



Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

C O M P A S S

Static Fault Analysis Support - Technical Manual

Technical Note Number: D33.3b

Version: 1.1

Date: June 2014

Public Document

<http://www.compass-research.eu>

Contributors:

Zoe Andrews, NCL
Richard Payne, NCL
Alexander Romanovsky, NCL
André Dider, UFPE
Alexandre Mota, UFPE

Editors:

Zoe Andrews, NCL
Richard Payne, NCL

Reviewers:

Alexandre Mota, UFPE
Joey Coleman, AU
Uwe Schulze, UB
Jan Peleska, UB

Document History

Ver	Date	Author	Description
0.01	18-06-2013	Richard Payne	Initial document version
0.02	28-04-2014	Richard Payne	Added initial introduction, HH profile, related work
0.03	08-05-2014	Richard Payne	Material added on Fault Analysis tool
0.04	16-05-2014	André Didier	Material added on Fault Tolerance Plugin, updated introduction and conclusion accordingly
0.05	12-06-2014	André Didier	Modified after reviewers' comments
1.0	25-06-2014	Zoe Andrews	Final version
1.1	07-07-2014	Richard Payne	Minor corrections and removed line numbers

Summary

Work Package 33 delivers static analysis tool support for reasoning in SysML and CML. This deliverable forms the documentation for Task 3.3.3 – static fault analysis. Deliverable D33.3 forms two parts: executable code and documentation.

The documentation is provided in two documents. This document, D33.3b, is the second part; the technical details of the static fault analysis support. D33.3b provides details on: background material on static fault analysis; the development of static fault analysis support for SysML; the SysML profile developed for static fault analysis; the tool development for providing static fault analysis support (via HiP-HOPS) in Artisan Studio; and the tool development for providing fault tolerance verification in Symphony.

This part of the deliverable highlights the main achievements of Task 3.3.3:

- The Fault Analysis profile¹ enables SoS engineers to develop fault analysis models in SysML, allowing SoS models to be developed with failure data and propagation logic decoration on the constituent parts of the SoS.
- Guidelines that describe how the fault modelling approach detailed in D24.2 can be used as input for the Fault Analysis Models and their analysis in HiP-HOPS.
- A SysML to XML mapping and translation provides the ability to analyse the Fault Analysis Model using the Fault Analysis Tool (via HiP-HOPS).
- Ergonomic profiling in Artisan Studio facilitates the definition of Fault Analysis Models in the SysML toolset and provides basic static checks on the model.
- A SysML to CML mapping extension provides the ability to translate models (as per D24.2) of fault-tolerant SoSs from SysML into CML with a Symphony plugin specifically designed for fault tolerance verification.

The first part of the Deliverable, D33.3a - User Manual, provides details on obtaining and installing the Fault Analysis Tool for SysML and fault tolerance verification for CML, and also how to use the static Fault Analysis Tool within Artisan Studio and the fault tolerance verification plugin in Symphony.

¹Full name: Static Fault Analysis Profile

Contents

1	Introduction	6
2	Related Work	7
2.1	Fault Analysis Tool Requirements	7
2.2	Fault Analysis Tool Comparison	7
2.3	Fault Analysis Tool Conclusion	12
3	HiP-HOPS Fault Analysis in SysML	14
3.1	The Fault Analysis Architectural Framework	14
3.2	Ergonomic Profiling in Artisan Studio	20
3.3	SysML to HiP-HOPS XML Translation	23
3.4	Discussion: From Fault Modelling to Fault Analysis	27
4	SysML to CML for Fault Tolerance Verification	30
4.1	Plugin architecture	32
5	Conclusions	36
A	Fault Analysis Architectural Framework Additional Diagrams	38
A.1	Fault Analysis Architectural Framework Viewpoint Definitions . .	38
A.2	Rule Definition Views	46

1 Introduction

This task continues work in fault modelling in Task 2.4.2, providing static fault analysis in SysML and CML. There are two strands of work reported in this document: i) fault analysis in SysML; and ii) fault tolerance verification in CML. For both strands we propose the use of SysML for high-level SoS architectural modelling as used in the COMPASS project.

We take slightly different approaches in the two strands. In the first strand, given an SoS architectural model defined using SysML, we propose the use of a SysML profile to markup the model elements with failure data and failure propagation annotations. The SysML model may then be processed and output to an external fault analysis tool (HiP-HOPS [PWP⁺11]) which produces fault trees and FMEA tables.

In the second strand, we continue on the fault modelling work of Task 2.4.2 through the automatic translation of SysML models defined using the Fault Modelling Architectural Framework into CML and performing formal verification to ensure the correctness of recovery mechanisms in a SoS model. In the occurrence of a fault, a recovery mechanism in a fault-tolerant system model puts the system into the same state as if the fault had not happened. The success in the formal verification assures this behavior.

In Section 2, we compare several external fault analysis tools to determine the tool which best suits our requirements. Section 3 describes the first strand of work – fault analysis in SysML, and in Section 4 we describe the second strand – fault tolerance verification in CML. Finally we draw conclusions and discuss future work in Section 5.

2 Related Work

In this section, we consider possible external fault analysis tools that can be used in T3.3.3. We first define requirements for such a tool, then consider each tool in turn. Finally, given our requirements, we draw conclusions for the justification of the selected tool.

2.1 Fault Analysis Tool Requirements

In this section we show a criteria list for choosing a tool to be used for static fault analysis.

Active Development Is the tool still supported by the tool providers? Is there any active development to the tool? How stable is the tool?

Inputs How are models input into the tool? Can models be developed in *external modelling* tools and be provided as input to the (analysis) tool?

Scope for use with SysML If externally developed models may be input, how close is the required syntax to SysML? Can models be easily translated?

Analysis Performed What analysis is performed by the tool? In Static Fault Analysis, we require fault tree analysis – both qualitative (cut sets etc.) and quantitative (e.g. probability of a failure occurring within x time units). It would also be beneficial for a tool to output the complete fault tree to enable custom analysis, such as determining the causal chain of a fault in order to identify suitable recovery points.

Outputs How are results provided to the user? Can they be used by external tools (e.g. Artisan Studio)?

Openness Is the tool commercial, free or open source? Is there a version of the tool (full or trial) readily available (for testing purposes)?

2.2 Fault Analysis Tool Comparison

2.2.1 HiP-HOPS

HiP-HOPS (Hierarchically Performed Hazard Origin & Propagation Studies)² is a compositional safety analysis tool [PWP⁺11] which creates fault trees given

²<http://hip-hops.eu>

failure data of the components of a system, the deviations at component's outputs and the ways in which the failures may propagate through the system. The fault trees are analysed and results output by the tool.

Active Development

The tool is being used in the MAENAD³ EU project and is actively supported. The latest version of the tool was released in October 2012⁴.

Inputs

The tool may be executed from the command line and takes, as an input, an annotated model in the form of an XML file. This file uses a schema file (written in XSD) to define the syntax required by HiP-HOPS. Documentation is available which describes the required XML syntax. The HiP-HOPS tool provided may be run within Matlab/Simulink, in which annotated system models may be defined. These models are, however, simply output as XML. Therefore HiP-HOPS is compatible with an external modelling tool.

Scope for use with SysML

Given the XSD schema mentioned above, we may generate an XML file using an external modelling tool. Using the Artisan Studio API we may translate the model to an XML file. Alternatively, we could consider translating the underlying XMI file to the HiP-HOPS XML file, though this would require Artisan Studio to output profiled data to the XMI file. The format of HiP-HOPS input files appears straightforward to replicate given a SysML model.

The system model aspect is simply a hierarchical description, with a system composed of components, linked by their ports. Each component may be further defined as a system. This architectural structure could be modelled in the internal block diagram (IBD) of SysML. The additional failure annotations may be provided by a SysML profile.

Analysis Performed

HiP-HOPS generates (synthesises) fault trees, given an annotated system model. These trees are created by combining the local failure data of system components and subsystems. The fault trees are analysed to first obtain minimal cut sets, and then used for quantitative analysis and the FMEA. The FMEA indicates the causes of system failures by individual component failures.

³<http://www.maenad.eu/>

⁴According to the last modified date of the executable file available on the website. Note that the tool has been updated since this comparison was carried out, latest version now March 2014.

Outputs

The results are output in XML form, which are used to produce HTML documents containing the fault tree, cut sets and FMEA data. It is possible that the XML document could also be interpreted by Artisan Studio to provide a representation of the results in SysML.

Openness

The HiP-HOPS tool is a closed commercial tool, but it does have a freely available evaluation version that allows analysis of models with no more than 20 components.

2.2.2 AltaRica with ARC

AltaRica⁵ is a formal modelling language used to describe hierarchical models – which may include states and events, thus allowing behaviour of dynamic and state-based models. AltaRica’s project had three phases, two phases in four years to achieve the formal language and two years to build the workbench.

The ARC tool⁶ supports model checking for the AltaRica modelling language. The ARC model checker has been extended with functionality to generate minimal cut sets and “minimal sequences” (ordered sequences) of failure events from an annotated AltaRica model [GPKV11].

Active Development

The ARC model checker was last updated in June 2013. The report on the extension to generate minimal cuts was published in October 2011. As there is commonality between the authors of this report and the ARC tool developers it is assumed that there is continued support for these features in the ARC tool.

Inputs

The input is an AltaRica model, which is assumed to be a textual file with a .alt extension. The file specifies components as nodes which have “containers” for defining states, events, transitions and subnodes.

Scope for use with SysML

The textual input format for the model checker can be generated by the Artisan Studio API to translate from a SysML model.

⁵<https://altarica.labri.fr/forge/>

⁶<https://altarica.labri.fr/forge/projects/arc>

AltaRica models are more focused on states and state transitions than hierarchical descriptions, so state machines (or possibly activity diagrams) appear to be the closest SysML diagrams to an AltaRica model. Whilst an AltaRica model is based on a labelled transition system (which should translate from SysML state machines relatively easily) there are features of the language that would need further investigation such as flows and synchronisation vectors. The fault modelling aspects also require “tags” (which could be modelled by stereotypes in SysML) to be added to the model.

Analysis Performed

The analysis is aimed at systems with many states and to ensure scalability the results are limited to minimal cuts and minimal sequences. It is therefore not possible to obtain the complete fault tree or FMEA tables and quantitative analysis is not considered.

Outputs

The exact nature of the output format is unknown, but appears to just be a command line output. It is possible to generate a .dot file that can be opened in a Dot Viewer tool⁷.

Openness

The ARC tool is open source and freely available. It is assumed that this is also the case for the fault analysis extension.

2.2.3 FSAP/NuSMV-SA

Formal Safety Analysis Platform (FSAP) is a graphical interface for the NuSMV-SA model checking and safety analysis engine⁸.

FSAP aims to provide support to GFML (Generic Failure Mode Library), GSRL (Generic Safety Requirements Library), custom failures, results and fault tree viewers.

NuSMV-SA is an extension to the NuSMV symbolic model checker⁹ providing safety analysis capabilities (CTL and LTL property checking and invariant formula simulation). The additions to NuSMV include: fault tree construction,

⁷DOT file format: [http://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](http://en.wikipedia.org/wiki/DOT_(graph_description_language))

⁸<https://es.fbk.eu/tools/FSAP/>

⁹<http://nusmv.fbk.eu/>

traces for minimal cut set, ordering analysis of minimal cut sets and FMEA tables. NuSMV-SA may be run independently of FSAP, although it is not possible to get only NuSMV-SA. FSAP and NuSMV-SA are available together.

FSAP is available for evaluation or teaching purposes. To get a license, one should contact FSAP team at: fsap@fbk.eu.

Note that Mhenni et al. [MCR⁺12, MNKC13] also propose a methodology using NuSMV for model checking safety requirements, but this does not include safety analysis such as FTA or FMEA. For this they propose to link to MeDISIS, but have not implemented this link due to the inaccessibility of the tools [MCR⁺12].

Active Development The latest release of NuSMV (v2.5.4) was in October 2011. The latest release for FSAP/NuSMV-SA (v1.2.4) was announced in December 2006.

Inputs FSAP/NuSMV-SA appears¹⁰ to take an XML file as input for failure annotations (they call it a SAT, Safety Analysis Task) along with a textual SMV file for the system model. Safety properties are written in CTL or LTL. A pattern library of safety properties appears to be included within the tool.

