Model Engineering in a Modular PSA

Thomas Friedlhuber

Thesis for the degree of

DOCTEUR DE L'ECOLE POLYTECHNIQUE

in Computer Science

handed in by

Thomas FRIEDLHUBER

Model Engineering in a Modular PSA

Defended on October 13, 2014

Jury Members:

Frédéric BOULANGER	Supélec	Rapporteur
Jean-Jacques LESAGE	ENS Cachan	Rapporteur
Yiannis PAPADOPOULOS	University of Hull	Rapporteur
Ioannis PAPAZOGLOU	National Center for Scientific Research	Examiner
Renaud ANNAVAL	EDF CIPN	Examiner
Antoine RAUZY	Ecole Polytechnique	Ph.D adviser
Mohamed HIBTI	EDF R&D	Co-adviser

Thèse présentée pour le grade de

DOCTEUR DE L'ECOLE POLYTECHNIQUE

en Informatique

par

Thomas FRIEDLHUBER

Ingénierie des modèles dans une EPS modulaire

Soutenue le 13 octobre, 2014

Membres du jury:

Frédéric BOULANGER	Supélec	Rapporteur
Jean-Jacques LESAGE	ENS Cachan	Rapporteur
Yiannis PAPADOPOULOS	University of Hull	Rapporteur
Ioannis PAPAZOGLOU	National Center for Scientific Research	Examinateur
Renaud ANNAVAL	EDF CIPN	Examinateur
Antoine RAUZY	Ecole Polytechnique	Directeur de Thèse
Mohamed HIBTI	EDF R&D	Co-encadrant

Abstract

For the purpose of PSA (Probabilistic Safety Analysis) for complex industrial systems, often PSA models in the form of fault and event trees are developed to model the risk of unwanted situations (hazards). While the recent decades, PSA models have gained high acceptance and have been developed massively. This led to an increase in model sizes and complexity. Today, PSA models are often difficult to understand and maintain.

This manuscript presents the concept of a modular PSA. A modular PSA tries to cope with the increased complexity by the techniques of modularization and instantiation. Modularization targets to treat a model by smaller pieces (the "modules") to regain control over models. Instantiation aims to configure a generic model to different contexts. Both try to reduce model complexity.

A modular PSA enables to apply new functionality for model management. Current PSA model management is often characterized to be limited or inefficient. This manuscript shows new methods to manage the evolution (versions) and deviations (variants) of PSA models in a modular PSA. The concepts of version and variant management are presented in this thesis. In this context, a method to compare and fusion PSA models is precised. Model comparison provides important feedback to model engineers and model fusion targets to combine the work from different model engineers (concurrent model engineering).

Apart from model management, methods to understand the content of PSA models are presented. The methods focus on highlighting the dependencies between modules rather than their contents. Dependencies are automatically derived from a model structure. They express relations between model objects (for example a fault tree may have dependencies to basic events). The visualization of those dependencies (for example in form of a model cartography) can constitute a crucial aid to model engineers for understanding complex interrelations in PSA models.

Within the scope of this thesis, a software named "Andromeda" has been developed at EDF R&D to test and evaluate the concepts around a modular PSA. Andromeda is based on a modular and extensible architecture that can be customized to specific needs of customers. Apart from research interest, it has recently gained industrial interest. Andromeda has potential to augment existing PSA tools by specific functionality they lack and to promote common modeling standards and techniques within the international PSA community.

Résumé

Dans le cadre de l'EPS (Étude Probabiliste de Sûreté) pour des systèmes industriels complexes, souvent les modèles EPS sont développés sous la forme d'arbres de défaillances et d'événements pour modéliser les risques des situations non attendues (hasard). Au cours des dernières décennies, les modèles EPS ont été de plus en plus acceptés et développés. Cela a conduit à un accroissement de la taille des modèles et de leur complexité. Aujourd'hui, les modèles EPS sont souvent difficiles à comprendre et maintenir.

Ce manuscrit présente le concept de l'EPS modulaire. Une EPS cherche à faire face à la complexité croissante par les techniques de modularisation et d'instanciation. La modularisation vise à traiter les modèles en petits morceaux (les "modules") pour mieux comprendre ces modèles. L'instanciation a pour but de configurer un modèle générique à des contextes différents.

Une EPS modulaire permet d'appliquer des nouvelles fonctionnalités pour gérer les modèles. La gestion actuelle des modèles EPS a souvent ses limites. Cette thèse montre de nouvelles méthodes pour gérer les évolutions (versions) et les variations (variantes) des modèles EPS dans une EPS modulaire. Les concepts de versions et de variantes sont présentées dans cette étude. Une méthode de comparaison et fusion des modèles EPS est précisée. La comparaison des modèles apporte des informations intéressantes aux ingénieurs de modèles et la fusion des modèles permet de combiner leurs différents travaux.

Des méthodes pour comprendre le contenu des modèles EPS sont également introduites. Ces dernières mettent l'accent sur les dépendances entre les modules plutôt que sur leur contenu. Les dépendances viennent automatiquement d'une structure de modèle. Elles expriment les relations entre les objets de modèle (par exemple un arbre de défaillances peut créer des dépendances vers des événements de base). Les possibilités de visualiser ces dépendances (par exemple sous la forme d'une carthographie de modèles) peuvent être une aide précieuse pour les ingénieurs de modèle afin d'analyser les correlations complexes dans les modèles EPS.

Dans le cadre de cette thèse, un logiciel nommé "Andromeda" a été développé au sein du département Recherche et Développement d'EDF pour tester et évaluer les concepts autour d'une EPS modulaire. Andromeda se base sur une architecture modulaire et extensible qui peut être adaptée aux besoins spécifiques des utilisateurs. En plus de l'intérêt de la recherche, Andromeda a gagné récemment celui de l'industrie. Il a le potentiel de compléter les outils EPS actuels par des fonctionnalités spécifiques et de faire appliquer à la communauté internationale EPS des techniques et normes de modélisation communes.

Acknowledgements

First and foremost, I would like to thank my supervisor Mohamed Hibti, who guided me during my time at EDF R&D. I really appreciated to work with him, his motivation to explore new areas and I enjoyed to listen to his technical and philosophical statements about Linux, Open Source and in particular Emacs.

Next, I want to express my gratitude to Antoine Rauzy. His strong technical skills in probabilistic safety assessment and computer science on one side and his leading capacities influenced my time at EDF and LIX considerably.

I greatly enjoyed working in the department MRI at EDF. I would like to thank Anne-Marie Bonnevialle and Philippe Nonclercq from EDF for their great management efforts. My gratitude also to Marc Bouissou, Valentin Rychkov, Dominique Vasseur, Hassane Chraibi, Richard Quatrain, Michel Balmain, Alain Sibler, Gilles Deleuze and Yves Dien for the interesting discussions and to my roommates Johanna Mérand and Le Duy Tu Duong. I also want to thank the EDF divisions CNEN, SEPTEN, CPIN and DIN for collaborating work realized in the scope of the thesis.

My special thanks to Roland Donat for his deep technical understandings, his encouragements to delay lunch breaks for the benefit of running and his professional way he exercises his role as PDG in our startup company I recently founded with him.

I also appreciated discussing and working together with my colleagues from LIX, namely Tatjana Prosvirnova, Pierre-Antoine Brameret, Abraham Cherfi and Michel Batteux.

Further, I thank the participants of the seminaire "SDFX", in particular Agnès Lanusse for her advanced insights on model engineering.

I would like to thank the reviewers of my thesis, Frédéric Boulanger, Jean-Jacques Lesage and Yiannis Papadopoulos for their constructive and pertinent comments and experience as well as the examiners Ioannis Papazoglou and Renaud Annaval.

Lastly, I profoundly thank my family, my girlfriend Audrey Depond and her family for their great support during the last three years. My special thanks to Audrey who encouraged me continuously to fully benefit from life and to travel around the world whenever possible even when writing a thesis.

Contents

CHAPTER 1

Introduction

The thesis *Model Engineering for a Modular PSA* has been launched in June 2011 for a period of three years in a partnership between EDF R&D and LIX. EDF R&D is the research institute of EDF (Electricité De France) [1], one of the world's largest producer of electricity. LIX [2] (Laboratoire d'Informatique de l'École Polytechnique) is the laboratory of computer science of Ecole Polytechnique [3], a French engineering school of higher education and research. The thesis was supervised by Prof. A. Rauzy (LIX) and M. Hibti (EDF R&D).

1.1 Context of the Thesis

Probabilistic risk assessment (PRA) (also referred to as probabilistic safety analysis (PSA)) targets to evaluate risk by the means of probabilistic studies. It finds application in various domains, e.g. at nuclear engineering, transport, finance etc. in order to analyse occurrence characteristics of unwanted situations (so-called "*hazards*").

In the classical approach, to model the different risk scenarios, PRA is based on safety models in the form of fault- and event trees. Those models are referred to as PSA models. They are typically engineered by model- and safety engineers (often with support of experts of various domains).

Since PRA became an internationally accepted (and required) approach to assess risk of nuclear power plants, PSA models have been developed massively in nuclear engineering facilities, so at EDF.

PSA models for nuclear power plants exhibit some particular characteristics. At first, the life-cycles of PSA models are typically very long. Risk analysis is already required in the design phase of nuclear plants to reveal safety problems before the actual plant construction starts. During the construction phase (while a nuclear plants gets physically built), PRA is a method to monitor modifications or installations, which had not been considered at design phase. Once the plant construction is finished, the operational phase begins (while a plant produces energy). Slight modifications of the nuclear plant during this phase

are typical to enhance safety or productivity aspects. PSA models are needed to verify and justify any modifications. Finally, when a nuclear plant exceeds its life-time, the deconstruction phase begins. PSA models during deconstruction phase help to take the right decisions. In total, PSA models are required for a period of at least 70 years.

The new challenge of PSA models is the management of knowledge, people and software to enable PSA model engineering throughout long life cycles. Throughout their life-cycles, PSA models are subject to extensions to encode stricter safety requirements or new technical enhancements. Safety requirements are imposed by safety authorities. In France, this is the ASN (Autorité de Sûreté Nucléaire). Any extension of PSA models leads typically to more detailed and more complicated models. In the recent years, the number of model components (the number of objects in PSA models) increased dramatically. PSA models for nuclear plants contain by far more components compared to models used in other fields. They are considered to be one of the most complex models regarding their sizes.

Complex models can impose difficulties on developing them any further. A model engineer is required to understand a PSA model, before taking the right modification actions. Long life cycles of PSA models complicate the situation: Model engineers may change job positions or even leave a company, what aggravates knowledge transfer. Therefore, models are required to be designed in an understandable and clear way.

Another trend can be observed: Whereas PSA models initially have been created by one single engineer, nowadays models (because of their complexity) often require multiple developers to work on the same model at a time (otherwise, development would take too much time). New requirements on software tools and functionality are imposed to enable a so-called "*concurrent model engineering*" to synchronize different development actions.

1.2 Objective of the Thesis

The objective of this thesis is to investigate new modeling concepts and software tools around a so-called "*Modular PSA*" in order to improve efficiency and effectiveness of PSA model engineering in a challenging context. The context is characterized by developing large, complex PSA models of long life cycles by different engineers at a time. The developed concepts may be considered to enhance existing guidelines and PSA development processes to regain "*control*" over PSA models.

The thesis responds to the problematic on two levels, a conceptional level and a software level. The conceptional level investigates concepts and methods of model engineering, whereas the software level analyses software functionality and the composition of tool chains [1].

The thesis was influenced by two preexisting achievements. The first is the development of an open format to store PSA models, the so-called "Open PSA Model Exchange Format" [4], which permits to work with real-life PSA models at EDF in a clear and structured manner. The second achievement are general thoughts and ideas to treat PSA models more modular, what is referred to as *modular PSA*. This idea was first investigated in [5].

A first achievement of the thesis is the introduction and specification of a modeling framework for a modular PSA. The principle goal of the framework is the mitigation of PSA model complexity. This is obtained by two concepts: The first one makes it possible to treat models by smaller pieces - so-called "modules". The second one is the possibility to instantiate PSA models according to specific contexts. The framework is a generic framework in the sense, that it is not limited to the domain of PSA models.

[1] A tool chain refers to a total of tools and their interactions to obtain a common goal, for example PSA engineering.

Other type of models can be realized within the same framework. This is necessary to incorporate other models than fault- and event trees to risk assessment.

Next, an adequate way to "manage" PSA models in a modular PSA is presented. Model management refers to the general strategy, concepts and functionality with respect to the development, maintenance and storage of PSA models. It provides basic functionality such as model comparison and model fusion. In short, model comparison permits to compare two PSA models and to visualize differences between them. And model fusion permits integrating components of one model in another model. Further, concepts to manage variants and versions of PSA models are demonstrated. Variants permit to adapt PSA models to specific contexts without directly modifying the actual model contents. They can for example be used to express plant specific circumstances in PSA models. Version management tracks and maintains the evolution of models. It permits for example to recover ancient versions of individual model parts or of the whole model.

Another objective of the thesis is to present methods for better understanding the composition of PSA models as a whole, of individual model objects and their inter-relations, e.g. which basic events are "used" in which fault trees. In the modular approach, relations between model objects are referred to as "dependencies". Functionality is presented to analyse and to highlight those dependencies. A navigation concept is shown to explore a model by its dependencies and composition. In the same context - to improve clarity of models - an application based on graph theory is presented to visualize the fault tree layer of PSA models on a system level and to highlight safety critical components in a compact and intuitive way. The application is an opportunity to bridge knowledge between system- and safety analysts.

The modular PSA framework and the introduced methods justify to reconsider the whole PSA model engineering process from "*A-Z*". The overall engineering process consists of several phases such as model design, creation and verification. Another objective is to investigate new conceptual aspects, that may optimize the different phases of model engineering. A special focus is given to the automatic generation of model parts, to concurrent model engineering and outsourcing strategies and to the automation of development processes.

Automatic model generation targets to generate parts of PSA models automatically, e.g. from another modeling language. A new model type named *event sequence diagram* (ESD) is presented for this purpose. At EDF, ESD models are already used to design (to specify) event trees, though in a rather basic manner. In future, ESD models may not only serve to design event trees, but also to validate coherence between event trees and design and to generate event trees automatically from design. In conjunction with fault tree generation methods (which already exist at EDF), event tree generation can automate large parts of PSA development.

Model outsourcing gives companies the opportunity to externally develop parts of their PSA models. This, however, requires new methods and a strategy to synchronize the different model development actions. Fortunately, a modular conception of PSA models facilitates to cooperate on the same model. A method of concurrent model engineering to synchronize work between different developers is presented.

Finally, a scripting interface is demonstrated, that can automate modeling activities and processes. Scripts express modeling instructions (e.g. the creation of a fault tree) in textual, compact form. They are testable, reproducible and efficient in their execution [2].

Particularly for this thesis, a software named *Andromeda* has been developed. Initially planned as EDF R&D research software to evaluate and test the developed modular PSA concepts, Andromeda found soon industrial interest beside the thesis. The main interests are to improve readability of large, complex PSA models and to be able to compare two PSA models in order to detect and cross-check modifications.

[2] The execution of a script is the application of its instructions.

Beside the implementation of the modular PSA approach and its concepts, Andromeda provides interesting characteristics of an open, generic and extensible modeling platform, prepared to cope with altering software demands throughout long life-cycles of PSA models. It reflects a tooling philosophy to customize tool chains to the often specific needs of companies and to efficiently adapt them whenever needed.

1.3 Organization of the Thesis

Chapter 2 presents an overview about methodologies used in probabilistic safety assessment.

Chapter 3 gives a brief historical overview about PSA usage and evolution since the 60's.

Chapter 4 points out nowadays and future challenges about PSA assessment.

Chapter 5 introduces the preliminary works of the "Open PSA Model Exchange Format" and the "modular PSA" idea.

Chapter 6 describes the refined modular PSA framework worked out in this thesis, in particular the definition of modules and the context sensitive model instantiation technique.

Chapter 7 presents concepts and functionality concerning the development, maintenance and storage of PSA models in a modular PSA.

Chapter 8 demonstrates special functionality to improve readability of PSA models.

Chapter 9 reconsiders the overall model engineering process in a modular PSA.

Chapter 10 reveals architectural insights of the modeling software Andromeda (that has been developed at EDF R&D in the scope of this thesis).

PART I

INTRODUCTORY PART

CHAPTER 2

PSA Concepts

Probabilistic Safety Analysis in the classical approach is based on fault and event tree models. Both encode Boolean formulas and can be used in conjunction with fault or event tree linking methods [6]. The classical approach gained high acceptance in safety assessment for critical systems such as nuclear power plants.

In this section, the composition of PSA models in the classical way (fault- and event trees) is presented. Further, the concept of probabilistic risk assessment is introduced.

Section 2.1 presents PSA models in their classical form based on fault and event trees.

Section 2.2 introduces to the concept of probabilistic risk assessment.

Section 2.3 concludes this chapter.

2.1 PSA Models

PSA models are probabilistic models, that encode the likelihood of unwanted situations, so-called "hazards". In the classical form, PSA models consist of fault and event trees.

Note that this thesis focuses on so-called "static" PSA models. Those kind of models cannot express temporal dependencies between model objects (contrary to so-called "dynamic" models). The explications in the following sections refer to the static definition of PSA models, which consist of static fault trees and static event trees.

2.1.1 Fault Trees

A fault tree (static definition) encodes a Boolean formula over events in order to express the likelihood of a so-called "top event". The top event represents an unwanted situation (a hazard).

Fault trees follow a deductive approach: Starting from the top event, combinations of causative events are investigated, recursively. Events that occur represent "failures" (this is the inverse logic of safety analysts).

In PSA models of EDF, thousands of fault trees are used.

The fault tree method is well-documented. Exhaustive documentation can be found at [7, 8, 9].

Fault trees are presented graphically in form of *directed acyclic graphs* (DAGs), referred to as *fault tree diagrams*. The nodes of a fault tree diagram represent events, denoted by a rectangle containing the event name and an event symbol. The different symbols are explained in the sequel of this section. The top-event is drawn at the top of the diagram. Its input events are drawn beneath, recursively.

Figure 2.1 shows a fault tree diagram for a fire hazard. The top event represents the fire hazard.

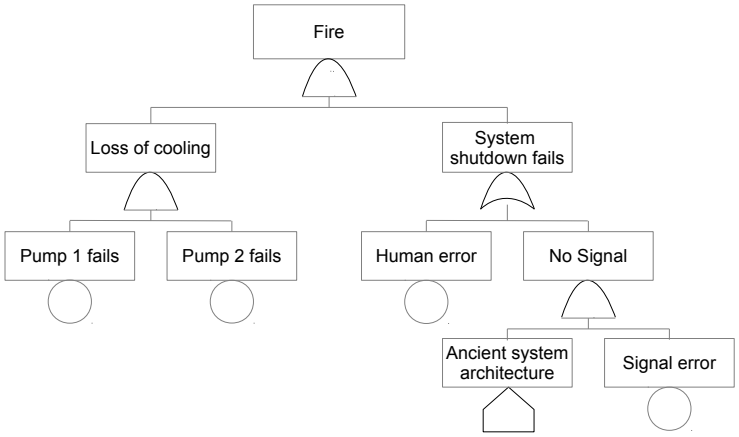
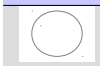


Figure 2.1. Example of a fault tree

Events not further investigated are either *basic events* or *house events*. Those are also referred to as *terminal events*. Basic events are associated with a probability distribution function describing the event occurrence likelihood. House events are Boolean constants, that are either *True* or *False* (typically "True" for event occurrence and "False" for non-occurrence, though certain PSA software may inverse this logic).

Table 2.1 shows the list of terminal events with their graphical representation.

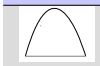
Table 2.1. Terminal events in fault trees.

Graphical symbol	Event	Characteristics
	Basic event	terminal event occurring with a certain probability
	House event	terminal event with constant event occurrence

Events that are further investigated are called *gates*. Gates are intermediate events, which are connected to one or multiple events, called the "gate inputs". Gates specify further a *gate type*. A gate type defines which combinations of failing / non-failing gate inputs will let the gate fail. Mathematically, it describes a Boolean function over gate inputs. Typical gate types with their graphical symbols are listed in Table 2.2.

Depending on literature, further gate types are considered. However, concerning static fault trees, more gate types will not give more opportunities to encode Boolean formulas. The number of different Boolean

Table 2.2. Overview of gate types.

Graphical symbol	Name	Characteristics
	AND gate	event occurrence in case all gate inputs occur
	OR gate	event occurrence in case at least one gate input occurs
	K/N gate	event occurrence if at least K out of N gate inputs occur
	NAND gate	event occurrence in case not all gate inputs occur
	NOR gate	event occurrence in case no gate input occurs
	XOR gate	event occurrence in case exactly one gate input occurs

formulas over n gate inputs is 2^{2^n}, e.g. for 3 gate inputs, there are already 256 different gate types (different Boolean functions). But due to functional completeness[1] of the *NAND* gate, any gate type can be expressed by a combination of *NAND* gates (gate types are explained in Figure 2.2). Also the set of *NOR* and *XOR* gates is functionally complete. Note that the set of *OR*, *AND* and *K/N* (minimum K out of N inputs fail) gates is not functionally complete.

Often the term *external gate* occurs. An external gate is a link to another gate, which "belongs" to a different fault tree. This allows to reduce the size of individual fault trees. This kind of modularization eases the work of model engineers (they can deal with smaller fault trees). However, concerning fault tree quantification, computational issues are not impacted (reduced) by the usage of external gates.

Another interesting characteristic is coherence: A fault tree is said *coherent* if the following condition holds: "given an event set S_1 that fails the top-event, any set S_2 with $S_1 \subseteq S_2$ also fails the top event" [11]. The interpretation is that failing events cannot "repair" a situation. Fault trees containing only gates of type *OR*, *AND* or *K/N* are always coherent. Note that the reverse statement is not necessarily true. Non-Coherent fault trees can dramatically impact (increase) computational issues at fault tree quantification. They complicate also the determination of upper / lower bounds for approximation errors (those errors result from neglecting "less important" fault tree parts during quantification).

2.1.2 Event Trees

Alike fault trees, *event trees* encode Boolean formulas. Contrary to fault trees, which follow a deductive (top-down) concept, event trees focus on the evolution of events and thus follow an inductive (bottom-up) concept.

In PSA models of EDF hundreds of event trees are used.

Starting from a so-called *initiating event* (the first event to consider in an accident scenario), all consequential events are derived, recursively. A consequential event is an event that occurs due to the occurrence / non-occurrence of another event.

In the case of nuclear power plants, consequential events are called *function events*. Each function event

[1] A set of Boolean operators is said to be functional complete if it can express (possibly by combination) any other Boolean operators [10].

represents a recovery action to mitigate a critical situation. The recovery action can be successful or failing. According to whether recovery actions are successful or not, different event evolutions (sequences) are deducted.

The deduction of events ends in so-called *sequences*. Each sequence describes one specific event evolution. Sequences can lead to so-called *consequences* which describe a certain system state. Different sequences can lead to the same consequence. Also, a sequence can lead to various consequences at a time.

Figure 2.2 shows an example of an event tree. The accident scenario is initiated by an obstacle that occurs in front of a bike (the obstacle occurrence is the initiating event). As mitigating actions (function events), the cyclist can brake in front and back. For avoiding a *crash* it is sufficient to have one working brake. In case both brakes fail, the accident is unavoidable and the bike will break. However, wearing a helmet, the consequence *hospital* can be avoided.

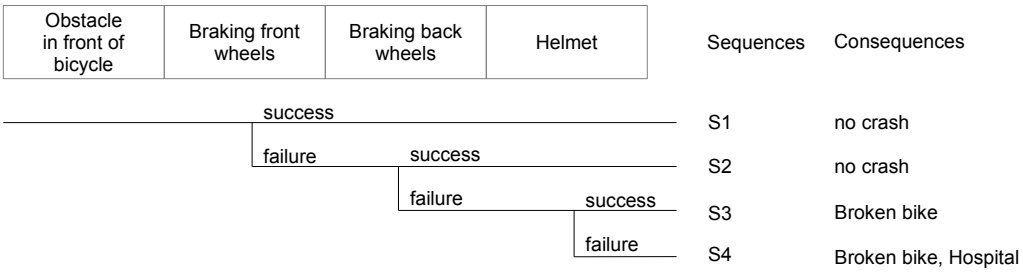


Figure 2.2. Event tree example

2.1.3 Linking Event Trees with Fault Trees

Initiating events and function events of event trees can be linked to fault trees in order to model occurrence characteristics. Generally, there are two approaches: The first approach is the "fault tree linking" approach (FTL), the second the "event tree linking" approach (ETL). In FTL, fault trees "behind" function events may share same basic events, i.e. function events may depend on each other. In ETL, the opposite is true, i.e.

An exhaustive comparison between FTL and ETL is given in [6].

Practically, FTL leads to small event trees connected to large fault trees, whereas in ETL large event trees and small fault trees are created. Additionally, in ETL event trees are often linked to other event trees (though also in FTL event trees can be linked to each other).

PSA models at EDF are based on the FTL approach. Nevertheless, some event trees are quite large in size.

In Figure 2.3 the FTL method is illustrated: All function events are linked to fault trees. A function event occurs (fails) in case the associated fault trees fail (and inverse). The example shows two of the linked fault trees, which are possibly dependent (same events may occur in both fault trees).

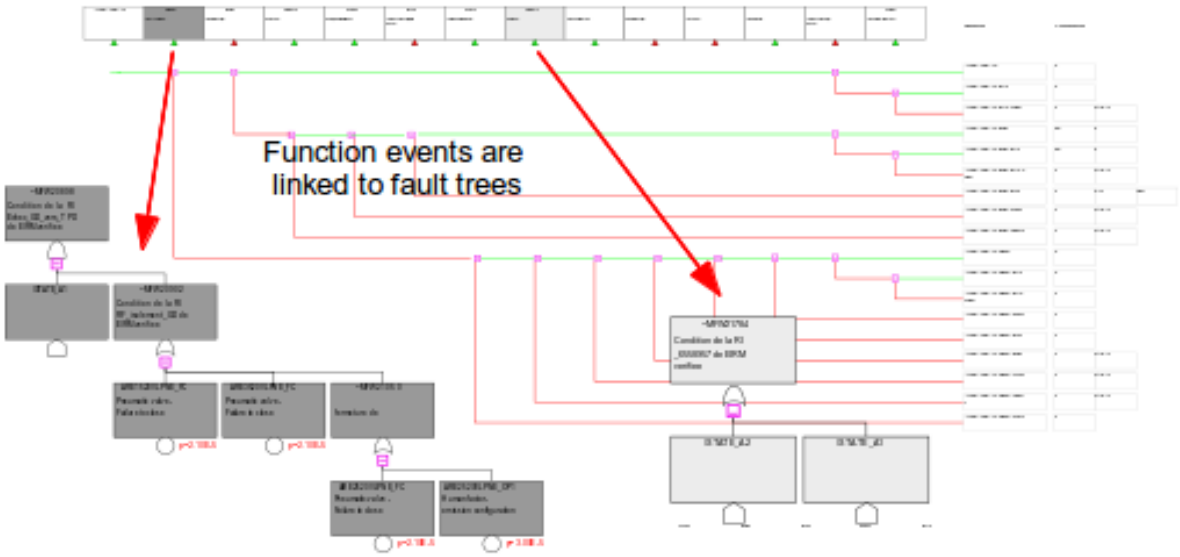


Figure 2.3. Principle of FTL: Function events are linked to fault trees.

2.2 Probabilistic Risk Assessment

Probabilistic risk assessment (PRA), also named *probabilistic safety analysis* (PSA) targets to investigate risk associated with a complex engineered technological entity such as an airliner or a nuclear power plants. The goal of PSA is to determine system failures that may occur due to design, during operation, test or maintenance phases.

A vast spectrum of PSA Literature exists, for example [12, 13, 14, 15, 16, 17].

In the case of nuclear power plants, risk quantification serves as diagnosis tools and justification base for safety authorities. The French safety authority is the ASN (Authorité Sureté Nucléaire).

Risk analysis answers primarily the following questions [12]:

- What can happen?
- What likely is it to happen?
- Given that it occurs, what are the consequences?

Plenty of methods exist to answer these questions. One method is the analysis of fault and event trees, abbreviated as FTA (fault tree analysis) and ETA (event tree analysis).

However, FTA / ETA should not be considered to be the best or the only methods for safety analysis. According to [18], modeling techniques for system / safety modeling should generally be evaluated against the capabilities to:

- Detect unavailability of systems
- Be used for a variety of different situations
- Detect all failures (completeness)
- Understand, communicate and use results
- Reveal principle risks and how to avoid them

In [18, p. 57], the NRC (the U.S. Nuclear Regulatory Commission) presents an overview of other methods (for nuclear PSA):

Table 2.3. Overview over famous concepts in safety engineering for nuclear power plants.

Method	Applicability	Characteristics
Phased-mission	Evaluation of components, systems, or functions undergoing phased mission	Qualitative, quantitative, time-dependent, non-repairable components only; assumes instantaneous transition
Markov analysis	Model and evaluation of components or systems	Quantitative, time-dependent, multiphased inductive; complexity increases rapidly; practical only for simple systems
GO	Evaluation of components, systems, or functions	Quantitative, time-dependent; modeling process complex; success oriented; has potential for modeling complete nuclear plant
FMEA	Identification of hazardous or dependent components or systems	Qualitative, inductive; considers only one failure at a time; simple to apply; provides orderly examination
MORT	Identification of hazards for improving safety	Qualitative; also used for accident investigation
Reliability block diagram	Model and evaluation of components or systems	Quantitative
Signal flow	Model and evaluation of components or systems	Quantitative; assumes constant failure and repair rates

2.2.1 Quantitative Risk Assessment

Quantitative risk assessment is the procedure to obtain measurable results about risk, which are expressed by numeric values. It plays an important role not only for safety engineers but also for safety authorities to justify required safety levels.

Typical results of quantitative risk assessment are:

- Hazard Probabilities: Probability that a hazard occurs at least once in a certain time frame (typically the *mission time*).
- Hazard Frequencies: Frequency, how many times a hazard occurs during a fixed time frame (typically per year).

Quantitative risk analysis of PSA models (fault and event trees) is done per hazard. A hazard is represented either by a

- Fault tree
- Sequence
- Consequence

For quantifying a hazard, the following steps are performed:

1 ***Generation of the so-called "master fault tree"*** : The master fault tree is one single fault tree, which represents the hazard (sequences and consequences of event trees can be converted into a single fault tree). The master fault tree can be very large in size.

2 ***Cutset Determination*** Determination of event combinations that let the master fault tree fail. Those combinations are referred to as "cutsets".

3 ***Quantification*** Different types of quantification can be performed upon cutsets:

- Occurrence characteristics: The occurrence likelihood or frequency of each cutset and the hazard.
- Importance analysis: Feedback about the relevance of events (of the cutsets). More information about importance analysis can be found in [19, 20].
- Sensitivity analysis: Feedback about uncertainty of determined likelihoods and frequencies. PSA

quantification is based on assumptions and the dimension of the failure can be estimated.

In [21] the principle aspects of fault tree quantification are described.

2.2.2 Qualitative Risk Assessment

Qualitative risk assessment is a kind of risk assessment, that focuses more on qualitative aspects, whereas precise numeric values (for example probability values of hazards) are minor. It aims primarily to understand risk. Thus, it reflects the initial idea of risk assessment.

Quantitative risk analysis is related to qualitative risk analysis in the following way: Both, qualitative and quantitative risk assessment help to understand risk, but quantitative risk assessment uses measurable expressions (e.g. hazard probability is 4e-5), whereas qualitative uses relational or interpreted expressions (e.g. "*hazard occurs in case events a and b occur*" or "*hazard probability is very likely*").

For example, the determination of minimal cutsets (without quantifying them) can be used to visualize important relations between events. Minimal cutsets can therefore be regarded as a qualitative analysis result.

By assigning probability ranges to expressions such as "likely", "possibly", "unlikely" etc., a quantitative information can be converted into a qualitative information.

The benefit of qualitative information is that they comprise an evaluation / interpretation of risk results (whereas numeric values are "just" numbers which are not yet interpreted).

Throughout this thesis, the notion of qualitative risk assessment is used in a more general way: It is understood as the application of any method or software tool that helps to develop a "*feeling*" for the actual risk, without basing on numerical values.

2.2.3 Risk Visualization

One famous way to visualize the risk of hazards is to use a so-called "risk consequence matrix". The risk consequence matrix is a two dimensional matrix where one dimension represents the likelihood and the other the consequence severity of hazards (see Figure 2.4).

Once an event has been analysed, it can be assigned to a matrix field according to its likelihood and severity. In the example, the event "Loss of Power" has been rated to be "Unlikely", but its consequence severity is rated "Very High".

The actual risk is understood as the multiplication of the two dimensions (risk = likelihood $*$ severity). The event "Loss of Power" has been identified to constitute a "High Risk".

The risk consequence matrix remains an important medium to present complicated analysis results in an understandable form. It is often used to present results to management.

2.2.4 Risk Assessment Tools

Since computer science found its way to support risk assessment, many PSA tools have been developed. For nuclear power plants, however, only some tools have become popular. In the sequel, a short overview about those is given (without evaluation).

Many European nuclear facilities use "RiskSpectrum PSA" [22] for risk assessment, a software developed by Scandpower. For the thesis, this software plays a central role as "real" PSA models of EDF have been

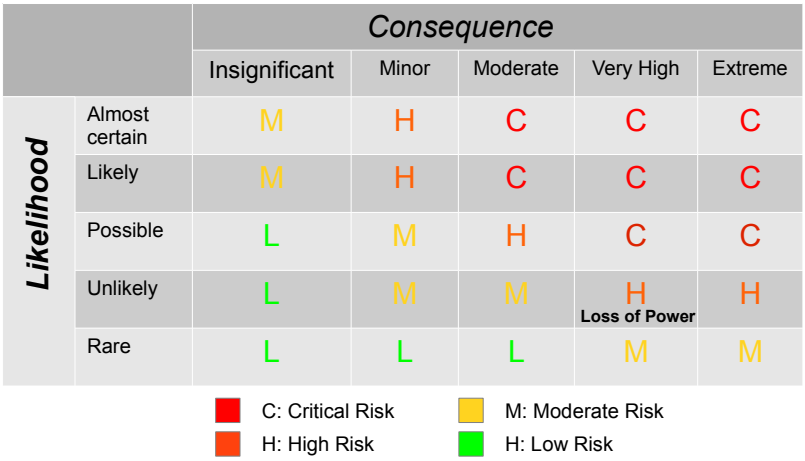


Figure 2.4. Risk Consequence Matrix: Hazards are located within the matrix due to their likelihood and consequence severity.

exported from this software to test approaches developed in this thesis.

For U.S. nuclear plants, CAFTA from EPRI [23] is used for the purpose of risk assessment and risk monitoring.

Other PSA software to mention (for the nuclear sector) is "Saphire" [24] developed by Idaho National Laboratory for the Nuclear Regulatory Commission (NRC), "RiskMan" [25] (distributed by ABSConsulting) and "FinPSA" [26, 27]. The latter is developed by Radiation and Nuclear Safety Authority of Finland (STUK) and VTT Technical Research Centre of Finland.

2.3 Conclusion

The concepts developed in this thesis are related to the development of PSA models. In this chapter, important basics about PSA models and their assessment have been given. These basics are required to fully understand subsequent chapters.

CHAPTER 3

PSA from the 60's to Nowadays

Probabilistic safety assessment has been used for a while. However, it is important to understand that PSA assessment as required today, differs from its initial usage. To say, the context in which PSA models are developed, changed considerably since the introduction of the PSA methodology.

In this chapter, a brief historical overview of probabilistic risk assessment is given with a particular regard on PSA usage for nuclear plants.

Further readings about PRA history, in particular for the nuclear industry, can be found at [28, 29]. The role of PSA in the NASA space shuttle programs (another interesting domain where PSA contributed significantly to enhance safety related issues) can be explored in [30].

Section 3.1 presents the initiation (the birth) of probabilistic safety assessment.

Section 3.2 describes the first applications of PSA methodology for nuclear power plants.

Section 3.3 describes the first serious incident in a nuclear power plant and its impact on PSA methodology.

Section 3.4 points out how enhancements of computer science improved and at the same time complicated PSA assessment.

3.1 Initial Fault- and Event Tree Assessment

Fault tree analysis (FTA) is a rather "old" concept. It has been invented at Bell Laboratories in 1962 by H. A. Watson to identify potential failures in the Minuteman ICBM (Intercontinental Ballistic Missile) system. The Minuteman system was a project at Boeing and part of the U.S. nuclear defense program.

The application of fault tree analysis at Boeing proved successfully that accidents can be identified and avoided before they happen. Boeing identified the goal of risk assessment as to "identify, evaluate, and eliminate or control potential hazards as early in the system life-cycle as possible" [31].

The initial fault tree approach defined the most important gate types that are still in use today. Fault tree quantification was mainly done by simulation techniques[1] due to lack of computational power at this time to evaluate fault trees exhaustively. Today, mainly minimal cutset analysis [32] or BDD [33] techniques are used to obtain more precise (with respect to approximations) and reproducible (deterministic) results.

In 1966, due to its success in the Minuteman program and its rather intuitive and effective usage, the method of fault tree analysis was chosen to analyse potential hazards in civil aviation at Boeing.

In 1970, the U.S. Federal Aviation Administration (FAA) constrained failure probability criteria for aircraft systems and equipment. The satisfaction of the new requirements could be proofed with FTA methods. In the consequence FTA gained importance beyond Boeing.

3.2 Wash Report 1975

The Wash Report, also known under the title "The Reactor Safety Study", was a safety report published in 1975 by the Nuclear Regulatory Commission (NRC) [34]. The report informed about methodologies used for risk assessment of U.S. nuclear power plants. Guided by a committee of specialists under Professor Norman Rasmussen, it contained concrete results (i.e. likelihoods) of risk quantification, obtained with FTA and ETA methods.

The report can be seen as a milestone as it indicates the beginning of PSA for nuclear power plants. Further, it is assumed that the concept of event tree analysis (ETA) has been invented within the scope of the Wash Report (though no real proof can be given). ETA provided an efficient method to present large fault trees in a hierarchical and compact manner. The larger fault trees got, the more ETA gained success to keep PSA models well-arranged.

The report was the first try to perform risk assessment for nuclear power plants fully by probabilistic methods. However, the report was heavily criticized for underestimating the actual risk of power plants, especially for not taking into account the matter of uncertainties.

In 1978, a further review was published by the NRC to investigate the criticized issues of the Wash Report. The report was lead by Professor Harold Lewis and became known as the "Lewis Report" [35]. The report confirmed several critical points about the Wash Report and contained proposals to improve risk assessment. The NRC accepted the critics and withdrew the Wash Report in 1979.

3.3 Three Miles Accident

A severe U.S. nuclear power plant accident in 1979, the *Three Mile Island* (TMI) accident, occurred in Dauphin County, Pennsylvania. In the consequence of technical problems and wrong decisions due to some misunderstandings of the actual plant state, a partial core meltdown happened in one of the two Three Mile Island pressurized water reactors [36].

In the aftermath of the accident, radioactive material was released intentionally (to mitigate the coremelt accident) in form of waste water. The release has been instructed by the NRC.

The TMI has been rated as an *Accident With Wider Consequences* within the international nuclear event scale. It marks the most severe U.S. nuclear power accident in history (stand feb. 2014).

Retrospectively, the TMI had also a positive effect concerning the improvement of PSA methodology: It

[1] A fault tree can be quantified by randomly simulating errors according to event probabilities.

led to serious FTA research and to broad discussions. The Wash Report was credited with identifying the small loss-of-coolant accidents as the major threat to safety [37, 38]. A particular interest was to investigate how to apply FTA for nuclear power plants. Important outcomes were the *NRC Fault Tree Handbook NUREG–0492* of 1981 [7] and the *PRA procedures guide* of 1983 [18]. Those documents created the base for future risk assessment.

3.4 Enhancements in Computer Science

The first time, modern computer software with a graphical interface has been used to support PSA assessment, was in 1987. With the first version of *Saphire*, fault trees could be graphical drawn, edited and analyzed. In the following years, more efficient PSA software got developed, what impacted the characteristics of PSA model engineering.

Enhancements in computer science are the main reason, why PSA models could be extended in the recent decades. An increase of computational power permitted to quantify more complex scenarios. Progresses at software engineering provided helpful functionality to create and modify models more efficiently. Concurrent methods, where a group of engineers work together on one risk study, influenced the way of PSA engineering.

The enhancement of PSA models has led to an increase of complexity of PSA models and tools.

3.5 Conclusion

In this chapter, the history of PSA assessment has been summarized. Though the main methods of PSA did rarely change, the context in which PSA models are engineered has changed considerably.

CHAPTER 4

Present and Future Challenges

PSA models and software tools are subject to a changing context that is sensitive to cultural, political, environmental, scientific and economic evolutions. In the case of nuclear plants, those evolutions must be taken seriously as models are used over long time periods.

In this chapter, current and future challenges of PSA models and their development are described.

Section 4.1 analyses characteristics and challenges of nowadays and future PSA models. A special focus is given to PSA model complexity.

Section 4.2 describes challenges related to the development of PSA models.

Section 4.3 concludes this chapter by emphasizing the need for advanced PSA concepts and tools.

4.1 PSA Models

In this section, challenges concerning PSA models are analysed.

4.1.1 Increasing Model Complexity

The usage of PSA software made it possible to develop more complex models in order to respond to new demands from safety control authorities or technical enhancements. PSA models grew quickly in size over the years. Table 4.1 gives an overview about the evolution of model sizes. The table shows the number of model elements per element type for the PSA models "900", "1300", "N4" and "EPR". Those models have been developed chronologically at EDF in this order since the 90's (though each model is updated periodically). Finally, Figure 4.1 shows the total number of model elements per model (the sum of each table column).

The statistic can be explained as follows: Between the model *1300*, *1450* and *EPR*, the number of model

Table 4.1. Evolution of the number of model components in different PSA models at EDF

Type	900	1300	N4 (1450)	EPR
Basic Events	7671	5374	5754	6808
Basic Event References	12041	8075	8565	15567
CCF Group	450	292	218	563
Consequences	115	210	288	1246
Consequence Reference	11204	12935	23738	38077
Event Trees	773	340	412	1096
Fault Trees	3363	2593	2019	3321
Function Event	305	381	472	603
Function Event Reference	17155	12470	21323	36849
Gates	12569	7741	7071	13914
Gate References	16806	10292	9954	18685
House Events	818	406	399	412
House Event References	6978	3137	2256	4418
Initiating Events	285	340	413	1349
Initiating Event References	773	340	412	1096
Parameters	1737	1080	1845	2045
Parameter References	12246	9017	8686	10899
Rules	1023	714	1125	1024
Rule References	793	394	606	1341
Sequences	11060	8467	16786	27460

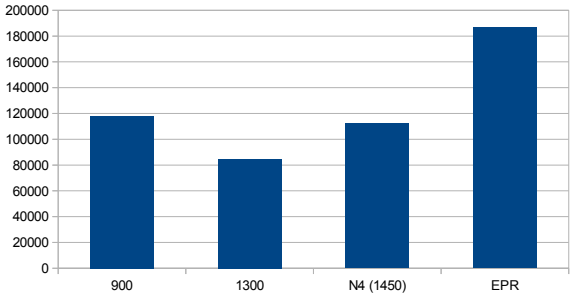


Figure 4.1. Evolution of model sizes: The statistic shows the total number of model components in different PSA models developed at EDF

components have more than doubled, due to technical enhancements of physical systems: The EPR reactor is considered to be far more complex than the *1300*.

However, the *900* contains more components than the *1300* though the *900* reactor is considered less complicated. The reason is, that the *900* contains site-specific adaptions for the power plants of "Fessenheim" and "Bugey"[1]. This fact can also be observed in the number of house events: House events are often used to model site specific behavior and the *900* contains at least twice as house events as other architectures.

The statistic indicates that PSA model sizes are indeed increasing. The situation is supposed to become even worse when PSA models should reflect a wider range of hazards (in particular external hazards such as seism, flooding etc.) or satisfy additional or stricter requirements imposed by safety authorities.

Generally it can be said that, the more components and inter-relations (between components) a model comprises, the more complicated it gets to understand its composition, i.e. the clarity of models gets reduced by increasing model sizes.

But not only the number of model components has increased. Also the size of individual model components

[1] Fessenheim and Bugey refer to two French nuclear power plants of EDF.

has increased. Figure 4.2 shows a large fault tree and Figure 4.3 a large event tree example. Both examples origin from a real PSA model at EDF. The fault tree has been cut into an upper and a lower part to fit into one diagram (the lower part continues the upper part on the right side). It consists of 89 gates, whereas 28 are external gates, which refer to gates of further fault trees.

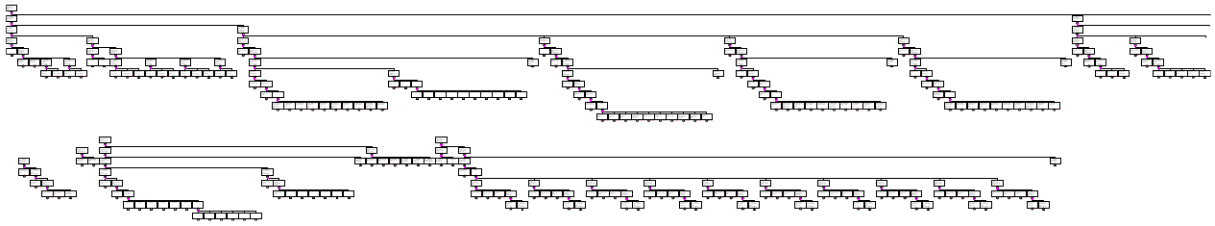


Figure 4.2. Example of a large fault tree: The fault tree contains 89 gates, where 28 are external gates referring to further fault trees

The event tree has been printed vertically to fit on the page: Function events are on the left side, sequences and consequences on the upper side of the page. The tree consists of 375 sequences, every sequence leading to at least one consequence. Some consequences serve as initiator in other event trees.

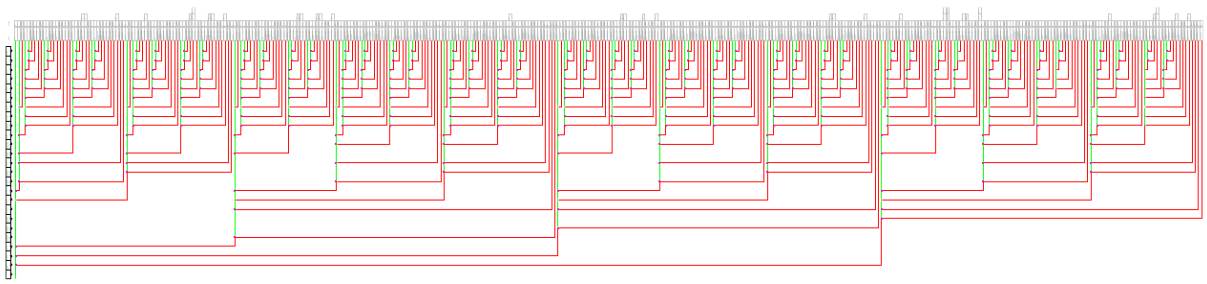


Figure 4.3. Example of a large event tree: The event tree contains 375 sequences

The examples illustrate how individual model components increased in complexity. Complex model components hamper engineers and analysts to reveal risk-related insights of a system or a behavior. The challenge is to find counter methods to reduce model complexity.

4.1.2 Modeling Redundancy

The examples given in Figure 4.2 and Figure 4.3 reveal more than a complexity concern: Both, the fault and event tree contain repeating patterns (substructures of fault and event trees which occur multiple times within the same tree). Those patterns often indicate a kind of modeling redundancy.

Modeling redundancy constitutes a particular problematic as model development can get:

- Time-consuming: Redundant parts have to be identified and modifications have to be applied various times in each redundant part.
- Error-prone: The identification of redundant parts is not obvious. Modifications may not be applied or not equally be applied to all redundant parts.
- Expensive: The financial consequence of spending more time on model development and on identifying and correcting modeling errors is an increasing cost for PSA model development.

Similar to software redundancy (redundancy of source code at software development), modeling redundancy should be avoided whenever possible. Possibilities to cope with redundancy problems are to apply

techniques of abstraction, instantiation and modularization. At software development, those techniques are intensively exploited, especially at component based software engineering [39, 40], where a software is composed by a set of software pieces.

PSA models may provide flag mechanisms[2] to "instantiate" fault trees differently what reduces redundancy problems. However, these possibilities may soon reach their limits (they risk to complicate and "blow up" PSA models).

In particular for event trees, new methods are required to avoid model redundancy. Redundancy in event trees is created when the application of a set of mitigating events (function events) is identically in different sequences. Modifications must be applied carefully to event trees: The risk of creating different mitigations in otherwise redundant event tree parts is considerable.

4.1.3 Low level Modeling Language

Unfortunately, the language of PSA models (fault and event trees) it not really adequate to cope sufficiently with redundancy problems.

The reason is that PSA models are generally designed as a low level modeling language. They may be adequate to interface with quantification engines, but future PSA development should consider higher level modeling languages such as AltaRica [41] or Figaro [42] as primary modeling language used by model engineers. Those languages are able to model much more succinctly and intuitively and they can generate lower level languages such as current PSA models (see Figure 4.4).

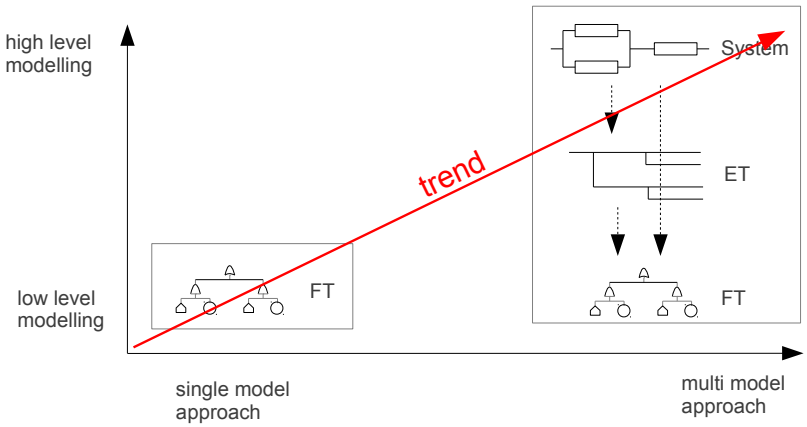


Figure 4.4. Trend towards higher level modeling languages: In order to model more compact and intuitive, higher modeling languages are required. Lower model languages such as fault trees are suggested to be automatically generated.

In Figure 4.4, another trend is indicated: The trend towards multi model approaches where risk engineering is not based on one individual model but more on a set of models.

The trend towards higher level languages can also be observed at software engineering. Whereas initially programs have been developed in *assembly languages*, higher level languages got soon introduced to develop more complex programs with acceptable efforts: Languages such as *C*, *C++* etc. and later *Java* replaced assembly languages in most cases. At the same time, it is possible to generate code from those

[2] Flag mechanisms are PSA software dependent. Riskspectrum for example provides the configuration of "boundary condition sets" and "house events" to **configure** fault trees.

in lower level languages (such as assembly languages) for execution reasons. Later on, even higher level methods such as *component based software engineering* [40] and recently *model based software engineering* methods provide new possibilities of developing and managing complex software.

4.2 PSA Model Development

In this section, challenges concerning PSA developed are analysed.

4.2.1 Two-Dimensional Model Development

At EDF, typically one generic PSA model is designed per reactor type. For example, one PSA model *900*[3] serves as generic model for all "900" nuclear plants.

However, so-called "plant-specific" constraints are required to reflect specific circumstances for particular power plants. They are often realized by the means of house events (or other kind of flag mechanisms), which "adapt" a PSA model to specific conditions.

Thus, the evolution of PSA models becomes two dimensional: Figure 7.18 illustrates the two dimensional model evolution: So-called "*variants*" indicate the diversity of models (for example to model plant specific conditions) and so-called "*versions*" indicate temporal evolutions.

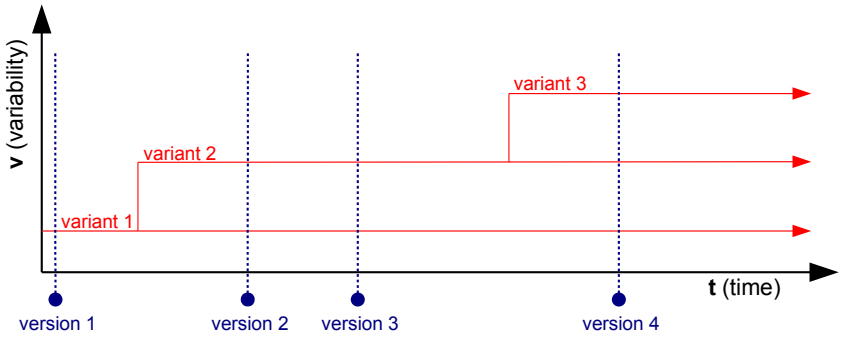


Figure 4.5. Two dimensional model development: Variants and versions express the diversity and temporal evolution of models

The risk of modeling plant specific evolutions is to blow up model sizes and to reduce readability aspects. Care must be taken to choose the right methods to keep models maintainable.

In future, plant specific model evolutions are supposed to become even more important: The reason is the incorporation of external hazards (seism, flooding, fire) - which are supposed to be rather plant specific.

The challenge is to find new concepts to *manage* the two dimensional model evolutions, i.e. a strategy to manage model versions and variants is required. For this purpose, the terms *version management* and *variant management* are getting introduced later on in this thesis.

4.2.2 Concurrent Model Development

Whereas initially PSA models got developed by one individual model engineer, nowadays PSA models are developed by a group of engineers, what is referred to as "concurrent model development".

[3]The model 900 represents power plants of the 900 MWe class.

Concurrent model development permits to assign experts (model engineers with special knowledge for a certain model part) on different model parts. Each expert is responsible to develop and maintain his part. Model development is said to be performed in parallel.

The current challenge of concurrent model development is to find a strategy to synchronize model actions between different engineers. Each engineer works typically on his private model and those models have to be "brought together" into one model, which contains the sum of modifications of all engineers.

More precisely, a method to "fuse" models (to combine them) is required. Currently, model fusion is not yet automated. It is obtained by manually replicating modifications of model engineers to the same model what is inefficient and error-prone. A more efficient method is required to support engineers in synchronizing their modifications.

4.2.3 Long Life Cycles

Typically, PSA models have long life cycles, that outreach the operational period of nuclear power plants. A life cycle describes the different phases of a product or a service over its life time (the period since its creation to the point in time it is no longer needed).

PSA Models are created in the design phase of plants to reveal design errors and to justify safety standards.

During the operational phase of nuclear plants, PSA models serve as diagnosis tool for system failures and to justify plant modifications to safety authorities.

The final phase is the plant deconstruction phase. During this phase, PSA models serve as diagnosis and decision making tool.

The life cycle of a PSA model ends once the plant deconstruction is completed. In total, life times of models may outreach periods of 70 years.

Long life cycles of PSA models pose particular constraints on PSA concepts and software. Some issues to consider are:

- Requirements on PSA assessment are evolving (see Section 4.2.4).
- Knowledge is to transfer between risk engineers.
- New software tools may have to be incorporated in existing tool-chains (see Section 4.2.5).
- Current PSA tools may no longer be used one day.
- Technical enhancements of physical systems may have to be incorporated in models.
- New outcomes of research institutes may impact the current PSA methodology.

4.2.4 Evolving Safety Requirements

Since the PSA methodology has become an accepted approach to assess the risk of nuclear power plants, safety authorities developed requirements for its application. A requirement example is the definition of an upper limit of a "core damage frequency" (CDF)[4].

With the years, the requirements were refined (generally they became stricter). Several reasons can be made out (though not exhaustive):

- Technical enhancements: New reactor types or plant installations got developed, typically more sophisticated and complicated than ancient ones. As PSA studies reflect technical compositions, safety requirements got adapted accordingly.

[4] A core damage in a nuclear power plant is one of the most serious accidents in a nuclear plant.

- Progress in computer science: Computational power and enhancements in software engineering made it possible to create and quantify larger and more complex models. This permitted to tighten requirements of model engineering and quantification.
- Accidents: Accidents such as the “Three Miles Accident”, “Chernobyl” and recently “Fukushima” lead generally to stricter requirements.

The challenge is to find a way of modeling and tooling that facilitates the incorporation of new, typically stricter requirements.

4.2.5 Evolving Tool Chains

Initially, in the 90’s, when first PSA software tools found industrial usage, companies considered in principle one single PSA tool to perform any PSA related task (PSA modeling, quantification, visualization, documentation etc).

Now, more than 20 years later, software engineering and tooling philosophy changed considerably. Generally, software is much more complex. It is often not sufficient to base on one single software tool, but on a set of tools: So-called “tool chains” are preferable, where various tools are considered to interface each other in order to complete a common goal. Tool chains can be quite flexible to adapt to new software requirements by enhancing or replacing individual tools.

However, tool chains require to establish interfaces between software tools. Software vendors may be politically concerned to interface with other tools (creating interfaces means also to support software from other, possible competing software vendors).

4.3 Conclusion

In this chapter, important challenges for nowadays and future PSA assessment have been stated concerning both PSA models and PSA model development.

PSA model engineering for nuclear power plants is particularly challenging as models can get quite complex and difficult to enhance throughout their long life times of possibly more than 70 years.

Since their development, current PSA models have not even endured half of their life time. New PSA methods and software are needed to prepare for the challenges of future PSA studies.

CHAPTER 5

Preliminary Work

Safety engineers start to become aware of PSA challenges. Initiatives such as the *Open PSA Initiative* [43] and the *Phoenix Project* (a project at EPRI [23]) claim to invent the next generation of risk assessment concerning tools and methods. But also industry, such as EDF R&D in France, invests in improving their PSA engineering methods and tools.

In this chapter, preliminary work with regard to this thesis is presented. One is the idea of a *modular PSA* (developed at EDF R&D), the other is the *Open PSA Model Exchange Format* (an outcome of the Open PSA Initiative).

Section 5.1 introduces to the original modular PSA idea of Mr. Hibti.

Section 5.2 presents the so-called *Open PSA Model Exchange Format*.

Section 5.3 concludes this chapter.

5.1 Previous Thoughts About a Modular PSA

This section summarizes the work realized by M. Hibti about a modular PSA.

5.1.1 Context

The idea of a modular PSA was first published in 2008 [5]. At this time, new requirements were analysed to incorporate external hazards in PSA models at EDF.

The incorporation of external hazards depends typically on the geographical location of power plants. The difficulty is that PSA models at EDF do not exist for each specific plant, but for different plant architectures (for example a PSA model “900” represents all plants of 900 MW PWR).

Current PSA software often provides kind of flag mechanisms to activate / deactivate model parts specifically for plants (or other criteria). For example, in RiskSpectrum [22], house events and BCC sets [1] serve for this purpose. However, those flag configurations risk to "blow up" model sizes at the expense of model clearness and maintainability aspects (enhancing models any further may become difficult).

New ideas were investigated to model external hazards with acceptable efforts and without blowing up model sizes. At this time, model sizes have already been rated "large".

The goal of a modular PSA was to find a way to share common model data between different PSA models and also to improve clarity of models. This should be reached by the principle of modularization: By treating a model by several individual model pieces (and not as a whole) was considered to improve modeling characteristics.

5.1.2 Concept

The original approach intends to develop a generic data kernel, which is surrounded by so-called "*modules*". Modules contain specific PSA model components (for example plant or hazard specific components). The kernel provides commonly used model objects and needs to be instantiated.

For example, in order to model external hazards, one could consider to have a *Seismic*, *Flooding* and a *Fire* module.

Each module specifies

1. PSA components which are considered as specific in some context
2. A set of instructions to instantiate the generic kernel

Figure 5.1 illustrates the modular PSA idea of Hibti by the example of a level one PSA [2]:

- The generic kernel consists of internal events[3].
- Modules represent external events[4] and other model objects which "reuse" the generic kernel such as a level two specification.

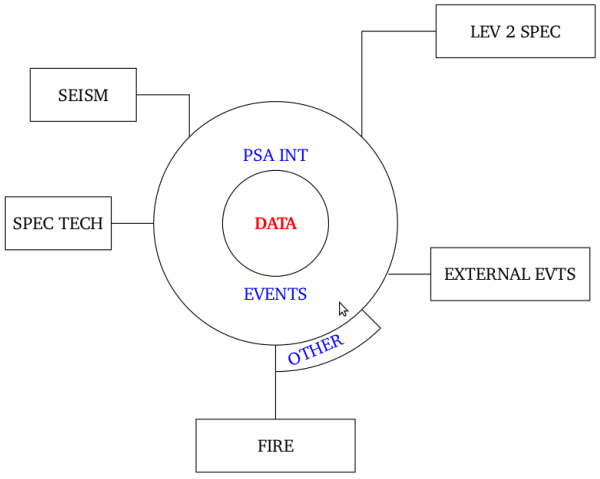


Figure 5.1. Level one PSA in a modular PSA [5]

[1] A BCC set is a conditional construct in RiskSpectrum called "Boundary Condition Set".
[2] A level one PSA identifies the potential accident sequences that may lead to a core damage[18].
[3] Internal events are events which occur "inside" a plant, e.g. the loss of a diesel generator.
[4] External events are events which occur "outside" a plant but which may affect the plant, e.g. flood, seism, ...

5.1.3 Hybrid PSA

One of the limits of the classical approach with fault- and event trees is that it is difficult to consider dynamic phenomena (for example repair actions).

The original modular PSA idea mentioned therefore the idea of a so-called "Hybrid PSA". In a Hybrid PSA, a PSA model is not merely based on fault- and event trees, but also on other model types (for example Markov models [44], dynamic fault trees [45], BDMPs [46] etc).

A Hybrid PSA can possibly be realized in a modular PSA by expressing dynamic aspects by modules.

