Probabilistic Safety Analysis with XFTA

Antoine B. Rauzy

PUBLISHED BY THE ALTARICA ASSOCIATION
ALTARICA-ASSOCIATION.ORG

First printing, October 2020
ISBN: 978-82-692273-0-7

To Woody

Les Oiseaux de Passage

Oh ! vie heureuse des bourgeois ! Qu'avril bourgeonne
Ou que décembre gèle, ils sont fiers et contents.
Ce pigeon est aimé trois jours par sa pigeonne ;
Ça lui suffit, il sait que l'amour n'a qu'un temps.

Ce dindon a toujours béni sa destinée.
Et quand vient le moment de mourir il faut voir
Cette jeune oie en pleurs : "C'est là que je suis née ;
Je meurs près de ma mère et je fais mon devoir".

Elle a fait son devoir ! C'est à dire que oncques
Elle n'eut de souhait impossible, elle n'eut
Aucun rêve de lune, aucun désir de jonque
L'emportant sans rameurs sur un fleuve inconnu.

Et tous sont ainsi faits ! Vivre la même vie
Toujours pour ces gens là cela n'est point hideux.
Ce canard n'a qu'un bec, et n'eut jamais envie
Ou de n'en plus avoir ou bien d'en avoir deux.

Ils n'ont aucun besoin de baiser sur les lèvres,
Et loin des songes vains, loin des soucis cuisants,
Possèdent pour tout cœur un vicère sans fièvre
Un coucou régulier et garanti dix ans !

Oh ! les gens bienheureux!. . . Tout à coup, dans l'espace
Si haut qu'il semble aller lentement, un grand vol
En forme de triangle arrive, plane, et passe.
Où vont ils ? Qui sont-ils ? Comme ils sont loins du sol !

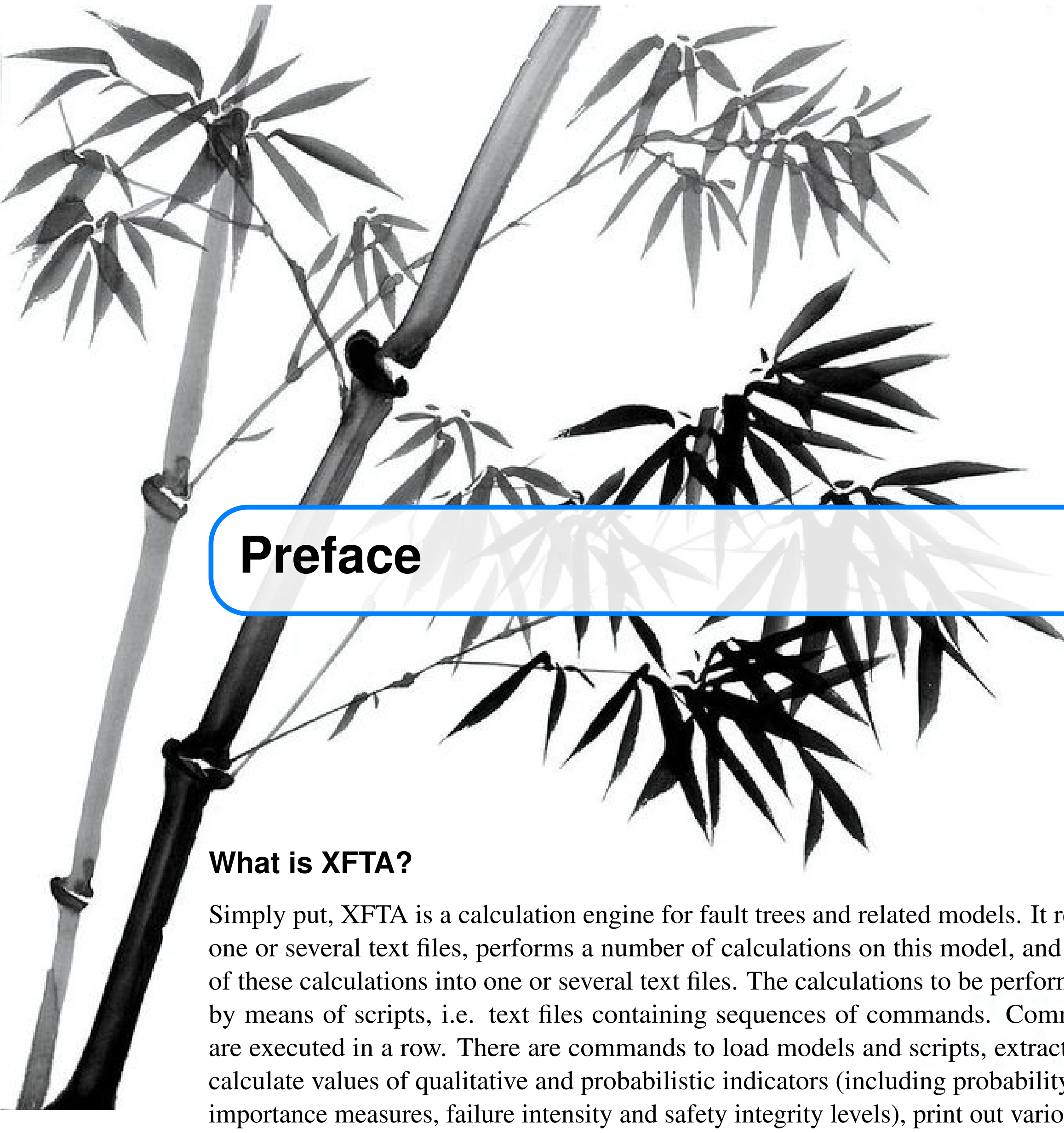
Regardez les passer ! Eux, ce sont les sauvages.
Ils vont où leur desir le veut, par-dessus monts
Et bois, et mers, et vents, et loin des esclavages.
L'air qu'ils boivent ferait éclater vos poumons.

Regardez-les ! Avant d'atteindre sa chimère,
Plus d'un, l'aile rompue et du sang plein les yeux,
Mourra. Ces pauvres gens ont aussi femme et mère
Et savent les aimer aussi bien que vous, mieux.

Pour choyer cette femme et nourrir cette mère,
Ils pouvaient devenir volailles comme vous.
Mais ils sont avant tout les fils de la chimère,
Des assoiffés d'azur, des poètes, des fous.

Regardez-les, vieux coq, jeune oie édifiante !
Rien de vous ne pourra monter aussi haut qu'eux.
Et le peu qui viendra d'eux à vous, c'est leur fiente.
Les bourgeois sont troublés de voir passer les gueux.

Jean Richepin / Georges Brassens



Preface

What is XFTA?

Simply put, XFTA is a calculation engine for fault trees and related models. It reads a model into one or several text files, performs a number of calculations on this model, and prints out results of these calculations into one or several text files. The calculations to be performed are described by means of scripts, i.e. text files containing sequences of commands. Commands of a script are executed in a row. There are commands to load models and scripts, extract minimal cutsets, calculate values of qualitative and probabilistic indicators (including probability of the top-event, importance measures, failure intensity and safety integrity levels), print out various data, in a word to perform all operations involved in probabilistic safety analyses.

The syntax for equations that describe basic and intermediate events is quite intuitive. If you know what a fault tree is, you should have no problem to write down your models. The syntax for commands is slightly more specific. However, the same scripts are used over and over and this book provides the reader with scripts for the most common analyses. In a word, even a non experienced analyst should be able to make a simple use of XFTA quickly.

The above presentation is however quite reductive because XFTA can be used for much more advanced purposes.

First, XFTA provides a full-fledged object-oriented language to design models: S2ML+SBE. S2ML+SBE is the combination of S2ML, which stands for system structure modeling language and SBE, which stands for stochastic Boolean equations. Systems of stochastic Boolean equations are the underlying mathematical framework of fault trees and reliability block diagrams: any fault tree or reliability block diagram is eventually a system of stochastic Boolean equations. S2ML is a coherent and versatile set of object-oriented constructs that help to design and to structure models (Batteux, Prosvirnova, and Rauzy 2018; Rauzy and Haskins 2019). Indeed, you are not forced to use S2ML constructs if you do not want to, i.e. you can just write down systems of stochastic Boolean equations. Nevertheless, object-oriented constructs provide a significant additional expressive power and induce no additional cost, as S2ML+SBE models are automatically (and efficiently) translated into systems of stochastic Boolean equations.

S2ML+SBE is a textual modeling language. In addition to this textual format, XFTA supports the Open-PSA format, which is a XML grammar systems of stochastic Boolean equations. The original Open-PSA format (Epstein and Rauzy 2008) has been extended to support S2ML constructs. The two formats, S2ML+SBE and (extended) Open-PSA are thus equivalent.

Second, XFTA implements state-of-the-art algorithms and data structures to assess fault trees. There are actually two main approaches to assess fault trees. The first one consists in extracting minimal cutsets of the top event, then in performing calculations of probabilistic indicators from these minimal cutsets. The second one consists in building a binary decision diagram encoding the structure function of the top event, then in performing calculations of probabilistic indicators from this binary decision diagram. In this second approach, minimal cutsets can also be extracted, but starting from the binary decision diagram rather than from the model itself.

As we shall explain, both approaches have their advantages and drawbacks. This is the reason why XFTA implements them both.

The XFTA Project

I started developing XFTA in 2012, right after I came back to academia after a few years spent in industry. My motivations were twofold.

First, as a computer scientist, I wanted to explore the possibilities of top-down, i.e. MOCUS like, algorithms to extract minimal cutsets. I introduced the binary decision diagram technology in the reliability engineering field at the beginning of the nineties. My first article on the topic has been published in 1993 (Rauzy 1993). For almost two decades, alone or in collaboration with Yves Dutuit and Jean-Pierre Signoret, I did a lot of research and development to adjust this technology, originally developed for the validation of digital circuits, to the particular context of fault tree analysis. This work has been successful in two ways: First, binary decision diagram algorithms eventually outperform all other available technologies on models with up to several hundreds of basic events and gates. Second, they made it possible to revisit completely the mathematical foundations of fault tree analysis, which needed actually to be brushed up. The reader interested in this topic can refer to my 2008 survey (Rauzy 2008a), which summarizes the key ideas and gives pointers to the literature.

Nevertheless, despite of my efforts, I have not been able to make the binary decision diagram technology work fully on very large models developed in the framework of nuclear probabilistic safety analyses. These models involve up to several thousands of basic events and gates. Algorithms able to assess them are essentially model pruners: their objective is not to assess the model fully but rather to extract as efficiently as possible the most relevant scenarios of failure, i.e. the most probable minimal cutsets, leaving aside all others. It is not rare that only a small proportion of the basic events of the model show up in the extracted minimal cutsets (Epstein and Rauzy 2005; Epstein, Rauzy, and Wakefield 2006; Epstein, Wakefield, and Rauzy 2002). It remains that these cutsets concentrate hopefully the risk, and for this very reason focus the attention of analysts.

The minimal cutsets extraction algorithm implemented in XFTA does this pruning job, and does it very efficiently. This algorithm is remotely inspired from the MOCUS method (Fussel and Vesely 1972). Technically however, it uses principles and techniques stemmed from modern SAT solvers (Davis, Logemann, and Loveland 1962; Kroening and Strichman 2017). For this reason, I call it a *branch-and-test* algorithm. At the time I am writing these lines, XFTA is the probably the most efficient available calculation engine, academic and commercial tools included. It makes it possible to assess models that are far beyond the reach of most of the other tools.

The second category of reasons that pushed me to develop XFTA was the experience I acquired over the years on probabilistic risk and safety assessment models as a safety analyst and I would say as a citizen. Working on nuclear models all over the world, my buddy Steven "Woody" Epstein—to the memory of whom this book is dedicated—and I realized how much the deployment of probabilistic risk and safety analyses is hampered by the economic conditions in which models are designed. The market for fault tree (and related formalisms) assessment tools is a niche market. Although these tools are essential in the process of assessing the reliability of virtually all technical systems presenting risks for themselves, their operators and their environment, including nuclear power plant, chemical and oil and gas plant, airplanes, trains, cars and many others, only very limited budgets are dedicated to their development and their maintenance. As a consequence, they are what they are and do not evolve much over the years. In particular, each and every commercial tool has its own format for models, making it extremely difficult to transfer a model designed with one tool into another tool. This, in turn, makes nearly impossible cross-verification of results, quality assurance on models, peer reviews, and more generally new ideas to enter into the field.

Woody and I decided thus to launch the Open-PSA initiative, an open and transparent public forum to disseminate information, independently review new ideas, and spread the word. We focused on the design of a model exchange format that would allow the portability of models and methods. We have been successful at that, at least technically: the format exists (Epstein and Rauzy 2008) and has been validated via multiple experiments (Epstein, Nusbaumer, et al. 2007; Epstein, Reinhart, and Rauzy 2008a; Epstein, Reinhart, and Rauzy 2008b; Epstein, Reinhart, and Rauzy 2010a; Epstein, Reinhart, and Rauzy 2010b; Hibti, Friedlhuber, and Rauzy 2012a; Hibti, Friedlhuber, and Rauzy 2012b).

I decided to illustrate the possibilities offered by the Open-PSA format by developing a new calculation engine, that would be freely available for anybody interested in the subject, individuals, students, professors, academic researchers, tools developers, risk and safety analysts, regulators. . . Hence XFTA.

I completed the first version version of the tool after a couple of years. The version 1.3.1 has been ported on several platforms (Windows, Unix and Mac). Portability is not really an issue for XFTA, although porting a software is never easy, as it is developed in very standard C++.

Then the XFTA project stayed asleep for a (rather long) while. The Open-PSA format, despite its interest and adoption by a few tool makers has not received the attention it should have. The reason is probably that no one, including regulatory bodies, shows much will to have a public debate on models and methods, taking the political risk of opening the Pandora box of their quality. Moreover, the academic world being what it is (or more exactly what it became during the last one or two decades), developing software is not a rewarded activity, to say the least. But a few persons believed in the project. Among them, Emmanuel Clément who, aside of his work as an engineer, developed Arbre Analyste, a graphical user interface for XFTA (Clément, Thomas, and Rauzy 2014). As XFTA, Arbre Analyste is free of use. As of today, Arbre Analyste has been downloaded an impressive number of times. Emmanuel convinced me to go back to work. So back I am, with a bunch of ideas to make both the Open-PSA format and XFTA evolve.

These ideas go into two main directions.

First, add, as input language of XFTA, a full-fledged object-oriented language. In other words, to make fault tree design benefit of the advances realized the last decade on model-based safety assessment, notably by the design of the AltaRica 3.0 modeling language (Batteux, Prosvirnova, and Rauzy 2019) and of the S2ML+X paradigm (Batteux, Prosvirnova, and Rauzy 2018; Rauzy and Haskins 2019). This new language is called S2ML+SBE because it combines, as already said, S2ML and with (systems of) stochastic Boolean equations (hence SBE).

Second, improve the calculation of probabilistic risk indicators, which was the initial demand of Emmanuel Clément. With that respect, the objective is to embed, in addition to the branch-and-test algorithm to extract minimal cutsets, an another algorithmic approach relying on the binary decision diagram technology (Rauzy 2008a). Moreover, I found ways to improve the calculations performed on minimal cutsets, solving at least some of the problems encountered so far, especially when the probability of basic events are high.

The new versions of XFTA (numbered from 2.0.0) presented in this book improves thus very significantly the previous versions (numbered from 1.3.0).

Frequently Asked Questions

Here follows some more or less short answers to frequently asked questions about the Open-PSA format and XFTA.

What is the Open-PSA format? The first versions of the Open-PSA format (until version 3) came as a XML grammar to describe fault trees and event trees. A model written in this format is not really human friendly. On the other hand, the objective of the format is not to be read by human, but to be easily parsed by computer programs. A XSD schema that describes the grammar is available).

The current version of XFTA accepts two input languages: Open-PSA format extended with S2ML constructs and S2ML+SBE. The two languages are equivalent and the extended version of Open-PSA is compatible with the previous ones, up to some minor exceptions. S2ML+SBE clarifies, simplifies and generalizes a number of concepts of the Open-PSA format.

Can any one use the Open-PSA format and S2ML+SBE? Yes. Definitely yes. The Open-PSA format and S2ML+SBE are public and can be used as such or modified by anybody, for any purpose. The documents describing them are distributed under the Creative Commons Attribution (CC BY 4.0) license.

What does XFTA stand for? FTA stands for indeed for fault tree analysis. The X comes from the fact that I started the development of the tool when I was professor at École Polytechnique commonly called in French "*l'X*".

Can any one use XFTA? Yes. Definitely yes. XFTA is freely available. Anybody can use it, free of charge, including for commercial purposes. Indeed, XFTA is distributed as is, in the hope it will be useful, but without any warranty, even the implied warranty of merchantability or fitness for a particular purpose. The only condition put by the license is that if you embed XFTA in your own tool, first you must make clear in the documentation of your tool that it is embedding XFTA, second you should not attempt to decompile XFTA, disassemble it or otherwise attempt to discover its source code.

Who owns XFTA? Since 2014, I hosted the intellectual property of XFTA in the non for profit AltaRica Association (concretely an association declared under the status of French "*loi de 1901*"). The main goal of the AltaRica Association is to promote model-based approaches in systems engineering in general and in reliability engineering in particular.

Having the XFTA intellectual property hosted in a legal entity makes it possible to establish formal contracts (for its operational maintenance, its development or its use) with public or private institutions, with only one proviso: these contracts must aim at developing the scientific and technical knowledge about probabilistic safety analyses and at serving the common good. Moreover, their results must be made public.

How big is XFTA? In total, XFTA represents slightly over 100,000 lines of C++ code. About forty percent of these lines consist in a library of reusable classes implementing containers, XML management tools and similar things. The other sixty percent consist in XFTA itself. From a software engineering perspective, XFTA is a medium size project. Not a big one of course, but not a small one neither, given that it is a concentrate of sophisticated algorithms.

Why XFTA is not open-source? Open-source has two major interests. First, it makes it possible for people all over the world to contribute to the development (and the debugging) of the software. Second, it ensures that the institutions (companies, universities or simply individuals) using the software can maintain it—typically by recompiling it on new platforms and/or with upgraded compilers—in case its main developer(s) are called by other duties.

Regarding the first interest, I do not think this applies to software such as XFTA. As already said, XFTA is a dense concentrate of sophisticated algorithms with tight interactions. Without my help, I sincerely doubt that anybody can really enter into the code, and even less to modify it without risking to create unexpected and undesired side-effects. In that, I share fully the views of a Peter Naur claiming that programming is theory building (Naur 1985).

In my humble opinion, "crowd-coding" applies only to software platforms consisting of a kernel designed by a small and highly coordinated team of developers, surrounded by applications that are highly independent from the kernel and from one another. The Linux operating system offers a paradigmatic example of such a platform. XFTA simply does not belong to this category of software. This said, the Open-PSA format (and to a lower extent S2ML+SBE) aims precisely at being the kernel of fault tree (and related formalisms) applications.

In contrast, the second interest of open-source applies fully to XFTA. However, I never got any support from any institution (public or private) to develop XFTA, not to speak about a reward. Making it open-source would mean taking a huge risk of being looted. I simply do not believe open-source licenses would prevent anybody to copy-paste the source code, modify it marginally and pretend that it is a new software. The AltaRica Association is a much too light structure to be able to defend itself and myself against such a practice.

The time I am spending in the development of XFTA, which counts in thousands of hours, is a time I cannot spend in writing articles or submitting project proposals. As of today, it is not rewarded, nor even recognized by academic authorities. That's life. I have no other choice than to accept this situation. But at least, I want to protect my intellectual property.

In summary, not distributing XFTA under an open-source license is by no means a matter of principle. I just do not want to give up the hope of getting some support to develop and maintain it. I wish that the development of tools like XFTA, which contribute to the common good, can be supported collectively, "from each according to his ability; to each according to his needs". I wish the conditions for XFTA to be distributed open-source will be eventually fulfilled. The sooner, the better.

Is there any warranty that XFTA will be maintained in the future? No, for the reasons indicated above. This said, also for the reasons indicated above, I wish a solution will be found in the near future to support the maintenance in operational conditions and the development of XFTA. I know that engineers in companies have really hard time to make their management understand that supporting the development of a software without being the owners of this software can be eventually beneficial for their company. But I do not want to give up.

Is XFTA certified? No. In fact, none of the tools used to support probabilistic risk and safety analyses is. They are "qualified by experience", whatever it means. Of course, I am testing extensively XFTA before any new release. I keep also track of bugs. The models I am using to test XFTA are both models coming from industry and "artificial" models designed to test this or that feature. This set of models as well as the scripts used to assess them could be easily turned into a more formal qualification kit, itself publicly released.

Acknowlegments

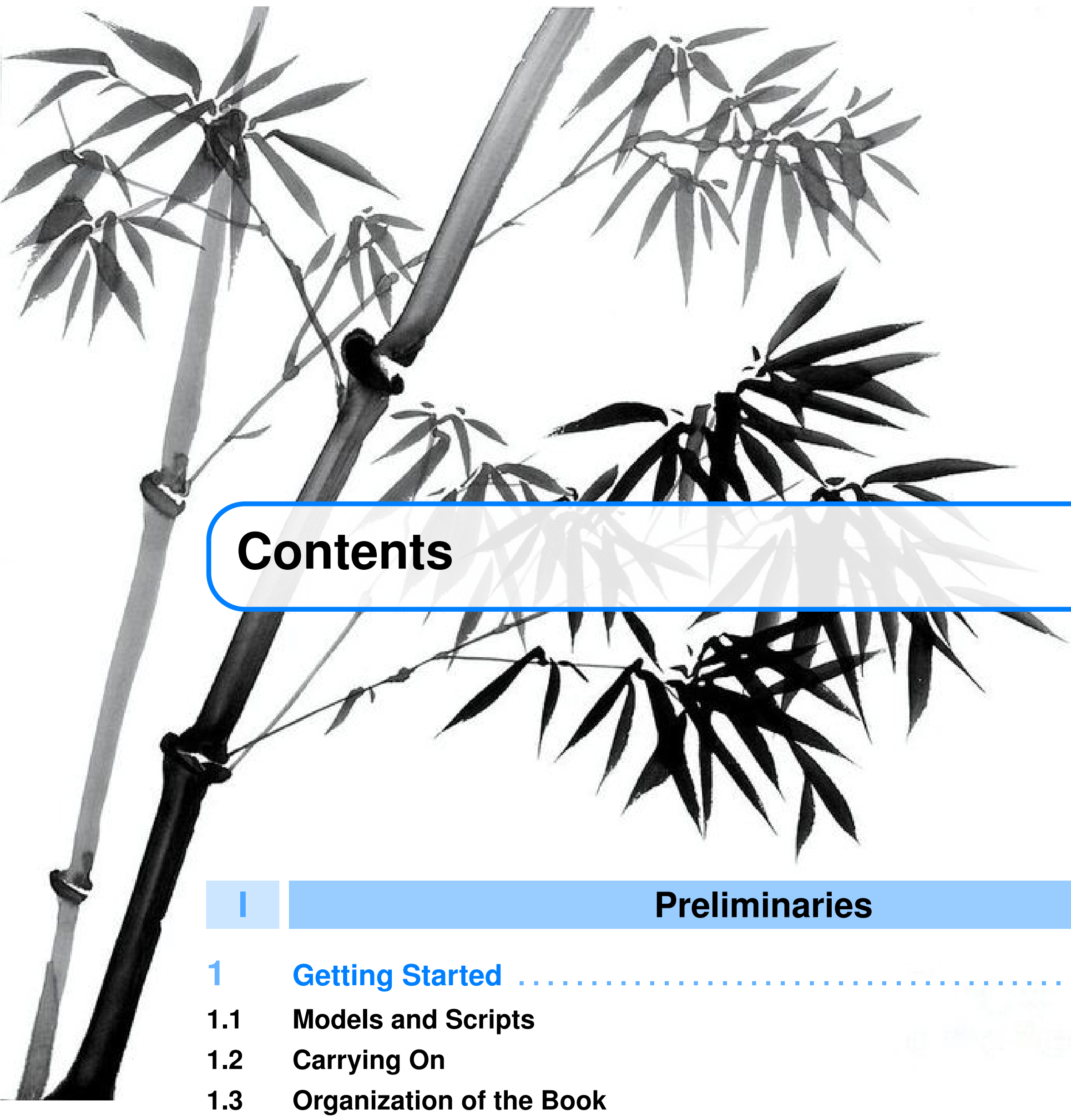
I would like to thank here again people who played an important role in the development of XFTA.

