minimumCutSets` are the computed MCS. Each MCS in return has the following properties:

  - `failure` the local top-level failure event caused by this MCS.

  - `basicEvents` the basic events that cause the `failure` event.

  - `severity` the severity level as defined in the `failure` event.

  - `probability` the probability level as defined in the FDIRParameters.

  - `detection` is the detection level as defined in the FDIRParameters.

  - `criticality` is the criticality level defined by the product of `severity * detection * probability`.

  - `meanTimeToFailure` is the MTTF of the `failure` event when only considering the `basicEvents`. The inverse of this property is used for classification of the probability level.

  - `steadyStateDetectability` is the long-term detectability of the `failure` event.

  - `meanTimeToDetection` is the mean time that passes between the occurrence of the `failure` event and the detection of the `failure` event.

> ℹ️ The number of MCS can increase exponentially with the number of basic events in the fault tree. It is therefore recommended to set the `maxMinimumCutSetSize` to a value of interest to the analyst. For example, if the MCSAnalysis is carried out to perform a Double Failure Analysis (i.e. only combinations of two basic events), then the `maxMinimumCutSetSize` can be set to 2.

6.3. Quantitative Analysis

VirSat FDIR provides the quantitative analysis procedures described in the following. They can be attached to any fault.

ReliabilityAnalysis

The reliability analysis computes metrics for judging reliability. It is executed with the **Perform Analysis** button.

- `remainingMissionTime` is the duration time frame of the analysis. It is by default set to the missionTime defined in the FDIRParameters.

- `timestep` is the granularity of the anaylsis. A reliability value will be computed for each `timestep`.

- `reliability` is the probability that the fault has not occurred once after time `remainingMissionTime`.

- `meanTimeToFailure` is the expected amount of time until the fault occurs.

- `reliabilityCurve` is a list of reliability values. A point is generated for each `timestep` up to the `remainingMissionTime`.

AvailabilityAnalysis

The availability analysis computes metrics for judging availability. It is executed with the **Perform Analysis** button.

- `remainingMissionTime` is the duration time frame of the analysis. It is by default set to the missionTime defined in the FDIRParameters.

- `timestep` is the granularity of the anaylsis. An availability value will be computed for each `timestep`.

- `availability` is the probability that the fault has currently failed at time `remainingMissionTime`.

- `steadyStateAvailability` is the long-term availability.

- `availabilityCurve` is a list of availability values. A point is generated for each `timestep` up to the `remainingMissionTime`.

ObservabilityAnalysis

The observability analysis computes metrics for judging observability. It is executed with the **Perform Analysis** button.

- `remainingMissionTime` is the duration time frame of the analysis. It is by default set to the missionTime defined in the FDIRParameters.

- `timestep` is the granularity of the anaylsis. An availability value will be computed for each `timestep`.

- `detectability` is the probability that the fault has currently occurred and is currently observed at time `remainingMissionTime`.

- `steadyStateDetectability` is the long-term detectability.

- `detectabilityCurve` is a list of detectability values. A point is generated for each `timestep` up to the `remainingMissionTime`.

6.4. Using STORM

VirSat FDIR ships with its own native model checking engine for performing anaylsis. However, for highly complex models it might be desirable to use a dedicated model checking engine. Natively, VirSat FDIR supports the STORM model checker (http://www.stormchecker.org/) as analysis engine. The analysis engine can be configured in the FDIR preferences page. It can be accessed as follows:

- Open the Eclipse Preferences Page via **Window › Preferences**.

- Go to **Virtual Satellite › FDIR**.

The engine configuration has the following properties:

- `Execution Engine` determines the model checking engine. It has the following options:

  - `Custom` is the default engine. Here, the native model checking engine is used. No further setup is required.

  - `STORM DFT` is the fault tree model checker of STORM. The fault tree is passed to STORM and analysed there. Requires a setup STORM execution environment.

  - `Custom + STORM` uses native algorithms for pre-processing fault trees, and turning them into mathematical models (Markov Chains or Markov Automata). The mathematical models are passed to STORM and analysed there. Requires a setup STORM execution environment.

- `Execution Environment` determines the STORM execution environment, if an execution engine requiring STORM is selected. The following options are available:

  - `Docker` STORM will be downloaded as a docker image and started within a docker container. Requires an active docker installation.

  - `Local` STORM will be locally executed. Requires a local installation of STORM.

Chapter 7. FDIR Reporting

While in some cases it might be more desirable to use the model itself as a means of communicating the FDIR design, sometimes it is of interest to share the model in a textual form. This might be especially interesting for those who do not have access to the model, do not have VirSat FDIR, or simply because customers require such documentation as a deliverable. VirSat FDIR provides some means to generate reports from the model that can be shared with collaborators, customers, etc. Learn in this section how to use the report generation and what its limits and capabilities are.