5.2 Open PSA Model Exchange Format

The Open-PSA Model Exchange Format (OPSAMEF) is a XML format [47] that defines an open and clear representation of PSA models [4].

It was first published in 2007 as an outcome of the *Open-PSA Initiative*, an informal group of PSA analysts and researchers who aim to contribute to the development of the next generation of PSA software and methods.

The OPSAMEF format has been evaluated in [48].

For this thesis, the role of the format was twofold:

- It improved successfully clarity of PSA models, what motivated to continue further investigations such as the kick-off of this thesis.
- It enabled to convert PSA models at EDF into an open, readable format to test new concepts by the means of "real" industrial models.

5.2.1 Objectives

The main objectives of the Open-PSA Model Exchange Format are the following:

- *Clarity of models:* Provision of a clear structure and semantic for PSA models.
- *Portability of the models between different software:* The Model Exchange Format allows the sharing of PSA models between different PSA software tools (see Figure 5.2).

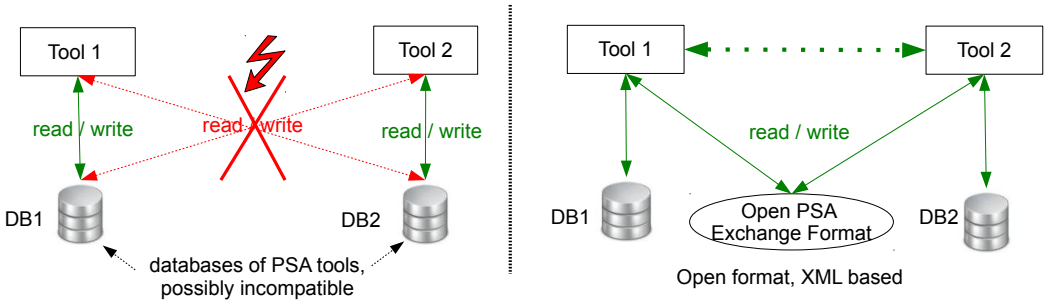


Figure 5.2. The Open PSA Model Exchange Format: PSA models can be exchanged between PSA software tools

- *Quality assurance of calculations:* Currently, it is difficult to verify quantification results. With the Model Exchange Format, models can be quantified by different calculation engines. This eases not

only to verify the correctness of quantification engines [5], but also to detect modeling errors: Using different engines yields a wider spectrum of feedback concerning modeling errors (quantification engines may give important feedback about modeling problems).
In 2012, the Open PSA Initiative announced the first release of the software "XFTA", a free fault tree quantification engine developed by A. Rauzy [19]. The interface of XFTA is based on text-files, which eases the integration of XFTA in existing PSA software tools.

5.2.2 A Four-Plus-One Layers Architecture

In order to provide a clear structure, the Model Exchange Format groups model elements into five different layers, where four layers are organized hierarchically and a fifth constitutes an independent extra-layer for reporting (see Figure 5.3).

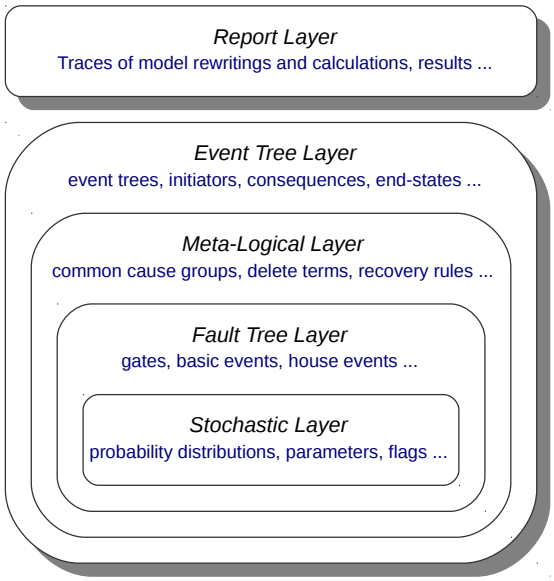


Figure 5.3. Architecture of the Model Exchange Format

1. *Stochastic Layer:* Stochastic model aspects: probability distributions, parameters etc.
2. *Fault Tree Layer:* Fault tree components: gates, basic events, house events etc.
3. *Meta-Logical Layer:* Constructs to "flavor" fault trees: common-cause groups, delete terms, recovery rules etc.
4. *Event Tree Layer:* Event tree components: initiating events, sequences, consequences etc.
5. *Report Layer:* Constructs to store and protocol quantification results:
   - quantification result: list of minimal cutsets, probabilities, frequencies, importance and sensitive analysis etc.
   - quantification parameter: quantified object, cutoff values, optimization parameter
   - environment: PSA model version, type of used quantification engine etc.

5.3 Conclusion

In this chapter, two preexisting works have been presented.

[5] A quantification engine refers to the PSA software that quantifies a PSA model.

The original idea of a modular PSA stated important characteristics and possible advantages of a modular treatment in general. This work served as central theme.

The Open PSA Model Exchange Format provided the possibility to test the approaches developed in the scope of the thesis with real PSA models. The thesis has been launched in same context as the format specification: to improve clearness and readability of PSA models.

PART II

SAFETY ENGINEERING IN A MODULAR PSA

CHAPTER 6

Principle of a modular PSA

It has been mentioned, that PSA models (in particular those for nuclear power plants) grew in size and complexity over the recent years (see Section 4.1.1). New PSA concepts and tools are needed to ensure efficient PSA engineering without losing quality of models.

In this chapter, the principle of a modular PSA is explained. A modular PSA treats a model by smaller pieces (called modules) with the aim to improve clarity of models, to reduce model complexity and to provide new modeling capabilities for managing large models.

The principal concepts of a modular PSA are a modularization technique for obtaining smaller model pieces and an instantiation technique for flavoring (adapting) models to different contexts. A context represents the scope, objective and application in which a PSA model is interpreted.

With the instantiation technique, model engineering in a modular PSA can be divided into three disciplines: domain engineering (the definition of a type of models), the actual model development (the creation of a model) and model instantiation (the interpretation of a model specific to a context).

The approach extends preliminary work introduced in Section 5.1.

The chapter is organized as follows:

Section 6.1 introduces the concept of a modular PSA.

Section 6.2 provides a technical specification for domain engineering, model development and model instantiation in a modular PSA.

Section 6.3 clarifies the differences and extensions to the ideas about a modular PSA of M. Hibti (see Section 5.1).

Section 6.4 presents formats to encode the content of a PSA model and to specify model adaptions.

Section 6.5 illustrates the conception of PSA models in a modular PSA.

Section 6.6 concludes the chapter.

6.1 Concept of a Modular PSA

A modular PSA is a generic concept to improve modeling capabilities and clarity of models. It is not limited to the domain of PSA models.

In this section, the composition and instantiation of models in a modular PSA is presented.

6.1.1 Modularization

A model in a modular PSA is split into a set of so-called "*modules*" (see Figure 6.1). A module is an individual model component such as a fault tree, an event tree, a basic event etc.

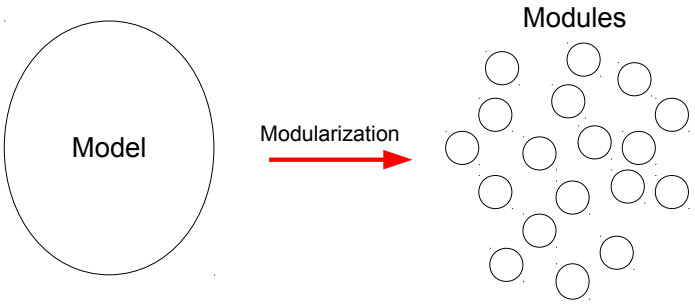


Figure 6.1. Principle of a modularization

6.1.2 Generic Models

A modular PSA knows two forms of models: Generic models and instantiated models. Generic models are kind of preliminary models that need to get "instantiated" to obtain full models (instantiated models).

The composition of a generic model is illustrated in Figure 6.2. A modular PSA contains a set of (generic) models. Each model provides its modules. A module is a component that is not contained in other components.

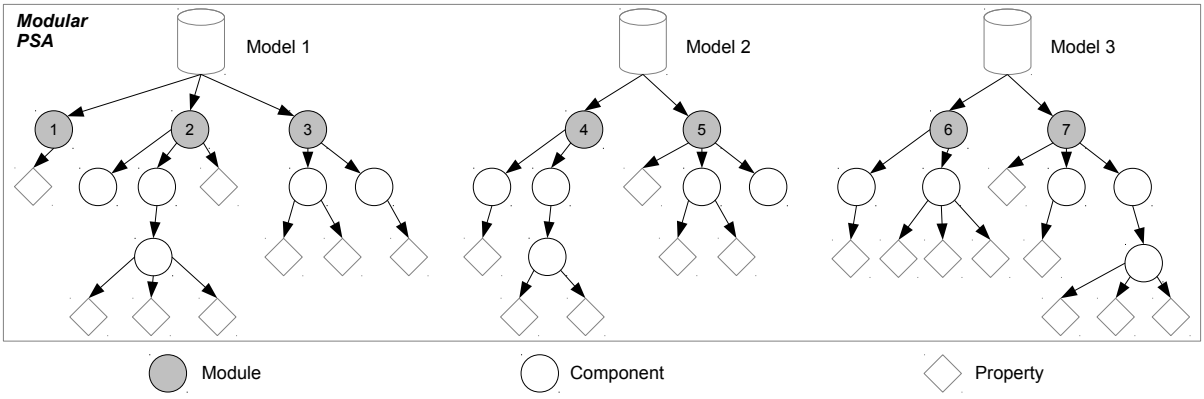


Figure 6.2. Model Composition in a modular PSA

Components (in particular modules) can contain further components (so-called "subcomponents") and properties. Technically a component serves as container for a set of subcomponents and properties.

Modules are considered as autonomous components in the sense that their existence does not depend on the existence of other components.

(Non-) module examples in a PSA model:

- Fault trees are modules.
- Gates are not considered as modules. They are contained in fault trees and therefore they are not independently defined.
- Basic events are modules. Though they may be used in fault trees, they are not contained in them (they are only referenced from them).
- Event trees are modules.
- Sequences are contained in event trees. They are not considered as modules.

Properties store values. In the simplest form a value is an elementary value (a constant) such as a name or a number (for example the name of a fault tree or the probability of a basic event). In general, property values are so-called "terms" which can depend on other components. Technically, terms are formulas over references and constants.

References are like "pointers" towards components. In an instantiated model, a reference is associated with a concrete component it refers to. But in a generic model, a reference is solely "indicating" the referenced component. The indication consists to precise the name and the kind (for example fault tree or event tree) of the referenced component.

Models provide components and properties in a "tree" structure: Components are the nodes and properties the leaves. Consequently, properties and components cannot be contained in two different components.

6.1.3 Instantiated Models

An instantiated model is obtained from a generic model by instantiation. It differs from its generic form in three points:

- References (which occur in property terms) are associated with concrete components. They are said to be "resolved".
- Property values (terms) may have been "adapted" to alternative terms. An adaption is only virtually applied (like an overlay) and does not impact the generic model.
- Components may have been "deactivated". A deactivated component is interpreted like not being present in a model. All contained components and properties are deactivated as well.

Figure 6.3 shows an instantiated model. The red arrows indicate resolved references. The red properties indicate adapted properties.

In an instantiated model, properties develop concrete dependencies to other components. For example a property *input* of a gate (the property declares gate inputs) develops dependencies to gate inputs (gates, basic events and house events) at instantiation.

Module Dependencies

Module dependencies are dependencies between modules in the sense "which module requires which other modules". They are derived from properties and references: If a property contains a reference towards a component, then the module containing (recursively) the property depends on the module containing (recursively) the component. A module can depend on itself.

In Figure 6.3, the following modules are dependent:

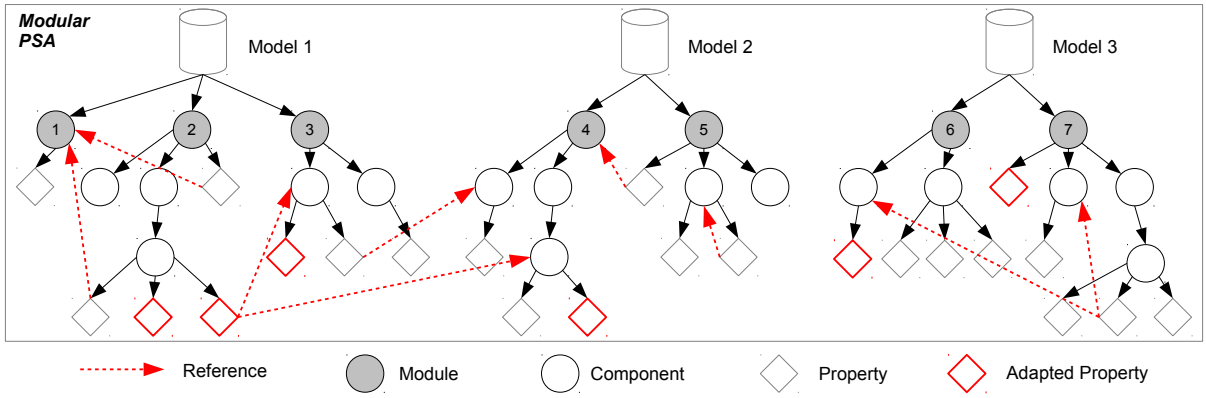


Figure 6.3. Model Composition in a modular PSA

- Module 2 depends on modules 1, 3 and 4.
- Module 3 depends on module 4.
- Module 5 depends on modules 4 and 5.
- Module 7 depends on module 6 and 7.
- Module 7 depends on module 7.

Adapted Modules

Adapted modules are modules where at least one (recursively) contained property has been adapted (at instantiation).

The set of adapted modules is derived from the set of adapted properties. In the example of Figure 6.3, the modules 2,3,4,6 and 7 are adapted. They contain (recursively) minimum one adapted property.

Component Deactivation

A special property called the “enabled” property serves to activate / deactivate components at instantiation. Its value is of elementary type “Boolean”. Figure 6.4 demonstrates the deactivation principle. If the property is set to “False”, then the component containing the property is treated not being present in the model. All its (recursively) contained properties and components are deactivated as well.

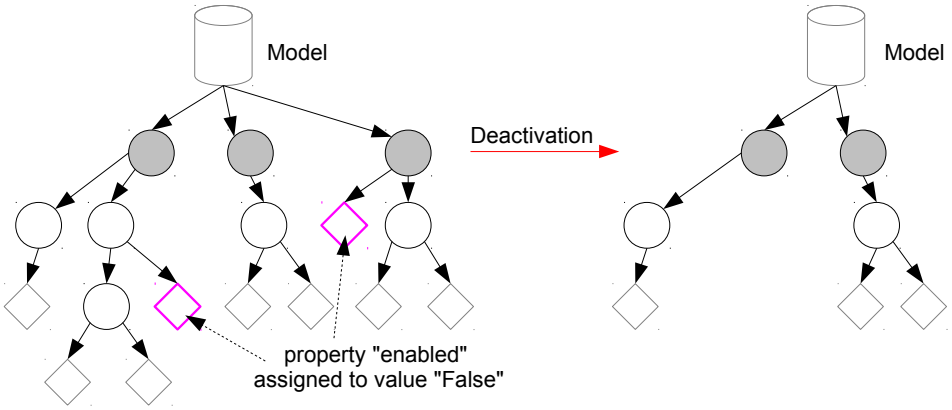


Figure 6.4. Component Deactivation: The “enabled” property can deactivate components

By default, components are always activated. The "enabled" property can be set in the generic model. At instantiation, it can get adapted (as any property values).

6.1.4 Model Instantiation

Model instantiation serves to instantiate a generic model due to a specific context in order to obtain an instantiated model. A context represents a specific situation or circumstance, for example a certain nuclear plant or hazard type.

The concept of model instantiation makes it possible to adapt models to specific contexts. To develop models in dependence to contexts is called *context sensitive modeling*. Context sensitive modeling is a core concept in a modular PSA: By simply altering a context (i.e. without "*touching*" models), one can express different situations. For example, a context can configure to deactivate certain function events in an event tree, to substitute gates or basic events in fault trees, to modify parameters of basic events etc.

Instantiation Process

The instantiation process instantiates a generic model in a context τ. Context τ is called the *instantiation context*.

Figure 6.5 illustrates the instantiation process. The instantiation process can be written as a function $f(M, \tau)$, which maps a model M to a model M' given a context τ. Model M' (the instantiated model) differs from M (the generic form) in the following points:

- Whereas module dependencies in M are solely indicated, in M' they are "resolved" (see Section 6.1.4). In Figure 6.5, resolved dependencies are indicated by black arrows (between modules). Technically, module dependencies are derived from properties and references (see Section 6.1.3).
- Modules in M' may comprise adaptions respective the modules in M. In Figure 6.5 adapted modules are indicated by red circles. Technically, (recursively) contained properties are adapted.
- Components in an instantiated module (in particular a module itself) may be deactivated. Component deactivation can be achieved by property adaption (by adapting property "enabled" to "False"). They are not specially indicated in Figure 6.5.

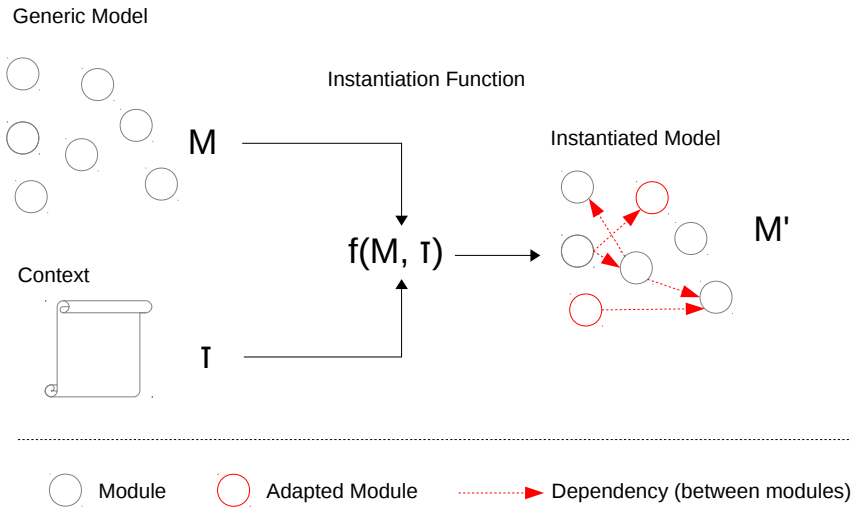


Figure 6.5. Model Instantiation: A model is instantiated in dependence to a context τ.

More generally, a set of models gets instantiated in dependence to a context τ (see Figure 6.6). The instantiation function becomes a function $f(M1, M2, \ldots, Mn, \tau)$ which maps a set of models $M1$, $M2$, $\ldots$, Mn to a set of models $M1'$, $M2'$, $\ldots$, Mn' given a context τ.

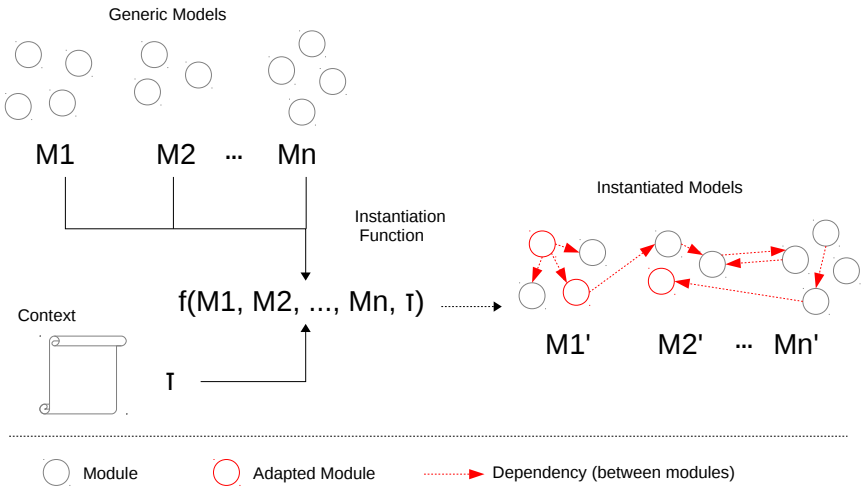


Figure 6.6. Model Instantiation in general: A set of models is instantiated in dependence to a context τ.

Dependency Resolution

Dependency resolution is the process to determine module dependencies in an instantiated model. The step has been illustrated in Section 6.1.3 (though without the notion of contexts).

To determine the dependent modules of a module M involves the following steps:
Step 1: The set of references contained in M is determined. References occur in properties terms.

Step 2: The references are resolved with respect to the instantiation context τ: The result is a set of components (not necessarily modules) that are referenced by the module.

The resolution procedure to resolve a single reference is illustrated in Figure 6.7. A reference R is resolved sequentially in an ordered list of models 1 to n. As soon as it is possible to resolve R in a model (by matching the reference to one of its components) the resolution is finished. If no model can resolve the reference, the reference remains "unresolved". Unresolved references indicate a kind of model inconsistency (of an instantiated model).

Context τ impacts the resolution in two points:

1. The list of models 1 to n (including its order) is derived from τ.
2. Certain components may be adapted / deactivated in τ and in consequence R is possibly resolved differently (this point is indicated in 6.7 by the "impacts" arrow).

Within one model, a reference is always resolved uniquely (or not at all). Uniqueness is achieved by introducing the following two constraints:

- A reference provides information about the name and the type of the referenced component.
- Only components that are unique by name and type can be referenced (so-called "*global elements*").

Note that a reference can principally be resolved in several models. In this case, the first model capable to resolve the reference "wins" (as specified in 6.7).

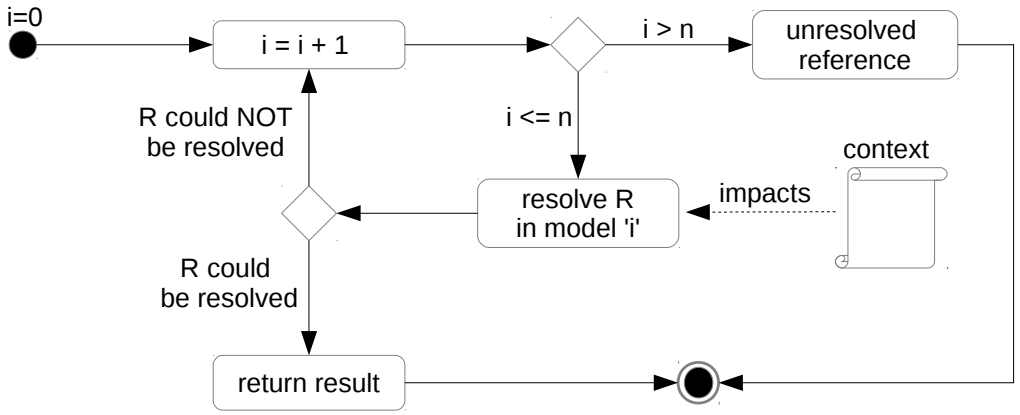


Figure 6.7. Reference Resolution: A reference R is resolved sequentially for an ordered set of models 1 to n. The resolution is finished as soon as the reference can be resolved in a model or if it cannot be resolved at all.

Step 3: Finally, module dependencies are derived from resolved references. Each resolved reference towards a component C indicates a module dependency towards C (if C is a module) or towards the module (recursively) containing C (if C is a subcomponent).

Default Context

If no instantiation context is specified, a *default context* is used to instantiate models. The default context comprises the following characteristics:

- Module dependencies are only resolved in the owning model (the model that defines the dependency). Consequently, in Figure 6.6, $M1', M2'$ and $M3'$ would not develop any dependencies between each other.
- Modules are not adapted. Figure 6.6 would not contain any red circles.

A model instantiated with the default context is called a *standard model* or *default model*.

A model that is instantiated with a context different from the default context is called a *variant*.

6.2 Technical Specification

In a modular PSA, model engineering can be divided into three disciplines:

- Domain Engineering: Specification of underlying model schemes (called “domains”): Each model must conform to a domain.
- Model Composition: Specification of models, i.e. of model objects and their relations.
- Model Instantiation: Instantiation of a set of models in dependence to a specific context.

This section precises the disciplines of model engineering in a modular PSA.

6.2.1 Domain Engineering

A domain specifies composition rules and internal behavior (called a "scheme") for a group of model elements. Each model is associated with a domain. For example, PSA models are associated with the "PSA Domain", which defines the underlying scheme for PSA models.

Figure 6.8 shows the composition of domains in form of an UML class diagram. A domain defines component types, operators and property types. Further it declares a model type that is representative for the domain.

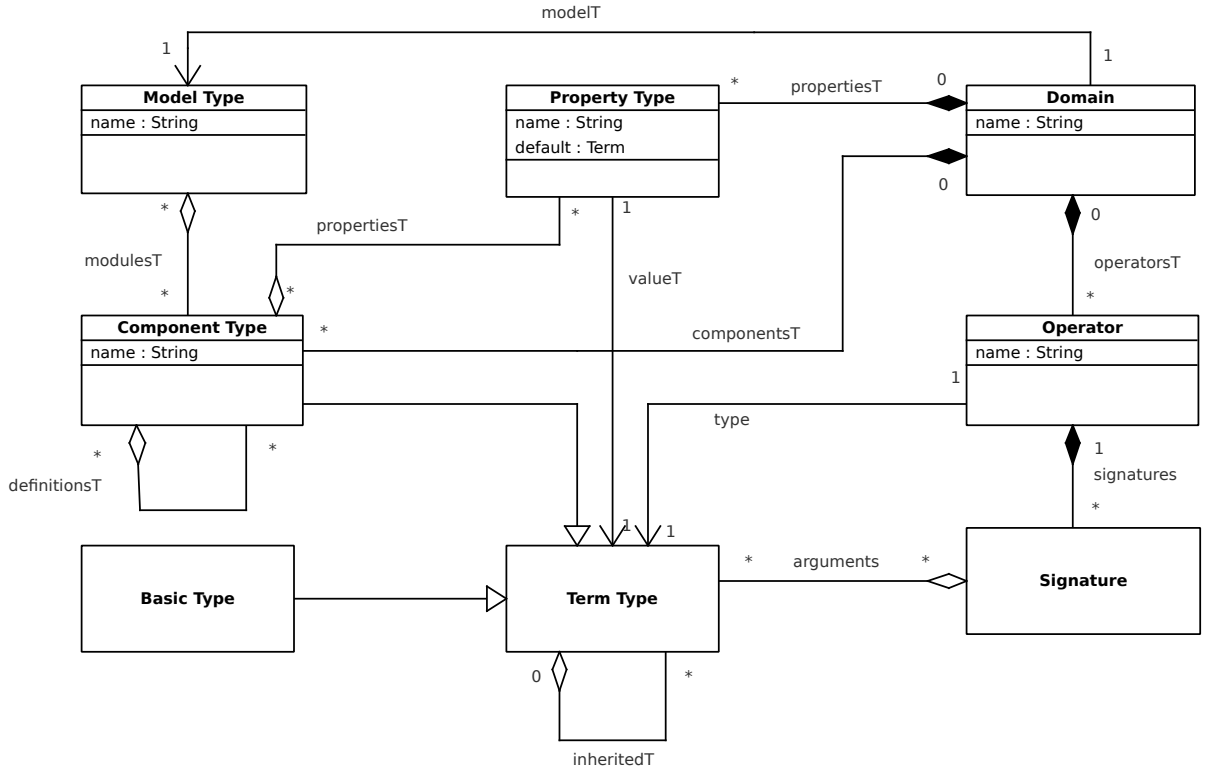


Figure 6.8. Domain Engineering in a modular PSA

The model type declares a set of component types that can be contained in a model (as modules). The component types are typically defined in the same domain as the module type (but not necessarily).

Each component type declares a set of property types indicating the kind of properties that can be stored in a component. The property types are typically defined in the same domain as the component type (but not necessarily).

Each property type declares a value type (a term type). The value type indicates the kind of values a property can take.

Each term type can be a component type or a basic type (an elementary type such as strings or integer values).

Operators are used to describe more complex terms. An operator has a type (kind of the "return type" of an operation) and one or several signatures. Each signature describes how to instantiate the operator by a set of arguments.

Let *Types* be the set of all types (model types, component types, property types, term types and opera-

tors).

The remainder of this section describes domain engineering in a formal manner. In the following let *xyTypes* be the set of all "xy" types (for example *ModelTypes* is the set of all model types).

Domains

A *domain* is a tuple $\langle name, modelT, componentsT, operatorsT, propertiesT \rangle$, whereas

- $name \in String$ is the name of the domain.
- $modelT \in ModelTypes$ is the model type of the domain. Each domain is associated with one characteristic model type. For example the "PSA Domain" is associated with the model type "PSA Model Type".
- $componentsT$ is a set of $ComponentTypes$ that is defined by the domain.
- $operatorsT$ is a set of $OperatorTypes$ that is defined by the domain.
- $propertiesT$ is a set of $PropertyTypes$ that is defined by the domain.

Let $Domains$ be the set of all domains.

Model Type Definition

A model type is a tuple $\langle name, modulesT \rangle$, whereas

- $name$ is the *type name*.
- $modulesT$ is a list of $ComponentTypes$ indicating the kind of components which are considered as the *modules* of the domain.

The component types which are referred by $modulesT$ do not necessarily have to be defined in the same domain as the model type. A domain provides types, but a model type can specify module types from any domains. In the UML diagram above this is indicated by the relation "UML aggregation" for $modulesT$ (whereas the domain uses the relation "UML composition" to define components).

This permits to inherit domains by reusing type definitions of other domains.

Component Type Definition

A component type is a tuple $\langle name, propertiesT, definitionsT, inheritedT \rangle$, whereas

- $name$ is the *type name*.
- $propertiesT$ is a set of $PropertyTypes$ called the *property definitions*.
- $definitionsT$ is a set $ComponentTypes$ called the *component types*. They indicate the kind of components which can be defined in a component of this type.
- $inheritedT$ is a set of $TermTypes$ called the *inherited types* (see Section 6.2.1).

Each component type is a term type.

The set $propertiesT$ declares the "variables" of the component. At model development, properties can be assigned to terms (variable values).

The set $definitionsT$ indicates the kind of components which can be contained in a component. For example fault trees can contain gates (both are components). Event trees can contain sequences. However, in PSA models, fault and event trees are the only examples of components which contain other components.

Term Types

A term type is a tuple $\langle name, inheritedT \rangle$ whereas

- *name* is the *type name* of the term.
- *inheritedT* is a set of $TermTypes$ called the *inherited types* to specify types from which the type is inherited from.

A type is said to be compatible to its inherited types. Type compatibility is required for matching a list of arguments (terms) to signatures (of operators).

Let T_1 and T_2 be two term types with T_1 having inherited types *inheritedT*, then T_1 is compatible to T_2, written as $T_1 \sim T_2$, if

- $T_1 = T_2$ or
- $\exists T \in inheritedT$: $T \sim T_2$.

Operator Definition

An operator is a tuple $\langle name, type, signatures \rangle$, whereas

- *name* is the *operator name*.
- $type \in TermTypes$ indicates the *result type* of the operation.
- *signatures* is a set of *signatures*. Each signature is a list of $TermTypes$ indicating the kind of arguments for the operator.
  A signature is written as $[TT_1, TT_2, \ldots, TT_n]$ where TT_i is a term type.
  An operator with signature "[ ]" specifies a *constant*.

Arithmetic Example: The operator "+" has signatures $[int, int]$ and $[float, float]$ to sum integer or floating point values.

The term types *int* and $float$ are in return operators. For example, the operator *int* serves to define integer numbers. It has an unlimited number of signatures over constant definitions (representing the integer numbers):
[0],
$[1], [2], [3], [4], [5], [6], [7], \ldots$
$[-1], [-2], [-3], [-4], [-5], [-6], [-7], \ldots$

Each integer number is an operator with zero arguments (indicating a constant). For example "7" is an operator with signature "[ ]".

Property Type Definition

A property type is a tuple $\langle name, valueT, default \rangle$, whereas

- *name* is the *property name*.
- $valueT \in TermTypes$ is the *term type*.
- $default \in$ "$valueT'' \cup \{\perp\}$ is the *default value* of the property. It is either a value conform to *valueT* or $\perp$. If it is $\perp$, the property is said *mandatory*, otherwise *optional*.

PSA Model Example:

The property type ("label", String, "") is used to assign a model element to a short description. By default, the label is an empty description.

The property type ("value", Real, 0.0) is used to assign a parameter to a value (a "real" number). By default, the value is zero.

The property type ("top-gate", GT, $\perp$) serves to declare a gate as top gate in a fault tree. GT is the component type for gates. The property is mandatory (it must be specified otherwise the fault tree is considered *inconsistent*).

The property type ("inputs", $GCList$, $\perp$) serves to assign a gate to its inputs. $GCList$ is a list type to store gate components. A gate component is either a gate, a basic event or a house event.

6.2.2 Model Composition

In this section, the development of models is precised.

Figure 6.9 shows the composition of models in form of an UML class diagram.

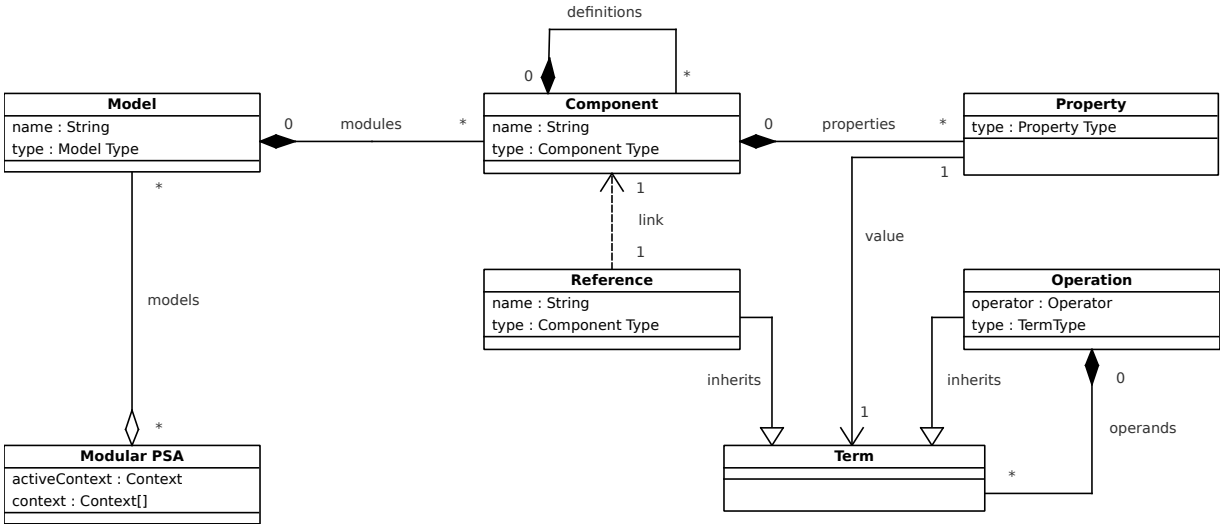


Figure 6.9. Model Composition in a modular PSA

A modular PSA consists of a set of models. Each model provides its modules in form of components (that are not contained in others).

Each component can contain further components (subcomponents). It defines further a set of properties.

Each property is assigned to a value (a term). A term is either a reference or an operation. Note that components are not considered as terms though component types are term types.

A reference describes a dependency to another component (without explicitly referring to a specific component).

An operation is a function over operands (arguments in form of terms). Operations with no arguments represent constants (elementary values).

In the rest of this thesis, let $Elements$ be the set of all model elements (models, components, references, properties, operations, terms).

Further, a pseudo element $\perp$ is defined to indicate particular outcomes of operations. However, $\perp$ is considered as an invalid element: $\perp \neq Elements$.

Any element is assigned to a $type \in Types$. Let $type : Elements \cup \{\perp\} \rightarrow Types \cup \{\perp\}$ be the function

to return the type. The type of $\perp$ is $\perp$.

The remainder of this section describes model composition in a formal manner.

Modular PSA

A modular PSA contains a tuple $\langle models, contexts, activeContext \rangle$ in which

- $models$ is a set of models. The set represents the models which are contained in the modular PSA.
- $contexts$ is a set of contexts. The set represents the instantiation contexts which are available in the modular PSA. Contexts are defined in Section 6.2.3.
- $activeContext \in contexts$ is the *active context* indicating the currently applied instantiation context.

Models

A model is a tuple $\langle name, type, modules \rangle$, whereas

- $name \in String$ is the *model name*.
- $type \in ModelTypes$ is the *model type*.
- $modules$ is a set of components. The set represents the modules of the model.

Let $Models$ be the set of all models.

Let $Modules$ be the set of modules, a subset of $Components$.

The functions $name(M)$ and $modules(M)$ are introduced to return $name$ and $modules$ of a model M.

The set of modules must be valid: Let M be a model with $type(M) = \langle name, modulesT \rangle$, then
$\forall M \in modules$: $type(M) \in modulesT$

Folders

Models store their modules in so-called "Folders". A folder is an organizational unit to group modules similar as directories group files in a file system.

A folder is a tuple $\langle name, modules, folders \rangle$ whereas

- $name$ is the folder name
- $modules$ is a set of modules stored in the folder.
- $folders$ is a set of folders called the *subfolders*. They are contained in the folder. They serve to organize modules hierarchically.

Let $Folders$ be the set of all folders. Let $F_\perp \in Folders$ be the "root folder" indicating a folder to store modules in the most "upper folder" of a model. That folder is the only folder that is not contained in another folder.

Let $folder : Components \rightarrow Folders$ be a function to return the folder of a component. Let C be a component, then $folder(C)$ delivers

- the folder that stores C in case C is a module.
- otherwise $folder(C')$ with C' is the component that defines C (see Section 6.2.2).

Components

A component is a tuple $\langle name, type, properties, definitions \rangle$, whereas

- $name \in String$ is the *component name*.
- $type \in ComponentTypes$ is the *component type*.
- $properties$ is a set of properties. Each property indicates a variable of the component and their terms variable values.
- $definitions$ is a set of components representing the set of contained components.

Let $Components$ be the set of all components.

The functions $name(C)$, $properties(C)$ and $definitions(C)$ are introduced to return the $name$, $properties$ and $definitions$ of a component C.

A set of properties $properties$ must be valid for a component C. Let $propertiesT$ be the property types defined by $type(C)$. Then, the following conditions must hold:

- $\forall P_1, P_2 \in properties : type(P_1) \neq type(P_2) \vee P_1 = P_2$
- $\forall P \in properties \exists PT \in propertiesT : type(P) = PT$.

Modules

A module is a component that is defined by a model.

Let $Modules$ be a subset of $Components$ indicating the set of all modules.

References

A reference is a tuple $\langle name, type \rangle$ whereas

- $name \in String$ is the *reference name*. This name is identical with the name of the referenced element.
- $type \in ComponentTypes$ is the *referenced type* indicating the type of the referenced element.

Let $References$ be the set of all references.

The function $name(R)$ is introduced to return the $name$ of a reference R.

Example 1: Let BT be the domain type for basic events. The reference ("be1", BT) is referring to a basic event named "be1".

Example 2: Sequences (of event trees) define references to consequences. Let CT be the domain type of consequences. Then the reference ("CD", CT) refers to a consequence named "CD" (for "core damage").

Terms

Terms serve to assign properties (the "variables" of components) to "values".

Each type is associated with a term type.

Let $Terms$ be the set of all terms:

- Each reference is a term.
- Each operation (in particular constants) is a term.

Operations

An operation is a tuple $\langle operator, operands \rangle$ whereas

- $operator \in Operators$ is called the *operator*.
- $operands$ is a list of terms called the *operands*.

Let $Operations$ be the set of all operations.

The operands of an operation must be valid: Let O be an operation with operator Op and operands $T_1, T_2, \ldots, T_n$. Then Op must have a signature $[type_1, type_2, \ldots, type_n]$ such that
$type(T_1) \sim type_1, type(T_2) \sim type_2, \ldots, type(T_n) \sim type_n$.

The operator "$\sim$" expresses type compatibility, see Section 6.2.1.

An operation O with operand Op and operands $[A_1, A_2, \ldots, A_n]$ can be denoted as $Op(A_1, A_2, \ldots, A_n)$.

An operation $Op()$ (zero operands) can be written as Op and is called a *constant*.

Example:
Let *int* be a term type and $0, 1, 2, \ldots$ constants of type *int*.
Let "+" be an operator of type *int* and $[int, int]$ a signature of it.
Let "$*$" be an operator of type *int* and $[int, int]$ signatures of it.
Then "$*(+(4,6), *(4,8))$", "$+(5, +(5,3))$" and "$*(4, +(5,2))$" are terms (of type "*int*").

Properties

A property is a tuple $\langle type, value \rangle$, whereas

- $type \in PropertyTypes$ is the *property type*.
- $value$ is the *property value*.

Let $Properties$ be the set of all properties.

The functions $type(P)$ and $value(P)$ are introduced to return the *name* and *value* of a property P.

A property value must conform to the term type of *type*.

Example 1: The property `(label, ‘‘This is a parameter’’)` can be used to provide a short description. The `label` property is available for all component types of a modular PSA.

Example 2: The property `(value, ‘‘3.5e-10’’)` can be used to assign a basic event to a probability value. The `value` property is reserved to certain component types (e.g. basic events and parameters).

Characteristics

Containment Let "$\in$" be an operator to indicate whether an element is contained in another element.

Let E be an element and T a term, then $E \in T :=$

- $(E == T)$ or
- $(E \in A_1) \vee (E \in A_2) \vee \ldots \vee (E \in A_n)$ with $T = Op(A_1, A_2, \ldots, A_n)$

Let E be an element and C a component, then $E \in C :=$

- $(E == C)$ or
- $\exists P \in properties(C) : E \in value(P)$ or
- $\exists C \in definitions(C) : E == C$

Let E be an element and M a model, then $E \in M :=$

- $(E \in modules(M))$

Let $\in_{Rec}$ be the operator to indicate if an element is recursively contained in another.

$E_1 \in_{Rec} E_2 :=$

- $E_1 \in E_2$ or
- $\exists E_2'$: $(E_2' \in E_2) \wedge (E_1 \in_{Rec} E_2')$

Property Value The function *property* from $Components \times PropertyTypes$ to $Terms \cup \{\bot\}$ returns the property value of a component:

$$property(C, PT) = \begin{cases} V & \text{if } \exists P \in C : P = (PT, V) \\ default & \text{with } PT = (name, valueT, default) \text{ otherwise} \end{cases}$$

If property PT is not set, the default value for PT is returned. The default value is $\bot$ in case PT is mandatory.

Enabled / Disabled Elements Components can be disabled / enabled. If disabled they are treated like not present. This can be used to assemble a specific set of modules and to deactivate modules not required.

The activation / deactivation of components is done via a property type called the "enabled" property type $P_{Enabled}$. The property type has value type "Boolean" and its default value is "True" (the property type is defined as $\langle$"*enabled*"$, Boolean, True\rangle$).

A component C is said *enabled*, if $property(C, P_{Enabled}) = True$.

A component is said *disabled*, if it is not enabled.

Example: A fault tree FT defines three gates $G1, G2$ and $G3$. In order to deactivate $G2$, it can set the "enabled" property to value "False". The resulting fault tree has only gates $G1$ and $G3$.

And setting the "enabled" property of the fault tree to "False" results in deactivating the whole fault tree.

Owning Model The *owning model* of an element E is a model $M \in Models$ with $E \in_{Rec} M$.

The owning model contains (recursively) its elements. By definition, each model element has exactly one owning model.

Let $model : Elements \to Models$ be the function to return the owning model of an element.

6.2.3 Model Instantiation

In this section, the instantiation principle of the modular PSA concept is precised.

An instantiated model is a tuple $\langle M, \tau \rangle$ where

- M is the model to instantiate given in its generic form.
- τ is a context called the *instantiation context*.

Context

Models in a modular PSA need to be instantiated due to a specific context. Contexts reflect kind of specific circumstances, that are not directly encoded in models. At instantiation, models get specifically adapted to the respective context.

A context defines instantiation parameters:

- A ordered list of includes $[I_{-m}, I_{-m+1}, \ldots, I_{-1}, I_0, I_1, \ldots, I_{n-1}, I_n]$, ordered from -m to +n.
- A ordered list of adaption rules $[r_1, r_2, \ldots, r_n]$.

Let $includes(M, \tau)$ be a function that returns the list of includes for a model M respective a context τ.

Let $rules(M, \tau)$ be a function that returns the list of adaption rules for a model M respective a context τ.

Includes An include is a tuple $\langle M, recursive \rangle$, whereas

- $M \in Models$ is a model called the *included model*.
- $recursive \in Boolean$ indicates whether the includes is *recursive* or not (*non-recursive*).

A model is always including itself (reflexive). The reflexive include for a model M is defined by $I_0 = \langle M, False \rangle$:

$$\forall M \in Models : I_0 \in includes(M, \tau)$$

The includes of a model M can be written in the following way:

```
Position      Model        recursive
 -m:        <model_1>     <rec_1>     // Include -m
 -m+1:      <model_2>     <rec_2>
 ...
 -1:        <model_m>     <rec_m>
  0:        M             False       // Include 0 (reflexive include)
 +1:        <model_m+1>   <rec_m+1>
 +2:        <model_m+2>   <rec_m+2>
 ...
 n:         <model_m+n>   <rec_m+n>   // Include +n
```

Includes at position -m to -1 are called *replacing includes*. Includes at position +1 to +n are called *completing includes*. Include 0 exists by definition and cannot be manually specified. Both, m and n may be zero. If m is zero, the defining model does not contain any replacing includes. If n is zero, the model does not contain any completing ones.

Adaption Rules Adaption rules are used to adapt a model at instantiation by flavoring property values (of components) respective to a given context.

An adaption rule is a function r: $Components \times PropertyTypes \times Terms \rightarrow Terms \cup \{\perp\}$.

Let C be a component, PT a property type and T a term, then $r(C, PT, T)$ adapts the term T assigned to the property PT of component C (see Section 6.2.3).

Let $Rules$ be the set of all adaption rules.

Default Context Let $\tau_\perp$ be the *default context*. The default context has the following property: $\forall M \in Models$:

- $includes(M, \tau_\perp) = [\langle M, false \rangle]$.
- $rules(M, \tau_\perp) = []$.

Instantiation Function

The instantiation of a model M is expressed by two functions:

- $term : Components \times PropertyTypes \times Context \rightarrow Terms \cup \{\perp\}$ is called the *property adaption*.
- $resolve : References \times Contexts \rightarrow Components \cup \{\perp\}$ is called the *reference resolution*.

Let C be a component, PT a property type and τ a context. Then $term(C, PT, \tau)$ is the property term after instantiation respective a context τ. $\perp$ indicates an invalid term.

Let R be a reference and τ a context. Then $resolve(R, \tau)$ is the component or $\perp$ indicated by R respective a context τ. If the function returns $\perp$, the reference can not be resolved.

The two functions are specified in the sequel.

Property Adaption Let $[r_1, r_2, \ldots, r_n]$ be a set of adaption rules of model M in context τ. The rules are applied in sequence. Let T_i be the term after rule r_i has been applied, then the property adaption for M is given by

$$term(C, PT, \tau) = T_n$$

where

- $T_0 = property(C, PT)$
- $T_i = r_i(C, PT, T_{i-1}) \quad$ for $1 \leq i \leq n$

Reference Resolution Model instantiation requires to resolve references in order to identify referenced components.

Let R be a reference, M the owning model of R and τ an instantiation context. Then the reference resolution of R for M is given by

$$resolve(R, \tau) = it(R, includes(M, \tau), \tau)$$

In order to resolve a reference R, a component C must match R by name and type: Let $C = resolve(R, \tau)$ with $C \neq \perp$, then

$$name(C) = name(R) \wedge type(C) = type(R)$$

The function "it" iterates over the included models $[M_1, M_2, \ldots, M_n]$ of M and "tries" to resolve R. If R cannot be resolved in a model M_i, the resolution is performed in the successive model M_{i+1}. As soon as the reference is resolved, the procedure terminates. If the reference could not be resolved in any of the included models, the function returns $\perp$.

Let $L = [I_1, I_2, \ldots, I_n]$ be a ordered list of includes for resolving a reference R. Function "it" is specified

as follows (the "@" operator joins two lists):

$$it(R, L, \tau) = \begin{cases} \bot & \text{if } L = [] \\ \hline it(R, includes(M_1, \tau)@[I_2, \ldots, I_n], \tau) & \text{else if } I_1 = (M_1, True) \\ \hline resolve_M(R, M_1) & \text{else if } I_1 = (M_1, False), \\ & \quad resolve_M(R, M_1) \neq \bot \\ \hline it(R, [I_2, I_3, \ldots, I_n], \tau) & \text{otherwise} \end{cases}$$

Finally, to resolve a reference in one specific model, a function $resolve_M$ from $References \times Models$ to $Components \cup \{\bot\}$ is introduced:

$$resolve_M(R, M) = \begin{cases} C & \text{if } \exists C \in Components, C \in_{Rec} M : \\ & \quad name(C) = name(R) \ \wedge \ type(C) = type(R) \\ \bot & \text{otherwise} \end{cases}$$

If $resolve(R, \tau) = \bot$, then R is an *unresolved reference* in context τ.

6.2.4 Consistency

A model is said to be *consistent* if it holds a set of criteria. A model that is not consistent, is said *inconsistent*.

A modular PSA is consistent, if all contained models are consistent. Otherwise it is inconsistent.

The following criteria must be satisfied to have a consistent model.

Uniqueness of Components

Components of the same model must be unique by name and type. Let M be a model. The criterion is satisfied if

$$\forall C_1, C_2 \in Components, C_1 \in_{Rec} M, C_2 \in_{Rec} M, C_1 \neq C_2 : \\ name(C_1) \neq name(C_2) \vee type(C_1) \neq type(C_2)$$

Two components of the same type that are named equally (in the same model) constitute an inconsistency that has to be corrected in the model.

This criteria leads to the question how to avoid naming collisions, which may get likely in large models. One could think of namespaces or kinds of scope. However, for the case of PSA models at EDF, those additional mechanisms were not necessary. EDF uses strong naming conventions in PSA models to ensure uniqueness of model components. In addition, the idea to refer to model elements uniquely by name and type is derived from the format "OPSAMEF" (see Section 5.2) which was used to convert real PSA models from RiskSpectrum to Andromeda (so the test models satisfied this criteria).

However, in general one should consider a mechanism to avoid naming collisions (though not further analysed in the scope of this manuscript).

Unresolved References

Let M be a model. A reference R is an *unresolved reference* respective a context τ if

$$resolve(R, M, \tau) = \bot$$

An unresolved reference expresses that a dependency between model elements could not be resolved.

A model must not contain unresolved references. Let M be a model instantiated respective a context τ, then

$$\forall R \in_{Rec} M : resolve(R, \tau) \neq \bot$$

Mandatory Properties

Mandatory properties of components must be set: Let PT be a mandatory property type for a component C, then

$$\exists P \in properties(C) : type(P) = PT \ \wedge \ value(P) \neq \bot$$

6.2.5 Modeling Operations

Modeling operations are always applied to the generic model. Then the generic model gets instantiated. Note however that a user "sees" models always in their instantiated form. But as soon as he/she modifies a component, the modification itself is applied on the generic model and the model automatically reinstantiated.

The benefit of the modular PSA framework is its generic specification that permits to express functionality and characteristics on a generic level. Consequently, modeling operations can be applied to models regardless their type. Some generic modeling operations are precised in the sequel.

Component Creation

Models can create components (modules) and components can define further components (sub components).

To create a new component C' in a model M (or a component C), two criteria must be satisfied:

- M must not contain another component that has the same name and type as C'.
- The type of C' must be a valid component type for M (or C).

Component Import

A *component import* refers to the replication of one or several components from a *source model* into a *target model*, whereas the source model differs from the target model.

Components of the same model must be unique by name and type (see 6.2.4). This consistency criteria must be satisfied after a module import.

Let C be a component to import and C_{Set} the set of involved components (C and any component recursively contained in C). Then C_{Set} is not supposed to conflict with any components C' in M:

$\forall C' \in_{Rec} M \ \nexists C'' \in C_{Set} : (name(C') = name(C'')) \wedge (type(C') = type(C''))$

For each conflicting component $C'' \in C_{Set}$ three possibilities exist to resolve the conflict:

- Component C'' is not imported at all.
- Component C' is overwritten by component C''.
- Component C' is merged with component C''. In this case a new component is created which represents the fusion of C' and C'' (see Section 7.4).

The described procedure imports component C and its subcomponents. However, dependent components are not imported. The set of dependent components depends on a context τ. The import procedure can be extended to a so-called *recursive import* to import extended modules as well.

For example, a model A contains a fault tree ft and two basic events:

```
Model A
├── Def Fault-Tree ft
│   ├── Def gate1 OR
│   │   ├── be1*
│   │   └── gate2*
│   └── Def gate2 AND
│       └── be2*
├── Def Basic-Event be1
└── Def Basic-Event be2
```

The format used to describe the example fault tree is explained in Section 6.5.3.

An *import* of ft imports also $gate1$ and $gate2$, but not automatically $be1$ and $be2$.

On the contrary, a *recursive import* of ft in a context τ imports the basic events as well (assuming they are referenced in τ from ft).

The non recursive import can be dangerous as $be1*$ and $be2*$ may be resolved unexpectedly (or not at all). However, software tools can recommend (or even force) to import recursively and to cross check dependencies after an import.

In large PSA models, a recursive import can quickly import hundreds of components. For example, the recursive import of an event tree imports function events, consequences, fault trees, basic events, initial events and parameters as well.

Component Modification

Changing a component C consists in assigning one of its properties to a new term value. Let $type(C)$ assigned to property types $propertiesT$ and $PT \in propertiesT$. Then property PT of C is assigned to a term T in the following way:

- if $\exists P \in C \ type(P) = PT$, then $value(P)$ is changed to T.
- else a new property P is created with $type(P) = PT \ \wedge \ value(P) = T$ and added to the list of properties of C.

Component Renaming

A special case of modification is renaming: Renaming a component is to modify its name (technically to change the term of property $name$). To rename a component C of a model M in context τ to a new name N leads to different situations. If another component C' with $name(C') = N$ and $type(C') = type(C)$

exists already in the same model, the component cannot be renamed. Otherwise there are different possibilities:

- Rename component C without renaming any references. Consequently, references point either to another component C' with $C' \notin M$ or to $\perp$ (unresolved reference). This depends on the include configurations of models.
- Rename component C and likewise any references which are pointing to C in context τ (All references R with $resolve(R, \tau) = C$). In this case, renamed references may reside in other models than M.
- Rename component C and likewise any references which are pointing to C in the default context τ_D. All references of M having the same name and type as C are renamed. References from other models than M are not impacted.

Component Duplication

Component duplication consists in replicating a component C of a model M to a component C' which is stored in M as well.

To duplicate a component C in model M involves the following steps

- Creation of a component C' with same type as C but a unique name respective M.
- Copy properties P from C to C'. $\forall PT : value(C, PT) = value(C', PT)$.
- Duplicate all components contained in C. Then rename all references of C' which point to a component of C to the respective duplicated component.

The described procedure replicates component C. However, dependent components are not replicated. If dependent components are replicated as well - what is a *recursive duplication* - all references of C' which point to one of the dependent components must be renamed to point to the replicated components.

This is illustrated by the following example, where fault tree ft is duplicated:

```
to replicate:                duplication:                 recursive duplication:
├── Def Fault-Tree ft        ├── Def Fault-Tree ftR !     ├── Def Fault-Tree ftR   !
│   ├── Def gate1 OR         │   ├── Def gate2R OR  !     │   ├── Def gate2R OR    !
│   │   ├── be1*             │   │   ├── be1*             │   │   ├── be1R*        !
│   │   └── gate2*           │   │   └── gate2R*    !     │   │   └── gate2R*      !
│   └── Def gate2 AND        │   └── Def gate2R AND !     │   └── Def gate2R AND   !
│       └── be2*             │       └── be2*             │       └── be2R*        !
├── Def Basic-Event be1      ├── Def Basic-Event be1      ├── Def Basic-Event be1R !
├── Def Basic-Event be2      ├── Def Basic-Event be2      ├── Def Basic-Event be2R !
├── Def Basic-Event be3      ├── Def Basic-Event be3      ├── Def Basic-Event be3
```

Basic event $be3$ is never duplicated as it is not referenced in ft. The "!" indicates the renamed parts after replication.

Component Deletion

Deleting a component consists in removing it and all its contained components from its owning model M. Removing a component C in context τ can lead to several possibilities (a model engineer has to decide):

- None of the references pointing to C are removed.
- All references which reference C or one of its (recursively) contained components in context τ are removed as well.

- All references which reference C or one of its (recursively) contained components in the default context τ_D are removed as well.

6.3 Comparison to the Original Modular PSA

The presented concept of a modular PSA extends and differs in some points from the original ideas about a modular approach (presented in Section 5.1).

In this section, a brief comparison between the two approaches is given. The comparison provides a deeper understanding of the advantages of the concept.

6.3.1 Module Definition

Hibti describes modules as "different parts [of a PSA model], with respect to the scope, the objective, and the application." [5, p. 3]

In the new approach, the granularity of the module definition has been decided to be on a more detailed level: Modules in the new approach are individual model objects, which have a generic and an instantiated form. They permit to assemble models by selectively picking individual model objects.

6.3.2 Context

The new approach introduces the notion of a so-called "*context*". A context is an abstract construct, which impacts the instantiation process to obtain a context specific model.

The need to deal with different contexts has already been indicated in the original approach: It was described as the "scope, the objective, and the application" [5] of a PSA. However, it was technically not further precised.

6.3.3 Generic Model

In the new approach, there is no definition of a generic kernel, which could be common to all derived PSAs. Instead, a set of generic models is required for an instantiation. This set can vary respective to different instantiations.

A generic kernel has the following disadvantage: It may be difficult to identify a generic kernel adequate for all PSAs. The set of what is considered as "shared data" may differ dramatically between different PSAs. With an increasing number of derived PSAs, a generic kernel may be impossible to define without blowing up the size of the kernel. And model objects which are not part of the kernel are not "reusable" in other PSAs.

On the contrary, by having an individual set of generic models for each instantiation permits to reuse any model data no matter if considered "common" or "PSA specific". No explicit separation between "common" and "specific" data is required. Indeed, this separation becomes instantiation specific and is not realized on model level.

The new approach generalizes the original approach: If different PSAs are based on the same set of generic models, this set can be regarded as the "generic kernel".

6.3.4 Dependency Management

Dependencies between model objects and their management has become one of the major concepts in the new approach. In the original approach, object dependencies have not been analysed in particular.

6.4 Format Specifications

In this section, two formats are presented. The first permits to present models of a modular PSA in a unique way. The second serves to specify adaption rules for property adaption.

6.4.1 Generic Representation of a modular PSA

In this section, a format is presented to represent models of a modular PSA in a compact way. The format serves to indicate modular PSA examples in this thesis without formally specifying components, references, operators, properties etc.

Each model of a modular PSA is stated individually.

The format is best explained by an example:

```
Def University-Model example
├── Def Building b1
│   ├── label = This is the main building
│   ├── Def Room room1
│   │   ├── label = This is the meeting room
│   │   └── people = [Professor p1*, Student s1*, Student s2*]
│   ├── Def Room room2
│   │   ├── label = This is room2
│   │   └── people =
│   │       ├── Professor p1*
│   │       ├── Student s1*
│   │       └── Student s2*
│   └── Def Room room3
│       ├── label = This is room3
│       └── people = []
├── Def Student s1
│   ├── label = This is student s1
...
```

The model is listed hierarchically in a tree structure. Nodes (of the tree structure) have child nodes what indicates that elements are contained in other. The symbol "`*`" indicates references, e.g. `s1*` indicates a reference to a student named `s1`. The expressions in each line conform to the explications of Table 6.1.

Properties can either be stated to the right as one term expression (for example property `people` of `room1`) or below in tree form (for example property `people` of `room2`). Both express the same term.

Operators which serve only for the purpose of type conversion, are not required to be explicitly stated. For example "`float(2.5)`" to convert a constant to a Real number is not necessary ("`2.5`" is sufficient). Also "`gate-type(OR)`" to convert a constant to a gate type is only a type conversion ("`OR`" is sufficient). Anyway, type conversions can be performed automatically and are not required to be stated explicitly.

Table 6.1. Compact Format to describe a modular PSA

Expression	Description	Example
DEF <Type> <Name>	Definition of a component or model of type Type with name Name	DEF Room room1
<Property> = <Term>	Property with name Property is assigned a term Term	label = This is student s1
<Property> = [<T1>,<T2>,...,<Tn>]	Property with name Property is assigned a list of terms	people = [Professor p1*, Student s1*,Student s2*]
<Type> <Name>*	Reference to component of type Type with name Name	Professor p1*

6.4.2 Format for Adaption Rules

To specify adaption rules, a specific format has been developed. This format is explained in this section.

Example of Adaption Rules

The format is best introduced by example:

```
gate[name=gate3]:gatetype         SET        and
*:*                               CHANGE-REF gate_reference gate2 gate4
*:documentation                   REPLACE    Bugey Flamanville
fault_tree[label=/fire/]:enabled SET        false
```

Each line represents one adaption rule. The first rule serves to adapt the gate type of gate `gate3` to an `AND` gate. The second rule redirects any gate references `gate2` to `gate4`. The third rule replaces any occurrences of `Bugey` by `Flamanville` in the whole model documentation. The last rule deactivates all fault trees whose label contains the string "`fire`" (denoted as regular expression).

Format Specification

Adaption rules have been introduced in Section 6.2.3. In the sequel, let T_i be the term after application of rule r_i.

In the format, rules are of the form
`<SELECTOR>: <PROPERTY> <INSTRUCTION>`, whereas

- `<SELECTOR>` is called the element selector and encodes a condition $Cond(C)$ where C is a component.
- `<PROPERTY>` refers to a property type PT.
- `<INSTRUCTION>` represents an adaption function $f(T, A_1, A_2, \ldots, A_n)$ to adapt T.

