The Open-PSA Initiative

Open-PSA Model Exchange Format

2.0.d-120-g703be91

October 4, 2017

Title	Open-PSA Model Exchange Format
Version	2.0.d-120-g703be91
Creation date	2007-08-01
Last modification date	2017-05-01
Editors	Epstein Steven, Rauzy Antoine

Contributors

This document presents the "Open-PSA Model Exchange Format". The redaction of this representation format is a shared effort. The following persons contributed to various extents to the current version of the document: Becker Guenter, Čepin Marko, Contini Sergio, Ducamp François, Epstein Steven, Herrero Santos Roberto, Hibti Mohamed, Kampramanis Ioanis, Klügel Jens, Meléndez Asensio Enrique, Perez Mulas Arturo, Nusbaumer Olivier, Quatrain Richard, Rauzy Antoine, Rauzy Pablo, Reinhart Mark, and Sörman Johan.

CONTENTS

1 The Open-PSA Initiative **1**

2 Introduction **3**
2.1 Why Do We Need a Model Exchange Format? . . . 3
2.2 Requirements for the Model Exchange Format . . . 5
2.3 Choice of XML . . . 5
2.4 A Four-Plus-One Layer Architecture . . . 6
2.5 Formalism . . . 7
2.6 Organization of the document . . . 8

3 Anatomy of the Model Exchange Format **9**
3.1 Elements of a Model . . . 9
3.1.1 Variables, Terms and Containers . . . 9
3.1.2 Stochastic Layer . . . 10
3.1.3 Fault Tree Layer . . . 10
3.1.4 Meta-Logical Layer . . . 10
3.1.5 Event Tree Layer . . . 11
3.2 Structure of a Model . . . 12
3.2.1 Relationships between elements of a model . . . 12
3.2.2 Giving more structure to a model . . . 13
3.2.3 Containers as name spaces . . . 13
3.2.4 Definitions, Labels, and Attributes . . . 13

4 Fault Tree Layer **16**
4.1 Description . . . 16
4.2 XML Representation . . . 18
4.3 Extra Logical Constructs and Recommendations . . . 22
4.3.1 Model-Data and Components . . . 22
4.3.2 Solving Name Conflicts: Public versus Private Elements . . . 24
4.3.3 Inherited attributes . . . 25

4.3.4 Recommendations . . . 25

5 Stochastic Layer **27**
5.1 Description . . . 27
5.2 Operations . . . 30
5.2.1 Numerical Operation . . . 30
5.2.2 Boolean Operations . . . 32
5.2.3 Conditional Operations . . . 32
5.3 Built-Ins . . . 34
5.3.1 Description . . . 34
5.3.2 XML Representation . . . 36
5.4 Primitive to Generate Random Deviates . . . 37
5.4.1 Description . . . 37
5.4.2 XML Representation . . . 40
5.5 Directives to Test the Status of Initiating and Functional Events . . . 42
5.5.1 Description . . . 42
5.5.2 XML Representation . . . 42

6 Meta-Logical Layer **43**
6.1 Common Cause Groups . . . 43
6.1.1 Description . . . 43
6.1.2 XML representation . . . 45
6.2 Delete Terms, Recovery Rules, and Exchange Events . . . 46
6.2.1 Description . . . 46
6.2.2 All Extra-Logical Constructs in One: the Notion of Substitution 47
6.2.3 XML Representation . . . 48

7 Event Tree Layer **51**
7.1 Preliminary Discussion . . . 51
7.2 Structure of Event Trees . . . 53
7.2.1 Description . . . 53
7.2.2 XML Representation . . . 55
7.3 Instructions . . . 57
7.3.1 Description . . . 57
7.3.2 XML Representation . . . 58

8 Organization of a Model **62**
8.1 Additional Constructs . . . 62
8.1.1 Consequences and Consequence Groups . . . 62
8.1.2 Missions, Phases . . . 63
8.2 Splitting the Model into Several Files . . . 63
8.3 Organization of a Model . . . 64

9 Report Layer **66**
9.1 Preliminary Discussion . . . 66
9.2 Information about calculations . . . 67
9.3 Format of Results . . . 68
9.3.1 Minimal Cut Sets . . . 68
9.3.2 Statistical measures . . . 68

9.3.3 Curves 70

10 Bibliography **71**
10.1 Basic PSA References 71
10.2 Difficulties with PSA 71
10.3 Novel Approaches 72

LIST OF TABLES

4.1 Semantics of Boolean connectives 18

5.1 Numerical Operations, their number of arguments, and their semantics . 30
5.2 Boolean operators, their number of arguments, and their semantics . . . 32
5.3 Built-ins, their number of arguments, and their semantics 36
5.4 Primitive to generate random deviates, their number of arguments, and their semantics 38
5.5 Directives to test the status of initiating and functional events 42

7.1 States of Functional Events for the different paths of the Event Tree in Fig. 7.2 54

LIST OF FIGURES

2.1 Architecture of the Model Exchange Format 6

3.1 The main elements of a model, their layers, and their dependencies . . . 12

4.1 A Fault Tree 17
4.2 A Fault Tree with Three Components 23

5.1 Meaning of parameters τ, θ, and π of the "periodic-test" built-in 35
5.2 Multi-phase Markov graph for the "periodic-test" built-in 35

7.1 A Small Event Tree 51
7.2 Structure of an Event Tree 53

8.1 Containers and the constructs they can define 65

CHAPTER

1

THE OPEN-PSA INITIATIVE

The Open Initiative for Next Generation of Probabilistic Safety Assessment

As we enter a time in which safety and reliability have come to the attention of the public, especially in the face of climate change and a nuclear renaissance, efforts are being made in the direction of the "next generation" of Probabilistic Safety Assessment with regards to software and methods. These new initiatives hope to present a more informative view of the actual models of systems, components, and their interactions, which helps decision makers to go a step forward with their decisions.

The Open Initiative for Next Generation PSA provides an open and transparent public forum to disseminate information, independently review new ideas, and spread the word. We want to emphasize openness which leads to methods and software with better quality, better understanding, more flexibility, encourage peer review, and allow the transportability of models and methods.

We hope to bring to the international PSA community the benefits of an open initiative, and to bring together the different groups who engage in large scale PSA, in a non-competitive and commonly shared organization.

Two of our most important activities will be as a standards body and clearing house for methodologies for the good of PSA. In this way, researchers, practitioners, corporations, and regulators can work together in open cooperation.

Over the last 5 years, some non-classical calculation techniques and modeling methods in nuclear PSA have been extensively studied. The concern of these investigations has been to end the use of (1) numerical approximations for which we do not know the error factors, (2) modeling methods which leave out perhaps critical elements of the actual plant, and (3) lack of good man-machine and organizational modeling techniques. From all these investigations, some alarming issues related to large, safety critical PSA models have been raised, which we feel need to be addressed before new calculation engines or next generation user interfaces are put into place:

- Quality assurance of calculations
- Unfounded reliance on numerical approximations and truncation
- Portability of the models between different software
- Clarity of the models
- Completeness of the models
- Modeling of human actions
- Better visualization of PSA results
- Difficulty of different software working with the same PSA model
- Lack of data and software backward and forward compatibility
- No universal format for industry data

New calculation engines and user interfaces and a computer representation for large, safety critical PSA models, which is independent of PSA software, represent a step forward in addressing the above issues.

As our first activity, we have created a working group to begin the creation of a model exchange format for PSA models. Other working groups in the other aforementioned areas are expected to follow the success of the first one.

We believe that each of you, who are reading this manifesto, have similar ideas. Let us enter into an open forum together and work together to know the limits of our methods, to push those limits, and to expand our understanding.

CHAPTER

2

INTRODUCTION

2.1 Why Do We Need a Model Exchange Format?

Over the years, research efforts have been made in the direction of "next generation" PSA software and "declarative modeling", which try to present a more informative view of the actual systems, components, and interactions which the model represents. The concern of these studies has been to end the use of approximations: numerical approximations for which we do not know the error factors, and modeling approximations which leave out perhaps critical elements of the actual system under study. From all these investigations, some issues related to large nuclear PSA models have been raised, which need to be addressed before putting new calculation engines or next generation user interfaces into place. To address these issues enumerated below, a "Model Exchange Format", a representation which is independent of all PSA software, must be in place. In this perspective, software would retain their own internal representation for a model; moreover, each software would also be able to share models and industry data by means of the Model Exchange Format.

Quality assurance of calculations At the moment, a model built with one software, cannot be simply quantified with another software, and visa versa; there are too many software-dependent features used by modelers to make inter-calculation comparisons a one-step process. The Model Exchange Format will allow models to be quantified by several calculation engines, resulting in stronger quality assurance.

Over reliance on numerical approximations and truncation While this cannot be solved directly by the Model Exchange Format, as new calculation engines are completed, the Model Exchange Format will allow new engines to be snapped into new (or existing) user interfaces without changing the model or user interface software.

Portability of the models between different software At the moment, models are essentially non-portable between calculation engines, as pointed out above. The Model Exchange Format allows complete, whole models to be shared right now between software; the bonus will be on each software to correctly interpret the model representation.

Clarity of the models For one who examined a number of large nuclear PRA models, it is obvious that just looking at the basic events, gates, and fault trees/event trees is of little help in understanding the "where", "why", and "how" of model elements: common cause failures, initiating events, sequence information, alignment information, systems and trains, flags, logic of recovery rules, or the dreaded "delete terms". The Model Exchange Format employs what is becoming known as structured modeling. Structured Modeling takes its name from the structured programming movement in the 1970s. Before that time, variables, arrays, and other data structures, were used with no definitions or explanations. Structured programming techniques forced programmers to "declare variables" at the beginning of a program by name and also by the type of variable it was: an integer, a real number, and so on. In this way the meaning of the program became clearer, and calculation speeds were increased. Structured Modeling, as applied to PRA models and software, has the same goal of making the meaning of the model more clear, more transparent, and to improve the speed and accuracy of the calculation. The user interface to create such a model is not of concern here. The concern is to discover the distinct model elements which are needed to quantify and clarify large PRA models.

Completeness of the models Without clarity, there can be no knowledge of the completeness of the model, since their very size and complexity strains the brain. The Model Exchange Format will create more survey-able models.

Difficulty of different software working with the same PSA model As more risk applications are being requested (seismic, fire, balance of plant assessments, risk monitors, release calculations), difficulties are arising because each risk application and major PSA software have different internal data formats. The Model Exchange Format will enable easy sharing of model data between applications, and specialized software would be available for all models.

Lack of data and software backward and forward compatibility Again, as more diverse software need to interact, such as safety monitors, calculation engines, and fault tree editors, the need to have data and programs separate becomes of high importance. The Model Exchange Format solves this problem by allowing programs to change without the need for the data format to change and for other programs to change their operations.

No universal format for industry data The Model Exchange Format will be a perfect way to publish industry data, such as common cause, failure rates, incidents, and initiating event frequencies.

2.2 Requirements for the Model Exchange Format

To be acceptable and widely accepted, the Model Exchange Format for PSA must fulfill a number of requirements. The following list is an attempt to summarize these requirements.

Soundness The Model Exchange Format must be unambiguous. The semantics of each construct must be clearly given in such way that no two correct implementations of the Model Exchange Format can differ in their interpretation of models (however, they may differ at least to a certain extent, in the results they provide if they use different calculation methods).

Completeness The Model Exchange Format should cover as much as possible; not only all aspects of PSA models, but also references to external documentations and format of the results. These issues have to be covered by the Model Exchange Format in order to make models actually portable and to be able to cross check calculations.

Clarity The Model Exchange Format should be self-documenting to a large extent. The constructs of the Model Exchange Format should reflect what the designer of the model has in mind. Low level constructs would help in making the format universal (any model can eventually be represented by means of a FORTRAN or C program, not to speak of a Turing machine or a Church lambda term), but constructs which are at too low a level would be of little help, and even counter-productive, for model review.

Generality It should be possible to cast all the existing models into the Model Exchange Format without rewriting them from scratch. The translation of existing models should be automated, at least to a large extent. Moreover, any existing tool should be able to use the Model Exchange Format as its representation language. Indeed, most of the tools implement only a subpart of the Model Exchange Format, but the Model Exchange Format should be a superset of the underlying formalisms of all existing tools.

Extensibility The Model Exchange Format should not restrict developers if they wish to introduce interesting new features in their tools. This means that it should be easy to introduce new constructs into the Model Exchange Format, even if these constructs are not recognized by all tools. On the other hand, these new constructs should be clearly identified; their semantics should be clear and public in such way that any other developer can embed the feature in his own tool.

2.3 Choice of XML

To create the Model Exchange Format, we must make formal definitions for representing existing PRA models and define a syntax to write them. The Model Exchange Format is defined as an XML document type. XML is widely used on the Internet as a common way for programs to share data. It is well structured and makes it possible to give explicit name to each construct. XML, therefore, is well suited for structured modeling. By

giving the elements of a model a formal designation ("this is an initiating event", "this is a basic event", and so on), quantification results and understanding of the model can be improved.

XML presents another major advantage for tool developers: many development teams have more or less already designed its own XML parser and many such parsers are anyway freely available on the Internet. Therefore, the choice of an XML based syntax discharges programmers of PSA tools of the tedious task to design parsers and to perform syntactic checks. Moreover, due to their tree-like structure, it is easy to ignore parts of an XML description that are not relevant for a particular purpose. Therefore, tools which do not implement the whole Model Exchange Format can easily pick up what they are able to deal with.

2.4 A Four-Plus-One Layer Architecture

The Model Exchange Format relies on a four-plus-one layer architecture, as pictured in Fig. 2.1. Each layer corresponds to a specific class of objects/mathematical constructs.

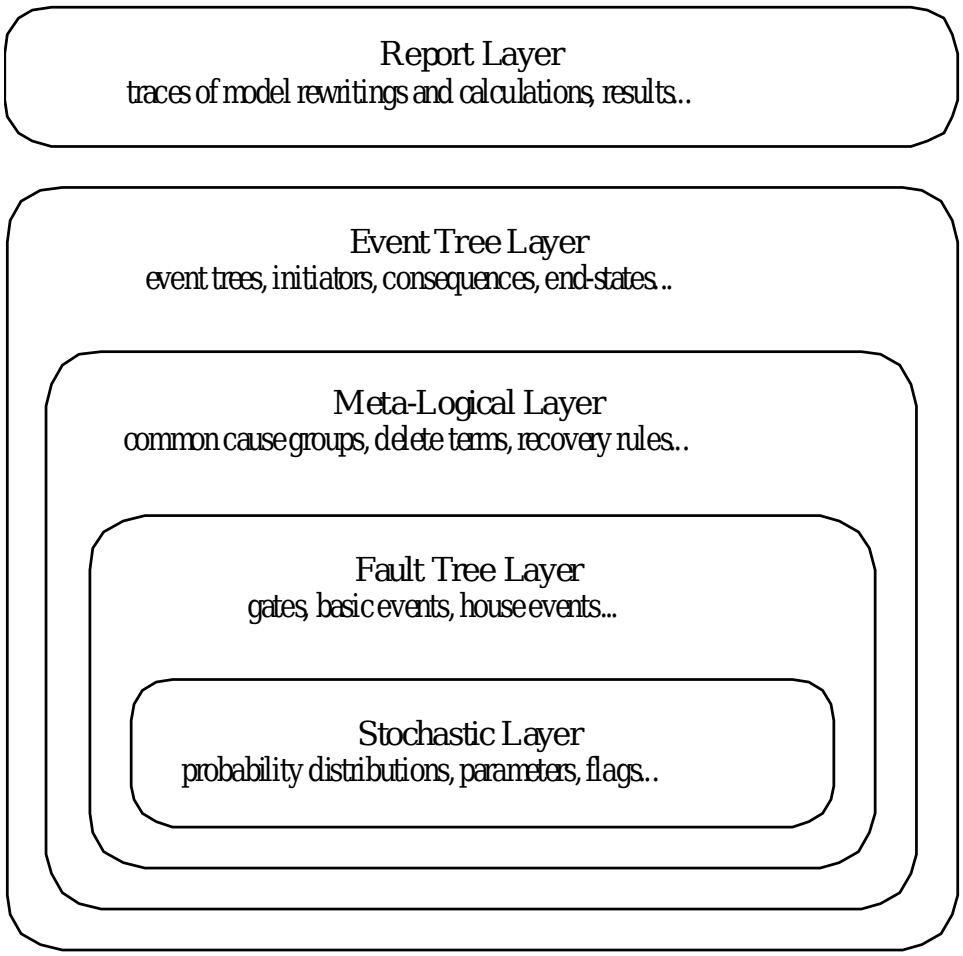


Fig. 2.1: Architecture of the Model Exchange Format

- The first, or stochastic, layer is populated with all stochastic aspects of models:

probability distributions for the failure rates of basic events, parameters of these distributions and distributions of these parameters, flags, etc.