Scope for use with SysML Whilst there appears to be an XML format for inputting failure annotations, there was no XSD file readily available¹¹ to determine its format, only a number of examples¹². If this format were available it would probably be feasible to generate this file and the SMV file of the system behaviour from a SysML model.

Analysis Performed The tool appears to perform model checking of safety properties as well as FTA, FMEA and simulation-based analysis.

Outputs The fault tree analysis appears to output its results in two text files: one for the events and one for the gates. Ordering and traces about minimal cut sets, as well as FMEA tables are also output as text files.

Openness FSAP/NuSMV-SA is an in-development tool. The tool appears not to be fully developed and only available (outside of the projects for which it was developed) for evaluation and teaching purposes.

¹⁰<https://es.fbk.eu/tools/FSAP/screenshots.html>

¹¹Correct at the time of exploring the fault analysis tool options

¹²<https://es.fbk.eu/tools/FSAP/examples/>

2.2.4 MeDISIS

MéDISIS is a method (not yet a tool) presented in [DIK10, CIKD11] based on SysML and AltaRica concepts. In [DIK10] they define an algorithm that can be easily implemented in tools like Artisan Studio. The main contribution of the method is that it uses AltaRica's concept of flows, as it has not been handled directly in other approaches using UML or AADL and is a basic feature considering systems modelling. Depending on the level of architecture modelling, it is also suitable for SoS.

It also provides a database of “dysfunctional behaviour” (erroneous behaviour) for standard components that enforces knowledge reuse by using, checking and updating it within some company [CDIK13].

In [CDIK13], the following analyses are performed using the MéDISIS method:

FMEA Automatic synthesis of FMEA table for failure modes study.

AltaRica Mapping to evaluate failure scenarios.

Simulink Conversion to Simulink to perform simulation and fault injection.

2.2.5 MDA

MDA is an approach that uses Model-based Deviation Analysis [HCW02]. It consists of formally modelling the nominal behaviour of the system and modelling deviations either in parallel with the nominal model or in its input – defined using NuSMV. As it is more concerned about input values, it may not apply to all SoS architectures and system types. In [JHMW06], the authors show how to model deviations directly in Simulink for Model-based *Safety* Analysis, but suggest that a better approach is to insert deviations automatically, which is left as future work.

2.3 Fault Analysis Tool Conclusion

Overall the HiP-HOPS tool was chosen because:

- Its use of a standardised XML interface complete with documentation on how to use it and XSD files of the expected format made it feasible to link it up with Artisan Studio

- It produces full fault trees, not just cut sets. Such provision of more detailed results enables a deeper understanding of the system, as well as quantitative analysis and optimisation
- It is a mature and industry strength tool with active development. Therefore not only would the Fault Analysis Tool we developed be more future proof than comparable alternatives, but also the developers of HiP-HOPS would be available to provide us with any extra information we needed to develop the Fault Analysis Tool

The main downside of the HiP-HOPS tool is that it is not free, but it seems to be the case that all industrial strength tools for fault analysis are commercial. The availability of an evaluation version of HiP-HOPS makes it possible to explore the functionality of the tool without having to purchase it.

3 HiP-HOPS Fault Analysis in SysML

This task develops an extension to the SysML tool Artisan studio by implementing support for a Fault Analysis framework, in the form of a Fault Analysis SysML profile, and the necessary tool development to enable fault analysis of a SysML model using the external HiP-HOPS tool.

In Section 3.1 we define the *Fault Analysis Architectural Framework*. This framework is defined using the COMPASS AF Framework (CAFF) [HHP⁺13]. Given the Fault Analysis Architectural Framework definition, three subsequent tasks are required to produce robust, usable fault analysis tools:

1. a consistent embedding of the Fault Analysis Architectural Framework in the form of a SysML profile in Artisan Studio – the Fault Analysis profile, which is usable by SoS engineers familiar with SysML modelling;
2. the translation of a SysML model which applies the Fault Analysis profile to the notation required by the HiP-HOPS tool – XML; and
3. providing the ability to run the analysis from within Artisan Studio.

This tooling effort is outlined in Figure 1.

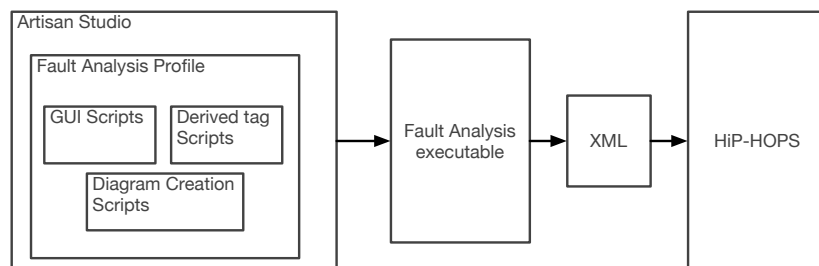


Figure 1: Fault Analysis tool chain

In Section 3.2 we describe briefly the ergonomic profiling for Artisan Studio, which addresses Task 1 above. The SysML to XML translation and analysis tool that execution addresses Tasks 2 and 3 above is summarised in Section 3.3.

3.1 The Fault Analysis Architectural Framework

In the Fault Analysis Architectural Framework, we use the HiP-HOPS concepts and map the required input to a SysML model. In this section, we apply the CAFF to define the Fault Analysis Architectural Framework. The CAFF approach requires that viewpoints are identified for a pattern or architectural framework (in

this case the relationships between those viewpoints are described) and the syntax (or permitted modelling elements) of the viewpoints are defined. The CAFF approach also advocates recording the context of the pattern and its viewpoints, typically through use cases, and a collection of rules which constrain the viewpoints.

We first define the ontology, the collection of concepts that are used when defining the profile stereotypes. Subsequently, we identify the Fault Analysis Architectural Framework views, the relationships between those views in a Viewpoint Relationship View, define each view with Viewpoint Definition Views and finally provide rules for each viewpoint, in Rule Definition Views. Contrary to the CAFF approach, we do not define use cases for the Fault Analysis Architectural Framework or its viewpoints.

3.1.1 Ontology and Concepts

The ontology and concepts of the Fault Analysis Architectural Framework are given in this section. In Figure 2, the top-level element is the *Fault Analysis Model*, which contains the *SoS* and a collection of *Optimisation Parameters* for the model. Note that whilst we base our terminology on that used in HiP-HOPS, we adjust the terms somewhat to use the language of SoS – hence the use of an *SoS* model element, and subsequent use of *Constituent* systems.

At the next level, in Figure 3, the *SoS* contains a collection of *Constituents* and *Lines*. *Constituents* have *Ports* which are connected by (SysML) *Connectors* and those *Connectors* are combined to form multiway connectors known as *Lines*. Each *Constituent* must have at least one *Implementation*, which may in turn contain a collection of *Components* and *Lines*. Each *Component* must have at least one *Implementation*, which can be broken down into further *Components* and *Lines*.

The *Failure Logic* ontology is given in Figure 4. A *Line End*, representing an endpoint of a *Line*, contains optional *Propagation Logic*. The *Propagation Logic* contains compulsory *Port Expressions* and optional *Failure Classes*. In addition, an *Implementation* may contain *Basic Events* and *Output Deviations*. The *Output Deviations* reference a *Port* contained by the *Constituent/Component* owning the *Implementation*.