7.1. Using the SAVOIR/FDIR Report Template

VirSat FDIR supports the generation of an FDIR document. The document itself can be a pdf, excel sheet, HTML page, etc.

### 7.1.1. Creating and Using the Report Template

Learn in this section how to use the SAVOIR/FDIR Report Template in your project, and how to use it so that in the end a report is generated. The report generation happens in three steps:

1. First, the user needs to add or create a reporting template. VirSat FDIR by default provides a SAVOIR/FDIR compliant FDIR reporting template. The template can be added by first selecting the desired project where it should be added to, and then choosing **Apps › SAVOIR/FDIR Report**.
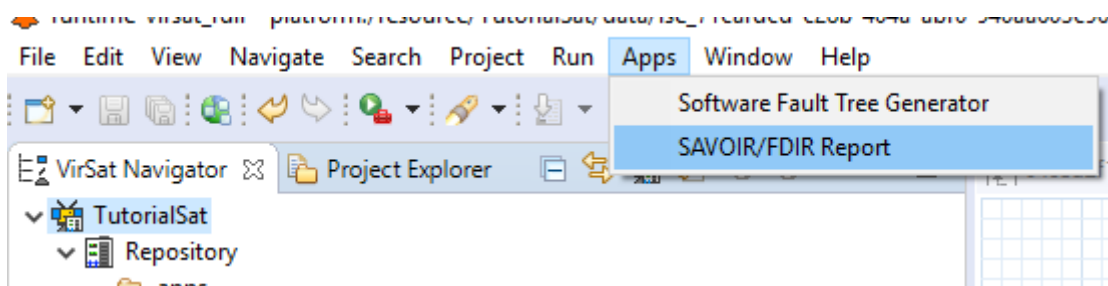

    Since report templates are handled as Apps, the user must first activate the usage of apps for the project! Please refer to the Virtual Satellite CORE Manual on how to activate Apps. After creating the reporting template, it can be found in the *apps* folder.

2. Then, the user can configure a report template. Here, items such as logos, additional desired text elements, etc. can be added or modified to customize the report for the project. Also, the user can define here which model in the repository should be the target for reporting. Editing a reporting template can be done by switching to the Report Design perspective via **Open Perspective › Report Design**. If the user desires to perform greater changes to the template, we refer to the official report designer documentation available at https://www.eclipse.org/birt/documentation/tutorial/. Setting the model to report on can be done by editing the **configuration** parameter. To edit it, go to **Report Design › Data Explorer › Report parameters › configuration**, and then double click the **configuration** parameter.
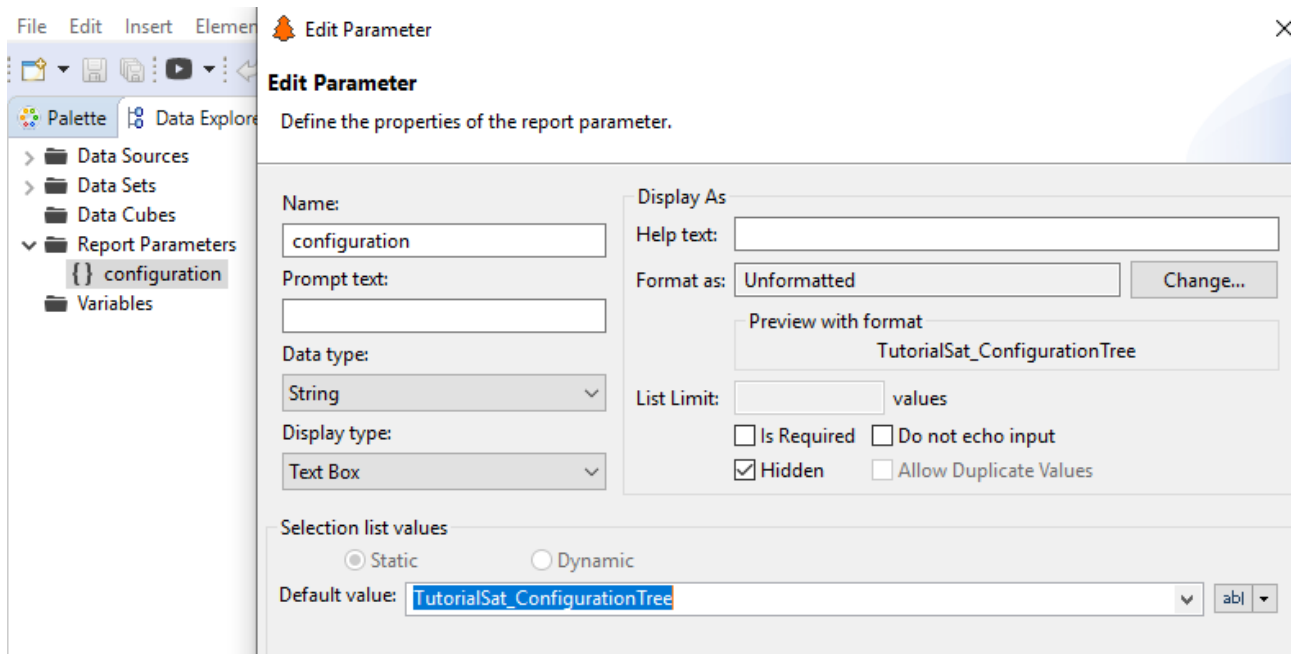
In the window, change the **Default value** option to the name of the tree that should be the target of the report.