- The second, or fault tree layer, is populated with logical components of fault trees (gates, basic events, house events). This layer is the core of PSA models. The two first layers can be used in isolation. Some existing tools implement them only.
- The third, or meta-logical, layer is populated with constructs like common cause groups, delete terms, and recovery rules that are used to give flavors to fault trees.
- The fourth, or event tree, layer is populated with event trees, initiating events, and consequences. The Model Exchange Format sees event trees as (graphical) programs. The execution of such a program produces a set of sequences, i.e., a set (a disjunction) of Boolean formulae. Probability distributions are also collected while walking the event tree.
- The fifth, or report layer, is populated with constructs to store results of calculations. This includes constructs to describe calculations (version of the model, used engine, used cutoffs, targeted group of consequences, calculated quantities, etc.) as well as minimal cut sets and other results.

This five-layer architecture helps to understand what the different elements of a model are and what their respective roles are. In a word, it is the backbone of the Model Exchange Format. However, it should be clear that any model will contain elements of the first fourth levels and that these elements may not be arranged by levels. For instance, a fault tree description will probably contain probability distributions of basic events as well as common cause groups. Again, the five-layer architecture intends to differentiate elements according to their meanings and operational behaviors.

2.5 Formalism

Throughout this document, we shall present a number of syntactic constructions, such as Boolean formulae, probability distributions, and so on. These constructions will eventually be represented by means of XML terms. However, XML is a bit too verbose to make clear the underlying mathematical nature of objects at hand. Therefore, we shall use an XML schema language to define constructs.

There are several formal ways to describe an XML grammar. The most popular approach is to use one of XML schema languages, such as the XML Document Type Definition (DTD), XML Schema Definition (XSD), RELAX NG (REgular LAnguage for XML Next Generation), Schematron. The Model Exchange Format used to use the DTD for its formal schema; however, mainly due to the DTD's lack of maintainability, age, and limitations (a lack of context awareness), RELAX NG Compact (RNC) has been chosen as a modern replacement. The RNC has a non-XML syntax and leverages regular expression operators (similar to the DTD and Extended Backus Naur Form) and is structured in a very concise and human-readable form, and unlike the DTD, the RNC is feature-rich enough to support the MEF grammar. Even though the RNC language is very self-descriptive, please consult with RELAX NG Compact Tutorial and the specification to

gain familiarity. In addition to the schema, we shall present the grammar of the Model Exchange Format by means of examples.

The RNC schemas describing the Model Exchange Format can be combined into the main schema for validation purposes. The MEF XML schemas in various other formats are provided at the MEF schemas repository. These schemas are auto-generated from the RNC schema in this specification.

It is worth noting that the XML descriptions we are giving here can be extended in any way to fulfill the needs of a particular tool. In particular, comments and pointers to documentation should be added here and there to the model.

2.6 Organization of the document

The remainder of this document is organized into six chapters, corresponding to the introductory overview, one chapter per layer of the architecture of the Model Exchange Format and one additional chapter for models as a whole.

- Chapter 3 gives an overview of main elements of a model and shows how these elements are organized. It discusses how to split a description into several files, how to solve naming conflicts, etc.
- Chapter 4 presents the fault tree layer. The fault tree layer is not the lowest one in the hierarchy. However, fault trees are the most basic and the central concept of PSA models. For this reason, we put it in front.
- Chapter 5 presents the stochastic layer, i.e., all the mechanisms to associate probability distributions to basic events.
- Chapter 6 presents the meta-logical layer.
- Chapter 7 presents the event tree layer.
- Chapter 8 discusses the organization of models.
- Finally, Chapter 9 presents the report/results layer, i.e., the normalized format for results of assessment of PSA models.

CHAPTER

3

ANATOMY OF THE MODEL EXCHANGE FORMAT

This chapter presents the anatomy of the Model Exchange Format, i.e., the main components of a model and their relationships. We assume the reader is familiar with the fault tree/event tree methodology.

3.1 Elements of a Model

3.1.1 Variables, Terms and Containers

Elements of a model are, as expected, components of fault trees/event trees, namely basic events, gates, house events, probability distributions, initiating events, safety systems, consequences, etc. Conceptually, it is convenient to arrange most of these elements into one of the three categories: terms, variables, and containers.

Variables Variables are named elements. Gates, basic events, house events, stochastic parameters, functional events, initiating events, and consequences are all variables. A variable is always defined, i.e., associated with a term.

Terms Terms are built over variables, constants, and operators. For instance, the Boolean formula "primary-motor-failure or no-current-to-motor" is a term built over the basic event "primary-motor-failure", the gate "no-current-to-motor", and the Boolean operator "or". Similarly, the probability distribution "1-exp(-lambda*t)" is a term built over the numerical constant "1", the failure rate "lambda", the time "t", and the three arithmetic operators "-", "exp", and "*" ("lambda" and "t" are variables). Note that variables are terms.

Containers According to our terminology, a model is nothing but a set of definitions of variables. Since a brute list of such definitions would lack of structure, the Model Exchange Format makes it possible to group them into containers. Containers have names and can be themselves grouped into higher level containers. For instance, a fault tree is a container for definitions of gates, house-events, basic events, and parameters of probability distributions. Similarly, an event tree is a container for definitions of initiating events, functional events, sequences, etc.

We are now ready to list the main elements of a model. The exact content and role of these different elements will be detailed in the subsequent chapters.

3.1.2 Stochastic Layer

Stochastic variables and terms Stochastic expressions are terms that are used to define probability distributions (associated with basic events). Stochastic variables are called parameters. For instance, "1-exp(-lambda*t)" is a stochastic expression built over the two parameters "lambda" and "t".

From a programming viewpoint, it is convenient to group definitions of parameters into (stochastic) containers. The stochastic layer is populated with stochastic parameters, expressions, and containers.

3.1.3 Fault Tree Layer

Boolean formulae, Basic Events, House Events, and Gates Boolean formulae, or formulae for short, are terms built over the usual set of constants (true, false), connectives (and, or, not, etc.), and Boolean variables, i.e., Basic Events, Gates, and House Events. Boolean variables are called events, for that is what they represent in the sense of the probability theory. Basic events are associated with probability distributions, i.e., with (stochastic) expressions. Gates are defined as Boolean formulae. House events are special gates that are defined as Boolean constants only.

Fault Trees According to what precedes, a fault tree is container for definitions of parameters, basic events, house events, and gates.

The fault tree layer is populated with all elements we have seen so far.

3.1.4 Meta-Logical Layer

The meta-logical layer contains extra-logical constructs in addition to fault trees. These extra-logical constructs are used to handle issues that are not easy to handle in a purely declarative and logical way.

Common Cause Groups Common cause groups are sets of basic events that are not statistically independent. Several models can be used to interpret common cause groups. All these models consist in splitting each event of the group into a disjunction of independent basic events.

Substitutions Delete terms, exchange events, and recovery rules are global and extra-logical constraints that are used to describe situations such as physical impossibilities, technical specifications, or to modify the probability of a scenario according to some physical rules or judgments about human actions. In the Model Exchange Format, these extra-logical constructs are all modeled by means of the generic notion of substitution.

3.1.5 Event Tree Layer

As we shall see, event trees must be seen as a (semi-)graphical language to describe and to combine sequences. Elements of this language are the following.

Event Trees Event Trees define scenarios from an Initiating Event (or an Initiating Event Group) to different end-states. In the Model Exchange Format, end-states are called Sequences. The same event tree can be used for different Initiating Events. Along the scenarios, "flavored" copies of fault trees are collected and/or values are computed. Flavors are obtained by changing values of house events and parameters while walking along the tree. Event Trees are containers according to our terminology. They contain definition of functional events and states.

Initiating Events, Initiating Event Groups Initiating Events describe the starting point of an accidental sequence. They are always associated with an event tree, although they are in general declared outside of this event tree. The Model Exchange Format makes it possible to chain event trees. Therefore, the end-state of a sequence of an event tree may be the initiating event of another event tree. Initiating Events are variables, according to our terminology. Initiating event groups are sets of initiating events. Despite of their set nature, initiative events are also variables because an initiating event group may contain another one (the initiating terms are set operations).

Functional Events Functional Events describe actions that are taken to prevent an accident or to mitigate its consequences (usually by means of a fault tree). Depending on the result of such an action, the functional event may be in different, e.g., "success" or "failure". Functional Events label the columns the graphical representation of Event Trees.

Sequences, Branches Sequences are end-states of branches of event trees. Branches are named intermediate states.

Instructions, Rules Instructions are used to describe the different paths of an event tree, to set the states of functional events, to give flavors of fault trees that are collected, and to communicate with the calculation engine. Rules are (named) groups of Instructions. They generalize split-fractions of the event tree linking approach, and boundary condition sets of the fault tree linking approach.

Consequences, Consequence groups Consequences are couples made of an initiating event and a sequence (an event tree end-state). Consequences are named and defined. They are variables according to our terminology. Like Initiating Events, Consequences can be grouped to study a particular type of accident. Consequence Groups are also variables (the consequence terms are set operations).

Missions, Phases In some cases, the mission of the system is split into different phase. The Model Exchange Format provides constructs to reflect this situation.

3.2 Structure of a Model

3.2.1 Relationships between elements of a model

The elements of a model, their layer, and their dependencies are pictured in Fig. 3.1. This schema illustrates the description given in the previous section. Term categories are represented by rectangles. Variables categories are represented by rounded rectangles. A variable category is always included in a term category (for variables are terms). The three container categories, namely models, event trees, and fault trees, are represented by dashed rectangles. Dependencies among categories are represented by arrows.

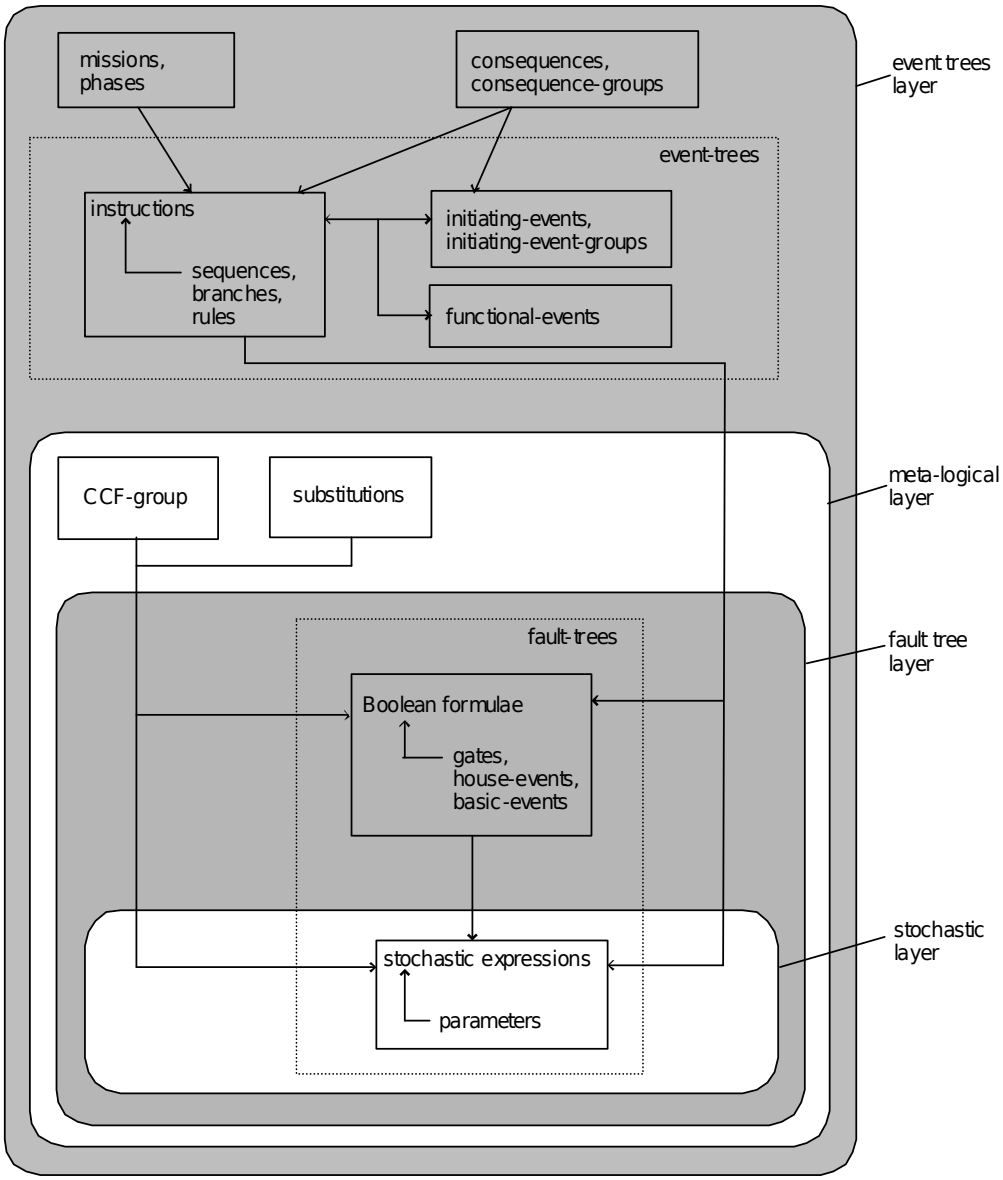


Fig. 3.1: The main elements of a model, their layers, and their dependencies

3.2.2 Giving more structure to a model

A model (like a fault tree or an event tree) is a list of declarations. The Model Exchange Format does not require structuring these declarations: they can be given in any order, provided that the type of an object can be decided prior to any use of this object. Fault trees and event trees provide a first mean to organize models. This may be not sufficient, especially when models are big. In order to structure models, the Model Exchange Format provides the analyst with two mechanisms.

First, declarations can be grouped together by means of user defined containers. Such a container is just a XML tag. It has no semantics for the model. It just makes it possible to delimit a set of objects of the model that are physically or functionally related (for instance, the different failure modes of a physical component).

Second, the Model Exchange Format makes it possible to associate user defined attributes to the main components. For instance, we may define an attribute "zone" with a value "room33" for all constructs describing components located in the room 33. This indirect mean is very powerful. It can be used extensively to perform calculations or changes on a particular subset of elements.

3.2.3 Containers as name spaces

Once declared, elements are visible and accessible everywhere in the model. This visibility means in turn that an object of a given type, e.g., parameter or event, is unique. No two distinct objects of the same type can have the same name. This constraint seems to be fine and coherent. However, some tools do not obey the rule: two gates of two different fault trees and representing two different functions may have the same name. It is not possible to reject this possibility (as a bad modeling practice), because when models are large and several persons are working in collaboration, such name conflicts are virtually impossible to avoid.

To solve this problem, the Model Exchange Format considers containers, i.e., not only fault trees and event trees but also user defined containers, as name spaces. By default, objects defined in a container are global, but it is possible to declare them as local to the container as well. In that case, they are not visible outside the container, and tools are in charge of solving potential name conflicts.

3.2.4 Definitions, Labels, and Attributes

Here follows some additional useful elements about the Model Exchange Format.