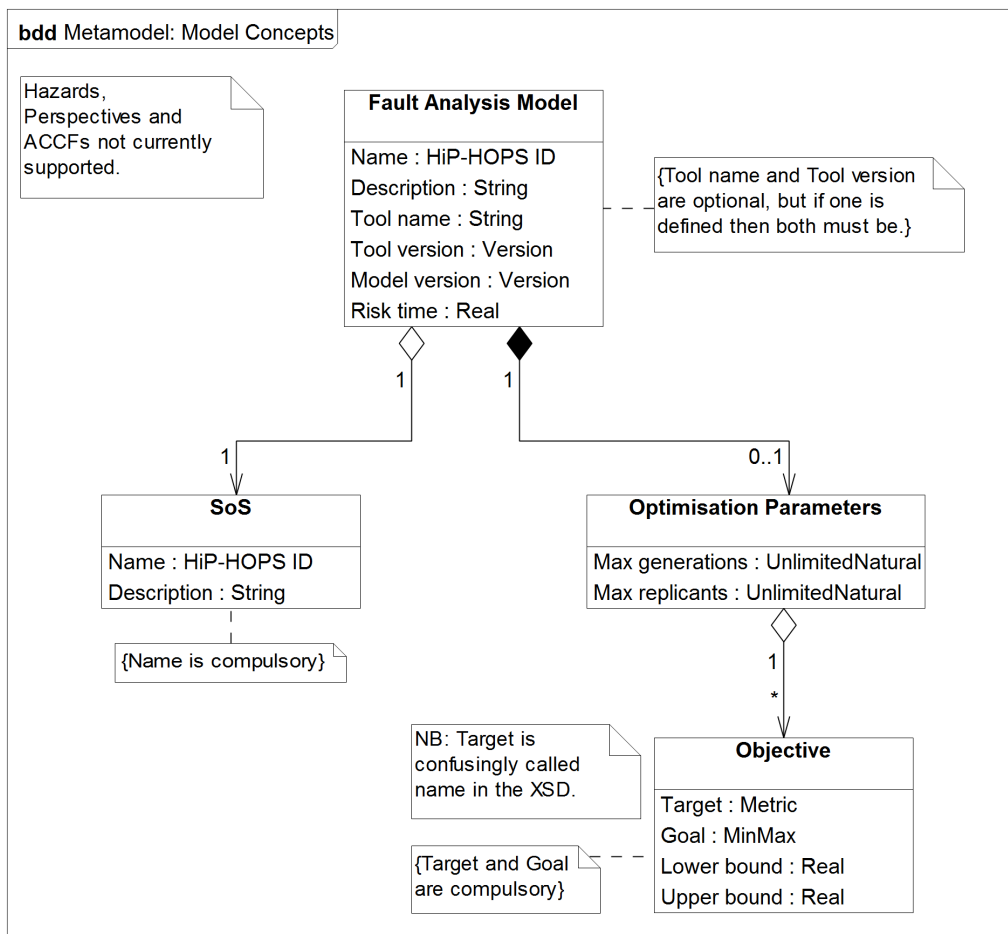


Figure 2: HiP-HOPS Ontology – Model Concepts

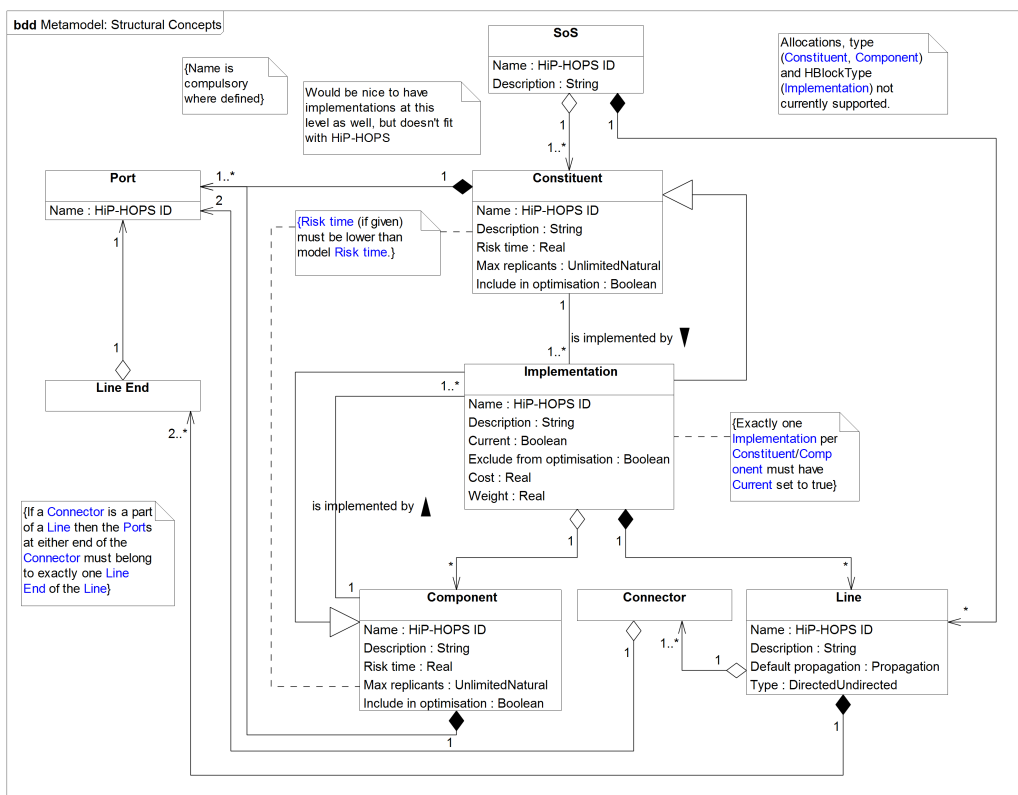


Figure 3: Fault Analysis Architectural Framework Ontology – Structural Concepts

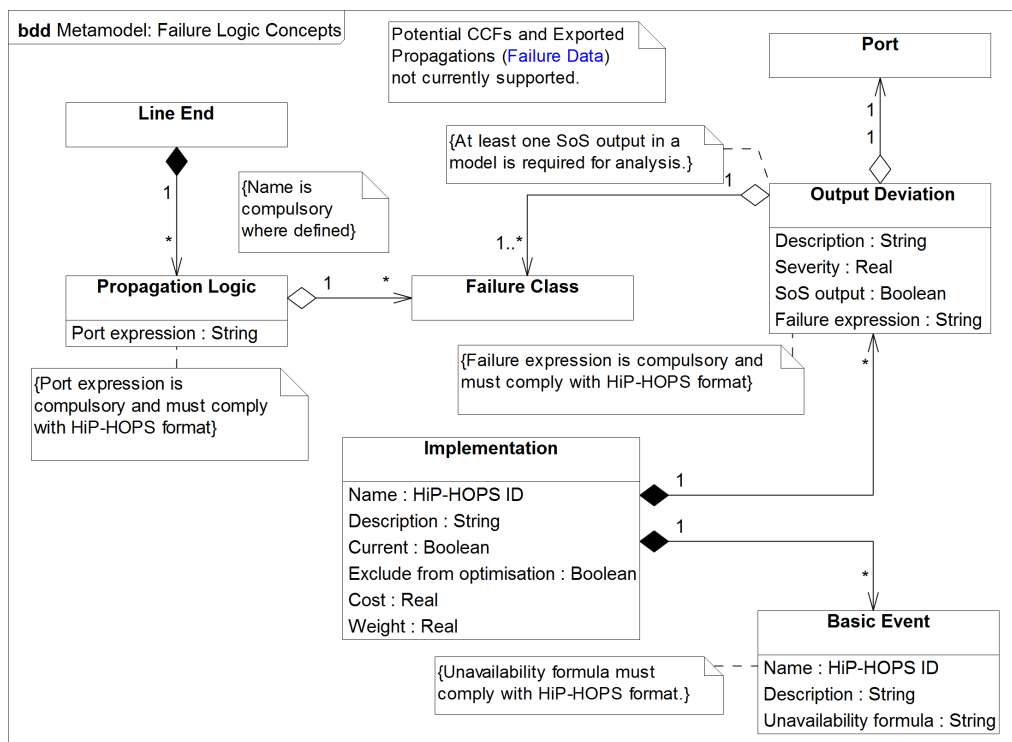


Figure 4: Fault Analysis Architectural Framework Ontology – Failure Logic Concepts

3.1.2 Fault Analysis Architectural Framework Viewpoints and Their Relationships

The Fault Analysis Architectural Framework has several viewpoints shown in Figure 5, divided into three broad categories: the Model Definition, Structure and Failure Logic. The Model Definition contains a single viewpoint: the *Model and Optimisation Definition Viewpoint*. There are four Structural viewpoints: *SoS Definition Viewpoint*, *SoS Connections Viewpoint*, *Implementation Definition Viewpoint* and *Implementation Connections Viewpoint*. There are three Failure Logic viewpoints: *Failure Class Definition Viewpoint*, *Line Definition Viewpoint* and *Implementation Failure Definition Viewpoint*.

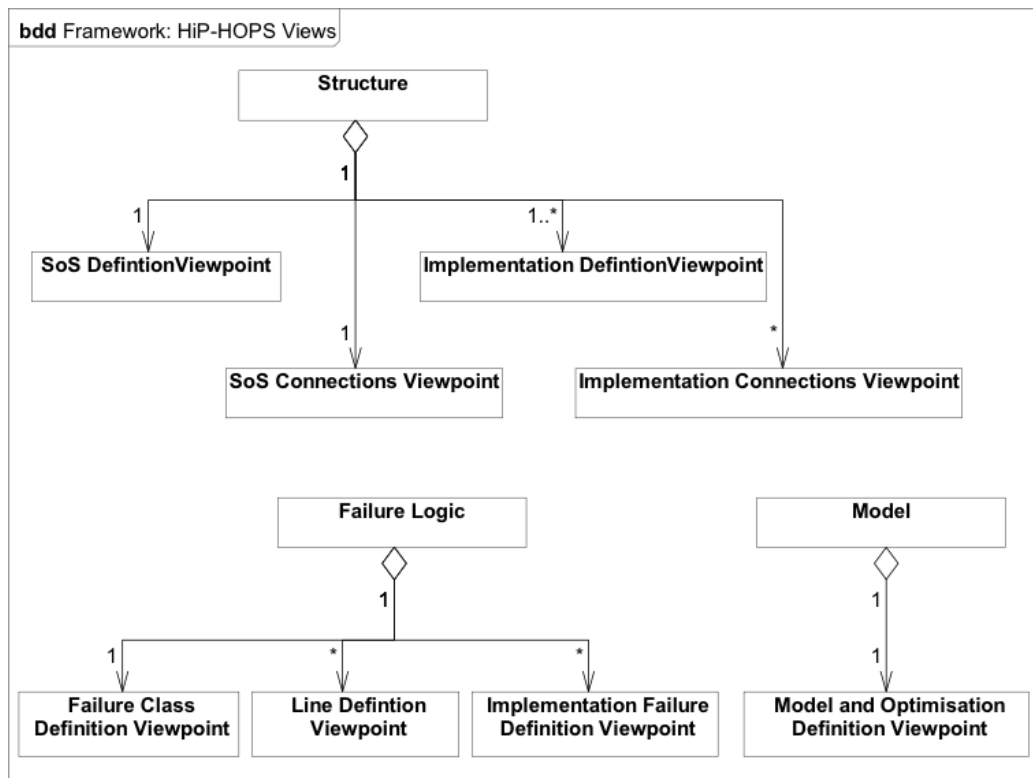


Figure 5: Fault Analysis Architectural Framework Profile Viewpoints

From Figure 6, we see that the *Model and Optimisation Definition Viewpoint* identifies the SoS to analyse. The SoS is subsequently defined in the *SoS Definition Viewpoint* in terms of its Constituent Systems (CSs) and Lines. The *SoS Connections Viewpoint* describes the connections between the CSs. Lines are associated with the connectors that they combine to form multiway connectors. The CSs identified in the *SoS Definition Viewpoint* have their Implementations defined in

Implementation Definition Viewpoints. This viewpoint defines Implementations in terms of their Components and Lines. Each Component has an associated *Implementation Definition Viewpoint* in which its Implementation is defined and may be further decomposed into sub-components and Lines. The components identified in an Implementation Definition Viewpoint are connected in an *Implementation Connection Viewpoint*, and Lines are associated with the connectors that they combine to form multiway connectors. The Lines identified in the *SoS Definition Viewpoint* and *Implementation Definition Viewpoints* are defined in *Line Definition Viewpoints*. The failure information for Implementations is defined in Implementation Failure Definition Viewpoints. Finally the Failure Classes used in Line Definition Viewpoints and *Implementation Failure Definition Viewpoints* are defined in a *Failure Class Definition Viewpoint*.

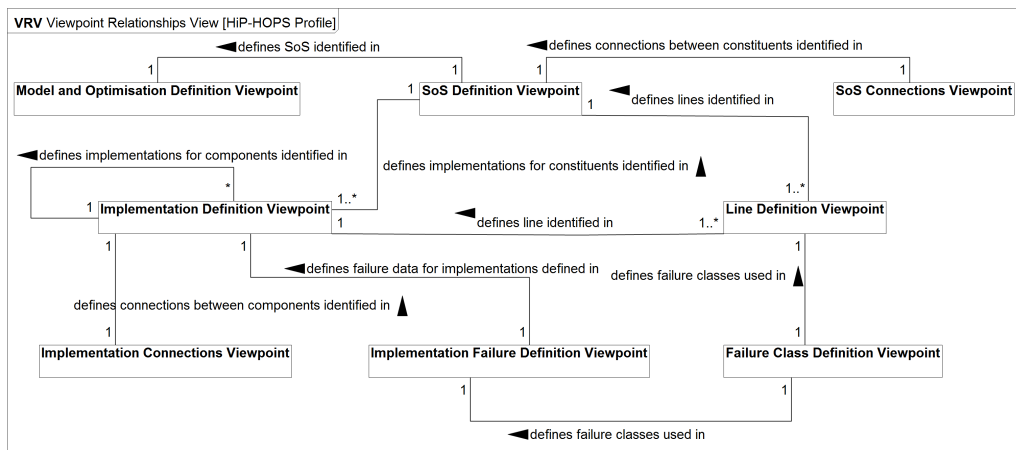


Figure 6: Fault Analysis Architectural Framework Profile Viewpoint Relationship View

The viewpoints are described briefly in Table 1. The remainder of the CAFF views for the Fault Analysis Architectural Framework are given in Appendix A.

3.2 Ergonomic Profiling in Artisan Studio

The ergonomic profiling available in Artisan Studio is a feature enabling the modification of the tool GUI to provide more comprehensive profile support. This section describes the ergonomic profiling effort indicated in Figure 7.

In this task, we implemented: GUI scripts for toolbars; scripts to automate Fault Analysis diagram creation; and scripts for derived tags. The purpose of these tasks are to aid in the construction of Fault Analysis models, and as such we believe

Table 1: The Fault Analysis Architectural Framework viewpoints

Name	Purpose of Viewpoint
Model and Optimisation Definition	Defines the global parameters of an <i>Fault Analysis Model</i> , including parameters for optimisation (if required).
SoS Definition	Defines the <i>SoS</i> in terms of its <i>Constituents</i> and <i>Lines</i> .
SoS Connections	Shows the connections between the <i>Constituents</i> of an <i>SoS</i> and associates each connection with a <i>Line</i> .
Implementation Definition	Defines an <i>Implementation</i> of a <i>Constituent</i> or a <i>Component</i> in terms of its <i>Components</i> and <i>Lines</i> .
Implementation Connections	Shows the connections between the <i>Components</i> of an <i>Implementation</i> and associates each connection with a <i>Line</i> .
Failure Class Definition	Defines the possible <i>Failure Classes</i> of the <i>Fault Analysis Model</i> .
Implementation Failure Definition	Defines the <i>Basic Events</i> of an <i>Implementation</i> along with its <i>Output Deviations</i> .
Line Definition	Defines the <i>Propagation Logic</i> associated with each end of a <i>Line</i> .

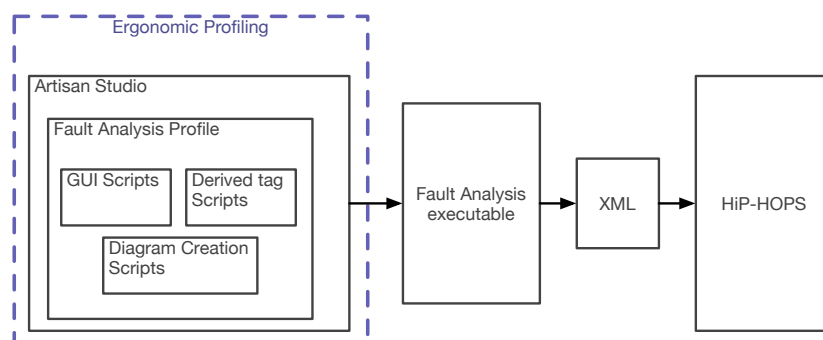


Figure 7: Ergonomic profiling in the Fault Analysis tool chain

that these are best seen through the user manual (part A of this deliverable) and through the use of the tool. In this section, therefore, we provide an overview of the the final ergonomic profiling task from this collection of implementations: the derived tags.