My old friends Yves Dutuit and Jean-Pierre Signoret who made me discover reliability engineering ("la sûreté de fonctionnement" as we use to say in French) and so many other things. Their help and support went far beyond professional matters. I keep in my mind and heart the happy years of our intense collaboration.

Steven "Woody" Epstein who left us on February the 5th 2019, to go cooking, drinking wine and doing mathematics in another world. I miss him a lot. Without Woody, there would have been no Open-PSA initiative and eventually no XFTA.

Last but not least, Michel Batteux with whom I am collaborating now for more than 10 years. Michel ported XFTA on Linux and Mac OS.

Antoine Rauzy
Trondheim, October 2020

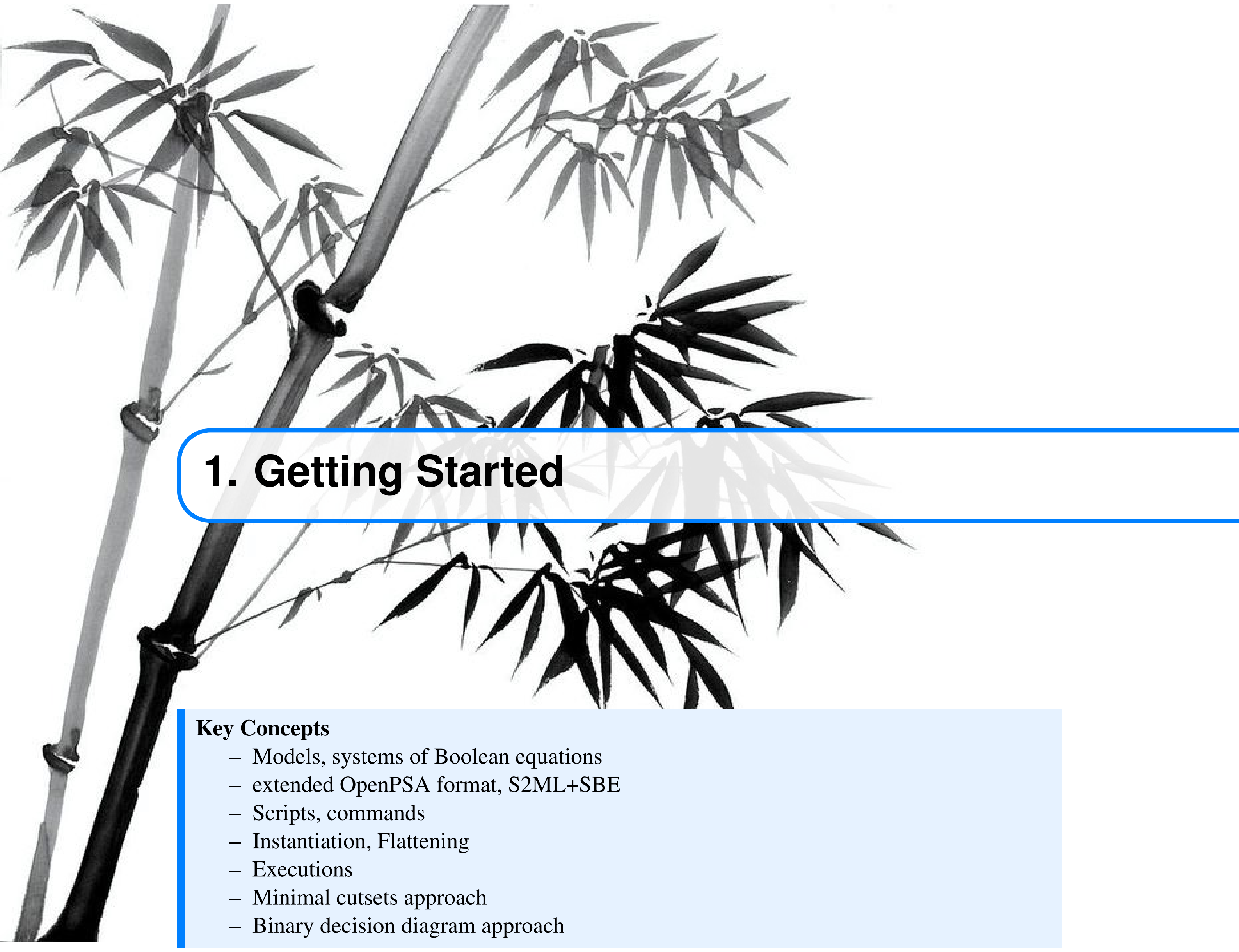


Contents

IV Appendix

I

Preliminaries



1. Getting Started

Key Concepts
- Models, systems of Boolean equations
- extended OpenPSA format, S2ML+SBE
- Scripts, commands
- Instantiation, Flattening
- Executions
- Minimal cutsets approach
- Binary decision diagram approach

1.1 Models and Scripts

1.1.1 Models

XFTA a calculation engine for fault trees, or more exactly for *systems of stochastic Boolean equations* that are the underlying mathematical framework of fault trees and reliability block diagrams. A typical XFTA session consists thus in loading a model, performing some calculations on this model and printing out results of these calculations into one or more files. All the files that XFTA inputs and outputs are text files, i.e. they can be read and write by simple text editors[1].

Let us start with models. Consider the fault tree pictured in Figure 1.1.

This fault tree is made of a top event `top`, three intermediate events `g1`, `g2` and `g3` and five basic events `A`, `B`, `C`, `D` and `E`. Events are actually Boolean variables.

Assume that `A`, `B`, `C` `D`, and `E` are associated respectively with probabilities 0.001, 0.03, 0.05, 0.04 and 0.0009.

XFTA supports two input languages for models: *S2ML+SBE*, which is a textual language, and the (extended) *Open-PSA format*, which is the XML equivalent to S2ML+SBE. S2ML+SBE is easy to read for human, conversely to the Open-PSA format. For computers, it is quite the contrary. This

[1] My favorite text editor, under Windows®, is Notepad++. It has many convenient features, including the possibility to define your own syntax highlighting.

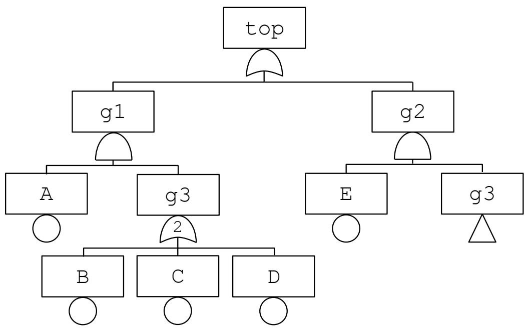


Figure 1.1: A fault tree

```
gate top = g1 or g2;
gate g1 = A and g3;
gate g2 = E and g3;
gate g3 = atleast 2 (B, C, D);
basic-event A = 0.001;
basic-event B = 0.03;
basic-event C = 0.05;
basic-event D = 0.04;
basic-event E = 0.0009;
```

Figure 1.2: S2ML+SBE code for the fault tree pictured in Figure 1.1

is the reason why XFTA supports both format. As I guess that the reader is a human, the models given in the remainder of this chapter will be written in S2ML+SBE.

The system of stochastic Boolean equations (written in S2ML+SBE) encoding the fault tree pictured in Figure 1.1 is given in Figure 1.2.

This code is rather straightforward. It consists in nine equations, one per variable of the model. The equations describing intermediate events are prefixed with the keyword `gate` while equations defining basic events are prefixed with the keyword `basic-event`. The left member of the equation is the variable that is defined by the equation and the right member its definition. The definition is a Boolean formula in the case of intermediate events, and a probability, or more exactly a probability distribution, in the case basic events.

Note that the intermediate event `g3` is defined by a 2-out-of-3 gate, i.e. it is realized when at least 2 out of `B`, `C`, and `D` are.

The model given in Figure 1.2 must be saved in a text file, e.g. "`FaultTree001.sbe`". Although this is not required by the tool, we use the extension "`.sbe`" for files containing S2ML+SBE models.

1.1.2 Scripts

To load this into XFTA and to perform calculations on it, we need to write a script. As for models, scripts are written into text files. Although this is not required by the tool, we use the extension "`.xfta`" for XFTA script files. The script to assess the model recorded into the file "`FaultTree001.sbe`" could be recorded, for instance, into the file "`Script001.xfta`".

Figure 1.3 shows a script to assess our model. This script consists of five commands, one per line. Note that all commands terminate with a semicolon "`;`", which makes it possible for a command to spread over several lines.

The first command simply loads the model, which is recorded into the file `FaultTree001.sbe`.

The second command builds the target model. As S2ML+SBE is an object-oriented language,

```
load model "FaultTree001.sbe";
build target-model;
build BDT top;
compute probability top output="prb.tsv";
print minimal-cutsets top output="mcs.tsv";
```

Figure 1.3: A script to assess the model given in Figure 1.2

the first step of any assessment consists in transforming the source model into a mathematically equivalent system of stochastic Boolean equations, which is itself a S2ML+SBE model in which all object-oriented constructs have been resolved. The command `build target-model` performs this transformation, which consists in two steps called respectively *instantiation* and *flattening*.

The third command extracts minimal cutsets of the top event `top`. These minimal cutsets are stored into an internal data structure, a binary decision tree (hence the name `BDT-MCS`), waiting for being processed or printed out.

The fourth command computes the probability of the top event and saves the result into the file "`prb.tsv`". `tsv` stands for *tabulation separated values*. TSV files are text files containing data that can be loaded into spreadsheet tools such as Excel®. Most of the results of calculations performed by XFTA are stored into TSV files, so to facilitate their post-processing by external tools. In our case, after the execution of the script, the file "`prb.tsv`" contains the two following lines:

```
top-event mission-time probability
top 0 8.80595e-06
```

The probability of top event `top` (calculated at time 0) is thus 8.81×10^{-6}.

The fifth and last command prints out the minimal cutsets into the TSV file "`mcs.tsv`". After the execution of the script, the file "`mcs.tsv`" contains the following lines:

```
1  2e-06 0.223964 A  C  D
2  1.8e-06 0.201568 C D E
3  1.5e-06 0.167973 A B C
4  1.35e-06 0.151176 B  C  E
5  1.2e-06 0.134378 A B D
6  1.08e-06 0.120941 B  D  E
```

This file, as all TSV files, is organized as a matrix. Each row (line) gives a minimal cutset. The first three columns give respectively the rank of the minimal cutset (minimal cutsets are sorted in decreasing order of their probabilities), its probability and its contribution, i.e. its probability divided by the sum of the probabilities of the minimal cutsets. The remaining columns give the basic events of the minimal cutsets. As we shall see, it is possible to change this default way of printing out minimal cutsets, e.g. it is possible not to print ranks, probabilities and contributions.

1.1.3 Executions

Once XFTA installed (see Chapter 2 for the installation procedure), to run it, you need to open a command prompt and to change the current directory to the directory in which your model and script files are located (or at least from which they can be easily accessed). Then, it suffices to call XFTA with the name of the script as argument. On Windows®, the command could be as follows.

```
xftar.exe Script001.xfta
```

In our example, the execution of XFTA produces (or overwrites) the two files "`prb.tsv`" and

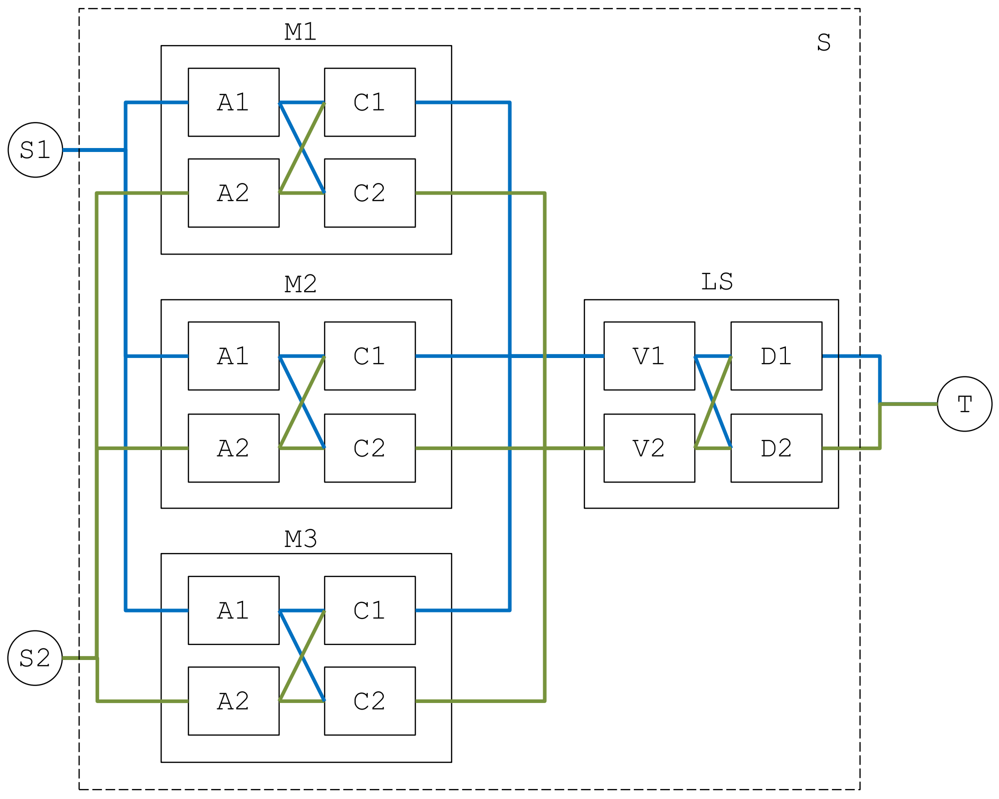


Figure 1.4: A tracking system

“mcs.tsv”.

1.2 Carrying On

1.2.1 Models

The model of the previous section was a very simple fault tree. Much larger and more complex models can be designed in S2ML+SBE: time dependent probability distributions rather than simple point estimates can be associated with basic events, extra-logical constructs such as common cause failure groups can be introduced, and of course, S2ML object-oriented constructs can be used. We shall not present all of these features here. It will take actually several chapters to describe them all.

Consider however the tracking system pictured in Figure 1.4.

This highly redundant system processes information coming from two redundant sources S1 and S2 (external to the system). The information coming from each source is acquired in triplicated acquisition modules M1, M2 and M3. Each acquisition block consists of two acquisition chains, one for each source. Each chain consists itself of an acquisition block Ai and a calculator Ci. Results of calculations are inputted into voters V1 and V2 working according to a 2-out-of-3 logic. Finally, the outputs of the two voters are aggregated into two calculators D1 and D2 that send the information to the target T (external to the system). Voters V1 and V2 and calculators D1 and D2 are part of the same logic solver LS.

This system is small. To study its reliability, it is thus possible to design a fault tree or a reliability block diagram “by hand”. It is however much better, for the sake of clarity, validity and maintainability of the model, to reflect the information we have on the system in the model.

A possible model for the tracking system is given in Figures 1.5, 1.6 and 1.7. This model can be seen as a hierarchical reliability block diagram.

```
class BasicBlock
   flow in = false;
   state failed = 0.1;
   flow out = in and not failed;
end

class AcquisitionBlock
   extends BasicBlock;
   state failed = exponential(lambda);
   parameter lambda = 1.23e-4;
end

class Calculator
   extends BasicBlock;
   flow in1 = false;
   flow in2 = false;
   flow in = in1 or in2;
   state failed = Weibull(alpha, beta, 0);
   parameter alpha = 5.67e+4;
   parameter beta = 3;
end

class Voter
   extends BasicBlock;
   flow in1 = false;
   flow in2 = false;
   flow in3 = false;
   flow in = atleast 2 (in1, in2, in3);
   state failed = exponential(lambda);
   parameter lambda = 2.64e-7;
end
```

Figure 1.5: Basic components of the model of the tracking system pictured in Figure 1.4

```
class AcquisitionModule
   AcquisitionBlock A1;
   AcquisitionBlock A2;
   Calculator C1;
   Calculator C2;
   flow C1.in1 = A1.out;
   flow C1.in2 = A2.out;
   flow C2.in1 = A1.out;
   flow C2.in2 = A2.out;
end

class LogicSolver
   Voter V1;
   Voter V2;
   Calculator D1
      parameter alpha = 3.29e+6;
      end
   Calculator D2
      parameter alpha = 3.29e+6;
      end
   flow D1.in1 = V1.out;
   flow D1.in2 = V2.out;
   flow D2.in1 = V1.out;
   flow D2.in2 = V2.out;
end
```

Figure 1.6: Parts of the model of the tracking system pictured in Figure 1.4

```
block TrackingSystem
   flow S1 = true;
   flow S2 = true;
   AcquisitionModule M1;
   AcquisitionModule M2;
   AcquisitionModule M3;
   LogicSolver LS;
   flow M1.A1.in = S1;
   flow M1.A2.in = S2;
   flow M2.A1.in = S1;
   flow M2.A2.in = S2;
   flow M3.A1.in = S1;
   flow M3.A2.in = S2;
   flow LS.V1.in1 = M1.C1.out;
   flow LS.V1.in2 = M2.C1.out;
   flow LS.V1.in3 = M3.C1.out;
   flow LS.V2.in1 = M1.C2.out;
   flow LS.V2.in2 = M2.C2.out;
   flow LS.V2.in3 = M3.C2.out;
   flow failed = not LS.D1.out and not LS.D2.out;
end
```

Figure 1.7: Main block of the model of the tracking system pictured in Figure 1.4

Without entering into too much details at this point, the code given in Figure 1.5 declares classes, i.e. reusable, on-the-shelf, modeling component, for each type of component involved in the system: acquisition blocks, calculators and voters. All these classes inherit from (are specialized versions of) the class `BasicBlock` that represents basic blocks of reliability block diagram.

State of components are represented by means of Boolean state variables, which are associated with probability distributions: exponential distributions for acquisition blocks and voters, Weibull distributions for calculators. These distributions take parameters, which can be modified when a component of the given type is created (instantiated, in the object-oriented jargon). State variables play thus the same role as basic events in fault trees. They are actually the same concept and the keywords `basic-event` and `state` are interchangeable.

Boolean flow variables are used to represent the flow of information, matter or energy circulating into the network of components and inside each component. Flow variables play the same role as intermediate events in fault tree. They are actually the same concept and the keywords `gate` and `flow` are interchangeable.

Figure 1.6 declares classes to represent parts of the tracking system, namely acquisition modules and the logic solver. These classes declares instances of the basic components and connect them, just as on the diagram given in Figure 1.4.

Finally, Figure 1.7 shows the main block of the model, i.e. the representation of the tracking system itself. As for parts, its declares components, here three acquisition modules and one logic solver and connects them.

This model gives a flavor of what can be done using object-oriented constructs. Models designed in way reflect the architecture of the system, they avoid tedious and error prone copy/paste operations and help to capitalized knowledge not only inside a model but from model to model.

1.2.2 Assessments

Except for trivial models, it is not possible to calculate the probability of the top event of fault tree directly from the system of stochastic Boolean equations. The formula associated with the top event has to be put in some normal form from which the probability can be computed, or at least estimated.

As of today, two normal forms have been proposed: sums of minimal cutsets and binary decision diagrams, which encode sums of disjoint products. XFTA provides commands to implement both approaches.

The first steps of the assessment of a model with XFTA are almost always the same, whether basic or advanced functionalities are used and whether the minimal cutsets or the binary decision diagram approach is chosen:

1. First, the model is loaded from one or several files.
2. Second, it is instantiated and flattened, i.e. transformed into a mathematically equivalent system of stochastic Boolean equations. Common cause failure models are expanded in this phase.
3. Third, the system of stochastic Boolean equations is normalized and optimized for a given top event, e.g. constants are propagated, connectives are coalesced or grouped and so on. These rewritings aim at achieving a better efficiency during the subsequent steps.

At this point, the two assessment processes diverge.

In the mimimal cutsets approaches, the subsequent steps are as follows.

4. Minimal cutsets of the top event are extracted, for a given cutoff (maximum order and minimum probability of cutsets). These minimal cutsets can be printed out into a text file.
5. Various probabilistic indicators are calculated from the minimal cutsets. Results are printed into one or more text files.

In the binary decision diagram approach, the subsequent steps are as follows.

```
load model "Models/TrackingSystem.sbe";
build target-model;
print target-model output="tgt.sbe";
build BDD TrackingSystem.failed;
compute probability TrackingSystem.failed
   mission-time=1000 output="prb.tsv";
build ZBDD-from-BDD TrackingSystem.failed;
print minimal-cutsets TrackingSystem.failed
   mission-time=1000 output="mcs.tsv";
```

Figure 1.8: A script to assess the model of the tracking system

4. The binary decision diagram encoding the top event is built.
5. Various probabilistic indicators are calculated from this binary decision diagram. Results are printed into one or more text files.
6. A second binary decision diagram, technically a zero-suppressed binary decision diagram, that encodes minimal custets is build from the binary decision diagram encoding the top event. Minimal cutsets encoded by the second diagram are printed out into a text file.

In both cases, the normalization step (step 3) is transparent for the user: it is automatically performed when the extraction of minimal cutsets or the construction of the binary decision diagram is launched. Moreover, it is fast (all operations it involves are of linear time complexity with respect to the size of the model).

As an illustration, Figure 1.8 shows a XFTA script to assess our model of the tracking system. Note that the third command prints out the target model, i.e. the system of stochastic Boolean equations generated from the source model.