Definitions versus references For the sake of the clarity (and for XML specific reasons), it is important to distinguish the declaration/definition of an element from references to that element. For instance, we have to distinguish the definition of the gate "motor-fails-to-start" (as the Boolean formula "primary-motor-failure or no-current-to-motor"), from references to that gate into definitions of other gates.

In the Model Exchange Format, the definition of a variable or a container, for instance a gate, is in the following form.

```
<define-gate name="motor-fails-to-start">
    ...
</define-gate>
```

References to that gate are in the following form.

```
...
<gate name="motor-fails-to-start"/>
...
```

So, there are two tags for each element (variable or container) of the Model Exchange Format: the tag "define-element" to define this element and the tag "element" to refer this element. Note that the attribute "name" is systematically used to name elements.

Labels It is often convenient to add a comment to the definition of an object. The Model Exchange Format defines a special tag "label" to do so. The tag label can contain any text. It must be inserted as the first child of the definition of the object.

```
<define-gate name="motor-fails-to-start">
    <label>Warning: secondary motor failures are not taken into account␣
↪here.</label>
    ...
</define-gate>
```

Attributes Attributes can be associated with each element (variable or container) of the Model Exchange Format. An attribute is a pair (name, value), where both name and value are normally short strings. Values are usually scalars, i.e., they are not interpreted. In order to allow tools to interpret values, a third field "type" can be optionally added to attributes. The tags "attributes" and "attribute" are used to set attributes. The former is mandatory, even when only one attribute is defined. It must be inserted as the first child of the definition of the object, or just after the tag label, if any.

```
<define-gate name="motor-fails-to-start">
    <label>Warning: secondary motor failures are not taken into account␣
↪here.</label>
    <attributes>
        <attribute name="zone" value="room33" />
        ...
    </attributes>
    ...
</define-gate>
```

The RNC schema for the XML representation of labels and attributes is as follows.

```
name = attribute name { Identifier }

label = element label { text }
```

```
attributes =
  element attributes {
    element attribute {
      name,
      attribute value { xsd:string },
      attribute type { xsd:string }?
    }+
  }
```

CHAPTER

4

FAULT TREE LAYER

The Fault Tree layer is populated with logical components of Fault Trees. It includes the stochastic layer, which contains itself the probabilistic data. The stochastic layer will be presented in the next section.

4.1 Description

Constituents of fault trees are Boolean variables (gates, basic events, and house events), Boolean constants (true and false) and connectives (and, or, k-out-of-n, not, etc.). Despite of their name, fault trees have in general a directed acyclic graph structure (and not a tree-like structure) because variables can be referenced more than once. The simplest way to describe a fault tree is to represent it as a set of equations in the form "variable = Boolean-formula". Variables that show up as left hand side of an equation are gates. Variables that show up only in right hand side formulae are basic events. Finally, variables that show up only as left hand side of an equation are top events. Such a representation imposes two additional conditions: first, the set of equations must contain no loop, i.e., that the Boolean formula at the right hand side of an equation must not depend, even indirectly (recursively), on the variable at the left hand side. Second, a variable must not show up more than once at the left hand side of an equation, i.e., gates must be uniquely defined. Fig. 4.1 shows a Fault Tree. The corresponding set of equations is as follows.

$$
\begin{aligned}
TOP &= G1 \vee G2 \\
G1 &= H1 \wedge G3 \wedge G4 \\
G2 &= \neg H1 \wedge BE2 \wedge G4 \\
G3 &= BE1 \vee BE3 \\
G4 &= BE3 \vee BE4
\end{aligned}
$$

On the figure, basic events are surrounded with a circle. Basic events are in general associated with a probability distribution (see Chapter 5).

House events (surrounded by a house shape frame on the figure) are represented as variables but are actually constants: when the tree is evaluated house events are always interpreted by their value, which is either true or false. By default, house events take the value false. Negated house events (gates, basic events) are represented by adding a small circle over their symbol.

A formal description of constructs of Fault Trees is given under the RNC schema in Listing 4.1. This description allows loops (in the sense defined above), multiple definitions and trees with multiple top events. The presence of loops must be detected by a specific check procedure. If a variable or a parameter is declared more than once, tools should emit a warning and consider only the last definition as the good one (the previous ones are just ignored). In some circumstances, it is of interest to define several fault trees at once by means of a unique set of declarations. Therefore, the presence of multiple top events should not be prevented. We shall see what parameters and expressions are in the next chapter.

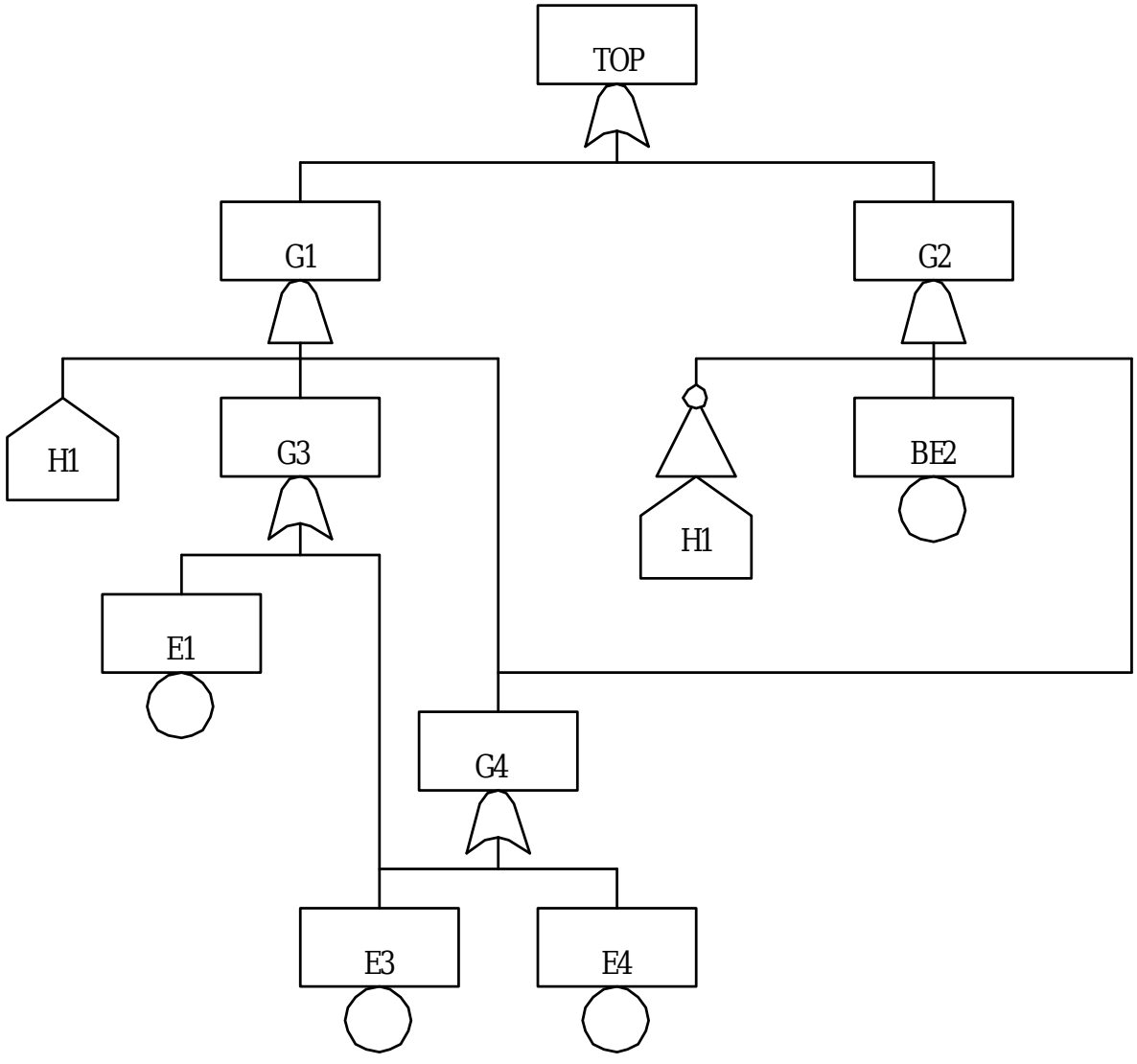


Fig. 4.1: A Fault Tree

The semantics of connectives is given in Table 4.1. Note that connectives "and", "or", "xor", "iff", "nand", and "nor" are associative. Therefore, it suffices to give their semantics when they take two arguments, i.e., two Boolean formulae F and G.

Table 4.1: Semantics of Boolean connectives

Connective	Semantics
and	F and G is true if both F and G are true, and false otherwise
or	F or G is true if either F or G is true, and false otherwise
not	not F is true if its F is false, and false otherwise
xor	F xor G is equivalent to (F and not G) or (not F and G)
iff	F iff G is equivalent to (F and G) or (not F and not G)
nand	F nand G is equivalent to not (F and G)
nor	F nor G is equivalent to not (F or G)
atleast	true if at least **k** out of the Boolean formulae given as arguments are true, and false otherwise. This connective is also called *k-out-of-n*, where **k** is the integer and **n** is the Boolean formulae given in arguments
cardinality	true if at least **l** and at most **h** of the Boolean formulae given as arguments are true, and false otherwise. **l** and **h** are the two integers (in order) given as arguments.
imply	F implies G is equivalent to (not F or G)

Dynamic Gates
In a second step, it would be of interest to incorporate to the Model Exchange Format "inhibit" gates, "priority" gates, and "triggers" (like in Boolean Driven Markov processes). All of these dynamic gates can be interpreted as "and" gates in a Boolean framework. In more general frameworks (like Markovian frameworks), they can have different interpretations, and provide mechanisms to accurately model backup systems, limited amount of resources, etc. The complexity of the assessment of this kind of model is indeed much higher than the one of Boolean models (which is already at least NP-hard or #P-hard).

4.2 XML Representation

The RNC schema for the XML description of fault trees is given in Listing 4.1 and Listing 4.2.

This description deserves some comments.

- It leaves for now the tags "define-parameter" and "expression" unspecified. We shall see in the next chapter how these tags are used to define the probability distributions.
- Similarly, the tag "define-component" will be explained in the next section.
- Although the Model Exchange Format adopts the declarative modeling paradigm, it is often convenient to use variables in formulae before declaring them. The

Model Exchange Format, therefore, refers to variables with the generic term "event", possibly without a "type" attribute.

- By default, the value of a house event is "false", so it is not necessary to associate a value with a house event when declaring it. We shall see in Section 7.3 how to change the value of a house event.
- Although events are typed (they are either gates, house events or basic events), two different events cannot have the same name (within the same name space), even if they are of different types. This point will be explained in the next section.
- A "pass-through" gate can be defined without a connective but with a single argument event for its formula.

Listing 4.1: The RNC schema of XML description of Fault Trees

```
fault-tree-definition =
  element define-fault-tree {
    name,
    label?,
    attributes?,
    (substitution-definition
     | CCF-group-definition
     | event-definition
     | component-definition
     | parameter-definition
     | include-directive)*
  }

component-definition =
  element define-component {
    name,
    role?,
    label?,
    attributes?,
    (substitution-definition
     | CCF-group-definition
     | event-definition
     | component-definition
     | parameter-definition
     | include-directive)*
  }

role = attribute role { "private" | "public" }

model-data =
  element model-data {
    (house-event-definition
     | basic-event-definition
     | parameter-definition
     | include-directive)*
  }
```

```
event-definition =
  gate-definition | house-event-definition | basic-event-definition

gate-definition =
  element define-gate { name, role?, label?, attributes?, formula }

house-event-definition =
  element define-house-event {
    name, role?, label?, attributes?, Boolean-constant?
  }

basic-event-definition =
  element define-basic-event {
    name, role?, label?, attributes?, expression?
  }
```

Listing 4.2: The RNC schema of the XML representation of Boolean formulae

```
formula =
  event
  | Boolean-constant
  | element and { formula+ }
  | element or { formula+ }
  | element not { formula }
  | element xor { formula+ }
  | element iff { formula+ }
  | element nand { formula+ }
  | element nor { formula+ }
  | element atleast {
      attribute min { xsd:positiveInteger },
      formula+
    }
  | element cardinality {
      attribute min { xsd:nonNegativeInteger },
      attribute max { xsd:nonNegativeInteger },
      formula+
    }
  | element imply { formula, formula }

event =
  element event {
    reference,
    attribute type { event-type }?
  }
  | gate
  | house-event
  | basic-event

event-type = "gate" | "basic-event" | "house-event"
```

```
gate = element gate { reference }

house-event = element house-event { reference }

basic-event = element basic-event { reference }

Boolean-constant =
  element constant {
    attribute value { Boolean-value }
  }

Boolean-value = "true" | "false"
```

The attribute "role" is used to declare whether an element is public or private, i.e., whether it can be referred by its name everywhere in the model or only within its inner most container. This point will be further explained in the next section. This attribute is optional, for by default, all elements are public.

The fault tree pictured in Fig. 4.1 is described in Listing 4.3. In this representation, the house event "h1" has by default the value "true". Basic events are not declared, for it is not necessary, so no probability distributions are associated with basic events.

Listing 4.3: XML description of Fault Tree pictured in Fig. 4.1

```
<?xml version="1.0" ?>
<!DOCTYPE opsa-mef>
<opsa-mef>
    <define-fault-tree name="FT1">
        <define-gate name="top">
            <or>
                <gate name="g1"/>
                <gate name="g2"/>
            </or>
        </define-gate>
        <define-gate name="g1">
            <and>
                <house-event name="h1"/>
                <gate name="g3"/>
                <gate name="g4"/>
            </and>
        </define-gate>
        <define-gate name="g2">
            <and>
                <not>
                    <house-event name="h1"/>
                </not>
                <basic-event name="e2"/>
                <gate name="g4"/>
            </and>
        </define-gate>
```

```
        <define-gate name="g3">
            <or>
                <basic-event name="e1"/>
                <basic-event name="e3"/>
            </or>
        </define-gate>
        <define-gate name="g4">
            <or>
                <basic-event name="e3"/>
                <basic-event name="e4"/>
            </or>
        </define-gate>
        <define-house-event name="h1">
            <constant value="true"/>
        </define-house-event>
    </define-fault-tree>
</opsa-mef>
```

4.3 Extra Logical Constructs and Recommendations

4.3.1 Model-Data and Components

The Model Exchange Format provides a number of extra-logical constructs to document and structure models. Labels and attributes are introduced in Section 3.2.4. They can be associated with a declared element in order to document this element. Fault trees are a first mean to structure models. A fault tree groups any number of declarations of gates, house events, basic event, and parameters.

It is sometimes convenient to group definitions of house events, basic events, and parameters outside fault trees. The Model Exchange Format provides the container "model-data" to do so.

The Model Exchange Format makes it possible to group further declarations through the notion of component. A component is just a container for declarations of events and parameters. It has a name and may contain other components. The use of components is illustrated by the following example.

Fig. 4.2 shows a fault tree FT with three components A, B, and C. The component B is nested into the component A. The XML representation for this Fault Tree is given in Listing 4.4. With a little anticipation, we declared basic events. Note that components and fault trees may also contain definitions of parameters. Note also that the basic event BE1, which is declared in the component A, is used outside of this component (namely in the sibling component C).