3.2.1 Derived Tag Scripts

We concentrate on the derived tag scripts due to the fact that these derived tags enable a degree of model validation and prevention of compatibility issues (with HiP-HOPS). The derived tags broadly fall under five categories:

1. Those that highlight the composition and aggregation relationships that have been defined for a block. The reasoning behind this is that such a relationship can be defined in the SysML model without being visible on any diagrams. This enables the user to validate the relationships that have been defined. The derived tags that fall under this category are: *hiphops-has-optimisation-params*, *hiphops-has-objective*, *hiphops-has-constituent*, *hiphops-has-line*, *hiphops-has-component*, *hiphops-has-output-deviation*, *hiphops-has-basic-event* and *hiphops-has-propagation-logic*.
2. Those that highlight other relationships that have been defined for a model element or view. Similarly to the previous set of derived scripts, some other relationships (such as generalisation) may be defined but not visible in any views so these are highlighted to enable user validation. The derived tags that fall under this category are: *hiphops-has-port*, *hiphops-is-implemented-by* and *hiphops-includes-connector*. Derived tags may also be used in this way to prevent a user entering redundant data – this is the case for the derived tag *hiphops-model-fail-classes*. This tag identifies the valid *Failure Classes* defined in a given view and allows the user to just identify the view in which *Failure Classes* of a *Fault Analysis Model* are defined.
3. Those that provide additional details about a model element that is being referenced in another tag. For example, when referencing a port the default is to just show the name of the port in the tag – sometimes it is useful to know which block owns the port as well. The derived tags that fall under this category are: *hiphops-connectors-full-names* and *hiphops-full-port-name*.
4. Those that provide information about the tool being used to define the *Fault Analysis Model* (and also the version of the *Fault Analysis Model*) for debugging purposes. The derived tags that fall under this category are: *hiphops-tool-name*, *hiphops-tool-version* and *hiphops-model-version*.

- Those that check the name and description of model elements comply with HiP-HOPS restrictions on these values. If an invalid character is used in the name or description of model elements it is either removed or replaced by a valid character in the respective derived tag. The user can then inspect the derived tag to determine how their model elements will be named according to the HiP-HOPS model and analysis. The derived tags that fall under this category are: *hiphops-identifier* and *hiphops-description*.

3.3 SysML to HiP-HOPS XML Translation

The Fault Analysis Tool (executable) for Artisan Studio translates a SysML model, defined using the Fault Analysis Architectural Framework, into an XML file conforming to the HiP-HOPS XML schema as indicated in Figure 8.

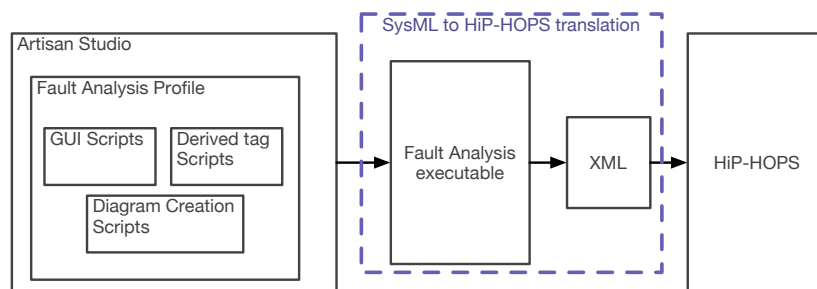


Figure 8: Translation in the Fault Analysis tool chain

The translation also performs error checking to ensure that the model being translated conforms to the schema and meets the (informally defined) consistency and correctness rules for the HiP-HOPS tool. The Fault Analysis Tool was developed as a VB.Net application and built as an executable that may be launched through Artisan Studio. The application architecture is outlined in Figure 9. The `SysMLtoHiPHOPS.vb` module receives input from the user in the form of interaction with GUI elements. This module has several routines which invoke functions and routines of the underlying `SysML2HH.vb` module. Two ‘helper’ modules provide: constant values of the required Artisan Studio stereotype identifiers (`Stereotypes.vb`); and functions for error checking and generating unavailability formulae (`SysML2HHUtilities.vb`). The application developed also includes two further modules, one that defines some generic helper functions for use with Artisan Studio (`StudioUtils.vb`) and one that defines an exception class for the purposes of error handling (`Exceptions.vb`).

The SysML model queries are performed using the Artisan Studio API – using

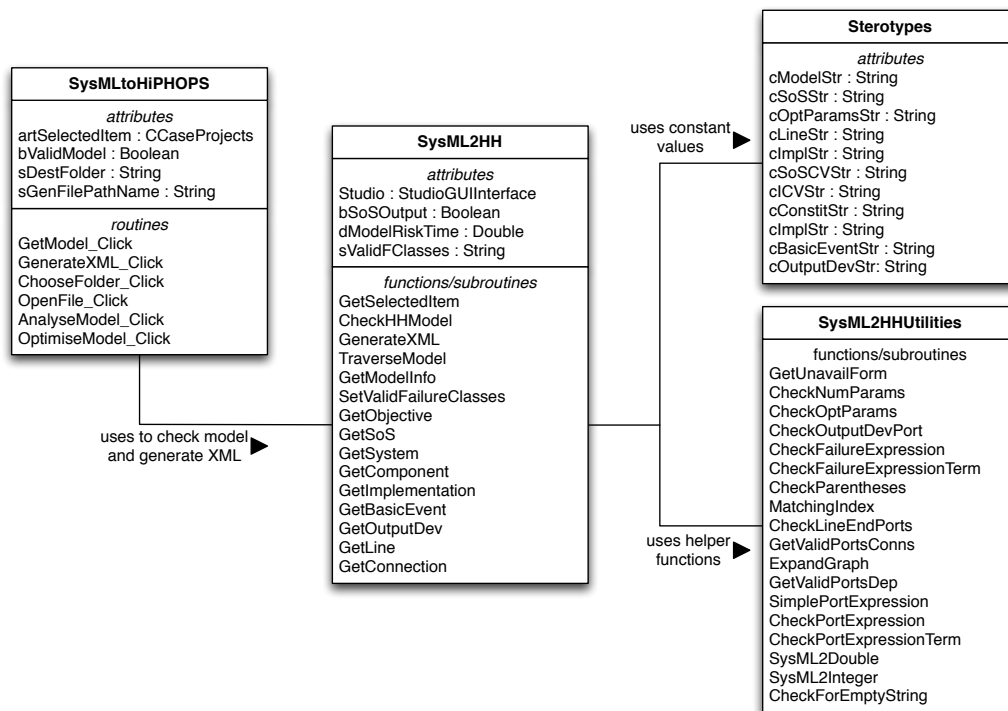


Figure 9: Fault Analysis tool architecture

Automation¹³. Automation provides the Fault Analysis Tool with the ability to make requests to Artisan Studio through its API – requiring Artisan Studio to be running, the relevant SysML model to be open and a specific model element to be selected – in this case a *Fault Analysis Model* model element. The Fault Analysis Tool is supplied as a GUI application, initiated through context menus in Artisan Studio. The application has a series of behaviours based on button-oriented interaction.

3.3.1 Fault Analysis Tool Behaviour

In this section, we briefly discuss the typical flow of behaviour of the tool. All behaviours originate in the `SysMLtoHiPHOPS.vb` module.

Choose save destination The `ChooseFolder_Click` routine, executed on the press of the ‘Choose Folder’ button, provides a selection interface to the users file system via the built-in `FolderBrowserDialog` class. The se-

¹³[http://msdn.microsoft.com/en-us/library/windows/desktop/ff486375\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff486375(v=vs.85).aspx)

lected destination is stored ready for use when saving the generated XML file and HiP-HOPS results.

Obtain model The `GetModel_Click` routine, executed on the press of the ‘Get Model’ button, identifies which item has been selected in Artisan Studio and checks that it is a *Fault Analysis Model*. The currently selected item is obtained through a call to the `GetSelectedItem` routine of the `SysML2HH.vb` class which queries the Artisan Studio API, and a call to the `CheckHHModel` queries the selected model and tests if it has the correct stereotype applied. The user may not proceed until they have selected the appropriate model element.

Generate XML File The `GenerateXML_Click`, invoked upon pressing the ‘Generate XML’ button, first checks that the previous steps have been performed successfully. If this is the case, a call is made to the `GenerateXML` routine of the `SysML2HH.vb` class. This routine creates an `XDocument` (XML) object, populates the object through a call to the `TraverseModel` routine and saves the file to the specified location. The `TraverseModel` routine traverses the *Fault Analysis Model* in Artisan Studio and produces the required XML tags and data (that conform to the HiP-HOPS XML schema). This, and subsequently invoked routines, form the bulk of the functionality of the Fault Analysis Tool. The traversal and XML document population is in itself not complex, however this process does include error checking that is described in more detail in Section 3.3.2.

Perform fault analysis and model optimisation The final behaviours provided by the Fault Analysis Tool are given in the `AnalyseModel_Click` and `OptimiseModel_Click` routines. The two behaviours take the generated XML file, and execute the HiP-HOPS program as a system process. Results are given in an XML file (at a user-specified destination) and displayed by HiP-HOPS in a web browser.

3.3.2 Fault Analysis Tool Error Checking

In this section we outline the error checking that is implemented within the Fault Analysis Tool in order to ensure that the model created in SysML is compatible with the HiP-HOPS tool. A custom defined exception `SysML2HHException` has been defined to handle issues with the compatibility of the SysML model with the requirements for the HiP-HOPS tool. The `SysML2HHException` inherits the functionality of the (standard) `ApplicationException`. The exception is caught by the `GenerateXML_Click` sub-routine, which alerts the user to

the issue via a `MsgBox` and outputs the error message from the exception to the *Generation status* text box on the Fault Analysis Tool.

The error checking code falls under one of four categories:

Non-empty tags Certain information is required for analysis of the model, meaning that a number of the tags may not be empty (see Appendix A of D33.3a). This is a straightforward check that utilises the `CheckForEmptyString` sub-routine for “value-type”¹⁴ variables and is carried out inline with other error checking and model processing for “object-type” variables. In addition to those tags listed in Appendix A of D33.3a, the Fault Analysis Tool also checks that each model element has a non-empty identifier (in theory, this is a redundant checking mechanism because Artisan Studio should not allow empty identifiers).

Numerical tags Some of the tags require numerical input, but numerical types cannot be specified for tags in Artisan Studio therefore a text type has been used for these tags and error checking in the Fault Analysis Tool ensures that numerical values have been provided for these tags. To do this the functions `SysML2Integer` and `SysML2Double` have been defined to parse the tag into an integer type or double type respectively. If this parsing fails an exception is thrown with a suitable error message.

Tag formatting constraints Some other tags have quite strict formatting requirements to enable analysis of the model. For example, the *Unavailability formula* of a *Basic Event* requires the format `type(paramId1=value1; paramId2=value2; ...)` where `type`, `paramId1`, `paramId2` are restricted to specific string literals and `value1` and `value2` are numerical values. The sub-routine `GetUnavailForm` is used to parse this expression to convert it to HiP-HOPS format and check for formatting errors. Other constraints have similar sub-routines to parse and check the data. See Section 2.2.6 of D33.3a for instructions on how to find out the formatting constraints for a given tag.

Model level constraints There are also a number of constraints that go across several model elements, for instance there must be at least one *Output Deviation* that is classified as an *SoS output*. These constraints are detailed in Appendix A of D33.3a and conformance to these constraints are checked in the Fault Analysis Tool (except where stated). For the example given, the variable `bSoSOutput` is initially set to `False` and is updated to `True` when an *Output Deviation* that has its tag *SoS output* set to “true” is found.

¹⁴Attributes of a model element are either stored as an object for all attributes that reference other model elements and as a string for all other attributes.

If the value of `bSoSOutput` is still `False` after the whole model has been parsed an exception is thrown.

3.4 Discussion: From Fault Modelling to Fault Analysis

In this section, we provide some initial, informal, discussion of links between previous work on the Fault Modelling Architectural Framework and the Fault Analysis Architectural Framework.

3.4.1 The Fault Modelling Architectural Framework

In our previous work we developed a Fault Modelling Architectural Framework (FMAF) for designing fault-tolerant SoSs [ADP⁺13] and demonstrated how fault-tolerant properties of such models could be verified [APR⁺13]. The FMAF defines a set of viewpoints that prompts an SoS developer to consider the impact of faults at the early stages of design, resulting in a coherent set of views that aid the stakeholders of the SoS to understand its erroneous and recovery behaviour.