If condition $Cond(C)$ is not satisfied in a rule r_i, then $T_i = T_{i-1}$ (the term remains unchanged).

If the condition $Cond(C)$ is satisfied, $T_i = f(T_{i-1}, A_1, A_2, \ldots, A_n)$

Element Selector The element selector encodes the condition $Cond(C)$. It is of the form

`<SUB-SELECTOR-1> / <SUB-SELECTOR-2> / ... / <SUB-SELECTOR-n>`

Each `<SUB-SELECTOR-i>` is of the form

`<ELEMENT-TYPE> [ <SUB-CONDITION-1>, <SUB-CONDITION-2>, ... ,<SUB-CONDITION-m>]`

Each "<SUB-CONDITION-i>" is of the form

<PROPERTY-TYPE> <BOOLEAN-OPERATOR> <PROPERTY-VALUE>

<PROPERTY-TYPE> indicates a property type and <PROPERTY-VALUE> a property value (a term).

<BOOLEAN-OPERATOR> is one of the following: "=" (equal), "! =" (not equal), ">" (larger), "<" (smaller, ">=" (larger or equal), "<=" (smaller or equal) and "=/" (test against a regular expression). All operators are case insensitive.

The condition $Cond(C)$ of a rule r_i to select a component C is satisfied if the element selector of r_i selects C.

"An element selector <SELECTOR> selects C", if "<SUB-SELECTOR-n> selects C".

A "<SUB-SELECTOR-n> selects C", if the following conditions are satisfied:

- $name(type(C))$ = <ELEMENT-TYPE>
- for all <SUB-CONDITION-i> = <T_i> <O_i> <V_i>:

  $$property(C, T\_i)\ \ Op\ \ V\_i$$

  whereas Op is the Boolean operator indicated by <O_i>
- "<SUB-SELECTOR-(n-1)> selects parent(E)" (for $n > 1$), whereas $parent(E)$ is the element that contains E.

Instructions Some instructions are:

- SET <V>: This instruction encodes the adaption function $f(T, V) = V$
- REPLACE <R> <V>: This instruction encodes the adaption function $f(T, R, V) = replace(T, R, V)$, whereas $replace(A, B, C)$ is the replacement function replacing any occurrence of B (denoted as regular expression) in A by C.
- CHANGE-REF gate_reference gate2 gate4 encodes the adaption function $f(T, RT, RN, RN') = T'$ whereas in T' any references of type RT and name RN have been renamed to RN'.

Conclusion

The format targets to be compact and efficient to apply. A more complex approach may allow to encode more powerful adaptions. However, to meet the goals of the modular PSA idea, the format has turned out to be efficient, effective and intuitive.

6.5 PSA Models in a modular PSA

In this section it is demonstrated how PSA models are built in a modular PSA.

6.5.1 Fault Trees

Example of a fault tree in a modular PSA:

```
Def PSA-Model
├── Def Fault-Tree ft1                                    (*1)
```

```
|   ├── label = fault tree for pump1
|   ├── Def Gate g1                                          (*2)
|   |   ├── label = failure pump1                            (*3)
|   |   ├── type = gate-type(OR)                             (*3)
|   |   └── inputs = [Gate g2*, Gate g3*]                    (*3)
|   ├── Def Gate g2                                          (*4)
|   |   ├── label = mechanical failure
|   |   ├── type = AND
|   |   └── inputs = [House-Event he10*, Basic-Event be10*]
|   ├── Def Gate g3                                          (*5)
|   |   ├── label = electrical failure
|   |   ├── type = AND
|   |   └── inputs = [House-Event he11*, Basic-Event be11*]
|   └── top-gate = Gate g1*                                  (*6)
...
```

Detailed Explications:
(*1): A fault tree `FT1` is defined.

(*2): The fault tree defines a gate, i.e. the gate is one of its component definitions.

(*3) The gate declares three properties: (1) The term type of the label property is a "String". (2) The operator `gate-type` converts a String "OR" to a "gate type" (which is the required term type). However, the operator `gate-type` serves only for type conversion and is not required. (3) The term type of `inputs` is a list of "Gate Components". Basic events, house events and gates inherit this type.

(*4 and *5) Two more gates similar as in (*2) are introduced. This time, the terms for property `inputs` are built over basic event and house event references.

(*6) The property `top-gate` for the fault tree: The property has term type "Gate". The term is a gate reference to "g1".

The house and basic events referred to from gates are not yet defined. They are defined as modules in separate sections, for example:

```
...
├── Def Basic Event be10
|   ├── label = mechanical failure
|   └── parameters = [p123*]
├── Def Parameter p123
|   ├── value = 1.e-10
|   └── distribution = exponential
...
```

6.5.2 Event Trees

Example of an event tree in a modular PSA:

```
Def PSA-Model
├── Def Event-Tree et1                                              (*1)
|   ├── label = fire in building123
|   ├── Def Sequence s1                                             (*2)
|   |   └── consequences = [Consequence OK*]
|   ├── Def Sequence s2
|   |   └── consequences = [Consequence P1*, Consequence OK*]
|   ├── Def Sequence s3
|   |   └── consequences = [Consequence CD*]
```

```
|   ├── initial-state = Initiating-Event fire*                    (*3)
|   └── mitigation =                                              (*4)
|       └── fork                                                  (*5)
|           ├── Function-Event mitigation1*
|           ├── path
|           |   └── Sequence-Reference s1*
|           └── path
|               └── fork
|                   ├── Function-Event mitigation2*               (*6)
|                   ├── path
|                   |   └── Sequence-Reference s2*
|                   └── path
|                       └── Sequence-Reference s3*
├── Def Initiating-Event fire                                     (*7)
|   └── link = Gate beFire*
├── Def Function-Event mitigation1                                (*8)
|   └── link = Gate be12*
...
```

Explications:
(*1) Definition of an event tree component (module)

(*2) Definition of a sequence. A sequence is linked to consequences via a property `consequences`.

(*3) Declaration of an initiating event for the event tree

(*4) The mitigation term for the event tree. The term consists of an operator `fork` which defines the mitigation sequences.

(*5) To specify the mitigation, two operators `fork` and `path` are used to create a tree-like structure. The "leaves" of the tree are (references to) sequences. Operator `fork` has signatures `[function-event,path, path, ...]` whereas (by default) the first path is considered as *success path* and subsequent paths as *failure paths*.

(*6) Linkage to a function event.

(*7) Definition of an initiating event. The event is linked to a basic event to define its occurrence characteristics.

(*8) Likewise initiating events, function events link a gate or a basic event. In the example, the term is a gate reference.

6.5.3 Representation Format for PSA Models

Fault trees, basic events and event trees, however, are stated even more compact throughout this thesis.

Example:

```
Def PSA-Model example model
├── Def Fault-Tree ft1
|   ├── Def gate1 OR                                        (*1, *2)
|   |   ├── be1*                                            (*3)
|   |   ├── be2*
|   |   └── gate2*
|   └── Def gate2 AND
|       └── be3*
├── Def Event-Tree et1
|     init*  | fe1*  |   fe2*      sequences   consequences  (*4)
```

```
                                    S0          C0*
                                    S2          C1*
                                    S1          C1*
                                    S2          C2*
├── Def Consequence c1
│   └── label = This is a consequence
├── Def Basic-Event be1
│   └── Parameter*                                        (*5)
...
```

The compact format is based on assumptions:

(*1) The first gate of a fault tree indicates the top gate (i.e. property "top-gate" is not needed). The type *GATE* after "Def" is not required as gates are the only components defined in fault trees.

(*2) The gate type is written next to the gate name (e.g. `Def xy OR` specifies an "OR-gate" named "xy").

(*3) References stated "below" a gate indicate gate inputs (property "gate-inputs" is assumed). The type name of a reference is only required if it is not obvious from the reference name (for example *gate*2∗ indicates obviously a gate reference).

(*4) Event trees are represented as typical "event-tree structure".

(*5) References stated "below" a basic event indicate its parameter specification (property "parameters" is assumed).

6.6 Conclusion

In this chapter, the concept of a modular PSA has been introduced.

The principle components of a modular PSA are "*modules*" such as fault trees, event trees and basic events. A modular PSA consists of various models, whereas each model can be of a different domain and provides its modules. Modules can develop dependencies between each other, even in case they are located in different models.

The powerful instrument of a modular PSA is a context sensitive model instantiation technique, which permits to flavor models (i.e. to adapt models without changing their content). A context provides adaption rules (in order to adapt the content of a model) and includes (in order to adapt dependencies between modules). One concept, that exploits this concept exhaustively, is named *variant management*, which is explained in Chapter 7.1.

The application of the modular PSA concept to express PSA models could benefit from preliminary work realized within the "Open PSA Model Exchange Format" *OSPAMEF* [4] specification. This format has been briefly introduced in Section 5.2. *OPSAMEF* identifies the principal objects of PSA models and their relations to others. Those objects have been taken as base to introduce component definitions. Further, *OSPAMEF* has a likewise philosophy of separating object definitions from usage (expressed by references).

However, a modular PSA is a generic framework. A domain engineering phase serves to provide new model types. The framework has to prove that it is adequate to express also other model types. For the case of PSA models, only few operators had to be introduced. The language of PSA models is rather low level and did not require to introduce complex terms. Indeed, the *mitigation* term of an event tree is the only term that comprises a certain complexity. All other terms are more or less a set of references.

CHAPTER 7

Model Management

Model engineering involves various activities for developing models, for example a version management to track model evolutions and consistency checks to increase model quality and efficiency at development. In this thesis, the set of all model-related activities is introduced with the term *model management*

Current PSA model engineering is based on a rather simple form of model management. For example, to track the evolution of models, full copies of models are performed. Those full copies do not provide the typical features of version engineering such as to recover ancient model versions, to merge and compare models etc. Consequently, model engineering can sometimes become tricky and inefficient. And missing features to detect model inconsistencies or to verify a set of applied modifications (to "cross-check" them) can decrease model quality.

In this chapter, the concepts around model management in a modular PSA are presented. It comes along with a set of functionality for example to track model versions, to compare models, to fusion models and to manage so-called "model variants".

Section 7.1 presents a method of variant management for a modular PSA that targets to manage model deviations.

Section 7.2 demonstrates how the evolution of models can be tracked by managing so-called "model versions". The principle of version management is explained.

Section 7.3 presents a feature to compare models and to visualize the differences.

Section 7.4 explains the method of model fusion (to "merge" a set of models).

Section 7.5 gives a brief overview about the detection of model inconsistencies.

Section 7.6 concludes the chapter.

7.1 Variant Management

Variant management is the development and management of so-called *variants*. A variant expresses a deviation to adapt a model to specific circumstances. Specific circumstances may be of physical or functional nature. For example, two PSA models for different nuclear power plants may vary because one model has to consider four Diesel generators and the other only two of them. Another example is that the probabilities of external hazards such as flooding or seism typically depend on the geographical location of power plants.

The number of variants may increase during the life-time of PSA models. New variants may become necessary due to technical requirements or technical enhancements of critical systems.

In this section, the concept of variant management for a modular PSA is presented. An article about variant management has been published in [49].

7.1.1 Variant Management at Software Engineering

Variant handling is often treated to be an issue of configuration management (CM) and therefore it is often solved by version control systems (VCS). In [50], the management of variants by CM is investigated.

Based on CM or not, there are several approaches of variant management: One is the so-called *Clone and Own* [50] method. New variants are developed by creating an exact copy of an existing variant. Then, modifications for the variant are applied. The advantage is that variants can be developed independently from each other. But the downside is that this approach does not take into account that variants may share functionality. Future modifications may have to be applied various times (once for each variant), what is not only time consuming but also error-prone.

Another approach (that mitigates some problems of *Clone-and-Own*) is to split a project (or a model) in various components, which can be developed independently from each other. A new variant is then created at integration phase when integrating a set of components in their appropriate versions. In [50], this approach is referred to as *Independent Component Teams*. It works quite well as long as components have relatively simple dependencies. But when dependencies between components get more complex, the selection of component versions at integration phase may become a complex issue. The difficulty is to ensure compatibility between the integrated component versions.

Further, the concept of *feature oriented software development* (FOSD) extends the component based method to configure variants via so-called "features" [51, 52]. Each feature describes a characteristic (a constraint) about the software. Given a set of features the set of required components (in their appropriate versions) can be derived.

7.1.2 Primitive Forms of Variant Management

In this section, current methods of handling variants in PSA models are introduced. The drawbacks with those are mentioned.

House Events to perform Variant Management

In PSA models, one way of variant management is to use house events. House events are like Boolean flags that can activate / deactivate sub trees in fault trees. They are configured individually for each

variant.

Figure 7.1 demonstrates the method. The variant specific parts of a fault tree are added below *AND* gates. Each *AND* gate contains a house-event that is set to *True* to enable the variant specific part, and to *False* to disable it.

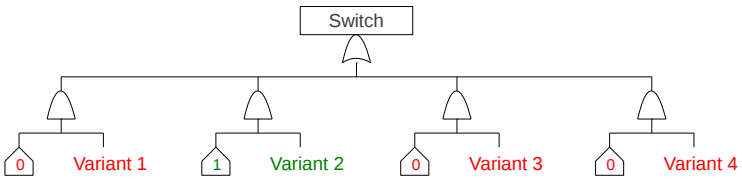


Figure 7.1. Using house events to configure variants.

Unfortunately, this method has some drawbacks: In large models, the development and configuration of house events can be time-consuming and error-prone. The challenge is also to ensure that exactly one variant specific part is enabled below *AND* gates and all other parts are disabled (to implement a "switch" behavior).

Maintaining separate Models

Another approach to express variants is to maintain each variant by an individual PSA model. To create a new variant, a model copy is created which is then separately maintained. This approach is like the mentioned "Clone and Own" concept at software engineering.

The drawback of this method is obvious: Any future modifications of common model parts must be replicated to all other variants. This is a typical problem of redundancy. The replication of modifications in other variants requires a good understanding of the PSA models in order to decide whether a certain modification is applicable for a variant or not.

Comparing different variants of the same industrial sector (for example nuclear branch) yields, that large parts of PSA models are indeed overlapping. The challenge is to find a strategy to maintain overlapping parts only once.

7.1.3 Technical Concept

In this section, the variability concept of a modular PSA is presented.

In a modular PSA the concept of model instantiation is used to handle variants. Models in their generic form do not know the notion of variants. But as soon as they get instantiated, they can adopt variant specific model characteristics.

Standard Model

A *standard model* or *default model* is an instantiated model $\langle M, \tau_\perp \rangle$ (see Section6.2.3), whereas $\tau_\perp$ is the default context.

The default context has been defined in Section 6.2.3.

Variant

A *variant* is an instantiated model $\langle M, \tau \rangle$, whereas τ is **not** the default context $\tau_\perp$.

A variant $\langle M, \tau \rangle$ represents a deviation from a standard model $\langle M, \tau_\perp \rangle$.

Consequently for each variant a new context is introduced. The instantiation parameters of the context (includes and adaption rules) reflect the specific circumstances of the variant.

Though not completely proper, the phrase "a model M in variant V" refers to the variant $\langle M, \tau \rangle$, whereas τ is the associated instantiation context of V.

A variant expresses deviations from a standard model. On a functional level, four operations can be identified to model deviations:

1. Adapting a module: This operation is realized by property injection (see Section 7.1.3) which uses **adaption rules** to adapt properties of a module E. Consequently, E becomes a module $f(E)$ whereas f modifies property values (terms) of E.
2. Substituting a module: A module E is substituted by a module E' by redirecting any module dependencies from E to E'. Technically, this method requires to configure **includes** in an appropriate way to resolve dependencies towards E'. The operation is described more in detail in Section 7.1.3.
3. Adding a module: A variant may require additional modules. New modules can be provided in a new (variant specific) model. The method requires to configure **includes** in order that existing modules resolve their dependencies towards the new ones.
   Adding modules can be seen as a special case of module substitution whereas the module to substitute is $\perp$.
4. Removing a module: Modules can be deactivated by setting their property *enabled* to *False*. Deactivated components are not part of an instantiated model, i.e. they are handled as being removed. Removing modules uses **adaption rules** to configure the set of deactivated modules.

The functions do not require any extensions of the modular PSA framework. They use exclusively the instantiation concept.

Variant engineering is illustrated by two examples. The first example in Figure 7.2 illustrates the creation of "*variant A*" by the means of a context "*context A*". Module *6* is adapted, modules *2* and *8* are substituted by *2A* and *8A*, *9A* is added and module *5* is removed.

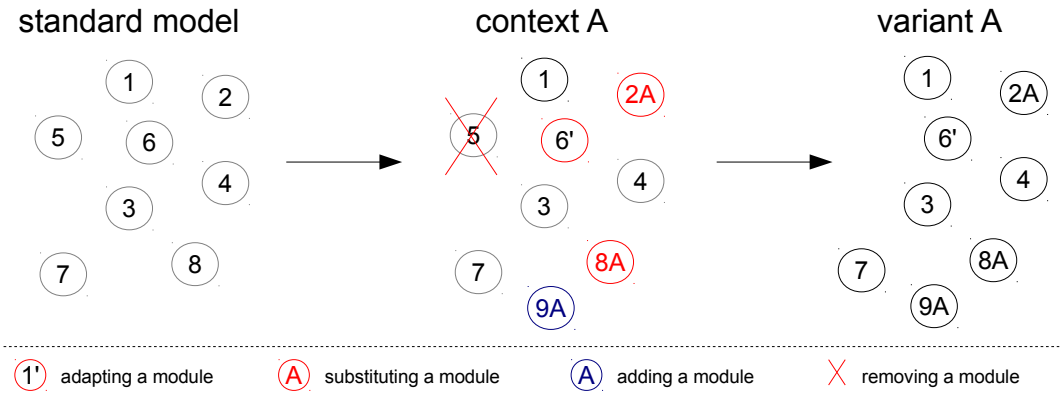


Figure 7.2. Variant Engineering of Variant A: A *variant A* is derived from a standard model.

The second example in Figure 7.3 shows another variant of the same standard model. In "*variant B*", two modules *5* and *8* are removed, *3* and *4* are adapted, *1* is substituted by *1B* and *9B* and *10B* are

added.

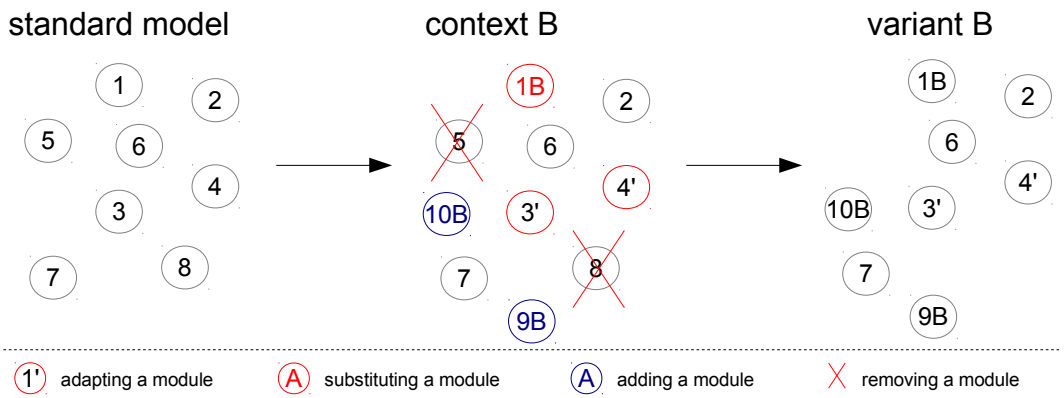


Figure 7.3. Variant Engineering of Variant B: A *variant B* is derived from a standard model.

Module Substitution

Module substitution targets to substitute modules of a model by alternative modules at model instantiation. The alternative modules are provided by a new model.

The method requires to configure *replacing includes* to resolve dependencies towards the alternative modules. The impact of replacing includes is that references are first tried to get resolved in the new model (that provides the alternative modules). Only in case a reference cannot get resolved in the new model, it is tried to get resolved in the owning model of the reference.

Figure 7.4 illustrates an example, where basic event *Be2* and *Be3* of *Model 1* are substituted. The alternative modules are provided in a *Model 1'*. The replacement requires to declare *Model 1'* as replacing include in *Model 1*. And *Model 1'* needs to include *Model 1* recursively so that parameter *P2* and *P3* of *Model 2* can be resolved (though *Model 1'* could as well directly include *Model 2*).

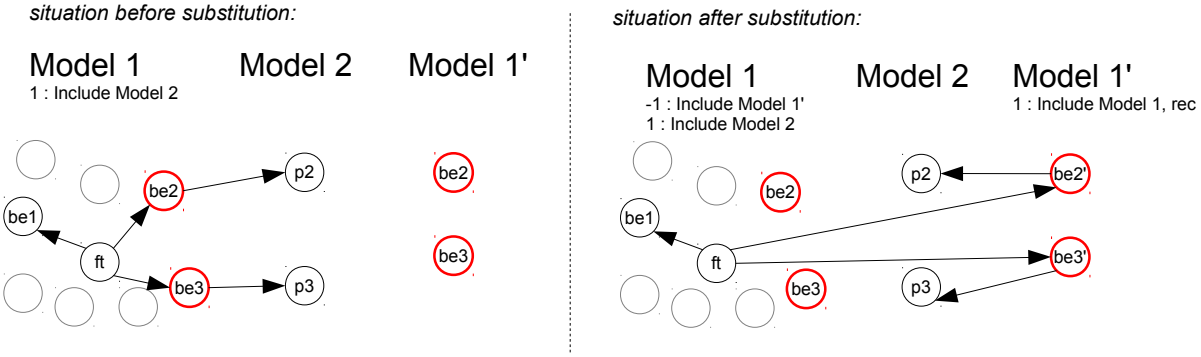


Figure 7.4. Example of model substitution: The left figure shows the situation before basic events *be2* and *be3* are substituted by alternative modules. The figure on the right shows the situation after the substitution.

A modular PSA can contain a set of models. Figure 7.5 illustrates the method of module substitution in general. For each model M the following steps must be performed to substitute modules:

1. Provide a new model M' defining alternative modules for the modules to substitute in M.

2. Declare the new model M' as replacing include in M (so that alternative modules of M' overload modules to substitute in M).
3. Declare M as recursive include in M' (so that include definitions in M can be reused in M').

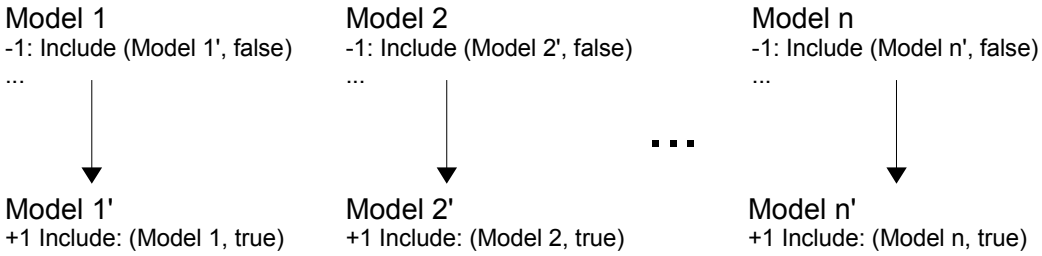


Figure 7.5. Module substitution in general: Modules are substituted by providing alternative module implementations in additional models. Replacing includes in the original models towards the additional models enable the substitution.

Property Injection

Property injection targets to adapt terms (the values) of component properties. It can also serve to deactivate components as deactivation is derived from the value of a special property (the *enabled* property).

Whereas "*module substitution*" is not affecting the content of models (it affects only the resolution of dependencies), property injection targets adapting directly the content of model elements by changing their properties. Those adaptations are however only virtually (like an overlay) and only valid as long as the variant is applied.

A modular PSA provides the construct of adaption rules (see Section 6.2.3) to flavor property values. Adaption rules change merely property terms of existing components. New components cannot be created by this method.

The impact of property injection is illustrated in Figure 7.6. Adaptation rules permit to adapt properties of modules and their subcomponents. In the example a module M with its subcomponents and properties is shown. The right side illustrates M after some of its properties have been adapted.

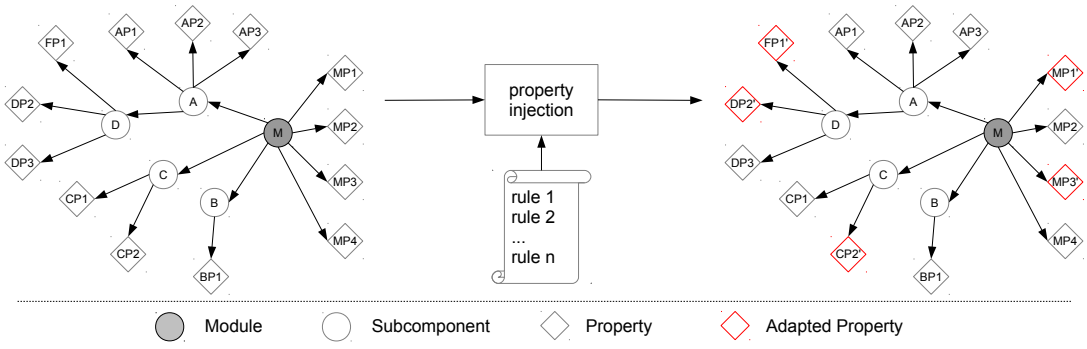


Figure 7.6. Principle of property injection: Property injection uses adaption rules to adapt property terms.

Evaluation: 1) In case modules differ "slightly" from a standard model, property injection is an adequate method for adapting modules. Adaption rules are compact and easy to manage.

2) In case modules differ "heavily" from a standard model and rules would become quite complex to adapt modules, the concept of module substitution can be the better choice.

7.1.4 Application

The benefit of variant management is best explained by examples. In the sequel, various applications are described.

Avoiding Redundancy

One (major) benefit of the variability technique in a modular PSA is to reduce modeling redundancy. Redundancy occurs when similar or identical parts of models are modeled various times. This can happen within one model (e.g. a certain gate is modeled equally in various fault trees) but also between different models (e.g. a set of basic events and parameters is modeled anew in each model). The danger of redundancy is a decreased readability, high modeling efforts (redundancy requires to apply future modifications multiple times) and the risk of modeling errors.

Example: Let "ft1" be a fault tree and "be1", "be2" and "be3" basic events. The modules 'ft1", "be1" and "be2" are identical in three models ("model A", "model B" and "model C"). But module "be3" is different (specific). The situation can be illustrated in tree form (FT = Fault Tree, BE = Basic-Event, (!) = interesting model part):

```
model A
├── Def FT ft1
│   ├── Def gate1 OR
│   │   ├── be1*
│   │   ├── be2*
│   │   └── gate2*
│   └── Def gate2 AND
│       └── be3*
├── Def BE be1
│   └── value=2.4e-7
├── Def BE be2
│   └── value=4e-8
└── Def BE be3
    └── value=1e-10 (!)
```

```
model B
├── Def FT ft1
│   ├── Def gate1 OR
│   │   ├── BE be1*
│   │   ├── BE be2*
│   │   └── gate2*
│   └── Def gate2 AND
│       └── be3*
├── Def BE be1
│   └── value=2.4e-7
├── Def BE be2
│   └── value=4e-8
└── Def BE be3
    └── value=1.5e-9 (!)
```

```
model C
├── Def FT ft1
│   ├── Def gate1 OR
│   │   ├── BE be1*
│   │   ├── BE be2*
│   │   └── gate2*
│   └── Def gate2 AND
│       └── be3*
├── Def BE be1
│   └── value=2.4e-7
├── Def BE be2
│   └── value=4e-8
└── Def BE be3
    └── value=5.6e-9  (!)
```

The three models of the example constitute a high degree of redundancy as large parts are identical and any modification is likely to be replicated to all other models as well. For example, a modification of *gate1* is probably to perform three times (in each model). Any objects except *be3* can be considered as redundant.

To avoid the redundancy problem of the example, the instantiation technique of a modular PSA can be applied to express variants. Two solutions can be considered in a modular PSA:

- Solution by adaption rules.
- Solution by includes.

Those are discussed in sequel.

Solution by Adaption Rules This solution requires to define three contexts. Each context defines the instantiation of a generic model:

```
generic model:
├── Def FT ft1
│   ├── Def gate1 OR
│   │   ├── be1*
│   │   ├── be2*
│   │   └── gate2*
│   └── Def gate2 AND
│       └── be3*
├── Def BE be1
│   └── value=2.4e-7
├── Def BE be2
│   └── value=4e-8
└── Def BE be3
    └── value=0         (!)
```

The generic model contains basic event "be3" with probability "0". The value is however not important as it will get overloaded.

To obtain "Variant A", the generic model is instantiated with the first context. This context defines the following adaption rule:
`basic-event[name==be3]:value SET 1e-10`

"Variant B" is obtained by using the second context containing adaption rule:
`basic-event[name==be3]:value SET 1.5e-9`

Finally, "Variant C" is obtained by adaption rule:
`basic-event[name==be3]:value SET 5.6e-9`

The benefit of this solution is that only one single model is developed (the generic model). All variants are obtained by instantiating this model. Another advantage of the approach is that with one single rule many adaptions can be applied (though in the example only one adaption has been made).

The disadvantage is that modifications of "be3" must be implemented as adaption rules, what may be less intuitive and what requires a good understanding (especially when models are large and several rules are needed). For example a model engineer is required to understand that any modification of "be3" in the generic model has no effect as it gets "overloaded" by the rule values. So he has to find a solution by using adaption rules to implement future modifications of specific parts (he has to think on a "meta-level").

Note that a possibility to mark "be3" specifically as a value to overload is not considered in the modular PSA approach. In general, different variants may adapt (overload) different model parts. Further, this would be against the principle to clearly distinguish content from variant management as the information about variable parts would be brought on model level (and not purely on instantiation level).

Solution by Includes This solution requires to define four models: Three models (named "be model A-C") to provide each time the respective basic event "be3". The fourth model is a commonly shared model:

```
be model A:
└── Def BE be3
    └── value=1e-10
```

```
be model B:
└── Def BE be3
    └── value=1.5e-9
```

```
be model C:
└── Def BE be3
    └── value=5.6e-9
```

```
Common model:
├── Def FT ft1
│   ├── Def gate1 OR
│   │   ├── be1*
```

```
    ├── be2*
    └── gate2*
└── Def gate2 AND
    └── be3*
├── Def BE be1
│   └── value=2.4e-7
└── Def BE be2
    └── value=4e-8
```

The shared model does not contain "be3" (though references to "be3" are declared).

Finally, the common model and the specific model "be model A" are instantiated (analog for "be model B" and "be model C"). An include configuration is required (for the common model) so that references to "be3" are resolved correctly:

```
Includes for the common model:
Position      Model       recursive
 +1:        be model A     False
```

The instantiated forms of the common model are variants. Each variant "sees" a different basic event *be3*, but the rest of the model is identical.

Note that includes are part of the instantiation context and not part of the common model. Consequently also this solution requires three contexts (one for each variant) to obtain the final modular PSAs.

The final modular PSAs can be stated in tree form (includes are indicated in angle brackets):

```
modular PSA A:
├── common model*
│   <+1, be model A, False>
│
└── be model A*
```

```
modular PSA B:
├── common model*
│   <+1, be model B, False>
│
└── be model B*
```

```
modular PSA C:
├── common model*
│   <+1, be model C, False>
│
└── be model C*
```

The advantage of this solution is that all four models (3 specific models and 1 common model) are "normal" PSA models which are easy to understand and develop by any model engineer (whereas rules may require a deeper understanding).

The disadvantage is that the solution is less elegant: Four models are required whereas the solution with adaption rules needed only 1 model. Also, it is required to introduce new modules for **each** specific adaption (in the example this is only "be3"). The solution with rules may define only a few rules to adapt many modules at once.

Using Adaption Rules

In this section, several examples illustrate the usage of adaption rules for expressing variants. The format used to specify adaption rules has been specified in Section 6.4.2.

Modifying the Value of Basic Events In the following example, it is shown how to assign for basic event *be1* a fixed probability value:

```
gate[name=be1]:model SET FIXED
gate[name=be1]:value SET 1.14e-4
```

Setting the attribute *model* to *FIXED* assigns a fixed probability distribution to the basic event.

Modifying the Initial State of an Event Tree The following example modifies the initiating event for all event trees that 1) have been created before 1998 by a 2) person named Steve and that 3) have initiating event *init1* assigned: The initiating event is adapted to *init2*.

```
event_tree[created<01.01.1998,modifiedBy=Steve,initial-state=init1]:initial-state SET init2
```

Deactivating Gates in Fault Trees The following example deactivates gate *gate1* in all fault trees:

```
gate[name=gate1]:enabled SET False
gate:input CHANGE-REF gate_reference gate1 null
```

The first rule deactivates the gate with name *gate1*.

The second rule deactivates any gate references to that gate. The *input* property of gates is assigned to a new term where reference *gate*1 is replaced by *null* (indicating the $\perp$ term).

Disabling Function Events The intention of disabling a function event is to disable any sequences in event trees, which may occur given this function event fails.

Principally, the deactivation is done the same way as for gates:

```
function_event[name=fe1]:enabled SET False
*:* CHANGE-REF function_event_reference fe1 null
```

However, to set a function event to *null* in event trees is tricky as the outcome (the semantic) is undefined for success and failure paths that have no function event assigned. To deactivate gates in fault trees is different as deactivated gates can simply be "removed".

The wanted behavior of deactivating a function event is however that the success path gets associated with a probability of "1" (100 per cent) and the failure path to a probability of "0" (as if the function event wouldn't exist).

Three solutions are possible to solve this problem:

Solution 1: Develop a new *rule instruction* named `REMOVE-EVENT` to remove a function event properly in event tree mitigations. The result of the instruction is a term that contains only the success path for $fe1$. The final rules are:

```
function_event[name=fe1]:enabled SET False
event-tree:mitigation REMOVE-EVENT fe1
*:* CHANGE-REF function_event_reference fe1 null
```

Solution 2: Set function event $fe1$ to be "never failing".

```
functional_event[name=fe1]:link CHANGE-REF basic_event_reference * BE_FALSE
```

whereas *BE_FALSE* is the name of a basic event with probability zero. The rule changes the *link* property of function event $fe1$. Though the function event is not deactivated, the event tree will ignore the failure paths (at least at quantification) as they are associated with zero probability.

Solution 3: Extend the semantics of event trees to define forks without function events as "never failing". This has been done in Andromeda. If a function event is not defined, the success path is automatically evaluated to probability one. Then, function events can simply be disabled by adaption rules as shown above.

Disabling Consequences Another interest may be the deactivation of consequences.

Example:

```
consequence[name=c1]:enabled SET False
event_tree:mitigation CHANGE-REF consequence_reference c1 null
```

The deactivation can be applied to a subset of event trees:

```
event_tree[name=/!PUMP!/,tag=FIRE]:mitigation CHANGE-REF consequence_reference c1 null
```

In this case, consequence $c1$ must not be deactivated as other event trees may still need it. Expression "$/!PUMP!/$" requires event trees to contain "$!PUMP!$" in their name to apply the rule. A *tag* is another property which serves to label components.

7.1.5 Variant Management in Andromeda

The concept of variant management has been implemented in Andromeda.

Extending a Common Model

The following screenshots show how variant management is realized by simply altering include configurations.

In Figure 7.7, a common model *fault_trees.psa* contains a fault tree module. The basic events (which are referred by the fault tree) are located in another model. This model exists twice (once for each variant) named *eventsA.psa* and *eventsB.psa* respectively. The project shown on the left side (of the screenshot) uses the fault tree with events *eventsA.psa*. The project on the right side uses the same fault tree, but *eventB.psa* as basic events.

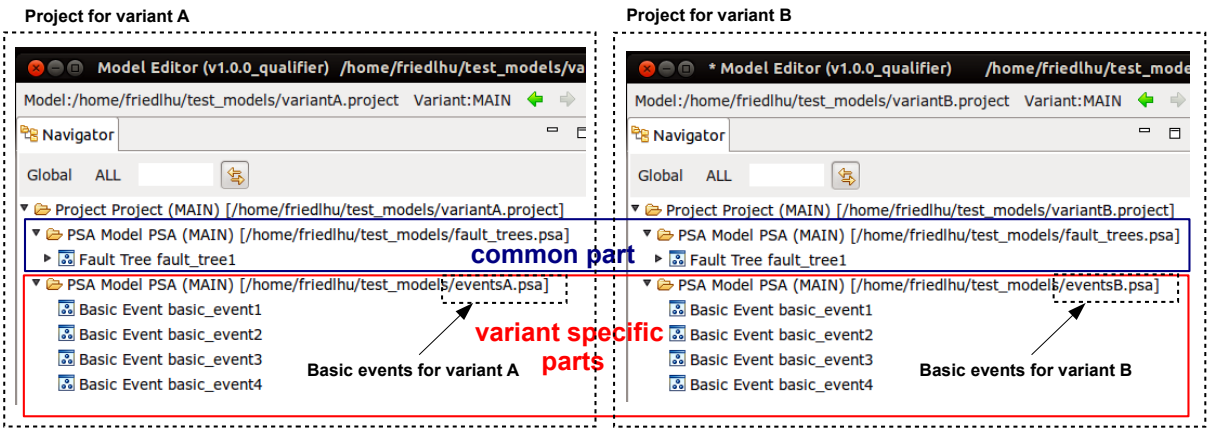


Figure 7.7. Extending a common model in Andromeda: Two projects share a common model and contain a specific model. The specific models contain basic events (the data of a PSA model).

The common model needs to include the respective specific models. The corresponding instantiated models (the variants) are shown in Figure 7.8. Whereas the fault tree structure is equal in both projects (common part), the basic events are project specific (variant specific parts).

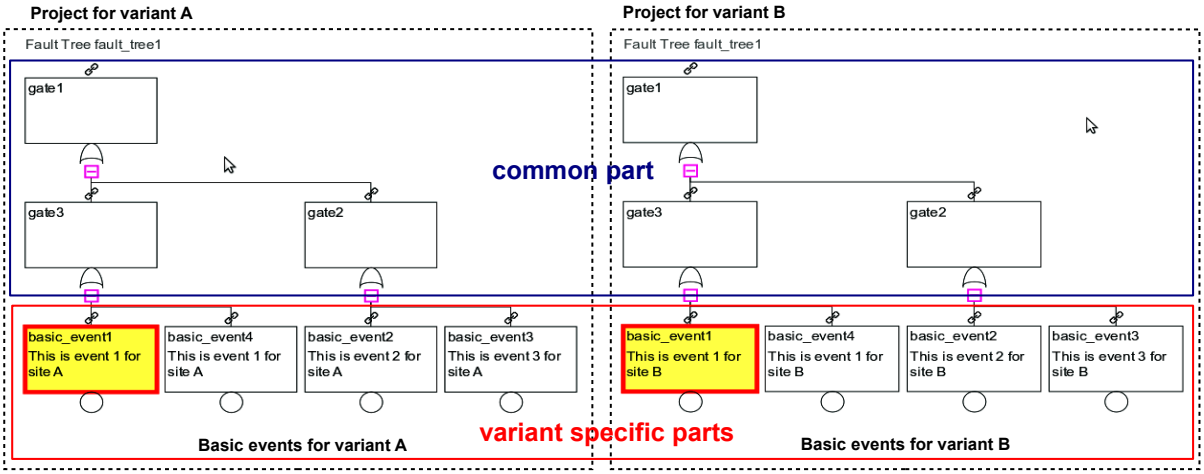


Figure 7.8. Using variant specific basic events: The "same" fault tree exists once for "variant A" and once for "variant B" with different data (basic events) assigned.

Typically, PSA models contain up to 10000 basic events. The example has been kept small in order to illustrate the concept. But the concept can be applied to large PSA models, where a large list of variant specific basic events can be prepared for each variant.

Variant Management by Property Injection

To demonstrate variant management by property injection, an example of adaption rules is introduced:

```
gate[name=gate4]:enabled set true
gate[name=gate2]:enabled set false
gate[name=gate3]:gatetype set AND
gate[name=gate1]:input CHANGE-REF gate_reference gate2 gate4
basic_event[name=basic\_event1]:value set 0.0001
```

Figure 7.9 shows the effect of property injection. On the right side, the standard model is shown (without property injection). On the left side, the variant (after property injection) is shown, i.e. after the model got adapted by adaption rules. Three differences can be observed: 1) Gate *gate2* got exchanged by *gate4*. 2) Gate type of *gate3* changed from an *OR* to an *AND* gate. 3) The value of *basic_event1* changed to 0.0001.

Comparing Variants

Andromeda has a functionality to compare variants graphically. The difference to the methods introduced in Section 7.3 is that all information are printed in one single diagram. However, this is only adequate to visualize small differences what is typically the case at variant management.

In Figure 7.10, a fault tree is shown in variant *FR* and compared against another variant *DE*. Red colored areas indicate parts that exist only in *FR* and blue areas those that exist solely in *DE*.

A graphical comparison for event trees is shown in Figure 7.11. For event trees, the concept allows to disable function events, whereas associated paths and sequences in the event trees vanish accordingly.

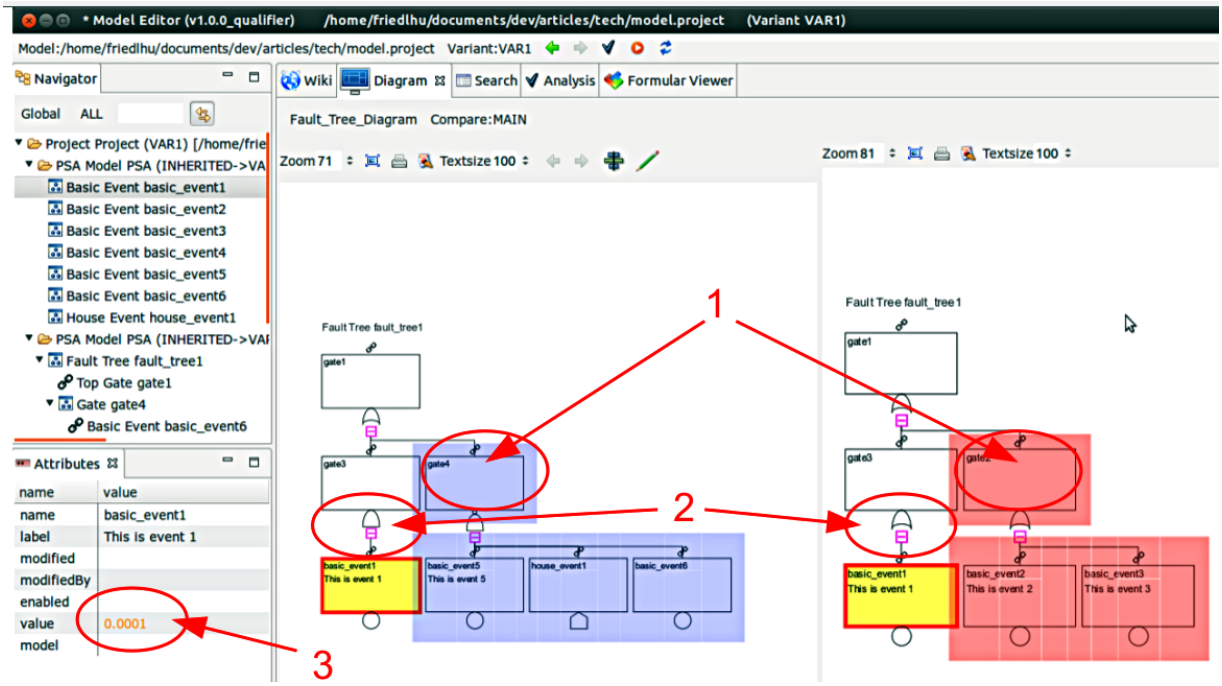


Figure 7.9. Variant management by property injection: The standard model is shown on the right side. The left side shows one of its variants obtained by applying the concept of property injection.

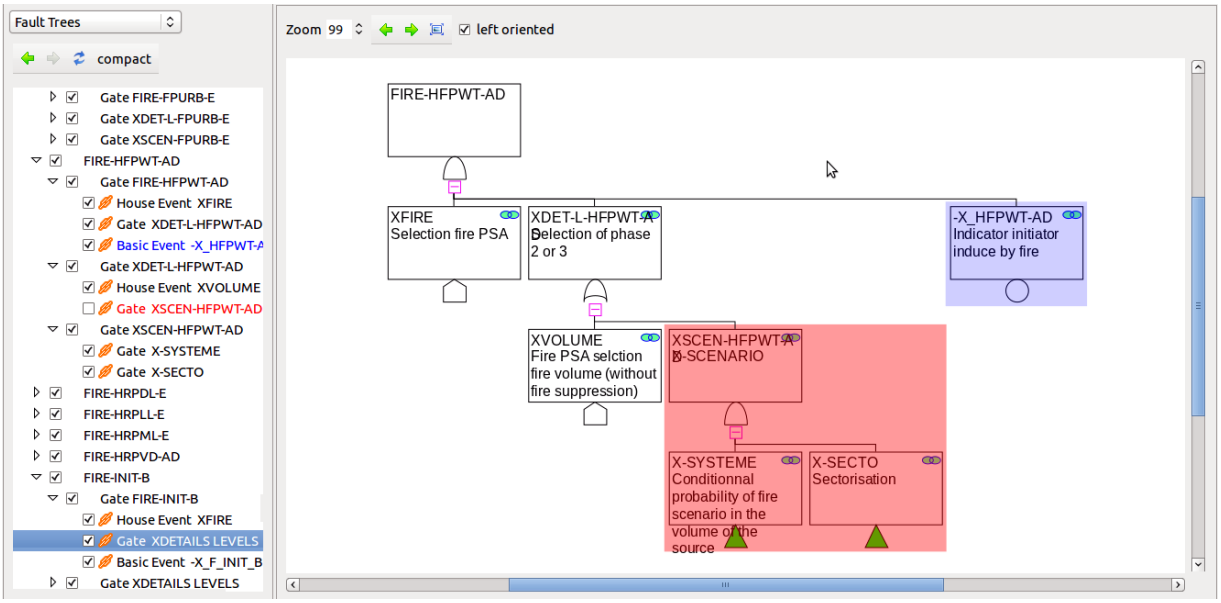


Figure 7.10. Variant Management of Fault Trees

7.1.6 Future Work

One future interest is to introduce ideas from *Feature oriented software engineering* (FOSD), which is already quite common for various software configurations [53] [52]. Examples of features oriented approaches are the Eclipse or Linux installation, where a user can configure features rather than individual elements.

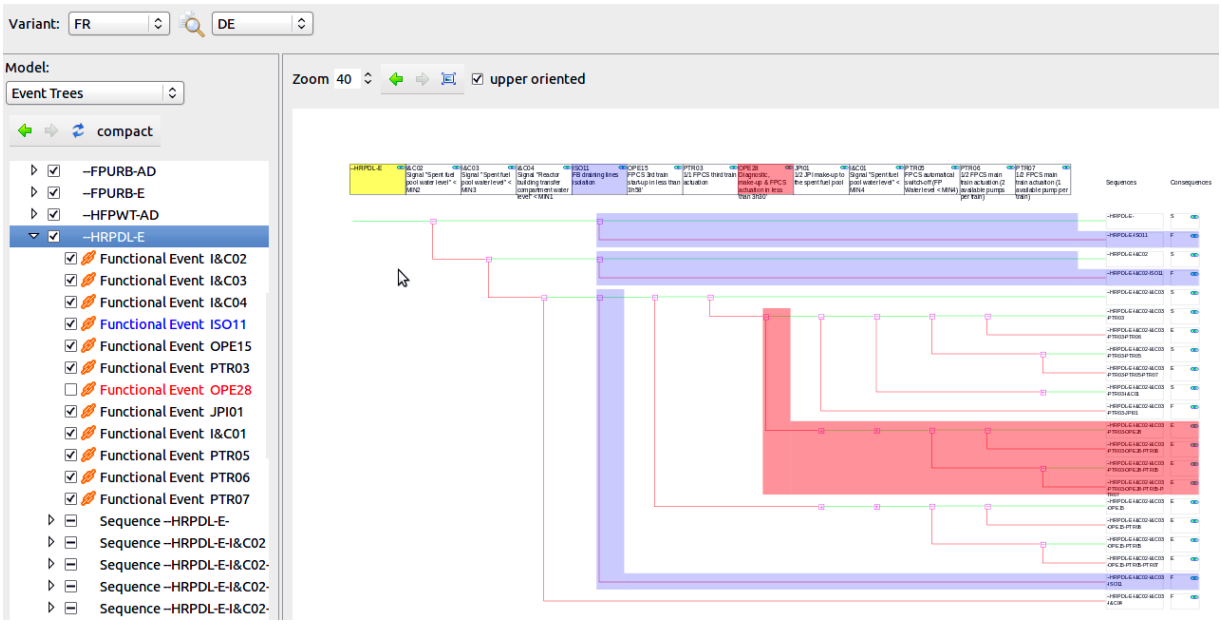


Figure 7.11. Variant Management of Event Trees

To give an example a user could have an interest to configure a variant that is only related to *FIRE* and *FLOOD*. In general, a configuration of features is a Boolean formula over constraints. In the example the formula is "FIRE and FLOOD".

In a simple form, features can be based on *tags* (short labels) that are assigned to model elements. In any case, the idea is to automatically derive (at "runtime") adequate instantiation parameter (adaption rules and includes) from a feature configuration.

A feature configuration is thus a meta-configuration for a context. It brings model configuration to a higher level which is supposed to be more compact and intuitive. Adaption rules and includes can be considered as a low level configuration that is generated from a feature configuration.

7.2 Version Management

Version management (VM) is a concept to manage so-called "versions" of models. Each version corresponds to a certain development state. It is crucial to keep track of modifications.

VM helps model engineers to complete various tasks, for example to replicate modifications of other models, to cross-check modifications between two points in time (to verify, that no modifications have been forgotten or accidentally applied) or to create modification reports. The latter can serve as official reports to justify modifications to safety control authorities.

However, in large PSA models, VM is a challenging task: Modifications are typically recorded manually in text files. Those protocols are time consuming to create and maintain. Also, they are subject to failures as tools are missing to efficiently verify the listed modifications. Current PSA tools do not provide the full functionality of VM systems. In other domains, for example at software engineering, VM has become a core concept.

In this section, a version management for a modular PSA is presented. In principal, existing VM technology is applied on the modules of a modular PSA.

7.2.1 The Idea of Version Control

Version management (VM) intends to manage the evolution of developments. At model engineering, those developments refer to models. So-called "versions" are used to indicate different development states. Versions are like "backups" (that can be restored) to preserve development states.

In detail, VM at model engineering can provide the following functionality:

Model comparison A possibility to compare two models and to determine the differences between them. A method of model comparison for a modular PSA is presented in Section 7.3.

Model fusion Model fusion targets to "merge" (to combine) the contents of two (or more) models. A method of model fusion for a modular PSA is presented in Section 7.4.

Restore ancient model versions: Sometimes, modifications applied to a model should be "taken back". To restore a model version is the method to recover a certain development state of a model (or parts of it).

Documentation of versions The different versions of models can be described in form of a documentation. The documentation contains typically the applied modifications since the last model version. It can also contain timestamps and information about the model engineer who applied the modifications. This kind of documentation is not to confuse with model documentation which primarily documents the content of models (and not its evolution).

7.2.2 Principle of VCS Systems

Version control systems (VCS or VCS systems) (also known as "revision control system") are software tools that help to maintain the evolution of so-called "configurable items" (CIs). A configurable item is an individual item at version control (typically a file e.g. a text file, a model file, an image etc.). Examples of VCS systems are GIT [54, 55, 56], CVS (Concurrent Version Control) [57] or SVN (Apache Subversion) [58].

At software engineering, VCS systems have become popular to support the challenges of software engineering. Once again, the analogy between model and software engineering is close. VCS systems can indeed be considered for model engineering, at least in case some prerequisites can be ensured.

Figure 7.12 illustrates a development supported by a VCS system.

Though different VCS systems may differ in their detailed concepts and notions, the following notions are common to the most popular ones:

CI: A configurable item (CI) is an individual item considered for version control.

Versions: Versions express different development states (different points in time of a development) of CIs.

Configuration: A configuration is a coherent set of CIs. It defines the set of CIs in their appropriate versions.

Tags: Tags are kind of short labels assigned to versions to associate them to a certain development state.

Branches (development lines): A branch or development line represents a development that is ongoing in parallel to other developments.

Main line: The main line represents the principle development. It is often used to create official

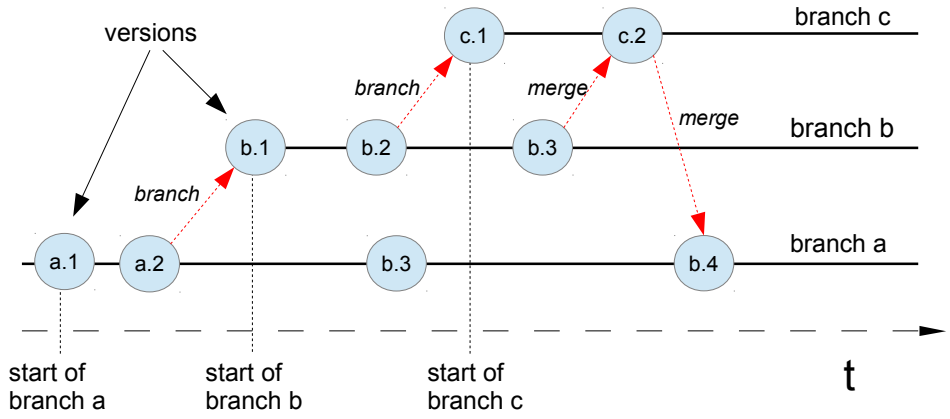


Figure 7.12. Principle of version control systems: Versions are created along branches. Each version expresses another development state. Branches represent parallel development activities.

releases of a development, while other lines than the main line are used by developers to provide their implementations.

Merging: Merging or fusion refers to the combination of two or more development lines. The result leads to a new CI version and can be stored in a new line or in one of the lines to merge.

Head version: The head version indicates the most recent version of a development line.

Checkout: Checkout is a VCS functionality and refers to recover a certain development version. The version can be more recent or ancient than the version one is currently working on.

Current version The current version indicates the version, one is currently working on. After any checkout, the current version becomes the checkout version.

Commit : Commit is a VCS functionality and refers to the creation of a new CI version. Typically, a comment can be stated when committing. Those comments serve as log information (though not to mistake with tags) to describe versions. Any commit updates the head version (to the new version created for the commit).

Branching : Branching is a VCS functionality that refers to the creation of a new development line. The name of the new branch and a version (to indicate the "start" of the branch) is required. The version is not required to be a head version, i.e. one can branch from ancient versions.

7.2.3 Managing PSA Models by VCS

PSA models can be supported by VCS systems. In order to exploit the features of VCS systems at most, it is recommended to save models in a readable form (e.g. a text format). VCS Features such as CI comparison (to compare the contents of two CIs) do typically require readable formats.

Additionally, a PSA model should be represented by various files (CIs). The idea is that each CI represents another model part (such as a fault tree, event tree etc.). Those parts can be managed individually with their private version histories etc.

The modular approach conforms to the mentioned characteristics: Each module can be stored in its own text file. In Andromeda for example, they can be saved as XML files (though by "default" the whole

model is saved in one XML file).

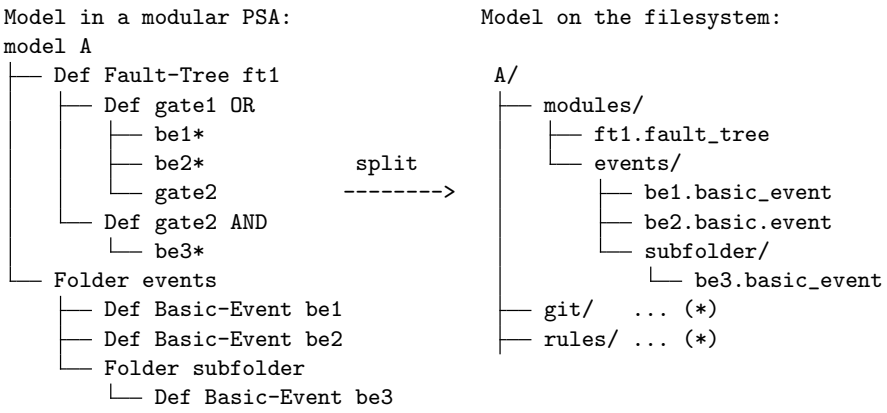
Finally, the configurable items (the assets to consider for version control) in a modular PSA are:

- **Modules:** Each module (fault tree, event tree etc.) is a configurable item.
- **Folders:** Each folder is a configurable item. Folders are used by models to organize modules (see Section 6.2.2).
- **Contexts:** Each context is a configurable item. Contexts define instantiation parameter (see Section 6.2.3).

Splitting a Model on the file system

By default, a whole model of a modular PSA is saved in one file. But VCS systems are typically based on files, i.e. each CI must be represented as an individual file. For this purpose a method has been specified to "split" PSA models, whereas each module is saved in its individual file.

The splitting method is illustrated by an example ("Folder" indicates that modules are stored in that folder):



(*) A model may store further information e.g. adaption rules, VCS data ("git" directory for GIT systems) etc.

The following rules are applied to "split" a PSA model (or any other kind of model):

- Models: Each model becomes a directory (the model directory). The directory name is the same as the model name.
- Folders: Each folder becomes a directory. Let $dir(F)$ be a function to return the directory created for a folder F.
  The root folder $F_\perp$ becomes a directory named "modules" that is stored in the model directory.
  Each subfolder SF of a folder F becomes a directory with name $name(SF)$ stored in $dir(F)$.
- Module: Each module is represented as an individual file. It is stored in the corresponding directory of its folder. To ensure unique file names, the file name of each module conforms to ``<name>.<type>'', whereas `name` is the module's name and `type` it's element type.

By different implementation of the splitting procedure, a different granularity of CIs can be achieved. In a simple case, a whole model is stored in one file. Consequently, only one (but large) CI is managed. To split a model as described on module level can create plenty of CIs. This is an advantage because:

- Individual modules can be restored.

- VCS systems such as GIT save disk space when model modifications overwrite only a few number of CIs but the majority of CIs remains unchanged. If the whole model is one CI, then any small model modification requires to rewrite the whole model on disk. However, there are VCS systems that save (incrementally) only differences to previous CI versions ("deltas") rather than the whole CI. In this case, this argument is not valid.
- VCS software features (CI comparison) can be applied on module level.
- Native OS (Operating System) tools to navigate and browse models can be used if every module is saved in its file below directories (e.g. file explorer or Linux Shell).

VCS Operations

Different VCS operations are required to manage CIs of a modular PSA.

Checkout In order to checkout a CI version, a model engineers selects either a branch or the CI version. If he selects a branch, the head version of the branch is taken as CI version. The checkout updates the CI in the respective version.

Figure 7.13 illustrates an example. Before the checkout, a version "a.4" is the current version. After the checkout of version "b.2" the current version moves accordingly to "b.2".

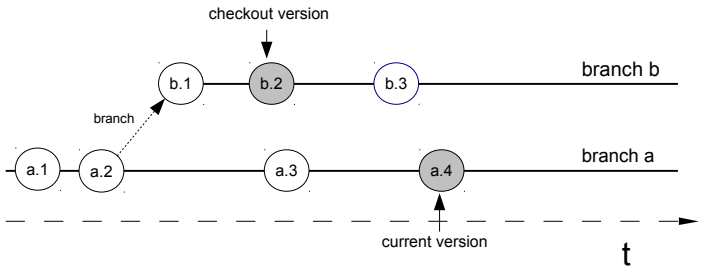


Figure 7.13. Model Checkout

If the current CI has been modified and not yet committed, the model engineer is required to commit (or reject) the CI modifications before he can perform the checkout. Otherwise the current modifications would get lost.

Commit In case the currently checked out version is the head version, a commit creates a new version. After commit, the new version is the head version (see Figure 7.14).

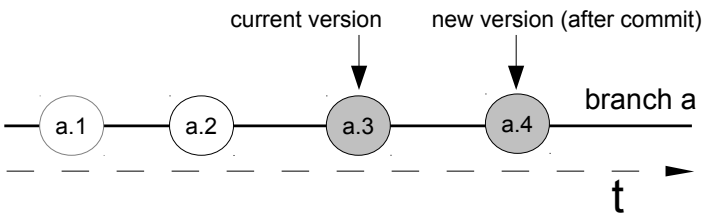


Figure 7.14. Committing a model as new head version

In case the currently checkout version is not the head version, a new branch is created to save the new CI version (see Figure 7.15).

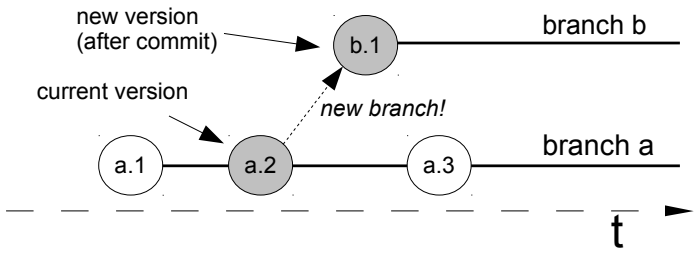


Figure 7.15. Committing a model in a new branch

Merge The merge is not considered to be performed by VCS systems because CIs are likely to become inconsistent after merge. The reason is, that VCS systems are not aware of the underlying model schemes. VCS systems merge CIs as normal text files and the merge result is not necessarily conform to the underlying model format, i.e. the domain.

Models are rather to merge by software tools which are based on the modular PSA approach. In Section 7.4, a concept to fusion models is presented.

Configuration Management

Version management in a modular PSA is based on configurations. A configuration is a **coherent** set of CIs.

In the sequel the configuration of an individual model (model configuration) and the configuration of a modular PSA (meta-configuration) are presented.

Model Configuration A *model-configuration* is a coherent set of modules.

Figure 7.16 illustrates an example with four CIs (configurable items) of a modular PSA. Each module is a CI and has its individual version and log history.