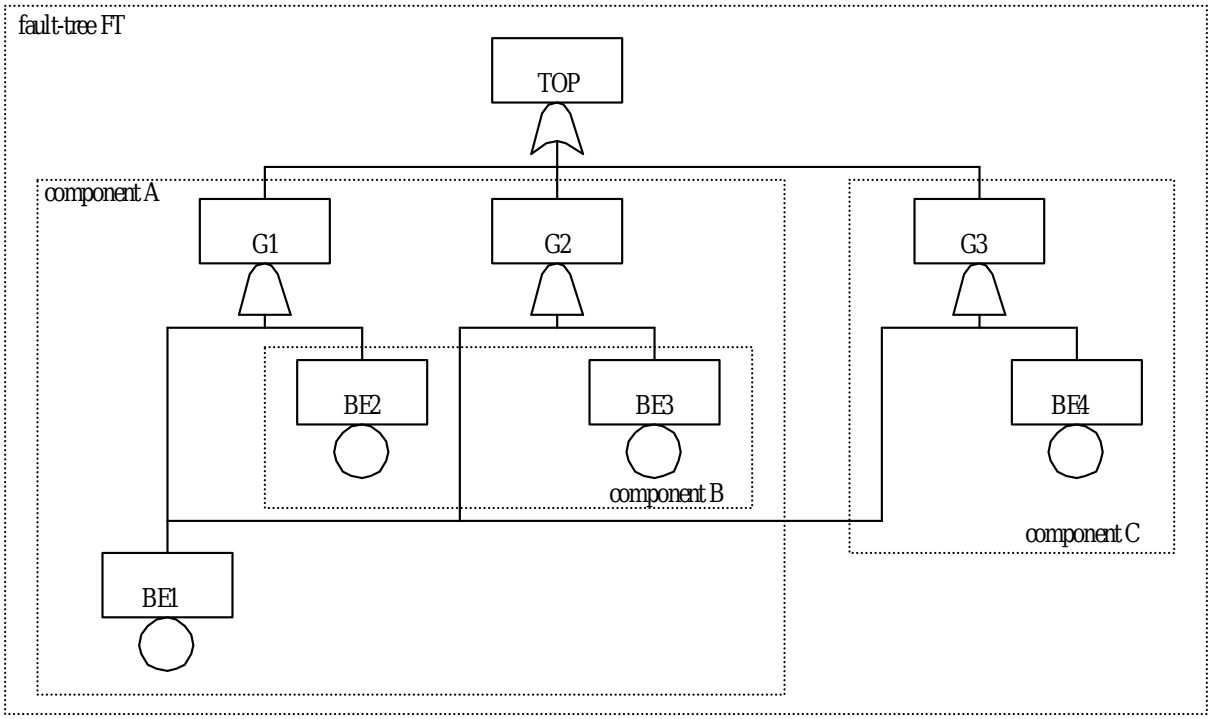


Fig. 4.2: A Fault Tree with Three Components

Listing 4.4: XML Representation for the Fault Tree pictured in Fig. 4.2

```
<define-fault-tree name="FT">
    <define-gate name="TOP">
        <or>
            <gate name="G1"/>
            <gate name="G2"/>
            <gate name="G3"/>
        </or>
    </define-gate>
    <define-component name="A">
        <define-gate name="G1">
            <and>
                <basic-event name="BE1"/>
                <basic-event name="BE2"/>
            </and>
        </define-gate>
        <define-gate name="G2">
            <and>
                <basic-event name="BE1"/>
                <basic-event name="BE3"/>
            </and>
        </define-gate>
        <define-basic-event name="BE1">
            <float value="1.2e-3"/>
        </define-basic-event>
        <define-component name="B">
            <define-basic-event name="BE2">
```

```
                <float value="2.4e-3"/>
            </define-basic-event>
            <define-basic-event name="BE3">
                <float value="5.2e-3"/>
            </define-basic-event>
        </define-component>
    </define-component>
    <define-component name="C">
        <define-gate name="G3">
            <and>
                <basic-event name="BE1"/>
                <basic-event name="BE4"/>
            </and>
        </define-gate>
        <define-basic-event name="BE4">
            <float value="1.6e-3"/>
        </define-basic-event>
    </define-component>
</define-fault-tree>
```

4.3.2 Solving Name Conflicts: Public versus Private Elements

By default, all elements of a model are public: they are visible everywhere in the model and they can be referred by their name. For instance, the basic event "BE1" of the fault tree pictured in Listing 4.4 can be just referred as "BE1". This principle is fairly simple. It may, however, cause some problem for large models, developed by several persons: it is hard to prevent the same name to be used twice, especially for what concerns gates (some software allow actually this possibility).

The Model Exchange Format makes it possible to declare elements of fault trees either as public or as private (to their inner most container). Unless declared otherwise, an element is public if its innermost container is public and private otherwise. For instance, if the component "A" of the fault tree pictured in Listing 4.4 is declared as private, then the component "B" (and its two basic events "BE2" and "BE3"), the gates "G1" and "G2", and the basic event "BE1" are private by default. There is no difference between public and private elements except that two private elements of two different containers may have the same name, while public elements must be uniquely defined.

There is actually three ways to refer an element:

- An element can be referred by its name. This works either if the element is public or if it is referred inside the container (fault tree or component) in which it is declared. For instance, if the basic event "BE1" is public, it can be referred as "BE1" anywhere in the model. If it is private, it can be referred as "BE1" only inside the component "A".
- An element can be referred by its full path (of containers), whether it is public or private. The names of containers should be separated with dots. For instance, the basic event "BE2" can be referred as "FT.A.B.BE2" anywhere in the model.

- Finally, an element can be referred by its local path, whether it is public or private. For instance, if the gate "G1" can be referred as "FT.A.G1" outside of the fault tree "FT", as "A.G1" inside the declaration of "FT", and finally as "G1" inside the declaration of the component "A". If the basic event BE1 is private (for a reason or another), it should be referred either as "FT.A.BE1" inside the component "C". In this case, the definition of the gate "G3" is as follows.

```
<define-gate name="G3">
    <and>
        <basic-event name="FT.A.BE1"/>
        <basic-event name="BE4"/>
    </and>
</define-gate>
```

The important point here is that it is possible to name two private elements of two different containers with the same identifier. For instance, if components "B" and "C" are private, it is possible to rename the basic-event "BE4" as "BE2". Outside these two components, the two basic events "B2" must be referred using their (local or global) paths.

4.3.3 Inherited attributes

Attributes associated with a container (fault tree, event tree or component) are automatically inherited by all the elements declared in the container. It is indeed possible to change the value of the attribute at element level.

4.3.4 Recommendations

Layered Models

In PSA models, fault trees are in general layered, i.e., arguments of connectives (and, or, etc.) are always either variables or negations of variables. Although there is no reason to force such a condition, it is recommended to obey it for the sake of clarity.

Use Portable Identifiers

In the XML description of fault trees, we intentionally did not define identifiers. In many fault tree tools, identifiers can be any string. It is, however, strongly recommended for portability issues to use non problematic identifiers, like those of programming languages, and to add a description of elements as a comment. This means not using lexical entities, such as spaces, tabulations, "." or "/", in names of elements, as well as realizing that some old tools cannot differentiate between capital and small letters.

The following is a general, recommended format that is likely to produce portable identifiers.

- Consistent with XML NCName datatype (XML Schema Part 2: Datatypes Second Edition, Derived datatypes, Section 3.3.7)
  - The first character must be alphabetic.
  - May contain alphanumeric characters and special characters like `_`, `-`.
  - No whitespace or other special characters like `:`, `,`, `/`, etc.
- No double hyphens `--`
- No trailing hyphen
- No periods `.` (reserved for references)

References to constructs, such as gates, events, and parameters, may include names of fault trees or components to access public or private members. This feature requires a period `.` between names; thus references may follow the pattern `fault_tree.component.event`.

In addition to the identifier format, the following is a set of recommendations for conforming tools to maximize the input acceptability:

- Avoid restricting the word character set (e.g., ASCII-only, English-only)
- Support popular character encodings (e.g., UTF-8, UTF-16)
- Provide case-sensitive identifier processing (e.g., no capital-letters-only restrictions)
- Sanitize input leading and trailing whitespace characters (i.e., insensitive to noise)

Role of Parameters, House Events, and Basic Events

Parameters, house events, and basic events should be always public, in order to facilitate their portability from one tool to another.

CHAPTER

5

STOCHASTIC LAYER

5.1 Description

The stochastic layer is populated with failure probabilities or failure probability distributions associated with basic events (in the event tree linking approach, functional events also can be associated with such a distribution). Probability distributions are described by (stochastic) expressions, which are terms, according to the terminology of Chapter 3. These expressions may depend on parameters (variables), so the stochastic layer can be seen a set of stochastic equations.

Stochastic equations associated with basic events actually play two roles:

- They are applied to calculate probability distributions of each basic event, i.e., for a given mission time t, the probability $Q(t)$ that the given basic event occurs before t. The probability distribution associated with a basic event is typically a negative exponential distribution of parameter λ:

$$Q(t) = 1 - e^{-\lambda t}$$

Note that, for the sake of the clarity, the Model Exchange Format represents explicitly the mission time as a parameter of a special type.

- Parameters are sometimes not known with certainty. Sensitivity analyses, such as Monte-Carlo simulations, are thus performed to study the change in risk due to this uncertainty. Expressions, therefore, are used to describe distributions of parameters. Typically, the parameter λ of a negative exponential distribution will be itself distributed according to a lognormal law of mean 0.001 and error factor 3.

Stochastic expressions are made of the following elements:

- Boolean and numerical constants

- Stochastic variables, i.e., parameters, including the special variable to represent the mission time
- Boolean and arithmetic operations (sums, differences, products, etc.)
- Built-in expressions that can be seen as macro-expressions that are used to simplify and shorten the writing of probability distributions (exponential, Weibull, etc.)
- Primitives to generate numbers at pseudo-random according to some probability distribution. The base primitive makes it possible to generate random deviates with a uniform probability distribution. Several other primitives are derived from this one to generate random deviates with normal, lognormal, or other distributions. Moreover, it is possible to define discrete distributions "by hand" through the notion of histogram.
- Directives to test the status of initial and functional events

Listing 5.1 presents the RNC schema for the constructs of the stochastic layer. Note that, conversely to variables (events) of the Fault Tree layer, parameters have to be defined (there is no equivalent to Basic Events).

Listing 5.1: The RNC schema for the constructs of the stochastic layer

```
expression =
  constant
  | parameter
  | operation
  | built-in
  | random-deviate
  | test-event

operation =
  numerical-operation | Boolean-operation | conditional-operation

built-in = exponential | GLM | Weibull | periodic-test | extern-function

random-deviate =
  uniform-deviate
  | normal-deviate
  | lognormal-deviate
  | gamma-deviate
  | beta-deviate
  | histogram

test-event = test-initiating-event | test-functional-event
```

The XML representation of the stochastic layer just reflects these different constructs.

Listing 5.2: The RNC schema for XML representation of expressions (main)

```
parameter-definition =
  element define-parameter {
    name,
```

```
    role?,
    attribute unit { units }?,
    label?,
    attributes?,
    expression
  }

parameter =
  element parameter {
    reference,
    attribute unit { units }?
  }
  | element system-mission-time {
      attribute unit { units }?
    }

units =
  "bool"
  | "int"
  | "float"
  | "hours"
  | "hours-1"
  | "years"
  | "years-1"
  | "fit"
  | "demands"

constant = bool | int | float

bool =
  element bool {
    attribute value { Boolean-value }
  }

int =
  element int {
    attribute value { xsd:integer }
  }

float =
  element float {
    attribute value { xsd:double }
  }
```

Operations, built-ins, and random deviates will be described in the following sections.

We believe that the formalism to define stochastic equations should be as large and as open as possible for at least two reasons: first, available tools already propose a large set of distributions; second, this is an easy and interesting way to widen the spectrum of PSA. The Model Exchange Format proposes a panoply of Boolean and arithmetic operators. More operations can be added on demand. A major step would be to introduce some

algorithmic concepts like loops and functions. At this stage, it does seem useful to introduce such advanced concepts in the Model Exchange Format.

5.2 Operations

5.2.1 Numerical Operation

Table 5.1 gives the list of arithmetic operators proposed by the Model Exchange Format. Their XML representation is given in Listing 5.3.

Table 5.1: Numerical Operations, their number of arguments, and their semantics

Operator	#arguments	Semantics
neg	1	unary minus
add	>1	addition
sub	>1	subtraction
mul	>1	multiplication
div	>1	division
pi	0	3.1415926535...
abs	1	absolute value
acos	1	arc cosine of the argument in radians
asin	1	arc sine of the argument in radians
atan	1	arc tangent of the argument in radians
cos	1	cosine
cosh	1	hyperbolic cosine
exp	1	exponential
log	1	(Napierian) logarithm
log10	1	decimal logarithm
mod	2	modulo
pow	2	power
sin	1	sine
sinh	1	hyperbolic sine
tan	1	tangent
tanh	1	hyperbolic tangent
sqrt	1	square root
ceil	1	first integer not less than the argument
floor	1	first integer not greater than the argument
min	>1	minimum
max	>1	maximum
mean	>1	mean

Listing 5.3: The RNC schema for XML representation of numerical operations

```
numerical-operation =
  element neg { expression }
  | element add { expression+ }
  | element sub { expression+ }
  | element mul { expression+ }
  | element div { expression+ }
  | element pi { empty }
  | element abs { expression }
  | element acos { expression }
  | element asin { expression }
  | element atan { expression }
  | element cos { expression }
  | element cosh { expression }
  | element exp { expression }
  | element log { expression }
  | element log10 { expression }
  | element mod { expression, expression }
  | element pow { expression, expression }
  | element sin { expression }
  | element sinh { expression }
  | element tan { expression }
  | element tanh { expression }
  | element sqrt { expression }
  | element ceil { expression }
  | element floor { expression }
  | element min { expression+ }
  | element max { expression+ }
  | element mean { expression+ }
```

Example

Assume, for instance, we want to associate a negative exponential distribution with a failure rate $\lambda = 1.23 \times 10^{-4} h^{-1}$ to the basic event "pump-failure". Using primitives defined above, we can encode explicitly such probability distribution as follows.

```
<define-basic-event name="pump-failure">
    <sub>
        <float value="1.0"/>
        <exp>
            <mul>
                <neg>
                    <parameter name="lambda"/>
                </neg>
                <system-mission-time/>
            </mul>
        </exp>
    </sub>
```

```
</define-basic-event>
<define-parameter name="lambda">
    <float value="1.23e-4"/>
</define-parameter>
```

5.2.2 Boolean Operations

Table 5.2 gives the list of Boolean operators proposed by the Model Exchange Format. Their XML representation is given in Listing 5.4.

Table 5.2: Boolean operators, their number of arguments, and their semantics

Operator	#arguments	Semantics
and	>1	$\wedge$
or	>1	$\vee$
not	1	$\neg$
eq	2	$=$
df	2	$\neq$
lt	2	$<$
gt	2	$>$
leq	2	$\leq$
geq	2	$\geq$

Listing 5.4: The RNC schema for XML representation of Boolean operations

```
Boolean-operation =
  element not { expression }
  | element and { expression+ }
  | element or { expression+ }
  | element eq { expression, expression }
  | element df { expression, expression }
  | element lt { expression, expression }
  | element gt { expression, expression }
  | element leq { expression, expression }
  | element geq { expression, expression }
```

5.2.3 Conditional Operations

The Model Exchange Format proposes two conditional operations: an "if-then-else" operation and a "switch/case" operation. The latter is a list of pairs of expressions introduced by the tag "case". The first expression of the pair should be a Boolean condition. If this condition is realized, then the second expression is evaluated, and its value returned. Otherwise, the next pair is considered.

The list ends with an expression in order to be sure that the switch has always a possible value. The XML representation for conditional operation is given in Listing 5.5.

Listing 5.5: The RNC schema for XML representation of conditional operations

```
conditional-operation = if-then-else-operation | switch-operation

if-then-else-operation =
  element ite { expression, expression, expression }

switch-operation = element switch { case-operation*, expression }

case-operation = element case { expression, expression }
```

Example

Assume, for instance, we want to give different values to the failure rate "lambda" depending on a global parameter "stress-level":

```
"lambda"=1.0e-4/h if "stress-level"=1,
"lambda"=2.5e-4/h if "stress-level"=2, and finally
"lambda"=1.0e-3/h if "stress-level"=3.
```

The value of "stress-level" will be modified while walking along the sequences of event trees or depending on the initiating event. Using primitives defined so far, we can encode the definition of "lambda" as follows.

```
<define-parameter name="lambda">
    <switch>
        <case>
            <eq>
                <parameter name="stress-level"/>
                <int value="1"/>
            </eq>
            <float value="1.0e-4"/>
        </case>
        <case>
            <eq>
                <parameter name="stress-level"/>
                <int value="2"/>
            </eq>
            <float value="2.5e-4"/>
        </case>
        <float value="1.0e-3"/>
    </switch>
</define-parameter>
```

5.3 Built-Ins

5.3.1 Description

Built-ins can be seen as macro arithmetic expressions. They are mainly used to simplify the writing of probability distributions. A special built-in "extern-function" makes it possible to define externally calculated built-ins. As for arithmetic operators, more built-ins can be added on demand to the Model Exchange Format. Here follows a preliminary list of built-ins. Table 5.3 summarizes this preliminary list.