The FMAF has been developed with respect to established dependability concepts [ALRL04]. An SoS *failure* [AIP⁺14] is defined as a deviation of the service provided by the SoS from expected (correct) behaviour. An *error* is defined as the part of the SoS state that can lead to its subsequent service failure. The adjudged or hypothesised cause of an error is called a *fault* and (in keeping with our chosen nomenclature) a failure of a constituent system (CS) can cause a fault of the SoS.

The set of viewpoints prescribed by the FMAF provide an approach for defining:

- faults, errors and failures of SoSs;
- relationships between faults, errors and failures and CSs;
- structural designs that enable fault tolerance;
- the behaviour of the SoS in the presence of errors; and
- the recovery behaviour provided by CSs.

For further details of these viewpoints, see [ADP⁺13].

3.4.2 Relationship with FMAF

The Fault Analysis Architectural Framework and FMAF are defined using a common framework – the CAFF. We may therefore make extensions and also combine them using their ontologies to identify shared modelling elements and relationships. In this section, we briefly consider the areas in which the Fault Analysis Architectural Framework and FMAF overlap.

The Fault Analysis Architectural Framework and the FMAF can be used in combination to provide a consistent and complementary set of fault-explicit viewpoints of an SoS model. The structural viewpoints of the FMAF could be used to derive and provide a skeleton Fault Analysis model – identifying the CSs making up the SoS and their composition. These elements may be used to populate initial *SoS Definition*, *SoS Connections*, *Implementation Definition* and *Implementation Connections* views. Note – the *Line* and *Line End* elements can not be derived from the FMAF, and therefore must be added (along with any Propagation Logic elements) manually in the Fault Analysis Architectural Framework.

Both aspects of the model should have a consistent understanding of the faults, errors and failures of the SoS. To achieve this we recommend that a single definition of these is used for both sets of viewpoints, by making use of the FMAF *Fault/Error/Failure Definition Viewpoint* (FEFDV). The FEFDV provides details of all of the relevant faults, errors and failures of the SoS and identifies any relationships that exist between them.

Given a model with a fully defined FEFDV, we suggest two key links to the Fault Analysis Architectural Framework:

- Any fault located in a CS as identified in a FEFDV should be traceable to a *Basic Event* in the Fault Analysis Architectural Framework. Conceivably, the fault property as defined in the FMAF could be decorated with information that may be used to derive *Basic Events*.
- A failure of the SoS as identified in a FEFDV should correspond to an *Output Deviation* in the Fault Analysis Architectural Framework and that *Output Deviation* should be identified as an *SoS output*. As above, failures in the FMAF could be given additional information to populate the *Output Deviations* in the Fault Analysis Architectural Framework.

Another source of *Output Deviations* may come from examining the scope of the interruptible regions that represent erroneous behaviour within an FMAF *Fault Activation View*. Further, FMAF *Recovery Views* could provide information about how *Constituents* can mask faults (and as such give an indication as to when de-

viations are *not* propagated across a *Constituent*).

Traceability links could be used as a semi-formal way (see for example [AIP⁺14]) of relating the complementary aspects of the Fault Analysis Architectural Framework and the FMAF. This may utilise the Traceability Pattern as described in [Per13], however the implementation of this is the subject of further research.

4 SysML to CML for Fault Tolerance Verification

In this section we provide an overview of the fault tolerance verification plugin for the Symphony tool platform¹⁵. The fault tolerance verification is based on concepts from [Ros97, p. 313], extended to CML in [ADP⁺13]. This verification provides static analysis for a given CML model using the CML model-checker [FMDW13].

The plugin can check any well-formed CML model and relies on the modifications of the SysML-to-CML mapping reported in [ADP⁺13] where SysML is used as the starting point of SoS modelling. The modifications give a CML semantics for those SysML fault-modelling elements.

The original definition of fault tolerance uses a stable failures refinement check [ADP⁺13, p. 22]: `Lazy refines NoFaults`, for example. Currently, however, the CML model-checker only supports deadlock-freedom, livelock-freedom, non-determinism, and traces refinement verifications [FMDW13]. We therefore need to adapt the original refinement check to use the supported checks. If we assume deadlock-freedom of the process under analysis, the adaptation is possible via a deadlock-freedom check of the parallel composition of the CML processes in the original refinement check. For the above example, the refinement check is equivalent to a deadlock-freedom verification of: `Lazy [|Alpha|] NoFaults`, where `Alpha` is the alphabet of the selected process.

The implementation of the plugin followed the standard plugin development procedures for the Symphony tool platform (refer to [CMCP13] for more information about plugin development for the Symphony Platform). The functional view of the plugin is shown in Figure 10. The Fault Tolerance Plugin allows the user to verify the correctness of the recovery mechanisms of a fault-tolerant system by selecting a CML process and choosing the appropriate verification from a popup menu. The plugin then:

1. Prepares files and folders for the selected CML process – the Model-Checker file names are defined.
2. Schedules five Eclipse jobs for the selected CML process:
 - (a) Deadlock-freedom verification;
 - (b) Livelock-freedom verification;
 - (c) Semifairness verification;

¹⁵All references to Symphony and Symphony Platform are based on version 0.3.2 and the relevant user manual [CMCP13].

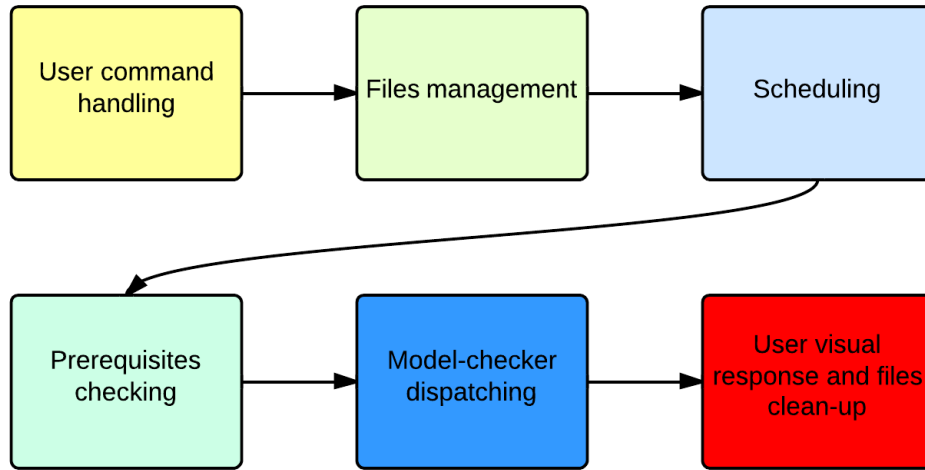


Figure 10: FT plugin functional view

- (d) Full fault tolerance verification;
- (e) Limited fault tolerance verification.

The first three of these verification jobs are explained in [ADP⁺13] and the final two are replaced by a deadlock-freedom verification as explained at the beginning of this section.

3. Checks the prerequisites for each job before their execution.
4. Dispatches each verification to the model-checker.
5. Handles the response, showing the results on the Symphony Platform.

The full fault tolerance verification considers all unwanted events, including unhandled failures. The plugin warns the user if a model is fully fault-tolerant¹⁶. The fault tolerance limited verification considers recovery of faults only (this behaviour can be customised to set the limiting conditions for this verification – see D33.3a).

Recall that both fault tolerance verifications use a deadlock-freedom verification of a parallel composition of processes. Those processes are:

Full fault tolerance a *no faults* and a *lazy abstraction* version of the selected

¹⁶Note that even in a well-designed system there are typically unrecoverable failures because no system is infallible. Therefore full fault tolerance is usually not achievable.

process.

Limited fault tolerance a *no faults* version of the selected process and a *lazy abstraction* version limited to a subset of the unwanted events.

For each job listed above, we use a CML process definition template in the `Message.properties` file. This template is used to generate a temporary CML file for the selected process. For example, for a process with name `P`, the resultant temporary CML file is:

```
process FT_P_Lazy = NoFaults_P [| Alpha_P |] Lazy_P

process Lazy_P =
begin
  @ (P [| E |] ChaosE) \\ (E union H)
end

process NoFaults_P =
begin
  @ (P [| E |] Stop) \\ H
end
```

In this case, the deadlock-freedom verification runs on the process `FT_P_Lazy`.

The generated CML files, in conjunction with the user-defined CML model and the verification properties, are all dispatched to the CML model-checker. The model-checker converts each property verification to a single FORMULA file. It then runs the file and processes the results, which are formatted by the FT plugin and shown to the user.

4.1 Plugin architecture

The plugin architecture follows Java programming best practices (design with responsibilities) and respects Symphony Platform structure.

The class diagram in Figure 11 shows the main classes of the plugin. The *SelectProcessHandler* class is responsible for selecting a CML process within the Symphony Platform. After selecting the process, the class calls its abstract method *doOnSelectedProcess*, which receives the process definition for which the verification is run. Subclasses override this method to inherit the process selection heuristics and implement the behaviour for each user command. Subclasses *Fault-*

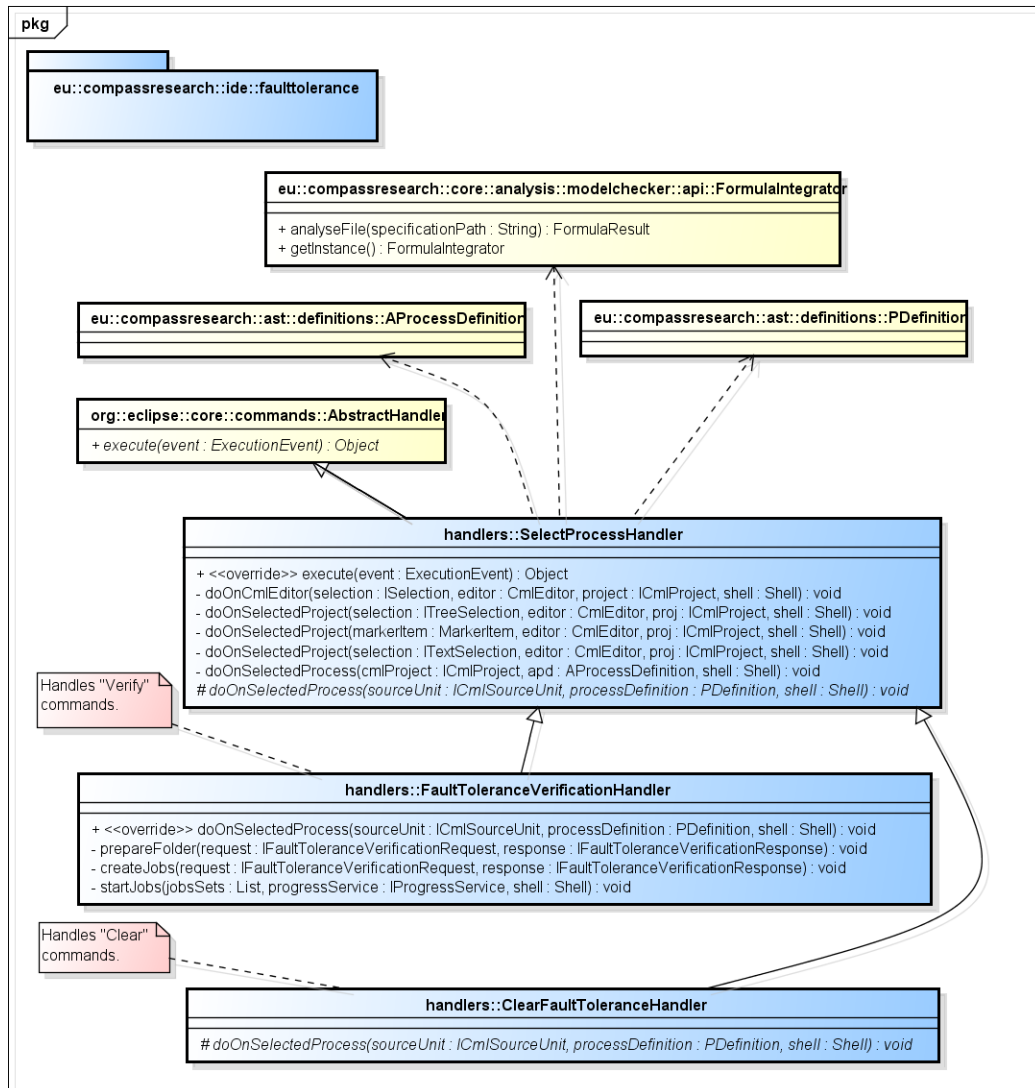


Figure 11: FT plugin class diagram

ToleranceVerificationHandler and *ClearFaultToleranceHandler* run the verification and clear its results, respectively.