3. Finally, the user can view the report using the **View Report** button, and selecting an export format.
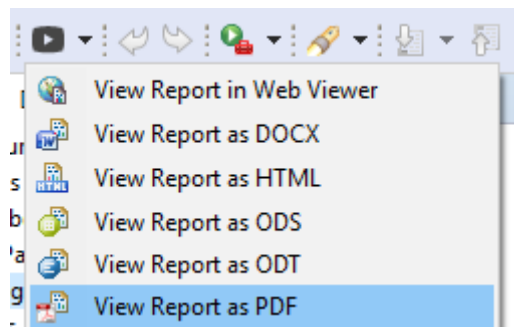
A web browser window will be opened and the user is given the option to open or save the generated report.

7.1.2. Contents of the Report Template

Learn in this section how the SAVOIR/FDIR Template is structured. Remember that by default all entries are filled automatically, but it is possible to edit them in case the auto filling proves to be undesirable for the current use case. The template has the following structure:

Header

The header is labelled as **FDIR Report**. The sub-header is labelled with the name of the project and the model element name of the report target.

Document Identification

The document identification loggs the meta data of the document. It has the following entries:

- `Project` the name of the project

- `Timestamp` the time and date of generating the document

- `Generated By` the name of the logged in user that generated the report

System Configuration

The system configuration gives a rough overview of the system considered in the report, by listing the system elements in a tree-like fasion. Each system element has the following attributes:

- `No.` the number in the listing

- `Name` the name of the system element

- `Responsible` the name of the discipline responsible for the system element

FDIR Configuration

The FDIR configuration prints the FDIRParameters category.

FMECA Report

The FMECA report prints each FMECA attached to any system element.

Reliability Prediction Report

The reliability prediction report prints each reliability analysis attached to any fault.

7.2. Excel Exports

In addition to the FDIR report, individual excel listings can be exported using the generic VirSat export functionality. The function is supported for any table. To use it, navigate to a table in an editor, and then press the **Export to Excel** button.

Chapter 8. FDIR Synthesis

VirSat FDIR offers a couple of services for generating model artifacts. Learn in this section how these services can be used, what they can do for you, and what their limitations are.

⚠ All features in this section are of experimental research nature! It is not recommended to use them in production if the accompanied risks are not acceptable!

8.1. Fault Tree Generation App

The fault tree generation app can generate complex fault tree models. It takes as input a simple model called the *bare* model, which purely models the fault relations. Also, it requires a set of *FDIR service* models. Each FDIR service model is a fault tree model, modeling how the propagation of a fault can be inhibited by the FDIR service. Each FDIR service must be defined within an ElementDefinition from the *Product Structures* concept. Furthermore, the user must supply an *FDIR configuration* via a dialog. Here, he can choose which service should be applied to which fault.

Out of all the given inputs, the fault tree generator will create a model, that extends the bare model with the FDIR service calls. Effectively, this means that the generated fault tree models the complete fault model with the logic FDIR services inhibiting fault propagation, and also failing to do so.

In order to add the app to your project, select your project and then **Apps › Software Fault Tree Generator**. Note that apps must be enabled your four project. Add the requested dependencies to your project and then you can call the app by executing *FaultTreeWithFDIRMechanismGenerator*.

8.2. Recovery Automata Synthesis

While recovery automata may be designed by hand, it is also possible to generate recovery automata which are optimal with respect to maximizing the system mean time to failure. The user must have prepared a fault tree model in order to use the recovery automaton synthesis. To call the synthesis procedure, go to a structural element instance (e.g. the FDIR subsystem), and in the section **FDIR Generation** press the **Generate Recovery Automaton** button. A new recovery automaton will be created and attached to the same structural element instance.

Details on how the synthesis algorithm operates can be found at the following locations:

- https://elib.dlr.de/123787/
- https://elib.dlr.de/126675/

Legal - License & Copyright

Product Version:	4.10.0
Build Date Qualifier:	2020-06-02T11:39:25Z
Travis CI Job Number:	