Exponential with two parameters This built-in implements the negative exponential distribution. The two parameters are the hourly failure rate, usually called λ, and the time t.

$$P(t;\lambda) = 1 - e^{-\lambda t}$$

Exponential with four parameters (Generalized Life Model or GLM) This built-in generalizes the previous one. It makes it possible to take into account repairable components (through the hourly repairing rate μ) and failures on demand (through the probability γ of such an event). It takes four parameters, γ, the hourly failure rate λ, μ, and the time t (in this order).

$$P(t;\gamma,\lambda,\mu) = \frac{\lambda}{\lambda+\mu} - \frac{\lambda-\gamma(\lambda+\mu)}{\lambda+\mu} \times e^{-(\lambda+\mu)t}$$

Weibull This built-in implements the Weibull distribution. It takes four parameters: a scale parameter α, a shape parameter β, a time shift t_0, and the time t (in this order).

$$P(t;\alpha,\beta,t_0) = 1 - \exp\left[-\left(\frac{t-t_0}{\alpha}\right)^{\beta}\right]$$

Periodic test In several applications, it is of interest to introduce some specific distributions to describe periodically tested components. A further investigation is certainly necessary on this topic. We tentatively give here a candidate definition (that is extracted from one of the tools we considered).

The "periodic-test" built-in would take the following parameters (in order).

λ	failure rate when the component is working.
$\lambda*$	failure rate when the component is tested.
μ	repair rate (once the test showed that the component is failed).
τ	delay between two consecutive tests.
θ	delay before the first test.
γ	probability of failure due to the (beginning of the) test.
π	duration of the test.
x	indicator of the component availability during the test (1 available, 0 unavailable).
σ	test covering: probability that the test detects the failure, if any.
ω	probability that the component is badly restarted after a test or a repair.
t	the mission time.

Fig. 5.1 illustrates the meaning of the parameters τ, θ, and π.

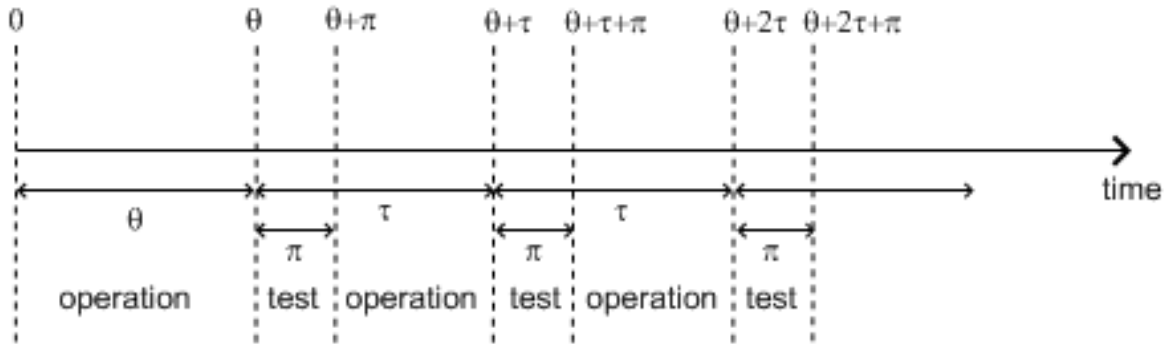


Fig. 5.1: Meaning of parameters τ, θ, and π of the "periodic-test" built-in

There are three phases in the behavior of the component. The first phase corresponds to the time from 0 to the date of the first test, i.e. θ. The second phase is the test phase. It spreads from times $\theta + n\tau$ to $\theta + n\tau + \pi$, with n any positive integer. The third phase is the functioning phase. It spreads from times $\theta + n\tau + \pi$ to $\theta + (n+1)\tau$.

In the first phase, the distribution is a simple exponential law of parameter λ.

The component may enter in the second phase in three states, either working, failed or in repair. In the latter case, the test is not performed. The Markov graphs for each of these cases are pictured in Fig. 5.2.

Fig. 5.2: Multi-phase Markov graph for the "periodic-test" built-in

Ai's, Fi's, Ri's states correspond respectively to states where the component is available, failed, and in repair. Dashed lines correspond to immediate transitions. Initial states are respectively A1, F1, and R1.

The situation is simpler in the third phase. If the component enters available this phase, the distribution follows an exponential law of parameter λ. If the component enters failed in this phase, it remains phase up to the next test. Finally, the Markov graph for the case where the component is in repair is the same as in the second phase.

The Model Exchange Format could also provide two simplified forms for the periodic test distribution.

Periodic-test with 5 arguments The first one takes five parameters: λ, μ, τ, θ, and t. In that case, the test is assumed to be instantaneous. Therefore, parameters

$\lambda*$ (the failure rate during the test) and x (indicator of the component availability during the test) are meaningless. There other parameters are set as follows.

- γ (the probability of failure due to the beginning of the test) is set to 0.
- σ (the probability that the test detects the failure, if any) is set to 1.
- ω (the probability that the component is badly restarted after a test or a repair) is set to 0.

Periodic-test with 4 arguments The second one takes only four parameters: λ, τ, θ, and t. The repair is assumed to be instantaneous (or equivalently the repair rate $\mu = +\infty$).

Extern functions The Model Exchange Format should provide a mean to call extern functions. This makes it extensible and allows linking the PSA assessment tools with complex tools to calculate physical behavior (like fire propagation or gas dispersion). This call may take any number of arguments and return a single value at once (some interfacing glue can be used to handle the case where several values have to be returned). It has been also suggested that extern function calls take XML terms as input and output. This is probably the best way to handle communication between tools, but it would be far too complex to embed XML into stochastic expressions.

Table 5.3: Built-ins, their number of arguments, and their semantics

Built-in	#arguments	Semantics
exponential	2	negative exponential distribution with hourly failure rate and time
GLM	4	negative exponential distribution with probability of failure on demand, hourly failure rate, hourly repair rate and time
Weibull	4	Weibull distribution with scale and shape parameters, a time shift and the time
periodic-test	11, 5 or 4	Distributions to describe periodically tested components
extern-function	any	call to an extern routine

5.3.2 XML Representation

The RNC schema for the XML representation of built-ins is given in Listing 5.6.

Listing 5.6: The RNC schema for XML representation of Built-ins

```
exponential = element exponential { expression, expression }

GLM = element GLM { expression, expression, expression, expression }

Weibull = element Weibull { expression, expression, expression, expression }
```

```
periodic-test = element periodic-test { expression+ }

extern-function = element extern-function { name, expression* }
```

Positional versus Named Arguments
We adopted a positional definition of arguments. For instance, in the negative exponential distribution, we assumed that the failure rate is always the first argument, and the mission time is always the second. An alternative way would be to name arguments, i.e., to enclose them into tags explicating their role. For instance, the failure rate would be enclosed in a tag "failure-rate", the mission time in a tag "time", and so on. The problem with this second approach is that many additional tags must be defined, and it is not sure that it helps a lot the understanding of the built-ins. Nevertheless, we may switch to this approach if the experience shows that the first one proves to be confusing.

Example

The negative exponential distribution can be encoded as follows.

```
<define-basic-event name="pump-failure">
    <exponential>
        <parameter name="lambda"/>
        <system-mission-time/>
    </exponential>
</define-basic-event>
```

5.4 Primitive to Generate Random Deviates

5.4.1 Description

Primitives to generate random deviates are the real stochastic part of stochastic equations. They can be used in two ways: in a regular context they return a default value (typically their mean value). When used to perform Monte-Carlo simulations, they return a number drawn at pseudo-random according to their type. The Model Exchange Format includes two types of random deviates: built-in deviates like uniform, normal or lognormal, and histograms that are user defined discrete distributions. A preliminary list of distributions is summarized in Table 5.4. As for arithmetic operators and built-ins, this list can be extended on demand.

Table 5.4: Primitive to generate random deviates, their number of arguments, and their semantics

Distribution	#arguments	Semantics
uniform-deviate	2	uniform distribution between a lower and an upper bounds
normal-deviate	2	normal (Gaussian) distribution defined by its mean and its standard deviation
lognormal-deviate	3 or 2	lognormal distribution defined by its mean, its error factor, and the confidence level of this error factor
gamma-deviate	2	gamma distributions defined by a shape and a scale factors
beta-deviate	2	beta distributions defined by two shape parameters α and β
histograms	any	discrete distributions defined by means of a list of pairs

Uniform Deviates These primitives describe uniform distributions in a given range defined by its lower- and upper-bounds. The default value of a uniform deviate is the mean of the range, i.e., `(lower-bound + upper-bound)/2`.

Normal Deviates These primitives describe normal distributions defined by their mean and their standard deviation (refer to a text book for a more detailed explanation). By default, the value of a normal distribution is its mean.

Lognormal distribution These primitives describe lognormal distributions defined by their mean $E(x)$ and their error factor EF. A random variable is distributed according to a lognormal distribution if its logarithm is distributed according to a normal distribution. If μ and σ are respectively the mean and the standard deviation of the associated normal distribution (the variable's natural logarithm), the probability density of the random variable is as follows.

$$f(x;\mu,\sigma) = \frac{1}{\sigma x\sqrt{2\pi}} \times \left[-\frac{1}{2}\left(\frac{\log x - \mu}{\sigma}\right)^2\right]$$

Its mean, $E(x)$, is defined as follows.

$$E(x) = \exp\left[\mu + \frac{\sigma^2}{2}\right]$$

The error factor EF for confidence level α is defined as follows:

$$EF_\alpha = \sqrt{\frac{X_\alpha}{X_{1-\alpha}}} = \exp[z_\alpha \cdot \sigma]$$

Where X_α is a left-tailed, upper bound corresponding to the α percentile, and z_α is the left-tailed z-score of the standard normal distribution for the α confidence level. Note that one-sided limits do not form confidence intervals.

$$z_\alpha = \sqrt{2} \cdot erf^{-1}(2\alpha - 1)$$
$$X_\alpha = \exp[\mu + z_\alpha \cdot \sigma]$$
$$X_{0.50} = e^\mu$$

For example, the error factor for a confidence level of 0.95:

$$z_{0.95} = -z_{0.05} = 1.645$$
$$X_{0.05} = \exp[\mu - 1.645\sigma]$$
$$X_{0.95} = \exp[\mu + 1.645\sigma]$$
$$EF_{0.95} = \sqrt{\frac{X_{0.95}}{X_{0.05}}} = \exp[1.645\sigma]$$

Once the mean and error factor are known for a certain confidence level, it is then possible to determine the distribution parameters of the lognormal law.

$$\sigma = \frac{\log EF_\alpha}{z_\alpha}$$
$$\mu = \log E(x) - \frac{\sigma^2}{2}$$

Alternatively, these parameters (μ, σ) of the underlying normal distribution can be supplied directly instead of the log-normal mean, error factor, and confidence level. These two parametrization schemes are distinguished by the number of parameters, i.e., 2 instead of 3.

Gamma Deviates These primitives describe Gamma distributions defined by their shape parameter k and their scale parameter θ. If k is an integer, then the distribution represents the sum of k exponentially distributed random variables, each of which has mean θ.

The probability density of the gamma distribution can be expressed in terms of the gamma function:

$$f(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k \Gamma(k)}$$

The default value of the gamma distribution is its mean, i.e., $k\theta$.

Beta Deviates These primitives describe Beta distributions defined by two shape parameters α and β.

The probability density of the beta distribution can be expressed in terms of the B function:

$$f(x;\alpha,\beta) = \frac{1}{B(\alpha,\beta)}x^{\alpha-1}(1-x)^{\beta-1}$$
$$B(x,y) = \int_0^1 t^{x-1}(1-t^{y-1})dt$$

The default value of the beta distribution is its mean, i.e., $\alpha/(\alpha+\beta)$.

Histograms Histograms are lists of pairs $(x_1, E_1), \ldots, (x_n, E_n)$, where the x_i's are numbers such that $x_i < x_{i+1}$ for $i = 1, \ldots, n-1$ and the E_i's are expressions.

The x_i's represent upper bounds of successive intervals. The lower bound of the first interval x_0 is given apart.

The drawing of a value according to a histogram is a two-step process. First, a value z is drawn uniformly in the range $[x_0, x_n]$. Then, a value is drawn at random by means of the expression E_i, where i is the index of the interval such that $x_{i-1} < z \leq x_i$.

By default, the value of a histogram is its mean, i.e.,

$$\mathbf{E}(X) = \frac{1}{x_n - x_0} \times \sum_{i=1}^{n} (x_i - x_{i-1}) \mathbf{E}(E_i)$$

Both Cumulative Distribution Functions and Density Probability Distributions can be translated into histograms.

A Cumulative Distribution Function is a list of pairs $(p_1, v_1), \ldots, (p_n, v_n)$, where the p_i's are such that $p_i < p_{i+1}$ for $i = 1, \ldots, n$ and $p_n = 1$. It differs from histograms in two ways. First, X axis values are normalized (to spread between 0 and 1); second, they are presented in a cumulative way. The histogram that corresponds to a Cumulative Distribution Function $(p_1, v_1), \ldots, (p_n, v_n)$ is the list of pairs $(x_1, v_1), \ldots, (x_n, v_n)$, with the initial value $x_0 = 0, x_1 = p_1$, and $x_i = p_i - p_{i-1}$ for all $i > 1$.

A Discrete Probability Distribution is a list of pairs $(d_1, m_1), \ldots, (d_n, m_n)$. The d_i's are probability densities. However, they could be any kind of values. The m_i's are midpoints of intervals and are such that $m_1 < m_2 < \ldots < m_n < 1$. The histogram that corresponds to a Discrete Probability Distribution $(d_1, m_1), \ldots, (d_n, m_n)$ is the list of pairs $(x_1, d_1), \ldots, (x_n, d_n)$, with the initial value $x_0 = 0, x_1 = 2m_1$, and $x_i = x_{i-1} + 2(m_i - x_{i-1})$.