The sequence diagram in Figure 12 shows the sequence of methods for verifying fault tolerance (class *FaultToleranceVerificationHandler*). Classes *IFaultToler-*

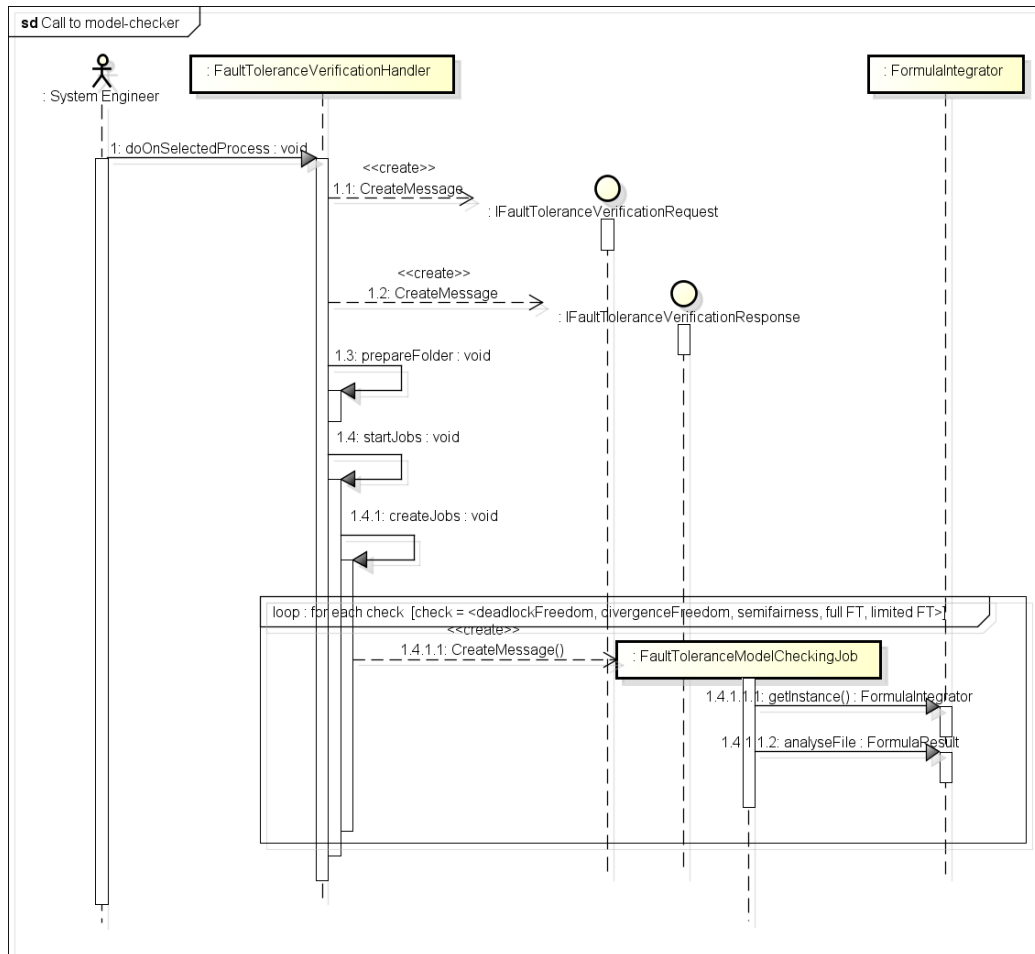


Figure 12: FT plugin sequence diagram and integration with the model-checker

anceVerificationRequest and *IFaultToleranceVerificationResponse* define methods to access the required parameters for executing all verification jobs and to show the results to the user. After their creation, instances of these classes are passed as parameters to all of the following methods (*prepareFolder*, *startJobs* and *createJobs*). Each job then gets the instance of model-checker's *FormulaIntegrator* and calls the *analyseFile* method.

4.1.1 Scheduling mechanism

The most elaborate algorithm in the architecture is the scheduling of the verification jobs, which establishes a sequence of execution. The algorithm allows some jobs to run in parallel although this is not supported in the current version of the model-checker. The model-checker has scalability issues for larger systems, so it is important to run only essential jobs. The Fault Tolerance Plugin has been implemented to support larger systems once the model-checker has been updated to scale for larger systems.

The scheduling mechanism considers prerequisites for each job:

- Deadlock-freedom, divergence-freedom and semifairness jobs will only run if all processes are fully defined in the CML model;
- Full fault tolerance jobs will only run if:
 - All processes are defined;
 - All processes are deadlock-free, divergence-free and semifair; and
 - The limited fault tolerance verification has not yet been run or if the process is limited fault-tolerant.
- Limited fault tolerance jobs will only run if:
 - All processes are defined;
 - All processes are deadlock-free, divergence-free and semifair; and
 - The full fault tolerance verification has not yet been run or if the process is *not* full fault-tolerant.

The jobs for deadlock-freedom, divergence-freedom and semifairness verifications can run in any order, but all three must run before limited and full fault tolerance verification jobs. Full and limited fault tolerance verification jobs can run in any order.

5 Conclusions

In this deliverable, we have presented the static fault analysis plugin for Artisan Studio and the fault tolerance plugin for the Symphony tool. The former allows a SysML modeller the ability to: define an SoS model in terms of its structure; provide failure data for the constituent elements, propagation of SoS errors through to SoS failures; and analyse the resulting model. The tool support constitutes SysML profile support for the newly developed Fault Analysis Architectural Framework, translation to XML and the invocation of an external fault analysis tool. The latter plugin provides support for a CML modeller to verify the correctness of a recovery mechanism, assuring the fault tolerance of the system model.

In the final section of this deliverable, we identify some areas of future work. The Fault Analysis Tool for Artisan Studio provides a proof of concept for fault analysis at the architectural modelling level. We see three main areas of future work in relation to this tool:

- In Section 3.4 we identify the relationship of the Fault Analysis Architectural Framework with previous work undertaken in fault modelling. Whilst this is a clear piece of future work, we consider a more substantial piece of work would be to consider the role of fault modelling and analysis in the SoS engineering guidelines produced in [HPH⁺13]. We may consider, for example the link between the needs and requirements of an SoS (and its constituents) and the faults, errors and failures of an SoS. Related to this is the possibility of using the results of analysis as a means to generate test cases. Both of these areas require considerable additional work.
- In the current implementation, the results of the Fault Analysis Tool are given in the form of an HTML page generated by HiP-HOPS. Future work would be the reporting of results within the SysML model, as indicated in the tool chain in Figure 13. This requires the matching of the output of the HiP-HOPS tool with the model elements in the original SysML model, and a clear understanding of how best to interpret the results in the visual language of SysML.
- Some aspects of the HiP-HOPS functionality are not yet supported in the Fault Analysis Tool, in particular there is no support for multiple *Perspectives* or for defining *Hazards*. Including support for Perspectives would allow hardware models to be separated from software models, thus enabling separation of concerns and enriching the opportunities for defining fault tolerance and optimising models. Hazards allow more complex failure scenarios to be defined. In future versions of the tool it would be valuable to

explore extensions to the Fault Analysis Tool that include more advanced features of HiP-HOPS such as these.

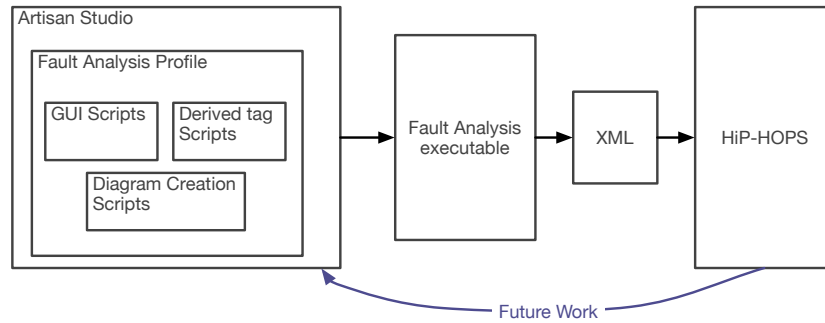


Figure 13: Future work in the Fault Analysis tool chain

For the FT plugin, we showed in Section 4 that the original definition of fault tolerance is achieved by a refinement-check, and we used an equivalent deadlock-freedom verification due to a limitation of the CML model-checker. The equivalence of these properties is only valid for deadlock-free processes. To overcome this limitation, we need the model-checker to be able to verify CML refinements. In this case, we would need to modify the plugin to run two property verifications and two refinement verifications (instead of five property verifications).

A Fault Analysis Architectural Framework Additional Diagrams

A.1 Fault Analysis Architectural Framework Viewpoint Definitions

For each viewpoint, we define the model elements that may be used in that view:

Model and Optimisation Definition Viewpoint

The *Viewpoint Definition View* for a *Model and Optimisation Definition Viewpoint* is given in Figure 14. The *Model and Optimisation Definition Viewpoint* contains a single *Fault Analysis Model*, which in turn contains a single *SoS* element and an optional *Optimisation Parameters* element, which may contain an arbitrary number of *Objective* elements.

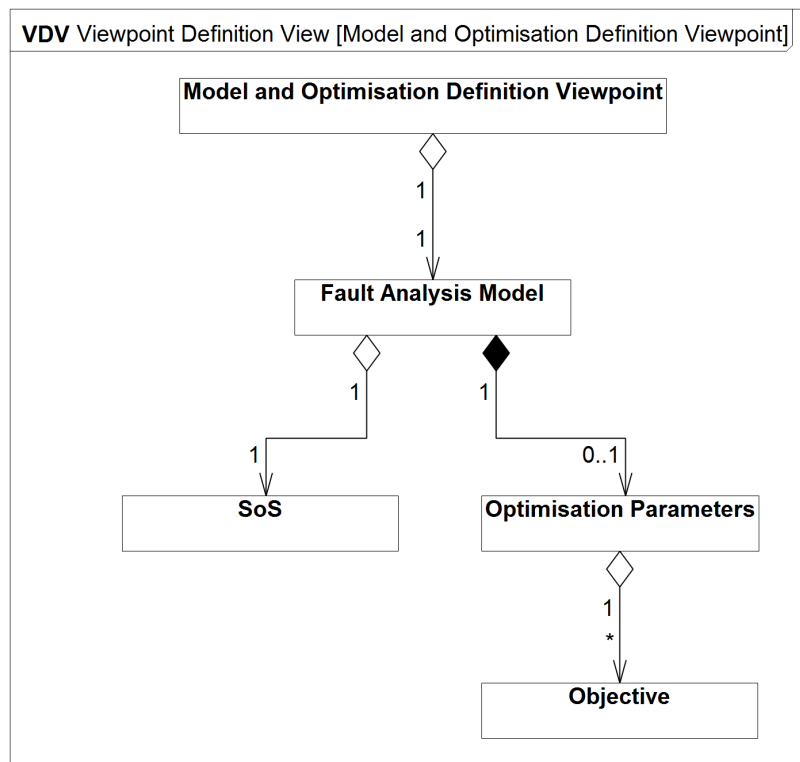


Figure 14: Fault Analysis Architectural Framework Profile Viewpoints

SoS Definition Viewpoint

The *Viewpoint Definition View* for a *SoS Definition Viewpoint* is given in Figure 15. The viewpoint contains a single *SoS* element, one or more *Constituent* elements and zero or more *Line* elements.