Both approaches have their advantages and drawbacks, that we could summarize as follows:

– The binary decision diagram approach is very efficient both in terms of computation time and memory consumption.
– Moreover, it provides exact results regarding the calculation of probabilistic risk indicators.
– However, for very large models, it is impossible to build the binary decision diagram associated with the top event, because it is much too large to fit in computer memory.
– For these models, the minimal cutsets approach is a good alternative because it plays the role of a model pruner, i.e. it seeks for the most probable minimal cutsets, which concentrate the risk, and discard the others.
– Nevertheless, values of probabilistic risk indicators calculated from minimal cutsets are only approximated.
– When the probabilities at stake are low, this is not an issue: approximations are good *mutatis mutandis*. When they get high, this may be an issue as calculated values may be be over-pessimistic and weight wrongly components with respect to their contribution to the overall risk.

The above summary lets vague at least two points: what is a big model and what is a high probability. These two questions are impossible to answer formally, but we shall try to give some empirical hints along these pages.

XFTA implements the calculations of the most common probabilistic risk indicators, both from binary decision diagrams and from minimal cutsets. For the latter, several approximation methods are provided. Available probabilistic indicators are the following.

– Top-event probability, possibly at different mission times.
– Importance measures of basic events, i.e. marginal importance factor, critical importance factor, diagnosis importance factor, risk achievement worth, risk reduction worth.

- Sensitivity analysis by means of a Monte-Carlo simulation on the top-event probability.
- Approximation of system reliability by means of the calculation of the failure intensity.
- Safety integrity levels for low demand and high demand systems, as required by IEC 61508 (*International IEC Standard IEC61508 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems (E/E/PE, or E/E/PES)* 2010) and daughters standards.

1.3 Organization of the Book

This book is organized into three parts and an appendix.

The first part is dedicated to this introduction and the installation of XFTA. Aside the present chapter, it contains only one Chapter 2, describing the procedure to install XFTA on different operating systems.

Part II is dedicated to models. It consists in six chapters:

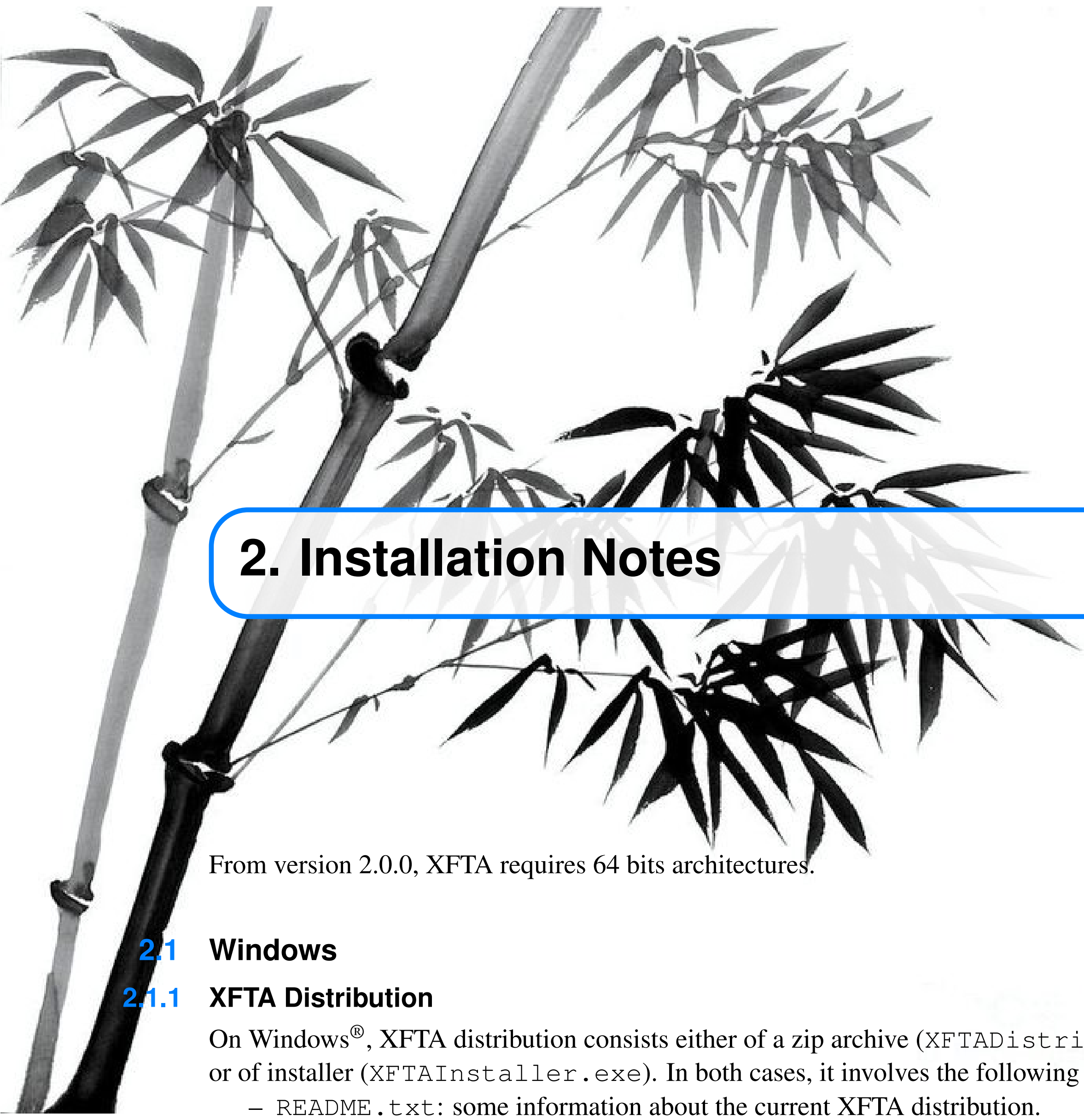
- Chapter 3 recalls the fundamentals about probabilistic risk and safety analyses. It aims at fixing the vocabulary that will be used throughout the remainder of the guide.
- Chapter 4 provides the reader with a guided tour of the two input formats, or modeling languages of the current version of XFTA, namely the Open-PSA format and S2ML+SBE.
- Chapter 5 introduces systems of stochastic Boolean equations, which are the core of models that can be designed and assessed by means of XFTA. It gives a precise mathematical definition of these systems and explains how they are encoded at the Open-PSA format.
- Chapter 6 is dedicated to probabilistic failure models associated with state variables of systems of stochastic Boolean equations, themselves representing failure modes of components of the systems under study. It gives a precise mathematical definition of each available failure model and explains how these models are encoded at the Open-PSA format.
- Chapter 7 is dedicated to extra-logical constructs such as common cause failure groups.
- Finally, Chapter 8 introduces constructs by which systems of stochastic Boolean equations are made a full fledged object-oriented modeling language.

Part III is dedicated to calculation procedures implemented in XFTA. It consists in the following chapters.

- Chapter 9 provides an overview of assessment methods.
- Chapter 10 provides an introduction to XFTA commands and scripts.
- Chapter 11 recalls the algebraic framework leading to the definition of minimal cutsets and presents the branch-and-test algorithm implemented in XFTA to extract them from a system of stochastic Boolean equations.
- Chapter 12 presents is dedicated to the binary decision diagram approach.
- Chapter 13 presents calculations of the top event probability.
- Chapter 14 presents calculations of importance measures.
- Chapter 15 presents sensitivity analyses.
- Chapter 16 presents calculations of safety integrity levels. This chapter describes also approximations of the reliability by means of the calculation of the failure intensity.

The appendix gathers some additional material. It consists of the following chapters.

- Appendix A recalls the main definitions and results of probability theory.
- Appendix B describes the probability distribution for periodically tested components.
- Finally, Appendix C gives some information about the successive releases of XFTA.



2. Installation Notes

From version 2.0.0, XFTA requires 64 bits architectures.

2.1 Windows

2.1.1 XFTA Distribution

On Windows®, XFTA distribution consists either of a zip archive (`XFTADistribution.zip`) or of installer (`XFTAInstaller.exe`). In both cases, it involves the following files.

- `README.txt`: some information about the current XFTA distribution.
- `INSTALL.txt`: some information about how to install XFTA.
- `XFTALicense.pdf`: XFTA License that you need to agree on before installing and using XFTA.
- `xfta.dll`: the dynamic load library (DLL) that implements the XFTA interpreter.
- `xftar.exe`: the executable file that calls the DLL.
- `xfta-api.h`: the application programmable interface.
- `xftar.cpp`: the C++ source code for `xftar.exe`.
- `xfta.bat`: a batch file making it possible to launch XFTA onto a script file.
- `s2ml+sbe.xml`: Notepad++ syntax highlighting rules for S2ML+SBE files (with extension `.sbe`).
- `xfta.xml`: Notepad++ syntax highlighting rules for XFTA script files (with extension `.xfta`).

The core of XFTA consists thus of the two files `xfta.dll` and `xftar.exe`.

⚠ **Visual Studio redistributable DLLs**
XFTA is compiled with Microsoft Visual Studio C++ 2019. You may need to install the Microsoft redistributable libraries to make it run (typically is you get the message "No xfta DLL"). You can find an installer of these libraries (“`vc_redist.x64.exe`”) on Microsoft website. For security reasons, you should download these libraries only from Microsoft websites.

2.1.2 Manual Installation

If you are familiar enough with Windows, you can just decompress the zip archive in the suitable folder, then make your life. This probably includes to add to your environment variable `PATH` the path to the installation folder.

Once the installation completed, to execute XFTA on a script file `myscript.xfta`, you have to open a command prompt and to enter:

```
xftar.exe myscript.xfta
```

XFTA prints error messages on the standard error file (`stderr`). The default output file is the standard output file (`stdout`).

2.1.3 "Automated" Installation

Alternatively, you can use execute the installer (by simply double clicking on it).

Here, I need to explain you what is going on. Fasten your seat belt, we gonna take off.

The installer creates first a folder in which it uploads the files described Section 2.1.1. By default, for the user `JohnDoe`, the installation folder is:

```
C:\Users\JohnDoe\AppData\Roaming\AltaRica Association\XFTA
```

Now to access `xftar.exe`, you have either to type the full path, which is admittedly not convenient, or tell Windows where to find it.

A way to do that could be to append the path to installation folder to the `PATH` environment variable (or more exactly to ask the installer to do that for you). That would be simple and efficient. Unfortunately, Windows is Windows and this does not work for three reasons:

1. There is not one `PATH` environment variable but two: one global for all the users and one local for the current user. When you ask the value of `PATH`, Windows returns the concatenation of both values. But when you want to modify it with a program, you need to modify one of them, but not both.
2. Still when you want to modify with a program the value of either path variable, the value that you give them must fit on a limited number of characters. This means that you just cannot concatenate paths to too many folders. Strangely enough, the problem does not exist when you set the value manually.
3. It is not easy to remove the path to the installation folder from `PATH` when you want to deinstall the software. Most installers just do not provide functionalities to do that.

This is the reason why Microsoft recommends to use the registry base instead of modifying `PATH`. The principle consists in adding a registry key, `xftar.exe` for XFTA, in:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\App Paths
```

This can be done automatically via the installer.

Now, you may expect that the executable `xftar.exe` is recognized as a Windows command and can be used as explained in the previous section. Unfortunately, for obscure reasons, this does not work. To launch XFTA on the script file `myscript.xfta`, you have to use the command `start`:

```
start /b xftar.exe myscript.xfta
```

The switch `/b` makes it possible to run the application (`xfta.exe`) in the same command prompt.

2.1.4 Integration with Notepad++

XFTA models and scripts are written into text files. This means that you need a text editor to create and modify them. I strongly advise to use Notepad++, which is a free, powerful, multi-purpose, user-friendly text editor. Notepad++ can be downloaded here:

`https://notepad-plus-plus.org/`

Notepad++ makes it possible to define syntax highlighting rules for your own language (and to modify those already embedded for virtually all popular programming languages). Moreover, it makes it possible to execute XFTA scripts directly from the editor.

To install these features, you shall need the files `s2ml+sbe.xml`, `xfta.xml` and `xfta.bat`.

The files `s2ml+sbe.xml` and `xfta.xml` define the syntax highlighting rules for respectively S2ML+SBE models and XFTA scripts. Using the menu `Language/User Defined Language` of Notepad++, it is possible to import (and modify at will) these rules, for files having extensions respectively `.sbe` and `.xfta`.

Running XFTA from Notepad++ is only slightly more complex. This is achieved in two steps:

– First, you have to click on submenu `Run` of the `Run`. There appears a small windows asking you which program to run. You have to enter in the command line the path to the file `xfta.bat`, i.e.

```
C:\Users\JohnDoe\AppData\Roaming\AltaRica
   Association\XFTA\Notepad++\xfta.bat
```

followed with the argument `$(FULL_CURRENT_PATH)`.

– Then, rather than to execute the command, you have to save it (e.g. under the name XFTA). You can then select the combination of keys making it possible to call it from the keyboard.

That's all.

2.2 Linux

With Linux, the situation is much simpler than with Windows. The distribution consists simply in a zip archive containing the following files.

– `README.txt`: some information about the current XFTA distribution.
– `INSTALL.txt`: some information about how to install XFTA.
– `XFTALicense.pdf`: XFTA License that you need to agree on before installing and using XFTA.
– `xftar`: the executable file that implements the XFTA interpreter.

To install XFTA, you have to decompress the zip archive in any folder, and to add to your environment variable `PATH` the path to that folder. Once installed, to run XFTA on a script file `myscript.xfta` you have to open a terminal and to enter:

```
xftar myscript.xfta
```

While executing script file, XFTA prints out error messages on the standard error file (`stderr`). The default output file is the standard output file (`stdout`).

2.3 Mac OS

Like on Linux, the situation is much simpler on Mac OS than with Windows. The distribution consists simply in a zip archive containing the following files.

– `README.txt`: some information about the current XFTA distribution.
– `INSTALL.txt`: some information about how to install XFTA.

– `XFTALicense.pdf`: XFTA License that you need to agree on before installing and using XFTA.
– `xftar`: the executable file that implements the XFTA interpreter.

To install XFTA, you have to decompress the zip archive in any folder, then to move the executable `xftar` into the folder `/usr/local/bin`. You should have the root password to do so. Once installed, to run XFTA on a script file `myscript.xfta` you have to open a terminal and to enter:

```
xftar myscript.xfta
```

While executing script file, XFTA prints out error messages on the standard error file (`stderr`). The default output file is the standard output file (`stdout`).

2.4 Application Programmable Interface

The application programmable interface of XFTA is described in the file `xfta-api.h`. It is made of a single C function: `int XFTA_EvalScriptFile(char* fileName);`. Calling this function executes the XFTA script file whose name is given in parameter.

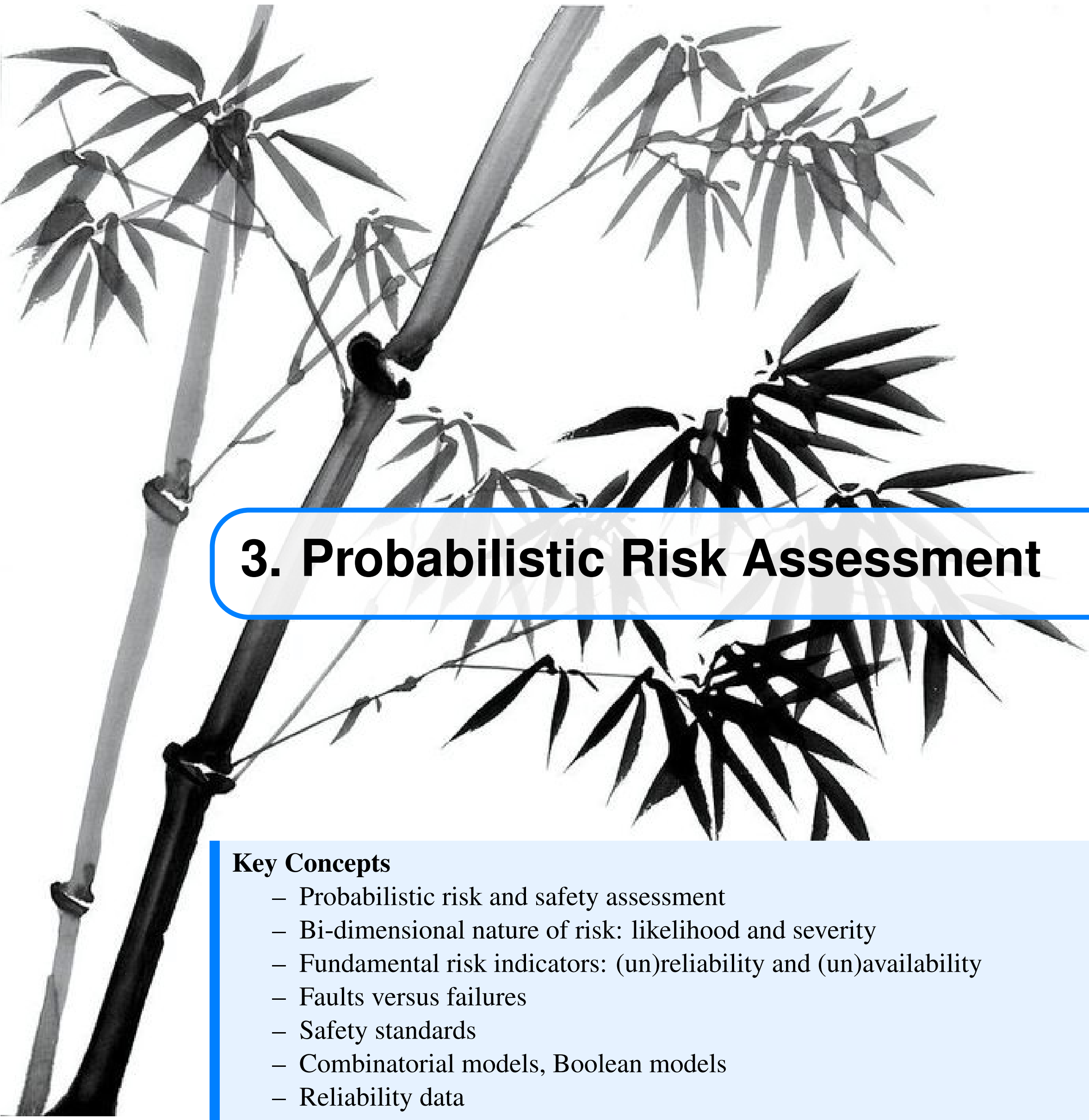
The source file `xftar.cpp` is for Windows. It just loads the DLL `xfta.dll` and calls the function:

```
XFTA_EvalScriptFile
```

on each argument of the command line. `xftar.cpp` is actually the source code of the executable `xftar.exe`.

II

Models



3. Probabilistic Risk Assessment

Key Concepts
- Probabilistic risk and safety assessment
- Bi-dimensional nature of risk: likelihood and severity
- Fundamental risk indicators: (un)reliability and (un)availability
- Faults versus failures
- Safety standards
- Combinatorial models, Boolean models
- Reliability data
- Computational complexity of assessment algorithms
- Epistemic versus aleatory versus computational uncertainty

This chapter recalls fundamentals about probabilistic risk and safety analyses. It aims at fixing the vocabulary that will be used throughout the remainder of the guide. The reader interested in thorough presentations of the domain should refer to introductory books, e.g. (Andrews and Moss 2002; Kumamoto and Henley 1996). It aims also at positioning XFTA in the probabilistic risk assessment process and discussing briefly the goals, means and limits of this process.

3.1 Risk and Safety Analyses

The operation of any technical system induces a risk for the system itself, its operators and its environment. This risk is impossible to avoid completely. The only question is therefore to assess whether it is socially acceptable, i.e. if accidents have a sufficiently low likelihood to occur. This is the role of safety/reliability analyses and more generally of reliability engineering.

These analyses are supported by models. As of today, the most popular models are fault trees and reliability blocks diagrams, i.e. models that can be designed and assessed with XFTA.

3.1.1 Definition of Risk

We call *risk* an event or a series of events that makes the system drift from a regular operation state to an incidental or accidental state. The risk is present in all human activities. It is bi-dimensional: a risk has a certain *likelihood* to occur and its consequences have a certain *severity*. Preventing a risk means either reducing its likelihood or the severity of its consequences, or both. It is in general impossible to eliminate fully the risk of operating a technical system. The only question is to lower it sufficiently to make it socially acceptable.

Assessing the risk requires thus studying it along its two dimensions: its likelihood and the severity of its consequences. Probabilistic risk assessment, in the sense we give here to this term, is about evaluating the likelihood of a risk. Assessing the severity of consequences is of course of primary importance, but it is industry specific and will not be discussed here.

A long journey has been made since the publication of the WASH 1400 report (Rasmussen 1975) followed by Three Mile Island nuclear accident (in 1979). Probabilistic risk analyses are nowadays widely accepted and used on a daily base in virtually all industries presenting significant risks for their operators, the public or the environment. The following safety standards and best practice guides (among others) recommend this approach.

– (*International IEC Standard IEC61508 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems (E/E/PE, or E/E/PES)* 2010)
– (*ISO/TR 12489:2013 Petroleum, petrochemical and natural gas industries – Reliability modelling and calculation of safety systems* 2013)
– (*Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment* 2004)
– (*Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment* 2004)
– (*ISO26262 Functional Safety - Road Vehicle* 2012)

Although everybody agrees that the likelihood of risks must be studied, many experts and non-experts are still reluctant to rely on probabilistic risk analyses to make decisions about the design and operations of technical systems. With good reasons. First, a catastrophic accident such as the melting of the core of a nuclear reactor is unacceptable, even if it is supposed to have a tiny probability. Second and more technically, one may have legitimate doubts that probabilities calculated via probabilistic risk analyses are "real" in any sense.

We shall not enter into this debate here, because we think it is pointless. Throughout this guide, we shall consider probabilities just as numerical indicators, without giving them any ontological status. These numerical indicators can be used to weight the different incident or accident scenarios described by the model and to study the sensitivity of risk indicators to variations of the parameters of the model. In other words, probabilities do not need to exist outside the model, provided that they reflect some relevant aspects of the system. As well explained by the famous aphorism of George P. Box, "all models are false, some are useful" (Box 1979).

This said, the choice of probabilities as the underlying mathematical framework of risk analyses is by no means arbitrary. First, a formal mathematical framework is anyway required to define consistent risk indicators. Second, as of today at least, no alternative framework exists that provides the same level of soundness, robustness and simplicity.

3.1.2 Fundamental Indicators

The reader has noticed that we made, at the end of the previous section, a conceptual shift which is all but innocent: we moved from likelihood, an informal concept, to probability, a formal one. Probabilistic risk analyses are not "likelihood risk analyses". They rely on non-ambiguous mathematical definitions, i.e. on formal models from which indicators can be calculated (in most of the cases, by a computer). In other words, although it is not possible to fully eliminate subjectivity when designing a model and setting its parameters, the model itself and the calculations made on this model are well defined mathematical objects on which it is possible to reason formally and to perform reproducible experiments. We believe that this is the only possible base for a scientifically founded and technically objective discussion. We recall below the basic concepts of system reliability theory that we shall use throughout this book.

Let S denote the system under study. Let T denote the date of the first failure of S. System reliability theory assumes that T is a random variable (see Appendix A for a precise definition). Note that this is not obvious from a mathematical standpoint, but we have no other choice than to accept this hypothesis, because the whole theory relies on it. T is called the *lifetime* of S.

Definition 3.1.1 — Reliability. The *reliability* $R_S(t)$ of S at time t is the probability that S experiences no failure during time interval $]0,t]$, given it was working at time 0. Formally,

$$R_S(t) \overset{def}{=} p(t < T)$$

The *unreliability*, or *failure cumulative distribution function* $F_S(t)$, is just the opposite.

$$F_S(t) \overset{def}{=} 1 - R_S(t)$$

$R_S(t)$ is a survival distribution. It is monotonically decreasing. Moreover, the following asymptotic properties hold.

$$\lim_{t\to 0} R_S(t) = 1 \tag{3.1}$$
$$\lim_{t\to\infty} R_S(t) = 0 \tag{3.2}$$

For systems that can be repaired, the concept of availability is used rather than the one of reliability.

Definition 3.1.2 — Availability. The *availability* $A_S(t)$ of S at time t is the probability that S is working at time t, given it was working at time 0.

$$A_S(t) \overset{def}{=} p(S \text{ is working at } t)$$

The *unavailability* $Q_S(t)$ is just the opposite.

$$Q_S(t) \overset{def}{=} 1 - A_S(t)$$

Here follows some additional definitions.

Definition 3.1.3 — Failure density. The *failure density* $f_S(t)$ is the probability density function of the law of T. It is the derivative of $F_S(f)$:

$$f_S(t) \overset{def}{=} \frac{\mathrm{d}F_S}{\mathrm{d}t}$$

For sufficiently small dt's, $f_S(t) \cdot dt$ represents the probability that the system fails between t and $t + dt$, given it was working at time 0.

The two following concepts are very frequently used, beyond probabilistic risk assessment.

Definition 3.1.4 — Mean Time To Failure. The *mean time to failure* $MTTF_S(t)$ describes the expected time to the first failure, i.e.

$$MTTF_S(t) \overset{def}{=} \int_{t=0}^{\infty} R_S(t)\mathrm{d}t$$

Definition 3.1.5 — Failure Rate. The *failure rate*, sometimes called *hazard function* $h_S(t)$ is defined as follows.

$$h_S(t) \overset{def}{=} \lim_{\Delta t \to 0} \frac{R_S(t) - R_S(t+\Delta t)}{\Delta t \cdot R_S(t)}$$

The following equalities hold from the definitions.

$$h_S(t) = \frac{f_S(t)}{R_S(t)} = \frac{f_S(t)}{1 - F_S(t)} \tag{3.3}$$

The above basic concepts are rigorously defined, if we accept the hypothesis that the first failure date of a system can be described as a random variable. They are defined as properties of the system, independently of any model to assess them.

The reliability engineering literature involves however several concepts that make implicit assumptions on the underlying model, without any reference to this model (most probably because there is no agreement on what should be this model). It is the case for instance of notions like the *probability of failure on demand* (PFD) and *probability of failure per hour* (PFH) defined by the mother safety standard (*International IEC Standard IEC61508 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems (E/E/PE, or E/E/PES)* 2010). Unfortunately, these notions play an important role in industrial practice as they are used to calculate *safety integrity levels*. The absence of well founded definition gives raise to multiple and incompatible interpretations and calculation algorithms.

3.1.3 Additional Vocabulary

We used above the words "degradation" and "failure" in an intuitive sense that must be made more precise. We shall follow here the definitions given in (*International electrotechnical vocabulary - Part 192: Dependability* 2015):

– A *failure* of a system or a component is an event that causes the termination of the function(s) this system or component normally performs. A failure occurs thus at a given instant and lasts no time.
– On the contrary, a *fault* is a state of a system or a component characterized by the inability of this system or component to perform the function(s) it normally performs, excluding inabilities due to maintenance actions or lack of external resources. Hence, a fault is the state resulting from a failure.

A *degradation* is thus a partial failure, i.e. a failure preventing the system or the component to perform fully the function(s) it normally performs. After a degradation, the system or the component is in a *degraded state*.

Strictly speaking, incidents or accidents at system level result thus from combinations of fault and degraded state rather than from failures and degradations.

The cited standard makes moreover the distinction between faults and errors. It defines *errors* as "discrepancies between a computed, observed or measured value of the condition and the true, specified or theoretically correct value or condition". An error is not a fault first because it may not prevent the system or the component to perform the function(s) it normally performs, and second because it may be non-permanent. We shall not make such distinction in the sequel as errors and faults can be treated in the same way from a modeling point of view.

There may be several reasons for which a system or a component is failed, i.e. a single system or component may have several faults. These faults are called *failure modes*. (*International electrotechnical vocabulary - Part 192: Dependability* 2015) recommends to call these states "fault modes" but the practice of calling them "failure modes" is so widely used that we shall keep this (inconsistent) terminology.

3.2 The Probabilistic Risk Assessment Process

3.2.1 Objectives and Means

Figure 3.1 sketches the probabilistic risk assessment process. It consists fundamentally of the following steps.

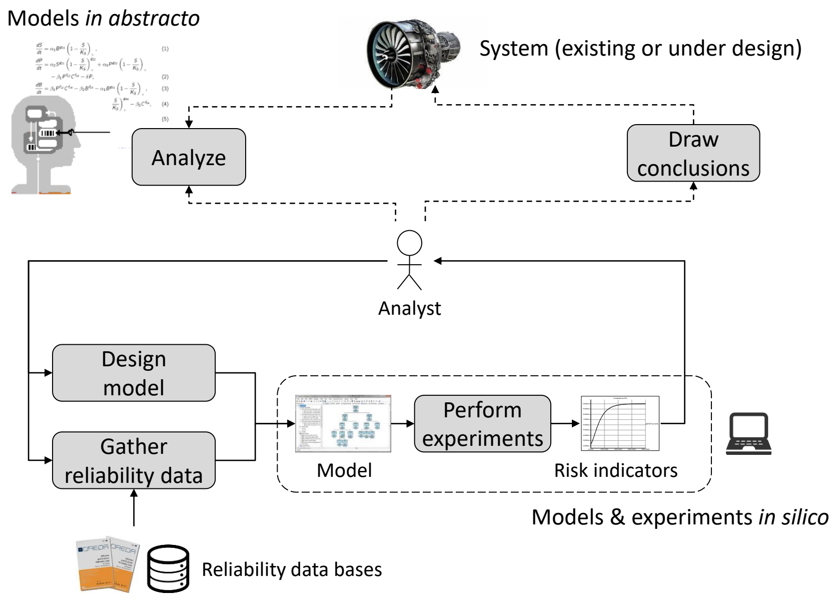


Figure 3.1: The probabilistic risk assessment process

First, the analyst tries to figure out how the system under study works and may fail. This includes a study of environmental conditions that may hinder the normal functioning of the system and initiates an incident or accident sequence. At this step, the analyst shapes a cognitive model of the system and its environment.

The second step consists in designing a computerized model of the system. To do so, the analyst relies on his cognitive model, possibly on previously designed models, and on data bases of reliability data, i.e. probabilistic failure models for the components of the system. Reliability data are obtained by different means: experience feedback on fleet of similar components operated in similar conditions, physical modeling, or simply expert judgment. Reliability data for components are stored into books or databases such as (*Reliability Prediction of Electronic Equipment: MIL-HDBK-217F.* 1995), OREDA (SINTEF and NTNU 2015) or (*IEC 61709:2017 - Electric components - Reliability - Reference conditions for failure rates and stress models for conversion* 2017).

The third step consists in calculating risk indicators from the model. Roughly speaking, two types of analyses are performed on the model:

– Qualitative analyses that consists in extracting scenarios of failure. These scenarios make it possible to validate the model and to check that the system is robust enough, e.g. that it has no single point failure, i.e. no component whose failure alone induces a failure of the system as a whole.
– Quantitative analyses that consists in assessing probabilistic indicators such as the reliability, the availability, the mean time to failure... The relative contribution of each component to the overall risk can be assessed as well.

In practice, the frontier between qualitative and quantitative analyses is blurred. As there are an enormous number of failure scenarios, even on small models, only the most probable ones are extracted. Moreover, probabilistic indicators are often calculated from the extracted failure scenarios.

Finally, the fourth step consists in drawing conclusions from the calculated risk indicators.

In practice, these steps are not performed in cascade. Some iterations may be actually necessary before reaching a stable state.

The whole process aims at two different objectives:

– Convince oneself that the system is safe and available enough to be operated. Both attributes are important. On the one hand, the risk of operating the system must be low enough to be socially acceptable. On the other hand, it must be possible to operate the system when it is needed, or at least most of the time it is needed. An aircraft that never takes off is perfectly safe, but definitely not useful.
– Convince stakeholders, notably clients and regulatory authorities, that everything has been made to make the system safe and available enough to be operated.

The second category of objectives explains why the whole probabilistic risk assessment process relies heavily on safety standards such as those listed in Section 3.1.1.

XFTA is a calculation engine for fault trees and related models. It is thus concerned mostly with the third step of the process. We shall see that it may play also a role in the second step, thanks to its powerful input language.

3.2.2 Classes of Models

There exists a wide variety of modeling formalisms dedicated to the design of probabilistic risk assessment models: fault trees, event trees, reliability block diagrams, Markov chains, stochastic Petri nets, dynamic fault trees, Boolean driven Markov processes, guarded transition systems, AltaRica. . . There are even more software tools supporting the design of models with these formalisms and implementing various assessment algorithms. Any attempt to list them would almost surely fail to be exhaustive.

It is however possible to establish a hierarchy of models, based on their mathematical properties. As I showed in my 2018 article (Rauzy 2018), there are eventually only three classes of models used in reliability engineering: combinatorial models, state automata and process algebras. These classes are nested in this sense that process algebras are strictly more expressive than state automata which are themselves strictly more expressive that combinatorial models.

The expressive power comes indeed with a price: as it increases, the difficulty of designing and validating models as well as the computational complexity of assessment algorithms increase dramatically.

Boolean models, i.e. fault trees and reliability block diagrams, are combinatorial models. They are the ones XFTA deals with. They occupy the lowest level of the hierarchy. They are the simplest or, to put it more formally, the less expressive. Nevertheless, assessing these models is already computationally expensive. Valiant showed in 1979 that calculating the exact value of the probability of the top-event of a fault tree is a #P-hard problem (Valiant 1979). In plain English, this means that no efficient algorithm is believed to exist to solve that problem.

Fortunately, for Boolean models, there exists efficient algorithms that push the limits of what is feasible. These algorithms calculate approximated values of risk indicators by ignoring failure scenarios whose probabilities are low enough. Although the accuracy of these approximations cannot be warrantied, it is reasonable for practical purposes.

Note however that ignoring low probability failure scenarios has its own downside. The more detailed the model, the lower the probability of individual scenarios of failure, i.e. eventually the larger the number of ignored scenarios. In the cited article, I called this phenomenon the *refinement paradox*. We shall come back on it in the second part of the guide.

In summary, in reliability engineering, a model results always of a trade-off between the accuracy of the description and the ability one has to perform calculations on this description within a reasonable amount of computational resources (running time and memory occupation).

Computational complexity rules reliability engineering.

The above considerations have one objective: to warn firmly the reader that there is no free lunch in reliability engineering. There is no point to design a very detailed model if risk indicators are too complex to calculate from this model. XFTA, although the most efficient tool available in its class is not a magic wand.

3.2.3 Limits

The conclusion of the previous section leads us to discuss the limits of the probabilistic risk assessment process.

First, the analyst can only put in the model what he knows about the system and its environment. There are unknown unknown (at least for the analyst). This is usually called *epistemic uncertainty* and crudely coined regarding models as *garbage in, garbage out*.

Second, reliability engineering is all about failures. By definition, no one knows when a failure will occur (otherwise it cannot be called a failure). One can however obtain probabilistic models for failures. This is usually called *aleatory uncertainty*. As explained above, failure models for different types of components come from different sources. The whole probabilistic risk assessment process relies heavily on the quality of reliability data. Here again, *garbage in, garbage out*.

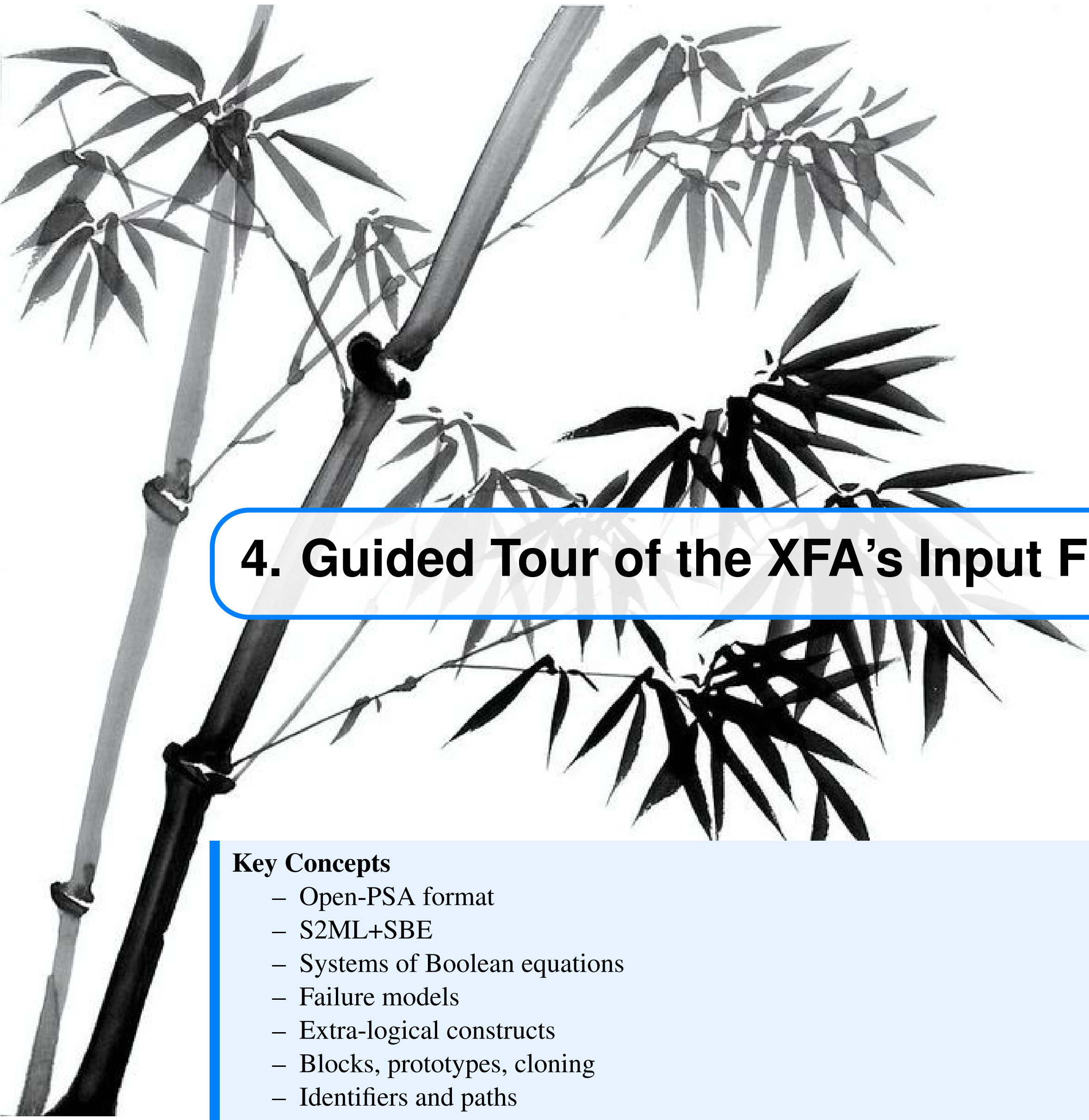
Third, as discussed in the previous section, calculations of risk indicators are provably computationally expensive. Consequently, designing a probabilistic risk assessment model results always of a trade-off: a fine grain model that represents the system's behavior very accurately can be eventually useless because only very approximated values of risk indicators can be calculated out of it. I called this phenomenon *computational uncertainty* in my 2018 article (Rauzy 2018).

Fourth, as the result of the specific compromise done by Boolean models, some features of systems cannot be represented faithfully. As an illustration, cold redundancies or reconfigurations can only be roughly approximated.

Fifth, low level modeling formalisms, such as fault trees and reliability block diagrams, introduce a distance between the system under study and the models. Consequently, models are hard to design, but more importantly to share with stakeholders and to maintain through the life-cycle of systems.

There is not much tools like XFTA can do about the knowledge the analyst has of the system (point one). However, thanks to object-oriented constructs (that we will present in depth Chapter 8), XFTA can facilitate knowledge capitalization, which in turn alleviates problems raised by epistemic uncertainties, and reduces the distance between the systems and the models (point five). Regarding the computational complexity of calculations (point three), XFTA helps to push the limits of what is feasible by providing best-in-class assessment algorithms. Nevertheless, it does not make all the problems vanish. Finally, XFTA cannot help regarding the quality of the reliability data (point two) and the inability of Boolean models to represent certain features of systems (point fourth).

To summarize the discussion of this section, the objective of XFTA is to provide reliability engineers with a powerful calculation engine for Boolean probabilistic risk assessment models. Nothing more, but nothing less.



4. Guided Tour of the XFA's Input Formats

Key Concepts
- Open-PSA format
- S2ML+SBE
- Systems of Boolean equations
- Failure models
- Extra-logical constructs
- Blocks, prototypes, cloning
- Identifiers and paths
- Classes, instantiation
- Graphical representations

This chapter introduces the two input formats, or modeling languages, supported by the current version of XFTA: the Open-PSA format on the one hand, S2ML+SBE on the other hand.

4.1 Preliminary Remarks

The "official" version of the Open-PSA format, if this makes any sense, is described in the document that Steven "Woody" Epstein and I wrote in 2018 (Epstein and Rauzy 2008). This version is version number 2.0d.

It consists of a XML grammar to describe (see e.g. (Ray 2003) for an introduction to XML):
- Systems of Boolean equations defining fault trees;
- Failure models associated with basic-events of fault trees;
- Extra-logical constructs such as common-cause failure groups;
- Event trees.

These elements are in line with the primary goal of the format, which was to deal with probabilistic risk assessment models coming from the nuclear industry. These models are for the most part combinations of event trees and fault trees (Kumamoto and Henley 1996).

However, none of the versions of XFTA implements event trees. The reason is that XFTA has been designed as a calculation engine for fault trees and that models that combine event trees and

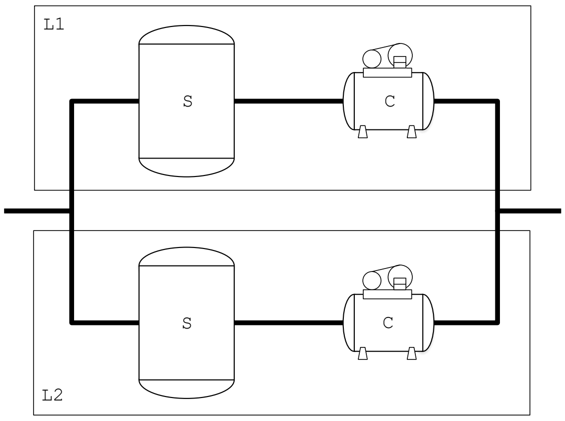


Figure 4.1: A putative oil & gas facility

fault trees are transformed into fault tree before being assessed. The fault tree resulting of this transformation is often called the *master fault tree* of the model.

The current version of XFTA implements a full-fledged object-oriented language, so-called S2ML+SBE, to author Boolean risk assessment models. S2ML+SBE belongs to the S2ML+X family of modeling languages (Batteux, Prosvirnova, and Rauzy 2018; Rauzy and Haskins 2019). It is thus both much more than the original Open-PSA format, as it extends it with object-oriented constructs, but also less, as it does not implement event trees. Moreover, conversely to the Open-PSA format, it is a textual language and not a XML grammar.

It was however unacceptable to have two different, and to some extent incompatible, input languages for XFTA. The decision I took was thus to update the Open-PSA format, i.e. to enhance it with S2ML constructs. In this way, the two input languages of XFTA are fully compatible, it would be even better to say strictly equivalent. The Open-PSA format comes has a XML grammar of SM2L+SBE.

Most of the constructs of the former version of the Open-PSA format are still available in the new version. There are however some minor differences, mostly regarding containers, paths and name spaces. Thanks to the object-oriented approach of S2ML+SBE, these concepts are no longer *ad-hoc* as they were in the original version of the format.

In the remainder of this chapter, we shall provide the reader with a guided tour of the (new version of) the Open-PSA format and S2ML+SBE. From now, “the Open-PSA format” shall stand for “the new version of the Open-PSA format”. We shall indicate where it is relevant the differences between the original and the new version of the format.

4.2 Illustrative Example

To illustrate our presentation, we shall consider throughout this chapter the putative oil & gas facility pictured in Figure 4.1.

This facility `F` is made of two lines, `L1` and `L2`, working in parallel. It is in charge of separating the mix of oil, gas and water coming from upstream (left on the figure), then to compress the gas and to send the compressed gas downstream (right on the figure). Each line consists thus of a separator `S` and a compressor `C`. One line is sufficient to ensure the production.

Both separators and compressors may fail. We shall assume for now that they have only one failure mode and that their probability of failures obey negative exponential distributions with respective rates $\lambda_{\mathtt{S}} = 1.23 \times 10^{-4}$ and $\lambda_{\mathtt{C}} = 5.67 \times 10^{-4}$.

Both the fault tree and the reliability block diagram describing the loss of the capacity provided

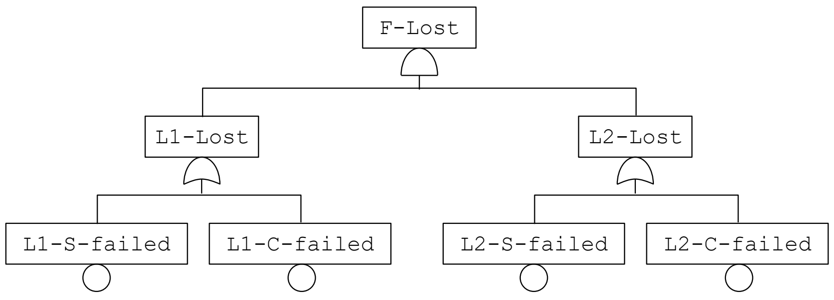


Figure 4.2: A simple fault tree describing the loss of the facility pictured in Figure 4.1

by our facility are trivial to design. Nonetheless, we shall use this toy example to illustrate what could be done if the facility were much more complex. Our objective here is not to present a realistic case study, but to introduce the constructs of the Open-PSA format and the S2ML+SBE language.

4.3 Systems of Boolean Equations

Figure 4.2 shows a simple fault tree describing the loss of the capacity provided by the facility pictured in Figure 4.1.

This fault tree says simply that the capacity provided by the facility F is lost if capacities provided by both lines L1 and L2 are lost, and that the capacity provided by a line is failed if either the separator S of that line is failed, or the compressor C of that line is failed.

This fault tree represents graphically the following system of Boolean equations.

$$\begin{aligned}
\texttt{F-lost} &= \texttt{L1-lost} \wedge \texttt{L2-lost} \\
\texttt{L1-lost} &= \texttt{L1-S-failed} \vee \texttt{L1-C-failed} \\
\texttt{L2-lost} &= \texttt{L2-S-failed} \vee \texttt{L2-C-failed}
\end{aligned}$$

4.3.1 Open-PSA format

A model at the Open-PSA format is made of one or more XML files. It consists essentially of a number of declarations of elements, such as gates (intermediate events), basic events, house events, parameters, common-cause failure groups....

The Open-PSA encoding for this system of Boolean equations is given in Figure 4.3.

We shall in the next section how probability distributions associated with basic events are represented.

For now, we can see that each intermediate event of the fault tree is declared by means of a tag define-gate and that it is referred to by means of a tag gate. This is a general principle of the Open-PSA format: an element of type LLL is always declared with a tag LLL and referred to with a tag LLL.

The attribute name is used to identify elements. We shall see the rules for identifiers of elements Section 4.6.

The remainder of the encoding is straightforward: intermediate events are described by means of Boolean formulas using connectives such as and and or, and references to basic and intermediate events.

4.3.2 S2ML+SBE

The principles of S2ML+SBE encoding of models are the same as those of the Open-PSA format. However, elements, here gates and basic-events, are directly declared with their type and referred

```xml
<?xml version="1.0" ?>
<!DOCTYPE open-psa >
<open-psa >
  <define-gate name="F-lost">
    <and>
      <gate name="L1-lost" />
      <gate name="L2-lost" />
    </and>
  </define-gate>
  <define-gate name="L1-lost">
    <or>
      <basic-event name="L1-S-failed" />
      <basic-event name="L1-C-failed" />
    </or>
  </define-gate>
  <define-gate name="L2-lost">
    <or>
      <basic-event name="L2-S-failed" />
      <basic-event name="L2-C-failed" />
    </or>
  </define-gate>
</open-psa >
```

Figure 4.3: Open-PSA description of the fault tree pictured in Figure 4.2

to simply with their name, i.e. declarations elements obey the following (meta-)grammatical rule.