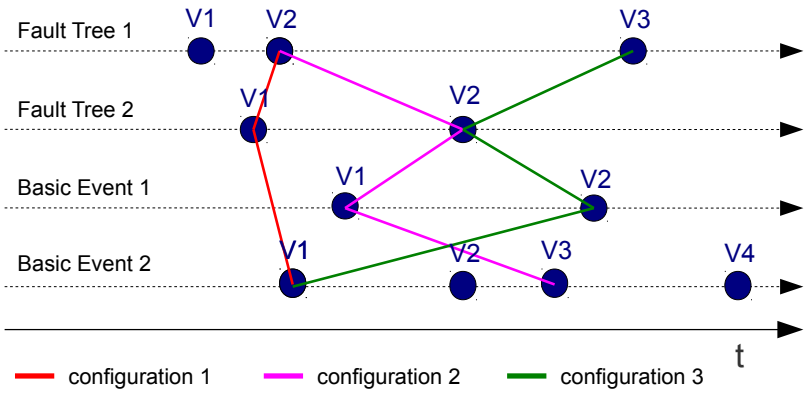


Figure 7.16. Model Configuration: Modules (two fault trees and two basic events) are configurable items (CI). A configuration is created by assembling CIs in their appropriate versions.

Ancient CI versions can be restored. In the example, this is the case for *configuration 3* which restores *Basic Event 2* in version *V1* (assuming *configuration 3* has been created after *configuration 2*).

However, to restore ancient CI versions must be done carefully as CIs have probably dependencies between each other and models can easily get inconsistent due to “unresolved references”.

Meta Configuration A modular PSA contains generally a set of models. A second configuration is needed to configure model configurations. A *meta-configuration* is a coherent set of model configurations.

Figure 7.17 shows an example of a meta-configuration. The CIs of the meta-configuration are model configurations.

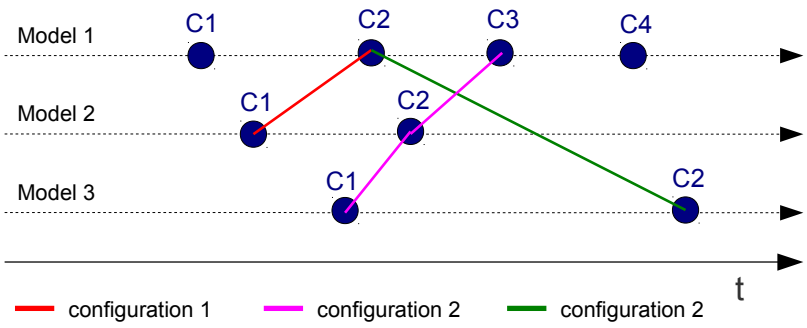


Figure 7.17. Meta-Configuration of a modular PSA: Each CI is a model configuration. A meta-configuration is a coherent set of model configurations.

Variant Management by Version Control

Version control can also serve to implement variants. Figure 7.18 shows the principle: Versions express chronological developments and variants parallel developments. In VCS systems, in particular “branches” are adequate to express variants.

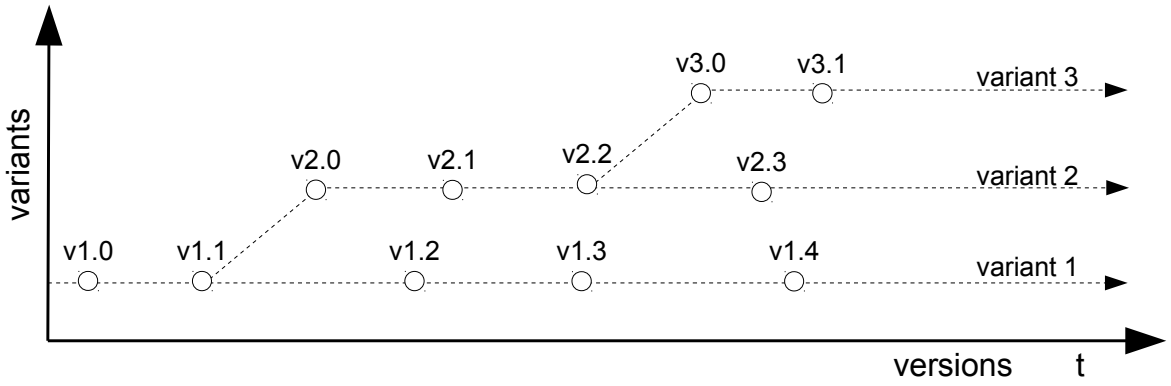


Figure 7.18. Version control to express variants.

However, the disadvantage of using VCS systems is that variant management becomes “hidden” for most model engineers that are not familiar with VCS systems. It is applied **before** models get loaded.

On the contrary, variants in a modular PSA have been defined as instantiated models specific to a context. The benefit is that the control to define variants is given to users when specifying an instantiation context. Variants can be easily debugged by analysing adaption rules and model includes. Further, rule based approaches such as adaption rules constitute a compact and reliable possibility for expressing variants (most VCS system do not support rules).

Conclusion: To express "simple" variant specific adaptions, VCS system can be used. But to specify more complex adaptions, variants in a modular PSA should be expressed on a higher level (by the means of instantiation contexts, in particular adaption rules).

7.3 Model Comparison

Model comparison targets to determine differences between models. To be aware about model differences is a crucial preliminary step for several tasks:

- To verify / analyse / cross-check model modifications: The differences between models give an important feedback "what has been done" or "what has been modified" since an earlier point in time (e.g. a previous model version).
- To fusion models (see Section 7.4): Model fusion consists in merging the differences between models.
- To automatically generate modification reports (e.g. "logbooks"): Modifications of PSA models, for example, may be to justify to control authorities. Generating them automatically is efficient and it guaranties correctness.

In this section, a method to compare models in a modular PSA is precised.

7.3.1 Concept

In general, the comparison method compares two sets of modules S_A and S_B.

The modules of each set are not required to source from the same model. Model comparison in a modular PSA is thus a generic method in two senses: It is neither domain specific (e.g. to the domain of PSA models) nor limited to compare two models:
In order to compare two models M_A and M_B, S_A contains the modules of M_A and S_B the modules of M_B. To compare two individual modules E_A and E_B, $S_A = \{E_A\}$ and $S_B = \{E_B\}$.

Matches

In principal, the comparison consists in finding *matches* between two module sets S_A and S_B. Each match is a module pair (E_A, E_B), whereas $E_A \in S_A \cup \{\bot\}$ and $E_B \in S_B \cup \{\bot\}$ indicating that E_A "corresponds" to E_B. The element $\bot$ indicates a module that could not be matched.

The overall set of matches between sets S_A and S_B is called a *map*. The following conditions must hold for a map Map:

1 ***Injectivity of A:*** $(A_1, B) \in Map \wedge (A_2, B) \in Map \Rightarrow (A_1 = A_2) \vee (B = \bot)$

2 ***Injectivity of B:*** $(A, B_1) \in Map \wedge (A, B_2) \in Map \Rightarrow (B_1 = B_2) \vee (A = \bot)$

3 ***Completeness of A:*** $\forall A \in S_A \exists B \in S_B \cup \{\bot\} : (A, B) \in Map$

4 ***Completeness of B:*** $\forall B \in S_B \exists A \in S_A \cup \{\bot\} : (A, B) \in Map$

5 $(\bot, \bot) \notin Map$

The constraints ensure that modules of one set can be uniquely matched to a module of the other set (or to $\bot$).

Match Type

Matched modules are not necessarily identical. The so-called *match type* gives feedback about the kind of match.

Each match (E_A, E_B) is assigned to a match type:

- EQUAL: E_A matches "exactly" E_B.
- MINOR: E_A matches E_B with "minor" differences. Slight differences represent those with minor impact e.g. label differences (a label is a "short" component description). Those differences do typically not impact risk quantification.
- MAJOR: E_A matches E_B with "major" differences. Major differences exhibit a major impact, e.g. the frequency of basic events. Those differences do typically impact risk quantification.
- A ONLY: The module E_A could not be matched to a module of S_B: $E_B = \perp$.
- B ONLY: The module E_B could not be matched to a module of S_A: $E_A = \perp$.

Equality

In the sequel, the equivalence of components is defined. Equivalence is required at model comparison to find matches (of any match type, not only for match type $EQUAL$).

Structural Equivalence Two components C_1 and C_2 are said *structurally equivalent* in a context τ if $C_1 \equiv C_2$.

$C_1 \equiv C_2 :=$

- $\forall PT \in propertyTypes : term(C_1, PT, \tau) = term(C_2, PT, \tau))$
- $\forall D_1 \in definitions(C_1) \exists D_2 \in definitions(C_2) : D_1 \equiv D_2$
- $\forall D_2 \in definitions(C_2) \exists D_1 \in definitions(C_1) : D_1 \equiv D_2$

In words: Two components are structurally equivalent if their properties and their defined components are equal.

In the remainder of this section, equivalence refers to structural equivalence.

Two components C_1 and C_2 are said *recursively structurally equivalent* respective a context τ if

- $C_1 \equiv C_2$ and
- the sets of dependent modules of C_1 and C_2 are *recursively structurally equivalent.*

In words: Two components are *recursively structurally equivalent* if their properties, their defined components and their dependent modules are equal.

Semantical Equivalence Two components C_1 and C_2 are said *semantically equivalent* in a context τ if $C_1 \cong C_2$.

$C_1 \cong C_2 :=$

- $\forall PT \in PropertyTypes$: $term(C_1, PT, \tau) \cong term(C_2, PT, \tau)$
- $\forall D_1 \in definitions(C_1) \exists D_2 \in definitions(C_2) : D_1 \cong D_2$
- $\forall D_2 \in definitions(C_2) \exists D_1 \in definitions(C_1) : D_1 \cong D_2$

$P_1 \cong P_2$ of two properties P_1 and P_2 expresses that their terms are semantically equal (though the term structure is possibly different). For example the term $T_1 = or(and(b1, b3), and(b1, b4), and(b2, b3), and(b2, b4))$

is structural different from the term $T_2 = and(or(b1, b2), or(b3, b4))$ but semantically they are equal: $T_1 \neq T_2$ but $T_1 \cong T_2$.

Note that semantic equivalence is **domain specific** whereas structural equivalence is **generic** and can be derived from the structure of models.

In words: Two components are semantically equivalent if their properties and their defined components express the same semantics.

Two components C_1 and C_2 are said *recursively semantically equivalent* in a context τ if

- $C_1 \cong C_2$ and
- the sets of dependent modules of C_1 and C_2 respective τ are *recursively semantically equivalent.*

In words: Two components are *recursively semantically equivalent* if their properties, their defined components and their dependent modules express the same semantics.

Reduction

Reduction is a function to "reduce" a component C to a component C' which contains "less" content. C' is kind of subset of C. Reduction is required to find module matches in case components cannot be fully (identically) matched.

A component C_2 is said a "reduced form" of C_1 (respective a context τ) if $C_1 \leq C_2$.

$C_1 \leq C_2 :=$

- $\forall PT \in propertyTypes : (term(C_1, PT, \tau) = term(C_2, PT, \tau)) \vee (term(C_2, PT, \tau) = \bot)$
- $\forall D_2 \in definitions(C_2) \exists D_1 \in definitions(C_1) : D_1 \leq D_2$

Figure 7.19 shows an example of a reduction. A module M is displayed once in its original form (left side) and once in a reduced form (right side). M in the reduced form contains a subset of properties and components.

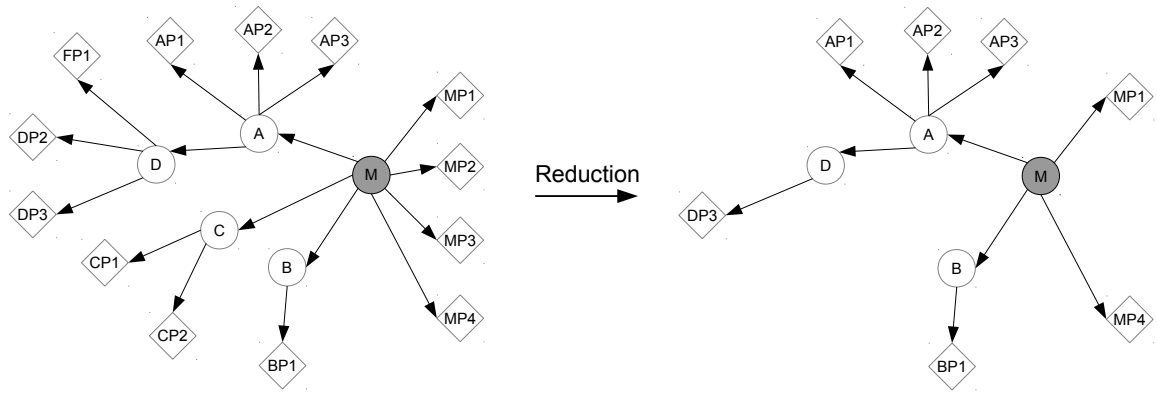


Figure 7.19. Module Reduction: A module M gets reduced from its original form (left side) to a reduced form (right side).

Match Determination

Matches are determined by comparing modules in reduced forms.

The principle of reduction is explained by Figure 7.20: Three reductions $R1$, $R2$ and $R3$ are introduced in order to obtain successively *reduced* forms of a module M:

- $R1$: Reduces modules to a form which serves as decision base to conclude *equality* between modules.
  If a match (M_A, M_B) is of type $EQUAL$ then $R1(M_A) \equiv R1(M_B)$.
- $R2$: Reduces modules to a form which serves as decision base to conclude *minor differences* between modules.
  If a match (M_A, M_B) is of type $MINOR$ then $R2(M_A) \equiv R2(M_B)$.
- $R3$: Reduces modules to a form which serves as decision base to conclude *major differences* between modules.
  If a match (M_A, M_B) is of type $MAJOR$ then $R3(M_A) \equiv R3(M_B)$.
  If $R3(M_A) \not\equiv R3(M_B)$ then (M_A, M_B) is not a match.

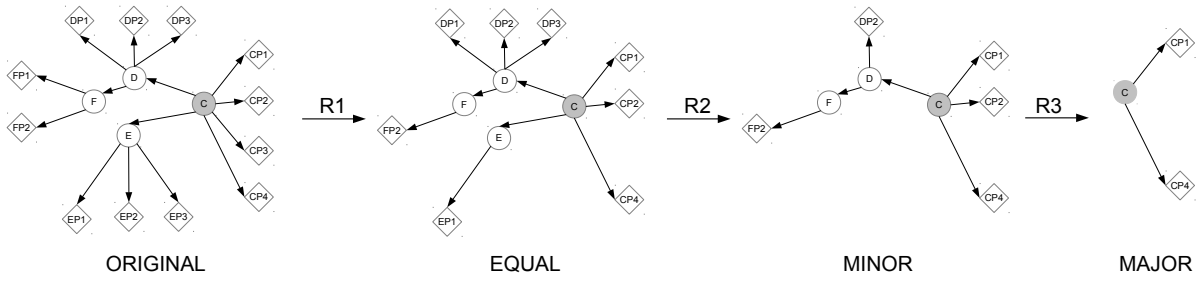


Figure 7.20. Principle of reduction functions: Three reduction functions (R1, R2 and R3) reduce a module M successively.

In order to successively reduce modules, the following constraint must be satisfied:

$$\forall C \in Components : R1(C) \leq R2(C) \leq R3(C)$$

Let S_A and S_B be two sets of modules to compare (of possibly different models). The match determination requires four phases:

Phase 1 Determination of matches of type $EQUAL$. The result is stored in Map_EQUAL:

The creation of equal matches is described by the following pseudo code (though faster algorithms exist but they are less intuitive):

```
1  Map_EQUAL := {}
2  forall (M_A in S_A)
3  do
4       forall (M_B in S_B)
5       do
6            if (R1(M_A) ≡ R1(M_B)) then
7                 Map_EQUAL := Map_EQUAL ∪ { (M_A, M_B) }
8                 S_A := S_A \ {M_A}
9                 S_B := S_B \ {M_B}
10           fi
11      od
12 od
```

The algorithm iterates over all possible pairs of modules and tries to match them in their reduced forms. If they are equal, a match of type $EQUAL$ is found and the modules are removed from S_A and S_B.

Phase 2 Determination of matches of type *MINOR*. The result is stored in *Map_MINOR*: The phase starts as soon as phase 1 has completed. The algorithm is analog to the one of phase 1 except that *Map_EQUAL* becomes *Map_MINOR* and the match condition of line 6 is

`if` $(R2(M_A) \equiv R2(M_B))$ `then`

Phase 3 Determination of matches of type *MAJOR*. The result is stored in *Map_MAJOR*: The phase starts as soon as phase 2 has completed. The algorithm is analog to the one of phase 1 except that *Map_EQUAL* becomes *Map_MAJOR* and the match condition of line 6 is

`if` $(R3(M_A) \equiv R3(M_B))$ `then`

Phase 4 : Determination of matches of type *A ONLY* and *B ONLY*. The result is stored in *Map_A_ONLY* *Map_B_ONLY*. Those are constructed of the remaining modules in S_A and S_B that could not be matched:
$Map\_A\_ONLY = \{M \mid M \in S_A\}$
$Map\_B\_ONLY = \{M \mid M \in S_B\}$

Example The match determination is illustrated by an example. Let S_A and S_B be two sets of modules to compare.

Figure 7.21 represents the first phase. The circles on the left side represent the modules of S_A. The circles on the right side the modules of S_B. The value of each circle is characteristic for its content (for example so-called "hash values" can be determined in computer science to get a characteristic value for objects). Let γ be the function to return the characteristic value of a module and M_A, M_B be two modules. Then $\gamma(M_A) = \gamma(M_B) \Leftrightarrow M_A \equiv M_B$.

Matched modules M_A, M_B of type *EQUAL* require $\gamma(R1(M_A)) = \gamma(R1(M_B))$.

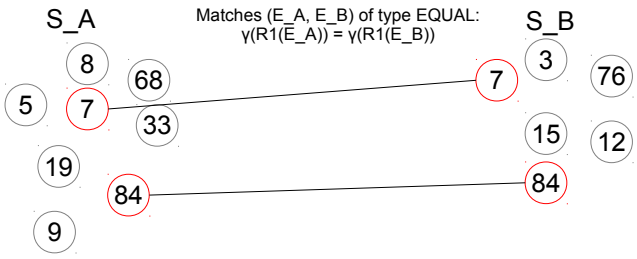


Figure 7.21. Phase 1: Matches of type *EQUAL*.

The second phase is shown in Figure 7.22. Matched modules while the first phase are no longer present. The characteristic values (the circle values) have (generally) changed because the modules have been further reduced for phase 2. Matched modules M_A, M_B of type *MINOR* require $\gamma(R2(M_A)) = \gamma(R2(M_B))$.

In a third phase, matches of type *MAJOR* are determined (see Figure 7.23). Matched modules M_A, M_B of type *MAJOR* require $\gamma(R3(M_A)) = \gamma(R3(M_B))$.

Finally, the remaining modules (after phase 3) are the modules that could not be matched in the previous phases. In the last phase, remaining modules of S_A are associated with type *A_ONLY* and those of S_B to *B_ONLY*. This is shown in Figure 7.24.

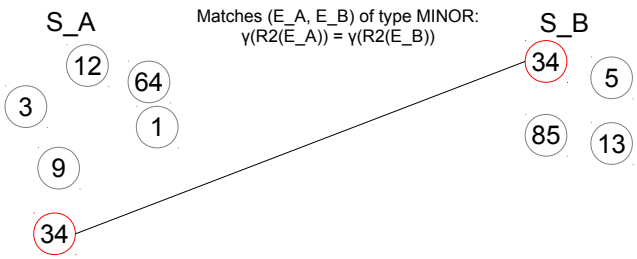


Figure 7.22. Phase 2: Matches of type *MINOR*.

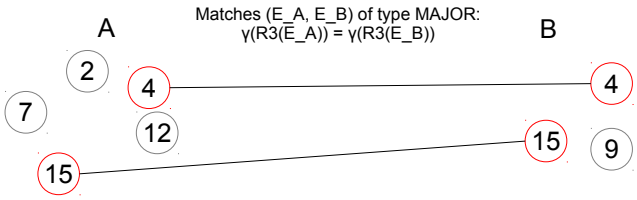


Figure 7.23. Phase 3: Matches of type *MAJOR*.

Figure 7.24. Phase 4: Unmatched Modules.

7.3.2 Configuration

The reduction functions $R1, R2$ and $R3$ depend on a configuration. A configuration assigns *importance* values to properties and components.

Importance

An *importance* is one of the values:

- *NONE*: A property / component has no importance for the comparison.
- *MINOR*: A property / component has minor importance for the comparison.
- *MAJOR*: A property / component has major importance for the comparison.
- *REQUIRED*: A property / component is required to match.

Function R_1 reduces a component by removing any properties / components of importance *NONE*.

Function R_2 reduces a component by removing any properties / components of importance *NONE* and *MINOR*.

Function R_3 reduces a component by removing any properties / components of importance *NONE*, *MINOR* and *MAJOR* (only *REQIURED* content remains).

Note that if a component is removed all its sub components and properties are removed as well no matter their importance configurations.

Rules

To assign importance values (in Andromeda), a specific file format has been developed to store *importance rules* (not to confuse with adaption rules).

Let C be a component and SC a subcomponent of C.

Two type of rules are provided. The first rule assigns an importance `IMPORTANCE` to SC if `type(C) = COMPONENT-TYPE` and `type(SC) = SUBCOMPONENT-TYPE`.

```
<COMPONENT-TYPE>/<SUB-COMPONENT-TYPE> <IMPORTANCE>
```

Further let P be a property of C. The second rule assigns an importance to `IMPORTANCE` to P if `type(C) = COMPONENT-TYPE` and `type(P) = PROPERTY-TYPE`.

```
<COMPONENT-TYPE>:<PROPERTY-TYPE> <IMPORTANCE>
```

Component and property types can be denoted as "`*`" symbol that serves as placeholder for any type.

The specified rules are applied sequentially. The first rule that can be applied for a component or property, is taken. If no rule can be applied, default rules are applied:

```
*/* MAJOR            (1*)
*:name REQUIRED      (2*)
*:* MINOR            (3*)
```

The default rules assign components to a *MAJOR* importance (1*). The *name* property is *REQUIRED* to match otherwise two components can never match (2*). Any other properties than the *name* property are of *MINOR* importance (3*).

The default rules require components to be equally named in order to match. Thus, two fault trees "ft1" and "ft2" can never match (with the default rules).

By overloading the default rules (by rules that are applicable before the default rules), components can match even if they are named differently. For example a rule `''*:name NONE''` is neglecting the name criteria of (2*). This permits to recognize "same" objects even in case they have been renamed between models. At EDF, for example, this is necessary as fault trees are generated from system models and their names may change after each generation.

Example

Let RS be a set of rules:

```
Fault-Tree:modified MAJOR              (1)
Fault-Tree/Gate MINOR                  (2)
Gate:gatetype NONE                     (3)
Gate:inputs MAJOR                      (4)
```

The default rules are not stated. They are implicitly applied if the last rule could not be applied (rule 4 in the example).

As an example, RS gets applied on a fault tree ft:

```
model A                          Importance:     Applied rule:
```

```
Def Fault-Tree ft                    -
├── name=ft                          REQUIRED        default rule
├── modified=2012                    MAJOR           rule 1
├── Def gate1                        MINOR           rule 2
│   ├── gatetype='OR'                NONE            rule 3
│   └── input=[be1*,be2*,gate2*]     MAJOR           rule 4
└── Def gate2                        MINOR           rule 2
    ├── gatetype='AND'               NONE            rule 3
    └── input=[be3*]                 MAJOR           rule 4
```

The result of $R1(ft)$ is:

```
Def Fault-Tree ft                          Importance:
├── name=ft                                REQUIRED
├── modified=2012                          MAJOR
├── Def gate1                              MINOR
│   └── input=[be1*,be2*,gate2*]           MAJOR
└── Def gate2                              MINOR
    └── input=[be3*]                       MAJOR
```

The result of $R2(ft)$ is:

```
Def Fault-Tree ft                          Importance:
├── name=ft                                REQUIRED
└── modified=2012                          MAJOR
```

The result of $R3(ft)$ is:

```
Def Fault-Tree ft                          Importance:
└── name=ft                                REQUIRED
```

In the sequel the fault tree ft (left side) is compared against another fault tree $ft2$ (right side). The differences are indicated by the symbol "!":

```
Def Fault-Tree ft                    Def Fault-Tree ft2
├── name=ft                          ├── name=ft
├── modified=2012                    ├── modified=2012
├── Def gate1                        ├── Def gate1
│   ├── gatetype='OR'                │   ├── gatetype='AND'        (!)
│   └── input = [be1*,be2*,gate2*]   │   └── input = [be1*,be2*]   (!)
└── Def gate2                        └── Def gate4
    ├── gatetype='AND'                   ├── gatetype='OR'         (!)
    └── input=[be3*]                     └── input=[be78*, he4]    (!)
```

The resulting match type is *MINOR* because $R1(ft) \neq R1(ft2)$ but $R2(ft) = R2(ft2)$. If property *modified* would change in $ft2$ then the match type becomes *MAJOR* (as $R2(ft) \neq R2(ft2)$ but $R3(ft) = R3(ft2)$).

Application

Rules to configure model comparison (not to confuse with adaption rules) can be used to detect the set of modified modules.

Rules to filter Modified Basic Events Rules can serve to filter (to find exclusively) modified basic events:

```
Basic-Event:name  REQUIRED          (1)
Basic-Event:value MAJOR             (2)

*/* IGNORE                         (3)
*:* IGNORE                         (4)
```

The first rule requires identical names for matching basic events. The rule is normally "default" but it must be stated as rule 4 overloads the default rules (in the example the default rules are never applied as 3) and 4) are always applicable).
Rules 2, 3 and 4 ensure that only the *value* property of basic events is taken into account to determine differences. Any other differences are ignored.

A comparison with the rules yields matches, whereas any match (E_A, E_B) of type $MAJOR$ represents a modified basic event with:

- $name(E_A) = name(E_B)$
- $term(E_A, \text{“}value''\text{”}, \tau) \neq term(E_B, \text{“}value''\text{”}, \tau)$

Rules to find Renamed Fault Trees Components may be renamed. Rules can serve to detect renamed components due to their similar structure:

```
Fault-Tree:name IGNORE          (1)
Fault-Tree/GATE REQUIRE         (2)
GATE:name REQUIRE               (3)

*/* IGNORE                      (4)
*:* IGNORE                      (5)
```

Rule 1 neglects the *name* of fault trees. Rule 2 and 3 specify a required criteria to match fault trees: They are matched in case they declare the same gates (whereas an equal name is sufficient to match gates (3)). Rules 4 and 5 are used to filter (to ignore) any other content.

A comparison with the rules yields matches, whereas any match (E_A, E_B) with

- E_A and E_B are basic events and
- $name(E_A) \neq name(E_B)$

represents a renamed basic event (E_A has been possibly renamed to E_B or inverse).

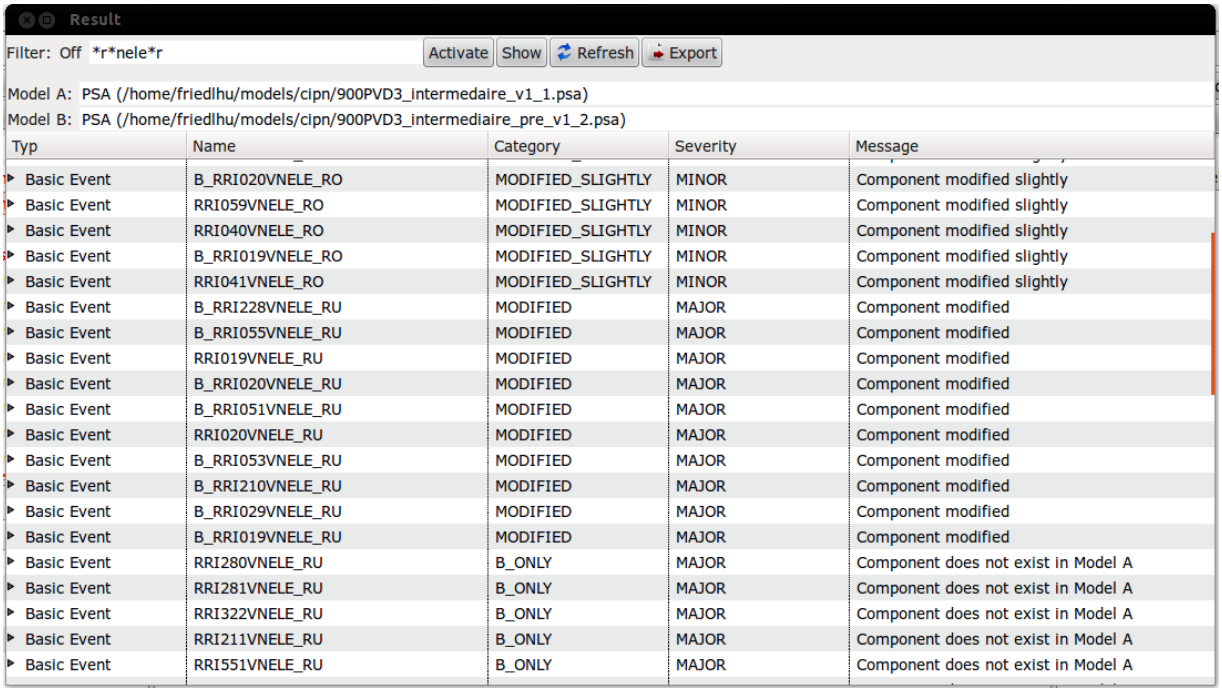
7.3.3 Visualization

A set of matches as a whole but also individual matches can be visualized.

Visualization of Matches

Once a set of matches has been determined, it can be visualized.

Screenshot of Figure 7.25 shows two "*real*" PSA models of EDF that have been compared in Andromeda. In total, about 15000 modules have been compared. The comparison procedure lasted less than 10 seconds. Finding matches is fast (complexity $n*log(n)$ with the "right" algorithm, n being the number of modules).



Typ	Name	Category	Severity	Message
Basic Event	B_RRI020VNELE_RO	MODIFIED_SLIGHTLY	MINOR	Component modified slightly
Basic Event	RRI059VNELE_RO	MODIFIED_SLIGHTLY	MINOR	Component modified slightly
Basic Event	RRI040VNELE_RO	MODIFIED_SLIGHTLY	MINOR	Component modified slightly
Basic Event	B_RRI019VNELE_RO	MODIFIED_SLIGHTLY	MINOR	Component modified slightly
Basic Event	RRI041VNELE_RO	MODIFIED_SLIGHTLY	MINOR	Component modified slightly
Basic Event	B_RRI228VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	B_RRI055VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	RRI019VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	B_RRI020VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	B_RRI051VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	RRI020VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	B_RRI053VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	B_RRI210VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	B_RRI029VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	B_RRI019VNELE_RU	MODIFIED	MAJOR	Component modified
Basic Event	RRI280VNELE_RU	B_ONLY	MAJOR	Component does not exist in Model A
Basic Event	RRI281VNELE_RU	B_ONLY	MAJOR	Component does not exist in Model A
Basic Event	RRI322VNELE_RU	B_ONLY	MAJOR	Component does not exist in Model A
Basic Event	RRI211VNELE_RU	B_ONLY	MAJOR	Component does not exist in Model A
Basic Event	RRI551VNELE_RU	B_ONLY	MAJOR	Component does not exist in Model A

Figure 7.25. List of matches after model comparison.: Each line represents one match. The "Category" in the example reflects the match type (Andromeda calls them differently).

One can notice a button *export* in the screenshot. This button serves to export the matches into a text format readable by external tools for further processing.

Visualization of Differences

Model engineers can work in two steps. In a first step they compare models to obtain the set of matches (for the whole model).

Then, in a second step they selectively depict individual matches (E_A, E_B) to figure out the concrete differences between E_A and E_B. For this purpose, Andromeda supports a graphical and a textual comparison.

The second step is completely independent from the first step. The first step serves "only" to offer a list of matches. But neither match types nor the applied importance rules do impact the comparison of the second step.

Textual Comparison

Figure 7.26 shows an example in Andromeda where two fault trees are compared textually.

The example demonstrates how two fault trees can be compared textually in a generic manner. The advantage of textual comparison is that the full set of differences becomes visible. For example, on the right side one can notice attributes *modified* and *modifiedBy* in blue color. Those information are not necessarily visible in a graphical comparison. The disadvantage is that, to understand the fault tree structure (or other module structures) in a textual format, a deeper experience is required.

Graphical Comparison

Figure 7.27 shows a graphical comparison for fault trees. Differences between the fault tree on the left

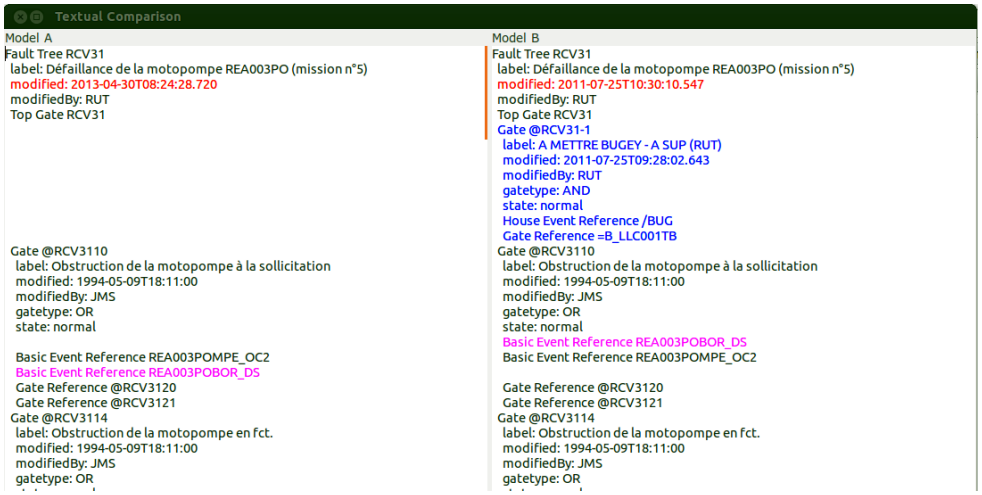


Figure 7.26. Example of a textual comparison (Andromeda): Two fault trees are compared. Modified sections are colored in red, moved sections in magenta and additional sections in blue color.

and the fault tree on the right are colored in blue (in the left fault tree). Reverse differences are colored in red (in the right fault tree).

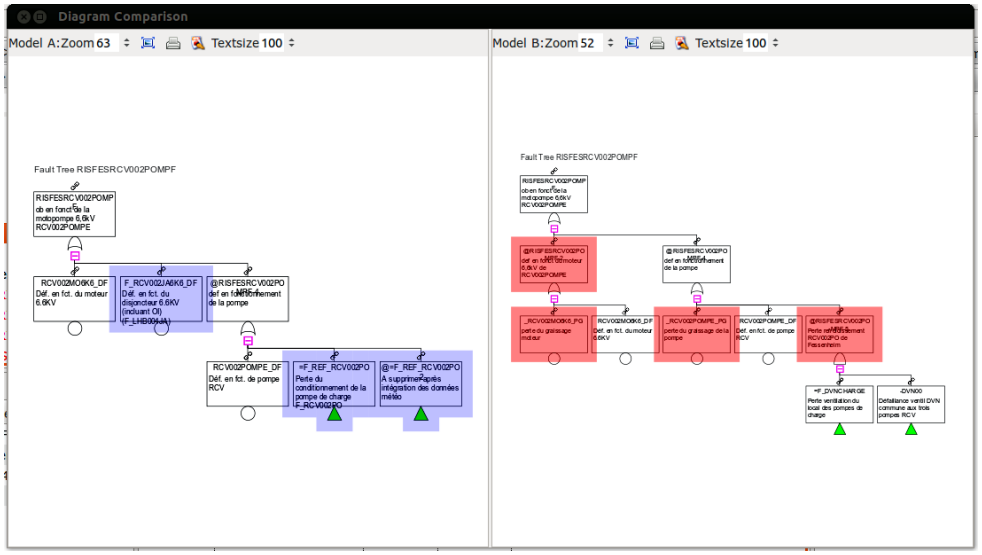


Figure 7.27. Graphical fault tree comparison in Andromeda: Two fault trees are compared graphically. Blue and red color indicate the differences.

Likewise, Figure 7.28 illustrates a graphical comparison for two event trees.

The results in Andromeda prove that a graphical comparison can highlight the differences between modules. At the same time, the kind of comparison is limited to the differences that are visible in the graphics. For example differences in labels or timestamps (time of last modifications) are not visible. Generally, graphical comparisons should thus always be regarded as an additional help to understand differences, but never as exhaustive comparison methods.

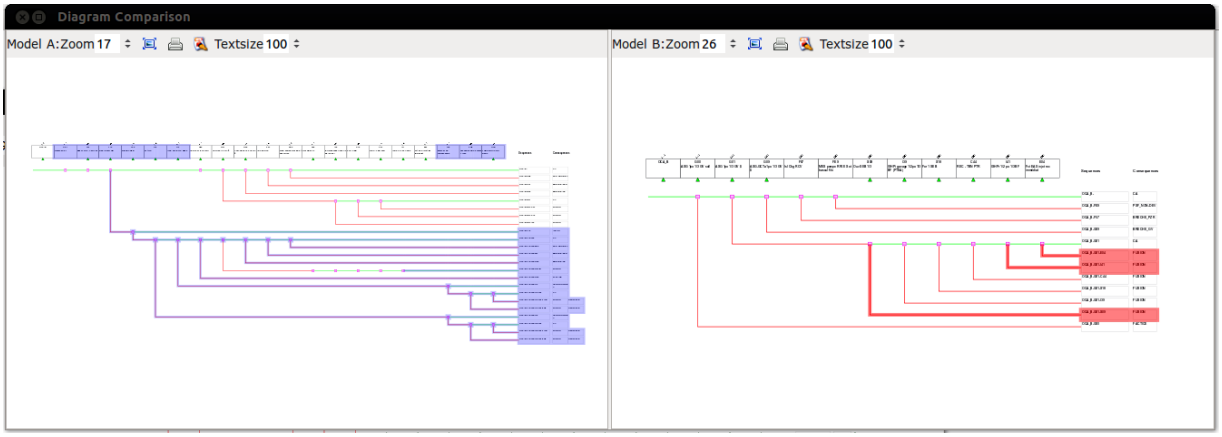


Figure 7.28. Graphical event tree comparison in Andromeda: Two event trees are compared graphically. Blue and red colors indicate the differences.

7.4 Model Fusion

In the last section, a method of model comparison demonstrated how models can be compared and differences between them determined. In this section it is analysed how those differences can be integrated. The integration kind of combines the content of two models. The result of the merge is another model or another model version. The integration procedure is called *model fusion* or *model merge*.

Currently, to integrate modifications of one PSA model into another, specific efforts are realized: Typically, model parts to integrate are replicated manually (by model engineers). Apart from the risk of replication errors, this procedure is suboptimal in terms of efficiency.

At software engineering, the process of integration is much more understood than at model engineering. Often, guidelines define the role of an integrator, a dedicated person, which is responsible to merge the different actions of developers. Software tools such as VCS systems offer several methods to support the integration process.

In this section, methods to fusion models in a modular PSA are presented. Those methods target to speedup the integration process and at the same time to reduce the risk of replication errors.

7.4.1 Technical Concept

In principal, model fusion is to combine two or more so-called *source models* into a new model called the *target model*. The target model contains the result of the fusion procedure called the *merge result*.

The target model may be one of the source models, i.e the merge result is written back to one of the source models.

A so-called *merge strategy* is applied to decide **how** models are merged.

Merge Strategies

In the sequel, two merge strategies are presented.

Two Way Merge The so-called *two way merge* merges two source models S_1 and S_2 into the target model M_T. Figure 7.29 illustrates the two way merge. The merge result relies only on the two source models (that's where the name "two way merge" comes from).

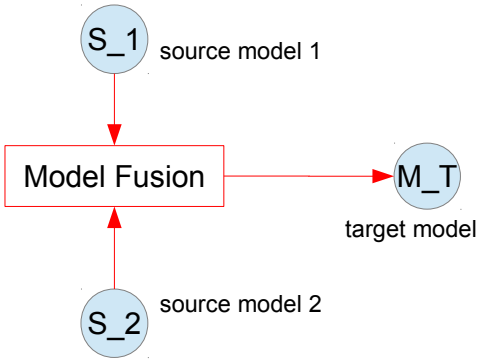


Figure 7.29. Principle of a two way merge: Two models S_1 and S_2 are merged. Model M_T represents the merge result.

In a modular PSA, the two way merge is performed in several steps:

Step 1:
The two way merge requires to perform a model comparison as a preliminary step (see Section 7.3). The result of the model comparison is a set of module matches Map of the form (M_1, M_2), whereas M_1, M_2 are modules with $M_1 \in S_1 \cup \{\perp\}$ and $M_2 \in S_2 \cup \{\perp\}$.

Step 2:
In a second step, all module matches of type $EQUAL$ are replicated to the target model M_T and discarded from Map. Those matches represent unmodified modules.

Step 3:
The third step consists of selecting a subset of matches Map' with $Map' \subseteq Map$. For example one can consider to merge a fault tree and all its dependent modules.

Step 4:
Each match $(M_1, M_2) \in Map'$ is merged step by step (by a model engineer). The following choices exist:

- M_1 is replicated to M_T.
- M_2 is replicated to M_T.
- A domain specific merge (see Section 7.4.1) is performed to obtain a module M' which is kind of a mix between M_1 and M_2. Finally, M' is replicated to M_T.

It is to mention that the two way merge can be (partly) automatized. For example, rules can be considered to specify when (under which conditions) to take M_1 and when to take M_2 of a match (M_1, M_2) for creating M_T.

Three Way Merge A so-called *three way merge* is a merge strategy to merge two source models (likewise a two way merge) into the target model but by taking an additional model into account called the *ancestor model*. The ancestor model represents typically an ancient model version, the two source models are based on (the models to merge are often branched from a common model version). It is used to automatically decide which modules are to replicate from which source model.

Figure 7.30 illustrates the tree way merge: Two source models S_1 and $S2$ are merged with respect to an ancestor M_A. $\Delta 1$ represents the differences between M_A and S_1 and $\Delta 2$ the differences between M_A and

S_2. The merge result M_T depends on $\Delta 1$ and $\Delta 2$.

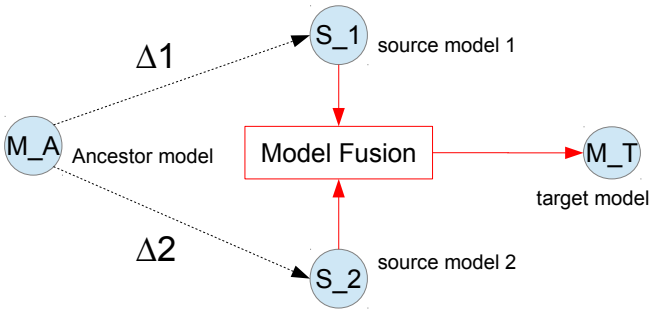


Figure 7.30. Principle of a three way merge: Two models S_1 and S_2 are merged whereas differences $\Delta 1$ and $\Delta 2$ to a common ancestor M_A are taken into account for creating M_T (the merge result)

The three way merge is performed in several steps:

Step 1:
Three model comparisons are performed:

- Comparison between S_1 and S_2: The result is a list of matches Map_{12}.
- Comparison between M_A and S_1: The result is a list of matches Map_{A1}.
- Comparison between M_A and S_2: The result is a list of matches Map_{A2}.

Step 2:
In a second step, all module matches in Map_{12} of type *EQUAL* are replicated to the target model M_T and discarded from Map_{12}. Those matches represent unmodified modules.

Step 3: In a third step the sets of modified modules are determined (since the ancestor):
$\Delta 1 = \{M_1 \mid \exists M \in Map_{A1} \text{ with } M = (M_A, M_1) \wedge type(M) \neq \text{EQUAL}\}$
$\Delta 2 = \{M_2 \mid \exists M \in Map_{A2} \text{ with } M = (M_A, M_2) \wedge type(M) \neq \text{EQUAL}\}$

whereas function *type* returns the match type of a match.

Step 4:
The fourth step consists of selecting a subset of matches Map'_{12} with $Map'_{12} \subseteq Map_{12}$.

Step 5:
Each match $(M_1, M_2) \in Map'_{12}$ is merged due to the following conditions:

- $M_1 \in \Delta 1 \wedge M_2 \notin \Delta 2$: The module has been modified in S_1 but not in S_2. S_1 is replicated to M_T.
- $M_1 \notin \Delta 1 \wedge M_2 \in \Delta 2$: The module has been modified in S_2 but not in S_1. S_2 is replicated to M_T.
- $M_1 \in \Delta 1 \wedge M_2 \in \Delta 2$: The module has been modified in both source models. This situation is called a "merge conflict".
  Human interaction is needed (as with a two way merge) to decide how to merge the two modules. A domain specific merge (see Section 7.4.1) can be performed to obtain a module M' which is kind of a mix of M_1 and M_2.

The benefit of a three way merge (respective a two way merge) is to limit the need of human interactions to resolve merge conflicts.

In the sequel an example of a three way merge is shown: `M_A` is the ancestor and `S_1` and `S_2` are the source models. Symbol "!" indicates modifications since the ancestor:

```
M_A (ancestor model):   S_1 (source model 1):    S_2 (source model 2)
```

```
├── Def FT ft1
│   ├── Def gate1
│   │   ├── be1
│   │   ├── be2
│   │   └── gate2
│   └── Def gate2
│       └── be3
├── Def BE be1
│   └── value=2.4e-7
├── Def BE be2
│   └── value=4e-8
└── Def BE be3
    └── value=1e-10
```

```
├── Def FT ft1
│   ├── Def gate1
│   │   ├── be1
│   │   │          (!)
│   │   └── gate2
│   └── Def gate2
│       └── be3
├── Def BE be1
│   └── value=2.4e-7
├── Def BE be2
│   └── value=4e-8
└── Def BE be3
    └── value=5.3  (!)
```

```
├── Def FT ft1
│   ├── Def gate1
│   │   ├── be1
│   │   ├── be2
│   │   └── gate2
│   └── Def gate2
│       └── be3
├── Def BE be1
│   └── value=3.5 (!)
├── Def BE be2
│   └── value=4e-8
└── Def BE be3
    └── value=6.7 (!)
```

The sets of modified modules (since the ancestor) are $\Delta 1 = \{ft1, be3\}$ and $\Delta 2 = \{be1, be3\}$. Consequently *ft1* is replicated from S_1 and *be1* from S_2. However, *be3* has been changed in both source models and constitutes a merge conflict.

The result of the merge M_T is:

```
M_T (merge result):
├── Def FT ft1                 (replicated from S_1)
│   ├── Def gate1
│   │   ├── be1
│   │   │
│   │   │
│   │   └── gate2
│   └── Def gate2
│       └── be3
├── Def BE be1
│   └── value=3.5              (replicated from S_2)
├── Def BE be2
│   └── value=4e-8
└── be3
    └── value=5.3 or 6.7   (!) (merge conflict)
```

Three way merge and VCS:
In version control systems (VCS, see Section 7.2), the ancestor and the source models are represented by *versions*. Many VCS systems permit to determine the ancestor automatically respective two model versions (the source models). VCS systems provide a version and merge history and can calculate the most recent ancestor version two versions have in common. It is important to determine a recent version in order to limit the number of merge conflicts.

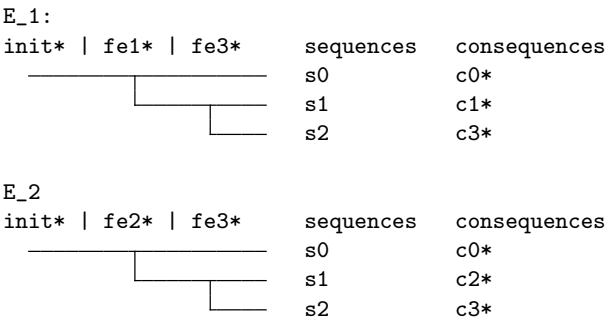
However, the automatic determination of an adequate ancestor version requires to have "proper" version and merge histories in VCS systems. Those may have been manipulated manually by engineers (accidentally or by intention) what leads to detect a "wrong" ancestor. Consequently, the merge result may be falsified.

Domain Specific Merge

The presented merge strategies (two and three way merge) do not really merge modules. They rather replicate them from source models.

However, sometimes it is necessary to merge two modules, for example in case of merge conflicts. Merge conflicts are to resolve manually or by the application of domain specific routines. Domain specific routines are in contrast to generic routines and are specific to the type of the modules to merge.

For example, for event trees, a specific method can be developed that merges two event trees. This method is presented in the sequel by an example:
Two event trees E_1 and E_2 should be merged. The event trees are defined as follows:



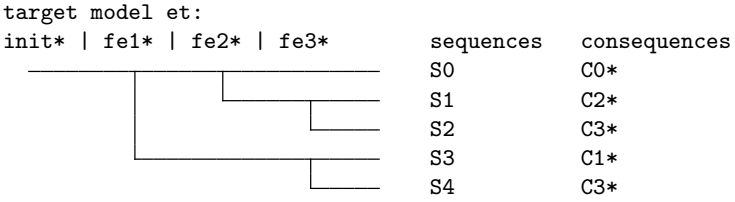
A possibility to merge event trees is to merge them by their sequences. Sequences of event trees can be written as logical expressions:

```
model A:                    model B:
c0  = !fn1 * !fe3           c0  = !fn2 * !fe3
c1 =  fn1 * !fe3            c2 =  fn2 * !fe3
c3  = fn1 * fe3             c3  = fn2 * fe3
```

To fusion the sequences of event trees, negated function events (that indicate success paths) are discarded and the equations combined:

```
c0  = ⊥
c1 =  fn1
c2 =  fn2
c3  = fn1 * fe3
c3  = fn2 * fe3
```

The sum of all sequences represents a system of equations. A new event tree can be derived from the set of equations:



However, the derived event tree may not be optimal in the sense that it may contain more sequences as necessary in order to express its Boolean formula. The problematic is identified in Section 9.2.2 in the context of event tree optimization.

7.4.2 Model Fusion in Andromeda

A *Merge Editor* has been developed in Andromeda that implements a two way merge. A three way merge has not yet been implemented. Before implementing a three way merge it is worth to consider the feedback (from industrial application) about the two way merge procedure. It is likely that a two way

merge is already an adequate procedure as it gives full merge control to model engineers whereas a three way merge is merging model artifacts automatically and may be too complicated and misleading.

Figure 7.31 shows a screenshot of the *Merge Editor* of Andromeda. The *Merge Editor* performs first a model comparison and than proposes the differences to merge. A model engineer selectively picks module matches to merge.

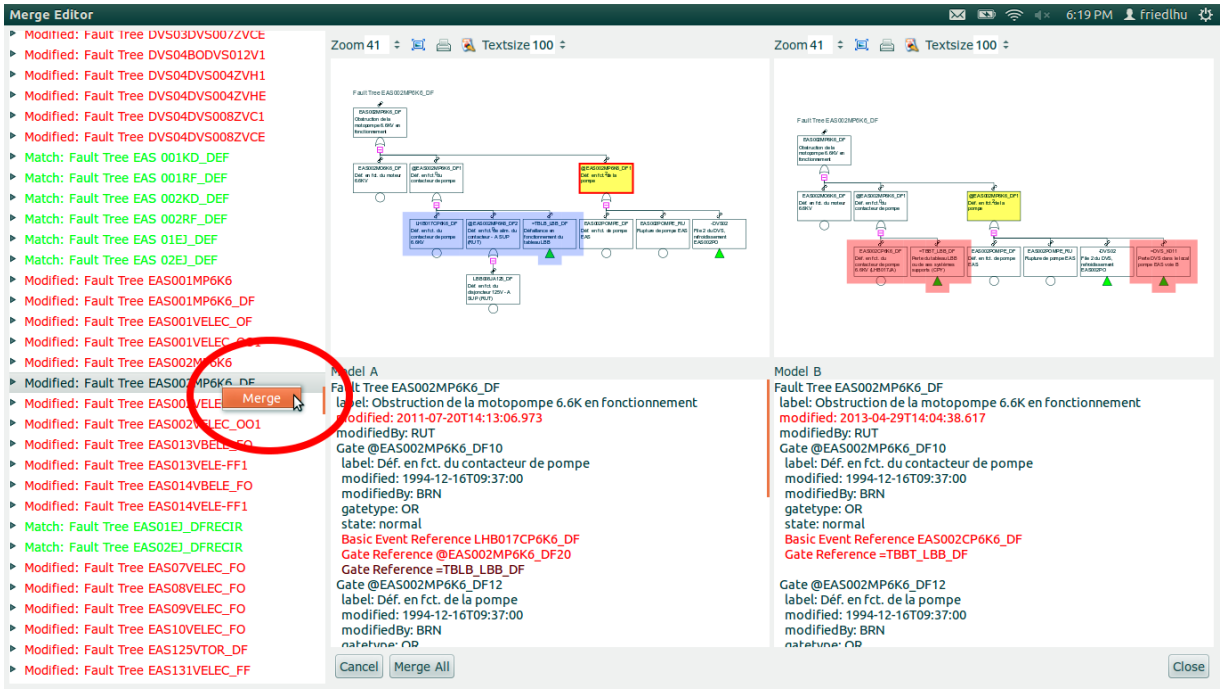


Figure 7.31. Screenshot of Merge Editor: Two models have been compared. The Merge Editor allows to navigate the comparison result and selectively merge modules.

7.5 Consistency Check

The introduced methods, in particular those of variant management can lead to inconsistent models. For example, if a variant does not contain a certain set of modules, but others are still referring to them, those references become "unresolved".

In this section, an overview about the matter of model consistency is given.

7.5.1 Detecting Inconsistencies

A consistency check targets to reveal inconsistencies of a model.

In PSA models, inconsistencies are for example:

- Unresolved references.
- References that cannot be uniquely resolved (may happen if components of same name and type occur multiple times within the same model).
- Fault trees without top gate.
- Parameters with no value assigned.
- Gates with no gate inputs defined.

Inconsistencies are sometimes not obvious as they can be on a semantic level (a model can still conform to its underlying domain though being inconsistent).

Undetected inconsistencies can lead to unexpected results (for example at quantification). To find the problem in large PSA models can be time consuming without the help of software tools.

7.5.2 Consistency Check in Andromeda

The idea in Andromeda was to establish a generic framework to check for inconsistencies. The framework is extensible so that new types of consistency checks can be added.

Andromeda defines a so-called extension point (see Section 10.3.1) to provide new consistency checks.

The extension point for model analysis is named:
`fr.edf.andromeda.analysis`.

To implement a new consistency checker, the extension point must be implemented by providing a function:
`public List<I_Element_Entry> analyse_element(I_Element element);`

This function returns a list of problems for a model element. To indicate no problem, the list must be empty.

A checker to detect unresolved references validates whether the element to check is resolvable under the current context or not. If not resolvable, a corresponding entry is added to the list of problems. Otherwise, the empty list is returned.

To run the new consistency checker, Andromeda iterates over all model elements and calls the implemented function. Once completed, any problems are shown in a graphical widget.

Figure 7.32 shows the result of the "unresolved references" detection.

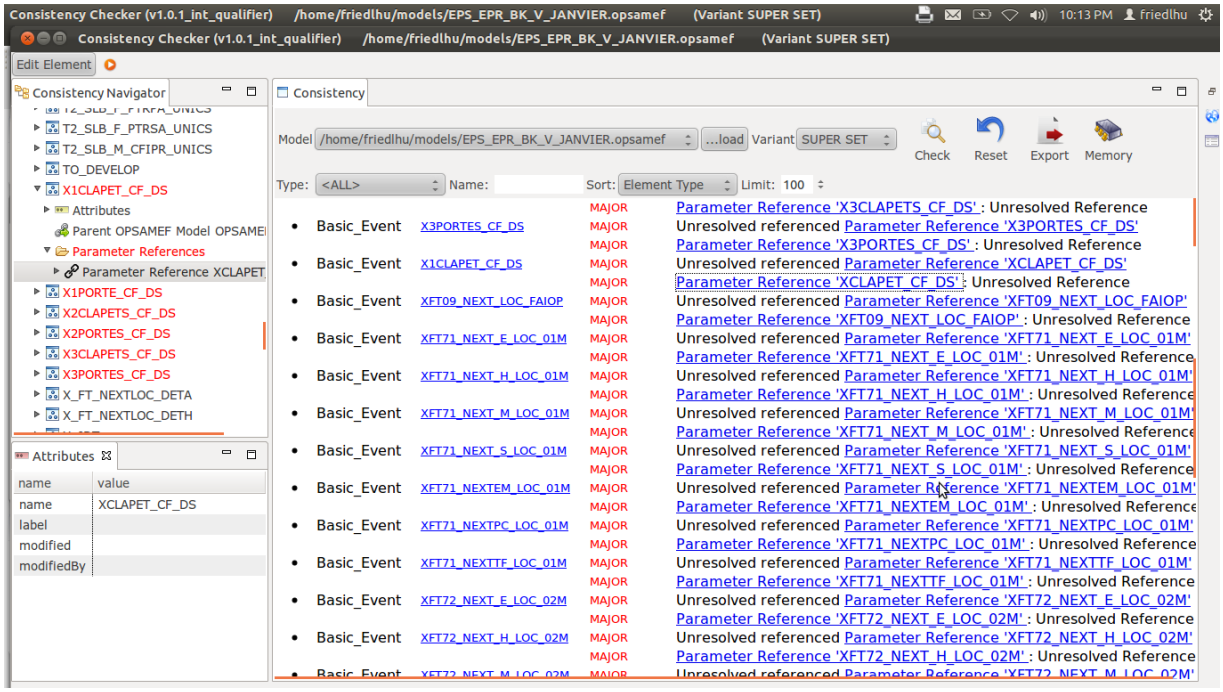


Figure 7.32. Screenshot of Consistency Checker.

The list of problems (in the example a list of "unresolved references") can be navigated and repaired one

by one.

7.6 Conclusion

In this chapter, methods to manage PSA models have been presented. Those methods are supposed to facilitate the process of model engineering. They are considered as mandatory to manage future PSA models whose complexities and sizes are continuously increasing.

A variant and version management have been introduced to manage the evolution of models along two dimensions: Variants manage the deviations and versions chronological evolutions. To support version management, a method of model comparison and a method of model fusion have been introduced.

For managing PSA models for nuclear power plants, the presented methods are supposed to gain high industrial interest. In particular the function of model comparison is required to verify model modifications and to automatically generate reports for justifying model modifications to safety authorities in an efficient and reliable manner.

CHAPTER 8

PSA Model Analysis

PSA model analysis is mainly understood as quantitative and qualitative risk assessment. However, to develop and extend PSA models, other types of analysis are required. Successful model engineering requires to understand the detailed composition of model objects and any relations between them. A new understanding of qualitative risk assessment is required: Any method that helps to understand PSA models helps to understand risk. In this context, computer science can provide new methods to reinforce qualitative risk assessment [59].

In this chapter, concepts and methods are presented to exploit information from a PSA model in a modular PSA. A special focus is given to the analysis of module dependencies.

Section 8.1 shows how to document models in a modular PSA.

Section 8.2 introduces the matter of dependency analysis and its interest at model engineering.

Section 8.3 presents an idea to use graph models (representing physical systems) to perform safety analysis on a system level.

Section 8.4 presents visualization methods to focus on pertinent (interesting) model parts.

Section 8.5 concludes this chapter.

8.1 Exploiting information from Documentation

The purpose of model documentation is to provide information of any kind about PSA models to model engineers and safety analysts. For example, documentation can inform about:

- The purpose of model objects (why they have been modeled).
- The detailed composition of model objects (how they are modeled).
- Relations between model objects (what are there dependencies).

- Specific background knowledge and context related information

Model documentation facilitates to understand models. It can support model engineers to develop models any further and safety engineers to interpret safety studies (e.g. PSA results).

Documentation can be classified in two categories:

- Internal documentation: Integrated in PSA models.
- External documentation: Located on file servers / Intranet / Internet.

Today, at EDF, both kind of documentation assets exist and are maintained. However, documentation is difficult to exploit. Internal documentation is realized in a rather basic manner: They are short textual descriptions (so-called "labels"). And external documentation is difficult to locate as it is not linked from the models and additional knowledge and efforts are necessary to find and access it.

In this section, new technical concepts and aspects are proposed to document models in a more intelligent way. The approaches base on technology successfully established in the world wide web. They intend to provide a more exhaustive model documentation and to establish a so-called "documentation network" in a modular PSA. An article concerning this subject has been published in [60].

8.1.1 Principle of a Documentation Network

Typically, PSA model documentation is not kept centrally at a single location, it is rather distributed over many locations. Some parts of the documentation may reside directly in PSA models (internal documentation), other parts may be stored in the Intranet, Internet or on local file systems (external documentation).

Such a distributed documentation of PSA models can be difficult to manage. Two major problems can be made out:

- Identification of documentation: How to figure out, what kind of PSA documentation exists and where is it stored?
- Information filtering: How to filter PSA documentation to view only relevant parts (and not to get disturbed by too many information)

The solution is to respect the distributed character of models and documentation assets and to **link** the different parts together by the means of a distributed documentation network.

Figure 8.1 illustrates the principle: Model objects (such as fault trees) reference related model objects, their documentation or their diagrams. The related objects may reside in the same model or in others. Further, external documentation (located in the Internet / Intranet) can be referenced.