5.4.2 XML Representation

The RNC schema for the XML representation of random deviates is given

Listing 5.7: The RNC schema for XML representation of random deviates

```
uniform-deviate = element uniform-deviate { expression, expression }

normal-deviate = element normal-deviate { expression, expression }

lognormal-deviate =
  element lognormal-deviate { expression, expression, expression? }

gamma-deviate = element gamma-deviate { expression, expression }

beta-deviate = element beta-deviate { expression, expression }

histogram = element histogram { expression, bin+ }

bin = element bin { expression, expression }
```

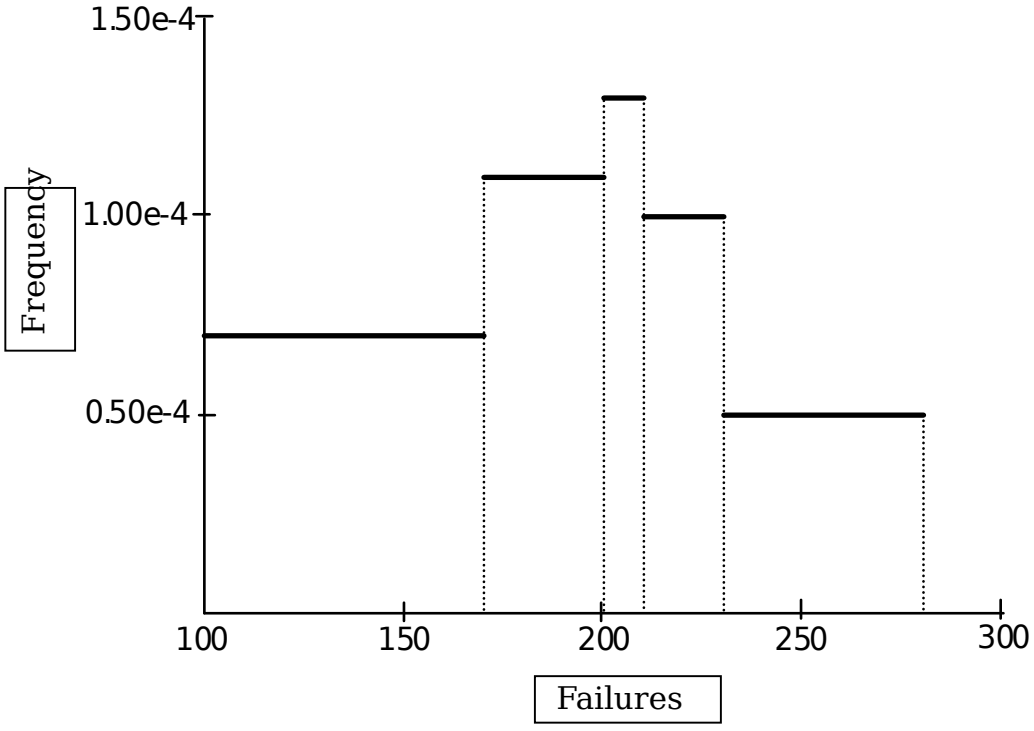
Example

Assume that the parameter "lambda" of a negative exponential distribution is distributed according to a lognormal distribution of mean 0.001 and error factor 3 for a confidence level of 95%. The parameter "lambda" is then defined as follows.

```
<define-parameter name="lambda">
    <lognormal-deviate>
        <float value="0.001"/>
        <float value="3"/>
        <float value="0.95"/>
    </lognormal-deviate>
</define-parameter>
```

Example

Assume that the parameter "lambda" has been sampled outside of the model and is distributed according to the following histogram.



The XML encoding for "lambda" is as follows.

```
<define-parameter name="lambda">
    <histogram>
        <float value="100"/>
        <bin> <float value="170"/> <float value="0.70e-4"/> </bin>
        <bin> <float value="200"/> <float value="1.10e-4"/> </bin>
        <bin> <float value="210"/> <float value="1.30e-4"/> </bin>
        <bin> <float value="230"/> <float value="1.00e-4"/> </bin>
```

```
        <bin> <float value="280"/> <float value="0.50e-4"/> </bin>
    </histogram>
</define-parameter>
```

5.5 Directives to Test the Status of Initiating and Functional Events

5.5.1 Description

The Model Exchange Format provides two special directives to test whether a given initiating event occurred and whether a given functional event is in a given state. The meaning of these directives will be further explained in Section 7.3.

Table 5.5 presents these directives and their arguments.

Table 5.5: Directives to test the status of initiating and functional events

Built-in	#arguments	Semantics
test-initiating-event	1	returns true if the initiating event of the given name occurred.
test-functional-event	2	returns true if the functional event of the given name is in the given state.

5.5.2 XML Representation

The XML representation for directives to test the status of initiating and functional events is given in Listing 5.8.

Listing 5.8: The RNC schema for XML representation of directives to test the status of initiating and functional events

```
test-initiating-event = element test-initiating-event { name }

test-functional-event =
  element test-functional-event {
    name,
    attribute state { Identifier }
  }
```

CHAPTER

6

META-LOGICAL LAYER

The meta-logical layer is populated with constructs like common cause groups, delete terms, recovery rules, and exchange events that are used to give flavors to fault trees. This chapter reviews all these constructs.

6.1 Common Cause Groups

6.1.1 Description

From a theoretical view point, one of the basic assumptions of the fault tree technique is that occurrences of basic events are independent from a statistical viewpoint. However, most of the PSA models include, to a large extent, so-called common cause groups. Common cause groups are groups of basic events whose failure are not independent from a statistical view point. They may occur either independently or dependently due to a common cause failure. So far, existing tools embed three models for common cause failures (CCF): the beta-factor model, the Multiple Greek letters (MGL) model, and the alpha-factor model. Alpha-factor and MGL models differ only in the way the factors for each level (2 components fail, 3 components fail, etc.) are given. The Model Exchange Format proposes the three mentioned models plus a fourth one, so-called phi-factor, which is a more direct way to set factors.

Beta-Factor The β-factor model assumes that if a common cause occurs, then all components of the group fail simultaneously. Components can fail independently. Multiple independent failures are neglected. The β-factor model assumes, moreover, that all the components of the group have the same probability distribution. It is characterized by this probability distribution and the conditional probability β that all components fail, given that one component failed.

Let $BE_1, BE_2, \ldots, BE_n$ be the n basic events of a common cause group with a probability distribution Q and a beta-factor β. Applying the beta-factor model on the fault tree consists in following operations.

1. Create new basic events BE_{CCF_i} for each BE_i to represent the independent occurrence of BE_i and BE_{CCF_i} to represent the occurrence of all BE_i together.
2. Substitute a gate $G_i = BE_{CCF_i} \vee BE_i$ for each basic event BE_i.
3. Associate the probability distribution (e.g., $\beta \times Q$) with the event BE_{CCF_i}.

Multiple Greek Letters The Multiple Greek Letters (MGL) model generalizes the beta-factor model. It considers the cases where sub-groups of $1, 2, \ldots, n-1$ components of the group fail together. This model is characterized by the probability distribution of failure of the components, and $n-1$ factors $\rho_2, \ldots, \rho_n$, ρ_k denotes the conditional probability that k components of the group fail given that $k-1$ failed.

Let $BE_1, BE_2, \ldots, BE_n$ be the n basic events of a common cause group with a probability distribution Q and factors $\rho_2, \ldots, \rho_n$. Applying the MGL model on the fault tree consists in following operations.

1. Create a basic event for each combination of basic events of the group (there are $2^n - 1$ such combinations).
2. Transform each basic event BE_i into an OR-gate G_i over all newly created event basic events that represent a group that contains BE_i.
3. Associate the following probability distribution with each newly created basic event representing a group of k components (with $\rho_{n+1} = 0$).

$$Q_k = \frac{1}{\binom{n-1}{k-1}} \times \left(\prod_{i=2}^{k} \rho_i \right) \times (1 - \rho_{k+1}) \times Q$$

For instance, for a group of 4 basic events: A, B, C, and D, the basic event A is transformed into a gate $G_A = A \vee AB \vee AC \vee AD \vee ABC \vee ABD \vee ACD \vee ABDC$, and the Q_k's are as follows.

$$\begin{aligned}
Q_1 &= (1 - \rho_2) \times Q \\
Q_2 &= \tfrac{1}{3} \times \rho_2 \times (1 - \rho_3) \times Q \\
Q_3 &= \tfrac{1}{3} \times \rho_2 \times \rho_3 \times (1 - \rho_4) \times Q \\
Q_4 &= \rho_2 \times \rho_3 \times \rho_4 \times Q
\end{aligned}$$

Note that if $\rho_k = 0$, then $Q_k, Q_{k+1}, \ldots$ are null as well. In such a case it is not necessary to create the groups with k elements or more.

Alpha-Factor The alpha-factor model is the same as the MGL model except in the way the factors are given. Here n factors $\alpha_1, \ldots, \alpha_n$ are given. α_k represents the fraction of the total failure probability due to common cause failures that impact exactly k components. The distribution associated with a group of size k is as follows:

$$Q_k = \frac{k}{\binom{n-1}{k-1}} \times \frac{\alpha_k}{\sum_{i=1}^{n} i \cdot \alpha_i} \times Q$$

Phi-Factor The phi-factor model is the same as MGL and alpha-factor models except that factors for each level are given directly.

Indeed, the sum of the ϕ_i's should equal 1.

6.1.2 XML representation

The RNC schema for the XML description of Common Cause Failure Groups is given in Listing 6.1. Note that the number of factors depends on the model. Tools are in charge of checking that there is the good number of factors. Note also that each created basic event is associated with a factor that depends on the model and the level of the basic event. The sum of the factors associated with basic events of a member of the CCF group should be equal to 1; although, this is not strictly required by the Model Exchange Format.

Listing 6.1: The RNC schema for the XML representation of CCF-groups

```
CCF-group-definition =
  element define-CCF-group {
    name,
    attribute model { CCF-model },
    label?,
    attributes?,
    members,
    distribution,
    factors
  }

members = element members { basic-event+ }

factors =
  element factors { factor+ }
  | factor

factor =
  element factor {
    attribute level { xsd:positiveInteger }?,
    expression
  }

distribution = element distribution { expression }

CCF-model = "beta-factor" | "MGL" | "alpha-factor" | "phi-factor"
```

Example

Here follows a declaration of a CCF-group with four elements under the MGL model.

```
<define-CCF-group name="pumps" model="MGL">
    <members>
        <basic-event name="pumpA"/>
        <basic-event name="pumpB"/>
        <basic-event name="pumpC"/>
        <basic-event name="pumpD"/>
    </members>
    <factors>
        <factor level="2">
            <float value="0.10"/>
        </factor>
        <factor level="3">
            <float value="0.20"/>
        </factor>
        <factor level="4">
            <float value="0.30"/>
        </factor>
    </factors>
    <distribution>
        <exponential>
            <parameter name="lambda"/>
            <system-mission-time/>
        </exponential>
    </distribution>
</define-CCF-group>
```

6.2 Delete Terms, Recovery Rules, and Exchange Events

6.2.1 Description

Delete Terms Delete Terms are groups of pairwise exclusive basic events, used to model impossible configurations. A typical example is the case where:

- The basic event a can only occur when the equipment A is in maintenance.
- The basic event b can only occur when the equipment B is in maintenance.
- Equipment A and B are redundant and cannot be simultaneously in maintenance.

In most of the tools, delete terms are considered as a post-processing mechanism: minimal cut sets containing two basic events of a delete terms are discarded. In order to speed-up calculations, some tools use basic events to discard minimal cut sets on the fly, during their generation.

Delete Terms can be handled in several ways. Let $G = \{e_1, e_2, e_3\}$ be a Delete Term (group).

- A first way to handle G, is to use it to post-process minimal cut sets, or to discard them on the fly during their generation. If a minimal cut set contains

at least two of the elements of G, it is discarded.

- A global constraint $C_G = \neg\binom{3}{2}(e_1, e_2, e_3)$ is introduced, and each top event (or event tree sequences) "top" is rewritten as $top \wedge C_G$.
- As for Common Causes Groups, the e_i's are locally rewritten in as gates:
  - e_1 is rewritten as a gate $ge_1 = e_1 \wedge \neg e_2 \wedge \neg e_3$
  - e_2 is rewritten as a gate $ge_2 = e_2 \wedge \neg e_1 \wedge \neg e_3$
  - e_3 is rewritten as a gate $ge_3 = e_3 \wedge \neg e_1 \wedge \neg e_2$

Recovery Rules Recovery Rules are an extension of Delete Terms. A Recovery Rule is a couple (H, e), where H is a set of basic events, and e is a (fake) basic event. It is used to post-process minimal cut sets: if a minimal cut set C contains H, the e is added to C. Recovery Rules are used to model actions taken in some specific configurations to mitigate the risk (hence their name).

Here several remarks can be made.

- It is possible to mimic Delete Terms by means of recovery rules. To do so, it suffices to assign the basic event e to the value "false" or the probability 0.0.
- As for Delete Terms, it is possible to give purely logical interpretation to Recovery Rules. The idea is to add a global constraint $H \Rightarrow e$, i.e., $\neg H \vee e$, for each Recovery Rule (H, e).
- Another definition of Recovery Rules as a post-processing is that the event e is substituted for subset H in the minimal cut set. This definition, however, has the major drawback by being impossible to interpret with a Boolean logic. No Boolean formula can withdraw events from a configuration.

Exchange Events Exchange Events are very similar to Recovery Rules. An Exchange Event (Rule) is a triple (H, e, e'), where H is a set of basic events, and e and e' are two basic events. Considered as a post-processing of minimal cut sets, such a rule is interpreted as follows. If the minimal cut set contains both the set H and the basic event e, then the basic event e' is substituted for e in the cut set. For the same reason as above, Exchange Events cannot be interpreted with a Boolean logic.

6.2.2 All Extra-Logical Constructs in One: the Notion of Substitution

Constructs that cannot be interpreted with a Boolean logic should be avoided for at least two reasons. First, models containing such constructs are not declarative. Second, and more importantly, they tighten assessment tools to one specific type of algorithms. The second interpretation of Recovery Rules and Exchange Events tighten the models to be assessed by means of the minimal cut sets approach.

Nevertheless, Recovery Rules and Exchange Events are useful and broadly used in practice. Fortunately, Exchange Events (considered as a post processing mechanism) can be avoided in many cases by using the instructions that give flavors to fault trees while walking along event tree sequences: in a given sequence, one may decide to substitute

the event e' for the event e (or the parameter p' for the parameter p) in the Fault Trees collected so far. This mechanism is perfectly acceptable because it applies while creating the Boolean formula to be assessed.

It is not yet possible to decide whether Recovery Rules (under the second interpretation) and Exchange Events can be replaced by purely declarative constructs or by instructions of event trees. This has to be checked on real-life models. To represent Delete Term, Recovery Rules and Exchange Events, the Model Exchange Format introduces a unique construct: the notion of substitution.

A substitution is a triple (H, S, t) where:

- H, the hypothesis, is a (simple) Boolean formula built over basic events.
- S, the source, is also a possibly empty set of basic events.
- t, the target, is either a basic event or a constant.

Let C be a minimal cut set, i.e., a set of basic events. The substitution (H, S, t) is applicable on C if C satisfies H (i.e., if H is true when C is realized). The application of (H, S, t) on C consists in removing from C all the basic events of S and in adding to C the target t.

Note that if t is the constant "true", adding t to C is equivalent to adding nothing. If t is the constant "false", adding t to C is equivalent to discard C.

This notion of substitution generalizes the notions of Delete Terms, Recovery Rules, and Exchange Events:

- Let $D = \{e_1, e_2, \ldots, e_n\}$ be a group of pairwise exclusive events (a Delete Term). Then D is represented as the substitution $(\binom{n}{2}(e_1, e_2, \ldots, e_n), \varnothing, \text{false})$.
- Let (H, e) be a Recovery Rule, under the first interpretation, where $H = \{e_1, e_2, \ldots, e_n\}$. Then, (H, e) is represented by the substitution $(e_1 \wedge e_2 \wedge \ldots \wedge e_n, \varnothing, e)$.
- Let (H, e) be a Recovery Rule, under the second interpretation, where $H = \{e_1, e_2, \ldots, e_n\}$. Then (H, e) is represented by the substitution $(e_1 \wedge e_2 \wedge \ldots \wedge e_n, H, e)$.
- Finally, let (H, e, e') be an Exchange Event Rule, where $H = \{e_1, e_2, \ldots, e_n\}$. Then (H, e, e') is represented by the substitution $(e_1 \wedge e_2 \wedge \ldots \wedge e_n \wedge e, e, e')$.

Note that a substitution $(H, \varnothing, t)$ can always be interpreted as the global constraint $H \Rightarrow t$.

6.2.3 XML Representation

The RNC schema for the XML description of substitutions is given in Listing 6.2. The optional attribute "type" is used to help tools that implement "traditional" substitutions.