```
ElementDeclaration ::=
   Type Path '=' Term ';'
```

In the case of intermediate events of fault trees, the `Type` is the keyword "`gate`", `Path` is the name of the event, and the `Term` is a Boolean formula (the definition of the event). We shall see Section 4.6 why the notion of `Path` is introduced here.

The S2ML+SBE description of the fault tree pictured in Figure 4.2, is thus as follows.

```
gate F-lost = L1-lost and L2-lost;
gate L1-lost = L1-S-failed or L1-C-failed;
gate L2-lost = L2-S-failed or L2-C-failed;
```

Chapter 5 provides a full specification of how systems of Boolean equations are encoded in the Open-PSA format and in S2ML+SBE.

4.3.3 Alternative Types

In this section, we used the fault tree terminology to set up the types of variables involved in systems of Boolean equations.

Both the Open-PSA format and S2ML+SBE make it possible to use a more variable oriented terminology, namely:

– `flow` instead of `gate`,
– `state` instead of `basic-event`,
– `source` instead of `house-event`.

It is however recommended not to mix up the two terminologies.

4.4 Failure Models

Now we have to associate probability distributions with the four basic events of our fault tree. In this case, all these probability distributions are negative exponential distributions. Both the Open-PSA format and S2ML+SBE provides constructs to encode the usual arithmetic operations. So it would be possible to use these constructs to associate probability distributions to basic events. However, both formats provide also built-in constructs that encode the most widely used parametric failure models such as the negative exponential distribution, which simplifies our task.

Now, the two basic events representing the failures of separators share their failure rate, λ_{S}, while the two basic events representing the failures of compressors share their failure rate, λ_{C}. Both the Open-PSA format and S2ML+SBE implement the notion of parameter. A *parameter* can be seen as a variable defined by a stochastic expression. In our example, the failure rates λ_{S} and λ_{C} can thus be represented by parameters.

4.4.1 Open-PSA format

Eventually, the Open-PSA code that associates negative exponential distributions to basic events of the fault tree pictured in Figure 4.2 is given in Figure 4.4. We assumed here that this code is stored in a separated XML file, but of course, nothing prevent to merge this file with the one in which the system of Boolean equations is stored.

This code is rather straightforward:

– Declaration of basic events are introduced by means of the tag `define-basic-event`, while those of parameters are introduced by means of the tag `define-parameter`.
– Just as basic events are referred to by means of the tag `basic-event`, parameters are referred to by means of the tag `parameter`.
– Names of basic events and parameters obey the same rules as those of gates (intermediate events) seen in the previous section.
– The tag `exponential` introduces the built-in construct for the negative exponential distribution.
– This construct takes one parameter, the failure rate, here defined by means of a parameter. We shall see however Chapter 6 that this construct may take an additional parameter, in case the mission time of the component is not the same as the one of the system.
– Values of parameters are constants, in this case floating point numbers.

4.4.2 S2ML+SBE

The principles of S2ML+SBE encoding of models are the same as those of the Open-PSA format. However, as gates, basic-events and parameters are directly declared with their type and referred to simply with their name.

The S2ML+SBE code to associate probability distributions to basic events of the fault tree pictured in Figure 4.2 is as follows.

```
basic-event L1-S-failed = exponential(lambda-S);
basic-event L1-C-failed = exponential(lambda-C);
basic-event L2-S-failed = exponential(lambda-S);
basic-event L2-C-failed = exponential(lambda-C);
parameter lambda-S = 1.23e-4;
parameter lambda-C = 5.67e-4;
```

In both the Open-PSA format and in S2ML+SBE, probability distributions associated with basic events are described by means of stochastic expressions, possibly involving parameters. Stochastic expressions implemented in XFTA are described Chapter 6.

```
<?xml version="1.0" ?>
<!DOCTYPE open-psa >
<open-psa >
  <define-basic-event name="L1-S-failed">
    <exponential>
      <parameter name="lambda-S" />
    </exponential>
  </define-basic-event>
  <define-basic-event name="L1-C-failed">
    <exponential>
      <parameter name="lambda-C" />
    </exponential>
  </define-basic-event>
  <define-basic-event name="L2-S-failed">
    <exponential>
      <parameter name="lambda-S" />
    </exponential>
  </define-basic-event>
  <define-basic-event name="L2-C-failed">
    <exponential>
      <parameter name="lambda-C" />
    </exponential>
  </define-basic-event>
  <define-parameter name="lambda-S">
    <float value="1.23e-4" />
  </define-parameter>
  <define-parameter name="lambda-C">
    <float value="5.67e-4" />
  </define-parameter>
</open-psa >
```

Figure 4.4: Open-PSA description of the probability distribution associated with the basic events of the fault tree pictured in Figure 4.2

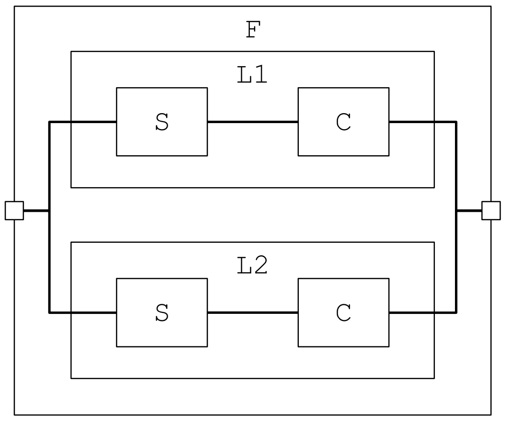


Figure 4.5: A hierarchical block diagram representing the facility pictured in Figure 4.1

4.5 Blocks (Take 1)

4.5.1 Hierarchical Block Diagrams

Designing a fault tree for our toy example was trivial. When the system gets large however, it may be not as simple. Or more exactly, many design choices have to be made. As a result, two different experts design usually two significantly different fault trees for the same system. This is already problematic for quality checking of models. This creates serious troubles regarding their maintenance, as the analyst who maintains the model is in general not the one who designed it, and the knowledge about models is extremely hard to capitalize, for both technical and economical reasons.

Hence the idea of designing models that would be "closer" to the system under study than fault trees. This may require the automated transformation of the models as designed into equivalent models that can be assessed by fault tree algorithms. However, this transformation process is implemented once for all, and its cost and the one of designing the modeling language are worth to pay given the expected benefits of the approach.

Hierarchical block diagrams are a good candidate to be the selected modeling language as they can reflect the architecture of the system under study.

Figure 4.5 shows a hierarchical block diagram representing the facility pictured in Figure 4.1.

This diagram shows both the architecture of the facility, by means of nested blocks, and the flows circulating through the network of components.

To make this abstract diagram a concrete model, we must simply add to the description:

- A Boolean variable `in` representing the input of each block, and another one, `out`, to represent its output;
- A Boolean variable `failed` to represent the state of each basic block;
- The equations describing how the flow circulates through the network.

The resulting system of Boolean equations is given in Table 4.1.

This system is what we want to obtain, but certainly not what we want to write. First, it does not reflect the architecture of the system, but in the naming conventions. Second, it would be tedious and error prone to have to write all of these equations by hand. Fortunately, S2ML constructs make things much easier to do.

4.5.2 Blocks: Basic Containers for Declarations

The first and most important concept of S2ML is the one of block. A *block* is a container for declarations of variables, parameters, extra-logical constructs and other blocks (and as we shall see,

Table 4.1: System of Boolean equations associating a behavioral description to the diagram pictured in Figure 4.5

F-out	=	L1-out ∨ L2-out			
L1-out	=	L1-C-out	L2-out	=	L2-C-out
L1-C-out	=	L1-C-in ∧ ¬L1-C-failed	L1-C-in	=	L1-S-out
L1-S-out	=	L1-S-in ∧ ¬L1-S-failed	L1-S-in	=	F-in
L2-C-out	=	L2-C-in ∧ ¬L2-C-failed	L2-C-in	=	L2-S-out
L2-S-out	=	L2-S-in ∧ ¬L2-S-failed	L2-S-in	=	F-in

other S2ML constructs). Blocks are *prototypes* in the sense of object-oriented theory (Abadi and Cardelli 1998).

If a block B contains the declaration of an element E, one says that B *composes* E.

For instance, in our example, the block F representing the facility composes the blocks L1 and L2 that represent the two lines. The block L1 composes in turn the block S and the block C. All these blocks compose Boolean variables in and out, and the blocks S and C compose in addition a Boolean variable failed and a parameter lambda.

In the Open-PSA format, block declarations are introduced by the tag block. This differs from the declaration of elements we have seen so far because blocks are never referred to (but indeed elements within blocks are).

The declaration of the block F is thus as follows.

```
<block name="F">
  ...
</block>
```

The ellipsis stands for all declarations of elements composed by the block F.

Similarly, block declarations in S2ML+SBE are introduced by the keyword block and terminated with the keyword end.

The declaration of the block F is thus as follows.

```
block F
  ...
end
```

Here again, the ellipsis stands for all declarations of elements composed by the block F.

Figure 4.6 shows a one-dimensional, tree-like, representation of the elements of the model.

Not to have a too large picture, the block L2 has been folded up. Its content is indeed similar to the one of the block L1.

Note that the model as a whole can be considered as an anonymous block.

4.6 Naming and Referring to Elements

Names of elements
This section is very important and therefore must be read with care. Both the Open-PSA format and S2ML+SBE rely on the same conventions to name and to refer to elements. These conventions must be understood prior designing any model.

Blocks, as the other containers, act as *name spaces*: within a block, two elements cannot have

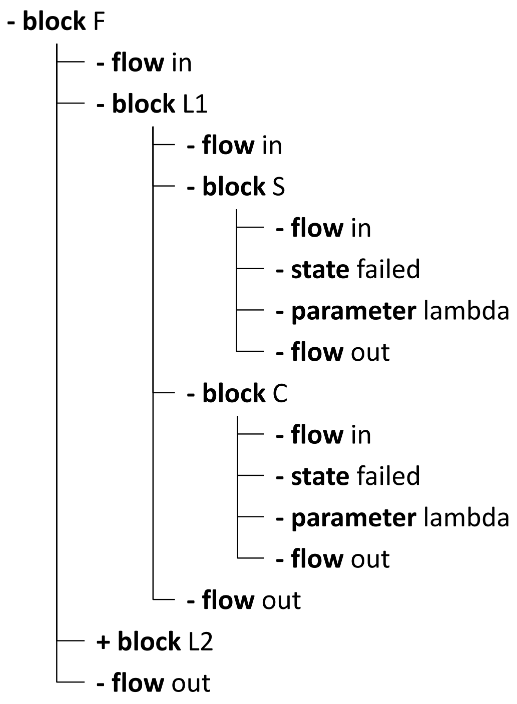


Figure 4.6: Elements of hierarchical block diagram representing the facility pictured in Figure 4.1

```
Identifier ::=
   (a-zA-Z_)(a-zA-Z0-9_-)*

Path ::=
      Identifier
   |  Identifier '.' Path
   |  'owner' '.' Path
   |  'main' '.' Path
```

Figure 4.7: Syntax of identifiers and paths for both the Open-PSA format and S2ML+SBE)

the same name even if they are of different types (e.g. a basic event and a parameter, or a gate and a block), but two elements declared in two different blocks can.

In our example, a flow variable `in` can be declared in blocks `F`, `L1`, `L2`... All these variables `in` are different. Similarly, a block `S` can be declared in blocks `L1` and `L2`. These two blocks `S` are different.

This raises two questions:

1. What are the allowed names for elements?
2. How to refer to an element which is declared in a block?

These two questions are answered by the notions of identifiers and paths.

4.6.1 Identifiers

In both the Open-PSA format and S2ML+SBE, identifiers of elements obey strict rules. Namely, an *identifier* starts with a letter or an underscore "_" followed by any number of letters, digits, underscores and hyphens "–".

The formal syntax of identifiers (for both the Open-PSA format and S2ML+SBE) is formally described Figure 4.7.

Consequently, "A", "in", "F-L1-S-in", "_42", and "_---" are valid identifiers. Indeed, it is highly recommended to use meaningful identifiers, and clearly, "_---" does not mean anything.

The case of characters is significant, which means that "Failed" and "failed" are different identifiers.

4.6.2 Paths

Within a block, and more generally a container, an element can be referred to by its name.

For instance, in our example, within the block S of the block L1 of the block F, variables in, out and failed as well as the parameter lambda can be referred by their names. Consequently, with this block, their declaration could be as follows (in S2ML+SBE).

```
   ...
   block S
     flow in = true;
     state failed = exponential(lambda);
     parameter lambda = 1.23e-4;
     flow out = in and not failed;
   end
   ...
```

Now, when it turns to define the variable out of the block F, one needs to refer to variables out of the block L1 and L2. This can be done using the dot notation:

```
block F
   ...
   flow out = L1.out or L2.out;
end
```

In other words, in a given current block, the path B.E refers to the element E of the block B composed by the current block. Hence in the block L1, C.out refers to the variable out composed by the block C, itself composed by the block L1.

This principle applies indeed to any number of nested blocks. In the block F, L1.S.lambda refers to the parameter lambda composed by the block S, itself composed by the block L1, itself composed by the block F.

It is thus possible to access any element defined anywhere in a block composed by the current block.

In fact, it is possible to access any element located anywhere in a hierarchy, using two additional directives:

– owner designates the parent block of the current block. For instance, in the block S composed by the block L1, owner designates the block L1. Consequently, owner.out refers to the variable out of the block L1, owner.C.out refers to the variable out of the block C composed by the block L1, and owner.owner.out refers to the variable out of the block F.
  Paths involving owner are thus relative to the current block.
– main designates the outer most block of the current hierarchy, i.e. the model as a whole. Consequently, in any block of the model, main.F.out refers to the variable out of the block F, and main.F.L2.out refers to the variable out of the block L2 of the block F.
  Paths involving main are thus absolute.

Although extremely powerful, or because they are extremely powerful, directives owner and main should be handled with much care. It is highly recommended to use only the usual dot notation, which requires to declare elements at the right of level of the block hierarchy.

Eventually, the grammar (in both the Open-PSA format and S2ML+SBE) for path is given in Figure 4.7.

4.7 Blocks (Take 2)

4.7.1 Redeclarations

In our model, we must connect ports of blocks. For instance, in the block `L1`, we must connect the variable `out` of the block `S` with the variable `in` of the block `C`. We can indeed do that using either the directive `owner` or the directive `main`, e.g.

```
...
block S
  ...
  flow out = in and not failed;
end
block C
  flow in = owner.S.out; // alternatively main.L1.S.out
  ...
end
...
```

This works fine for simple examples like the one presented here. However, it turns to be rather difficult to generalize to larger examples. Moreover, as we shall see it is incompatible with cloning and instantiation.

Another way to proceed consists in declaring variables first in the blocks the naturally belong to (with default values), then in re-declaring them at the suitable level so to represent the connections.

In the previous example, this could work as follows.

```
...
block S
  ...
  flow out = in and not failed;
end
block C
  flow in = true; // default value
  ...
end
flow C.in = S.out; // redeclaration
...
```

4.7.2 Cloning

So far, we were able to reflect the architecture of the system, which is indeed of interest, but not to simplify substantially the model. With that respect, it is clear that our description is not parsimonious: the description of line `L2` just duplicates the description of line `L1`.

S2ML provides the concept of cloning to simplify models in such a situation. Cloning is a programmatic way of doing a copy-paste, with the advantage that it keeps the memory that a copy-paste has been done.

The Open-PSA format code of a cloning directive is as follows.

```
...
<block name="L1">
  ...
</block>
<clones name="L2" block="L1" />
```

In the above code, attributes `name` and `block` can be given in any order.

Cloning in S2ML+SBE is just as simple:

```
...
block L1
  ...
end
clones L1 as L2;
```

Note that by combining cloning with parameter redeclaration, we can simplify further the model by obtaining the basic block `C` (of the line `L1`) from the basic block `S` (of the same line).

There is however a better way to achieve the same goal, as we shall see in the next section.

4.8 Classes

It is often the case that a system involves several components of the same type, e.g. pumps, valves, engines... The model of the system involves thus several copies of the same modeling elements, possibly with some minor variations. In our example, this is typically the case of basic blocks `S` and `C` that differ only by the value of the parameter `lambda`.

We have seen in the previous section that cloning can help to simplify models in such situation. Prototypes (blocks) and cloning are key notions of prototype-oriented programming (Abadi and Cardelli 1998; Noble, Taivalsaari, and I. Moore 1999).

Cloning a block requires that this block is defined somewhere in the model. There are case however where the same modeling component is generic and is used in many different models. Hence the idea of designing libraries of such components and to clone them when needed in the models.

Such on-the-shelf modeling component is called a *class*. The action of cloning it into a model is called *instantiation*. An instance of a class is called an *object*, which gives its name to object-oriented programming. Object-oriented programming is nowadays the dominant paradigm in software engineering. One can say without exaggeration that it is nearly hegemonic.

Both the Open-PSA format and S2ML+SBE provide a class/instance mechanism.

4.8.1 Open-PSA Format

In the Open-PSA format, declaring a class is done by means of a tag `class`. Except for that, classes are similar to blocks, i.e. they are containers for various elements including blocks and instances of classes.

The declaration of a basic block for our example could be as shown in Figure 4.8.

The class `BasicBlock` can then be instantiated in the block `L1` as follows.

```
...
<block name="L1">
  <instance name="S" class="BasicBlock" >
    <define-parameter name="lambda">
       <float value="1.23e-4" />
    </define-parameter>
```

```
<class name="BasicBlock">
  <define-flow name="in">
    <false />
  </define-flow>
  <define-state name="failed">
    <exponential>
      <parameter name="lambda" />
    </exponential>
  </define-state>
  <define-parameter name="lambda" >
     <float value="0.1" />
  </define-parameter>
  <define-flow name="out">
    <and>
      <flow name="in" />
      <not>
        <state name="failed" />
      </not>
    </and>
  </define-flow>
</class>
```

Figure 4.8: Declaration at Open-PSA format of a class to represent basic components

```
    </instance>
    ...
  </block>
  ...
```

Here, the value of the parameter `lambda` is modified (re-declared) within instance declaration, which can be seen as a container (a block). It would be indeed possible to re-declared it externally, as we have done for flow variables Section 4.7.1.

4.8.2 S2ML+SBE

Class declaration and instantiation in S2ML+SBE follows the same pattern as in the Open-PSA format.

Figure 4.9 shows a full S2ML+SBE model for the facility pictured Figure 4.1.

For the sake of the simplicity, we plugged the variable `in` directly on the variables `L1.S.in` and `L2.S.in`, and the variables `L1.C.out` and `L2.C.out` directly on the variable `out`.

4.9 Discussion

4.9.1 Other Structuring Constructs

As we shall see in Chapter 8, the paradigm S2ML+X, implemented by both the Open-PSA format and by S2ML+SBE provides a few other constructs to structure models. The key here is that the model as designed, e.g. the one given in Figure 4.9, is automatically transformed into the model as assessed, which is eventually equivalent to the system of Boolean equations given in Table 4.1.