Technical Components

A modern documentation may consist of the following assets:

- **Textual information:** Descriptions in textual form.
- **Diagrams:** Graphical information mainly to illustrate relations between model objects. Diagrams can either be automatically generated or manually composed. In PSA models, there are two well known diagram types: Fault- and event trees. Those diagrams can be derived from the content of a PSA model. Thus, they can be regarded as generated documentation assets.
- **Multimedia files:** Multimedia files can help to explain difficult circumstances in form of images, videos, simulations etc. It is not recommended to keep those kind of files in the model due to a risk

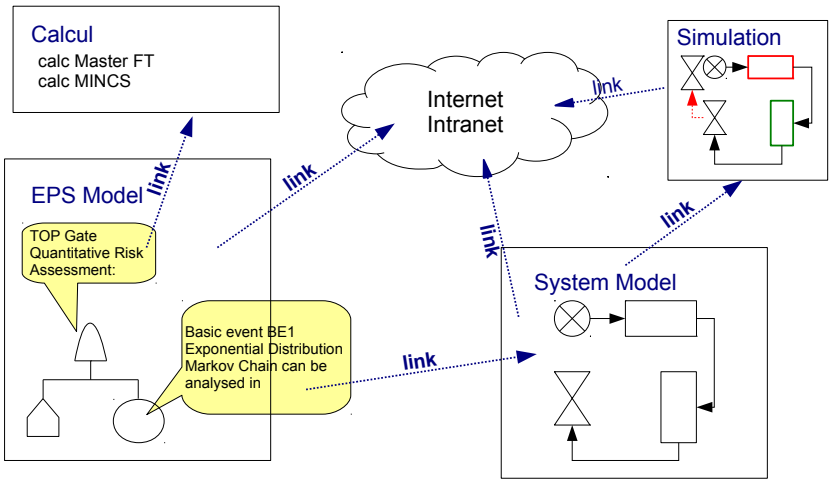


Figure 8.1. Linking distributed model documentation

of blowing up model sizes. However, they can be referenced ("linked") from other documentation assets.

- **Links:** Links can serve to reference related documentation assets. For the idea to establish a distributed documentation network, they gain special importance: Links permit to access quickly relevant documentation regardless their location. They can access diagrams, documentation of other model objects, information on the web etc. Links have the capacity to improve the exploitation of information significantly.

Assembling Documentation

Concerning the manner documentation is developed, documentation can be classified further into:

- Manual documentation: Explicit information obtained by experience and background knowledge of model engineers and safety analysts. This kind of documentation is manually developed. Nowadays model documentation is mainly of this type.
- Generated documentation: Information which can be derived from the content and structure of models, for example relations between model objects or hierarchical composition (e.g. which fault trees contain which gates).

The advantages of generating documentation automatically are twofold:

- Efficiency: Generation processes can provide documentation in very short time.
- Consistency: Generated documentation is consistent with the models it is derived from. If models are modified, documentation remains "up-to-date". In a modular PSA, another type of consistency becomes important: If the instantiation context changes, the generated documentation "adapts" automatically to the new context.

The final documentation is a mixture of generated and manual documentation. Figure 8.2 illustrates the assembling.

Templates The idea of templates is to customize a model documentation to different needs, communication mediums and representations, so-called "targets":

- They (the templates) can configure the automatic generation of documentation.
- They define the assembling of manual and generated documentation assets.

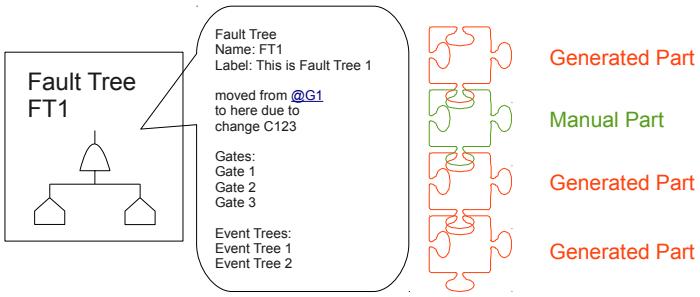


Figure 8.2. The final documentation is a mixture between generated and manual documentation

- They permit to define so-called "layouts", which impact the graphical representation (e.g. configuration of text sizes, colors, fonts, arrangement of text and diagrams etc.).

The principle of templates is illustrated in Figure 8.3. In the example, three templates configure the final documentation to different targets.

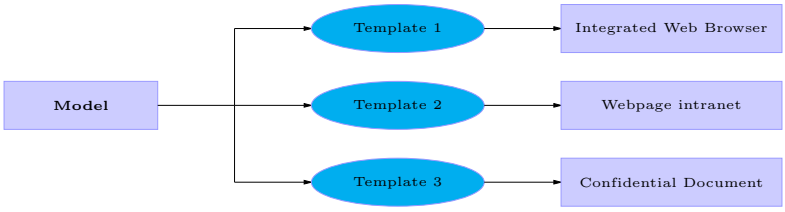


Figure 8.3. Adapting documentation by using templates

8.1.2 Model Documentation in a modular PSA

In a modular PSA, each model element has its individual documentation. By default, templates automatically derive structural information of elements and embed the result in documentation. For example, the documentation of a fault tree provides the list of event trees which link the fault tree. Basic events the list of fault trees, which require the respective events.

The main enhancement to existing PSA documentation is the introduction and treatment of links. To handle links, the URL (Unified Resource Locator) [61] specification has been extended to address model elements in a modular PSA. URLs are used to uniquely identify resources. They have become famous to address web pages in the world wide web (though they are not limited to this purpose).

URL Extension

In this section, a proposal for an URL extension is presented for embedding links in model documentation. The idea is that users can "follow" links by simply clicking on them, similar as it is the case in native web browsers. A link can serve to open another model object, its documentation, its diagrams or external documentation.

The URL scheme for web sites (for the protocol "Http" [62]) is the following:

```
http://<user>:<password>@<host>:<port>/<url-path>?<searchpart>#<fragment>
```

For example:

`http://john:secret@example.org:80/demo/example.cgi?land=fr&city=paris#history`

Whereas this scheme is sufficient to reference external documentation, it is unfortunately not sufficient to reference model objects. To achieve the latter, an extension is required leading to a new protocol "model":

`model://<model-path>:<element-type>/<element-name>?<action>`

The model protocol consists of the following components:

- `model`: The name of the protocol (obligatory)
- `model-path`: Relative or absolute path of a model (optional)
  If specified, then the URL is transferred to this model for further treatment. The model can be of a any domain. This is possible, because the addressing is not domain specific. Further, it permits to interconnect various model types by their documentation. For example one can link event sequence diagrams (ESDs, see Section 9.2) with event trees (though ESDs do generally not know the notion of event trees).
  If the model path is not specified, the link is "resolved" (see Section 8.1.2) in the current model (the model defining the URL).
- `element-type:` Element type (obligatory)
  Type of the model element which is referenced by the link (see Table 8.1).
- `element-name:` Element name (obligatory)
  Name of the model element which is referenced by the link. In a modular PSA, the name of model elements (of the same type) are unique within the same model.
- `action`: Action to apply (optional)
  Action to apply on the element referenced by the link (see Section 8.1.2).
  An action can be used to display a certain diagram (e.g. fault tree diagram), to open another documentation etc. If not specified, a default action is applied which is to open the model documentation of the model element.

Only components can be referenced by a link. An extract of components in a PSA model is given in Table 8.1:

Table 8.1. Extract of components in a PSA model

Element Type	Description
FAULT-TREE	fault trees
GATE	gates
EVENT-TREE	event trees
SEQUENCES	sequences
INITIATING-EVENT	initiating event
FUNCTION-EVENT	function event
BASIC-EVENT	basic events
HOUSE-EVENT	house events
PARAMETER	parameter
CONSEQUENCE	consequences
...	...

Actions

Actions embedded into URLs are a powerful instrument: They permit to embed software functionality into model documentation. Thus, model documentation becomes a tool itself for example to navigate

into PSA models by their documentation, to open diagrams, to edit models etc. As actions are a generic construct, any kind of PSA functionality can be considered to be embedded.

In order to open a model documentation, an action *open-doc* is provided:
`model://<model-path>:<element-type>/<element-name>?open-doc`

To open diagrams, the action *open-diagram* is intended:
`model://<model-path>:<element-type>/<element-name>?open-diagram#diagram-type=<diagram-type>`

whereas "<diagram-type>" is the type of diagram to open (e.g. an event tree diagram).

URL Address Resolving

URLs encode a sequence of characters. In order to obtain the model object and the action to apply, URLs must be "resolved".

Figure 8.4 demonstrates how URLs are resolved. In case an URL specifies a web address, a typical resolving step (as done in any web browser) is performed. In case an URL specifies an action to be applied on an external model, the URL is forwarded to this model, where it gets finally resolved.

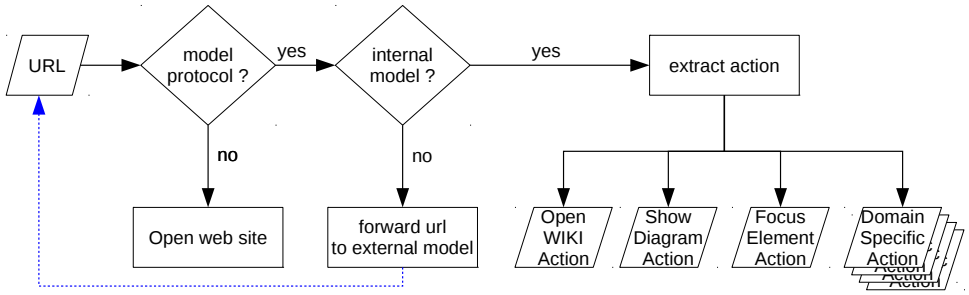


Figure 8.4. URL resolution in a modular PSA

If the URL specifies an action of the current model, the resolution is done in the following way:

- First, the referenced object is determined by its type and name
- Then, the action is determined, which should be applied to this object.
- Finally, the action is applied on the object (the action is "executed").

8.1.3 Application

The concept of model documentation has been implemented in Andromeda.

To "understand" the extended URL specification, an existing web browser technology has been integrated and adapted to support the "model" protocol. Consequently, any kind of actions can be embedded into model documentation to support model engineers in their daily life.

In Andromeda, new action types can be developed and used in future documentations by providing so-called "extension points" of type "fr.edf.andromeda.actions". The principle of extension points is explained in Section 10.3.1.

Templates and manual documentation are specified in the HTML (Hyper Text Markup Language [63]) format. HTML is a popular format many web pages are written in. To base on HTML allows to reuse existing web technology.

The screenshot in Figure 8.5 shows a model documentation of a fault tree. The documentation consists of a combination of generated and manual documentation. The generated part contains information about related model objects, e.g. its gates. This documentation is generated anew, each time one re-opens the fault tree documentation (this way it remains up-to-date). The manual part on the right side provides additional information and items, for example a graphic and a video. Clicking on the link "REQ 123" (the link at the lower right) results in opening a requirement model (another domain type in a modular PSA to specify requirements).

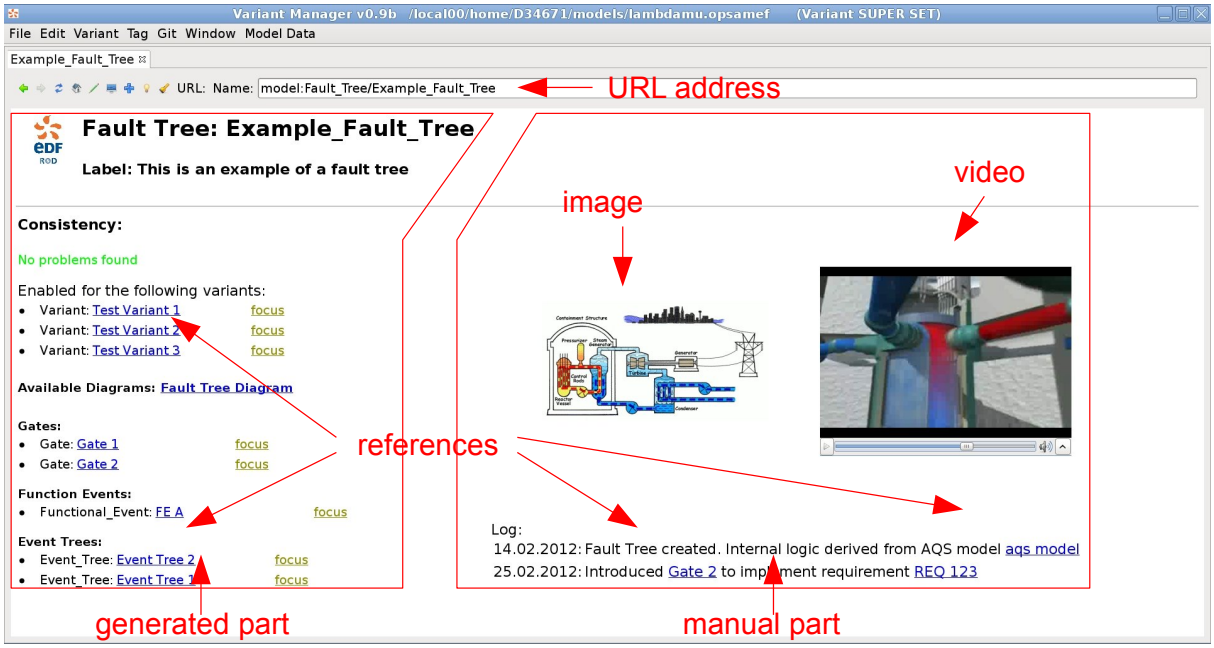


Figure 8.5. Model documentation in Andromeda: Example of an integrated model documentation consisting of a generated part (left part) and a manual one (right part)

Andromeda illustrates how PSA models can be navigated by their documentation: An address line on the top of the page allows to enter URLs. If no action is specified (by the URL), then the action "open-doc" (to view documentation) is applied. This way, one can use the address line like in any native web browser with the extension to interface with model objects of a modular PSA.

Figure 8.6 illustrates the interaction between documentation and diagrams: An action "focus" is used to ink an object of a diagram (in order to highlight it). In the example, by clicking on the "focus" link of "Gate 2", the corresponding gate of the fault tree is inked in yellow color.

The dialog shown in Figure 8.7 demonstrates how a documentation is assembled by using templates. The shown template is encoded in HTML language and assembles manual and generated documentation respective a certain layout. The generated part is referenced with *macros*, enclosed with three special characters (`$$$WithinDollars$$$`). When the final documentation gets created, all macros get automatically replaced by the corresponding generated parts.

Special dialogs are provided to specify links (see Figure 8.8). A model engineer can choose between a "model link" or a "web link". If specifying a model link, the prototype proposes the list of model elements that can be linked. The field "model" permits to link objects of another model. A web link can reference external documentation by "normal" URLs.

The URL address line can further be used to enter search expressions to filter a model for elements.

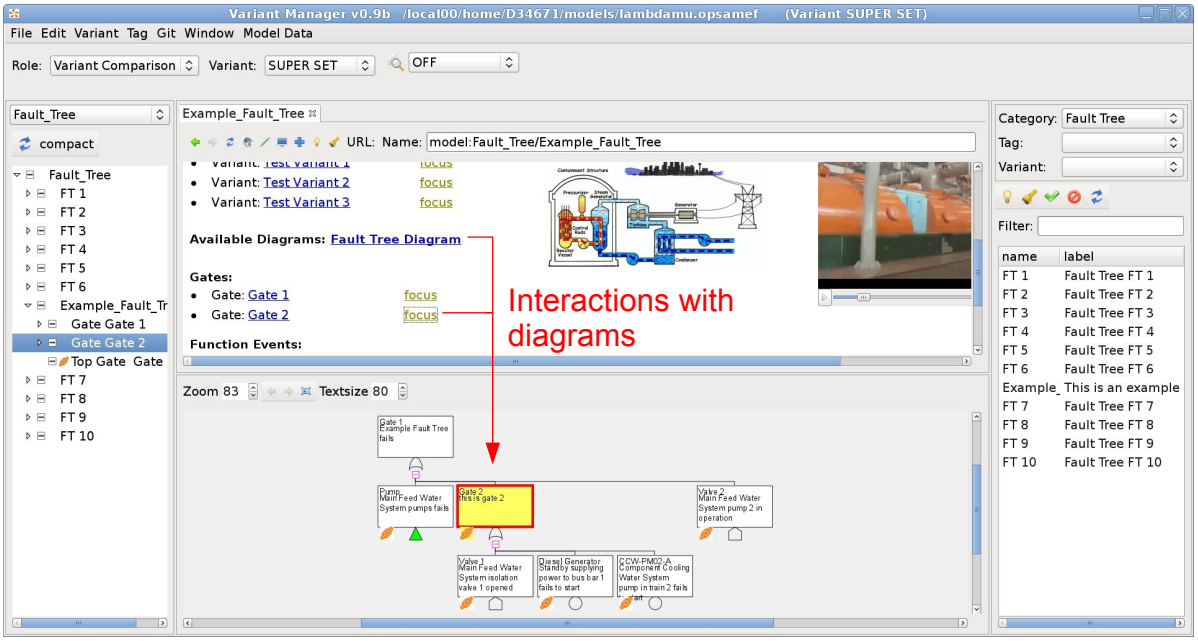


Figure 8.6. Interaction with model documentation

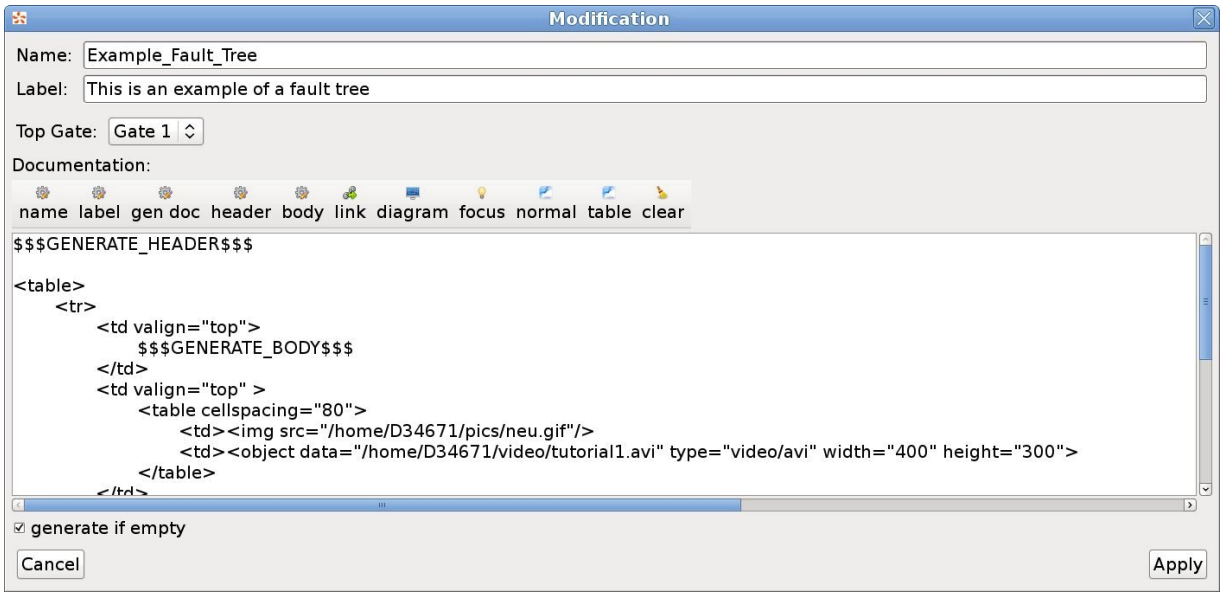


Figure 8.7. A template in Andromeda

Figure 8.9 shows an example, where any elements containing the string “asg14” in their names are filtered. When clicking on an element (of the search result), the corresponding documentation page is opened. The example illustrates the interaction of functionality with model documentation.

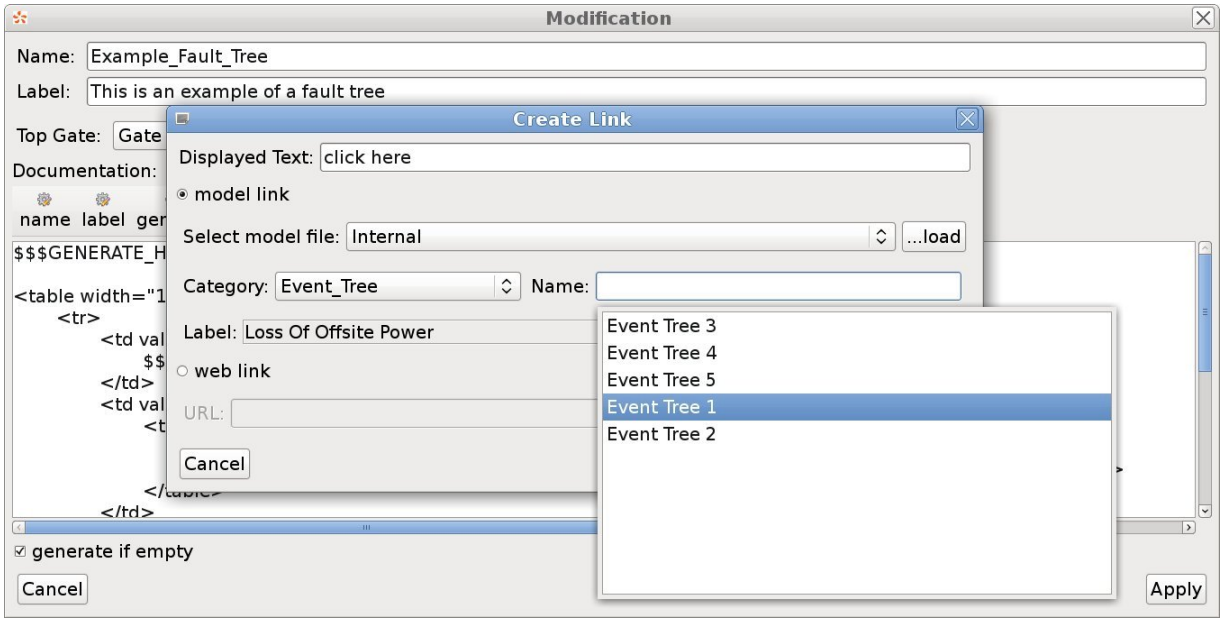


Figure 8.8. Linking the documentation of another object

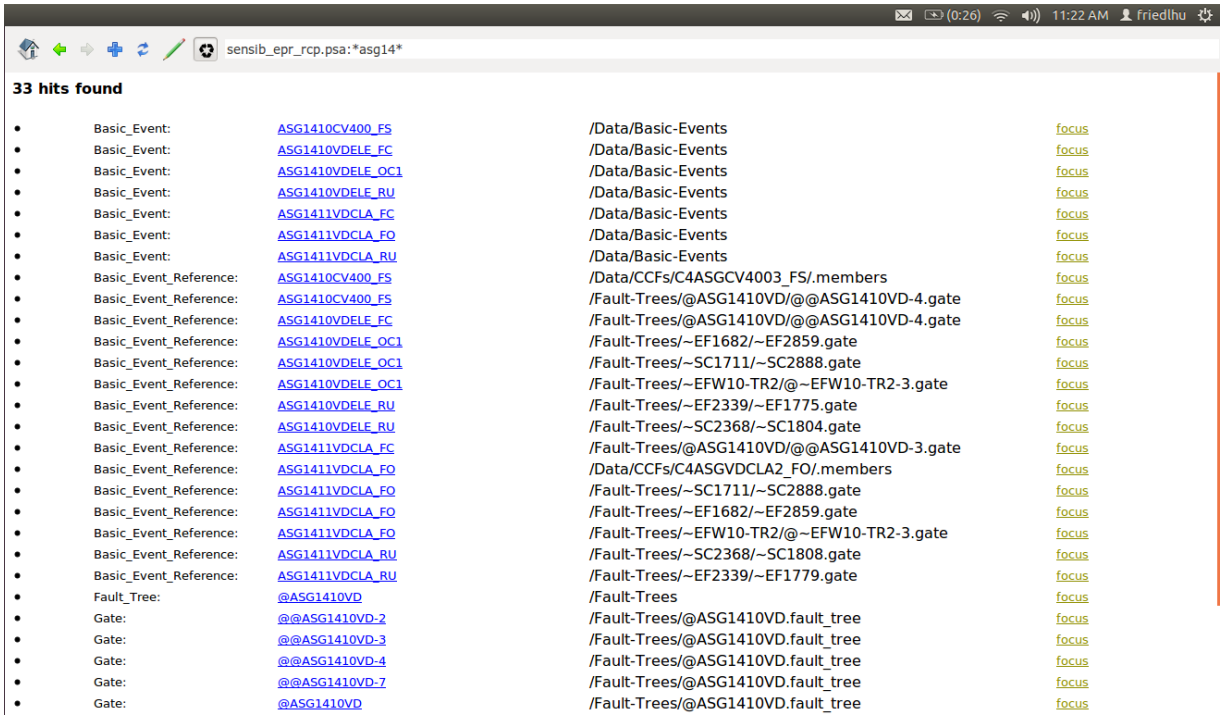


Figure 8.9. Filtering a model for elements.

8.2 Dependency Analysis

Dependencies between modules and their management play a major role in a modular PSA. To visualize those dependencies to model engineers is very important for understanding relations between objects. This section shows new possibilities to highlight dependencies, for example in form of a model cartography.

The process of determining dependencies between model objects is called *dependency analysis*. In a

modular approach, forward and backward dependency analysis can be performed. Those are explained in the next two sections by the means of an example model:

```
Model:
├── Def Fault-Tree ft1
│   ├── Def gate1 AND
│   │   ├── be1*
│   │   ├── be2*
│   │   └── gate2*
│   └── Def gate2 OR
│       └── be3*
├── Def Basic-Event be1
│   └── p1*
├── Def Basic-Event be2
│   ├── p1*
│   └── p2*
├── Def Basic-Event be3
│   └── p2*   │
├── Def Parameter p1
└── Def Parameter p2
```

8.2.1 Forward Dependencies

A forward dependency (M, M') from a module M to another module M' expresses, that M depends on M': M depends on a module M' in case M contains at least one reference that references an element of M'.

The set of forward references can be specified formally: Let τ be an instantiation context and MS a set of modules. Let $resolve(R, \tau)$ be the function to resolve references in a context τ (model instantiation has been introduced in Section 6.2.3). Then, the forward dependencies in MS respective a context τ are defined as

$$\{\ (M, M') \mid M, M' \in MS : \exists R \in References, R \in_{Rec} M\ :\ module(resolve(R, \tau)) = M'\ \} \setminus \{\ M, M\ \}$$

A dependency $(M,' M)$ can be written as

```
M ──> M'
```

In the example (given in Section 8.2), fault tree *ft1* has forward dependencies to three basic events. Basic event *be1* and *be2* develop dependencies to parameter *p1*, *be3* to *p2*. This can be written as:

```
ft1 ─┬─> be1 ──> p1
     ├─> be2 ─┬─> p1
     │        └─> p2
     └─> be3 ──> p2
```

Following the dependencies recursively leads to (forward) dependency paths. A dependency path can be stated as $\{M_1, M_2, \ldots\}$. If a module M_i occurs at least twice in a path, then a *circular dependency* exists. In this case, the dependency path contains an infinite number of modules.

The given example contains four dependency paths:

1. {ft1, be1, p1}
2. {ft1, be2, p1}

3. {ft1, be2, p2}
4. {ft1, be3, p2}

Dependency paths can be determined up to a certain *level*. The level defines the maximum number of elements in dependency paths.

For example, the dependency paths of level 2 are:

1. {ft1, be1}
2. {ft1, be2}
3. {ft1, be3}

8.2.2 Backward dependencies

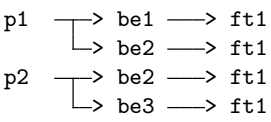
A backward dependency expresses the inverse relationship as a forward dependency.

A backward dependency (M, M') from a module M to another module M' expresses, that M is required in M': M is required in a module M' in case M' contains at least one reference that references an element of M.

The set of backward references can be specified formally: Let MS be a set of modules. The backward dependencies in MS in a context τ are defined as

$$\{\ (M, M') \mid M, M' \in MS : \exists R \in References, R \in_{Rec} M'\ :\ module(resolve(R, \tau)) = M\ \}\ \setminus\ \{\ M, M\ \}$$

The backward dependencies of the introduced example are :



The given example contains four backward dependency paths:

1. {p1, be1, ft1}
2. {p1, be2, ft1}
3. {p2, be2, ft1}
4. {p2, be3, ft1}

8.2.3 Cartography

Dependencies can be visualized textually and graphically. In this section, a so-called *cartography* is presented, which is a graphical method to visualize forward and backward dependencies.

A cartography is a visualization to explore iteratively the dependencies (for- and backward) of a module.

Figure 8.10 shows an example of a cartography for a fault tree (screenshot taken in Andromeda): Each rectangle represents a module. The fault tree module is painted in the center of the cartography. To the right side of it, forward dependencies are explored (iteratively). To the left side, backward dependencies are explored (iteratively).

One forward and one backward dependency path are focused at a time. The focused paths are indicated by green triangles. By clicking on the triangles, another path can be focused and the dependency level can be changed. To say, one can selectively navigate along the dependencies of modules.

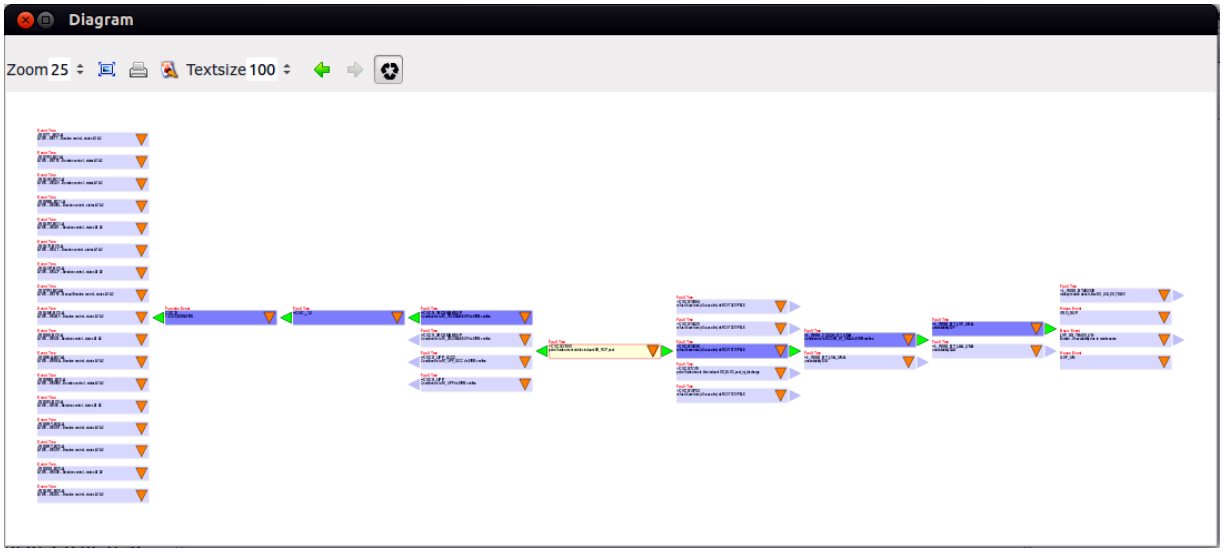


Figure 8.10. Cartography of a fault tree: Forward dependencies for the fault tree are printed to the right of the fault tree. Backward dependencies to the left side.

8.2.4 Usage- and Call Hierarchy

A second method for the visualization of for- and backward dependencies is the so-called “usage hierarchy” and “call hierarchy”.

The usage hierarchy represents forward references, the call hierarchy backward references. Both present dependencies in a tree form, whereas each tree node represents a module. Child nodes represent dependent modules. Leaves represent modules, which do not depend on others. They represent the last elements in dependency paths.

Figure 8.11 shows an example of a usage hierarchy (screenshot taken in Andromeda). The tree nodes can be extended to iteratively explore the forward dependencies of an event tree.

Figure 8.12 shows an example of a call hierarchy (screenshot taken in Andromeda): The tree nodes can be extended to iteratively explore the backward dependencies of a parameter.

Typ	Name	Label
Event Tree	TPCV1-FB2-A3	TPCV1 - LFW - F&B - RCPs stopped - state A3
Function Event	ISM01	1/4 MHSI (A,B)
Function Event	ISL16	1/4 INJ LHSI (P;w. exch)
Function Event	ISL38	1/4 INJ LHSI (A,B;w/o exch)
Function Event	SSS01	SSS avail.
Fault Tree	=SSS__01	
Function Event	CHR01	1/2 CHRS (IRWST)
Initiating Event	TPCV1-FB2-A3	TPCV1 - LFW - F&B - RCPs stopped - state A3
Function Event	SCD12	1MSS/1MSR SCD
Fault Tree	=SCD_12	
Consequence	SGP3	
Consequence	F	CORE MELT
Function Event	RCS04	1/2 PDS Op.
Fault Tree	=RCS__04	
Fault Tree	=RCSDV_FO_4_4_VPELE	Condition de la RI _FO_4_4_VPELE de EIRM verifiee
Fault Tree	~RCSDV203	refus de manoeuvre à l'ouverture de RCP6313VPELE
Fault Tree	=L_WISS_BT_LVP_UNA	unavailability LVP
Basic Event	RCP6313VPELE_OP1	Human factor - omission configuration
Fault Tree	=L_WISS_CC02H_D2_UNA	Condition de la RI C02H_D2_UNA de EIRM verifiee
Basic Event	RCP6313VPELE_FO	Motor-operated valve - primary coolant - Failure to open
Basic Event	RCP6313CV400_FS	Failure to close - valve contactor - 400V
Fault Tree	~RCSDV431	refus de manoeuvre à l'ouverture de RCP6323VPELE

Figure 8.11. Usage Hierachy: The usage hierarchy permits to explore forward dependencies iteratively.

Typ	Name	Label
Parameter	REA####POMPE_FR	RBWMS Pump - Failure to run
Basic Event	REA5120POMPE_FR	RBWMS Pump - Failure to run
CCF Group	C2REAPOMPE2_FR	CCF to run water RBWMS pumps
Fault Tree	~RBWMS2799	obstruction dans REA5120POMPE
Fault Tree	~RBWMS2736	obstruction dans le bloc incluant OD_BLOC_pumping_REA5
Fault Tree	=RBWMS_RCV_D1	1/2 RBWMS avail for CVCS dilution
Fault Tree	=CVCS_1VCT	Condition de la RI _1VCT de EIRM verifiee
Fault Tree	=CVC__03	
Function Event	CVC03	1/2 CVC VCT
Fault Tree	=RBWMS_RCV_D2	2/2 RBWMS avail for CVCS dilution
Fault Tree	=CVCS_2RCSMAKEUP	Condition de la RI _2RCSMAKEUP de EIRM verifiee
Fault Tree	=CVC__06	
Function Event	CVC06	2/2 CVC BORAT
Fault Tree	=CVCS_1RCSMAKEUP	Condition de la RI _1RCSMAKEUP de EIRM verifiee
Fault Tree	=CVC__12	
Function Event	CVC12	1/2 CVC BORATION
Event Tree	-WSWBS-BO1-A	ATWS - WSWBS - Boration control - states A1 A2
Event Tree	-WSLSV-BO1-A	ATWS - WSLSV - Boration control - states A1 A2
Event Tree	-WSLCP-BO1-A	ATWS - WSLCP - Boration control - states A1 A2
Event Tree	-WSLMF-BO1-A	ATWS - WSLMF - Boration control - states A1 A2
Event Tree	-WSTT_-BO1-A	ATWS - WSTT - Boration control - states A1 A2

Figure 8.12. Call Hierachy: The call hierarchy permits to explore backward references of a module.

8.3 Using Graphs to Visualize Systems

PSA models at EDF target to encode the risk of physical systems (nuclear power plants). Those critical systems may be specified in another modeling language. At EDF, the software *KB3* [64] is used to specify systems, which is based on the modeling language *FIGARO* [42].

System models can provide mandatory information to understand the content of PSA models. Unfortunately, it is currently often difficult to exploit information from system models (see Section 8.3.1).

This section presents how graph models can be used to compose physical systems and to perform safety

analysis. Graph models are capable to reflect architectural information of physical systems. At the same time they can serve to reveal safety problems. Thus, they can serve as a bridge between system and safety engineers.

An article to use graph models for safety assessment has been published in [65].

8.3.1 Problem Context

The value of exploiting system models is to understand the composition and relations of system components. This helps engineers to develop and validate PSA models.

Unfortunately, system engineering is often strictly separated from risk assessment. Consequently, models and software may differ, even though generation interfaces (methods to generate parts of risk models from system models) may exist. Due to technical or political reasons, system models may not be available for safety engineers.

Further, system models are typically composed in dedicated DSL languages (domain specific languages), which may contain specific constructs of system engineering. Though system engineers are familiar with those constructs, PSA engineers may have problems to understand those.

A new concept is needed to abstract from the complexity of system models and to present system information to safety engineers.

8.3.2 Graph Models

The idea is to use undirected graphs to model physical systems.

A graph (as defined in the scope of the thesis) is a 5-tuple (NS, ES, CS, S, T) whereas

- NS is a set of *nodes*
- ES a set of *edges*
- CS a set of *components*
- $S \in NS$ is the *source*
- $T \in NS$ is the *target*

Edges are used to connect nodes. They represent relations between physical system components. An edge is a tuple (N_1, N_2) whereas N_1 and N_2 are nodes. Let (N_1, N_2) be an edge of a graph (NS, ES, CS, S, T), then: $N_1 \in NS \wedge N_2 \in NS$.

Nodes can be (but not necessarily) *linked* to components. Components are typically physical components of a system, e.g valves, pumps etc. Different nodes may be linked to the same physical component. Components have unique names.

Let $c : Nodes \rightarrow Components \cup \{\perp\}$ be the function that returns the linked component. If $c(N) = \perp$ of a node N, then N is said *unlinked*.

Let (NS, ES, CS, S, T) be a graph and $N \in NS$, then $c(N) \in CS \vee c(N) = \perp$.

The meaning of graph source and target is explained in a little while.

Graphs can be visualized. Table 8.2 lists the graphical symbols of a graph model.

Figure 8.13 shows an example of a system model (left side) and its corresponding graph model (right side).

Table 8.2. Graphical symbols in a graph

Symbol	Name	Description
	Source Node	Represents the source node of a graph.
	Target Node	Represents the target node of a graph.
	Node	Represents any graph nodes apart from source and target.
	Edge	Represents a graph edge.
	Component	Represents a graph component whereas "Name" is the component name. Component names are written above nodes.

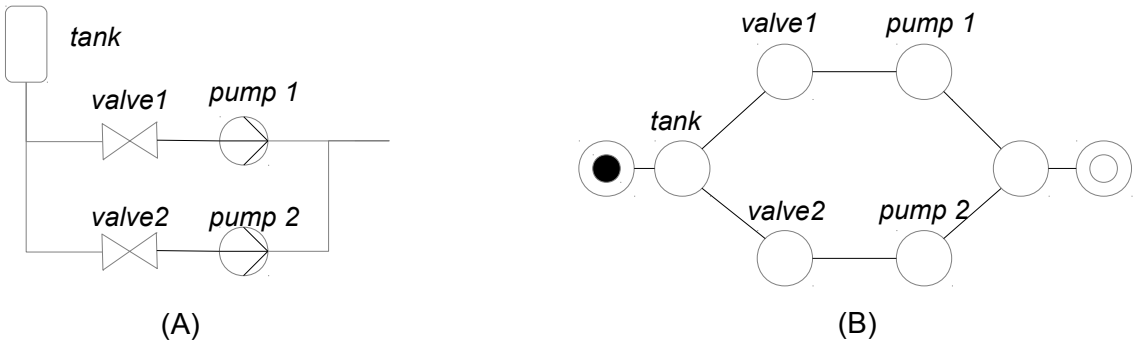


Figure 8.13. Graph model of a hydraulic system: (A) A technical system. (B) The corresponding graph model for the system.

Reliability Definition

Components are assigned to a *state*, which is either *working* or $failing$. Let σ be a state assignment. The state of a component C is retrieved by $\sigma(C)$.

Let Σ be the set of all state assignments.

A *path* of a graph (NS, ES, CS, S, T) between two nodes $A \in NS$ and $B \in NS$ is a sequence of edges $[N_1, N_2, \ldots N_k]$, whereas

- $N_1 = A$
- $N_k = B$
- $(N_i, N_{i+1}) \in ES$
- $c(N_i) = \bot \ \vee \ c(N_i)$ is *working*

The *length* of the path $[N_1, N_2, \ldots N_k]$ is k.

Two nodes A and B are said *connected*, if there is at least one path between them. Otherwise the nodes are said *disconnected*.

A graph with source S and target node T is said *connected* if S and T are connected. Otherwise it is said *disconnected*.

A system is considered *working* in case its corresponding graph is *connected*. A system is $failing$ if it is not *working*.

Cutsets

A *cutset* of a graph $G = (NS, ES, CS, S, T)$ is a set of components $Cut = \{C_1, C_2, \ldots, C_n\}, C_i \in CS$ such that

$$\forall \sigma \in \Sigma : (\forall C_i \in Cut : \sigma(C_i) = failing) \Rightarrow G \text{ is } disconnected$$

Let G be a graph with cutsets $CutS$. A cutset $Cut_1 \in CutS$ is said *minimal* for G, if $\nexists Cut_2 \in CutS : Cut_2 \subset Cut_1$.

Cutsets of order 1 are also referred to as *single point of failure*. They are regarded to be especially safety critical as the failure of one single component leads to an erroneous system.

Figure 8.14 shows an example of a cutset. The full list of cutsets of the example is: {*tank*}, {*valve1, valve2*},{*valve1, pump2*}, {*pump1, valve2*}, {*pump1, pump2*}. Cutset {*tank*} is a single point of failure.

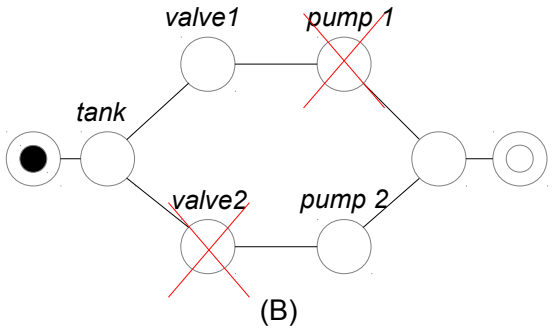


Figure 8.14. Example of a cutset: Failing components are indicated by red crosses. The graph is disconnected.

A system is said *always working* if its corresponding graph has no cutsets.

Likewise, a system is said *always failing* if its corresponding graph with components CS has the cutsets 2^{CS}. The only minimal cutset of an always failing system is {} (the "empty cutset"),

8.3.3 Modeling Systems of Systems

Systems are often composed of further systems, so-called "subsystems". In fault trees (which typically represent physical systems) this hierarchical aspect exists in the form of external gates. Those refer to gates of other fault trees (the subsystems). However, the fault tree language is not really adequate to understand hierarchical system aspects (it is too "low level").

In order to model systems of systems, components can be linked to graph models.

Let $g : Components \rightarrow Graphs \cup \{\perp\}$ be the function that returns the linked graph of a component. If $g(C) = \perp$ for a component C, then C represents an individual component and is said *terminal*. If $g(C) = G$, then C represents a system which is modeled by a graph G and is said *non terminal*.

Let G_1, G_2 be two graphs and N a node of G_1. G_2 is called a *sub graph* of G_1 if $G_2 = g(c(N))$.

Let $GS = \{G_1, G_2, \ldots, G_n\}$ be a set of graph models. A graph G is called *top graph* respective the set if $\forall G_i \in GS : G$ is not a sub graph of G_i.

Figure 8.15 shows a graph, whose components are linked hierarchically to further graphs.

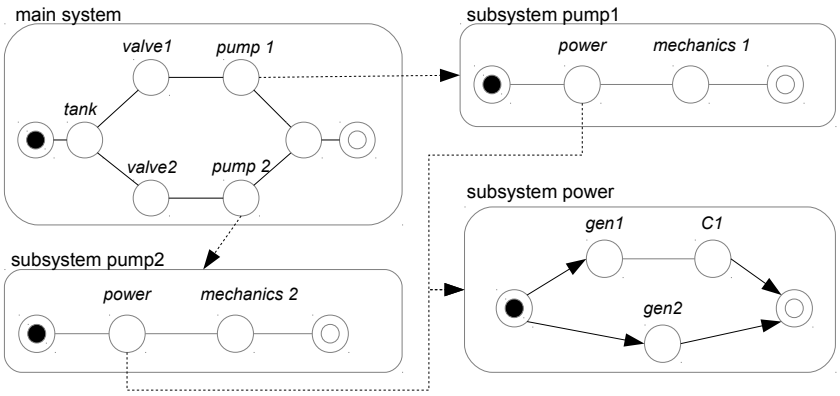


Figure 8.15. Modeling systems of systems: A system consists of two subsystems (for each pump). Those subsystems require in return a subsystem (for power)

Systems cannot be contained in subsystems (Circular dependencies are not allowed):
Let $[G_1, G_2, \ldots, G_n]$ be a sequence of graphs with "G_{i+1} is a sub graph of G_i", then
$\forall i, j, i \neq j : G_i \neq G_j$

Graphs may share same components. In the example, this is the case for subsystems "pump1" and "pump2": Both contain the component "power". Using the same components in different graphs creates dependencies between graphs, i.e. common cause failures.

A component (non terminal) is $failing$, if its linked graph is disconnected: Let C be a component and σ a state assignment with $g(C) = G$. Let further $Cut = \{C_1, C_2, \ldots, C_n\}$ be a cutset of G. The following condition must hold:

$$(\forall C_i \in Cut : \sigma(C_i) = failing) \Longrightarrow \sigma(C) = failing$$

Primary Cutsets

A cutset Cut is a *primary cutset* for a graph $G = (NS, ES, CS, S, T)$ if

- Cut is a cutset for G and
- $\forall C_i \in Cut : C_i \in CS$

A primary cutset may contain non terminal components.

Final Cutsets

A cutset Cut is a *final cutset* for G if

- Cut is a cutset for G and
- $\forall C_i \in Cut : C_i$ is $terminal$

The final cutsets of a graph G are obtained iteratively by expanding its cutsets by the cutsets of its sub graphs. The expansion is done by combinatorial multiplication:

Let $\{C_1, C_2, \ldots, C_n\}$ be a cutset of G and C_1 be linked to a graph with cutsets $\{Cs_1, Cs_2, \ldots, Cs_m\}$, then

$Cs_1 \cup \{C_2, C_3, \ldots, C_n\}$,
$Cs_2 \cup \{C_2, C_3, \ldots, C_n\}$,
...
$Cs_m \cup \{C_2, C_3, \ldots, C_n\}$

are cutsets of G (though potentially not minimal). The minimal cutsets are obtained by discarding all non-minimal cutsets.

The start of iteration is given by the primary cutsets of graphs. Those are then expanded.

The iteration ends when the cutset list solely contains final cutsets.

The algorithm to determine final cutsets is computational challenging, in particular the 1) determination of primary cutsets, 2) expansion of cutset lists and 3) discarding of non minimal cutsets.

Stepwise Cutset Analysis

Analysts may sometimes not want to obtain immediately the final cutsets (containing terminal components). It can be of interest to perform a stepwise analysis of primary cutsets in order to analyse a system hierarchically from its "top" to its "bottom".

The procedure is the following: First, the primary cutsets of a top graph are evaluated. The obtained result may contain non terminal components. The primary cutsets can be visualized in the top graph without having to deal with "details" from sub graphs.

Then, sub graphs linked to interesting components (those which occur in primary cutsets) are analysed. Then sub graphs of sub graphs are analysed and so on.

The stepwise approach helps to understand systems on different levels, whereas the level of detail is incrementally increased.

8.3.4 Application

A new domain type (called the "Graph Domain") has been developed to extend Andromeda by graph models. Figure 8.16 shows an example of a graph model in Andromeda. The source and target nodes are colored in green and red color rather than using the graphical symbols introduces in Table 8.2. The reason is that in large graphs nodes can become difficult to locate. By using colors, the graph source and target nodes are easy to detect.

Further, a converter has been developed to convert existing system models (developed with the software "KB3"[64]) to graph models in Andromeda.

Andromeda supports both, primary and final cutset analysis. Figure 8.17 shows Andromeda after a primary cutsets analysis has been performed (for the graph model on the right side). The cutset list is shown on the lower right. By clicking on a cutset, the corresponding nodes (nodes linked to a cutset component) are inked automatically in blue color. In the sequel, an engineer could decide to analyse the "blue" nodes.

In the screenshot, other nodes inked in blue color (though not filled) can be noticed. Those indicate so-called "transfer nodes". In Andromeda, a graph model can be split over several "pages" (to reduce the number of nodes per page). If an edge links two nodes of different pages, those nodes are considered as "transfer nodes" and inked in blue color (not filled). However, transfer nodes concern only the visualization method. They have no further impact. Also, pages are not to confuse with subgraphs which define a new

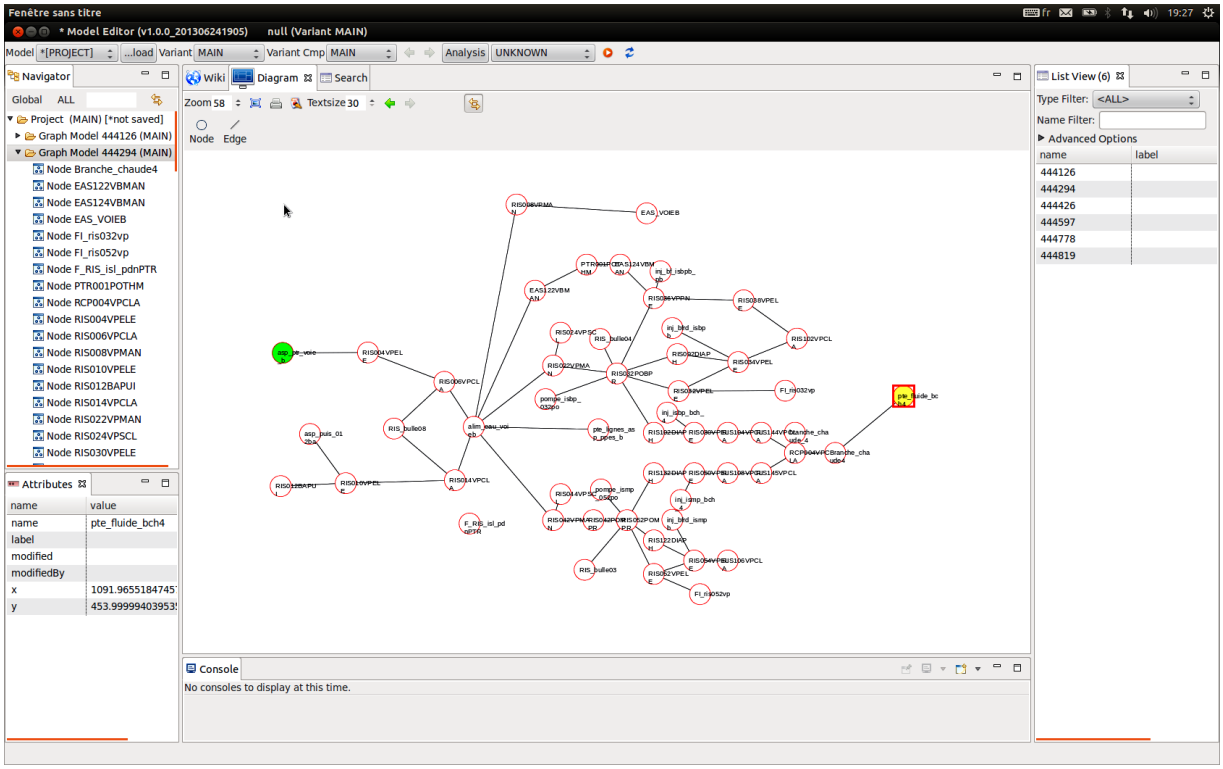


Figure 8.16. Graph models in Andromeda

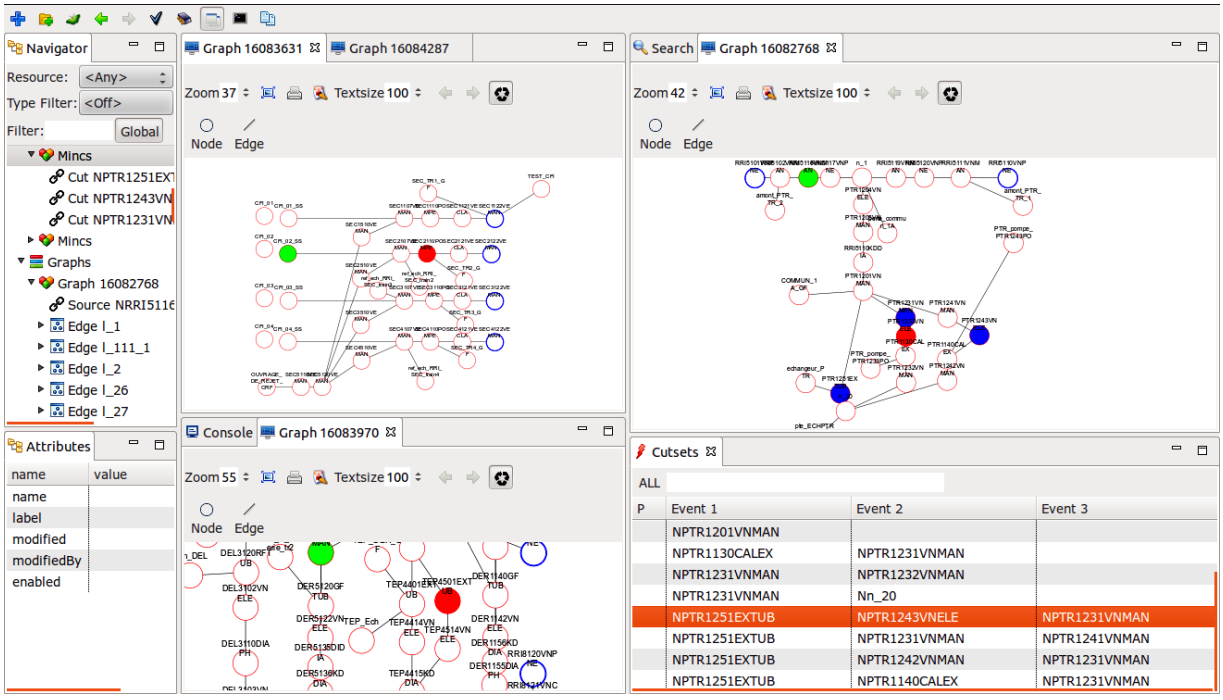


Figure 8.17. Cutset Visualization in Andromeda

graph in order to express reliability of certain nodes (whereas pages belong to the same graph).

8.3.5 Limitation

The presented form of graphs targets to provide an initial work that should be analyzed further. In order to model more complicated systems which can for example contain "K/N" behavior (minimum K out of N failing nodes will consequently fail another node), achievements obtained from "*Digraphs*" [66] or functional block diagrams [67] should be taken into account in order to enhance the current definition of graphs.

The difficulty thereby is to find a minimum set of constructs appropriate for modeling a large variety of systems and not to get too specific to certain system architectures (otherwise the benefit to have a very simple and unique language would get lost).

8.4 Visualization Techniques

In this section, some visualization techniques for PSA models are presented. Visualization is a graphical method that can ease to filter information to concentrate on pertinent model aspects.

8.4.1 Color Inking

Color inking uses colors to highlight pertinent information. Humans are specially capable to recognize graphical elements that are inked by colors, even if those are embedded in a much more complex scenario.

In Figure 8.18, an example from Andromeda is shown to ink different aspects of event trees. An engineer has selected (clicked) on a consequence in a PSA model. Automatically, the event tree highlights different aspects:

- Selected element: The selected element (the consequence) is inked in yellow color. In Andromeda, this behavior is consistent: Any time the selected element changes, all diagrams and other views indicate automatically the selected element by using the yellow color.
  The consequence occurs several times in the event tree diagram, whereas the occurrence the user actually clicked on is additionally framed in red color.
- Failing and successful mitigation I: Function events that have certainly failed (in all consequence paths) are inked in red color. Those applied successfully (in all consequence paths) are inked in green color. Orange color is used to ink those failing in a subset of all consequence paths. Function events that are not applied (regarding the selected consequence) are not inked.
- Failing and successful mitigation II: Any consequence path leading to the selected consequence is inked in colors. Red sections indicate failing mitigation. Green sections indicate successful mitigation.

Another color inking example is shown in Figure 8.19 to highlight the different sequences and consequences of a successful or failing mitigation action. The mitigation action is given by the function event a user has currently selected (clicked on).

8.4.2 Eliding Diagrams

Sometimes, fault- and event tree diagrams are too large to be displayed adequately. It can become challenging to retrieve information from them. The idea of eliding diagrams is to display only subsets of diagrams in order to reduce their sizes and to focus on specific parts.

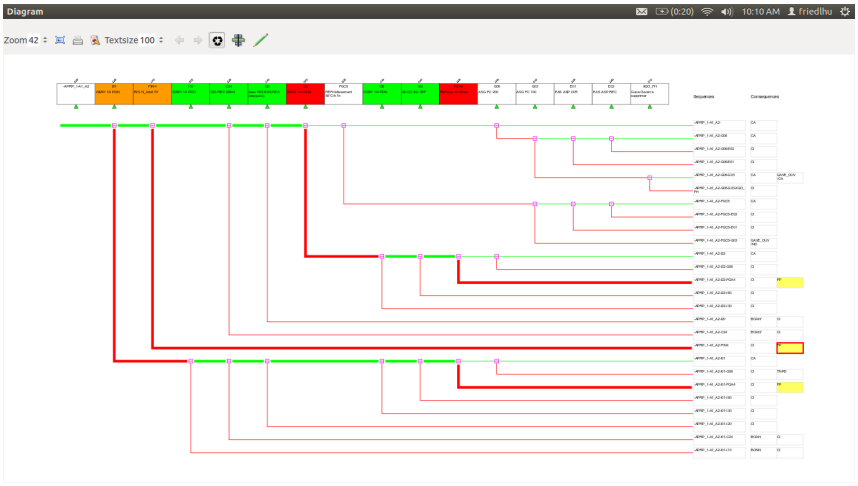


Figure 8.18. Color inking in event trees I: The mitigation respective a certain consequence is inked in colors.

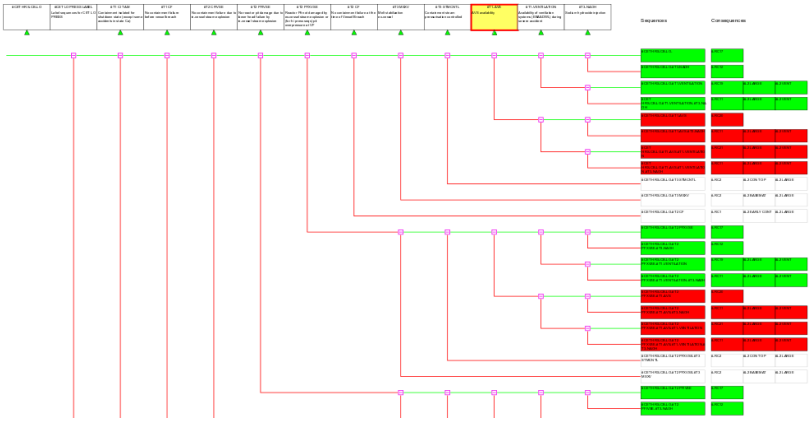


Figure 8.19. Color inking in event trees II: The consequences respective a successful / failing mitigation are inked in colors.

For example, fault trees can be "minimized" at gate level. A minimized gate shows solely the gate symbol and a special symbol indicating the minimized status without developing the fault tree any further. Figure 8.20 shows an example of a minimized gate in Andromeda.

Likewise, event tree diagrams can be minimized at "fork" level (a fork indicates the application of a function event). A minimized fork shows only the first path which is typically the "success path". All other paths (typically the "failure paths") are not developed any further. Figure 8.21 shows an example.

The user interface for minimizing diagrams is rather "simple": By clicking on the minimize / maximize symbol (the symbol indicating the minimization status of gates / forks), the diagrams are displayed accordingly.

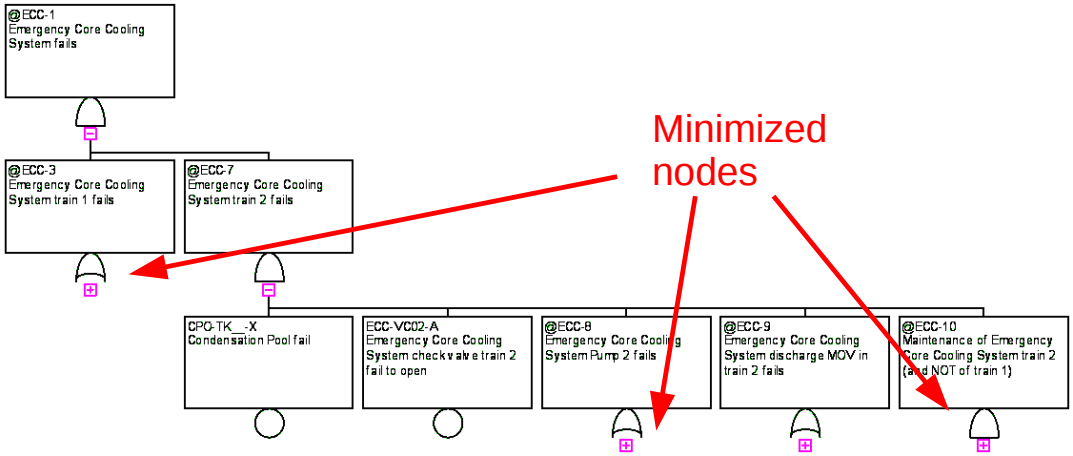


Figure 8.20. Minimizing gates in fault trees: Subtrees can be hidden by minimizing gates.