Listing 6.2: The RNC schema for the XML representation of exclusive-groups

```
substitution-definition =
  element define-substitution {
    name?,
    attribute type { xsd:string }?,
    label?,
    attributes?,
    element hypothesis { formula },
    element source { basic-event+ }?,
    element target { basic-event+ | Boolean-constant }
  }
```

Example

Assume that Basic Events "failure-pump-A", "failure-pump-B", and "failure-pump-C" are pairwise exclusive (they form a delete term) because they can only occur when, respectively, equipment A, B, and C are under maintenance and only one equipment can be in maintenance at once. The representation of such a delete term is as follows.

```
<define-substitution name="pumps" type="delete-terms">
    <hypothesis>
        <atleast min="2">
            <basic-event name="failure-pump-A"/>
            <basic-event name="failure-pump-B"/>
            <basic-event name="failure-pump-C"/>
        </atleast>
    </hypothesis>
    <target>
        <constant value="false"/>
    </target>
</define-substitution>
```

Example

Assume that if the valve V is broken, and an overpressure is detected in pipe P, then a mitigating action A is performed. This is a typical Recovery Rule (under the first interpretation), where the hypothesis is the conjunction of Basic Events "valve-V-broken" and "overpressure-pipe-P", and the added Basic Event is "failure-action-A". It is encoded as follows.

```
<define-substitution name="mitigation" type="recovery-rule">
    <hypothesis>
        <and>
            <basic-event name="valve-V-broken"/>
            <basic-event name="overpressure-pipe-P"/>
        </and>
```

```
    </hypothesis>
    <target>
        <basic-event name="failure-action-A"/>
    </target>
</define-substitution>
```

Example

Assume that if magnitude of the earthquake is 5, 6 or 7, the size of a leak of a given pipe P gets large, while it is small for magnitudes below 5. We can use an exchange event rule to model this situation.

```
<define-substitution name="magnitude-impact" type="exchange-event">
    <hypothesis>
        <or>
            <basic-event name="magnitude-5"/>
            <basic-event name="magnitude-6"/>
            <basic-event name="magnitude-7"/>
        </or>
    </hypothesis>
    <source>
        <basic-event name="small-leak-pipe-P"/>
    </source>
    <target>
        <basic-event name="large-leak-pipe-P"/>
    </target>
</define-substitution>
```

CHAPTER

7

EVENT TREE LAYER

7.1 Preliminary Discussion

The first three layers are rather straightforward to describe since there is a general agreement on how to interpret fault trees and probability distributions. The Event Tree layer is much more delicate to handle. The reason stands in the dynamic nature of event trees and the lack of common interpretation for this formalism. To illustrate this point, we shall consider the toy example pictured in Fig. 7.1.

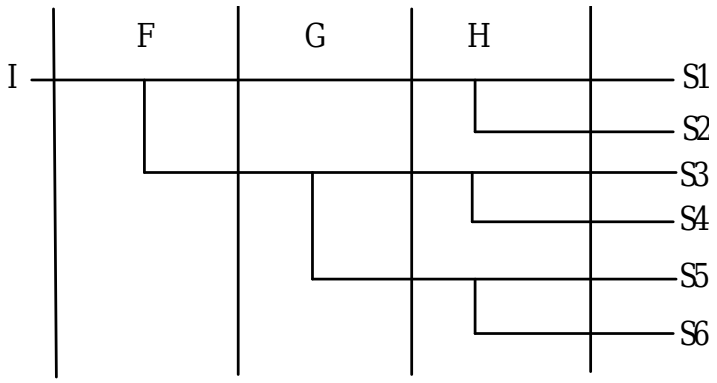


Fig. 7.1: A Small Event Tree

This event tree is made of the following elements.

- An initiating event I
- Three functional events F, G, and H
- Six sequences ending in six (a priori) different states S1 to S6

The expected interpretation of this event tree is as follows. A fault tree is associated with each functional event. This fault tree describes how the functional event may occur. For the sake of simplicity, we may assume that its top-event has the same name as the functional event itself. Upper branches represent a success of the corresponding safety mission, while lower branches represent its failure. Applying the so-called fault tree linking approach, we obtain the following interpretation for the sequences.

$$
\begin{aligned}
S1 &= I \wedge \neg F \wedge \neg H \\
S2 &= I \wedge \neg F \wedge H \\
S3 &= I \wedge F \wedge \neg G \wedge \neg H \\
S4 &= I \wedge F \wedge \neg G \wedge H \\
S5 &= I \wedge F \wedge G \wedge \neg F \\
S6 &= I \wedge F \wedge G \wedge H
\end{aligned}
$$

In practice, things are less simple:

- There may be more than one initiating event because the same event tree can be used with different flavors.
- Values of house events may be changed at some points along the branches to give flavors to fault trees. The value of a house event may be changed either locally to a fault tree, or for all the fault trees encountered after the setting point.
- The flavoring mechanism may be even more complex: some gates or basic events may be negated; some parameters of probability distributions may be impacted.
- The flavor given to a fault tree may depend on what has happened so far in the sequence: initiating event, value of house events, etc.
- Some success branches may not be interpreted as the negation of the associated fault tree but rather as a bypass. This interpretation of success branches is typically tool-dependent: some tools (have options to) ignore success branches; therefore, modelers use this "possibility" to "factorize" models.
- Branching may have more than two alternatives, or represent multi-states, not just success and failure, each alternative being labeled with a different fault tree.
- In the event tree linking approach, branching may involve no fault tree at all, but rather a multiplication by some factor of the current probability of the sequence.
- It is sometimes convenient to replace a sub-tree by a reference to a previously defined sub-tree. For instance, if we identify end-states S1 and S3 one the one hand, S2 and S4 on the other hand, we can merge the two corresponding sub-trees rooted. It saves space (both in computer memory and onto the display device) to replace the latter by a reference to the former.

In a word, event trees cannot be seen as a static description formalism like fault trees. Rather, they should be seen as a kind of graphical programming language. This language is used to collect and modify data when walking along the sequences, and even to decide when to stop to walk a sequence (in the event tree linking approach). The Model Exchange Format should thus reflect this programming nature of event trees.

7.2 Structure of Event Trees

7.2.1 Description

The Model Exchange Format distinguishes the structure of the event trees, i.e., the set of sequences they encode, from what is collected along the sequences and how it is collected. Let us consider for now only the structural view point. With that respect, an event tree is made of the following components.

- One or more initiating events
- An ordered set of functional events (the columns)
- A set of end-states (so called sequences)
- A set of branches to describe sequences

Branches end up either with a sequence name or with a reference to another branch (such references are sometimes called transfers). They contain forks. Each fork is associated with a functional event. The initiating event could also be seen as a special fork (between the occurrence of this event and the occurrence of no event). In the Model Exchange Format, alternatives of the fork are called paths. Paths are labeled by the state of the functional event that labels the fork.

Let us consider again the event tree pictured in Fig. 7.1. Assume that end states S1 and S3 on the one hand, S2 and S4 on the other hand, are identical, and that we merge the corresponding sub-trees. Assume moreover that the lowest success branch of the functional event H is actually a bypass. Then, the structure of the tree is pictured in Fig. 7.2. On this figure, the nodes of the tree are numbered from 1 to 8. The initiating event is represented as a fork. Finally, the branch (the sub-tree) rooted by the node 2 is named B1.

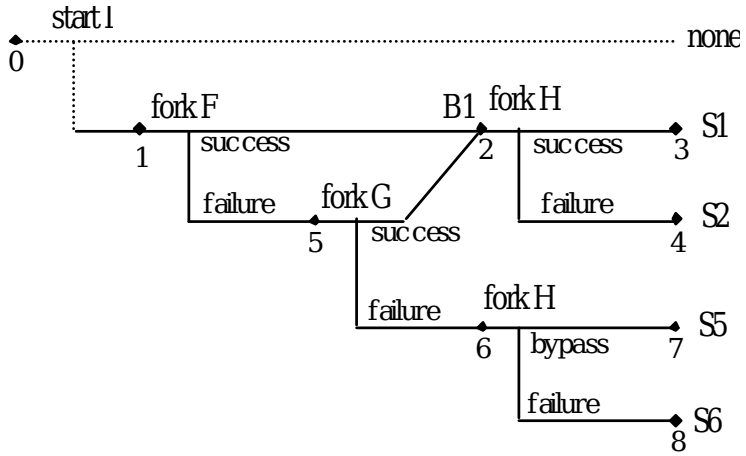


Fig. 7.2: Structure of an Event Tree

Components of the event tree pictured in Fig. 7.2 are the following.

- The initiating event I

- The three functional events F, G, and H
- The end states S1, S2, S5, and S6
- The branch B1
- The tree rooted by the initial node (here the node 1)

Forks decompose the current branch according to the state of a functional event. Usually, this state is either "success" or "failure". It may be "bypass" as well (as in our example for the path from node 6 to node 7). In the case of multiple branches, the name of a state is defined by the user.

Instructions to collect and to modify fault trees and probability distributions are applied at the different nodes. Instructions to be applied may depend on the initiating event and the states of functional events.

The states of functional events at a node depend on the path that has been followed from the root node to this node. By default, functional events are in an unspecified state, i.e., that the predicate "test-functional-event" (see Section 5.5) returns false in any case. Table 7.1 gives the states of functional events for all the possible paths starting from the root node of the event tree pictured in Fig. 7.2. Empty cells correspond to unspecified states.

Table 7.1: States of Functional Events for the different paths of the Event Tree in Fig. 7.2

path	F	G	H
1			
1-2	success		
1-2-3	success		success
1-2-4	success		failure
1-5	failure		
1-5-2	failure	success	
1-5-2-3	failure	success	success
1-5-2-4	failure	success	failure
1-5-6	failure	failure	
1-5-6-7	failure	failure	bypass
1-5-6-8	failure	failure	failure

As mentioned above, an event tree may be parametric: the same tree can be used for several initiating events. To implement this idea, the Model Exchange Format provides the analyst with the notion of group of initiating events. Such a group has a name and may contain sub-groups. Groups of initiating events may be freely defined inside or outside event trees. There is one condition, however: an initiating event can be used in only one tree.

7.2.2 XML Representation

We are now ready to explicitly define the XML grammar of the structure of event trees. Its RNC schema is given in Listing 7.1 and Listing 7.2. In these figures, we leave instructions unspecified, for they do not concern the structure of the tree and are the subject of the next section. Note that branches and functional events cannot be declared (nor referred to) outside event trees, for there would be no meaning in doing so.

Listing 7.1: The RNC schema of the XML representation of initiating events

```
initiating-event-definition =
  element define-initiating-event {
    name,
    attribute event-tree { Identifier }?,
    label?,
    attributes?,
    (collected-item | consequence | consequence-group)?
  }

initiating-event-group-definition =
  element define-initiating-event-group {
    name,
    attribute event-tree { Identifier }?,
    label?,
    attributes?,
    initiating-event+
  }

initiating-event =
  element initiating-event { name }
  | element initiating-event-group { name }

collected-item = basic-event | gate | parameter
```

Listing 7.2: The RNC schema of the XML representation of event trees and sequences

```
event-tree-definition =
  element define-event-tree {
    name,
    label?,
    attributes?,
    functional-event-definition*,
    sequence-definition*,
    branch-definition*,
    initial-state
  }

functional-event-definition =
  element define-functional-event { name, label?, attributes? }

sequence-definition =
```

```
  element define-sequence { name, label?, attributes?, instruction* }

branch-definition =
  element define-branch { name, label?, attributes?, branch }

initial-state = element initial-state { branch }

branch = instruction*, (fork | end-state)

fork =
  element fork {
    attribute functional-event { Identifier },
    path+
  }

path =
  element path {
    attribute state { Identifier },
    branch
  }

end-state =
  element sequence { name }
  | element branch { name }
```

Example

Consider again the event tree pictured in Fig. 7.2. The XML description for this example is given in Listing 7.3.

Listing 7.3: XML representation for the structure of the Event Tree pictured in Fig. 7.2

```
<define-event-tree name="my-first-event-tree">
    <define-functional-event name="F"/>
    <define-functional-event name="G"/>
    <define-functional-event name="H"/>
    <define-sequence name="S1"/>
    <define-sequence name="S2"/>
    <define-sequence name="S5"/>
    <define-sequence name="S6"/>
    <define-branch name="sub-tree7">
        <fork functional-event="H">
            <path state="success">
                <sequence name="S1"/>
            </path>
            <path state="failure">
                <sequence name="S2"/>
            </path>
        </fork>
```

```
    </define-branch>
    <initial-state>
        <fork functional-event="F">
            <path state="success">
                <branch name="sub-tree7"/>
            </path>
            <path state="failure">
                <fork functional-event="G">
                    <path state="success">
                        <branch name="sub-tree7"/>
                    </path>
                    <path state="failure">
                        <fork functional-event="H">
                            <path state="success">
                                <sequence name="S5"/>
                            </path>
                            <path state="failure">
                                <sequence name="S6"/>
                            </path>
                        </fork>
                    </path>
                </fork>
            </path>
        </fork>
    </initial-state>
</define-event-tree>
```

7.3 Instructions

7.3.1 Description

Fig. 7.2 gives the XML representation for the structure of an event tree. This structure makes it possible to walk along the sequences, but not to construct the Boolean formulae associated with each sequence. To do so, we need to fill the structure with instructions. Instructions are actually used for two main purposes:

- To collect formulae or stochastic expressions
- To define flavors of fault trees and probability distributions, i.e., to set values of house events and flag parameters

The collection of a top event consists in a Boolean product of the formula associated with the sequence and a copy of the fault tree rooted with the top event. In the Model Exchange Format, the operation is performed by means of the instruction “collect-formula”. The collection of an expression multiplies the current probability of the sequence by the value of this expression. In the Model Exchange Format, the operation is performed by means of the instruction “collect-expression”.

To give flavors to fault trees, i.e., to change the values of gates, house events, basic

events, and parameters, the Model Exchange Format introduces the four corresponding instruction: “set-gate”, “set-house-event”, “set-basic-event”, and “set-parameter”.

Sequences are walked from left to right. Therefore, when a value of an element is changed, this change applies on the current environment and propagates to the right. This default behavior can be changed by using the flag “direction”, which can take either the value “forward” (the default), “backward” or “both”. This feature should be handled with much care.

The flavor given to fault trees, as well as what is collected, may depend on the initial event and the current state of functional events. To do so, the Model Exchange Format provides an if-then-else instruction (the “else” part is optional) and the two expressions “test-initiating-event” and “test-functional-event”. These two instructions have been introduced in Section 5.5. Since the then- and else-branches of the “if-then-else” may contain several instructions, the Model Exchange Format introduces the notion of block of instructions.

Finally, some models require linking event trees. A special instruction “event-tree” is introduced for this purpose. It should be used only in sequence definitions, i.e., in end-state.

It is sometimes the case that the same values of house events and parameter flags are used at several places. Such a configuration is called a split-fraction in the event tree linking approach. The Model Exchange Format refers it as a rule, for it is a sequence of instructions.

7.3.2 XML Representation

The RNC schema for the XML representation of instructions is given in Listing 7.4.

Listing 7.4: The RNC schema for the XML representation of instructions

```
instruction = set | collect | if-then-else | block | rule | link

set = set-gate | set-house-event | set-basic-event | set-parameter

set-gate =
  element set-gate {
    name,
    attribute direction { direction }?,
    formula
  }

set-house-event =
  element set-house-event {
    name,
    attribute direction { direction }?,
    Boolean-constant
  }

set-basic-event =
```

```
  element set-basic-event {
    name,
    attribute direction { direction }?,
    expression
  }

set-parameter =
  element set-parameter {
    name,
    attribute direction { direction }?,
    expression
  }

direction = "forward" | "backward" | "both"

if-then-else = element if { expression, instruction, instruction? }

collect = collect-formula | collect-expression

collect-formula = element collect-formula { formula }

collect-expression = element collect-expression { expression }

block = element block { instruction* }

rule = element rule { name }

link = element event-tree { name }

rule-definition =
  element define-rule { name, label?, attributes?, instruction+ }
```

Example

Consider again the event tree pictured in Fig. 7.2. The XML representation for the structure of this tree has been given in Listing 7.3. Assume that the success branch of the lower fork on system H is a bypass. The XML description for the branches of this example is given in Listing 7.5. It is easy to verify by traversing this tree by hand so that it produces the expected semantics.