```
class BasicBlock
   gate in = false;
   basic-event failed = exponential(lambda);
   parameter lambda = 0.1;
   gate out = in and not failed;
end

block F
   house-event in = true;
   block L1
      BasicBlock S
         parameter lambda = 1.23e-4;
      end
      BasicBlock C
         parameter lambda = 5.67e-4;
      end
      gate C.in = S.out;
   end
   clones L1 as L2;
   gate L1.S.in = in;
   gate L2.S.in = in;
   gate out = L1.C.out or L2.C.out;
end
```

Figure 4.9: A full S2ML+SBE model for the facility pictured Figure 4.1

`fault-tree` and `model-data`
The original version of the Open-PSA format had a pre-notion of container, via the constructs `fault-tree` and `model-data`. These constructs were however not really blocks, as they were not name spaces. We kept these constructs, for the sake of compatibility, assigning them no meaning. It is however strongly recommended not to use them anymore.

4.9.2 Extra-Logical Constructs

One of the most fundamental assumptions of the fault tree method is that basic events are statistically independent. This assumption must be verified for calculations of risk indicators to be mathematically correct and there is no way around. However, basic events represent failure modes of components of systems under study and some of these failure modes may not obey the independence assumption, for two main categories of reasons:

– Their probabilities of occurrence may be correlated in some way.
– They may be exclusive one another, due to physical or operational rules.

Analysts have thus to cope with this kind of situations, i.e. to approximate as accurately as possible statistically dependent events with modeling constructs obeying the statistical independence assumption. Moreover, this must be done in such a way that models stay relatively easy to design and that one can keep track of approximations.

This is the reason why many of the concrete modeling environments provide the analysts with extra-logical constructs such as common cause failure groups, delete terms, exchange events and the like. This is unfortunate for at least two reasons. First, it gives the false impression that dependencies are taken into account when performing calculations of risk indicators, which is indeed false. Second, these extra-logical constructs are *ad-hoc* and tool-dependent, hampering the mathematical soundness of calculations and the portability of models.

As a general rule, these extra-logical constructs should thus be avoided. It remains that there is a legacy of models involving such constructs. This is the reason why the XFTA input languages provide constructs to represent common cause failure groups. These constructs will be detailed Chapter 7. Chapter 7 will also explain why XFTA input languages do not provide other extra-logical constructs.

4.9.3 Graphical Representations versus Models

Throughout this chapter, we represented (the models of) the oil & gas facility in different ways:

- As a pseudo process and instrumentation diagram (Figure 4.1);
- As a tree like representation (Figure 4.2);
- As a hierarchical block diagram (Figure 4.5);
- As a one dimension tree (Figure 4.6);

All these graphical representations are thus representations of the same system, and eventually (except for Figure 4.2) of the same model.

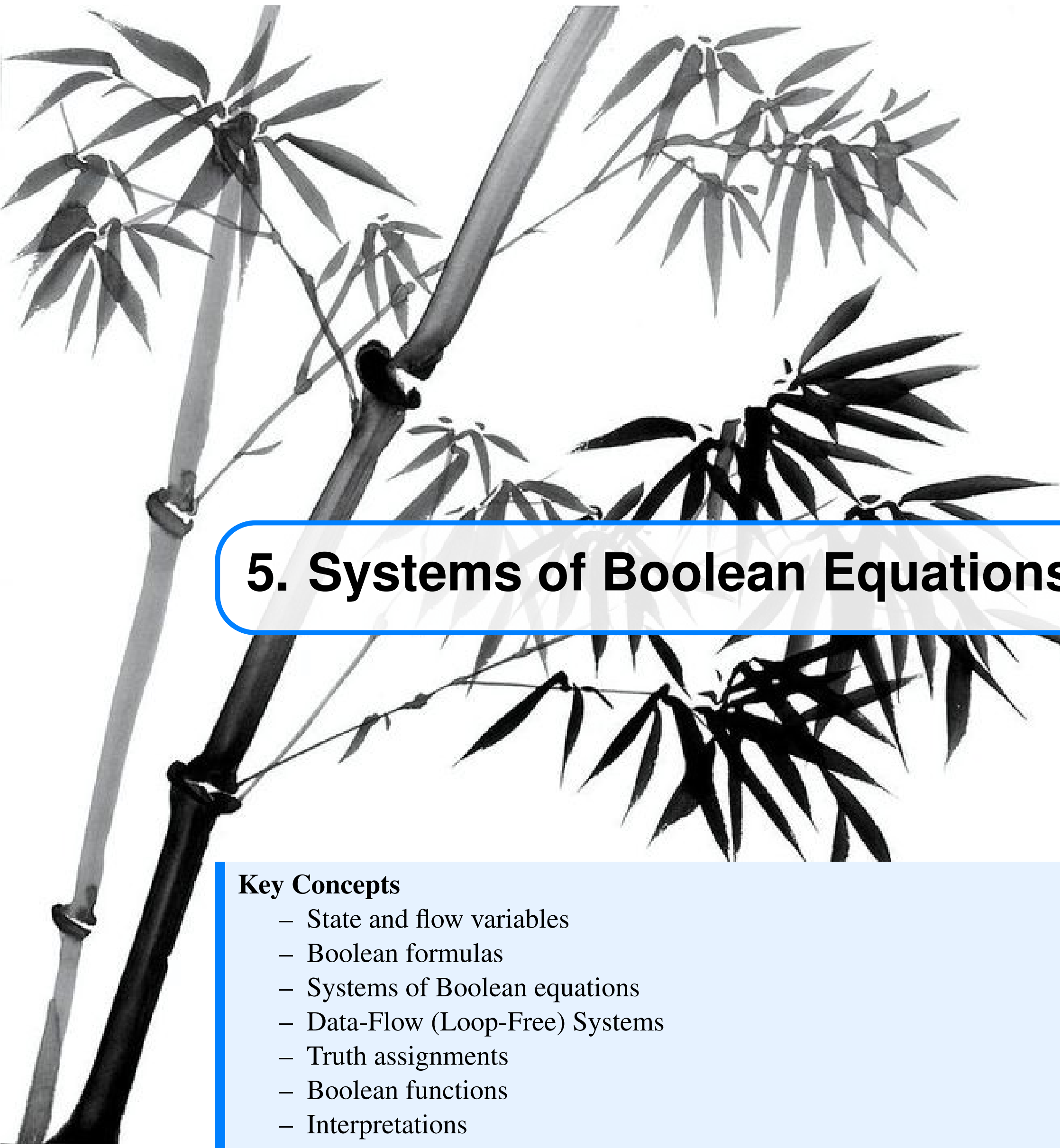
On the other hand, there are several parts of the model that we did not represent graphically, because it does make much sense:

- The probability distributions associated with basic events;
- The extra-logical constructs;
- The object constructs (cloning, instantiation).

For the latter, it would be however possible to use graphical notations such as UML (Rumbaugh, Jacobson, and Booch 2005) or SysML (Friedenthal, A. Moore, and Steiner 2011). However, it is not clear that this would be extremely helpful.

The key point here is that the model exists independently of its graphical representations. Graphical representations are useful to achieve a better communication, but the reference is the model, we mean here the textual model, no matter which grammar has been chosen (Open-PSA format, S2ML+SBE or any other).

This is the reason why neither the Open-PSA format, nor S2ML+SBE provide any guidance for graphical representations. They focus on what is essential: the model, only the model, but the whole model.



5. Systems of Boolean Equations

Key Concepts
- State and flow variables
- Boolean formulas
- Systems of Boolean equations
- Data-Flow (Loop-Free) Systems
- Truth assignments
- Boolean functions
- Interpretations
- Structure functions
- Entailment and Equivalence
- Boolean algebra identities
- Substitutions
- Coherence

This chapter introduces systems of stochastic Boolean equations, which are the core of models that can be designed and assessed by means of XFTA.

We shall first give a precise mathematical definition of these systems and show a number of classical properties (Sections 5.1-5.7). Then we shall explain how they are encoded at the Open-PSA format (Sections 5.8-5.9)

5.1 Boolean Formulas

Boolean formulas are built over the Boolean constants 1 (true) and 0 (false), a finite or denumerable set $\mathscr{V}$ of Boolean variables, and logical connectives "$\vee$" (or), "$\wedge$" (and) and "$\neg$" (not) as well as the parentheses "(" and ")". Throughout this guide, we shall use alternatively this logical notation for connectives and the alternative algebraic notation, used in many reliability engineering textbooks, which involves the symbols "$+$" (or), "$\cdot$" (and) and "$\bar{\ }$" (not).

Definition 5.1.1 — Boolean formulas. Let $\mathscr{V}$ be a finite or denumerable set of Boolean variables. The set of *Boolean formulas* built over $\mathscr{V}$ is the smallest set such that:

- Boolean constants 0 and 1 are Boolean formulas.
- Boolean variables of $\mathscr{V}$ are Boolean formulas.
- If f, $f_1 \dots f_n$ are Boolean formulas, then so are $f_1 \vee \cdots \vee f_n$, $f_1 \wedge \cdots \wedge f_n$, and $\neg f$.
- Finally, if f is a Boolean formula, then so is (f).

Parentheses are used to resolve ambiguities, like in $(A \vee B) \wedge (A \vee C)$.

Connectives have their usual precedence, i.e. "$\neg$" has priority over "$\wedge$" which has priority over "$\vee$". The formula $A \wedge B \vee \neg A \wedge C$ reads thus $(A \wedge B) \vee ((\neg A) \wedge C)$.

In the sequel, we shall denote Boolean variables by upper case letters, possibly with subscripts, e.g. A, E, $E_1 \dots$, and Boolean formulas by lower case letters, possibly with subscripts, e.g. f, g, $f_1 \dots$. We shall also omit Boolean when it will be clear from the context that we are speaking about Boolean variables and formulas.

The (finite) set of variables occurring in a formula f is denoted $\text{var}(f)$. It is recursively defined as follows.

Let f be a Boolean formula built over a finite or denumerable set $\mathscr{V}$ of Boolean variables. The set $\text{var}(f)$ set of Boolean variables occurring in f is the smallest set such that:

- $\text{var}(0) = \text{var}(1) = \emptyset$.
- $\text{var}(E) = \{E\}$ for any variable E.
- If f, $f_1, \dots f_n$ are formulas, then $\text{var}(f_1 \vee \cdots \vee f_n) = \text{var}(f_1) \cup \cdots \cup \text{var}(f_n)$, $\text{var}(f_1 \wedge \cdots \wedge f_n) = \text{var}(f_1) \cup \cdots \cup \text{var}(f_n)$ and $\text{var}(\neg f) = \text{var}(f)$.
- Finally, if f is a formula, then $\text{var}(\ (f)\ ) = \text{var}(f)$.

Indeed, the set of variables occurring in a formula is always finite.

5.2 Systems of Boolean Equations

Although presenting fault trees (and reliability block diagrams) as Boolean formulas is a convenient abstraction, in practice they are systems of Boolean equation rather than Boolean formulas.

Definition 5.2.1 — Systems of Boolean Equations. Let $\mathscr{V}$ be a finite or denumerable set of Boolean variables. A *system of Boolean equations* built over $\mathscr{V}$ is a finite set of equations in the form:

$$\begin{array}{rcl} V_1 & = & f_1 \\ & \vdots & \\ V_n & = & f_n \end{array}$$

where the V_i's are variables of $\mathscr{V}$ and the f_i's are Boolean formulas built over $\mathscr{V}$.

The notation $\text{var}(f)$ is lifted-up to systems of Boolean equations: let S be a system of Boolean equations built over a set of Boolean variables $\mathscr{F}$, then $\text{var}(S)$ is the union of the set of variables showing up in equations:

$$\text{var}(S) \overset{def}{=} \bigcup_{V=f \in S} \{V\} \cup \text{var}(f)$$

A variable V of $\text{var}(S)$ is a *flow variable* of S if it is the left member of an equation of S. V is *state variable* of S otherwise, i.e. if it occurs only in right members of equations of S.

S is *valid* if no variable occurs twice as the left member of an equation, i.e.

$$\forall V = f, W = g \in S,\ V \neq W$$

In the sequel, we shall assume that the systems of Boolean equations we consider are valid, unless explicitly said otherwise.

A flow variable V is a *root variable* if it does not occur in right members of equations of S. V is an *internal variable* otherwise.

S is *uniquely-rooted* if it has only one root variable.

Equations can be seen as definitions for flow variables. However, definition 5.2.1 does not prevent loops in the definitions of variables.

Let S be a system of Boolean equations built over a sets $\mathcal{V}$ of Boolean variables, let $V = f \in S$ and let W be another variable of $\mathrm{var}(S)$. Then the flow variable V *depends on* W, if and only if one of two following conditions hold.

– $W \in \mathrm{var}(f)$;
– There exists a variable $U \in \mathrm{var}(f)$, such that U depends on W.

Definition 5.2.2 — Data-Flow Systems. Let S be a system of Boolean equations over the set $\mathcal{V}$ of Boolean variables. Then S is *data-flow* or *loop-free* if none of its flow variables depends on itself, it is *looped* otherwise.

Both fault trees and reliability block diagrams are essentially valid, data-flow systems of Boolean equations. In most of the cases, they are uniquely rooted as well. Typically, in fault trees:

– Basic events are state variables.
– Intermediate events are flow variables.
– The top event is the unique root variable.

5.3 Interpretation

Boolean formulas are syntactic objects. Their semantics is defined in terms of Boolean functions, which are purely abstract objects.

Definition 5.3.1 — Truth assignment. Let $\mathcal{V} = \{V_1, \ldots V_n\}$, $n \geq 1$, be a set of Boolean variables. A *truth assignment* of $\mathcal{V}$ is a mapping from $\mathcal{V}$ to $\{0,1\}$, i.e. to Boolean constants.

There are 2^n possible different such truth assignments.

Truth assignments are first lifted-up into mappings from Boolean formulas to Boolean constants. Let σ be an assignment over $\mathcal{V}$ and f, f_1,$\ldots f_n$ be Boolean formulas built over $\mathcal{V}$, then:

– $\sigma(1) = 1$, $\sigma(0) = 0$.
– $\sigma(f_1 \vee \ldots \vee f_n) = max(\sigma(f_1), \ldots, \sigma(f_n))$,
– $\sigma(f_1 \wedge \ldots \wedge f_n) = min(\sigma(f_1), \ldots, \sigma(f_n))$,
– $\sigma(\neg f) = 1 - \sigma(f)$,
– Finally, $\sigma(\ (f)\ ) = \sigma(f)$.

Truth assignments are eventually lifted-up into mappings from systems of Boolean equations to Boolean constants. Let σ be an assignment over $\mathcal{V}$ and S be a system of Boolean equations built over $\mathcal{V}$. The σ is *valid* if the following condition holds.

$$\forall V = f \in S\ \sigma(V) = \sigma(f) \tag{5.1}$$

In case the system S is data-flow, the value of flow variables of S is uniquely determined by the values of its state variables, i.e. the following property holds.

Table 5.1: Truth tables

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

$\neg$	0	1
	1	0

Property 5.1 — Uniqueness of values of flow variables. Let S be a data-flow system of Boolean equations built over a set of Boolean variables $\mathcal{V} = \mathrm{var}(S)$ and let σ be a truth assignment of the state variables of S.

Then there exists a unique valid truth assignment σ' of variables of S such that $\sigma'(V) = \sigma(V)$ for all $V \in \mathcal{V}$.

Proof by induction. σ' can be built bottom-up starting calculating the values of flow variables V whose definition contains only state variables, then calculating the values of flow variables whose definition contains only state variables and flow variables whose value is already calculated and so on until the values of all the variables are calculated. This procedure is only possible if the system is data-flow.

An alternative way to define values of Boolean formulas under truth assignments is to use the well-known truth tables pictured in Table 5.1. Truth tables work however only for binary operators.

Definition 5.3.2 — Satisfaction, falsification. The truth assignment σ *satisfies* the formula f if $\sigma(f) = 1$, it *falsifies* f otherwise, i.e. if $\sigma(f) = 0$.

■ **Example 5.1** Consider for instance the formula $f = (A \wedge B) \vee (\neg A \wedge C)$. Truth assignments that satisfy and falsify f are given in the following table.

A	B	C	$A \wedge B$	$\neg A$	$\neg A \wedge C$	$(A \wedge B) \vee (\neg A \wedge C)$
0	0	0	0	1	0	0
0	0	1	0	1	1	1
0	1	0	0	1	0	0
0	1	1	0	1	1	1
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	1
1	1	1	1	0	0	1

Note that to build this table, we introduced a column for each subformula of f. ■

A Boolean formula f over a set $\mathcal{V}$ of Boolean variables can thus be interpreted as a mapping from truth assignments over $\mathcal{V}$ into $\{0,1\}$, i.e. eventually as a Boolean function over $\mathcal{V}$.

Definition 5.3.3 — Boolean functions. A *Boolean function* is a mapping from $\{0,1\}^n$ to $\{0,1\}$.

Definition 5.3.4 — Interpretation of Boolean formulas as Boolean functions. Let f be a Boolean formula built over a set of Boolean variables $\mathcal{V}$. Then f is interpreted as the unique Boolean function $[\![f]\!]$ such that for all truth assignment σ of $\mathcal{V}$, $[\![f]\!](\sigma) = \sigma(f)$.

This interpretation is extended to uniquely rooted, data-flow systems of Boolean equations.

Definition 5.3.5 — Structure functions. Let S be a uniquely rooted, data-flow system of Boolean equations built over a set of Boolean variables $\mathcal{V}$, and let $R \in \mathcal{V}$ be the root variable of S. Then S is interpreted as the unique Boolean function $[\![S]\!]$ such that for all valid truth assignment σ of $\mathcal{V}$, $[\![S]\!](\sigma) = \sigma(R)$.

In the reliability engineering literature, $[\![S]\!]$ is called the *structure function* of R.

Throughout this book, we shall keep the denomination "structure function" as a short hand for "the unique function in which the target top event is interpreted".

Note that a Boolean formula f (respectively a system of Boolean equations S) can be interpreted as a Boolean function over a set of variables $\mathcal{V}$ which is larger than $\text{var}(f)$. The values of variables of $\mathcal{V} \setminus \text{var}(f)$ just play no role.

This remark is important as it makes it possible to compare Boolean formulas even though they are not build over the same set of variables. If f and g are two Boolean formulas, we just have to compare their respective interpretations as Boolean functions over the set of variables $\text{var}(f) \cup \text{var}(g)$.

We shall now use this remark to state important properties.

5.4 Properties

Let us first define formally the notions of entailment and equivalence.

Definition 5.4.1 — Entailment and equivalence. Let f and g be two Boolean formulas built over a set of Boolean variables $\mathcal{V}$. Then,

– f *entails* g, which is denoted by $f \models g$ if any truth assignment over $\mathcal{V}$ that satisfies f satisfies g.

– f and g are *equivalent*, which we denote $f \equiv g$, if both $f \models g$ and $g \models f$, i.e. if f and g have the same set of satisfying truth assignments (over $\mathcal{V}$).

■ **Example 5.2** Consider the two formulas $f = (A \wedge B) \vee (\neg A \wedge C)$ and $g = B \wedge C$, then it is easy to prove that $g \models f$. One way to do that consists in using a truth table, as in example 5.1:

A	B	C		$(A \wedge B) \vee (\neg A \wedge C)$	$B \wedge C$
0	0	0		0	0
0	0	1		1	0
0	1	0		0	0
0	1	1		1	1
1	0	0		0	0
1	0	1		0	0
1	1	0		1	0
1	1	1		1	1

■

Boolean formulas obeys a number of equivalences. These equivalences define actually Boolean algebras (that we shall not study here). They are as follows.

Theorem 5.2 — Boolean algebras identities. Let f, g and h be Boolean formulas built over a set $\mathscr{V}$ of Boolean variables. Connectives "$\vee$", "$\wedge$" and "$\neg$" verify the following equivalences.

$$
\begin{array}{rclcll}
f \wedge g & \equiv & g \wedge f & & & \text{Commutativity of } \wedge \\
f \vee g & \equiv & g \vee f & & & \text{Commutativity of } \vee \\
f \wedge (g \wedge h) & \equiv & (f \wedge g) \wedge h & & & \text{Associativity of } \wedge \\
f \vee (g \vee h) & \equiv & (f \vee g) \vee h & & & \text{Associativity of } \vee \\
f \wedge (g \vee h) & \equiv & (f \wedge g) \vee (f \wedge h) & & & \text{Distributivity of } \wedge \text{ over } \vee \\
f \vee (g \wedge h) & \equiv & (f \vee g) \wedge (f \vee h) & & & \text{Distributivity of } \vee \text{ over } \wedge \\
f \wedge 1 & \equiv & 1 \wedge f & \equiv & f & \text{Neutral element for } \wedge \\
f \vee 0 & \equiv & 0 \vee f & \equiv & f & \text{Neutral element for } \vee \\
f \wedge 0 & \equiv & 0 \wedge f & \equiv & 0 & \text{Annihilator for } \wedge \\
f \vee 1 & \equiv & 1 \vee f & \equiv & 1 & \text{Annihilator for } \vee \\
f \wedge f & \equiv & f & & & \text{Indempotence of } \wedge \\
f \vee f & \equiv & f & & & \text{Indempotence of } \vee \\
f \wedge (f \vee g) & \equiv & f & & & \text{Absorption via } \wedge \\
f \vee (f \wedge g) & \equiv & f & & & \text{Absorption via } \vee \\
f \wedge \neg f & \equiv & 0 & & & \text{Complementation of } \wedge \\
f \vee \neg f & \equiv & 1 & & & \text{Complementation of } \vee \\
\neg\neg f & \equiv & f & & & \text{Double negation} \\
\neg f \wedge g & \equiv & \neg f \vee \neg g & & & \text{de Morgan's law for } \wedge \\
\neg f \vee g & \equiv & \neg f \wedge \neg g & & & \text{de Morgan's law for } \vee
\end{array}
$$

The proof is a simple application of the semantics of Boolean connectives.

5.5 Substitutions

Substitutions play an important role in assessment algorithms.

Definition 5.5.1 — Substitutions. Let f and g be formulas built over a set $\mathscr{V}$ of Boolean variables and let $V \in \mathscr{V}$. We denote by $f[V \leftarrow g]$ the formula obtained from f by substituting a copy of the formula g for each occurrence of V:

- $0[V \leftarrow g] = 0$ and $1[V \leftarrow g] = 1$.
- $V[V \leftarrow g] = g'$, where g' is a copy of g
- $W[W \leftarrow g] = W$ for any variable $W \neq V$.
- $(f_1 \vee \ldots \vee f_n)[V \leftarrow g] = f_1[V \leftarrow g] \vee \ldots \vee f_n[V \leftarrow g]$.
- $(f_1 \wedge \ldots \wedge f_n)[V \leftarrow g] = f_1[V \leftarrow g] \wedge \ldots \wedge f_n[V \leftarrow g]$.
- $\neg f[V \leftarrow g] = \neg f[V \leftarrow g]$.
- Finally, $(f)[V \leftarrow g] = (f[V \leftarrow g])$.

For instance, $(A \wedge B) \vee (\neg A \wedge C)[A \leftarrow 0] = (0 \wedge B) \vee (\neg 0 \wedge C)$ and $(A \wedge B) \vee (\neg A \wedge C)[A \leftarrow (D \wedge E)] = ((D \wedge E) \wedge B) \vee (\neg (D \wedge E) \wedge C)$.

Note that for any two formulas f and g and variable V, the following equality holds.

$$\mathrm{var}(f[V \leftarrow g]) \quad = \quad (\mathrm{var}(f) \setminus \{V\}) \cup \mathrm{var}(g) \tag{5.2}$$

where $\setminus$ stands for set difference.

The following property holds.

Property 5.3 — Substitutions of flow variables by their definition. Let S be a uniquely rooted, data-flow system of Boolean equations built over a set $\mathcal{V}$ of Boolean variables and let $V = f \in S$. Finally, let S' be the system of Boolean equations built over $\mathcal{V} \setminus \{V\}$ obtained from S by substituting V for f, i.e.

$$S' \overset{def}{=} \{W = g[V \leftarrow f], W = g \in S, W \neq V\}$$

Then, $S \equiv S'$.

By applying property 5.3 until the only flow variables is the root variable R, one can rewrite any uniquely rooted, data-flow system of Boolean equation S into an equivalent system $S^\star$ containing only one equation $R = f$.

This is in some way the syntactic counterpart of the definition of the interpretation of uniquely rooted, data-flow system of Boolean equations as Boolean functions.

Note however that the above rewriting may lead to a system $S^\star$ that is exponentially larger than the original system S, as illustrated by the following example.

■ **Example 5.3** Let S_n be the uniquely rooted, data-flow system of Boolean equation defined as follows.

$$\begin{array}{rcl} G_0 & = & C \\ G_1 & = & (A_1 \wedge G_0) \vee (B_1 \wedge \neg G_0) \\ G_2 & = & (A_2 \wedge G_1) \vee (B_2 \wedge \neg G_1) \\ & \vdots & \\ G_n & = & (A_n \wedge G_{n-1}) \vee (B_n \wedge \neg G_{n-1}) \end{array}$$

It is easy to see that the size of the right member of the equation doubles each time one substitute G_i for its definition in the definition of G_{i+1}. Consequently the size of $S_n^\star$ involve is exponential in n while the size S_n is linear in n. ■

The above example explains why systems of Boolean equations are preferable to mere Boolean formulas.

In practice, intermediate events of fault trees (and reliability block diagrams), not only avoid the above combinatorial explosion, but also make it possible to name losses of capacities, making in turn the model easier to design, to validate and to maintain.

5.6 Additional Connectives

In some application domains, including reliability engineering, it is convenient to introduce new connectives in addition to "$\vee$", "$\wedge$" and "$\neg$".

Definition 5.6.1 — Additional connectives. Let f, g and h be two Boolean formulas built over a set of Boolean variables $\mathcal{V}$. One defines the following connectives.

$f \Rightarrow g$	which is equivalent to	$\neg f \vee g$	implication
$f \Leftrightarrow g$	which is equivalent to	$(f \Rightarrow g) \wedge (f \Rightarrow g)$	equivalence
$f \oplus g$	which is equivalent to	$(f \wedge \neg g) \vee (\neg f \wedge g)$	exclusive or
$f \uparrow g$	which is equivalent to	$\neg f \wedge g$	nand, also called Sheffer's stroke
$f \downarrow g$	which is equivalent to	$\neg f \vee g$	nor
$f \,?\, g : h$	which is equivalent to	$(f \wedge g) \vee (\neg f \wedge h)$	if-then-else

It is easy to verify that all these connectives, but the implication and the if-then-else, are both associative and commutative.

Fault trees often represent voters. To do so, the special connective k-out-of-n is introduced. It is defined as follows.

Definition 5.6.2 — k-out-of-n. Let $f_1, f_2, \ldots f_n$, $n \geq 0$ be Boolean formulas built over a set of Boolean variables $\mathscr{V}$ and k be an integer. Then, $\texttt{atleast}\ k\ (f_1, \ldots, f_n)$ is a Boolean formula.

Moreover, let σ be a truth assignment. Then, $\sigma\,(\texttt{atleast}\ k\ (f_1, \ldots, f_n)) = 1$ if at least k out of the $\sigma(f_i)$'s are equal to 1 and 0 otherwise.

The k-out-of-n connective can be seen as a generalization of both connectives "$\wedge$" and "$\vee$" as it follows immediately from their definition that for any Boolean formulas $f_1, f_2, \ldots f_n$, the following equivalences hold.

$$f_1 \vee \ldots \vee f_n \equiv \texttt{atleast}\ 1\ (f_1, \ldots, f_n) \tag{5.3}$$
$$f_1 \wedge \ldots \wedge f_n \equiv \texttt{atleast}\ n\ (f_1, \ldots, f_n) \tag{5.4}$$

The reverse transformation is possible as well: k-out-of-n connectives can be rewritten using only connectives "$\wedge$" and "$\vee$".

The two following additional connectives are defined, for the sake of logical completeness.

– $\texttt{atmost}\ k\ (f_1, \ldots, f_n)$, which is true if and only if at most k out of $f_1, \ldots f_n$ are true.
– $\texttt{cardinality}\ l,h\ (f_1, \ldots, f_n)$, which is true if and only if at least l and at most h out of f_1, $\ldots f_n$ are true.

It is easy to verify that $\texttt{atleast}\ k$, $\texttt{atmost}\ k$ and $\texttt{cardinality}\ l,h$ are both associative and commutative. Moreover the following equivalences hold.

$$\texttt{atmost}\ k\ (f_1, \ldots, f_n) \equiv \neg\texttt{atleast}\ n-k\ (f_1, \ldots, f_n) \tag{5.5}$$
$$\texttt{atleast}\ k\ (f_1, \ldots, f_n) \equiv \neg\texttt{atmost}\ n-k\ (f_1, \ldots, f_n) \tag{5.6}$$
$$\texttt{cardinality}\ l,h\ (f_1, \ldots, f_n) \equiv \texttt{atleast}\ l\ (f_1, \ldots, f_n) \wedge \texttt{atmost}\ h\ (f_1, \ldots, f_n) \tag{5.7}$$

5.7 Coherence

Although XFTA implements a panoply of logical connectives and its core algorithms makes it possible to deal with any formula, one must always bear in mind that state variables (basic events) represents failed states (or lost capacities) of components of the system under study. Consequently, models are not arbitrary formulas, or to put it more exactly, XFTA is not designed to deal with arbitrary formulas. Its performance comes from a fundamental assumption: the models we deal with are coherent, or nearly coherent. Intuitively, this means that the more there are components failed in the system under study, the more likely the system itself is to be failed. In terms of models, this translates as the more basic events are realized (the more state variables are given the value true), the more likely the top event is to be realized (the more likely the root variable is to take the value true). The core algorithm of XFTA relies on the extraction of minimal cutsets of the model. All calculations of probabilistic indicators are performed on minimal cutsets. Considered as a formula, the set of minimal cutsets of a model is always coherent, no matter whether that model is coherent or not. Consequently, non coherent models can be used, but in very controlled way. We shall say more about this topics Chapter 11.

The notion of coherence (or monotony) is defined formally defined as follows.

Definition 5.7.1 — Coherence. Let f be a Boolean formula built over a set of Boolean variables $\mathscr{V}$. Moreover let V be a variable of $\text{var}(f)$.

Then f is *coherent* if for two variable assignments σ and σ' of $\mathscr{V}$ such that $\sigma(V) = 0$,

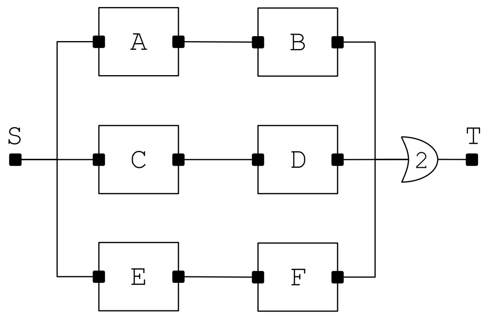


Figure 5.1: A reliability block diagram

Table 5.2: System of Boolean equations corresponding to the reliability block diagram pictured in Figure 5.1 obtained by means of a backward walk from `T` to `S`.

`T`	=	`atleast 2 (B-out, D-out, F-out)`			
`B-out`	=	`B-failed` ∨ `B-in`	`B-in`	=	`A-out`
`A-out`	=	`A-failed` ∨ `A-in`	`A-in`	=	`S`
`D-out`	=	`D-failed` ∨ `D-in`	`D-in`	=	`C-out`
`C-out`	=	`C-failed` ∨ `C-in`	`C-in`	=	`S`
`F-out`	=	`F-failed` ∨ `F-in`	`F-in`	=	`E-out`
`E-out`	=	`E-failed` ∨ `E-in`	`E-in`	=	`S`
`S`	=	`0`			

$\sigma'(V) = 1$ and $\sigma(W) = \sigma'(W)$ for all variables $W \neq V$, the following implication holds.

$$\sigma(f) = 1 \implies \sigma'(f) = 1$$

This definition extends naturally to uniquely rooted, data-flow systems of Boolean equations, by considering the value given to the root variable by σ and σ'.

It is easy to verify that a formula built only using connectives $\vee$, $\wedge$ and k-out-of-n is coherent. It is however possible to write coherent formulas using the other connectives, but this must be done cautiously. As an illustration, consider the reliability block diagram pictured in Figure 5.1.

A first way to build the system of Boolean equations corresponding to this diagram is to walk backward from its target node `T` down to its source node `S`. At each steps, one represents the conditions that makes the current node or port not reachable from `S`. The corresponding set of Boolean equations is given in Table 5.2.

This system of Boolean equations represents in some sense the logic of failure of the system under study. Indeed, it is coherent as it involves only connectives $\vee$, $\wedge$ and k-out-of-n.

Another way to see the reliability block diagram pictured in Figure 5.1 consists in establishing the conditions by which there exists a working path from `S` to `T`. This dual vision gives rise to the system of Boolean equations given in Table 5.3.

The absence of working path from `S` to `T` is then represented by the formula $\neg$`T`. This formula (and the system of Boolean equations) is coherent, despite it involves negations. It is actually strictly equivalent to the previous one. To check that it suffices to “push-down” negations using de Morgan’s laws and property 5.7.

Table 5.3: System of Boolean equations corresponding to the reliability block diagram pictured in Figure 5.1 obtained by establishing the conditions by which there exists a working path from S to T.

$$
\begin{array}{rcl rcl}
\texttt{T} & = & \texttt{atmost}\,1\,(\texttt{B-out},\texttt{D-out},\texttt{F-out}) & & & \\
\texttt{B-out} & = & \neg\texttt{B-failed}\wedge\texttt{B-in} & \texttt{B-in} & = & \texttt{A-out} \\
\texttt{A-out} & = & \neg\texttt{A-failed}\wedge\texttt{A-in} & \texttt{A-in} & = & \texttt{S} \\
\texttt{D-out} & = & \neg\texttt{D-failed}\wedge\texttt{D-in} & \texttt{D-in} & = & \texttt{C-out} \\
\texttt{C-out} & = & \neg\texttt{C-failed}\wedge\texttt{C-in} & \texttt{C-in} & = & \texttt{S} \\
\texttt{F-out} & = & \neg\texttt{F-failed}\wedge\texttt{F-in} & \texttt{F-in} & = & \texttt{E-out} \\
\texttt{E-out} & = & \neg\texttt{E-failed}\wedge\texttt{E-in} & \texttt{E-in} & = & \texttt{S} \\
\texttt{S} & = & 1 & & &
\end{array}
$$

5.8 Open-PSA Format

5.8.1 Declaration of Basic, Internal and House Events

As already explained, the Open-PSA format is declarative: each element of the model must be declared and can then be used. Actually, most of the elements can be used before they are declared, but this does not change the very principle.

Systems of Boolean equations encoding fault trees involve three types of variables:

– State variables, or equivalently basic events, that are associated with failure models.
– Flow variables, or equivalently gate events, that are defined by means of Boolean formulas.
– Source variables, or equivalently house events, that are special flow variables that are always defined by constants (true or false).

In the Open-PSA format, an element like a basic event, is always declared by means of a XML term of tag `define-basic-event` and referred to by means of term of tag `basic-event`.

As an illustration, consider the Boolean equation defining the flow variable `B-out` in the system of Boolean equations given in Table 5.2. This equation is encoded as follows.