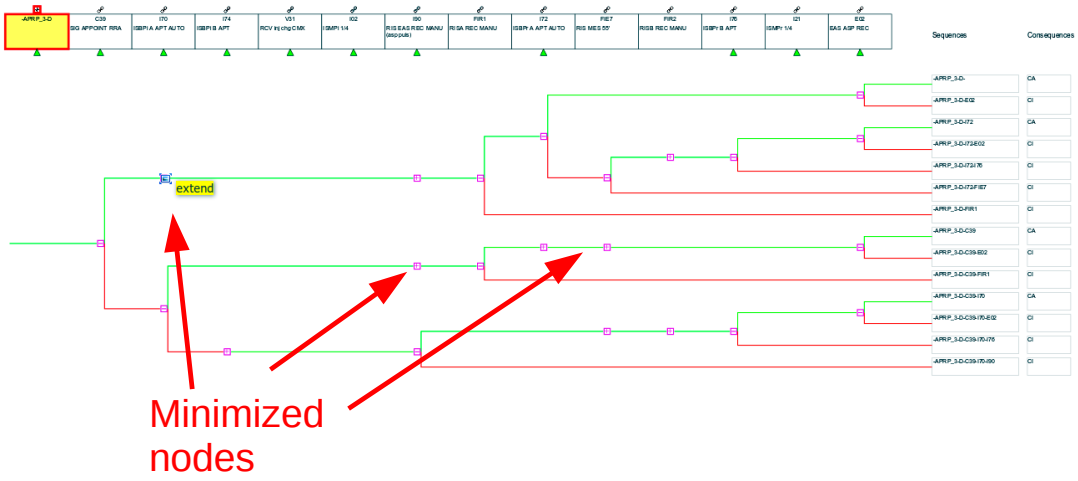


Figure 8.21. Minimizing forks in event trees: Failure paths can be hidden by minimizing forks.

8.5 Conclusion

In this chapter, some counter methods have been presented to support model engineers to understand complex models.

The presented idea to develop a decentralized documentation network can enhance the value of model documentation. The benefit of model documentation is not limited to provide information. The approach showed that PSA software functionality can be embedded in model documentation to execute any kind of operations for example to navigate models or to open diagrams.

The analysis and visualization of dependencies has been characterized to play a key role for reducing model complexity. In this context, a cartography method could illustrate how dependency analysis can support model engineers to understand the relation between objects. In a modular PSA, engineers can

navigate along module dependencies to iteratively explore the composition of models.

The usage of graph models for safety assessment has been demonstrated. The advantage of graphs is their simple composition, their ability to reflect the composition of physical systems and their capacity to analyse systems of systems: Starting from a top system, safety critical information (cutsets) is hierarchically derived for subsystems. The level of analysis detail becomes configurable.

Sometimes, not only PSA models but also individual fault- and event trees are complex and large. A method to elide diagram parts could illustrate how large diagrams can be handled in a compact manner. Further, diagram parts can be inked with different colors to visualize pertinent information. An interactive color inking method for event trees could ink event tree parts in dependence to the current focus of a model engineer.

CHAPTER 9

Model Development Process

In future, models (not only PSA models) are supposed to become more complex. New strategies to integrate models over a set of model assets are preferable, whereas some parts may be even outsourced. These new modeling strategies pose new constraints on collaborations between model engineers, processes and management.

In this context, the term *model development process* (MDP) is introduced. A MDP describes the different activities and tasks of model engineering. The term is derived from the term "software development process" (SDP), which has become a famous organizational instrument to describe the different phases of software engineering [68, 69].

In this chapter, the different phases of a MDP are described (model design, creation, integration, test, maintenance etc.). Further, different concepts, processes and strategies are described in detail, which can be considered in the different phases.

Section 9.1 describes the different phases of the model engineering process in a modular PSA.

Section 9.2 presents the modeling language of event sequence diagrams, which can be used during design phase to specify event trees.

Section 9.3 presents a scripting interface for a modular PSA in order to automate certain modeling activities.

Section 9.4 introduces to the idea of concurrent model engineering.

Section 9.5 shows how PSA quantification results of existing PSA software can be verified.

Section 9.6 concludes this chapter.

9.1 MDP Phases in a modular PSA

A model development process (MDP) describes the different phases of model engineering. In each phase, a set of activities and tasks is precised. The term "model development life-cycle" (MDLC) can be equally used to MDP.

An example of a MDP is the waterfall model. This process origins from software engineering, but can be transformed to model engineering (see Figure 9.1).

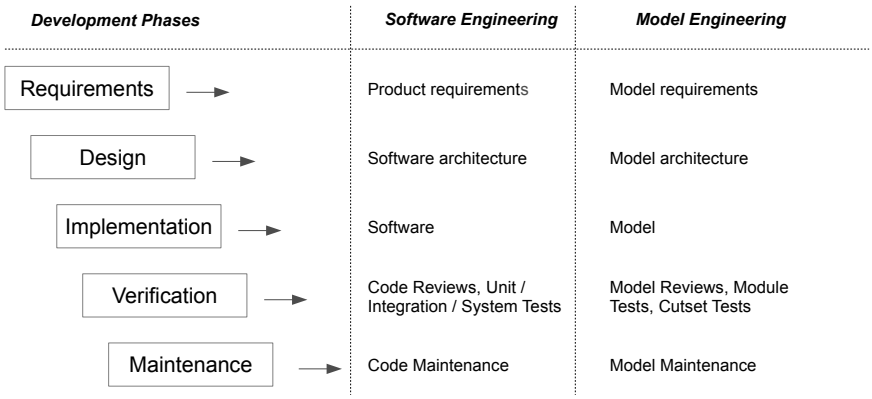


Figure 9.1. The waterfall applied to the development of PSA models

A MDP can be part of an overall model life-cycle plan, which covers beside the MDP also financial, management, marketing etc. issues. It can further be a subset of a system development life-cycle (for example to certify systems).

In the sequel, the phases to engineer PSA models in a modular PSA for nuclear plants are described. However, no exhaustive description can be given: A MDP must consider specific constraints and contexts of projects and companies.

9.1.1 Requirements Engineering

A requirement is a constraint or a general need which is to satisfy at PSA model engineering. It may concern the PSA model itself (functional requirement) or the manner PSA models are engineered (non-functional requirement).

Requirements are collected at the planning stage of a PSA project (before the actual PSA model is created). The process to identify, protocol, validate and manage requirements is called *requirement engineering*. Requirement engineering is important to avoid subsequent inconsistencies of PSA models at their quantification. It is critical to the success of a PSA model project.

It should be recognized that the intended applications of PSA may impose additional requirements on the scope of the PSA, on the modelling approaches and on the level of detail [70].

Fundamentals about requirement engineering are described in [71].

At EDF, requirements are coming from the French regulator ASN (Autorité de Sûreté Nucléaire), but also from EDF itself (internal requirements).

Examples of PSA model engineering requirements:

- Scope of the PSA [70, p.15]:
  - PSA Level (level one, two, three)
  - List of initiating events
  - List of hazards
  - List of operational modes (i.e. full power, low power or shutdown states)
  - Detail of modeling
- General model characteristics and model quality:
  - Maximum number of gates per fault tree.
  - Maximum number of sequences and function events per event tree.
  - Prohibition of certain modeling constructs which complicate models.
  - Choice of FTL (fault tree linking) or ETL (event tree linking) approach 2.1.3
- Software requirements:
  - PSA software tools to use (for modeling and quantification).
  - Interaction between tools.
  - Computational constraints. Example: quantification of PSA model should terminate within 24 hours.
- Cooperation policy (models can be engineered by different groups / companies):
  - clarification of responsibilities of the different model parts
  - chronological coordination (scheduling, release strategy)

Remark: Typically, control authorities constrain PSA quantification results: For example, frequency values for CDF (Core Damage frequency, level one PSA) and LERF (Large Early Release Frequency, level two PSA) hazards must not pass certain maximum frequency levels. However, a system can obviously not be repaired / improved by just creating a model that fits those requirements. Those requirements are not considered as modeling requirements, but as system requirements. If they are not hold, the system design - not the model - fails.

9.1.2 Model Design

In this phase, the principle composition, structure and organization of models and their objects (modules) of a modular PSA are described:

- Model policy: In a modular PSA, models can be obtained from existing ones by extending them (see Section 7.1.3) or they can be created from scratch (anew). Also, certain models may be commonly used (i.e. they are shared with other modular PSA projects). The model policy must be coherent with the cooperation policy (see Section 9.1.1) and the instantiation policy (see below).
- Module organization: In a modular PSA, modules can be organized in folders and different models. It is necessary to identify which model objects should be grouped together. For example, one can organize modules according to their hazard type (flooding, fire etc.). But one may also consider to group them hierarchically, i.e. to introduce subfolders to categorize component types (basic events, fault trees, event trees etc), to group very likely / unlikely events, to group recently created components (which are still to review) etc.
- Context determination: In which context(s) is the modular PSA to instantiate? A context can express plant specific PSA models (variants). A context can also express different aspects or parts of a PSA model, e.g. fire PSA, flooding PSA etc.
- Design of event trees. In Section 9.2, a formal language to design event trees in the form of event sequence diagrams, is presented.
- Design of fault trees. For each mitigating event (function event) of event trees, typically a fault tree is associated to model a (physical) component failure. In the design phase, the list of fault trees and

their modularization (fault trees may "use" further fault trees via external gates) is to determine.

A good design can promote a clear model structure in the modular PSA. It plays a key role to keep PSA models well organized, readable and understandable.

9.1.3 Model Development

The model development phase (or implementation phase) concerns the actual development of PSA models, i.e. the development of fault and event trees.

Modeling concepts and strategies to develop models in a modular PSA have been presented in Section 7.

One challenge of this phase is the synchronization of different development actions (from possibly different model engineers). The problem is known under the term *concurrent model engineering* and presented in 9.4.

9.1.4 Model Integration

Concurrent model engineering requires further to fusion (to merge) the different development actions. This activity is handled in the model integration phase.

Typically an "integrator" is in charge to merge models. For more information, see Section 9.4.

The model integration phase can be part of the model development phase (this depends on whether the integration activities play a central role or not).

9.1.5 Model Verification

The verification phase targets to verify whether the requirements (of the requirements phase) are satisfied or not.

Different methods exist to perform this validation. Those are presented in the sequel.

Reviews

Reviews concern the inspection of PSA models by experienced persons to validate mainly quality of models for the purpose of QA (Quality Assurance).

They can be official (to justify QA to safety authorities) or unofficial (to improve quality of models). A review process is performed by one or multiple reviewers. Reviewers can create quality reports, which serve as important input for model engineers to improve model quality.

Model Tests in General

Testing a model is to check whether a model conforms to a set of constraints. It is used to proof semantic and structural characteristics of models. It is further a diagnosis tool.

Model tests can for example serve to detect modeling errors:

1. Check whether model modifications have been applied correctly.
2. Check whether the modification of certain modules does not lead to any inconsistencies of other model parts (regression tests).

Tests are performed via the execution of so-called "test cases". The procedure that is applied at execution, is called the "test procedure" or simply the "test". The execution of a test case delivers a "test result".

Each test case defines:

- Test input: Any data, comprising models and their modules, the instantiation context and additional test data (for example expected results) needed by the test procedure to perform the test.
- Expected result: The expected result represents a set of constraints. A test is successful, in case the test result holds the set of constraints.
  For example, to validate a fault tree for not containing forbidden event combinations, its minimal cut sets (the test result) are validated to not contain any of the forbidden combinations (the constraints).

A set of test cases can be grouped to a test list. A test list is considered to be successful, if the number of failing test cases does not exceed a certain limit (an acceptance level). The limit can be stated absolutely or relatively to the number of total test cases. If the limit is zero, all test cases are required to succeed.

Regression Tests and Side Effects

In complex models, the relation between modules is not always obvious. Modifying a module may impact dependent modules in an unexpected manner. Regression tests ensure, that model modifications do not unintentionally "destroy" other model parts, which have already been considered as "working".

One reason, why relations between dependent model objects are sometimes not clear, are "side effects". Side effects refer to model artifacts encoding a behavior that is beyond the usual intended one. For example, to skip a known software failure of a quantification engine, one may create additional model constructs (for example fault tree gates) to avoid the software problem.

Side effects are regarded as bad modeling practices. They are difficult to understand and maintain if not well documented and they lead easily to unexpected regressions. However, in certain situation, they are unavoidable, because things would otherwise get very complicated. But they should be used with care, most rarely as possible and always together with meaningful documentation and regression tests.

Module Tests

Module tests validate individual modules. To test a module apart from others permits to confine the source of potential errors.

In particular in large PSA models, this kind of test can be important for an efficient identification of erroneous modules.

An example: The quantification of a PSA model delivers curious minimal cut sets. There is no hint from the quantification software. The identification of the problematic fault or event trees in the overall model seems impossible (PSA models can contain several thousands of components). On the contrary, module tests verify a model by testing its parts. The tests can be automated and executed in parallel (to increase efficiency).

A failing module test does not necessarily mean that the tested module is erroneous: The tested module may have dependencies to erroneous modules which fail the test.

In order to test an individual module without its dependent modules, so-called "test stubs" can be considered. Test stubs are artificial modules which exist only during the test phase.

Figure 9.2 illustrates the principle: Test stubs are used to replace any dependent modules of the module

to test. They have no dependencies to other modules (this is why they are called "stubs").

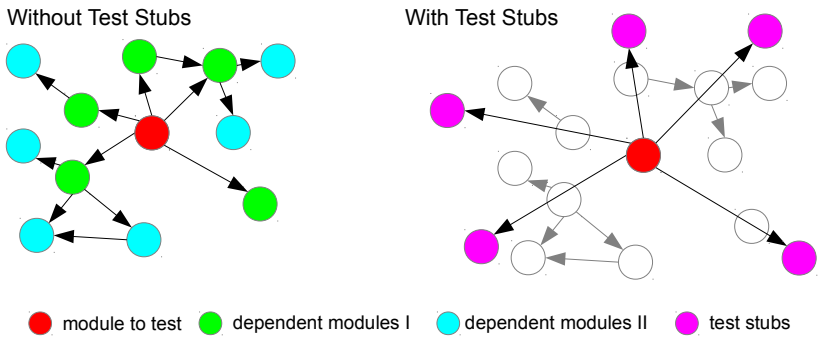


Figure 9.2. Test Stubs: Test stubs substitute dependent modules of the module to test. They have no dependencies.

In a modular PSA, the substitution of modules by test stubs is easily achieved in two steps:

1. A new model (test stub model) is created containing the test stubs.
2. A new context (test context) is created to let the module "see" the test stubs (a replacing include has to be defined to resolve the references of the module to test in the test stub model, see Section 7.1.3).

In addition, test stubs can be flavored to test a module exhaustively.

Integration Tests

Integration tests are tests that are performed once module tests have been successfully performed. In contrast to module tests, they concern whole model parts (if not a whole model). They test the interaction between modules after integration.

Even if each module is consistent in itself (module tests are successful), the interaction between modules may not work as expected.

Integration tests diagnose problems of the instantiation phase of a modular PSA (see Section 6.1.4). They reveal problems of reference resolution and context specifications.

Example: An integration test can verify whether a PSA model in context "Bugey" (Bugey is French nuclear power plant) contains basic events specific to the nuclear plant of "Chinon" (another French nuclear power plant). Any occurrence indicates a potential integration problem.

9.1.6 Model Maintenance

Little research on model maintenance has been forthcoming.

Not so for software / system maintenance. Swanson proposed initially three maintenance categories in [72] for software maintenance. Adapted to model engineering, they can be formalized in the following manner:

- Corrective maintenance: Reactive modification of a model performed after delivery to correct discovered problems.
- Adaptive maintenance: Modification of a model performed after delivery to keep the model usable in a changed or changing environment.

- Perfective maintenance: Modification of a model after delivery to improve performance or maintainability.

9.2 Using ESDs to design Event Trees

At EDF, event sequence diagrams (ESD) play an important role: They serve as initial specification to develop accidental sequences of events (event evolutions). Model engineers consider them in a preliminary step before developing the actual event trees of PSA models. Thus, ESDs can be considered in a design phase respective event trees. At EDF, they are moreover known as AQS diagrams (from French "Analyse Qualitative de Sequences", i.e. qualitative sequence analysis). In Appendix A.3, some examples of "real" ESDs are shown.

However, ESDs at EDF are currently not based on a formal language. They are rather graphical images without clear semantics. Textual descriptions complete behavior or constraints not covered in the diagrams. Consequently, ESDs cannot be processed by PSA software tools.

To base ESDs on a formal language is mandatory to enhance the methodology. Software tools could process ESD models in order to:

- Generate event trees from ESD models.
- Optimize event trees to some constraints (for example to respect a specific order of function events or to minimize the number of sequences).
- Check consistency between event trees and ESD models (validate whether event trees conform to their corresponding ESD specification)
- Generate ESDs from even higher level languages (event evolutions may be derived from further models)
- Programmatically develop ESDs by computer programs or scripts (see Section 9.3).

In this section, a framework for event sequence diagrams is presented. The framework is adapted to the needs of EDF and to derive event trees from ESDs. Preliminary work has been realized by S. Swaminathan et al. in [73, 74], though this work has its application mainly in dynamic risk assessment.

9.2.1 Framework of Event Sequence Diagrams

Starting from a so-called **initiating event**, events are recursively explored, till a so-called **end state** is reached. An end state represents a certain system state, which may be acceptable or not. A combination of events (in a certain order) leading to an end state is called a **sequence**. Several sequences may lead to the same **end state**.

A special kind of events are **conditions** and **comments**. Conditions represent Boolean constants: Either they are "True" (condition is satisfied) or "False" (condition is not satisfied). Comments are used to document an event evolution.

ESDs are represented as graphical diagrams. Figure 9.3 shows an example of an ESD: An obstacle occurs in front of a bicycle. In case back and front brakes fail, a crash is unavoidable. If only the back break works, a crash may be avoided (depending on a physical condition: the velocity and obstacle distance must hold some values).

Note that the description "`distance>10 OR velocity<15`" of the condition (in the example) has no further (semantic) meaning, it serves just as information (to a model engineer). Conditions are either "True" or "False" (constant values) no matter their description.

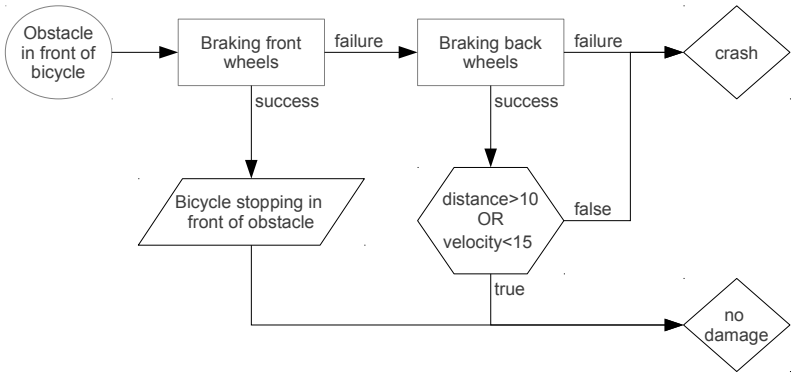


Figure 9.3. Example of an event sequence diagram

The graphical symbols of ESDs are explained in Table 9.1.

Table 9.1. Graphical symbols in ESDs

Element	Name	Description
	Event	An event has two outcomes: "success" (non-occurrence, painted below the event) or "failure" (event occurrence, painted to the right of the event).
	Condition	A constant event with two outcomes: "True" (condition is satisfied, painted below the event) or "False" (condition is not satisfied, painted to the right of the event).
	Initiating Event	The main event provoking the different scenarios.
	Comment	Used for descriptions.
	End State	Final state of a scenario.

The introduced framework does not permit to quantify ESD models. It serves to design event trees as a preliminary step before creating event trees. Reliability data (e.g. probability values for events) is not considered to be part of design. This kind of data gets associated at event tree generation (see Section 9.2.2).

9.2.2 Event Tree Generation

Event trees can be derived from ESD models [75], what is explained in the sequel.

Generation Algorithm

In this section, the algorithm to generate event trees is explained.

Step 1:
First, all sequences of the ESD are derived. Each sequence can be written as $[E_0, E_1, E_2, ..., E_n]$, whereas E_0 is an initiating event and E_n is an end state. Let S_{Set} be the set of all sequences of the ESD.

Step 2:
Next, each ESD sequence is translated into a sequence of an event tree. The translation is done as follows:

- E_0 is replaced by an initiating event (of PSA models).
- E_n is replaced by a consequence.
- for E_i with $0 < i < n$:
  - if E_i is a condition, then it is either replaced by a function event (associated with a house event) or it is "resolved" (see Section 9.2.2).
  - else if E_i is a comment, then it is removed (i.e. comments are ignored).
  - else replace E_i by a function event.

Step 3:
Finally, a unique order "$<_F$" is to determine for the function events in order to display the event tree.

Let $f(E_i)$ be a function that returns the generated function event of E_i or $\perp$ (if no function event has been generated). Let "$<_S$ " be the order over events in a sequence S, then

$$\forall S \in S_{Set}\, \forall E_i \in S, E_j \in S, f(E_i) \neq \perp, f(E_j) \neq \perp : \;\; E_i <_S E_j \Rightarrow \; f(E_i) <_F f(E_j)$$

An adequate order can be obtained by typical sorting algorithms, which are of polynomial complexity (pretty fast). The determination is not unique, i.e. event trees with different ordering (of function events) may express the same ESD.

The generated event tree of the ESD in Figure 9.3 is displayed in Figure 9.4.

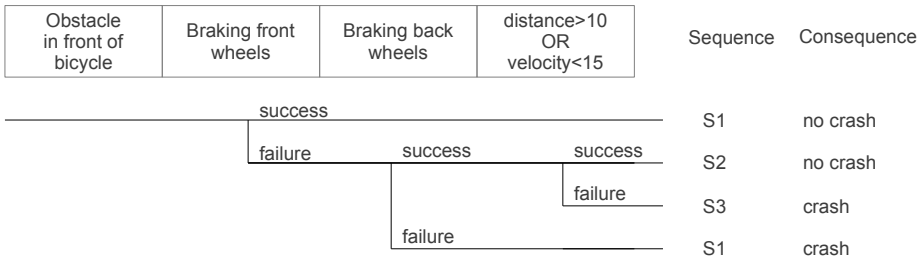


Figure 9.4. Generated event tree

Translation of Conditions

At event tree generation, conditions can be configured to be resolved or not. The configuration of this behavior is supposed to be done by model engineers launching the generation.

If conditions are not resolved, function events are introduced in event trees. They are linked to house events (Boolean flags in PSA models) to configure the satisfaction of conditions.

On the contrary, if conditions are to resolve, they do not occur in event trees. Instead, conditions are checked at generation time. If a condition is satisfied ("True"), the generation continues with the object linked to the "True" outcome of the condition (the event below the condition). Otherwise, the generation considers the object linked to the "False" outcome of the condition (the event to the right of the condition).

Figure 9.5 shows the generated event tree, if conditions are resolved and the condition (velocity and distance) is satisfied.

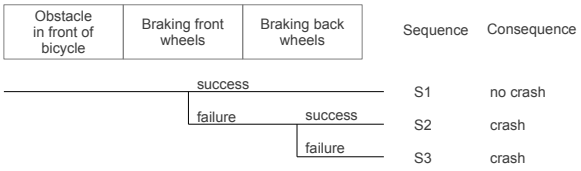


Figure 9.5. Generated event tree - satisfied condition

The generated event tree is not optimized in the following sense: The function event *Breaking back wheels* has no impact on consequences, i.e. it can be removed without changing the event tree semantic. Figure 9.6 shows the result after removing the function event:

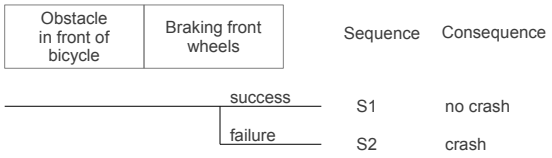


Figure 9.6. Event tree after removing unnecessary function events

Finally, if the condition of the ESD example is not satisfied, the event tree shown in Figure 9.7 is generated.

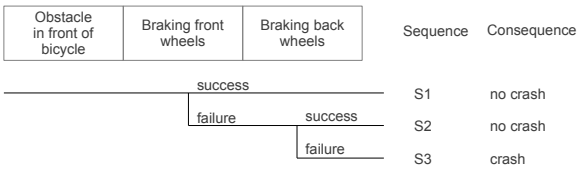


Figure 9.7. Generated event tree - unsatisfied condition

Event Tree Optimization

Typically, the ordering of function events in event trees conform to their chronological application (in time). This kind of ordering is intuitive and easy to understand.

However, often sequence evolutions evolve (partly) similar, i.e. some "patterns" in event trees are redundant leading to all typical maintenance problems of redundancy (modeling becomes time-consuming and error-prone). In addition, the number of sequences gets blown up, what leads to less comprehensible diagrams.

To avoid the problem, one can consider to change the order of function events in a way to treat common characteristics of sequences "earlier" in an event tree. The idea is to share modeling constructs on the left side of an event tree at most between different sequences in order to reduce the total number of sequences.

It is to note that some PSA tools support flag mechanisms (in order to flavor fault trees) when walking along the paths of event trees. Changing the order of function events can impact the flag configurations and thus Boolean formulas.

Two event trees et_1 and et_2 are semantically equal (denoted as $et1 \equiv et2$), if they represent the same

Boolean formula for each of their consequences.

An event tree can be optimized due to several constraints, for example to be "compact": An event tree is compact, if no other semantically equal event tree exists that has a lesser number of sequences: Let Set_E be the set of all event trees and $n(E)$ a function that delivers the number of sequences for an event tree E. An event tree E is compact if and only if $\forall(E' \in Set_E) : E \equiv E' \Rightarrow n(E) <= n(E')$.

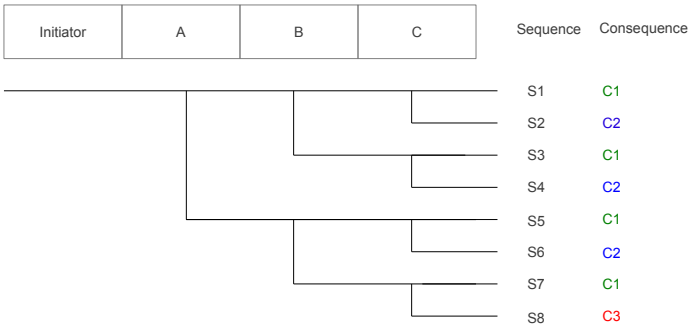


Figure 9.8. Example of a non compact event tree

Figure 9.8 shows an event tree that is not compact. The maximum number of sequences of an event tree is limited by 2^n (except event trees contain alternative paths, so-called "alternatives"), n being the number of function events.

In the example, the maximum number of sequences (given three function events) is 8. The actual number of sequences in the example event tree is 8 as well, i.e. the length of information used to encode the event tree is at a maximum level.

The event tree contains redundant information: The impact of functional event C is contained several times. The following pattern can be found three times:

Four times there is the information that $\neg C$ leads to $C1$, and three times that C leads to $C2$.

Figure 9.9 shows an semantically equivalent event tree, but with a different order of function events. The event tree is compact.

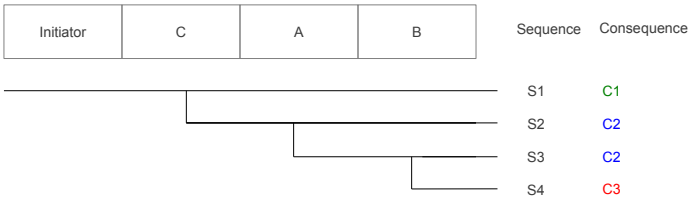


Figure 9.9. Example of a compact event tree

Updating PSA models with ESDs

The procedure to update a PSA model is:

1. Modify ESD model.

2. Generate event trees, consequences, function events and initiating events from ESD model.
3. Import event trees into the PSA model: If an event tree exists already (from an ancient generation), replace it (overwrite it).
4. Import initiating events, function events and consequences into the PSA model, but only those not yet contained (do not overwrite them).
5. For newly imported initiating events and function events: Connect them to reliability data, i.e. link fault trees or basic events which describe their occurrence characteristics.

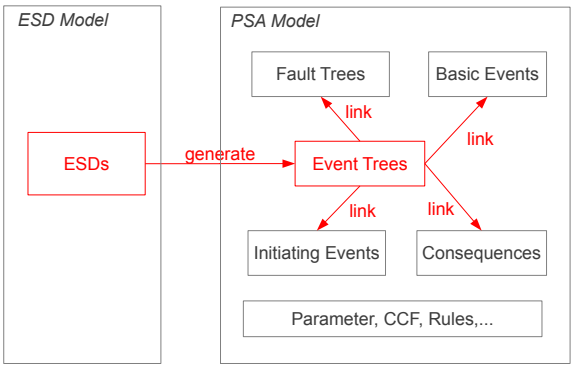


Figure 9.10. Updating a PSA model

An event tree of a modular PSA does not directly contain initiating events, function events or consequences. It contains rather references to them. This facilitates its integration into the PSA model: A generated event tree module is simply copied into the model. Its references are resolved automatically (when a model gets instantiated). This is illustrated in Figure 9.10.

9.2.3 ESDs in Andromeda

To model event sequence diagrams, a new domain type called “ESD” has been developed for the modular PSA framework of Andromeda. An ESD model can store a set of event sequence diagrams.

Figure 9.11 shows an event sequence diagram (left side) and its corresponding event tree (right side) in Andromeda. The event tree has been generated automatically from the diagram.

The ESD model of the example defines three so-called “requirements”. Requirement are used to group a set of events, they have only visualization character.

The events and end states of each requirement are painted “automatically”: Their geographical location is derived from the model structure. The arrows for the “failure” (inked in red color) and “success” (inked in green color) are determined by the means of path planning algorithms (“A-Star” algorithm, an extension of the Dijkstra algorithm [76]). This is to ensure collision free paths between graphical objects.

The advantages of painting ESD diagrams automatically (likewise fault- and event tree diagrams in Andromeda) are twofold:

- ESD diagrams can be elided and expanded (see Section 8.4.2): Model engineers can thus focus on pertinent information.
- A model engineer does not need to spend time to create “beautiful” diagrams.
- Diagrams are also available in case ESDs are created programmatically (e.g. by scripts, see Section 9.3.1).

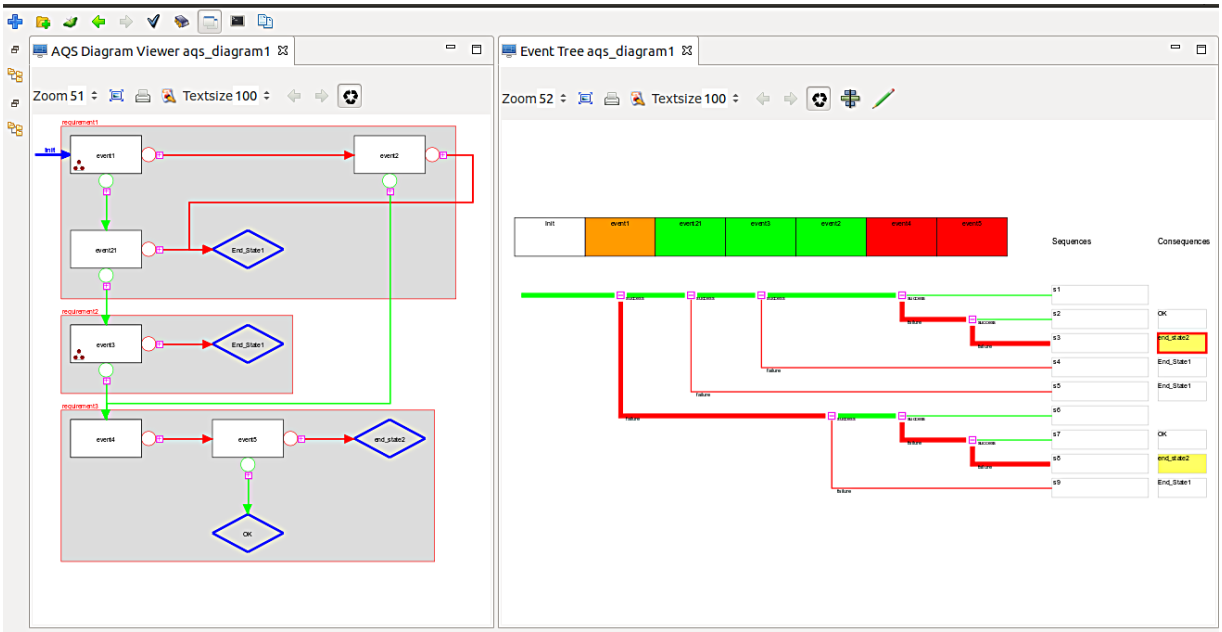


Figure 9.11. An event sequence diagram and its corresponding event tree.

9.2.4 Extension: Combining Event Sequence Diagrams (ESDs) with Graphs

In Section 8.3, graph models have been presented to visualize component failures on a system level.

Typically, events of ESDs represent system failures. ESD models can be extended to associate its events to graph models in order to associate system failures.

ESDs associated to graph models provide a complete design of PSA models: Graph models can be regarded as a preliminary design of fault trees (of component failures). Together with the preliminary design of event trees (the ESD model), a complete design is obtained.

Figure 9.12 illustrates the extended concept: Three events are connected to graphs (which may express failures of physical components). An event occurs if the corresponding graph fails, i.e. if source and target of the graph are disconnected.

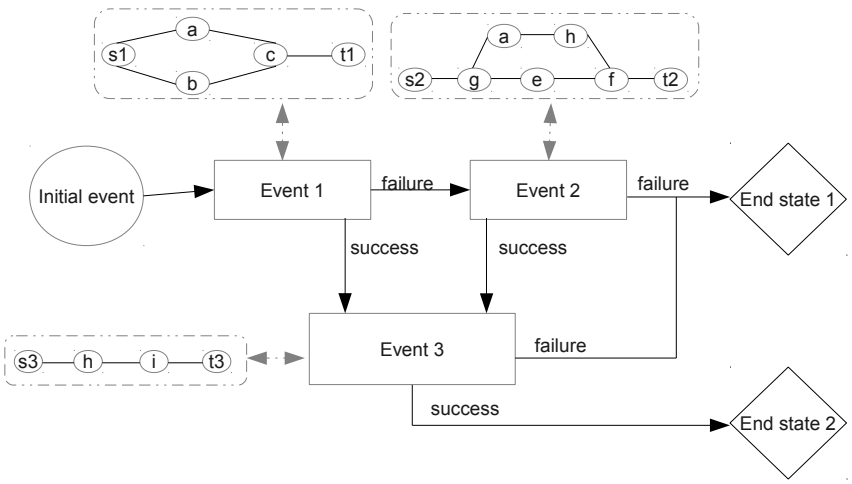


Figure 9.12. Connecting ESDs with Graphs models

The extended concept permits further to determine minimal cut sets from design. Quantification of cut sets is however not possible (as reliability data is not represented in graph models).

In ESD models, minimal cut sets are determined for end states. To determine minimal cut sets for an end state X, the following steps are necessary.

Step 1:
In a first step, the master graph is derived from the ESD model. The master graph encodes any event combinations to reach end state X. For example, if end state X is reached by the two sequences [event1, event2, event3] and [event2, event4], then the corresponding master graph is the graph shown in Figure 9.13. The master graph fails, if one the sequences occurs. The minimal cut sets of the master graph are therefore {event1, event2, event3} and {event2, event4},

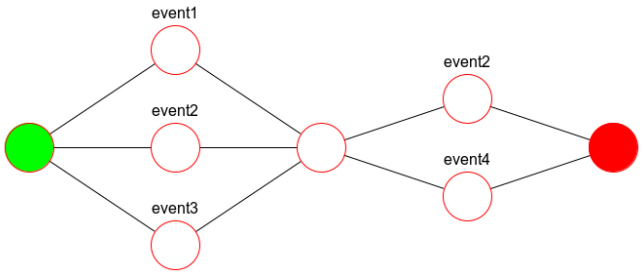


Figure 9.13. Master Graph (Screenshot Andromeda)

Step 2:
The nodes of the master graph are linked with further graphs, each expressing a component failure. The second step consists in expanding the cutsets by combinatorial multiplication: Let $\{e_1, e_2, \ldots, e_n\}$ be a cut set of the master graph and e_1 be linked to a sub graph with cut sets $\{cs_1, cs_2, \ldots, cs_m\}$, then

$cs_1 \cup \{e_2, e_3, \ldots, e_n\}$,
$cs_2 \cup \{e_2, e_3, \ldots, e_n\}$,
...
$cs_m \cup \{e_2, e_3, \ldots, e_n\}$

are cut sets of the master graph (though not necessarily minimal).

Step 3:
The third step it to discard all non minimal cut sets. A cutset cs_1 is minimal respective a cut set list l, if $\forall cs_2 \in l : cs_2 \not\subset cs_1$.

The determination of cut sets is a difficult problem (likewise for PSA models). In particular, step two and three are computational expensive.

9.3 Scripting Interface

Working with full scope PSA models may require appropriate software tools to browse models, to allow their modification, documentation and to perform import/export and merge procedures that cover the needs of version management and validation processes.

In this section, a so-called “scripting interface” is presented to process models in an automated or inter-

active way by the means of scripts. Scripts are small programs developed in an adequate programming language which serves end users (e.g. model engineers) to develop functionality in a higher level programming language.

The introduction of a scripting interface reveals new opportunities to automate processes at model engineering. Scripts are supposed to remain compact, readable and expressive. Their execution is reproducible, debug-able, testable and most important: efficient.

9.3.1 Principle of a Scripting Interface

A scripting interface is associated with various technical notions and concepts. Those are explained in the sequel.

Scripts

A scripting interface permits to execute so-called "scripts". The principle of scripts is illustrated in Figure 9.14: Scripts are computer programs (source code) that are interpreted at run-time (when a script gets executed) rather than compiled into another language (binary code) before execution.

This leads to the advantage that they can be developed and modified without having to (re-) compile a software. Scripts constitute thus an efficient and flexible way for providing software functionality to end users.

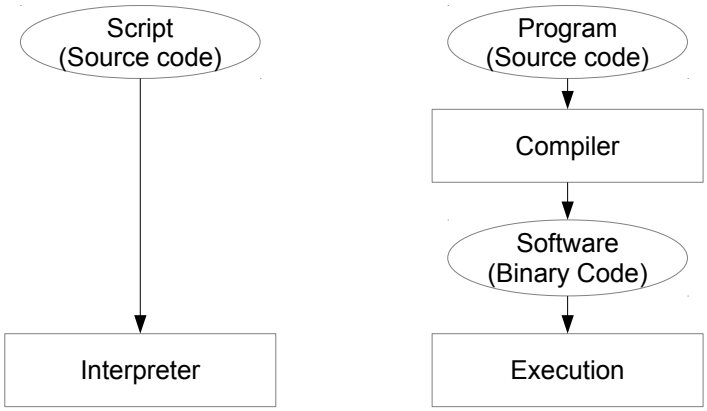


Figure 9.14. Difference between scripts and programs.

Scripts are composed of a set of instructions, specified in a textual format (the "source code"). They conform to an underlying language - the script language. A script language is a high level programming language, which may (or not) support control flow statements (such as "while" loops, "if-then-else" condition blocks etc) and variable usage. Indeed, the distinction between a typical programming language and a scripting language can be blurry.

Example:
To activate any house events related to *ASG* (containing "ASG" in their names), one may consider a script alike

```
forall (House_Event he : house_events)
do
   if (he.name matches /ASG/)
```

```
    do
        he.value := True
    od
od
```

Scripts induce a set of advantages:

- Their execution is efficient.
- Scripts are based on a textual format. Sophisticated editors can be used to simplify the creation of scripts (with features such as syntax highlighting, auto-completion etc.).
- Scripts are debug-able: Breakpoints may be used to diagnose problems while interpreting.
- Scripts are reproducible: They can be applied to a set of models and they can be easily re-launched at a later point in time.

Commands

In a more simple form, scripts become individual instructions, so-called "commands". Commands can be executed interactively (with human intervention) or automatically (without human intervention).

Commands are more or less simple statements in textual form, that consist of

- A command name (mandatory)
- Several arguments 0 to n (optional)

A subset of commands are shell commands which are entered by end users in a shell (see Section 9.3.1) to access system or application functionality.

Commands are entered in a command line interface (CLI) and executed by a command interpreter. Typically, CLIs constitute an interactive user interface (a model engineer can enter and execute commands one by one).

Shells

A shell is a software that provides functionality of a system (for example of an operating system or an application) to end users (for example a model- or safety engineer).

Shells such as the Linux "bash" or Windows "CMD" command line have become popular to interact with operating systems, for example to view, edit and copy files.

Shells can be classified mainly in one of two categories:

- Command line: an interactive CLI serves as interface to execute shell commands (of textual form)
- Graphical: a GUI (Graphical User Interface) serves as interface to provide system functionality.

Graphical interfaces are supposed to be "easy to learn". However, command line shells benefit from a set of advantages:

- Shell commands can easily be reproduced (e.g. for another problem type or at a later point in time). Often shells permit to browse through the ancient command history (e.g. "arrow up" in a Linux shell).
- Shell commands can easily be automated (see 9.3.1).
- To the experience of the author, CLIs are generally faster to develop than GUIs (though some software engineers may disagree).

Batch Processing

Scripts and in particular commands can be automated. The automation consists to execute scripts without human interaction. This mode of execution is also referred to as "batch-mode".

To run scripts and commands in batch-mode is an efficient possibility to enable batch processing: Batch processing is used to perform a series of repetitive tasks, often called "jobs".

In a simple form, each job is launched by one command. Consequently, a series of jobs becomes a sequence of command instructions (a batch program), that are executed one by one (in sequence). In a more complicated form, jobs can be executed in parallel. They may also be scheduled to run at a later point in time to synchronize with results of other jobs or to optimize computational issues (e.g. to balance the "payload" of jobs).

Patches

Scripts can be considered to deliver so-called "patches". A patch describes a sequence of modifications to apply on a model in order to "repair" or "improve" a model in some context.

Model engineers and safety analysts may work at different locations. To send PSA models via Internet can pose problems as models may be confidential data (though encryption is possible). Also, model sizes may exceed maximum sizes of communication mediums (for example emails).

Patches permit a new way of collaboration between model engineers and safety analysts. When safety engineers detect a problem in a PSA model (for example, if its quantification does not work in some sense), they can inform model engineers about the problem. The model engineer examines the problem and delivers a model patch. The safety engineer executes the patch and obtains (hopefully) a corrected model.

9.3.2 Implementation of a Scripting Interface

The presented concept of a scripting interface has been implemented in Andromeda.

Andromeda Shell

The shell of Andromeda is named the *Andromeda Shell*. It has been developed in two versions:

- Integrated Shell: An integrated shell (of Andromeda) is launched directly from inside Andromeda.
- Native Shell: A native shell of operating systems is used as shell (Linux shell "bash", Windows shell "cmd").

Native and integrated shells provide the same set of functionality. Though shell commands are entered textually, the execution of commands can open graphical widgets (windows).

Native Shell The native shell of Andromeda is used by launching Andromeda from command line with option "-cmd shell" (andromeda -cmd shell).

In Figure 9.15 the native shell "bash" (from Linux) is shown in interaction with Andromeda.

Integrated Shell An integrated Shell has been developed in Java for Andromeda (see Figure 9.16). The integrated shell is launched directly in Andromeda.

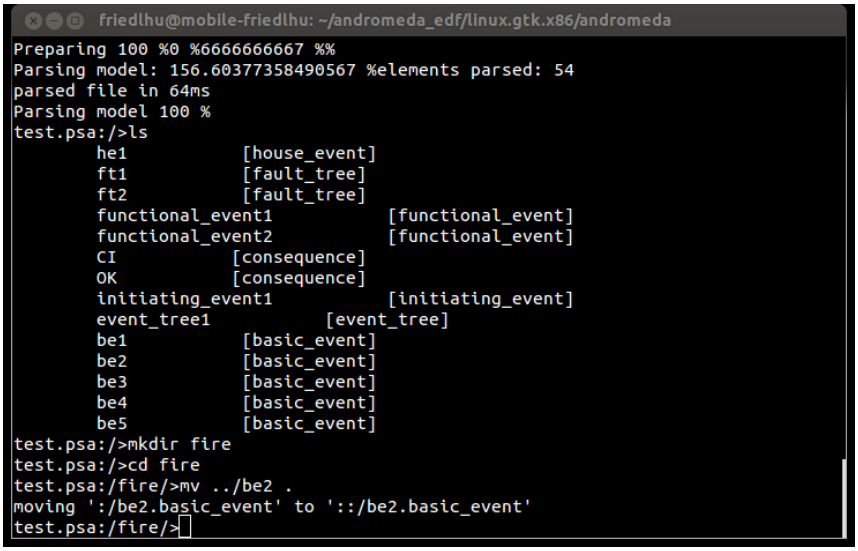


Figure 9.15. Native Shell of Andromeda (Linux "bash")

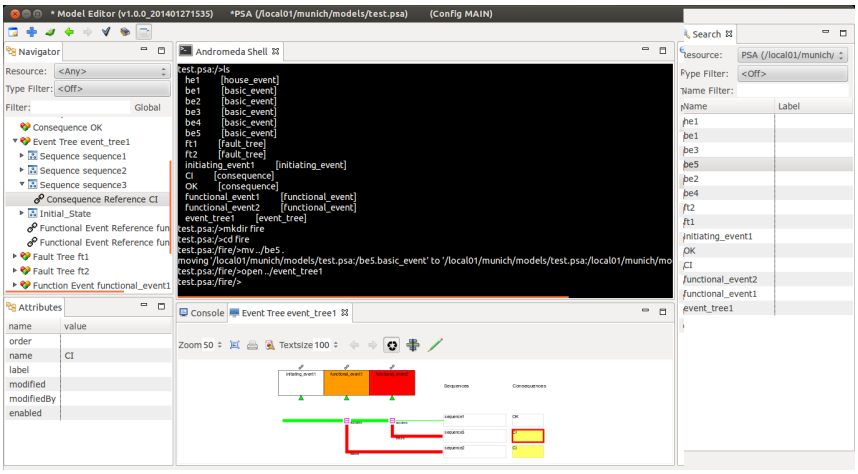


Figure 9.16. Integrated Shell of Andromeda

Shell Commands

In the Andromeda Shell, one can navigate into folders of a model like a file system. The current location of navigation is referred to as the *current location*. A shell command "cd"can be used to change the current location.

Example of commands are:

- `exec <SCRIPT>`: To execute a batch script (see Section 9.3.2).
- `cd <FOLDER>`: To change the current location.
- `mv <MODULES> <FOLDER>`: To move modules to another folder.
- `pwd`: To display the current location.
- `load <FILE>`: To load a model file. The model domain is recognized automatically.
  Example: "load /home/friedlhu/model.psa" (loads a model "model.psa").
- `open <MODULE> <DIAGRAM>`: To open a diagram.
  Example: `open fire/fault_tree1 FAULT_TREE_DIAGRAM` (opens the fault tree diagram of module

fault_tree1)

- `compare <MODEL1> <MODEL2> -f <FILE>`: To compare two models.
  Example: "compare model1.psa model2.psa -f results.csv"

A more exhaustive list of current shell commands can be found in the appendix A.6.

Batch Scripts

A shell command "`exec <SCRIPT>`" has been implemented to launch a batch script. A batch script in Andromeda is given by a sequence of shell commands, which are processed one by one.

Batch scripts are of the form (each "command i" represents a shell command):

```
command 1
command 2
command 3
...
command n
```

A batch script in Andromeda does not support control statements such as while loops. However, batch script commands can serve to launch scripts (such as JRuby scripts), which provide a large spectrum of programming statements.

Module Addressing

To process a module or a selection of modules by a shell command, modules have to be addressed. For example, the command "`mv <MODULES> <FOLDER>`" (which moves modules into a folder) requires to address the set of modules to move.

A unique way of addressing is presented, which supports to address modules in three ways:

1. `<MODEL-NAME>: <ABSOLUTE-MODULE-PATH> / <MODULE-PATTERN>`
   To address modules of a certain folder given by an absolute path.
   Example: `m1.psa:/fault_trees/ASG*.basic_event`
   to address all "ASG" basic events of folder "/fault_trees".
2. `<RELATIVE-MODULE-PATH> / <MODULE-PATTERN>`
   To address modules of a certain folder given by a relative path (relative to the current location).
   Example: `../fault_trees/ASG*.basic_event`
   to address all "ASG" basic events of folder "../fault_trees".
3. `<MODEL-NAME>: <MODULE-PATTERN>`
   to address modules globally regardless their location in the model.
   Example: `m1.psa:ASG*.basic_event`
   to address all "ASG" basic events of model "m1".

where:

- `MODEL-NAME` is the name of the model (containing the module(s) to address).
- `ABSOLUTE-MODULE-PATH` indicates a folder as absolute path. The first character of an absolute path is required to be the "/" symbol.
- `RELATIVE-MODULE-PATH` indicates a folder relative to the current location. The current folder is indicated by "." and the parent folder by "..".

- `MODULE-PATTERN` is a composite pattern of form `NAME.TYPE`. A module matches the pattern if its name matches `NAME` and its type matches `TYPE`. The special symbol "*" is a placeholder that matches any string (zero or multiple characters).

Providing new Shell Commands

The extension framework of Andromeda permits the conception of new shell commands in form of plugins. This offers the possibility to customize and to enrich the script language to specific needs (companies may have different requirements on PSA models).

A shell command in Andromeda is provided by implementing the extension point "fr.edf.andromeda.action". An extension is kind of a hook for applications to extend Andromeda. See 10.3.1 for information about extension points. The list of commands can thus be customized to specific needs.

Figure 9.17 shows how to extend the application with extension point for a command "test". The Java class "Action-Test.java", which implements the "test" command must implement the interface "I_Shell_Command". This interface requires to provide two functions:

1. `public void parse_args(String[] args, I_Shell shell) throws PSA_Exception`
   Called from the shell to parse the command inclusive its arguments.
2. `public void execute(I_Component arg0) throws PSA_Exception`
   Called from the shell to execute the command.

The command "test" (of the example) is interpreted by calling the two mentioned functioned in sequence. Any failures (at parsing or execution step) are displayed in the shell.

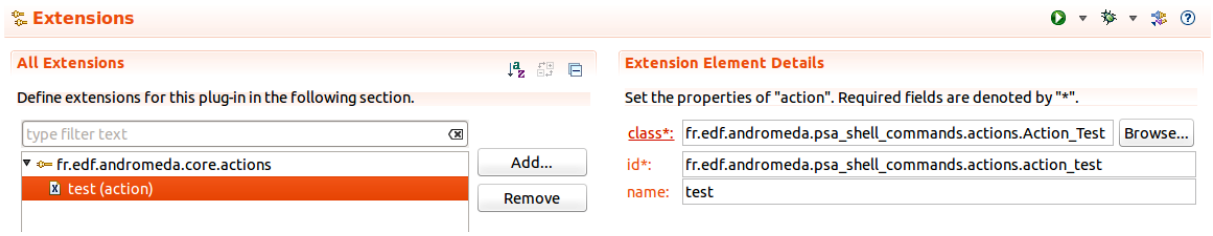


Figure 9.17. Extension point to add a shell command

Scripts

The scripting interface of Andromeda supports to execute "JRuby" scripts. JRuby is a scripting language derived from Ruby to enable scripting for Java applications [77].

A JRuby script can use compiled Java classes. As Andromeda is developed in Java, JRuby is a possibility to access the full functionality of Andromeda via scripts.

Example script in JRuby:

```
java_import "fr.edf.andromeda.api.App_API"
java_import "fr.edf.andromeda.api.Model_API"

model = Model_API.load_model(‘/home/friedlhu/model.psa’’, true,
$m_shell.get_shell())

App_API.add_model($m_shell.get_app(), model);
App_API.refresh_app($m_shell.get_app())
```

```
ft1 = model.get_global_element("ft1", PSA_Element_Types.s_fault_tree, true)
if (ft1)
  ft1.set_name("ft2")
  App_API.refresh_app($m_shell.get_app())
end
```

The example loads a model "model.psa" and adds it to the modular PSA (function App_API.add_model). Then, a fault tree named "ft1" is searched and renamed into "ft2".

JRuby scripts can be launched from the Andromeda Shell (integrated or native) by using the shell command "jruby": `jruby <SCRIPT-FILE>`.

Executed scripts run in the same heap space[1] as Andromeda. The advantage hereby is that scripts can directly manipulate Andromeda data (such as models) and control the application itself. For example, the instruction "refresh_app" of the example refreshes Andromeda in order to display the new model.

Future Developments

Another scripting language to consider is Jython [78]. Jython is the Java implementation of Python. Likewise JRuby, Jython can interface with Java classes. Further, the Jython project has announced to ensure compatibility to access "C-Jython" functionality, what permits to reuse "normal" Python scripts in Andromeda. For EDF R&D, this interest is twofold:

- Many scientific work is realized as Python scripts.
- The library "OpenTurns" [79] - an open source library to treat uncertainties (for safety assessment) developed in cooperation with EDF R&D - is available in a Python version.

9.3.3 Application

Scripts and commands can be considered to support or automate a vast amount of procedures (list not exhaustive):

- Updating procedures (for instance to (re)generate fault trees from system models and update them in PSA models)
- Comparing and merging PSA models
- Establishing a cartography of PSA models (to highlight dependencies)
- Supporting revision and review procedures (for instance to validate PSA models incrementally object by object)
- Documenting a PSA model and automatic document generation

In the sequel, several examples are presented.

Finding Modules

A *find* command has been implemented to find modules matching a certain search pattern and to list their dependencies.

In the following, an interactive shell session is displayed using the find command:

```
/> load /models/model.psa m1                                    (*1)
```

[1] applications store data in a memory called the "heap space"

```
m1:/> find *.fault_tree --list-dependencies --dep-level=-1    (*2)
1. m1:/fault_trees/ft1.fault_tree
  Dependencies:
    m1:/basic_events/be1.basic_event
    m2:/basic_events/be2.basic_event
    m3:/parameters/p1.parameter

2. m1:/fault_trees/ft2.fault_tree
  Dependencies:
    m1:/fault_tree/ft1.fault_tree
    m1:/basic_events/be3.basic_event
3. ...
```

(*1) First, a model "model.psa" is loaded as model "m1". The current location changes automatically to the new model (prompt becomes "m1:/>").

(*2) Next, the "find" statement searches for all fault trees in "m1" (current location) and lists their dependencies. The pattern "*.fault_tree" requires modules to match the fault tree type. The option "dep-level" specifies the maximum depth for analysing forward dependencies (the end-of-recursion when determining dependencies of dependencies).

The remaining lines display the found fault trees and their dependencies.

Moving Modules

Modules can be stored in folders. In the following, an interactive shell session is displayed using the move command ("mv") to move modules into another folder.

```
/> load /models/model.psa m1
m1:/> mkdir asg                       (*1)
m1:/> cd asg                          (*2)
m1:/asg/> mv :*asg*.basic_event ./    (*3)

Output:
moving m1:/basic_events/asg1.basic_event to m1:/asg/asg1.basic_event1
moving m1:/basic_events/m4asg5.basic_event to m1:/asg/m4asg5.basic_event1
moving m1:/test/testasg.basic_event to m1:/asg/testasg.basic_event1
...
```

(*1) create a new folder named "asg"
(*2) enter the new folder (the current location changes to "/asg")
(*3) move all basic events containing "asg" in their names to the "asg" folder ("./" indicates the current folder).

The addressing of modules has been presented in Section 9.3.2.

Importing Modules

In the next example, a fault tree *ft1* is imported from another model (inclusive all of its dependent modules). Existing modules are overwritten.

```
/> LOAD /models/model.psa m1
m1:/> LOAD /models/model.psa m2
m1:/> mkdir new
```

```
m1:/> import m2:/fault_trees/ft1 m1:/new --recursive --replace (*1)

Output:
copying m2:/fault_trees/ft1.fault_tree to m1:/new/ft1.fault_tree
copying m2:/basic_events/be1.basic_event to m1:/new/be1.basic_event
replacing m1:/basic_events/be2.basic_event by m2:/basic_event/be2.basic_event
...
```

(*1) Fault tree “ft1” is imported (from model “m2”) into the folder “new” (of model “m1”). The option *recursive* of the merge command imports (beside the module itself) dependent modules.
The option *replace* overwrites any existence of “ft1” in model “m1”.

Merging models

Merging models can be performed by using shell commands. The next example shows a batch script to merge models:

```
LOAD /models/modelA.psa m1.psa
LOAD /models/modelB.psa m2.psa

MERGE m1.psa:*.basic_event m2.psa    --batch   (*1)
MERGE m1.psa:*asg*.event_tree m2.psa --batch   (*2)

SAVE m2.psa /models/merge_result.psa           (*3)
```

(*1) Merge all basic events of “m1” into model “m2”. With the option “–batch”, merges are not required to be interactively confirmed.
(*2) Merge all “asg” event trees of “m1” into model “m2”.
(*3) Save the merge result (model “m2”) to a file.

Generic fault tree creation

With the implementation of a JRuby interface in Andromeda, PSA models can be developed in a programmatic manner.

An example of a JRuby script to create a fault tree (from scratch) can be viewed in the appendix A.5.

9.4 Concurrent Model Engineering

Due to their complexity and size, PSA models are often developed by a group of persons in parallel. This kind of development is referred to as *concurrent model engineering* and described in this section.

9.4.1 Outsourcing of Model Engineering

In future, certain parts of a PSA model may be provided from different companies. To say, parts of PSA models may be “*outsourced*”.

There are good reasons to outsource or buy parts of a PSA model:

- Expertise: Special knowledge of a subsystem is needed to create a certain model part.
- Budget: It can be cost-effective to buy model parts, which are commonly required from a multitude of customers.
- Risk reduction: Limitation of responsibility to a smaller model part.
- Efficiency: Buying model parts means to immediately obtain solutions.
- Resources: Model engineers or software tools may not be available to develop certain model parts.
- Reliability / QA: Externalized model parts may already be tested or qualified.

9.4.2 Example of Concurrent Model Engineering

In this section, an example is shown to demonstrate how different development activities can be managed.

The situation is given as follows: An integrator is in charge to merge the different model developments. A core model (which contains the principle modules for the project) is purchased from an external company. However, the core model needs to be extended. Two developers are in charge for this purpose. They develop additional modules on top of the core model.

The integrator deals with five development lines (see Figure 9.18):

- Core model line: to provide new releases of the core model (delivered by the external company).
- Two developer lines: to maintain the work of the two developers.
- Integration line: to prepare integrations or release candidates.
- Release line: to provide model releases for safety analysts. Release models are used to perform official PSA assessments.

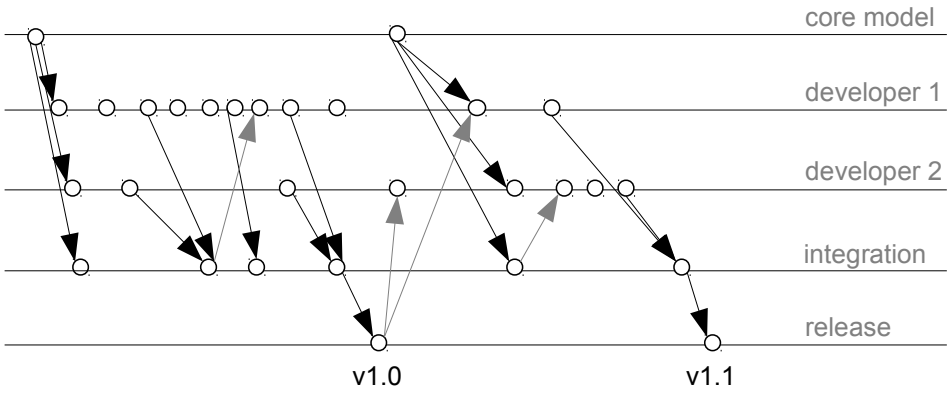


Figure 9.18. Development lines in a concurrent model engineering.

Any new core model version is merged in the developer lines (so that the developers can enhance the latest core model version) and in the integration line.

If a developer has accomplished a certain development step, module tests have to be run to ensure correct modeling and to identify regressions. If the tests run successful, the developer informs the integrator about his modifications.

Next, an integrator merges the developer lines into the integration line. Each integration is followed by a set of integration tests.

If the integration tests run successfully, a release is created by replicating the successfully integrated version into the release line. A tag (a label) is assigned to the release to uniquely refer to it.

If an integration fails, the developers may have to deliver new versions leading to a successive integration.

In the example, some merges are inked in gray color. Those indicate synchronizations of developers with the integration or release line. They are needed to base successive developments on the same (released) model state.

9.4.3 Model synchronization

To develop models by a group of engineers requires to synchronize the different development actions. One strategy is to apply the method of model fusion 7.4. However, if engineers work on the same model parts, merge conflicts may occur.

To avoid conflicts of simultaneously modifying the same model parts, locking mechanisms can be used. A locking mechanism follows a preventive strategy that prohibits engineers from modifying same model parts simultaneously.

In a modular PSA, the locking mechanism can be applied on module level. In order to have the permission to modify a certain module, a model engineer (a client) is required to *lock* it. A module can only be locked if it is not already locked by another engineer. Once an engineer has applied his modifications, he *releases* the lock (so that other engineers can modify the module).

The locking mechanism requires a centralized strategy: A server is in charge to grant or refuse lock demands of clients and to provide the reference model. The reference model represents the principal development state. Clients operate not directly on the reference model. Instead, local models at each client are used to represent a local development state (of a client). Local models can be synchronized with the reference model to commit, update or restore modules:

- commit: transfer module from client to server.
- update: transfer module from server to client, whereas the module at the server is more recent.
- restore: transfer module from server to client, whereas the module at the client more recent.

Figure 9.19 shows a possible course of actions between two clients working simultaneously on the same model.