Listing 7.5: XML representation of the branches of the event tree pictured in Fig. 7.2

```
<define-event-tree name="my-first-event-tree">
    ...
    <initial-state>
        <fork functional-event="F">
            <path state="success">
                <collect-formula> <not> <gate name="F"/> </not> </collect-
↪formula>
```

```
                <branch name="sub-tree7"/>
            </path>
            <path state="failure">
                <collect-formula> <gate name="F"/> </collect-formula>
                <fork functional-event="G">
                    <path state="success">
                        <collect-formula> <not> <gate name="G"/> </not> </
↪collect-formula>
                        <branch name="sub-tree7"/>
                    </path>
                    <path state="failure">
                        <collect-formula> <gate name="G"/> </collect-formula>
                        <fork functional-event="H">
                            <path state="bypass">
                                <!-- here nothing is collected -->
                                <sequence name="S5"/>
                            </path>
                            <path state="failure">
                                <collect-formula> <gate name="H"/> </collect-
↪formula>
                                <sequence name="S6"/>
                            </path>
                        </fork>
                    </path>
                </fork>
            </path>
        </fork>
    </initial-state>
</define-event-tree>
```

This example does not set any house events or flag parameters. To set a house event for all subsequent sub-tree exploration (including the next fault tree to be collected), it suffices to insert an instruction "set" in front of the instruction "collect".

```
<set-house-event name="h1"> <bool value="true"/> </set-house-event>
<collect-formula> <gate name="G"/> </collect-formula>
```

To set the same house event locally for the next fault tree to be collected, it suffices to set back its value to "false" after gathering of the fault tree.

```
<set-house-event name="h1"> <bool value="true"/> </set-house-event>
<collect-formula> <gate name="G"/> </collect-formula>
<set-house-event name="h1"> <bool value="false"/> </set-house-event>
```

The same principle applies to parameters.

Assume now that we want to set the parameters "lambda1" and "lambda2" of some probability distributions to "0.001" if the initiating event was "I1" and the functional event "G" is in the state failure and to "0.002", otherwise. This goal is achieved by means of an "if-then-else" construct and the "test-initiating-event" expression.

```
<if>
    <and>
        <test-initiating-event name="I1"/>
        <test-functional-event name="G" state="failure"/>
    </and>
    <block>
        <set-parameter name="lambda1"> <float value="0.001"/> </set-parameter>
        <set-parameter name="lambda2"> <float value="0.001"/> </set-parameter>
    </block>
    <block>
        <set-parameter name="lambda1"> <float value="0.002"/> </set-parameter>
        <set-parameter name="lambda2"> <float value="0.002"/> </set-parameter>
    </block>
</if>
```

Finally, we could imagine that the sequence S1 is linked to an event tree ET2 if the initiating event was I1 and to another event tree ET3, otherwise. The definition of the sequence S1 would be as follows.

```
<define-sequence name="S1">
    <if>
        <test-initiating-event name="I1"/>
        <event-tree name="ET2"/>
        <event-tree name="ET3"/>
    </if>
</define-sequence>
```

CHAPTER

8

ORGANIZATION OF A MODEL

This chapter discusses the organizations of models. It includes the definition of two additional constructs: the notions of consequence, consequence group, and alignment.

8.1 Additional Constructs

8.1.1 Consequences and Consequence Groups

It is often convenient to group sequences of event trees into bins of sequences with similar physical consequences (e.g., Core Melt). The Model Exchange Format provides the notion of consequence to do so. A consequence is characterized by an event tree, a particular initiating event for this event tree, and a particular sequence (end-state) of the same tree. Consequences are given a name. Groups of consequences can be defined as well. They are also given a name, and can include sub-groups. The RNC schema for the XML representation of declarations of groups of consequences is given in Listing 8.1.

Listing 8.1: The RNC schema of the XML representation of consequence groups

```
consequence-definition =
  element define-consequence {
    name,
    label?,
    attributes?,
    element initiating-event { name },
    element sequence { name }
  }

consequence-group-definition =
```

```
  element define-consequence-group {
    name, label?, attributes?, (consequence | consequence-group)*
  }

consequence = element consequence { name }

consequence-group = element consequence-group { name }
```

Note that consequences and consequence groups can be used as initiating events (see Section 7.2.2). This mechanism makes it possible to link event trees.

8.1.2 Missions, Phases

Phases are physical configurations (e.g., operation and maintenance) in which the plant spends a fraction of the mission time. Phases are grouped into missions. The time fractions of the phases of a mission should sum to 1. House events and parameters may be given different values in each phase. The RNC schema for the XML representation of phase declarations is given in Listing 8.2.

Listing 8.2: The RNC schema of the XML representation of Missions and Phases

```
alignment-definition =
  element define-alignment {
    name, label?, attributes?, phase-definition+
  }

phase-definition =
  element define-phase {
    name,
    attribute time-fraction { xsd:double },
    label?,
    attributes?,
    instruction*
  }
```

8.2 Splitting the Model into Several Files

So far, we have written as if the model fits completely into a single file. For even medium size PSA models this assumption not compatible with Quality Control. Moreover, such a monolithic organization of data would be very hard to manage when several persons work together on the same model.

A first way to split the model into several files is to use the XML notion of entities: in any XML file, it is possible to declare file entities in the preamble, and to include them in the body of the document. This mechanism is exemplified below.

```
<?xml version="1.0" ?>

<!DOCTYPE SMRF [
<!ENTITY file1 SYSTEM "file1.xml">
<!ENTITY file2 SYSTEM "file2.xml">
]>
<smrf>
    ...
    &file1;
    ...
    &file2;
    ...
</smrf>
```

This mechanism, however, has the drawback that XML tools have to actually include the files into the document, hence, making its manipulation heavier.

The Model Exchange Format proposes another simple mechanism to achieve the same goal: the tag include. This tag can be inserted at any place in a document. Its effect is to load the content of the given file into the model.

```
<opsa-mef>
    ...
    <include file="basic-events.xml"/>
    ...
</opsa-mef>
```

8.3 Organization of a Model

The Model Exchange Format introduces five types of containers: models at the top level, event trees, fault trees, components, and model-data. The Model Exchange Format introduces also eighteen constructs. Fig. 8.1 shows the containers and the constructs they can define.

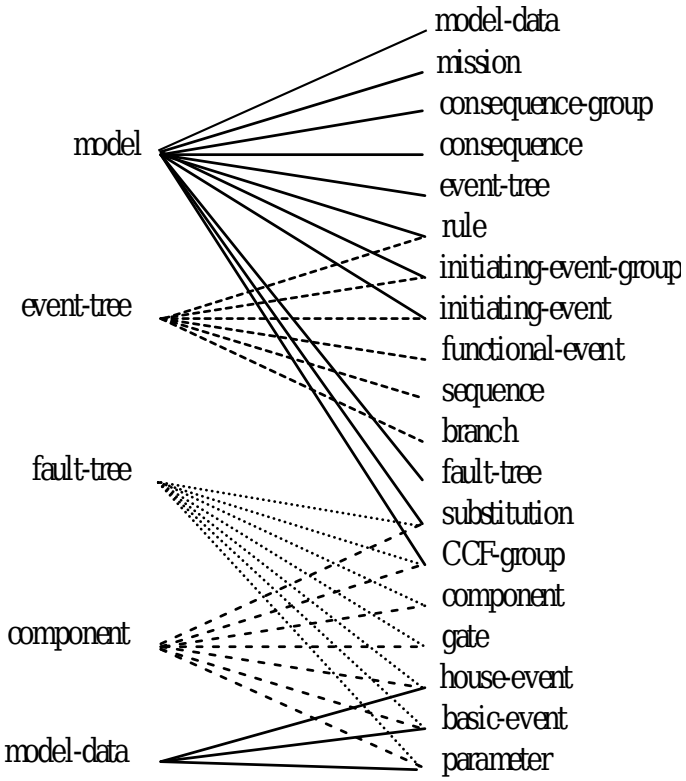


Fig. 8.1: Containers and the constructs they can define

Listing 8.3 gives the RNC schema of the XML representation of a model.

Listing 8.3: The RNC schema for the XML representation of a model

```
model =
  element opsa-mef {
    name?,
    label?,
    attributes?,
    (model-data
     |event-tree-definition
     | alignment-definition
     | consequence-group-definition
     | consequence-definition
     | rule-definition
     | initiating-event-group-definition
     | initiating-event-definition
     | fault-tree-definition
     | substitution-definition
     | CCF-group-definition
     | include-directive)*
  }
```

CHAPTER

9

REPORT LAYER

9.1 Preliminary Discussion

The report layer is populated with constructs to save results of calculations. These constructs fall into two categories:

- Constructs to tell which software, algorithm(s), and option(s) were used to produce the results
- The results themselves

It is almost impossible and probably not even desirable to normalize fully the report layer. Tools are very different from one another and produce a wide variety of results. New calculation methods are regularly proposed. To normalize everything would lead to a huge and anyway incomplete format. Moreover, the way results are arranged into reports depends on the study. It is also, at least to some extent, a matter of taste.

If the Model Exchange Format cannot give a formal structure for the report layer, it can at least suggest a style to describe what has been calculated and how it has been calculated. It can also provide a check-list of what should be included as information to make results truly exportable and importable. The existence of such report style would be very useful for reporting tools (whether they are graphical or textual): it would be much easier for these tools to extract the information they need from the XML result files.

9.2 Information about calculations

Here follows a non-exhaustive list of information about how the results have been obtained that can be relevant and other special or unique features of the model.

- Software
  - Version
  - Contact organization (editor, vendor, etc.)
  - ...
- Calculated quantities
  - Name
  - Mathematical definition
  - Approximations used
  - ...
- Calculation method(s)
  - Name
  - Limits (e.g., number of basic events, of sequences, of cut sets)
  - Preprocessing techniques (modularization, rewritings, etc.)
  - Handling of success terms
  - Cutoffs, if any (absolute, relative, dynamic, etc.)
  - Are delete terms, recovery rules or exchange events applied?
  - Extra-logical methods used
  - Secondary software necessary
  - Warning and caveats
  - Calculation time
  - ...
- Features of the model
  - Name
  - Number of: gates, basic events, house events, fault trees, event trees, functional events, initiating events
- Feedback
  - Success or failure reports
  - ...

9.3 Format of Results

PSA tools produce many kinds of results. Some are common to most of the tools, e.g., probability/frequency of some group of consequences, importance factors, sensitivity analyses. They fall into different categories. The following three categories are so frequent that is it worth to normalize the way they are stored into XML files.

- Minimal cut sets (and prime implicants)
- Statistical measures (with moments)
- Curves

9.3.1 Minimal Cut Sets

A first (and good) way to encode minimal cut sets consists in using the representation of formulae defined by the Model Exchange Format. However, it is often convenient to attach some information to each product, which is not possible with the formulae of the Model Exchange Format. An alternative XML representation for sums of products (sets of minimal cut sets are a specific type of sums of products) is given in Listing 9.1. More attributes can be added to tags "sum-of-products" and "product" to carry the relevant information.

Listing 9.1: The RNC schema for the XML representation of sums-of-products

```
sum-of-products =
  element sum-of-products {
    name?,
    attribute description { text }?,
    attribute basic-events { xsd:nonNegativeInteger }?,
    attribute products { xsd:nonNegativeInteger }?,
    product*
  }

product =
  element product {
    attribute order { xsd:positiveInteger }?,
    literal*
  }

literal =
  element basic-event { name }
  | element not { element basic-event { name } }
```

9.3.2 Statistical measures

Statistical measures are typically produced by sensitivity analyses. They are the result, in general, of Monte-Carlo simulations on the values of some parameter. Such a measure

can come with moments (mean, standard deviation), confidence ranges, error factors, quantiles, etc. The XML representation for statistical measure is given in Listing 9.2.

Listing 9.2: The RNC schema for the XML representation of statistical measures

```
statistical-measure =
  element measure {
    name?,
    attribute description { text }?,
    element mean {
      attribute value { xsd:double }
    }?,
    element standard-deviation {
      attribute value { xsd:double }
    }?,
    element confidence-range {
      attribute percentage {
        xsd:double { minExclusive = "0" maxExclusive = "100" }
      },
      attribute lower-bound { xsd:double },
      attribute upper-bound { xsd:double }
    }?,
    element error-factor {
      attribute percentage {
        xsd:double { minExclusive = "0" maxExclusive = "100" }
      },
      attribute value {
        xsd:double { minExclusive = "0" }
      }
    }?,
    quantiles?,
  }

quantiles =
  element quantiles {
    attribute number { xsd:positiveInteger },
    quantile+
  }

quantile = element quantile { bin-data }

bin-data =
  attribute number { xsd:positiveInteger },
  attribute mean { xsd:double }?,
  attribute lower-bound { xsd:double }?,
  attribute upper-bound { xsd:double }?
```

9.3.3 Curves

Two or three dimensional curves are often produced in PSA studies. A typical example is indeed to study the evolution of the system unavailability through the time. The XML representation of curves suggested by the Model Exchange Format is given in Listing 9.3.

Listing 9.3: The RNC schema the XML representation of curves

```
curve =
  element curve {
    attribute name { xsd:NCName }?,
    attribute description { text }?,
    (attribute X-title { xsd:string },
     attribute Y-title { xsd:string },
     attribute Z-title { xsd:string }?)?,
    (attribute X-unit { unit },
     attribute Y-unit { unit },
     attribute Z-unit { unit }?)?,
    element point {
      attribute X { xsd:double },
      attribute Y { xsd:double },
      attribute Z { xsd:double }?
    }*
  }

unit = "seconds" | "hours"
```

CHAPTER

10

BIBLIOGRAPHY

10.1 Basic PSA References

1. ASME RA-S-2002, "Standard for Probabilistic Risk Assessment for Nuclear Power Plant Applications", The American Society of Mechanical Engineers, 2002.
2. Roberts N. H., W. E. Vesely, D. F. Haasl, F. F. Goldberg, Fault Tree Handbook, NUREG-0492, US NRC, Washington, 1981.
3. W. Vesely, J. Dugan, J. Fragola, J. Minarick, J. Railsback, Fault Tree Handbook with Aerospace Applications, National Aeronautics and Space Administration, NASA, 2002
4. Regulatory Guide 1200, An Approach for Determining the Technical Adequacy of Probabilistic Risk Assessment Results for Risk-Informed Activities, US NRC, 2004.
5. US NRC Regulatory Guide 1.174 "An Approach for Using Probabilistic Risk Assessment in Risk-Informed Decisions on Plant-Specific Changes to the Licensing Basis", Revision 1, US NRC, 2002.

10.2 Difficulties with PSA

1. Čepin M., Analysis of Truncation Limit in Probabilistic Safety Assessment, Reliability Engineering and System Safety, 2005, Vol. 87 (3), pp. 395-403.
2. S. Epstein and A. Rauzy, Can we trust PRA?, Reliability Engineering & System Safety, Volume 88, Issue 3, June 2005, Pages 195-205

10.3 Novel Approaches

1. Antoine Rauzy, New algorithms for fault trees analysis, Reliability Engineering & System Safety, Volume 40, Issue 3, 1993, Pages 203-211

2. 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, Volume 58, Issue 2, November 1997, Pages 127-144

3. Poul Frederick Williams, Macha Nikolskaïa and Antoine Rauzy, Bypassing BDD construction for reliability analysis, Information Processing Letters, Volume 75, Issues 1-2, 31 July 2000, Pages 85-89

4. Y. Dutuit and A. Rauzy, Approximate estimation of system reliability via fault trees, Reliability Engineering & System Safety, Volume 87, Issue 2, February 2005, Pages 163-172

5. Čepin M. and B. Mavko, A Dynamic Fault Tree, Reliability Engineering and System Safety, 2002, Vol. 75, No. 1, pp. 83-91.

6. Albert F. Myers and Antoine Rauzy, Assessment of redundant systems with imperfect coverage by means of binary decision diagrams, Reliability Engineering & System Safety, Volume 93, Issue 7, July 2008, Pages 1025-1035