```
<define-gate name="B-out">
  <or>
    <basic-event name="B-failed"/>
    <gate name="B-in"/>
  </or>
</define-gate>
```

Similarly, the basic event `B-failed` would be declared as follows, assuming it is associated with a negative exponential distribution of parameter 0.001 (failure models and their encoding will be presented Chapter 6).

```
<define-basic-event name="B-failed">
  <exponential>
    <float value="0.001"/>
  </exponential>
</define-basic-event>
```

Finally, assuming the flow variable `S` is implemented by means of a house event, its declaration would be as follows.

```
<define-house-event name="S">
  <false />
</define-house-event>
```

Table 5.4: Logical connectives implemented by XFTA

Tag	Attributes	Arity	Semantics
`false`		0	Boolean constant.
`true`		0	Boolean constant.
`and`		≥ 0	True if and only if all its arguments are.
`or`		≥ 0	True if and only if at least one of its arguments is.
`atleast`	`min`	≥ 0	True if and only if at least `min` of its arguments are.
`not`		1	True if and only if its unique argument is false.
`nand`		≥ 0	True if and only if at least one of its arguments is false.
`nor`		≥ 0	True if and only if all of its arguments are false.
`atmost`	`max`	≥ 0	True if and only if at most `max` of its arguments are.
`cardinality`	`min`, `max`	≥ 0	True if and only if at least `min` and at most `max` of its arguments are.

In the above declarations, we used the usual fault tree terminology. It is also possible to declare and to refer variables with the terminology we used in this chapter, i.e.

– `flow` can be used instead of `gate`.
– `state` can be used instead of `basic-event`.
– `source` can be used instead of `house-event`.

For instance, the following declaration of `B-out` is equivalent to the above one.

```
<define-flow name="B-out">
  <or>
    <state name="B-failed"/>
    <flow name="B-in"/>
  </or>
</define-flow>
```

It is however recommended, for the sake of clarity, not to mix up the two terminologies.

Variable names
Although XML attributes can be any string, names for variables—and more generally for all model elements—obey strict rules. Namely they must start with a letter or an underscore character "_", followed by any number of letters, digits, underscore characters and hyphens "-".

Moreover, two elements declared in the same container cannot have the same name, even though they are of different types. This applies indeed to variables, e.g. a basic-event and a gate cannot be named alike.

5.8.2 Boolean Formulas

XFTA implements logical connectives proposed by the Open-PSA format. Table 5.4 reviews these connectives. The first column gives the XML tag that introduces the connective. The second one gives the attributes, if any, that should be provided. The third one gives the arity, i.e. the number of expected arguments of the connective. Finally, the last one describes the semantics of the connective.

Figure 5.2 shows a partial encoding of the system of Boolean equations given in Table 5.3.

```xml
<?xml version="1.0" ?>
<!DOCTYPE open-psa>
<open-psa>
  <define-gate name="R">
    <not>
      <gate name="T" />
    </not>
  </define-gate>
  <define-gate name="T">
    <atmost max="1" >
      <gate name="B-out" />
      <gate name="D-out" />
      <gate name="F-out" />
    </atmost>
  </define-gate>
  <define-gate name="B-out">
    <and>
      <not>
        <basic-event name="B-failed" />
      </not>
      <gate name="B-in" />
    </and>
  </define-gate>
  <define-gate name="B-in">
    <gate name="A-out" />
  </define-gate>
  ...
  <define-house-event name="S">
    <false />
  </define-house-event>
</open-psa>
```

Figure 5.2: Open-PSA encoding of the system of Boolean equations given in Table 5.3 (partial)

Note that nothing prevents to use nested formulas (such as a `and` of `or` of events). However, all safety standards recommend to introduce an intermediary event per logical gate. The only exception to this rule is stands in negations, as illustrated in our example.

Variables can be declared in any order, provided that there is no loop in the declarations (i.e. that the system of Boolean equations is data-flow).

5.9 S2ML+SBE

5.9.1 Declaration of Basic, Internal and House Events

In S2ML+SBE, declarations of basic elements take the form:

```
type name = value ;
```

Conversely to the Open-PSA format, elements are just referred to by their name.

As an illustration, consider the Boolean equation defining the flow variable `B-out` in the system of Boolean equations given in Table 5.2. This equation is encoded as follows.

```
gate B-out = B-failed or B-in;
```

Similarly, the basic event `B-failed` would be declared as follows, assuming it is associated with a negative exponential distribution of parameter 0.001 (failure models and their encoding will be presented Chapter 6).

```
basic-event B-failed = exponential(0.001, mission-time);
```

Finally, assuming the flow variable `S` is implemented by means of a house event, its declaration would be as follows.

```
house-event S = false;
```

In the above declarations, we used the usual fault tree terminology. It is also possible to declare and to refer variables with the terminology we used in this chapter, i.e.

– `flow` can be used instead of `gate`.
– `state` can be used instead of `basic-event`.
– `source` can be used instead of `house-event`.

For instance, the following declarations are equivalent to the above ones.

```
flow B-out = B-failed or B-in;
state B-failed = exponential(0.001, mission-time);
source S = false;
```

It is however recommended, for the sake of clarity, not to mix up the two terminologies.

5.9.2 Boolean Formulas

S2ML+SBE implements the same logical connectives as the Open-PSA format, i.e. those given in Table 5.4.

Figure 5.3 shows the encoding of the the system of Boolean equations given in Table 5.3.

Figure 5.4 gives the EBNF grammar for S2ML+SBE declarations of systems of Boolean equations.

Note that as for the Open-PSA format, formulas can be nested and variables can be declared in any order, provided that there is no loop in the declarations (i.e. that the system of Boolean equations is data-flow).

```
flow R = not T;
flow T = atmost 1 (B-out, D-out, F-out);
flow B-out = not B-failed and B-in;
flow B-in = A-out;
flow A-out = not A-failed and A-in;
flow A-in = S;
flow D-out = not D-failed and D-in;
flow D-in = C-out;
flow C-out = not C-failed and C-in;
flow C-in = S;
flow F-out = not F-failed and F-in;
flow F-in = E-out;
flow E-out = not E-failed and E-in;
flow E-in = S;
source S = false;
```

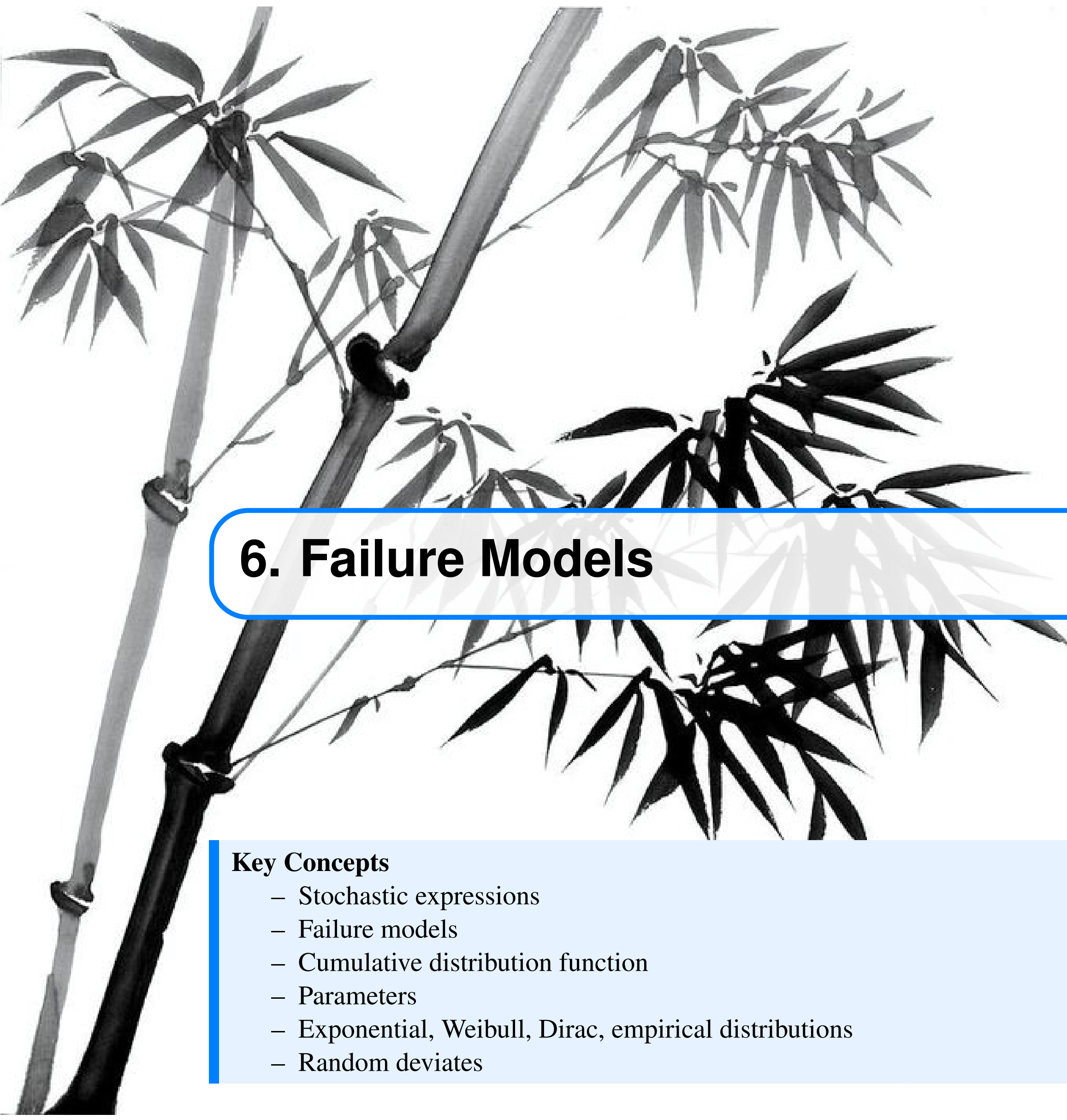
Figure 5.3: S2ML+SBE encoding of the system of Boolean equations given in Table 5.3

Usual precedence rules are applied: `not` has a higher priority than `and`, which itself has a higher priority than `or`.

Keywords
Keywords like `flow`, `state`, `and`, `or`... are reserved words, meaning that they cannot be used as names for elements (variables, parameters...).