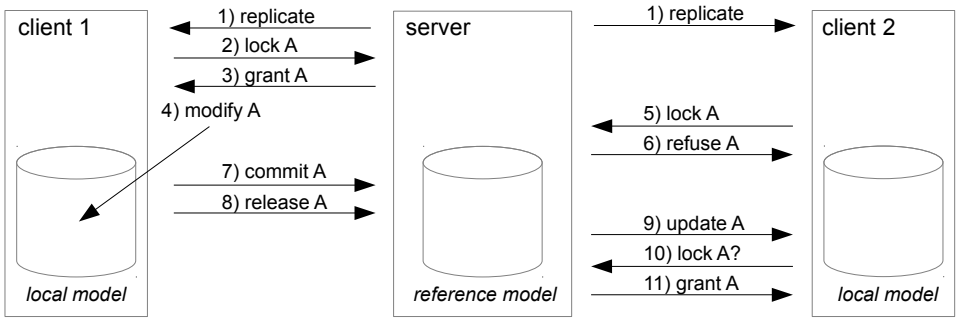


Figure 9.19. Principle of locking mechanism

Explication of actions:

1. The reference model is replicated to the local models of each client (update).
2. Client 1 demands to lock module “A”
3. The server grants the lock demand.
4. Client 1 modifies module “A” in its local model.
5. Client 2 demands to lock module “A”.
6. The server refuses the lock demand.

7. Client 1 commits the modification of module “A”.
8. Client 1 releases the lock of module “A”.
9. Client 2 updates its local model.
10. Client 2 demands to lock module “A”.
11. This time the server grants the lock demand of client 2.

A lock demand for a module is granted if two conditions hold:

- The module is not locked by any other client: The server maintains information, which modules are currently locked by which client.
- The module is synchronized: A module is synchronized, if it does not differ between the reference model and the local model.

Before releasing a lock of a modified module, the module must be synchronized. This requires either to perform a commit or a restore. A restore has the effect to discard modifications.

Figure 9.20 shows another course of actions. This time, client 2 modifies module “B” while client 1 modifies module “A”. However, client 1 restores module “A” (action 8) and updates the changes of client 2 (action 12).

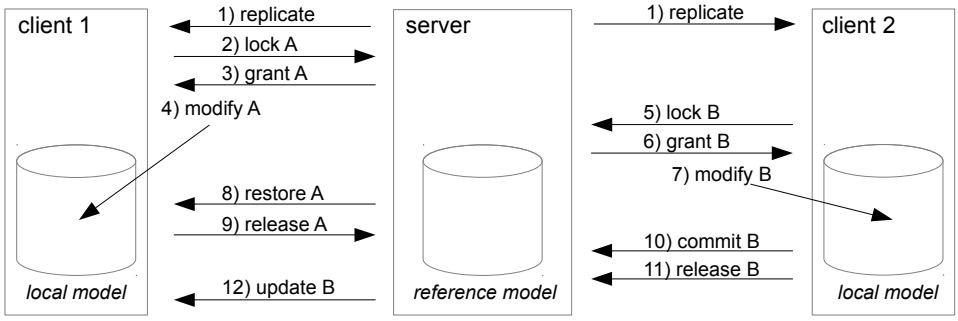


Figure 9.20. Simultaneous model modifications

Often, modifications impact not only one individual module, but a set of modules. The approach gets extended accordingly: A client can demand to lock a set of modules. The lock is granted if

- none of the modules (of the set) is locked by another client and
- each module (of the set) is synchronized in the local model (of the client)

The validation whether a module is synchronized can be implemented by using synchronization identifiers: Each module is assigned to its private identifier, marked as “sync-id”. At any time a module is modified, its sync-id is incremented. A module is synchronized in a local model if it has the same sync-id in the reference model.

If a module is committed, its sync-id is transferred to the reference model. If a module is updated / restored, its sync-id is transferred from the reference model to the local model.

9.5 Verification of PSA quantification

Typically, software tools to assess risk are difficult to verify. The reason is rather a tools problem than a conceptional problem. Often, models can not easily get exported to be quantified by alternative quantification engines. Additionally, missing standards and different PSA modeling techniques make models incompatible to be quantified with other risk engines.

Certainly, commonly used PSA tools claim to be *qualified by experience*. However, the semantic of software code is generally not proof-able. To say, there is never a guaranty, that a software is working correctly. When a software vendors offer "qualified" tools, it is only ensured that the software performed successfully a set of tests. Those tests may have been sufficiently exhaustive or not to cover the most common situations. But they are never complete.

The concern about quantification engines is that most of the model artifacts are neglected during quantification (due to approximations). Consequently, possible errors of quantification engines are likely to remain undetected. Another concern is that there is no reference list of cutsets to check whether a calculated list of cutsets is the one to expect. The problem is, in order to check whether a list of cutsets is correct, one would have to perform the cutset calculation either manual (what is practically impossible for large models) or automatically by another cutset quantification engine. The latter is exactly the idea of this section.

9.5.1 Quantification Process

In this section, the quantification of PSA models in a modular PSA is explained.

Model Setup

In a modular PSA, a project can contain several models. This is used to separate any analysis specifications from the actual PSA model.

PSA models should be considered as input data, and analysis specifications to configure quantification engines. To separate analysis specifications from PSA models permits to reuse specifications for quantifying other PSA models. For example, an analysis case "CDF" for determining the core damage frequency can be used in conjunction with different PSA models.

Consequently, the quantification of a PSA model quantification requires at least two models:

- The PSA model
- A quantification model

Both models can be implemented themselves as a set of models.

A quantification model specifies one or more analysis cases. An analysis case is either

- A consequence analysis case: To analyse a consequence.
- A sequence analysis case: To analyse a sequence of an event tree.
- A gate / fault tree analysis case: to analyze a gate of a fault tree

The quantification model must "include" the PSA model to resolve the object to analyse.

Quantification Process

Quantification results are presented in a dedicated model. So in total, three models are involved: One for the input data (the fault and event trees), one for the configuration (the quantification model) and one for the results.

To quantify a PSA model, the following steps are performed:

1. Create PSA model
2. Create analysis case

3. Build master fault tree (single fault tree representing the object to analyse, potentially of large size)
4. Quantify master fault tree
5. Create results

Steps 1-2 are performed by a model engineer.

Steps 3-5 are executed automatically by a PSA software.

Quantification in Andromeda

In this section, two quantification engines are introduced to verify (to cross-check) quantification results:

- XFTA [19]: A free quantification engine created by A. Rauzy within the Open PSA Initiative at LIX (laboratory of computer science at Ecole Polytechnique in France)
- JFTA: An experimental fault tree engine based on Java [2], developed in the scope of the thesis to test "smaller" fault trees in an integrated approach (JFTA is integrated in Andromeda).

Both, XFTA and JFTA can handle negative logic. Incoherent fault trees and success branches of event trees can be treated.

Beside the object to analyse, an analysis case specifies quantification parameter, which are transferred to the quantification engine. Those are for example truncation parameter such as the maximum cutset order or the minimum probability of cutsets.

The maximum cutset order is used to discard any cutsets of a larger order.

The minimum probability is used to discard any cutset having a lesser probability value.

In Figure 9.21 a XFTA quantification and in Figure 9.22 a JFTA quantification is shown.

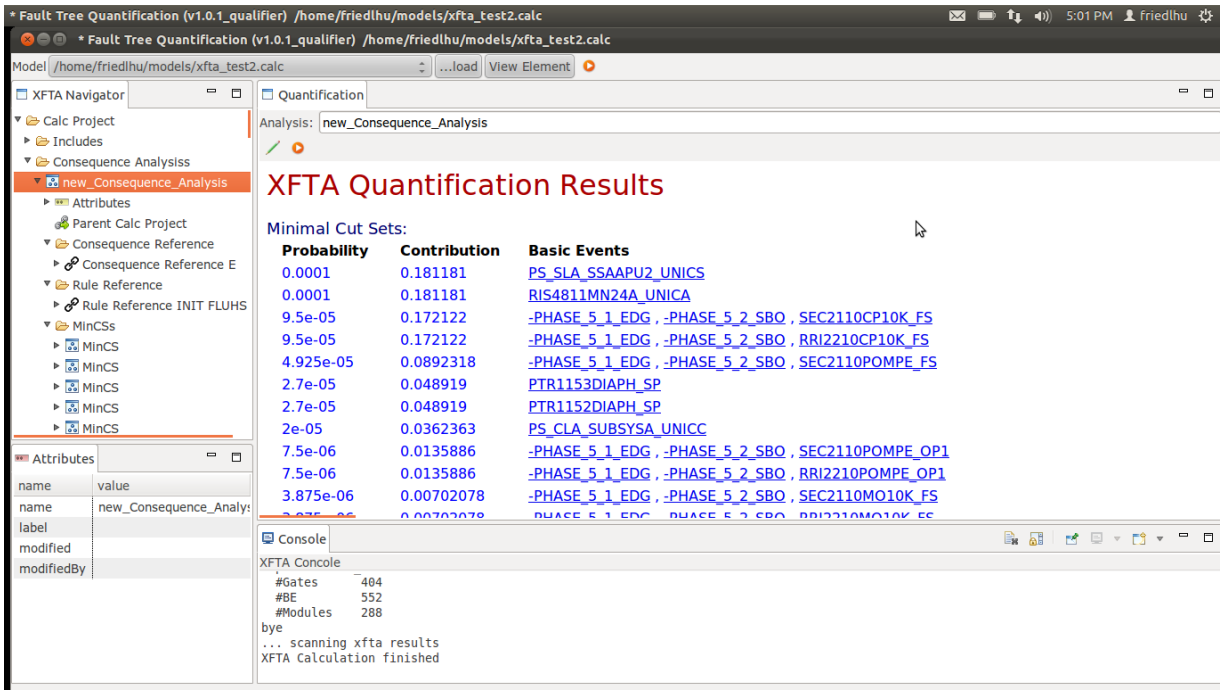


Figure 9.21. Fault tree quantification with XFTA.

[2] JAVA is an object oriented programming language

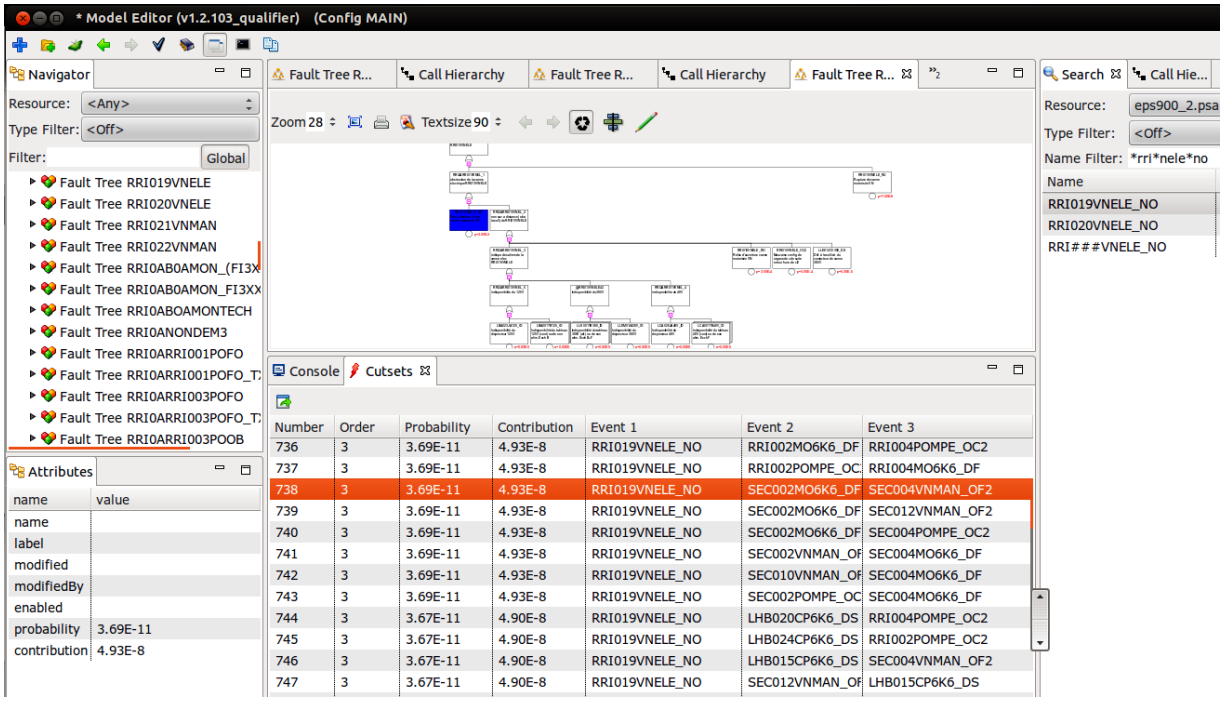


Figure 9.22. Fault tree quantification with JFTA.

9.5.2 Verification of quantification engines

Figure 9.23 illustrates the process to verify quantification results of commonly used quantification engines (such as “RSAT” from RiskSpectrum or “FTRex” from CAFTA).

First, the original PSA model has to be converted into a modular PSA. The result of the conversion is a PSA model (conversion 1) and a quantification model (conversion 2).

Next, the cutsets are calculated twice: Once with the quantification engine to verify and once with the alternative quantification engine. Both results are stored in the modular PSA (conversion 3).

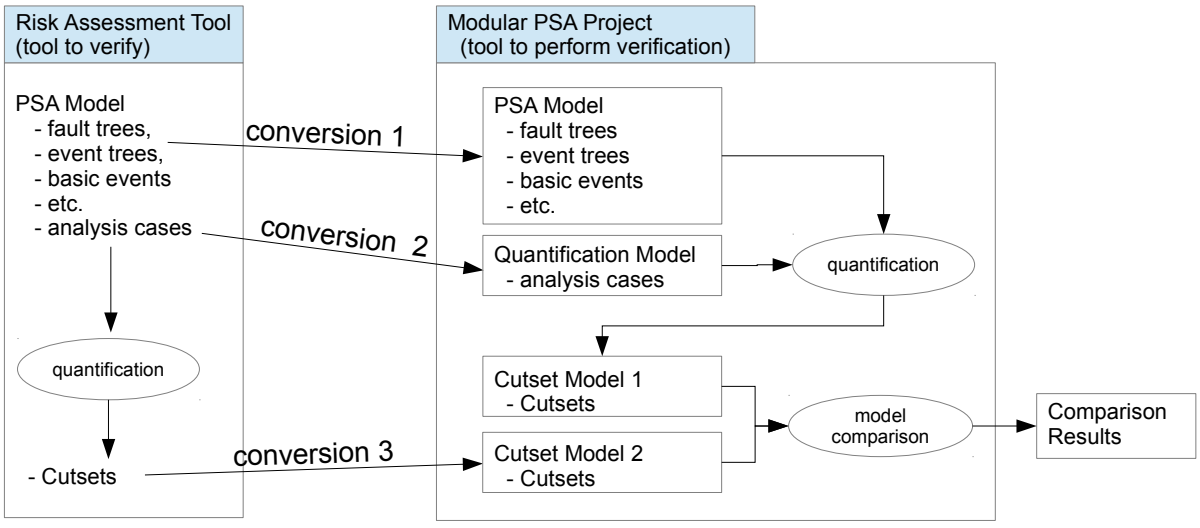


Figure 9.23. Process to verify quantification results

Finally, the two cutset lists are compared. If the lists are different, the quantification engines do obviously not deliver the same results. This does not necessarily mean that one of the engines does not work correctly. It can also indicate a different quantification parametrization. Before concluding an engine is not working properly, it is to ensure that both engines have been configured identically.

The step to compare cutset lists can be performed in a modular PSA by using the method of model comparison (see Section 7.3). Model comparison has been introduced as a generic method that can also compare cutset lists (cutsets are stored in a modular PSA).

Another possibility to compare cutset lists is to export both lists into a textual format. Then, any tool capable to compare textual files can be used to find differences between the sets. However, this possibility may face difficulties if the order of basic events (each cutset consists of basic events) differs between the lists. Then, the comparison may indicate textual differences though the lists are semantically equal. On the contrary, the model comparison of a modular approach can be configured to ignore ordering.

9.6 Conclusion

In this chapter the different phases during model engineering have been discussed in the scope of a MDL (Model Development Process). In any phase, a different set of activities is to consider.

Event sequence diagrams (ESDs) have been presented to support the design phase of PSA models. Based on a formal language they can be processed by PSA tools for example to generate event trees. Further, an approach has been shown to extend them by graph models to obtain a complete design view on PSA models.

To support model engineers and safety analysts in various activities, a scripting interface for a modular PSA has been introduced. The interface permits to interact with models and PSA applications in the form of shell commands and scripts. They can mean a real speedup in productivity and quality.

Next, some considerations on concurrent model engineering, in particular on outsourcing and collaboration have been given. The idea of a locking mechanism could illustrate how to develop PSA models in parallel (by several engineers) in a modular PSA.

Finally, a method to verify PSA quantification results has been shown. The method serves not only to reveal problems of PSA software tools, but also to develop a deeper understanding of model quantification and its parametrization.

CHAPTER 10

Andromeda

During the long life cycles of PSA models (for nuclear power plants), requirements are likely to change from time to time. In Section 4.2.5, the necessity for adapting software tools to specific contexts of companies has been described.

Unfortunately, most PSA software tools are not easy to adapt: Software requirements may conflict with those from other customers using the same software. And software architectures may not be designed to efficiently integrate new functionality.

In this section, the software Andromeda is presented. Andromeda is a generic modeling platform based on the modular PSA approach and created in the scope of this thesis at EDF R&D. It has first been presented in [80]. An article has been published in [81] to issue preliminary ideas of Andromeda.

The software comprises a modular software architecture designed to develop, extend and customize functionality. Though most of its current functionality has its application in the field of risk assessment, Andromeda - likewise the modular PSA concept - is not specific to the domain of PSA models and can find applications in other fields.

Section 10.1 elaborates the difficulty of software vendors to develop PSA software that is adapted to specific contexts (of companies).

Section 10.2 gives an overview about fundamental concepts and goals of Andromeda.

Section 10.3 reveals architectural details, in particular the modular composition of Andromeda and its extensible frameworks.

Section 10.4 illustrates the provision of Andromeda extensions from a software engineering point of view.

Section 10.5 concludes this chapter.

10.1 The Problem with Evolving Software Requirements

Andromeda is based on an open and extensible architecture to respond rapidly to new software requirements. The problem of evolving requirements is stated in the sequel.

Initial PSA software at EDF (and likewise in other companies) has been rather basic:

- One individual software tool (e.g. RiskSpectrum [22] at EDF) was considered to perform any PSA related tasks. The whole PSA functionality was provided by one individual software tool.
- PSA software did not require to implement customer specific functionality. Initially, PSA models were constructed rather simple and models have been of smaller sizes. Software vendors of PSA software could provide a general set of PSA functionality for all customers.

With the time, new requirements had to be incorporated for PSA engineering. For example, soon (in the 90's) the need came up to automatically generate fault trees from system models (EDF uses the software KB3 [42, 82] for this purpose). At the same time, EDF specific functionality gained higher importance, for example to verify that fault and event trees conform to EDF specific naming conventions.

For software vendors, the implementation of new requirements can pose serious problems. Requirements of different clients may be conflicting (they cannot get all implemented at a time) or they are economically not interesting when only "small" clients demand for it. Often, only requirements claimed by a majority of customers are implemented. Other requirements, in particular customer specific ones, may be retarded or even rejected. Customers may also have to wait a long time until requirements are decided to get implemented as a consensus between customers is needed. Consequently, alternative, probably less adequate approaches have to be considered.

10.2 Philosophy

Andromeda has been developed with the following objectives:

- Reducing complexity of models.
- Reducing complexity of software applications.
- Providing an open and extensible application architecture for PSA tools.
- Facilitate to interface with different PSA tools.
- Facilitate to link different models.
- Simplify to develop and integrate new modelling applications.
- Simplify to adapt existing functionality to specific needs.

To achieve these goals, Andromeda incorporates three principles, which are presented in the remainder of this section.

10.2.1 Extensibility

Andromeda is based on an extensible architecture with the capacity to respond to evolving PSA requirements by providing new functionality in form a so-called "plugins". Plugins are individual software parts that can be added or removed from an overall software installation.

Plugins can also serve to customize PSA software. The current difficulty of software vendors to develop PSA software that specifically considers the requirements and backgrounds of individual companies has been mentioned in Section 10.1. The extension concept of Andromeda considers to extend PSA software

to specific needs (i.e. to customize it).

The idea is that companies can assemble the final PSA software from a set of plugins. Any functionality that is not required can be removed or deactivated.

10.2.2 Connecting Functionality

Since ever, one of the most challenging subjects in computer science is to **reuse** (to share) already developed functionality in order to avoid redundant software development and to speedup the integration / development of new functionality.

The aim of Andromeda is to ease interfacing with existing functionality, concerning both internal (functionality provided by Andromeda) and external (provided by non Andromeda Applications) functionality.

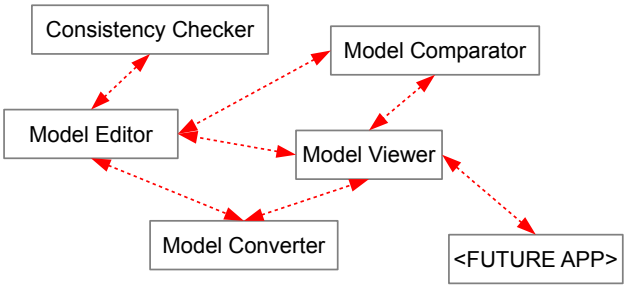


Figure 10.1. Applications interface with other applications

In Andromeda, internal functionality can be provided in the form of so-called "Andromeda Applications". Figure 10.1 illustrates the application philosophy to connect and reuse functionality. For example, an application "Consistency Comparator" (to compare models) is reused in the applications "Model Editor" and "Model Viewer", though in different contexts ("Model Editor" compares models to assist model engineers for cross checking modifications and "Model Viewer" compares models to highlight differences between model versions and variants). Application "Future App" indicates a future application to integrate.

To maximise the reuse of functionality, a major challenge is to develop **generic** software frameworks[1] and to establish clear interfaces between applications.

10.2.3 Connecting Models

Likewise the need to reuse software functionality between tools, it is also recommended to reuse model assets (parts of models) between different models.

A common solution to reuse model parts is to replicate / synchronize models via model conversions. However, this solution comprises some disadvantages:

- Model conversions may not be free of losses. Consequently, modifications of converted models are difficult to convert back[2].
- Software to convert and merge models has to be developed.
- Models become quite "static", replications are like "copy-paste" actions. Sometimes, however, it is preferable to have kind of dynamic links to other model assets. Dynamic links have two advantages:

[1] A framework provides generic software routines and tools for efficiently enhancing a software.
[2] Backward conversion refers to replicate modifications back to the original model, that has been converted

First, links can be redirected. Second, links refer always to the most current state of model assets (whereas replicated parts may be out of date).

The approach of Andromeda and a modular PSA is to link models (technically modules refer to other modules which may be located in a different model). The aim is to establish an *interconnected model world*. Figure 10.2 illustrates the idea: Models are not to consider as individual items, they are a rather integrated in a larger bunch of models.

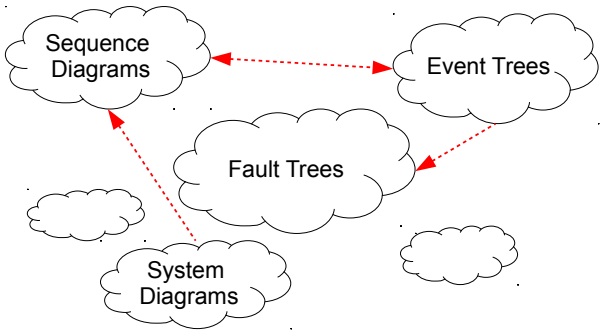


Figure 10.2. Idea of an interconnected model world.

The final idea consists in organizing generic applications around a model world to process models in a common manner (see Figure 10.3). The advantage is to require only **one** "Model Editor", **one** "Consistency Checker" etc. to process all kind of models regardless their types. New applications and models are supposed to be added or removed at any time.

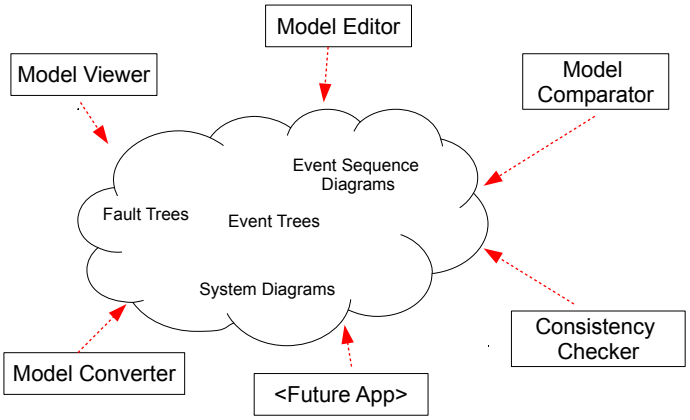


Figure 10.3. The final idea: Generic applications are organized around a bunch of interconnected models.

10.3 Architecture

A software architecture describes the principal software components and their relationships in a software system. In this section, the software architecture of Andromeda is detailed.

10.3.1 Eclipse Technology

Andromeda uses Eclipse technology to implement its concept of extensibility.

Eclipse RCP Framework

The Eclipse RCP Framework provides a framework to build Eclipse applications, so-called "*Rich Client Platform*" applications (RCP applications) [83, 84]. RCP applications are developed in the programming language Java [85, 86].

The most famous Eclipse RCP application is called *Eclipse*. Eclipse became famous in the last decade especially for the purpose of software development and recently for model engineering. For software development, Eclipse provides software development environments for JAVA, C++ and other programming languages. For model engineering, Eclipse provides the *Eclipse Modeling Framework* (EMF) [87].

All RCP applications implement *Equinox*, the Eclipse implementation of the OSGI standard (Open Services Gateway Initiative). OSGI is a specification of the OSGI Alliance [88] for a modular software architecture that defines how to administrate software bundles [89, 90]. The standard is particularly used to build flexible software systems where components can be added / removed when a software is build or while a software is executed (at runtime). OSGI finds applications in embedded systems, mobile telephones, cars etc.

Also Andromeda is an Eclipse RCP (Rich Client Platform) application. It is based on the Eclipse core functionality and additional functionality developed particularly for Andromeda.

Plugins

Eclipse RCP applications are entirely based on so-called "plugins" (indeed everything is a plugin, even the "core" (the kernel) of applications). Each plugin represents a software bundle (a software component), providing its set of functionality which becomes available once the plugin is installed.

Each plugin can exist in different versions. In each version, a plugin declares dependencies to other plugins (in their appropriate versions).

A dependency resolver ensures that the set of installed plugins remains consistent. At plugin installation, a SAT resolver determines possibly conflicting dependencies between sets of installed plugins.

As any RCP application, also Andromeda is based on plugins. By adding or removing plugins, Andromeda can be customized to specific needs.

The plugin architecture of Andromeda is fundamental for the extensibility concept. Any functionality is provided via plugins. By adding (removing) plugins, functionality can be customized and extended.

Plugins permit further to configure the interaction between applications. In Figure 10.4, two applications are shown, whereas the interaction is realized as application extensions. Thus, different customers can consider different sets of applications and customize their interactions.

Features

In fact, the process of adding plugins to a RCP application does not directly install plugins (though this has been possible in earlier RCP versions).

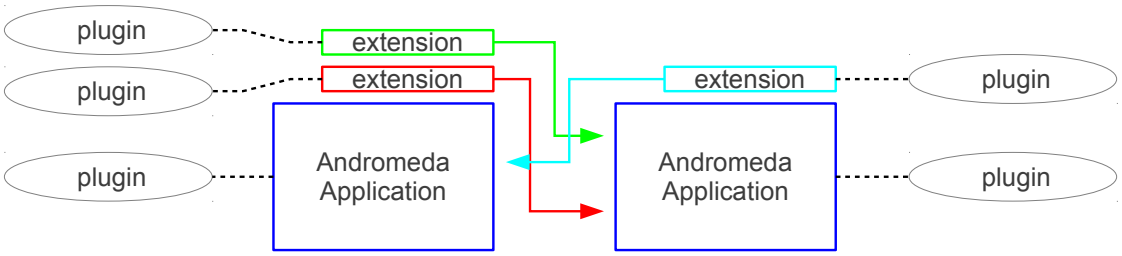


Figure 10.4. Plugin Architecture in Andromeda

Instead, Eclipse RCP is based on the installation of so-called "features": A feature groups a consistent set of plugins (in their appropriate versions). With features, a user does not need to deal with the installation of individual plugins.

Features express a user point of perspective - of higher level - on software functionality. For example, the functionality to develop Python scripts[3] may be provided as one feature containing two plugins: One for a graphical editor to create scripts and another one to execute them.

Each feature has its versions (similar to plugins). Features can contain other features and declare dependencies to other plugins or features. At feature installation, the SAT resolver validates installation consistency to ensure that there are no version conflicts between features and plugins due to conflicting dependencies.

Extension Points

To extend functionality, RCP provides the concept of so-called *extension points*. An extension point is a kind of "hook" in an existing RCP application to enhance it.

Each plugin can provide *extensions* by **implementing** extension points. Extension points constitute the lowest level of the extension mechanism.

An extension point is described by a schema. The schema is specified in XML and can require (among others) to specify a Java class for the implementation of an extension point. Applications can iterate over the list of extensions and instantiate specified Java classes to execute functionality. This is how the extension of applications is generally realized.

Further, plugins cannot only provide extensions (implement extension points), they can also define new extension points. Those can in return be instantiated by other plugins.

Platform Independency

Eclipse RCP applications are generally designed to be platform independent (though they can become platform dependent in case specific functionality is provided), i.e. they run under any operating system which provides an adequate Java Runtime Environment (Linux, Windows (all versions), MAC OS etc).

Andromeda is platform independent.

[3] Python is a scripting / programming language.

10.3.2 Modular Architecture

Andromeda is based on a modular software architecture. This kind of architecture requires that some (if not all) software components are organized in a modular manner: They can be added / removed to / from a software system. The modular architecture is useful to tailor a software to specific needs.

Figure 10.5 illustrates the modular architecture: In principal, the modular architecture consists of software modules (not confuse with modules of a modular PSA), of a generic kernel and of models. A generic kernel provides common functionality for processing models. Software modules (technically realized as plugins) provide functionality for example in form of so-called *Andromeda Applications*. Those are independent software components that use kernel functionality or functionality from other software modules to process models. Models are based on the modular PSA approach and are commonly processed by various software modules.

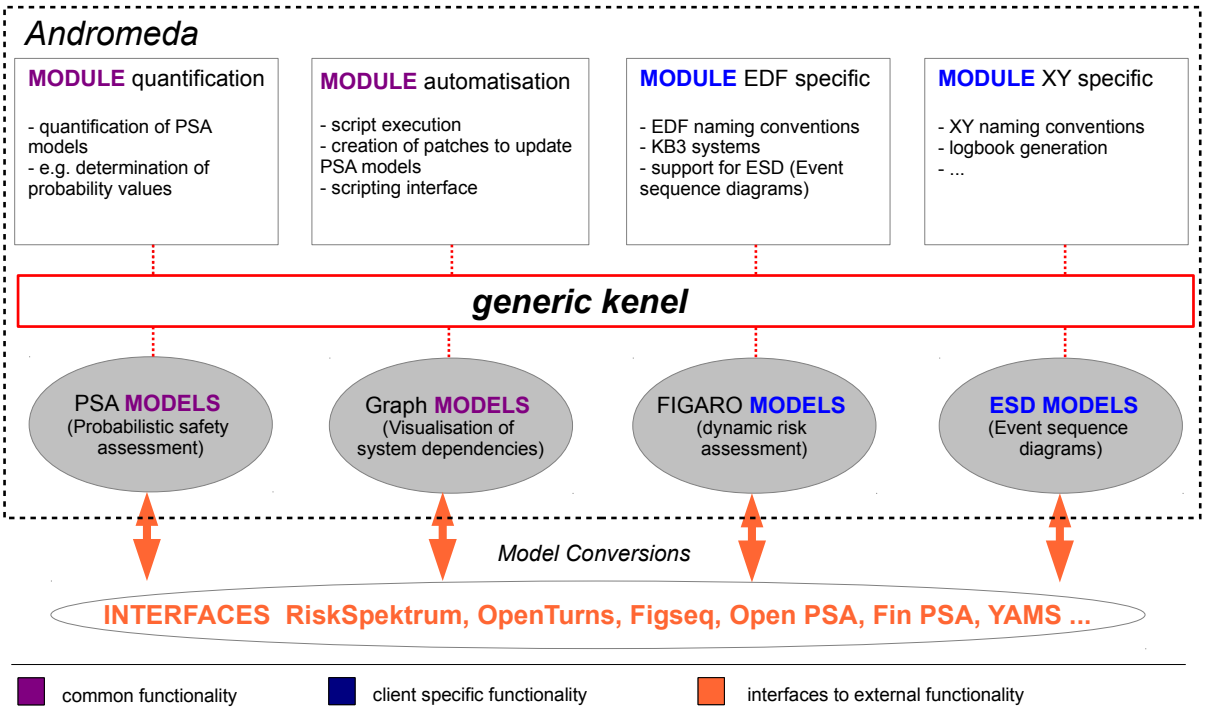


Figure 10.5. Architecture of Andromeda

Both, models and software modules can reflect common and client specific needs (though the separation between common and specific needs is not technical). Commonly used software modules and models are provided for several customers. Each customer can complete its software installation by specific modules and models. In the example a module *EDF specific* is illustrated to provide EDF specific functionality. And *Figaro* models reside only in EDF installations. Figaro is a higher level modeling language used - at the moment - only at EDF.

But not only for the purpose of software customization, also at software engineering a modular architecture has its benefits. Modular software components can be considered as black boxes. The internal composition is hidden, only interfaces (application programming interfaces (API)) of the components are specified. Components are interacting with each other by their interfaces. In detail, the advantages are:

- Software components can easily be developed rather independently from other ones. This eases concurrent software engineering where different software developers (of probably different software

companies) develop collectively on the same software system.
- It provides a clear and intuitive software design by forcing developers to specify interfaces. To "*get into a source code*", i.e. to understand an existing software becomes much easier if software is based on a clear design.

The modular software architecture is further supposed to exchange models with external applications. In the illustration in Figure 10.5 models are converted into various formats to be exchanged with RiskSpectrum (RiskSpectrum PSA), Open Turns [79] etc. Exchanging models pose a simple and effective interface between Andromeda and external applications.

10.3.3 Frameworks

A framework provides generic software routines and tools to efficiently extend a software.

The benefit of frameworks is to increase efficiency at software engineering by reducing the amount of software to develop and maintain. Generic framework routines are **reused** in any programming problem that is realized within the framework. Frameworks avoid code redundancy by using generic programming methods [91] and they ease the modification of source code: It is sufficient to modify once the framework in order to modify any problem, that is realized within the framework. Otherwise one would have to modify any problem separately, what may be time-consuming and error-prone (due to the risk of wrong source code replications). At software engineering, code redundancy is one of the major problems to avoid whenever possible.

The following frameworks are part of Andromeda:

- Extension Framework: A framework to extend Andromeda.
- Application Framework: A framework to add new functionality in form of applications (Andromeda Applications).
- Modeling Framework: A framework to implement the concept of a modular PSA.
- Diagram Generation Framework: A framework to generate diagrams from the content of models.
- Domain Generation Framework: A framework to generate domains for new type of models.
- Analysis Framework: A framework to check model consistency by verifying the satisfaction of constraints (model checking).

Extension Framework

The extension framework consists of a set of extension points and additional software routines and registries to manage extensions.

Extension points in Andromeda are either defined by

- One of the RCP (non Andromeda) plugins: Those concern mainly extension schemes to extend applications by so-called "Views" and "Editors" (graphical components in Eclipse applications).
- Andromeda itself: the core plugin of Andromeda defines 17 extension point schemes in order to extend Andromeda in several ways.

Some examples of Andromeda extensions are:

- Applications can be added (or removed): Once added they can be launched in Andromeda (directly or from other applications).
- Applications can be extended: Various generic possibilities exist to extend applications for example via popup menus, toolbars, dialogs etc. Further, application specific extensions can be provided.

For example, the application "Consistency Checker" can be extended to test user specific constraints (e.g. customer specific naming conventions).

- Domains can be added (or removed): By adding a domain, models of this domain can be processed.

Application Framework

Andromeda can execute so-called *Andromeda Applications*. Those are applications that are executed by the Andromeda application framework. They share the set of loaded models and they can interface with each other.

To run all applications within the same framework enables an integrated way of model engineering [92], where a complete set of functionality (to engineer models) is provided by one single software tool. The opposite strategy is to consider a set of software tools that are running independently from each other. Both strategies have their advantages and disadvantages.

Applications can be divided into two types:

Generic applications: Generic applications are domain independent, i.e. they are applicable to models regardless their type. Andromeda provides several applications of this kind. For example the *Model Editor* (to edit models) or the *Consistency Checker* (to validate models) are domain independent applications. Also the *Diagram Viewer* (to visualize diagrams) is a generic application whereas diagrams themselves are created with domain specific applications.

Specific applications: Specific applications are domain specific, i.e. they are solely applicable to specific model types. For example the application to quantify fault trees is (obviously) only applicable for PSA models.

Generic applications are often **extended** whereas extensions are domain specific. For example, the *Model Editor* is extended by wizards and dialogs to configure CCFs (Common Cause Failures), basic events etc. Also the *Consistency Checker* is extended to verify specific constraints in PSA models, for example that fault trees do not contain "circular" structures (where gates are recursively contained in themselves).

Generic Modeling Framework

Andromeda integrates a generic modeling framework. Applications can process any type of model that has been specified in form of a domain. In particular, the framework provides functionality to read, analyze, modify and save models. Models are commonly shared between Andromeda Applications. Once a model is loaded, it becomes available in all currently running applications. However, a model can only be modified by one application at a time (to avoid conflicts).

The generic treatment of models is achieved by using an abstract data model. This abstract data model is presented in Figure 10.6. The *internal model* is a structure of interfaces[4], model elements are based on. The *reflection model* is a structure to encode the domain of a model.

In comparison to the UML diagram of model composition presented in Section 6.2, the model structure in Andromeda is rather flat (only three interfaces exist to express all kind of model elements). This eases to treat model elements by generic routines. On the other hand, the reflection model is needed to obtain characteristics that are not encoded by the model structure (for example, whether an element is an operator or not, is encoded in the reflection model).

The different interfaces are explained in the sequel:

[4] An interface defines a set of operations.

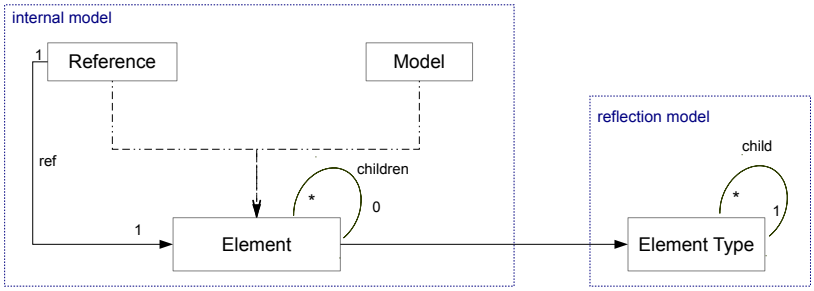


Figure 10.6. Internal data model of Andromeda models.

`Interface Element`
This is the basic interface implemented by all model elements (components, properties, operators, terms, models and references).

In general, the interface provides methods to manage *children*. A child is a contained model element. For example, the children of models are their contained components (their modules). The children of components represent properties and contained components. Properties contain one child that represents the associated term. The children of operators are their arguments.

However, any modeling operation of an element is verified against its type specification (which is part of the reflective model). For example, if an operator should contain a component (as argument), the generic routine to add new children would reject this operation as operators are not allowed to contain components (they can reference components however).

To base on a few generic interfaces to manage *children* has turned out to be a great advantage in developing generic software routines.

`Interface Reference`
This is the interface for all references. It implements itself the interface *Element*, so any reference is also a model element. The interface provides additionally functionality to resolve and configure references.

`Interface Model`
This is the general interface for all models. The interface provides special functionality for administrating (recursively) contained model elements.

It implements itself the interface *Element*, so any model is also a model element. A model can thus be embedded in a larger model and not "just" linked to it (though this possibility has not yet been further analysed up to now).

`Interface Element Type`
An element type defines the underlying scheme of model elements.

The element type provides information about the kind of element, for example whether an element is a component, a model, a term, an operator etc.

Further, it provides constraints about which kind of children under which conditions can be contained in an element. In short it defines how elements can be composed.

The element type is consulted by generic routines to determine and verify the set of valid operations that can be applied on an element.

Comparison to Eclipse Modeling Framework The Eclipse Modeling Framework (EMF) is a project to create and edit models, available in the Eclipse Platform. Some principles are common to those of

Andromeda, others are not.

Common points:
Models are specified in a meta modeling language. In EMF, this meta language is called *Ecore* [87], in Andromeda it is named *ADSL*. In both cases, automatic code generation creates internal data models and basic functionality. Domain specific functionality can be developed (via extension points).

Differences:
However, regarding model composition there is one big difference. EMF does not consider a phase of model instantiation to adapt models to different contexts. Andromeda is designed to satisfy the needs of a flexible modeling framework that is able to adapt models. EMF links model components statically (by fixed unique ids that cannot be easily changed) whereas Andromeda links them dynamically and a dependency resolution resolves them (at runtime) specific to a context.

The instantiation process in Andromeda is done "on the fly" at runtime without any conversion routines required. With EMF, one would typically first have to convert models. Surely this could be automated but the following disadvantages would remain:

- Backward conversion after model modifications (to replicate changes back into the generic model).
- PSA Models are quite large (conversions can take time).
- Software engineering problem to create "light weight" applications: EMF required many plugins and application sizes may increase quickly (for the case of applications providing very few functionality, this may provoke application reluctance).
- EMF tools are not always adapted to treat very large models efficiently (*).

(*) The EMF capacity to load and modify large PSA models has been analyzed. This required first to generate the necessary DSL (Domain Specific Language) artifacts in Eclipse in order to load and display the content of PSA models based on EMF (technically Eclipse plugins have been generated due to a XML scheme definition for PSA models).
Then, a real PSA model of EDF with more than 200.000 model elements (exported from RiskSpectrum) has been loaded. The loading time was about five minutes, the charged heap space more than 500 MB. Andromeda is a light weight modeling framework. The same model in Andromeda loaded in 20s and required only 130 MB of heap space.

However I rate this as a tools problem (the generated artifacts may not have been optimized to work with large models) and not as a problem of the EMF concept. The philosophy and community behind EMF is great and interfaces to convert Andromeda models to EMF and back may be considered in future versions in order to take advantage of the EMF ecosystem (though in an independent manner).

Domain Generation Framework

To add a new domain to Andromeda, a new plugin has to be developed to provide the domain. A domain generation framework has been developed to generate plugins for providing new domains.

The generation framework consists of a set of *meta routines* that generate Java source code due to a *domain specification*. The meta routines have been developed in Java (i.e. Java is used to generate Java). The domain specification is performed (by model engineers) in a dedicated domain specification language for a modular PSA. This language is called *ADSL* (Andromeda Domain Specification Language).

The principle of domain generation is illustrated in Figure 10.7. Each generated domain consists of an internal data model (providing a class for each element type that can be instantiated to create a model element of this type) and a reflection model (providing a type description for each model element).

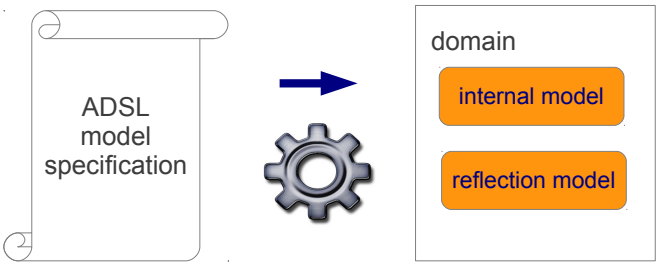


Figure 10.7. Domain Generation in Andromeda

The provision of a new domain involves the following steps:

1. Domain specification in ADSL format
2. Domain generation: The generation framework generates an internal data model and a reflection model for the domain. It generates further any declarations to embed the generated domain in the existing architecture.
3. The generated plugin can be extended by user specific functionality (optional).
   For this purpose any generated Java class conforms to `Gen_<NAME>` and is inherited by a class `<NAME>`. For example fault trees are represented by the class `Fault_Tree.java` that inherits a class `Gen_Fault_Tree.java` whereas the first class contains user specific routines and the second one generated ones. This permits also to **overload** generated routines.

By activating the generated plugin in Andromeda, the domain becomes available. As all plugins can be removed or added at any time, the set of available domains can be customized.

The situation from an application point of view is illustrated in Figure 10.8. Applications use the modeling framework to interact with models. Each type of models is represented by a domain that provides an internal and a reflection model.

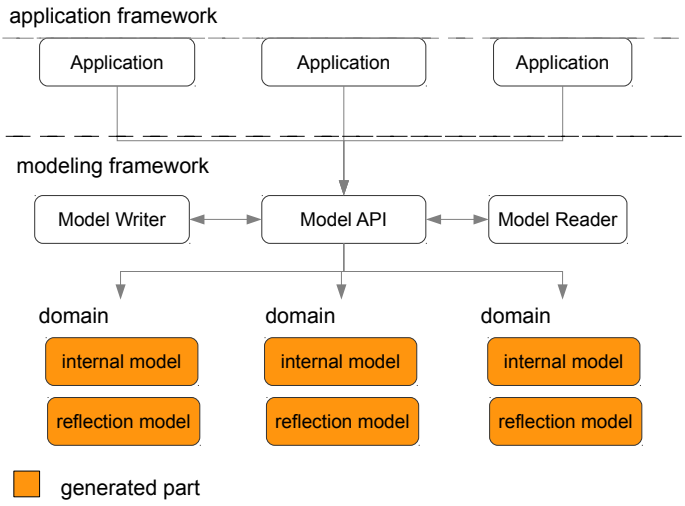


Figure 10.8. Domain Generation in Andromeda

Domain Inheritance

Andromeda supports domain inheritance. Domain inheritance is a method to extend (parts of) other domain specifications.

Domain inheritance is explained by an example: Let A and B be two domains.
Domain A defines three component types $C1$, $C2$ and $C3$. Domain B defines component type $C4$ and inherits $C1$ and $C3$ from domain A. The following lines illustrate the scenario:

```
Domain A: C1, C2, C3
Domain B: C4, A:C1, A:C3
```

There are several advantages of domain inheritance.

One advantage of domain inheritance concerns the development of domain languages: There is no redundancy at specification: If $C1$ of domain A changes one day, domain B remains up-to-date.

Another advantage is that domain inheritance leads to an inheritance of any functionality, that has been developed for the other domain. For example, if $C1$ is the fault tree component type, then any fault tree functionality developed in domain A can be used as well in domain B. Both domains share the same component type.

Domain Linkage

Domain linkage is the base of linking models of different types. Technically, components of one domain link (reference) components of another domain. In the following example, a domain A defines component types $C1$, $C2$ and $C3$ and a domain B defines a component type $C4$ whose property type $P1$ is of value type $C1$ (from domain A):

```
Domain A: C1, C2, C3
Domain B: C4 with C4.P1->A:C1
```

Components of type $C1$ are not considered to be contained in a model of domain B. However, components of type $C4$ can link components of type $C1$ by its property $P1$. Technically, they can create references.

Diagram Generation Framework

A diagram is a graphical representation of compositional, relational or / and behavioral aspects of modelled components.

Most diagrams in Andromeda are automatically derived from models. No explicit "diagram" needs to be created to store geometrical data of diagram elements (e.g. sizes and positions). Instead, generation routines (diagram generators) generate diagrams automatically for a model. A generation routine is typically domain specific. For example, fault and event tree generators generate fault and event trees for PSA models. Diagram generators can serve as well to provide a user interface to modify diagrams, i.e. they can be used as model editors.

While the generation routines are domain specific, the generated diagrams are not. They are based on a generic format. The composition of a diagram is illustrated in Figure 10.9. Diagrams consist of groups. Those groups contain nodes and edges. Nodes (can) represent model elements and edges serve to connect nodes. In order to visualize a node, primitives a used. Primitives are basic graphical shapes such as a rectangles, circles, lines or images (list not exhaustive).

The powerful instrument is to associate nodes to model elements and actions. Applications can provide functionality without having to "understand" the type of diagrams. For example, an application "cutset analysis" can associate any fault tree nodes (nodes linked to fault trees) with the functionality of cutset analysis when clicking on it (in the diagram). Thus, functionality can get easily embedded in diagrams independently from diagram generators or diagram types.

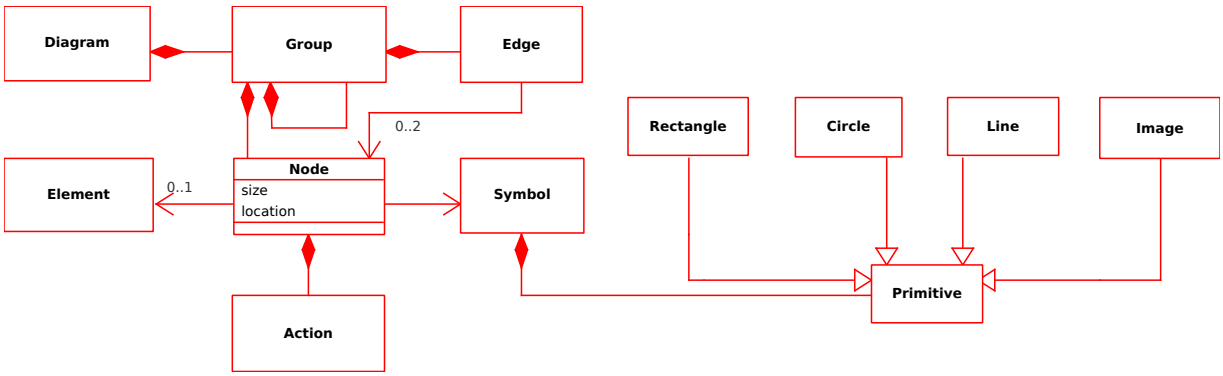


Figure 10.9. Generic Diagram Generation Framework

Nodes can offer themselves functionality by linking them to actions. An action is another generic construct that executes some functionality for example to open another diagram, to open a documentation page, to export a graphic, to edit a diagram etc. An action can be embedded in a popup menu (that opens at right click on a diagram node) or it is directly executed when clicking on the node. So-called "triggers" can be specified to specify under which conditions an action is executed. Those are by the way the same kind of actions that can be provided by the Andromeda Shell in form of shell commands (see Section 9.3) or embedded in model documentation in form of HTML links (see Section 8.1).

To base all diagrams on the same generic structure has several advantages. Diagrams can be processed by generic functionality. An example is the functionality "Diagram Export" that exports any kind of diagrams in form of "SVGs (Scene Vector Graphics)" or "PNGs (a bitmap format)" images. Another example is the *Diagram Viewer* to display (and also to edit) diagrams. The viewer can display any diagrams based on the generic format. Figure 10.10 shows the viewer displaying a fault tree, Figure 10.11 shows the same viewer displaying an event tree.

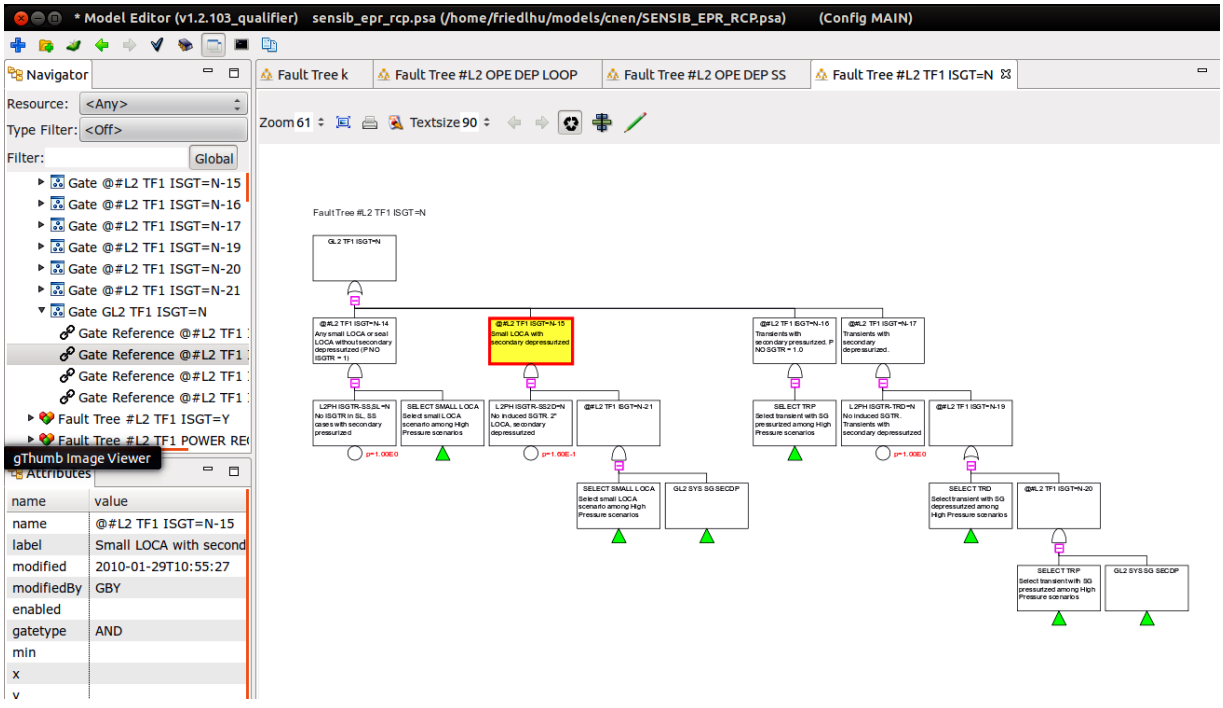


Figure 10.10. A fault tree diagram

Figure 10.11. An event tree diagram

To use the same application for displaying diagrams (the diagram viewer) has the advantage that diagrams can be easily browsed: For example, when clicking on the green triangle below a function event of an event tree, the diagram viewer switches the current diagram to display the linked fault tree diagram (though this behavior is configurable).

Analysis Framework

The analysis framework serves to provide new verifications to validate the consistency of models. It is in particularly important as models in a modular PSA are designed context dependent and consistency has to be ensured for each different context.

The principle of model consistency has been detailed in Section 7.5.

10.4 Andromeda Development

In this section, first the command pattern in form of Andromeda Actions is presented. Then, the development of Andromeda extensions and of new domains is detailed.

10.4.1 Command Pattern

A pattern (or design pattern) is a paradigm of computer science that specifies a reusable solution for a certain type of software problem.

The so-called “Command Design Pattern” or also “Action Design Pattern” encapsulates functionality (or information about functionality) into an object, whereas the functionality is executed at a later point in time on a “receiver” object.

The command design pattern is provided in Andromeda by so-called *Andromeda Actions*. Figure 10.12 shows the principle: Actions can be embedded in diagrams, in shell and scripting interfaces, in model documentation (Wiki), in popup menus, toolbars, application menus or in other actions (list non exhaustive). When an action is declared (declaration space) it is not necessarily yet clear on which receiver it should be invoked (execution space). This is why the separation between specification and execution of actions is mandatory.

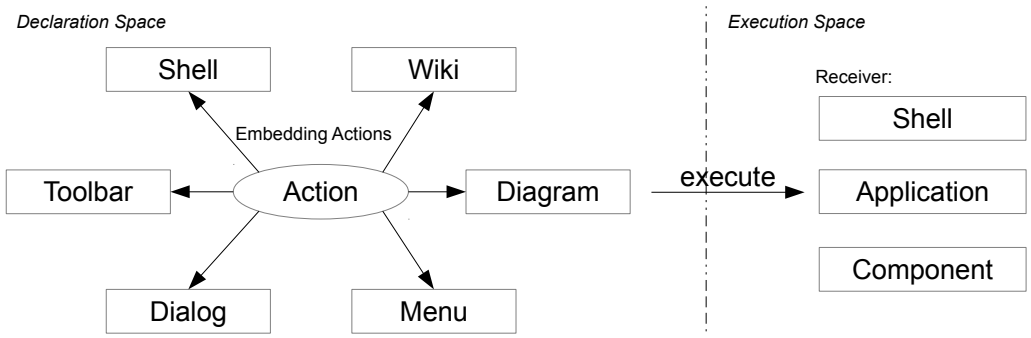


Figure 10.12. Principle of Andromeda Actions

Typically, the embedding entity of an action determines the receiver. For example, an action embedded in a shell is also invoked in that shell. An action embedded in a menu is invoked on the corresponding application of that menu. Some actions, however, may work only with specific receivers. In this case an action can require to be executed on a specific receiver.

In Andromeda, actions are required to implement the "I_Action" interface. Any action implements the function:

```
execute(I_Component) throws Execution_Exception
```

whereas

- *I_Component* is the receiver object.
- *Execution_Exception* is an exception thrown in case the execution of the action fails.

At execution, the "execute" function is invoked on the receiver.

Andromeda actions can further implement the interface "I_Undo_Redo" to provide provide "redo / undo" functionality.

Parametrization and results of actions is achieved via the "setter / getter pattern" (by implementing "setxy" and "getxy" methods). This provides a clear way of programming.

10.4.2 Extension Development

To extend Andromeda is particularly interesting as it does not require to have specific knowledge about framework or kernel (Andromeda Kernel) development. The extensions kind of "hook" into the existing frameworks to extend them.

Andromeda Extensions

Extension points are not an Andromeda artifact, they are one of the core mechanisms of any Eclipse based application.

To provide extensions, extension points must be instantiated. An extension point can be regarded as the underlying scheme for extensions.

The instantiation of an extension point is done in the so-called "manifest file" of plugins. A manifest file kind of "describes" a plugin (name, description, extensions, dependencies etc.). To instantiate an extension point, the following steps have to be performed:

- Open file "/META-INF/MANIFEST.MF" of a plugin.
- ->"Extensions"
- ->"Add..." (Create new extension, see Figure 10.13)

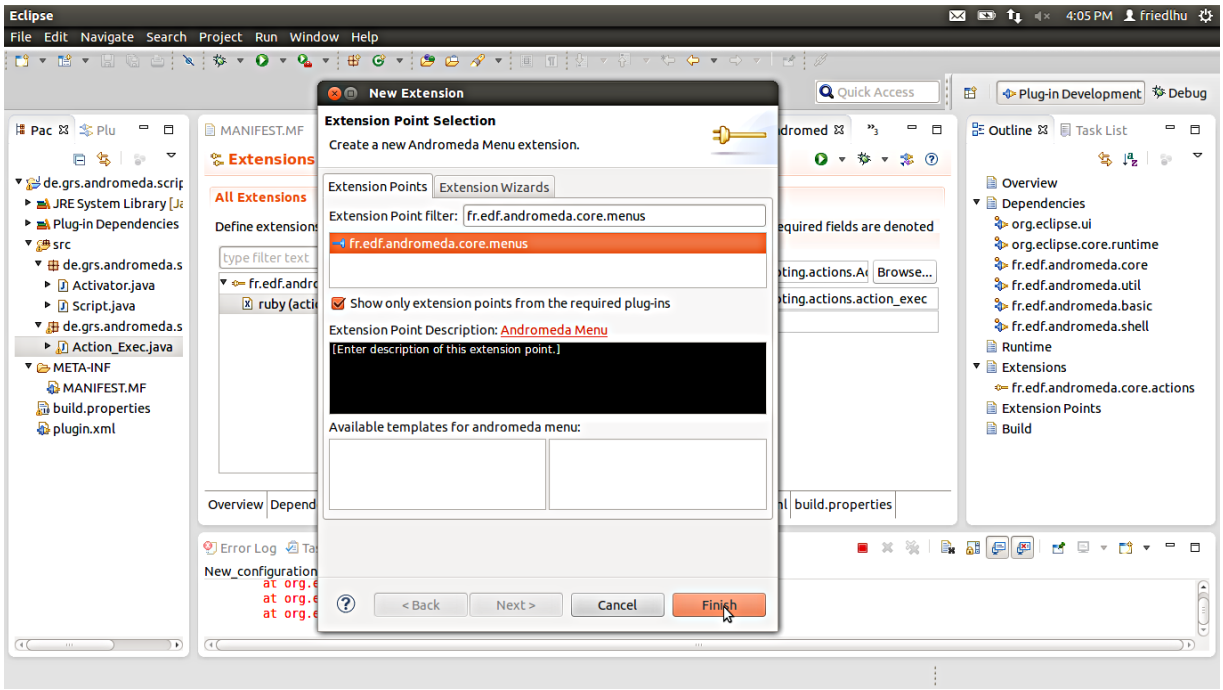


Figure 10.13. Instantiation of an extension points.

- The last step for providing an extension is to assign data. For example an extension for an "action extension point" must declare a Java class that executes the action.

Plugin developers can also define new extension points. Those extension points can be instantiated by any other plugins. An extension point requires (at minimum) to specify its scheme (via XML). How this can be done is documented in the Eclipse project. Note that the concept of extension points does not origin from Andromeda, it is an Eclipse concept.

For extending Andromeda, several extension points have been designed in Andromeda. The most interesting ones are described in the sequel:

Actions An action is a functionality that can be embedded in diagrams, in a shell (Andromeda Shell), in model documentation etc. Each action has its unique identifier.

Examples of actions are "Moving Modules", "Open Editor", "Open Diagram", "Load Model", "Save Model", "Quantify Fault Tree" etc. In total, more than 100 actions have been specified in Andromeda though only a few of them (less than 15) are mandatory for processing models in general and the rest of them are application specific.

The execution of an action depends on the kind of environment, in that an action has been embedded in.

For example, in a diagram a geometrical area can be used to trigger an action (when clicking on it). In shells, to enter an action in form of a command with arguments will launch the action.

The extension point for actions in Andromeda is:

```
fr.edf.andromeda.core.actions
```

Events Events are a concept to inform other software parts about the occurrence of an event that has been taken place. They are said to be "fired".

An event can be of any kind for example that someone clicked on a fault tree, that a quantification has finished, that a model has been saved etc. Events contain no further functionality, they only inform (declarative).