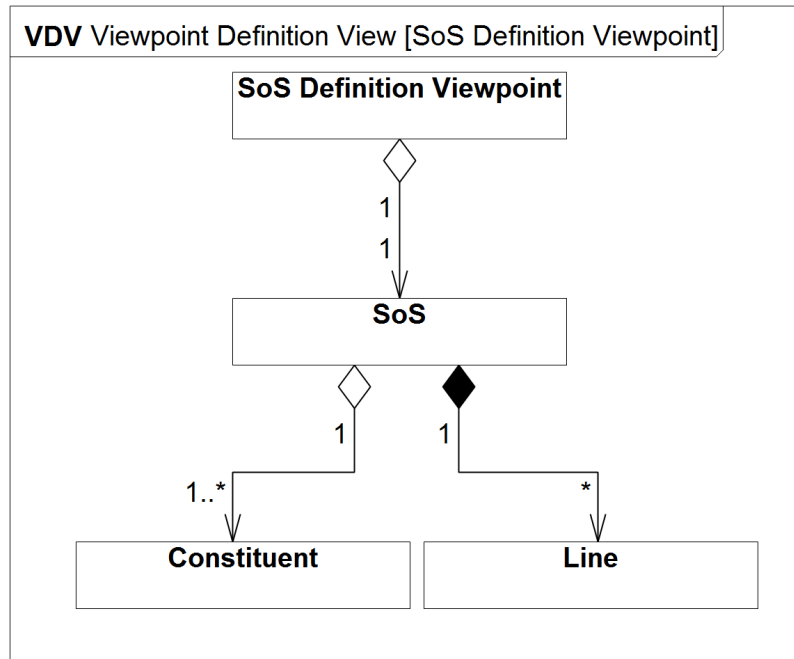


Figure 15: Fault Analysis Architectural Framework Profile Viewpoints

SoS Connections Viewpoint

The *Viewpoint Definition View*, for a *SoS Connections Viewpoint*, is given in Figure 16. The viewpoint contains a single *SoS*, the *Constituents* of the *SoS* and the *Lines*¹⁷ of the *SoS*. Each *Line* needs to be associated with a set of *Connector* elements that connect *Constituents* via the *Ports* that they expose.

¹⁷Note that in the current Fault Analysis Tool *Line* elements *must* be defined. In future releases we intend to make this optional for simple cases.

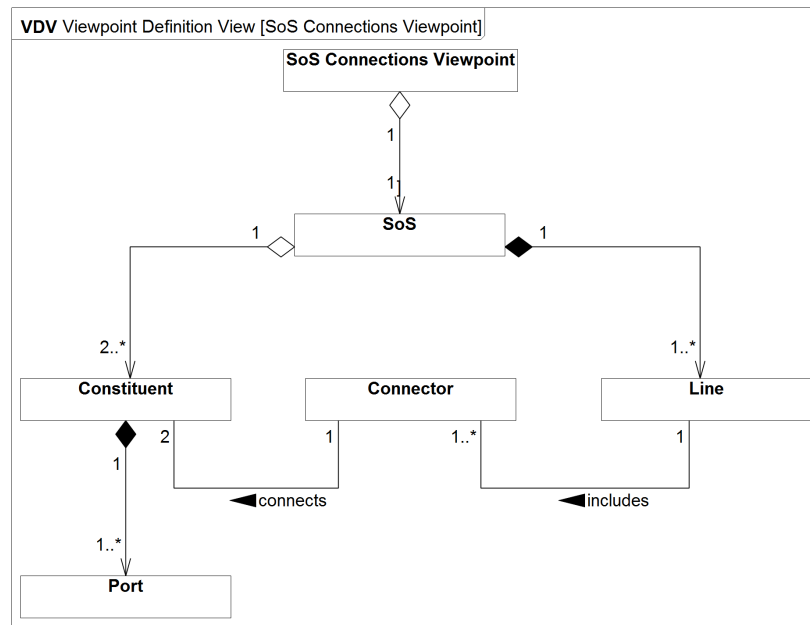


Figure 16: Fault Analysis Architectural Framework Profile Viewpoints

Implementation Definition Viewpoint

Figure 17 depicts the *Viewpoint Definition View* for the *Implementation Definition Viewpoint*. The viewpoint contains either a *Constituent* or *Component* element. A collection of *Implementation* elements is defined for that *Constituent* or *Component*. An *Implementation* contains zero or more (sub-) *Component* elements and zero or more *Line* elements.

Implementation Connections Viewpoint

The *Viewpoint Definition View*, for an *Implementation Connections Viewpoint*, is given in Figure 18. The viewpoint contains a single *Implementation*, the *Components* of the *Implementation* and the *Lines*¹⁸ of the *Implementation*. Each *Line* needs to be associated with a set of *Connector* elements that connect *Components* via the *Ports* that they expose.

¹⁸Note that in the current Fault Analysis Tool *Line* elements *must* be defined. In future releases we intend to make this optional for simple cases.

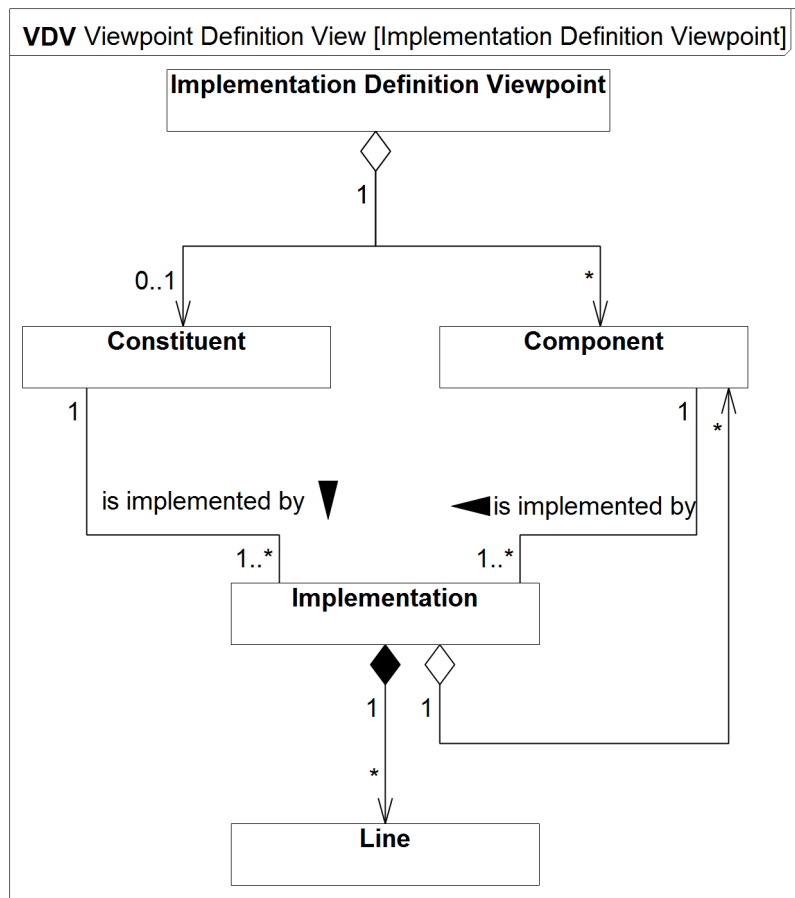


Figure 17: Fault Analysis Architectural Framework Profile Viewpoints

Implementation Failure Definition Viewpoint

Figure 19 shows the *Viewpoint Definition View* for the *Implementation Failure Definition Viewpoint*. The viewpoint contains an *Implementation* element. An *Implementation* contains zero or more *Basic Event* and *Output Deviation* elements.

Line Definition Viewpoint

The *Viewpoint Definition View* for the *Line Definition Viewpoint* is given in Figure 20. The viewpoint contains a single *Line* element, which contains two or more *Line Ends*. Each *Line End* represents a *Port* and contains zero or more *Propagation Logic* elements.

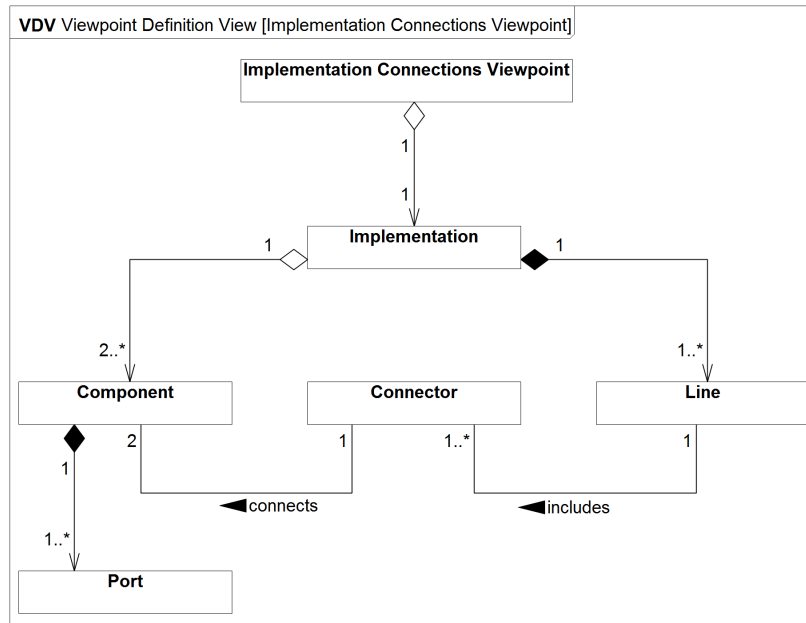


Figure 18: Fault Analysis Architectural Framework Profile Viewpoints

Failure Class Definition Viewpoint

The *Viewpoint Definition Viewpoint* for the *Failure Class Definition Viewpoint* is given in Figure 21. The viewpoint contains only a collection of *Failure Class* model elements.