The extension point for events is:

```
fr.edf.andromeda.core.events
```

Event Handler Event handlers are the corresponding construct to react on events. They pose an asynchronous manner to execute functionality whenever an event has been fired. They disconnect event propagation from event handling.

The extension point for event handlers is:

```
fr.edf.andromeda.core.event_handler
```

Domains Domains define the underlying structure of models. Domains are provided by the extension point:

```
fr.edf.andromeda.core.domains
```

However, for adding new domains it is not sufficient to implement solely a domain extension point. New domains are (semi-) generated by a domain generation framework (see Section 10.3.3).

Currently developed domains are: *PSA Domain* (PSA models), *ESD Domain* (event sequence diagrams), *Graph Domain* (hierarchical graph models) and *FT Domain* (fault tree models, actually a subset of PSA models).

Diagram Generation The diagram generation framework derives diagrams from models (see Section 10.3.3).

Examples of diagrams in PSA models are *Fault Tree Diagrams* and *Event Tree Diagrams*. But also a cartography (see Section 8.2.3) is created in form of diagram. In comparison to fault and event tree diagrams, a cartography is a generic diagram (that can be applied to any model type).

Each diagram is generated by a diagram generator. To provide a new diagram equals therefore to provide a new diagram generator. The extension point for diagram generators is:

```
fr.edf.andromeda.core.diagram_generator
```

Toolbars Toolbars provide one or more graphical items (so-called "toolbar items"), each of them associated with some functionality that gets executed when clicking on the respective toolbar item.

Each Andromeda application can implement toolbars. Each toolbar is uniquely named (for the purpose of identification) and can be extended by other applications.

The extension point to provide toolbars and toolbar items is:

```
fr.edf.andromeda.core.toolbars
```

In order to provide toolbar items, a toolbar must be "linked" to those. For this purpose, the same extension point can declare links. An example is

```
link: id=fr.edf.andromeda.app.model_editor.toolbars
    children=fr.edf.andromeda.basic.toolbars.open_model,
             fr.edf.andromeda.basic.toolbars.export_model
```

where a toolbar *fr.edf.andromeda.app.model_editor.toolbar* is linked to two toolbar items namely *fr.edf.andromeda.core.toolbars.open_model* and *fr.edf.andromeda.core.toolbars.open_model*. The example is a real extract of Andromeda to provide functionality to open and export models in the "Model Editor" application.

The interesting point is that "link" definitions are done independently from toolbar definitions. In the example the definition of "fr.edf.andromeda.app.model_editor.toolbars" is done via a separate extension point. This way, other applications (that may be developed in a future point in time) can hook into existing toolbars.

Also, toolbar items can be directly linked to actions that are executed when clicking on an item.

The application framework provides automatically (be default) toolbars for any Andromeda applications (though configurable).

Menus Menus provide one or more so-called menu entries. Each entry is associated to some functionality that is executed when clicking on the respective entry.

The extension point for menus and menu items:

```
fr.edf.andromeda.core.menus
```

Likewise toolbars, menus are linked to their items. The item is separated form menu declarations. Additionally, menus can be linked to further menus (submenus).

Also, menu items can be directly linked to actions that are executed when clicking on an item.

Popup Menus Popups menus are quite similar to menus. The difference is the way of displaying. Popup menus will open (i.e. they display their menu entries) when right clicking inside an application.

The way of linking menus to Andromeda Applications (or Andromeda Components) is analog to the way of linking toolbars (see above).

The extension point for popup menus is:

```
fr.edf.andromeda.core.popup_menus
```

The linkage of a popup menu to its items is equivalent to the one of menus.

Model Analysis A model analysis validates model elements against a condition. The condition is provided by a Java class that must be configured for the extension point.

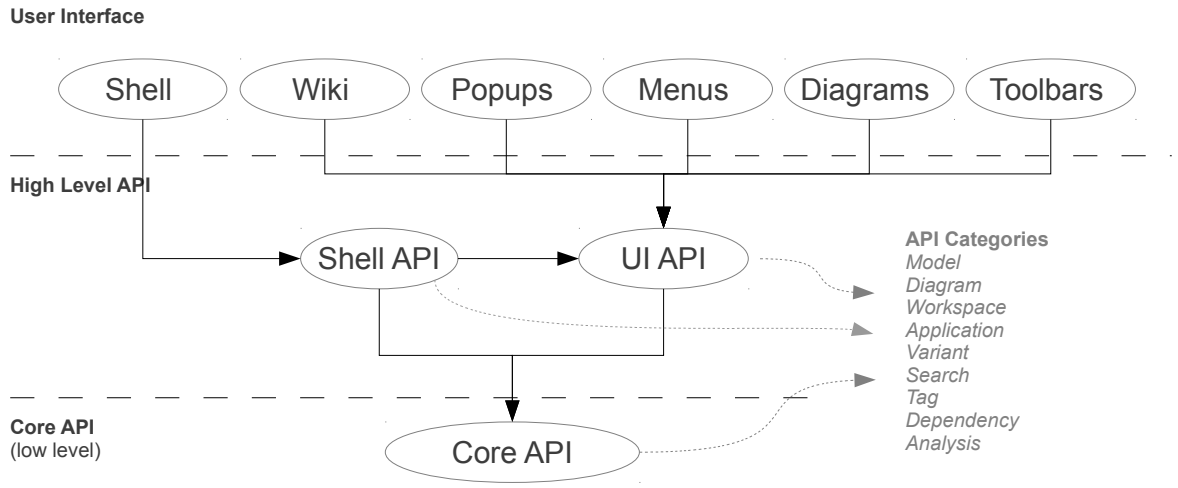
The extension point for model analysis is:

```
fr.edf.andromeda.analysis
```

Development API

Andromeda provides an API (Application Programming Interface). The API consists of several Java classes, each providing its functionality.

The following graphic shows the design of the API:



The API consists of a high level API and a core API.

The high level API can be divided into the Shell API and the UI API. Both intend to provide functionality for user interfaces, i.e. they reflect the typical use cases (scenarios) of Andromeda. The Shell API is needed as use cases of the Andromeda Shell (the shell commands) are typically processed slightly differently (mostly due to different input / output processing). Whenever possible, the Shell API reuses (wraps) the UI API to avoid redundancy.

The core API contains fundamental functionality, that is needed for the higher level API.

The next sections explain the design in detail.

User Interface The user interface provides functionality of the high level API to the user.

The user interface is not necessarily graphically. Any possibility that gives the user an opportunity to interact with Andromeda is considered as a user interface.

User interfaces are for instance diagram viewers, menus, popup menus, toolbars, wiki (an interface to display model documentation), the Andromeda native and integrated Shells etc.

User interfaces can be defined, extended or customized either in the classical way by programming or by the method of RCP extension points.

High Level API The high level API contains the implementation of typical Andromeda use cases (e.g. “Load Model”). Any functionality of the high level API is processed in three phases:

1. Pre-processing phase: In this phase, checks (keyword “thread-safe”) may be performed, complementary input data, choices or confirmations (e.g. confirmation before deleting an element) may be asked (to the user), data may be converted etc.
2. Execution phase: In this phase, the actual functionality is executed. Often, one or more functions from the Core API are called (wrapped).
3. Post-processing phase: In this phase, information, warnings or error messages may be shown to give feedback about the execution. Results of the execution may be visualized etc.

In the following, the different characteristics between the Shell API and the UI API are demonstrated by the example function “Open Model”:

The high level functionality “Open Model” of the UI API works the following way: A dialog is opened to ask the user which model to load (pre-processing phase). Then the model to open is read (by using the Core API) whereas a graphical progress bar gives feedback while parsing the model (execution phase). Finally, the model is registered and Andromeda is “refreshed” so that the model becomes visible in Andromeda (post-processing phase).

The functionality “Open Model” of the Shell API works slightly different: First, command line parameters from the Andromeda Shell are parsed to figure out which model to load (pre-processing phase). Then the model to open is read (by using the same Core API functions), whereas the progress is output (printed) into the shell while parsing the model (execution phase). Finally, the model is registered and Andromeda is “refreshed” so that the model becomes visible in the Andromeda (post-processing phase).

In the example, the pre-processing phase and the execution phases differ between the UI API and the Shell API. Nevertheless, in the execution phase, the same function “Load Model” from the Core API is called. The different ways of showing the progress is achieved by providing a different progress monitor object to the Core API function.

Note, as illustrated in the graphic above, the Shell API can use the full UI API functionality. For example, the function “Open Diagram” of the Shell API (command “open-diagram” in an Andromeda Shell) executes in return the function “Open Diagram” of the UI API to display the diagram (The function “Open Diagram” is not part of the Core API). So it is not true to say, that shell actions are limited to read and write operations (as native shells typically are). Shell actions provide the same complete set of functionality as any other UI actions.

Core API The Core API contains basic (low level) functionality. This functionality is characterized to be

- Reused by higher level functionality
- Parametrize-able in several ways
- Not directly accessed from user interfaces. Checks (e.g. thread-safe) may not be provided and input data needed to execute the function may be incomplete etc.

Naming of API classes To clarify which API classes belong to the core API and which ones to the high level API, the following naming convention is introduced:

For core API classes:
API_<CATEGORY>.java

For UI API classes: UI_API_<CATEGORY>.java

For Shell API classes: Shell_API_<CATEGORY>.java

CATEGORY subdivides the API into different categories, depending on the type of functionality. The following categories exist:

- Action: Functionality to execute Andromeda Actions.
- App: Functionality to manage Andromeda Applications.
- Dependency: Functionality to manage (in particular to resolve) module references.
- Diagram: Functionality to create, show and export diagrams.
- Model: Functionality to access and modify Andromeda models and modules.
- Search: Functionality to find model elements due to constraints.
- Tag: Functionality to tag model elements.
- Variant: Functionality to manage contexts (adaption rules and includes).
- Workspace: Functionality to create, configure and modify an Andromeda workspace (a workspace is a technical entity to store any items requires in a modular PSA: models, configurations, contexts etc.).

Example: Developing a formula viewer

Extending Andromeda by implementing extension points is one of the major principles of Andromeda. As an example a new application "Formula Viewer" is developed in the sequel that converts a fault tree into an equivalent Boolean formula. The example illustrates the principle of extending Andromeda.

At a first step, a new "view" and "perspective" extension is created to extend Andromeda to display the new application. The view serves as container to embed the application, the perspective extension serves to define the layout of the view. Both are Eclipse extension points. For more information about extending an Eclipse based application with views see the official documentation at [93].

Figure 10.14 shows the instantiation of the view and perspective extension points. The view extension point is

org.eclipse.ui.views,

The perspective extension point is

org.ecipse.ui.perspectiveExtensions,

The "class" attribute (visible on the right side in the screenshot) connects the view to the Andromeda application "Formula Viewer". To recognize the linked class as Andromeda application, the class must inherit the Andromeda interface `I_Application`. This interface provides functionality to interface with other applications, to interface with the Andromeda kernel and to access the set of models which are currently loaded in Andromeda.

The application needs to implement a function *selection_changed* to redisplay the Boolean formula whenever a user focuses on a new element.

```
public void selection_changed(I_Element e, Context c) {                (1*)
    m_label.setText("");

    if (e == null) {
        return;
    }
    if (e.get_element_type() == PSA_Element_Types.s_fault_tree)  {     (2*)
        Fault_Tree ft = (Fault_Tree) e;
```

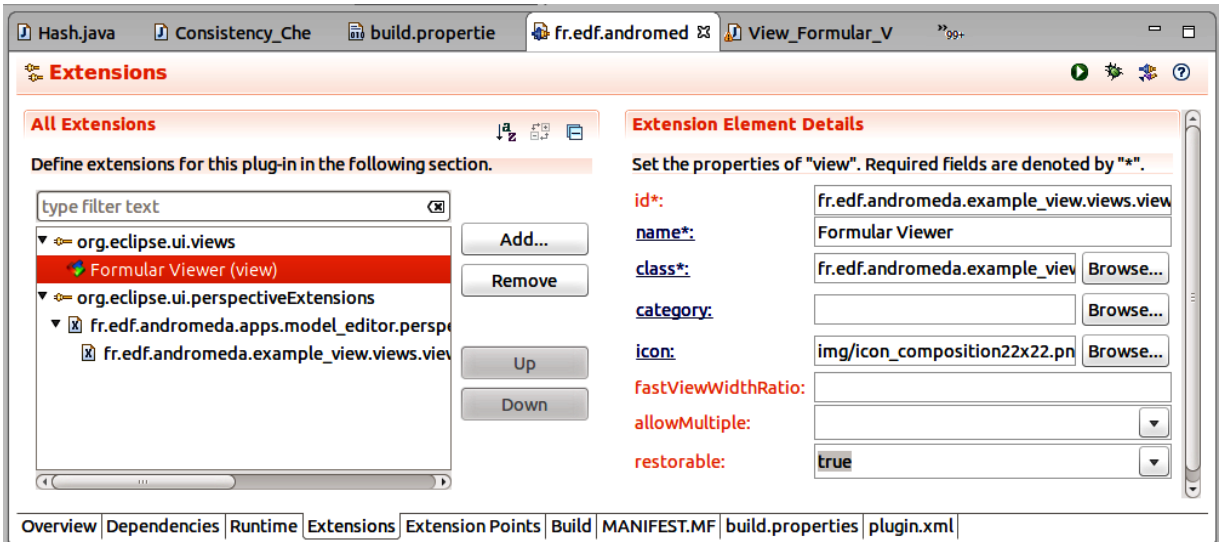


Figure 10.14. Adding a new view to Andromeda: A new view is added to Andromeda by implementing extension points.

```
        String s = get_formula(ft);                                    (3*)
        m_label.setText(ft.get_name() + " = " + s);                    (4*)
    }
}
```

(1*) Function *selection_changed* is called from the core system of Andromeda whenever the so-called "selected element" changes: The selected element represents a model element a user is currently focusing on. Typically, when clicking on an element (in any application), the "selected element" is updated automatically. This concept is provided by the Andromeda core system. The "Formula Viewer" only reacts whenever the currently select element changes to update the formula respectively.

(2*) A type check for element e is performed to ensure that the element is a fault tree component.

(3*) The function *get_formula* delivers the Boolean formula of a fault tree (as text). The algorithm is "straight forward" and not of further interest in this section.

(4*) The determined formula is displayed in a graphical form.

Figure 10.15 shows the final "Formula Viewer" integrated in Andromeda: The application determines the corresponding Boolean formula for the currently selected model element (in case the element is a fault tree). In the example, the formula has been determined for the fault tree that is shown below the formula.

Finally, the plugin that provides the new application can define new extension points (that can be instantiated by other applications). The example is extended such that the converted formula is itself implemented as extension. To do so, a new extension point is declared that requires to declare a class of the following interface:

```
boolean is_applicable(Element_Type type);
String get_formula(Element element);
```

whereas

- Function `is_applicable` returns "true" if the extension is able to convert elements of a certain type to their Boolean formula.

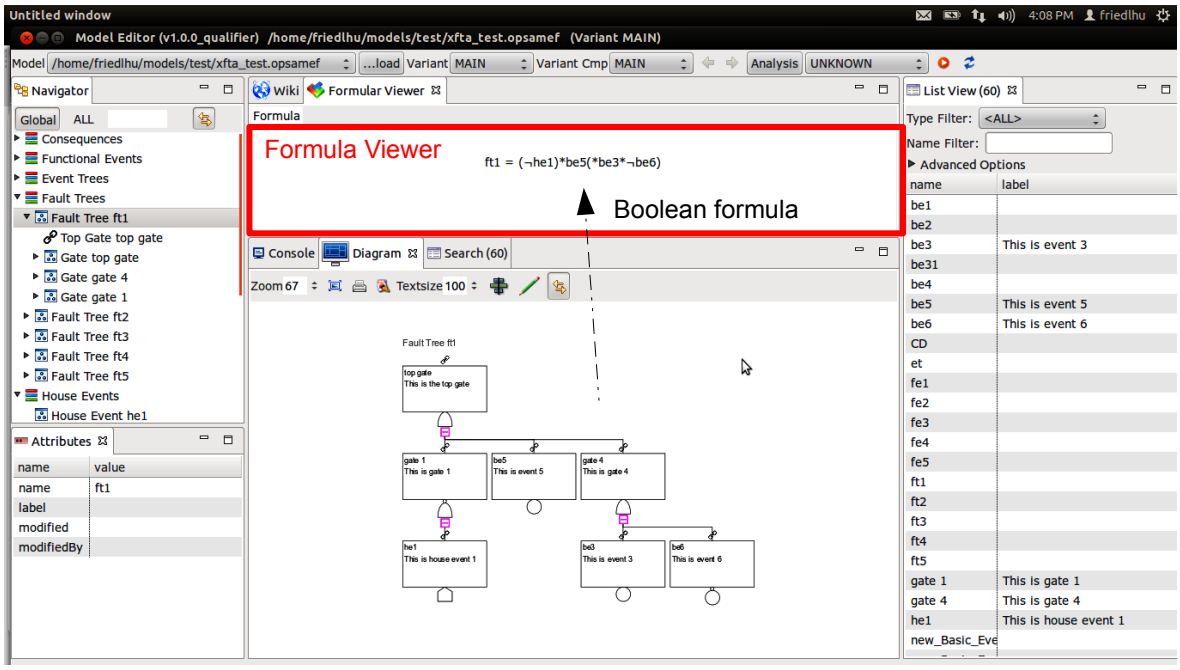


Figure 10.15. Formula Viewer: Screenshot of the developed Formula Viewer

- Function `get_formula` delivers the converted Boolean formula of an element.

Other applications can implement the extension point to convert other kind of elements (for example event trees or basic events) into their Boolean representation.

10.4.3 ADSL

In Andromeda, new domains are generated from a domain specification language. This principle has been introduced in Section 10.3.3. The domain specification developed of Andromeda is called ADSL (Andromeda Domain Specification Language).

ADSL is a language that has been developed within the Andromeda project in order to specify new domains that should get integrated in Andromeda. It is specified in textual format. The format has been designed to be simple, compact and readable.

ADSL supports two kind of inheritances. The first is domain inheritance (see 10.3.3) where parts of domains can be inherited. The second is to inherit new component definitions from existing ones. For example one could create a component “EDF Fault Tree” that extends the “Fault Tree” component by some specific properties.

In the sequel, the language of ADSL is illustrated by an example:

```
$MAIN$
MODEL: -MODEL_NAME Graph -ROOT Graph

$COMPONENTS$
Graph: -CHILD_LIST Node,Edge -PROPERTY Source,Target
Node:-PROPERTY x:String,y:String
Edge:-PROPERTY Source, Target
```

```
$PROPERTIES$
Source:-TYPE Node
Target:-TYPE Node
```

ADSL is divided into different sections. Each section begins with a section name that is enclosed in "$" symbols. In the example there are the sections *MAIN*, *COMPONENTS* and *PROPERTIES* (though there are more).

The main section specifies the domain name (Graph), the root element of the model (Graph) and a domain label.

In the *COMPONENTS* section, all component types of the domain are specified. The *CHILD_LIST* keyword is used to specify contained components types. The keyword *PROPERTY* serves to declare property types.

Properties can be specified either "inline" (within the component definition) or shared in another section called *PROPERTIES*. Each property has a value type that is important to interpret the property value (the value assigned to the property).

In the next example, domain inheritance is demonstrated. An *INCLUDE* section serves to load the ADSL specification of other domains:

```
$INCLUDE$
<DOMAIN_NAME>:-path <ADSL_FILE>
```

```
$INCLUDE$
psa:-path fr.edf.andromeda.psa/gen/config.xml

$MAIN$
MODEL: -MODEL_NAME ReqModel -ROOT ReqModel

$COMPONENTS$
ReqModel: -CHILD_LIST Req
Req:-PROPERTY description:String,ft:Fault_Tree[1,*]
```

The example defines the domain of a requirement model. A requirement has a description property. Further a property "ft" associates a list of fault trees. The expression `[1,*]` indicates a list operator over fault trees, whereas minimum one fault tree must be assigned ("*" indicates a maximum of unlimited).

The domain "*psa*" is the domain of PSA models. It is be included so that the "ft" property type can declare the fault tree type as value type.

10.5 Conclusion

In this chapter, the software Andromeda has been presented. Andromeda implements the modular PSA concept and it incorporates an open philosophy of modeling and tooling. The open philosophy is realized by an extension framework to provide companies a possibility to extend and customize PSA functionality to specific needs.

The interest of Andromeda is at first technically as it provides several functionality at model engineering to increase productivity and clearness of models. The functionality to compare models have gained industrial

interest. Apart from the technical point, Andromeda can serve to enforce the PSA community. It can provide a common platform to discuss and collaborate on a technical and standardized level. Initiatives such as the Open PSA Initiative can profit from tools like Andromeda as they can encourage software developers and risk engineers to join the community.

PART III

CONCLUSION

Modular PSA

Contributions

In this manuscript, the current way of PSA model engineering has been analysed and limitations have been pointed out. PSA models increased in size and complexity and are difficult to develop and maintain.

Several reasons could be made out that lead to the complex models of today. A first one is conceptional: The language of fault- and event trees is generally too basic and does not provide the compactness of high level modeling languages such as Figaro or AltaRica. Sometimes many modeling constructs in form of fault- and event tree are necessary to model risk whereas high level modeling languages could achieve same results by using few constructs.

Further, concepts are missing to manage evolutions (versions) and deviations (variants) of models. The current methods are not efficient and they tend to create modeling redundancy.

Another concern is software customization which is currently difficult to realize. Software vendors of commercial modeling software implement typically only those features that are commonly claimed by a majority of customers. Specific software needs of individual companies are financially not cost-effective or technically not feasible as they would conflict with software requirements of other customers.

To face the problematic of complex models, the idea of a modular approach has been elaborated. A technical framework has been introduced that enhances preliminary work of M. Hibti and the Open PSA Initiative. The principal idea of a modular PSA is to split a model into several parts called the modules. A model is assumed to be more easily handled by its modules than as a whole.

The technical framework introduces a set of notions. A modular PSA consists of various models in which each model is of a certain domain. Each domain is characteristic for a specific type of models. PSA models are in fact only one of several model types which can reside in a modular PSA. Further model types for a modular PSA have been presented throughout this thesis, for example graph models, ESD models, quantification models and cutset models.

The powerful instrument of a modular PSA is a context sensitive instantiation technique to express specific circumstances in models. Models are developed in a generic form and then instantiated in dependence to a context. At instantiation, modules can be flavored (technically property values of components are adapted) and the resolution of module dependencies can be managed (technically reference resolution between components is impacted). Those techniques permit to clearly separate the development of model content (the set of available model assets) from model assemling (the decision which model assets are combined together and in which manner).

A modular PSA leads to a set of new concepts for model management. A variant management has been presented to manage the deviation of models. It has been shown how to adapt models to specific circumstances for example to a specific nuclear power plant. Technically, variants use the instantiation technique of a modular PSA to express model deviations. The concept has been demonstrated by different use cases, for example the deactivation of function events in event trees or the deactivation of gates in fault trees.

A method to compare models has been presented. The result of a model comparison are the differences between two models in form of so-called “matches”. Each match represents a pair of matched modules. An associated match type indicates the kind of differences between the modules. An algorithm has been presented to determine the list of matches and their match types. The differences between two models can serve to generate modification reports (for example for safety authorities). Further, it gives

model engineers important feedback to verify modifications. An implementation in Andromeda has found industrial interest at EDF.

The principle to fusion models in a modular PSA has been explained. Model fusion represents a form of concurrent model engineering which is supposed to play a key role for future PSA model development in order to cope with an increasing model complexity. Model fusion is a generic concept to merge two (or more) models. Two merge strategies have been introduced. The first strategy is the *two way merge* and targets to simply replicate modules from two source modules into a target model. However, the decision which modules to replicate is thereby typically taken manually by a model engineer or model integrator. The second strategy is the *three way merge*, where an ancient model state (the ancestor) is additionally taken into account to automate the two way merge in large parts. However, the *two way merge* gives full control about the merging process and therefore can be preferable.

Another focus has been given to model complexity in general and the understanding of complex models. A modular approach offers new ways to visualize relations between model elements. Methods to highlight relations between model objects have been shown. For example, a cartography could visualize for- and backward dependencies of modules. In PSA models, it can display the set of parameters used in a certain fault tree or the set of fault trees used in a certain event tree. Further, a method to visualize system models in form of a unique and simple language (graphs) has been shown. As fault trees are typically constructed for system models, their visualization can contribute to understand the composition of fault trees. Note however that the current definition of graph models is an initial work and should be extended to model more complicated systems.

The characteristics provided by the modular PSA framework enhance not only model development, but several other phases of model engineering such as model design, integration, verification and maintenance. A new modeling language named ESD models has been presented which can be used at a design phase to specify the evolution of events. Further, a method has been shown to generate event trees of ESD models. This signifies an important step to generate PSA models from higher level modeling languages. The automation of processes by shell commands and scripts has been shown. Scripts can contain a set of instructions. Their execution is efficient and reproducible what gives an opportunity to automate modeling activities of various kinds. In future, scripts can lead to a significant increase of productivity at model engineering.

Limitations and Perspectives

A modular PSA treats references in a dynamic manner: An address resolver is required to resolve references due to specific contexts in order to obtain corresponding components. The dynamic address resolving procedure requires references to indicate the name and the type of the component to reference and this poses two limitations.

The first limitation is that one must be careful whenever components get renamed. Software should provide functionality to ask a user if references should be renamed accordingly whenever components get renamed. Note also that the set of references which refers to a certain component is context dependent. A model engineer modeling solely in a specific context (e.g. for a certain model variant) may not correctly rename the references for all contexts. In addition, models may be commonly used (shared) in different modular PSAs and software may only rename the references in one particular modular PSA. Model engineers must pay attention to “update” any model impacted by a renaming procedure.

A second limitation is that the current definition of references does not permit to define static references. The idea of those would be to model “hard links” between model elements which do not depend on a

specific context. Static and dynamic references could coexist in one model: Static ones would express fixed relations between model elements (which will never change no matter the current context) and dynamic references would express adaptable relations (which depend on a specific context).

The reason why static references were not considered in the modular PSA is that they reflect the state of the art modeling and the thesis tried to focus on dynamic references. Further, the simplicity of the fault and event tree language poses (up to now) no particular problems to express any references in a dynamic manner. However, to model in more complicated modeling languages, static references should become part of the framework.

Further, in the scope of the thesis, the manuscript focused uniquely on modeling PSA, Graph and ESD models though it is a generic framework that may serve to model other languages. However, it is yet unclear whether it scales with more complicated (higher) languages such as Figaro or AltaRica. PSA and graph models consist of rather simple constructs. Higher level modeling languages may require to extend the current framework by new constructs. For example, in order to avoid address conflicts (where components of the same type and name are conflicting), scope limiting techniques such as namespaces or an enhanced address resolution to resolve dynamic references due to additional constraints (and not only due to name and type) should be considered. However, concerning the PSA models used at EDF, naming conflicts are currently rare (due to strong naming conventions that are to apply at PSA modeling at EDF).

Finally, it is not yet clear which impact generic modeling has on the daily work of model engineers. The manuscript proposed to introduce roles such as developers and integrators. At software engineering, those roles have successfully found application and acceptance. They reflect the idea to separate between developing content assets (the job of developers) and assembling / instantiating a product from assets (the job of integrators). However, it must be analyzed whether model engineers and companies are ready to accept and afford a kind of separation in the model engineering process.

Andromeda

Contributions

The research tool Andromeda has been extensively developed throughout the thesis. Andromeda is a generic model engineering platform, developed at EDF R&D, that implements the concept of a modular PSA. Almost all features and concepts demonstrated in this thesis have been implemented and tested in Andromeda.

Andromeda differs not only in the modeling approach from other tools, but also in its tooling and development philosophy. The software is designed to be highly extensible. For this purpose it is totally based on plugins and each plugin provides functionality. New plugins can be developed and added or removed, what can customize the software to specific requirements of customers. Several frameworks have been developed to facilitate the development of extensions.

Andromeda is not limited to the domain of PSA models, it is constructed independently from specific models or applications. Generic concepts such as model comparison, model fusion, variant- and version management, cartography etc. can be applied for any model that is based on the modular PSA approach.

PSA models at EDF are developed and quantified with the software *RiskSpectrum PSA* [22]. In fact, RiskSpectrum PSA constitutes only one of several (commercial) software products of a whole product family (called *RiskSpectrum*). It has gained high application in the field of nuclear power plants (mainly

European ones).

However, in the engineering divisions at EDF, functionality is required to manage evolutions of models. Ideas to manage PSA models with variants and version have been described in Chapter 7. Currently, commercial PSA tools do not provide the often customer specific functions. In this context, Andromeda could reveal new opportunities in particular of comparing PSA models and generating modification reports.

Another interesting characteristic of Andromeda is its capacity to generate models (or parts of them). In Section 9.2, event tree diagrams were introduced to generate event trees. Fault trees can be generated from system models. And a scripting interface has been demonstrated in Section 9.3 to automate modeling activities and to speedup and customize the process of model engineering. In future, functionality to automate processes are rated to become even more important. Automation poses one of the effective instruments to face model complexity and increasing model sizes.

Finally, the modular architecture of Andromeda permits to specifically adapt the software to the needs of customers. Customization is considered as a core requirement to cope with the complexity of PSA models. Andromeda can find usage to complete functionality of popular software tools that are often limited to respond to problems on a (too) general level. For example, to find model elements due to certain characteristics can become very specific. Andromeda plugins can take the specific context, history and experiences of companies into account when responding to problems.

Limitations and Perspectives

Software Aspects

Andromeda is an Eclipse RCP (Rich Client Platform) application and has been designed upon a plugin architecture. The plugin architecture facilitates to customize functionality and to enable an integrated software approach (where various functionality is proposed within the same software tool).

Nevertheless, the plugin architecture limits the scope to develop other software applications (than Andromeda) which are not based on the concept of plugins. For example one could consider to create web applications or "light weight" applications dedicated to provide a specific functionality (and which should not require the RCP architecture in order to limit application sizes and to facilitate application deployment). Engineers may not want to deal with the overall set of functionality and prefer to limit the scope to the necessary. Web applications would have the advantage to avoid application installation and updates. They provide a common and familiar user interface and they may ease concurrent model engineering (models could be kept on server side and commonly processed by different model engineers at a time). Therefore, in a future version of Andromeda, an architectural separation may be applied in order to separate RCP dependent functionality from general (non RCP dependent) functionality. The latter could then more easily be provided to other software tools (for example in the form of a library).

Another challenge is to provide Andromeda (or a future version of it) to the PSA community (for example in the form of a "Open Source" project). Beside strategical and political issues to consider, also technically Andromeda must be prepared for this step. Other companies and developers may join the project. Often they may just want to add some extensions to Andromeda but not to deal with the core functionality. Therefore, a clear API (Application Programming Interface) is required to separate the development of architectural parts from the programming of Andromeda extensions. Whereas a separation on plugin level is already performed in large parts, a clear documented API is still missing and a deployment strategy is required to release the core (the kernel) of Andromeda without (or a minimal set of) extensions.

PSA Community

The future hope of Andromeda is to enforce the international PSA community to rely on a common modeling technique when communicating and clarifying problems. Currently, collaborations for risk assessment exist already. For example, EDF exchanges ideas and discusses problems with engineers using CAFTA (a popular (commercial) PSA software used for U.S. nuclear plants). However, the use of different PSA software (EDF uses RiskSpectrum) complicates collaborations. And converting PSA models between tools still poses a major problem (mainly politically).

To successfully distribute a common technique like a **standard** requires to provide techniques or ideas without commercial interest. Otherwise companies are generally too reluctant to accept new standards. The next step at EDF is therefore to convert Andromeda from a research project into an "Open Source" or "Freeware" project[5]. In any case, the modular architecture can encourage developers to develop extensions in form of plugins. The long-term interest of EDF is to profit from extensions developed at other PSA related companies that contribute to the project.

The major technical point that can enforce communities is however not the software Andromeda itself but its philosophy to enhance clearness of models and to open and **standardize** interfaces. The initiative "Open PSA" [43] has noticed the need for open standards since a while. That's why the "Open PSA Model Exchange Format" (OPSAMEF) [4] has been developed and is still improved (currently OPSAMEF version 3.0 is specified). But initiatives and standards need also tools or at least *reference tools* [6] to animate the community. The future role of Andromeda can indeed be to establish a common platform of distribution and discussion to test and evaluate new approaches within the PSA community.

[5] Whereas both target to distribute a software at no charge, the difference between "Open Source " and "Freeware" is more or less that "Open Source" publishes also source code and "Freeware" does not (though this depends on license details).

[6] A reference tool serves as an exemplary implementation of a software or a standard so that tool developers can verify to be coherent with a common concept or standard.

PART IV

APPENDICES

APPENDIX A

Images and Figures

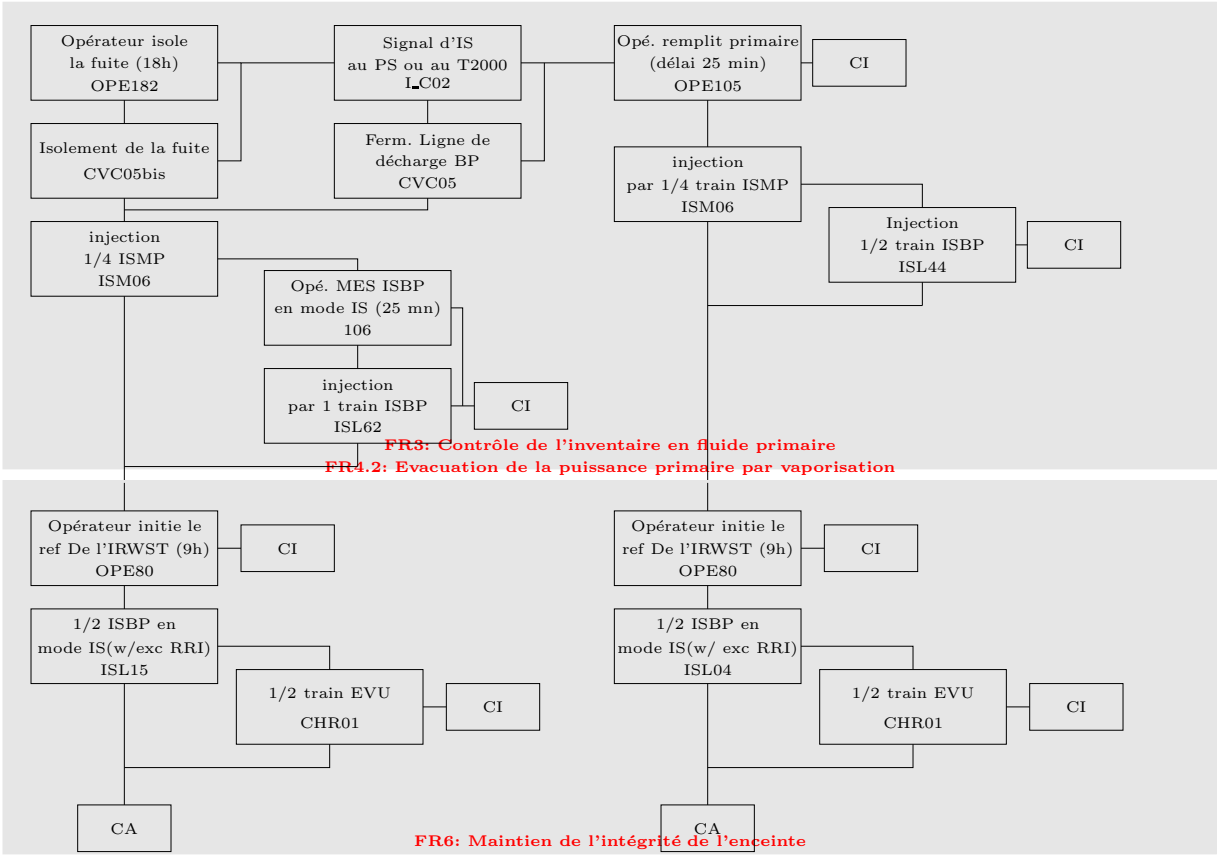
A.1 Scheme for Andromeda Adaption Rules

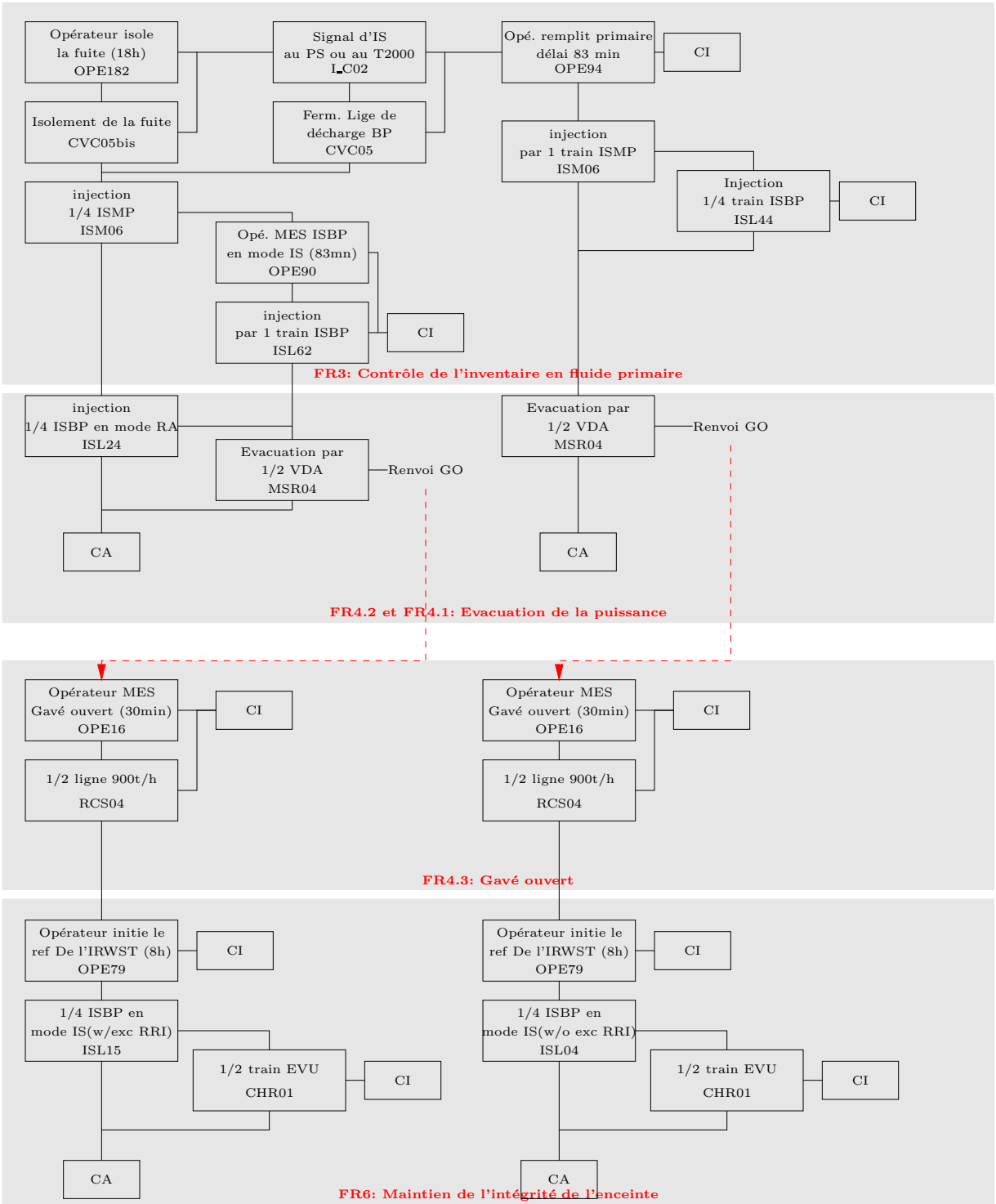
```
<VARIANT> ::= <VARIANT-NAME> [ <RULE-LIST> ]
<RULE-LIST>  ::= <NEW-LINE> <RULE> [ RULE-LIST]
<RULE> ::= <SELECTOR-LIST> ':' <PROPERTY-TYPE>  <ACTION> <ACTION-VALUE>
<SELECTOR-LIST> ::= <SELECTOR> [ | '/' <SELECTOR-LIST> ]
<SELECTOR> ::= <SELECTOR-TYPE> ['[' CONDITION-LIST ']' ]
<SELECTOR-TYPE> ::= '*' | <ELEMENT-TYPE>
<CONDITION-LIST> ::= <CONDITION> [ | ',' <CONDITION-LIST> ]
<CONDITION>:= <PROPERTY-TYPE> '=' ( <PROPERTY-VALUE> | <PROPERTY-MATCH> )
<ACTION> ::= 'SET' | 'REPLACE'
<ACTION-VALUE> ::= <STRING-LIST>
<STRING-LIST> ::= String  [' ' <STRING-LIST>]

<NEW-LINE> ::= \\n
<VARIANT-NAME> ::= String
<PROPERTY-VALUE> ::= String
<PROPERTY-TYPE> ::= String
<ELEMENT-TYPE ::= String
<PROPERTY-MATCH> ::= '/' Regular-Expression '/'
```

A.2 Components of a PSA Model

element type	module
Fault tree	true
Gate	false
basic event	true
parameter	true
CCF Group	true
Event tree	true
Initiating Event	true
Functional Event	true
Sequence	true
Consequence	true
Rule	true
Path	false
Fork	false
Alternative	false
Initial State	false
Members	false
Factors	false
Substitution	false
Cdf	false
Constant	false

A.3 Traditional Event Sequence Diagrams





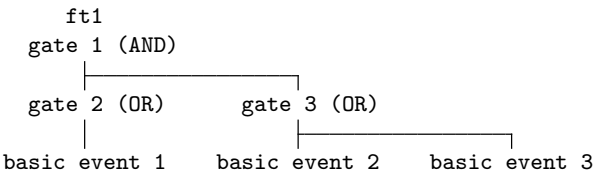
A.4 Pseudo Code of Reference Resolver

```
function resolve(reference):
resolve(reference, reference->model)

function resolve(reference,  model):
```

```
  Model model  = get_model_of(reference)
  includes = get_includes(model)

  for include in includes do
    Model model = include->model;

    if (include->is_recursive) do
     resolve(reference, include->model)
    else
      element = find(reference, model)
      if (element<>unresolved_element)
        return element
      fi
  fi
  end for
  return unresolve_element

function find(reference, model):
  for element in model->elements do
    if (element.name == reference.name AND
        element.type == reference.referencing_type)
      return element
    fi
  end for
  return unresolved_element
```

A.5 Ruby Script for automatic fault tree generation

As an example to demonstrate the JRuby scripting interface, the following fault tree is developed:



The corresponding JRuby script:

```
new_fault_tree = model.create_child("new_fault_tree",
PSA_Element_Types.s_fault_tree, nil)
# create gates
gate_1 = new_fault_tree.create_child("gate_1", PSA_Element_Types.s_gate, nil)
# set as and gate
gate_1.set_attribute(PSA_Attribute_Types.s_gatetype, "and")

gate_2 = new_fault_tree.create_child("gate_2", PSA_Element_Types.s_gate, nil)
# set as or gate
gate_2.set_attribute(PSA_Attribute_Types.s_gatetype, "or")

gate_3 = new_fault_tree.create_child("gate_3", PSA_Element_Types.s_gate, nil)
# set as or gate
gate_3.set_attribute(PSA_Attribute_Types.s_gatetype, "or")

# create basic events
be_1 = model.create_child("basic_event_1", PSA_Element_Types.s_basic_event, nil)
be_2 = model.create_child("basic_event_2", PSA_Element_Types.s_basic_event, nil)
```

```
be_3 = model.create_child("basic_event_3", PSA_Element_Types.s_basic_event, nil)

# set top gate to gate 1 (name of top gate must be equal to the name of
# gate 1)
top_gate_reference = new_fault_tree.create_child(gate_1.get_name(),
PSA_Element_Types.s_top_gate_reference, nil)

# add gate 2 and gate 3 to gate 1
gate_1.create_child(gate_2.get_name(), PSA_Element_Types.s_gate_reference, nil)
gate_1.create_child(gate_3.get_name(), PSA_Element_Types.s_gate_reference, nil)

# add basic event 1 to gate 2
gate_2.create_child(be_1.get_name(), PSA_Element_Types.s_basic_event_reference, nil)

# add basic event 2 and basic event 3 to gate 3
gate_3.create_child(be_2.get_name(), PSA_Element_Types.s_basic_event_reference, nil)
gate_3.create_child(be_3.get_name(), PSA_Element_Types.s_basic_event_reference, nil)
```

A.6 List of Shell Commands

Command	Description
add-config	Add workspace configuration
andromeda-install	Install new Andromeda features
andromeda-update	Update Andromeda
convertrs	Convert PSA models
cd	Change folder
cp	Copy modules
create	Create new module
diff-diagram	Compare diagrams
diff-gui	Compare two modules graphically
diff-model	Compare two models
diff-text	Compare two modules textually
edit-documentation	Edit model documentation
edit-ext	Edit module with an external editor
exec	Execute script
exec-cmd	Execute system command
export	Export module(s) to filesystem
find	Find any modules that conform to a search expression
getattr	Get an attribute value of module
import	Import module(s)
jfta	Quantify fault- and event trees with JFTA
load	Load a model
load-variant	Load a variant
ls	List modules of the current folder
lsconfig	List workspace configurations
lsmodel	List workspace models
mkdir	Create folder
mv	Move module
open	Open diagram
open-preferences	Open preferences
open-wiki	View documentation
open-workspace	Load workspace
pwd	Output current shell location
quit	Quit Terminal
rm	Remove module
save	Save model
select	Focus on module (highlight)
setattr	Select module attribute
shell	Open another shell
switch-config	Switch workspace configuration
xfta	Quantify fault- and event trees with XFTA

A.7 Development Milestones

Important architectural milestones:

1. 06.2011: Development started to create a library for the creation of applications that target to analyse PSA models.
2. 03.2012: With the new domain type of *Event Sequence Diagrams*, the application got independent from the domain of *PSA Models*. A new Framework was added to generate new Model Domains from a Domain Specification.
3. 07.2012. Decision taken to base on Eclipse RCP framework. Development started to convert project to a RCP Eclipse application.

4. 12.2012: Application Framework “Andromeda” is born: Andromeda Applications can be build in modular way (by the means of Andromeda extension points)
5. 04.2013: With the introduction of Andromeda Projects, where a project joins and reuses several Andromeda Models, the architecture was adapted to realize integrated model engineering.
6. 10.2013: Introduction of Andromeda Workspaces

Important feature milestones:

1. 06.2011 Export of SVG Graphics for a PSA model
2. 09.2011 Model Navigation and Diagram Viewer for PSA models added
3. 10.2011 Variant Management and Variant Comparison for PSA models added
4. 11.2011 Model Documentation and Model Linking added
5. 03.2012 New domain for event sequence diagrams added
6. 04.2012 Variant Management and Model Documentation become generic (domain independent).
7. 07.2012 Development of Model Comparison started
8. 04.2013 New domain type *Graphs* added
9. 04.2013 Concept of Andromeda Projects added
10. 05.2013 New domain type *Fault Trees* added (subset of PSA models)
11. 06.2013 New concept of Variant Management, that stores variants independent from models
12. 07.2013 Development of Model Integration
13. 09.2013 Development start of Andromeda Command Line (Batch)
14. 10.2013 Development of Andromeda Workspace concept

List of Figures

List of Tables

List of Publications

- T. Prosvirnova, M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, and A. Rauzy. The altarica 3.0 project for model-based safety assessment. In *Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS 2013*, York (Great Britain), September 2013. IFAC.
- Ait-Ferhat D., Friedlhuber T. & Hibti M. An andromeda extension for network based safety assessment. In *American Nuclear Society Winter meeting on Risk Management for Complex Socio-technical Systems*, 2013.
- T. Friedlhuber, M. Hibti, and A. Rauzy. Reinforcement of qualitative risk assessment - proposals from computer science. In A. Yamaguchi, editor, *Proceedings of PSAM Topical Conference in Tokyo*, April 2013.
- T. Friedlhuber, M. Hibti, and A. Rauzy. Automated generation of event trees from event sequence/-functional block diagrams and optimisation issues. In A. Yamaguchi, editor, *Proceedings of PSAM Topical Conference in Tokyo*, April 2013.
- T. Friedlhuber. An integrated approach of model based safety engineering. In *The 3rd International Workshop on Model Based Safety Assessment*, March 2013.
- Friedlhuber T. Andromeda - a new open psa workbench. In *Open PSA Workshop 2012*. EDF, December 2012.
- T. Friedlhuber, M. Hibti, and A. Rauzy. Documentation d'une etude probabiliste de sûreté. In Institut pour la Maîtrise des Risques, editor, *18ème Congrès de Maîtrise des Risques et Sûreté de Fonctionnement*, October 2012.
- T. Friedlhuber, M. Hibti, and A. Rauzy. Variant management in a modular psa. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, June 2012.
- T. Friedlhuber, M. Hibti, and A. Rauzy. Overview of the open psa platform. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, June 2012.

Bibliography

[1] Electricité De France. Edf. http://www.edf.com.

[2] Laboratoire d'Informatique de l'École polytechnique. Lix. http://www.lix.polytechnique.fr.

[3] École polytechnique. http://www.polytechnique.edu.

[4] Epstein S. & Rauzy A. Standard model representation format for probabilistic safety assessment, version 2.d. Technical report, The Open PSA Initiative, 2007-2008.

[5] Hibti M. Vers une EPS modulaire [Towards a modular PSA] . In *Actes du congré lambda/mu*, La Rochelle, Octobre 2010.

[6] O. Nusbaumer and A. Rauzy. Fault tree linking versus event tree linking approaches: A mathematical and algorithmic reconciliation. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, June 2012.

[7] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl. Fault tree handbook. Technical report, DTIC Document, 1981.

[8] Wen-Shing Lee, DL Grosh, Frank A Tillman, and Chang H Lie. Fault tree analysis, methods, and applications a review. *Reliability, IEEE Transactions on*, 34(3):194–203, 1985.

[9] Antoine Rauzy. New algorithms for fault trees analysis. *Reliability Engineering & System Safety*, 40(3):203–211, 1993.

[10] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Academic press, 2001.

[11] Antoine Rauzy and Yves Dutuit. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. *Reliability Engineering & System Safety*, 58(2):127–144, 1997.

[12] Tim Bedford and Roger Cooke. *Probabilistic risk analysis: foundations and methods*. Cambridge University Press, 2001.

[13] Mark Stewart and Robert E Melchers. *Probabilistic risk assessment of engineering systems*, volume 2. Springer, 1997.

[14] Michael Stamatelatos, Homayoon Dezfuli, George Apostolakis, Chester Everline, Sergio Guarro, Donovan Mathias, Ali Mosleh, Todd Paulos, David Riha, Curtis Smith, et al. Probabilistic risk assessment procedures guide for nasa managers and practitioners. 2011.

[15] Ioannis A Papazoglou, Zoe Nivolianitou, Olga Aneziris, and Michalis Christou. Probabilistic safety analysis in chemical installations. *Journal of Loss Prevention in the Process Industries*, 5(3):181–191, 1992.

[16] Stephen R Watson. The meaning of probability in probabilistic safety analysis. *Reliability Engineering & System Safety*, 45(3):261–269, 1994.

[17] Marvin Rausand. *Risk assessment: theory, methods, and applications*, volume 115. John Wiley & Sons, 2013.

[18] JW Hickman et al. Pra procedures guide: a guide to the performance of probabilistic risk assessments for nuclear power plants. *NUREG/CR*, 2300, 1983.

[19] A. Rauzy. *An Open-PSA Fault Tree Engine.* Open PSA Initiative, 2012.

[20] Y. Dutuit and A. Rauzy. Importance factors of coherent systems: a review. *To appear in Journal of Risk and Reliability*, 2013.

[21] A Rauzy. Anatomy of an efficient fault tree assessment engine. In *Proceedings of international joint conference PSAM*, volume 11, 2012.

[22] Riskspectrum. http://www.riskspectrum.com/.

[23] Electrical Power Research Institute. Epri. http://www.epri.com.

[24] CL Smith, ST Wood, WJ Galyean, JA Schroeder, and MB Sattison. Saphire 8 volume 2-technical reference. Technical report, Idaho National Laboratory (INL), 2011.

[25] DJ Wakefield, SA Epstein, Yongjie Xiong, and Mr Kamyar Nouri. Riskman^{TM}, celebrating 20+ years of excellence. In *Proceedings of PSAM'10 conference*, pages 7–11, 2010.

[26] Teemu Mätäsniemi. Probabilistic risk analysis and decision making with finpsa. *Research highlights in safety & security*, page 76.

[27] Ilkka Niemelä Tero Tyrväinen. IC modelling in FinPSA software, 2012.

[28] MR Hayns. The evolution of probabilistic risk assessment in the nuclear industry. *Process Safety and Environmental Protection*, 77(3):117–142, 1999.

[29] William Keller and Mohammad Modarres. A historical overview of probabilistic risk assessment development and its use in the nuclear power industry: a tribute to the late professor norman carl rasmussen. *Reliability Engineering & System Safety*, 89(3):271–285, 2005.

[30] Elisabeth Paté-Cornell and Robin Dillon. Probabilistic risk analysis for the nasa space shuttle: a brief history and current work. *Reliability Engineering & System Safety*, 74(3):345–352, 2001.

[31] AF Hixenbaugh. Fault tree for safety. Technical report, DTIC Document, 1968.

[32] A. Rauzy. Mathematical Foundation of Minimal Cutsets. *IEEE Transactions on Reliability*, 50(4):389–396, december 2001.

[33] A. Rauzy. An Brief Introduction to Binary Decision Diagrams. *RAIRO-APII-JESA*, 30(8):1033–1051, 1996.

[34] US Nuclear Regulatory Commission et al. Reactor safety study: An assessment of accident risks in us commercial nuclear power plants. appendix vii, viii, ix and x. 1975.

[35] Harold Walter Lewis, Robert J Budnitz, WD Rowe, HJC Kouts, F Von Hippel, WB Loewenstein, and F Zachariasen. Risk assessment review group report to the us nuclear regulatory commission. *Nuclear Science, IEEE Transactions on*, 26(5):4686–4690, 1979.

[36] GR Corey. A brief review of the accident at three mile island. *IAEA Bulletin*, 21(5):54–59, 1979.

[37] United States. President's Commission on the Accident at Three Mile Island. *The need for change, the legacy of TMI: report of the President's Commission on the Accident at Three Mile Island*. The Commission, 1979.

[38] Mitchell Rogovin. Three mile island: A report to the commissioners and to the public. Technical report, Nuclear Regulatory Commission, Washington, DC (USA), 1979.

[39] George T Heineman and William T Council. Component-based software engineering: putting the pieces together. 2001.

[40] Ivica Crnkovic. Component-based software engineering—new challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[41] T. Prosvirnova, M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, and A. Rauzy. The altarica 3.0 project for model-based safety assessment. In *Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS 2013*, York (Great Britain), September 2013. IFAC.

[42] Marc Bouissou, Henri Bouhadana, Marc Bannelier, and Nathalie Villatte. Knowledge modelling and reliability processing: Presentation of the figaro language and associated tools. In *IFAC/IFIP/EWIC-S/SRE Symposium*, pages 69–75, 1991.

[43] S Epstein, A Rauzy, and FM Reinhart. The open psa initiative for next generation probabilistic safety assessment. *Kerntechnik*, 74(3):101–105, 2009.

[44] Lawrence Rabiner and Biing-Hwang Juang. An introduction to hidden markov models. *ASSP Magazine, IEEE*, 3(1):4–16, 1986.

[45] J Bechta Dugan, Salvatore J Bavuso, and Mark A Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *Reliability, IEEE Transactions on*, 41(3):363–377, 1992.

[46] Bouissou M. & Bon J-L. . A new formalism that combines advantages of fault-trees and markov models: Boolean logic driven markov processes. *Reliability Engineering and System Safety page 149-163*, 82(Issue 02):149–163, November 03.

[47] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998.

[48] Steven Epsteina, F Mark Reinhartb, and Antoine Rauzyc. Validation project for the open-psa model exchange using riskspectrum® and cafta®. In *Proceedings of 10th International Probabilistic Safety Assessment and Management Conference*, pages 25–29, 2010.

[49] T. Friedlhuber, M. Hibti, and A. Rauzy. Variant management in a modular psa. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, June 2012.

[50] Dalgarno M & Beuche D. Variant management. In *3rd British Computer Society Configuration Management Specialist Group Conference*, May 2007.

[51] Beuche D., Papajewski H. & Schröder-Preikschat W. Variability management with feature models. Technical report, University Magdeburg.

[52] Apel S. & Kästner C. An overview of feature-oriented software development. Technical report, Journal of Object Technology, July 2009.

[53] Czarnecki K. & Antkiewicz M. . Mapping features to models: A template approach based on superimposed variants. Technical report, University of Waterloo.

[54] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

[55] Scott Chacon, editor. *Git Internals*. PeepCode Press, 2008.

[56] Scott Chacon and Junio C Hamano. *Pro git*, volume 288. Springer, 2009.

[57] Simon Yuill. Concurrent versions system. *Software Studies: A Lexicon*, page 64, 2008.

[58] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. *Version control with subversion.* " O'Reilly Media, Inc.", 2004.

[59] T. Friedlhuber, M. Hibti, and A. Rauzy. Reinforcement of qualitative risk assessment - proposals from computer science. In A. Yamaguchi, editor, *Proceedings of PSAM Topical Conference in Tokyo*, April 2013.

[60] T. Friedlhuber, M. Hibti, and A. Rauzy. Documentation d'une etude probabiliste de sûreté. In Institut pour la Maîtrise des Risques, editor, *18ème Congrès de Maîtrise des Risques et Sûreté de Fonctionnement*, October 2012.

[61] Tim Berners-Lee, Larry Masinter, Mark McCahill, et al. Uniform resource locators (url). 1994.

[62] Tim Berners-Lee and Dan Connolly. Hypertext markup language-2.0. Technical report, RFC 1866, November, 1995.

[63] Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.

[64] M. Pillière M. Gallois. Benefits expected from automatic studies with kb3 in psas at edf. *Proceedings of PSA99, Washington*, 1999.

[65] Ait-Ferhat D., Friedlhuber T. & Hibti M. An andromeda extension for network based safety assessment. In *American Nuclear Society Winter meeting on Risk Management for Complex Socio-technical Systems*, 2013.

[66] Marc BOUISSOU. Digraphs: Le retour the come-back of digraphs.

[67] Ioannis A. Papazoglou. Functional Block Diagrams and Automated construction of Event Trees. *Reliability Engineering & System Safety*, 61(3):185 – 214, 1998.

[68] Robert C Gentleman, Vincent J Carey, Douglas M Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, Laurent Gautier, Yongchao Ge, Jeff Gentry, et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology*, 5(10):R80, 2004.

[69] Ivar Jacobson, Grady Booch, James Rumbaugh, James Rumbaugh, and Grady Booch. *The unified software development process*, volume 1. Addison-Wesley Reading, 1999.

[70] International Atomic Energy Agency. *Development and Application of Level 1 Probabilistic Safety Assessment for Nuclear Power Plants*. Publication (STI/PUB). International Atomic Energy Agency, 2010.

[71] Klaus Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[72] Burton E. Swanson. The dimensions of maintenance. In *Intl. Conf. on Software Engineering*, pages 492–497. IEEE Computer Society, 1976.

[73] S Swaminathan and C Smidts. The event sequence diagram framework for dynamic probabilistic risk assessment. *Reliability Engineering & System Safety*, 63(1):73–90, 1999.

[74] S Swaminathan and C Smidts. The mathematical formulation for the event sequence diagram framework. *Reliability Engineering & System Safety*, 65(2):103–118, 1999.

[75] T. Friedlhuber, M. Hibti, and A. Rauzy. Automated generation of event trees from event sequence/-functional block diagrams and optimisation issues. In A. Yamaguchi, editor, *Proceedings of PSAM Topical Conference in Tokyo*, April 2013.

[76] S Skiena. Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pages 225–227, 1990.

[77] Charles O Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 2011.

[78] Samuele Pedroni and Noel Rappin. *Jython essentials*. " O'Reilly Media, Inc.", 2002.

[79] Anne Dutfoy, Ivan Dutka-Malen, Alberto Pasanisi, Régis Lebrun, Fabien Mangeant, Jayant Sen Gupta, Maurice Pendola, Thierry Yalamas, et al. Openturns, an open source initiative to treat uncertainties, risks' n statistics in a structured industrial approach. In *41èmes Journées de Statistique, SFdS, Bordeaux*, 2009.

[80] Friedlhuber T. Andromeda - a new open psa workbench. In *Open PSA Workshop 2012*. EDF, 2012.

[81] T. Friedlhuber, M. Hibti, and A. Rauzy. Overview of the open psa platform. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, June 2012.

[82] I Renault, M Pilliere, N Villatte, and P Mouttapa. Kb3: Computer program for automatic generation of fault trees. In *Reliability and Maintainability Symposium, 1999. Proceedings. Annual*, pages 389–395. IEEE, 1999.

[83] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse rich client platform*. Addison-Wesley Professional, 2010.

[84] Wikipedia. Eclipse rcp. http://wiki.eclipse.org/index.php/Rich_Client_Platform.

[85] Paul Dietel. *Java how to program*. PHI, 2009.

[86] Cay S Horstmann and Gary Cornell. *Core Java 2: Volume I, Fundamentals*. Pearson Education, 2002.

[87] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[88] OSGi Alliance. Open services gateway initiative, 2009.

[89] Richard Hall, Karl Pauls, Stuart McCulloch, and David Savage. *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[90] OSGi Alliance. *Osgi service platform, release 3*. IOS Press, Inc., 2003.

[91] David Eck. Generic programming. 2013.

[92] T. Friedlhuber. An integrated approach of model based safety engineering. In *The 3rd International Workshop on Model Based Safety Assessment*, March 2013.

[93] Eclipse Foundation. Eclipse. http://www.eclipse.org.