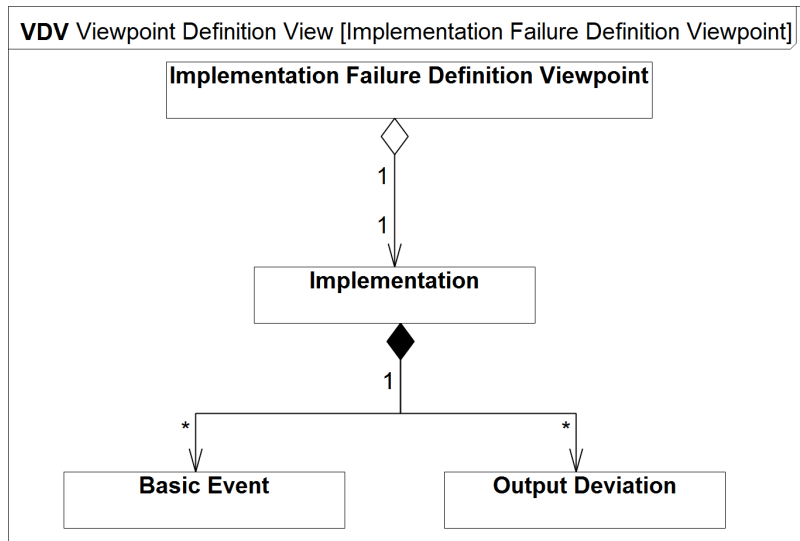


Figure 19: Fault Analysis Architectural Framework Profile Viewpoints

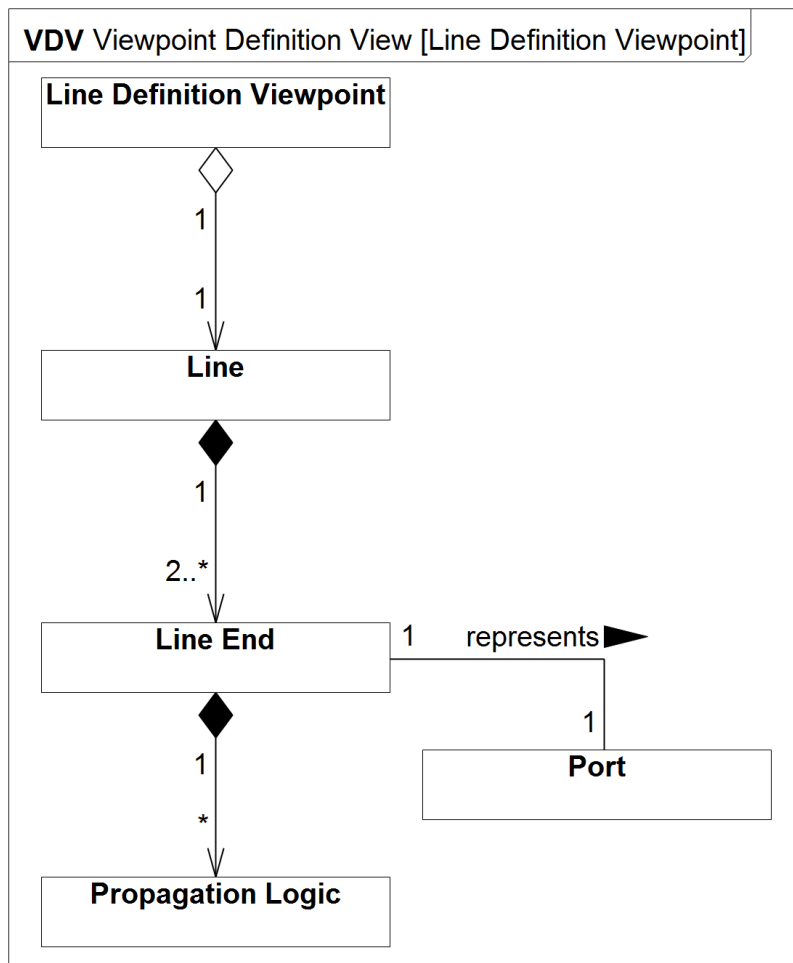


Figure 20: Fault Analysis Architectural Framework Profile Viewpoints

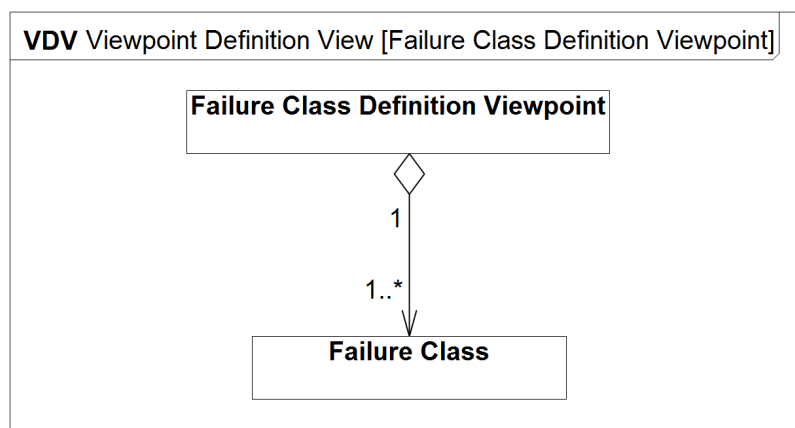


Figure 21: Fault Analysis Architectural Framework Profile Viewpoints

A.2 Rule Definition Views

Finally, we may define a collection of rules for the viewpoints in the Fault Analysis Architectural Framework. This set of rules is similar to a static semantics for the profile and may be encoded in Artisan Studio through ergonomic profiling and static model-correctness checks. We define a set of rules broadly split up into the viewpoints of the profile; some rules naturally refer to model elements in other diagrams

Model and Optimisation Definition Viewpoint

RDV Rule Definition View [Model and Optimisation Definition Viewpoint]

Rule 1: There must exist exactly one Model and Optimisation Definition Viewpoint in a Fault Analysis Model.

Rule 2: Only one Fault Analysis Model element may exist in a Fault Analysis Model.

Rule 3: Exactly one SoS must be defined for the Fault Analysis Model.

Rule 4: Tool name and Tool version are optional, but if one is defined then both must be (NB: these are automatically defined using derived tags).

Rule 5: The Target and Goal of an Objective are compulsory.

SoS Definition Viewpoint

RDV Rule Definition View [SoS Definition Viewpoint]

Rule 1: There must exist exactly one SoS Definition View per Fault Analysis Model.

Rule 2: The SoS defined in the SoS Definition View must be identified by the Fault Analysis Model of the Model and Optimisation Definition View.

Rule 3: The SoS must contain at least one Constituent.

Rule 4: If the SoS has more than one Constituent, it must have at least one Line (NB: In future versions of the Fault Analysis Tool Lines may be implicitly defined where possible via Connectors in the SoS Connections View).

Rule 5: The Risk time on a Constituent (if defined) must be lower than the Risk time of the owning Fault Analysis Model.

SoS Connections Viewpoint

RDV Rule Definition View [SoS Connections Viewpoint]

- Rule 1: There must exist exactly one SoS Connections View per Fault Analysis Model.
- Rule 2: The SoS defined in the SoS Connections View must be identified by the Fault Analysis Model of the Model and Optimisation Definition View and defined in an SoS Definition View.
- Rule 3: All of the Constituent model elements identified in the SoS Definition View must be included in the SoS Connections View.
- Rule 4: All of the Line model elements identified in the SoS Definition View must be included in the SoS Connections View (NB: In future versions of the Fault Analysis Tool, Lines may be implicitly defined where possible via Connectors in the SoS Connections View).
- Rule 5: All Connectors must be connected to a Port on a Constituent.
- Rule 6: All Lines included on an SoS Connections View must be associated with at least one Connector.

Implementation Definition Viewpoint

RDV Rule Definition View [Implementation Definition Viewpoint]

- Rule 1: The 'top-level' model element must be either a Constituent or a Component.
- Rule 2: There must exist an Implementation Definition View for each Constituent and each Component.
- Rule 3: There may be multiple Implementations (and Implementation Definition Viewpoints) per Constituent or Component.
- Rule 4: One and only one Implementation must have their hiphops-current tag set to TRUE.
- Rule 5: Only one 'top-level' model element may be included per Implementation Definition View.
- Rule 6: The Risk time on a Component (if defined) must be lower than the Risk time of the owning Fault Analysis Model.

Implementation Connections Viewpoint

RDV Rule Definition View [Implementation Connections Viewpoint]

Rule 1: For every Implementation model element that has one or more Components, there must exist exactly one Implementation Connections View.

Rule 2: Only one Implementation may be included in a Implementation Connections View.

Rule 3: The Implementation defined in the Implementation Connections View must be defined in an Implementation Definition View.

Rule 4: All of the Component model elements identified in the Implementation Definition View for the Implementation in this Implementation Connections View must be included in the Implementation Connections View.

Rule 5: All of the Line model elements identified in the Implementation Definition View for the Implementation in this Implementation Connections View must be included in the Implementations Connections View (NB: In future versions of the Fault Analysis Tool, Lines may be implicitly defined where possible via Connectors in the Implementations Connections View).

Rule 6: All Connectors must be connected to a Port on a Component.

Implementation Failure Definition Viewpoint

RDV Rule Definition View [Implementation Failure Definition Viewpoint]

Rule 1: There must be exactly one Implementation in an Implementation Failure Definition View.

Rule 2: The Failure Class of an Output Deviation must be defined in the Failure Class Definition View associated with the Fault Analysis Model (NB: this rule is not currently enforced by the Fault Analysis Tool).

Rule 3: The Failure expression of an Output Deviation has the following constraints:

The Failure expression is written in Boolean logic, using operators AND and OR (NB: future versions may also support Boolean operators NOT and XOR and Pandora temporal operators PAND, POR and SAND).

Operands can be:

1. The Name of a Basic Event defined within the same Implementation Failure Definition Viewpoint.
2. Input deviations of format : <Failure Class>-<Port> where <Port> must be defined within the same Constituent/Component and <Failure Class> must be defined within the Failure Class Definition View associated with the Fault Analysis Model
- (3. A Local or Global Goto -- *not currently supported*).
- (4. An exported deviation from the target of an allocation link -- *not currently supported*).

Rule 4: The Unavailability formula of a Basic Event must be a valid HiP-HOPS Unavailability formula: see HiP-HOPS User Manual.

Line Definition Viewpoint

RDV Rule Definition View [Line Definition Viewpoint]

Rule 1: There must exist at least two Line Ends on a Line Definition View.

Rule 2: If the Line is owned by an SoS, then each Port represented by a Line End must exist on a Constituent owned by that SoS.

Rule 3: If the Line is owned by an Implementation, then each Port represented by a Line End must exist on a Component owned by that Implementation.

Rule 4: If a Propagation Logic element has a Failure Class, this must be defined in the Failure Class Definition View associated with the Fault Analysis Model (NB: this rule is not currently enforced by the Fault Analysis Tool).

Rule 5: All Ports referred to in a Port expression of a Propagation Logic element must be associated with a Line End that is also defined in the same Line Definition View.

Rule 6: All Failure Classes referred to in a Port expression of a Propagation Logic element must be defined in the Failure Class Definition View associated with the Fault Analysis Model.

Rule 7: The set of Line Ends defined for a Line must be those that represent Ports that are connected to each other in the respective Connections View.

Failure Class Definition Viewpoint

RDV Rule Definition View [Failure Class Definition Viewpoint]

Rule 1: There must exist one Failure Class Definition View for a Fault Analysis Model.

Rule 2: There must exist at least one Failure Class model element.

References

- [ADP⁺13] Zoe Andrews, André Didier, Richard Payne, Claire Ingram, Jon Holt, Simon Perry, Marcel Oliveira, Jim Woodcock, Alexandre Mota, and Alexander Romanovsky. Report on Timed Fault Tree Analysis – Fault modelling. Technical report, COMPASS Deliverable, D24.2, September 2013.
- [AIP⁺14] Zoe Andrews, Claire Ingram, Richard Payne, Alexander Romanovsky, Simon Perry, and Jon Holt. Traceable Engineering of Fault-Tolerant SoSs. In *International INCOSE Symposium, Las Vegas, USA*, June-July 2014. To appear.
- [ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.
- [APR⁺13] Zoe Andrews, Richard Payne, Alexander Romanovsky, André LR Didier, and Alexandre Mota. Model-based development of fault tolerant systems of systems. In *2013 IEEE International Systems Conference (SysCon 2013)*, Orlando, USA, April 2013.
- [CDIK13] Robin Cressent, Pierre David, Vincent Idasiak, and Frederic Kratz. Designing the database for a reliability aware Model-Based System Engineering process. *Reliability Engineering & System Safety*, 111(0):171 – 182, 2013.
- [CIKD11] R. Cressent, V. Idasiak, F. Kratz, and P. David. Mastering safety and reliability in a model based process. In *Reliability and Maintainability Symposium (RAMS)*, Lake Buena Vista, Florida, USA, January 2011.
- [CMCP13] Joey W. Coleman, Anders Kaels Malmos, Luís D. Couto, and Richard Payne. Third release of the COMPASS Tool — Developer Documentation. Technical report, COMPASS Deliverable, D31.3b, November 2013.
- [DIK10] Pierre David, Vincent Idasiak, and Frédéric Kratz. Reliability study of complex physical systems using SysML. *Reliability Engineering System Safety*, 95(4):431–450, 2010.

- [FMDW13] Adalberto C Farias, Alexandre C Mota, André LR Didier, and Jim Woodcock. Model Checking Support. Public D33.1, COMPASS, September 2013.
- [GPKV11] Alain Griffault, Gérald Point, Fabien Kuntz, and Aymeric Vincent. Symbolic computation of minimal cuts for AltaRica models. Technical Report Research Report RR-1456-11, LaBRI, Université de Bordeaux, September 2011.
- [HCW02] Mats P.E. Heimdahl, Yunja Choi, and Mike Whalen. Deviation analysis through model checking. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 37–46, 2002.
- [HHP⁺13] Finn Overgaard Hansen, Stefan Hallerstede, Simon Perry, Jon Holt, and Klaus Kristensen. Initial Report on Guidelines for Architectural Level SoS Modelling. Technical report, COMPASS Deliverable, D21.2, March 2013.
- [HPH⁺13] Jon Holt, Simon Perry, Finn Overgaard Hansen, Alvaro Miyazawa, Klaus Kristensen, and Ralph Hains. Initial Report on Guidelines for Systems Engineering for SoS. Technical report, COMPASS Deliverable, D21.3, September 2013.
- [JHMW06] Anjali Joshi, Mats P.E. Heimdahl, Steven P. Miller, and Mike W. Whalen. Model-Based Safety Analysis. Technical Report NASA/CR-2006-213953, University of Minnesota, Minneapolis, Minnesota 55455, February 2006.
- [MCR⁺12] F. Mhenni, J.-Y. Choley, A. Rivière, N. Nguyen, and H. Kadima. SysML and Safety Analysis for Mechatronic Systems. In *Mechatronics-REM 2012*, Paris, France, November 2012.
- [MNKC13] F. Mhenni, N. Nguyen, H. Kadima, and J.-Y. Choley. Safety Analysis Integration in a SysML-Based Complex System Design Process. In *2013 IEEE International Systems Conference (SysCon 2013)*, Orlando, USA, April 2013.
- [Per13] Simon Perry. Report on Modelling Patterns for SoS Architectures. Technical report, COMPASS Deliverable, D22.3, February 2013.
- [PWP⁺11] Y. Papadopoulos, M. Walker, D. Parker, E. Råde, R. Hamann, A. Uhlig, U. Grätz, and R. Lien. Engineering Failure Analysis and Design Optimisation with HiP-HOPS. *Engineering Failure Analysis*, 18:590–608, 2011.

- [Ros97] Andrew William Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.