```
FlowVariableDeclaration ::=
   ('flow' | 'gate') Path '=' BooleanFormula ';'

StateVariableDeclaration ::=
   ('state' | 'basic-event') Path '=' StochasticExpression ';'

SourceVariableDeclaration ::=
   ('source' | 'house-event') Path '=' ('false' | 'true') ';'

BooleanFormula ::=
      'false' | 'true'
   |  BooleanFormula ('or' BooleanFormula)+
   |  BooleanFormula ('and' BooleanFormula)+
   |  not BooleanFormula ';'
   |  Connective '(' BooleanFormulaArguments ')'
   |  '(' BooleanFormula ')'

Connective ::=
      'and' | 'or' | 'nand' | 'nor'
   |  'atleast' Integer | 'atmost' Integer
   |  'cardinality' Integer Integer

BooleanFormulaArguments ::=
    BooleanFormula ( ',' BooleanFormula )*
```

Figure 5.4: EBNF grammar for S2ML+SBE declarations of systems of Boolean equations



6. Failure Models

Key Concepts

– Stochastic expressions
– Failure models
– Cumulative distribution function
– Parameters
– Exponential, Weibull, Dirac, empirical distributions
– Random deviates

In fault tree analysis, probability distributions are associated with basic events so to calculate various probabilistic risk indicators. These probability distributions are often called failure models in the reliability engineering literature. In both the Open-PSA format and S2ML+SBE, they are described by means of stochastic expressions. This chapter describes stochastic expressions implemented in XFTA.

6.1 Stochastic Expressions

6.1.1 The Two Roles of Stochastic Expressions

Stochastic expressions play actually two roles in fault tree analysis.

First, they are used to associate a probability with basic events. This probability may vary through the time. Intuitively, a component that ages is more and more likely to be failed. This is the reason why one speaks about probability distribution. The distribution is here understood as the evolution of the probability through the time.

For instance, associating a negative exponential distribution of parameter λ with a basic event `E` means that the probability of `E` obeys the following function of the time t.

$$Q_E(t) \overset{def}{=} 1-\exp(-\lambda.t) \tag{6.1}$$

Such a probability distribution is called a *failure model*.

Second, stochastic expressions are used to describe the uncertainties on the probability themselves, whether they depend on the time or not. For instance, the parameter λ of the above negative exponential distribution may be known only up to some uncertainty. This uncertainty may be itself represented by a probability distribution. For instance, we may assume that λ is uniformly distributed between two bounds, λ_{min} and λ_{max}.

The constructs that describe such distributions are called *random deviates*.

Chapter 15 presents how random deviates are used to perform sensitivity analyses on the values of probabilistic risk indicators.

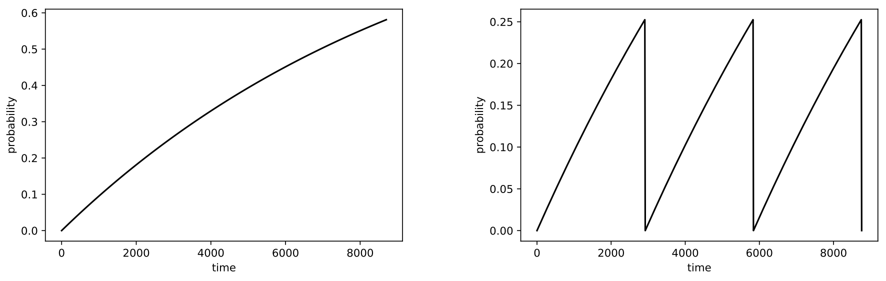
6.1.2 Cumulative versus non Cumulative Distribution Functions

If the system under study is not maintained, i.e. if none of its components is repairable or replaceable, probability distributions associated with basic events are cumulative, i.e. that for any time $0 \leq t < t'$ and any basic event E, the following inequality holds.

$$Q_E(t) \quad \leq \quad Q_E(t') \tag{6.2}$$

Many industrial systems are however periodically maintained, which means that the above inequality is not verified and that availability of the components and the system (their probabilities to be working at time t, given it was working at time 0) is not the same a their reliability (the probability that they worked without interruption between time 0 and at time t).

Figure 6.1 shows two distributions. The one on the left is cumulative. It is actually a negative exponential distribution. The one on the right is not cumulative. It corresponds actually to a periodically tested component which is assumed to be as good-as-new after each maintenance intervention.



(a) Cumulative distribution function (b) Non-cumulative distribution function

Figure 6.1: Cumulative versus non-cumulative distribution functions

The Open-PSA format, S2ML+SBE and XFTA provide constructs to represent both types of distributions.

If at least one of the basic events is associated with a non-cumulative distribution, the availability of the system as a whole is different from its reliability. This is by no means a problem if the analyst wants to study the former. Fault tree algorithms can anyway only calculate the availability of the system at time t. Under some conditions, its reliabibility can only be approximated. We shall present XFTA algorithms to do so in Chapter 16.

6.1.3 Categories of Stochastic Expressions

XFTA implements a wide range of stochastic expressions:

– Arithmetic expressions such as addition, multiplication...
– Boolean expressions, including inequalities between arithmetic expressions.
– Conditional expressions such as if-then-else.
– Parametric failure models such as the exponential distribution, the Weibull distribution...
– Random deviates such as the uniform distribution, the normal distribution...

– Empirical distributions defined by sets of points.

Moreover, it is possible to define parameters, which can be seen as stochastic variables.

Ideally, it would be possible to use any of these constructs as an argument of any other one. In practice however, some combinations are meaningless or raise calculation problems. For instance, it is meaningless to have a failure model as a parameter of a random-deviate or to introduce calculations in empirical distribution. Eventually, the constraints on daughter expressions (arguments) of an expression are the following.

– Failure models take arguments that are either ground arguments or expressions calculated from random-deviates.
– Random-deviates take only ground arguments, i.e. expressions that can be calculated once for all. For instance, they cannot depend on the time.
– Empirical distributions involve only numerical constants.

In the remainder of this chapter, we shall review in turn all of these expressions and give their Open-PSA and S2ML+SBE syntaxes.

6.2 Parameters

As said above, *parameters* can be seen as stochastic variables. Once defined, they can be referred to as many times as needed.

6.2.1 Open-PSA Format

Similarly to logical variables, parameters are declared with the tag `define-parameter` and referred to with the tag `parameter`, e.g.

```
<define-basic-event name="Pump33Failed">
  <exponential>
    <parameter name="pumpFailureRate" />
  </exponential>
</define-basic-event>

<define-parameter name="pumpFailureRate" >
  <float value="1.23e-5"/>
</define-parameter>
```

Indeed, the same parameter can be used for different basic events. Moreover, the definition of parameters can be any stochastic expression and may involves other parameters.

⚠ **Parameter names**
Although XML attributes can be any string, names for parameters—and more generally for all model elements—obey strict rules. Moreover, two elements declared in the same container cannot have the same name, even though they are of different types. This applies indeed to parameters, e.g. a basic-event and a parameter cannot be named alike. See Section 4.6.

⚠ **Loops in parameter definitions**
A parameter can depend on other parameters. However, it cannot depend eventually on itself. In other words, the system of stochastic equations defining parameters must be data-flow.

6.2.2 S2ML+SBE

Declarations of parameters in S2ML+SBE obey the same principles as the declarations of (state) variables, except that they are introduced by the keyword `parameter`. Figure 6.2 shows the EBNF grammar of parameter declarations as well as the different categories of stochastic expression (see previous section).

```
ParameterDeclaration ::=
   'parameter' Path '=' StochasticExpression ';'

StochasticExpression ::=
         ParameterReference
   |   Constant
   |   ArithmeticExpression
   |   TrigonometricExpression
   |   AggregationExpression
   |   BooleanExpression
   |   Inequality
   |   ConditionalExpression
   |   FailureModel
   |   RandomDeviate
   |   '(' StochasticExpression ')'

ParameterReference ::=
   Path
```

Figure 6.2: EBNF S2ML+SBE grammar for declarations of parameters and stochastic expressions

Here follows an example of declaration of parameter and reference to this parameter.

```
basic-event Pump33Failed = exponential(pumpFailureRate);
parameter pumpFailureRate = 1.23e-5;
```

6.3 Base Expressions

We shall review first base expressions: constants, references to parameters, arithmetic operations. . .

6.3.1 References and Constants

The basic stochastic expressions are *constants* and *references to parameters*. Table 6.1 describes the implementation of these basic elements in the Open-PSA format. In S2ML+SBE, these elements are given directly.

Table 6.1: References and constants

Tag	Attributes	Semantics
`parameter`	`name`	Reference to the parameter of the given name
`false`		Boolean constant
`true`		Boolean constant
`int`	`value`	Integer
`float`	`value`	Floating point number
`pi`		Constant π
`mission-time`		Current mission time

The Open-PSA format distinguishes Boolean constants, integers and floating point numbers. Hence the three constructs presented in Table 6.1. The Open-PSA format provides also a construct for $\pi = 3.14156\cdots$.

Finally, the Open-PSA format (and S2ML+SBE) provides a special construct for the mission time: `mission-time`. `mission-time` can be seen as a parameter whose value is set by the calculation engine. XFTA makes it possible to perform most of the calculations at different mission times (see Section 13.2.1.2).

Here follows a few examples of references and constants at the Open-PSA format.

```
<parameter name="pumpFailureRate" />
<false />
<true />
<int value="42"/>
<float value="1.23e-5"/>
<pi />
<mission-time />
```

6.3.2 Arithmetic Operations

XFTA implements *arithmetic operations* supported by most of the programming languages. Table 6.2 gives the list of these operations.

Table 6.2: Arithmetic operators

Tag	Arity	Semantics
`neg`	1	Unary minus
`add`	≥ 2	Addition
`sub`	2	Subtraction
`mul`	≥ 2	Multiplication
`div`	2	Division
`abs`	1	Absolute value
`exp`	1	Exponential
`log`	1	Logarithm
`log10`	1	Logarithm base 10
`mod`	2	Modulo
`pow`	2	Power
`sqrt`	1	Square root
`ceil`	1	Bigger integer below
`floor`	1	Smaller integer above

In the Open-PSA format, these operators are simply implemented with tags whose name is the name of the operator, e.g.

```
<mul>
  <float value="2.0"/>
  <parameter name="pumpFailureRate" />
</mul>
```

In S2ML+SBE, arithmetic operations can be written similarly to the Open-PSA format, or in the usual way. Assume for instance that A, B and C are three parameters. The expression $A \times B + (1 - A) \times C$ can be written in two ways as follows.

```
add(mul(A, B), mul(sub(1, A), C))
A*B + (1-A)*C
```

Figure 6.3 gives the EBNF grammar for arithmetic expressions in S2ML+SBE.

```
ArithmeticExpression ::=
      ArithmeticExpression ( '+' ArithmeticExpression )+
   |  ArithmeticExpression '-' ArithmeticExpression
   |  ArithmeticExpression ( '*' ArithmeticExpression )+
   |  ArithmeticExpression '/' ArithmeticExpression
   |  ArithmeticOperator '('
         ArithmeticExpression (',' ArithmeticExpression)+ ')'

ArithmeticOperator ::=
   'neg' | 'add' | 'sub' | 'mul' | 'div' |
   'abs' | 'exp' | 'log' | 'log10' | 'mod' | 'pow' | 'sqrt' |
   'ceil' | 'floor'

TrigonometricExpression ::=
   TrigonometricOperator '(' ArithmeticExpression ')'

TrigonometricOperator ::=
   'sin' | 'cos' | 'tan' |
   'asin' | 'acos' | 'atan' |
   'sinh' | 'cosh' | 'tanh'

AggregationExpression ::=
   Aggregator '(' ArithmeticExpression (',' ArithmeticExpression)+ ')'

Aggregator ::=
   'min' | 'max' | 'mean'
```

Figure 6.3: EBNF S2ML+SBE grammar for arithmetic, trigonometric and aggregation expressions

As illustrated by the above code, usual priorities of operators are applied, i.e. the unary minus - has a higher priority than the division /, which has a higher priority than the multiplication *, which has a higher priority than the subtraction −, which has a higher priority that the addition +.

Table 6.3 gives the priorities of S2ML+SBE operators, in decreasing order.

6.3.3 Trigonometric Functions

In addition to arithmetic operations, XFTA implements regular *trigonometric functions*. Table 6.4 gives their list.

6.3.4 Aggregation Functions

XFTA implements *aggregation functions*. Table 6.5 gives their list.

6.3.5 Boolean Expressions

XFTA implements the usual *Boolean expressions*, with essentially the same syntax as for Boolean formulas associated with flow variables. Table 6.6 gives the list of Boolean connectives. Priorities

Table 6.3: Priorities of S2ML+SBE operators, in decreasing order

Priority	Operator	Semantics
9	`or`	Boolean disjunction
8	`and`	Boolean conjunction
7	`not`	Boolean negation
6	==, ! =, <, >, <=, >=	Inequalities
5	+	Addition
4	-	Subtraction
3	*	Multiplication
2	/	Division
1	-	Unary minus (`neg`)
0	all others	

Table 6.4: Trigonometric functions

Tag	Arity	Semantics
`sin`	1	Sine
`cos`	1	Cosine
`tan`	1	Tangent
`asin`	1	Arc sine
`acos`	1	Arc cosine
`atan`	1	Arc tangent
`sinh`	1	Hyperbolic sine
`cosh`	1	Hyperbolic cosine
`tanh`	1	Hyperbolic tangent

of these connectives are recalled Table 6.3

The EBNF grammar of S2ML+SBE Boolean expressions is given in Figure 6.5.

6.3.6 Inequalities

XFTA implements *inequalities*. Table 6.7 describes the corresponding operators.

In S2ML+SBE, it is possible to write inequalities in their usual form, using symbols “==” (equal), “!=” (different), “<” (less than), “>” (greater than), “<” (less or equal to), and “>=” (greater or equal to).

- `A==B` is equivalent to `eq(A, B)`.
- `A!=B` is equivalent to `df(A, B)`.
- `A<B` is equivalent to `lt(A, B)`.
- `A>B` is equivalent to `gt(A, B)`.
- `A<=B` is equivalent to `leq(A, B)`.
- `A>=B` is equivalent to `geq(A, B)`.

The EBNF grammar of S2ML+SBE inequalities is given in Figure 6.5.

6.3.7 Conditional Expressions

Finally, XFTA implements *conditional expressions*. Both the Open-PSA Format and S2ML+SBE provide two conditional expressions: “`if-then-else`” and “”. The former takes three arguments: the condition, the then-expression and the else-expression. The latter takes a list of pairs of expressions, introduced by the tag “case” as arguments. The first expression of the pair must be a Boolean condition. If this condition is realized, then the second expression is evaluated and its

Table 6.5: Aggregation functions

Tag	Arity	Semantics
`min`	≥ 1	Minimum value of the arguments
`max`	≥ 1	Maximum value of the arguments
`mean`	≥ 1	Mean value of the arguments

Table 6.6: Boolean connectives

Tag	Arity	Semantics
`and`	≥ 1	Conjunction
`or`	≥ 1	Disjunction
`not`	1	Negation

value returned. Otherwise, the next pair is considered. The list must end with an expression, in order to be sure that the switch has always a possible value.

Table 6.8 describes available condition expressions.

Assume that we want to give different values to a failure rate `lambda` depending on a global parameter `stressLevel`:

$$\begin{aligned}
\texttt{lambda} &= 1.00 \times 10^{-4}/h \quad \text{if} \quad \texttt{stressLevel} = 1\\
\texttt{lambda} &= 2.50 \times 10^{-4}/h \quad \text{if} \quad \texttt{stressLevel} = 2\\
\texttt{lambda} &= 1.00 \times 10^{-3}/h \quad \text{if} \quad \texttt{stressLevel} = 3
\end{aligned}$$

Using primitives defined so far, we can encode the definition of `lambda` at the Open-PSA format as shown in Figure 6.4.

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

Figure 6.4: Definition of the parameter `lambda`

S2ML+SBE has a different syntax for conditional expressions. The EBNF grammar of S2ML+SBE conditional expressions is given in Figure 6.5.

Table 6.7: Inequalities

Tag	Arity	Semantics
`eq`	2	$=$
`df`	2	$\neq$
`lt`	2	$<$
`gt`	2	$>$
`leq`	2	$\leq$
`geq`	2	$\geq$

Table 6.8: Conditional expressions

Tag	Arity	Semantics
`ite`	3	if-then-else
`switch`	≥ 1	switch
`case`	2	(condition, value) of switch

6.4 Parametric and Empirical Failure Models

6.4.1 Parametric Failure Models

Parametric failure models are widely used in reliability engineering. They provide a simple and efficient way to define time-dependent probabilities to be associated with basic events. From a modeling language perspective, they can be seen as macro-expressions that are used to simplify the writing of probability distributions.

Conversely to most of the fault tree tools, XFTA implements relatively few such parametric failure models, for two reasons. First, it provides constructs to "program" parametric failure models at will. Consequently, only the most frequently models need to be implemented, for the sake of convenience. Second, it provides, via empirical distributions, an efficient mechanism to embed any distribution, up to minor approximations due to linear interpolation. More models could however be easily added.

In the sequel, we shall give, for each parametric failure model, the unavailability Q_t of the component at time t. All these models take implicitly the mission time t as a parameter. t is set up via the calculation engine (it is accessible via the pseudo-constant `mission-time`).

⚠ **Parametric failure models**
Some simplifications and changes have been made in parametric failure models descriptions from the version 2.0.0 of XFTA. In particular, the mission time is now preferably kept implicit. However, parametric distributions `exponential`, `GLM` and `Weibull` are available under the two forms, i.e. with either implicit or explicit mission times.

Table 6.9 lists the parametric failure models implemented in the current version of XFTA. The second column gives the arity (or possible arities) of the distribution, i.e. its number of arguments.

In both the Open-PSA format and in S2ML+SBE, failure models obey the same syntactic rules as arithmetic functions like `exp`, `log`, `pow`...Parameters are given in order.

6.4.1.1 Point Estimate Probabilities

Strictly speaking, point estimate probabilities are not probability distributions. They give a certain probability p invariant through the time to the considered event.

Formally, a *point estimate probability* is thus a constant probability distribution function:

$$Q_t \overset{def}{=} p \tag{6.3}$$

```
BooleanExpression ::=
      StochasticExpression ('or' StochasticExpression)+
   |  StochasticExpression ('and' StochasticExpression)+
   |  'not' StochasticExpression

Inequality ::=
   StochasticExpression ('==' | '!=' | '<' | '>' | '<=' | '>=')
       StochasticExpression

ConditionalExpression ::=
      IfThenElseExpression
   |  SwitchExpression

IfThenElseExpression ::=
   'if' BooleanExpression 'then' StochasticExpression 'else'
       StochasticExpression

SwitchExpression ::=
   'switch' '{'
      ('case' BooleanExpression ':' StochasticExpression ',')+
      'else' StochasticExpression
      '}'
```

Figure 6.5: EBNF S2ML+SBE grammar for Boolean and conditional expression

Table 6.9: Parametric failure models

Tag	Arities	Arguments
`exponential`	1, 2	Failure rate, mission-time (optional)
`GLM`	3, 4	Probability of failure on demand, failure rate, repair rate, mission-time (optional)
`Weibull`	3, 4	Scale parameter, shape parameter, time shift, mission-time (optional)
`periodic-test`	3, 4, 10	See below
`failure-on-demand`	2	Probability, frequency

Point estimate probabilities are widely used in probabilistic risk analyses. Nuclear probabilistic risk analyses for instance make a nearly exclusive use of point estimate probabilities.

In both the Open-PSA format and S2ML+SBE, it suffices to give directly the value p, which can be any stochastic expression.

6.4.1.2 Exponential Distribution

The exponential distribution also called negative exponential distribution, is by far the most widely used distribution in time-dependent analyses.

An *exponential distribution* of *rate* (or *transition rate*), λ, where λ is a positive real number, is a cumulative distribution function verifying:

$$Q_t \overset{def}{=} 1 - e^{-\lambda \times t} \tag{6.4}$$

Figure 6.6 shows exponential distributions for different values of the parameter λ.

The exponential distribution characterizes the *Markovian hypothesis* that the system under study is memoryless.

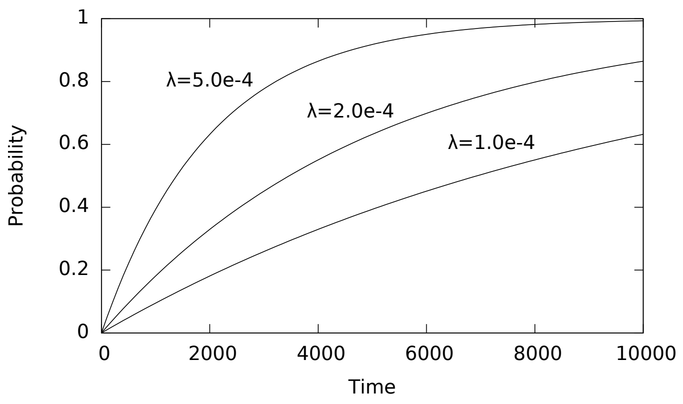


Figure 6.6: Exponential distributions for different values of the parameter λ

It is widely used in probabilistic risk analyses because, in absence of further information, it is reasonable to assume a constant failure rate. Moreover analytical calculations are much, much easier if exponential distributions are assumed. In authors experience, it is often the main justification, although unavowed, for the use of this distribution.

The mean time to failure of a system or a component obeying an exponential distribution of parameter λ is $1/\lambda$. Reciprocally, if the observed mean time to failure of a system or a component is T, then it may be a reasonable assumption to associate this system or this component with an exponential distribution of parameter $\lambda = 1/T$.

The Open-PSA format expression encoding an exponential distribution with a failure rate `lambda` is a follows.

```
<exponential>
  <parameter name="lambda" />
</exponential>
```

The S2ML+SBE version of this expression is built similarly:

```
exponential(lambda)
```

With explicit mission-times, it gives for instance:

```
<exponential>
  <parameter name="lambda" />
  <mul>
    <float value="0.25" />
    <mission-time />
  </mul>
</exponential>
```

and:

```
exponential(lambda, 0.25*mission-time)
```

6.4.1.3 GLM Distribution

GLM stands for Gamma-Lambda-Mu. This distribution is an extension of the exponential distribution to represent the unavailability of repairable components. It takes three parameters: a probability of failure on demand $\gamma \in [0,1]$, a failure rate $\lambda > 0$ and a repair rate $\mu > 0$ (in this order). Its definition is as follows.

$$Q_t \overset{def}{=} \gamma \times e^{-(\lambda+\mu)\cdot t} + \frac{\lambda}{\lambda+\mu} \times \left(1 - e^{-(\lambda+\mu)\cdot t}\right) \tag{6.5}$$

The Open-PSA format expression encoding a GLM distribution with a probability of failure on demand `gamma`, a failure rate `lambda` and a repair rate `mu` is a follows.

```
<GLM>
  <parameter name="gamma" />
  <parameter name="lambda" />
  <parameter name="mu" />
</GLM>
```

The S2ML+SBE version of this expression is built similarly:

```
exponential(gamma, lambda, mu)
```

As for the exponential distribution, it is possible to give the mission time explicitly as the fourth argument.

6.4.1.4 Weibull Distribution

The exponential distribution assumes a constant transition rate. This may be not realistic in case the system or the component under study is subject to aging effects: on the one end, in the beginning of its life-time is it subject to *infant mortality*, i.e. its failure rate starts quite high and decreases until the system or the component is made free of initial problems. On the other end, after a long period of use, the system or the component may be subject to *aging and wear-out effects*, i.e. it failure rate starts increasing again. This gives raise to the famous *bath curve* of the transition rate.

The Weibull distribution is often use to represent (partly) these aging effects.

The *Weibull distribution* takes three parameters: a *scale parameter* $\alpha > 0$, a *shape parameter* $\beta > 0$, and a time shift $t_0 > 0$. is a cumulative distribution function verifying:

$$Q_t \overset{def}{=} 1 - e^{-\left(\frac{t-t_0}{\alpha}\right)^{\beta}} \tag{6.6}$$

Indeed, if $t < t_0$, then $Q_t = 0$.

When $\beta < 1$, the Weibull distribution can be used to represent infant mortality. When $\beta = 1$, it is equivalent to the exponential distribution. Finally, when $\beta > 1$, the Weibull distribution can be used to represent aging effect.

Note however that, in practice, it is rather uneasy to tune the parameters of Weibull distributions so to fit with experience feedback data.

Figure 6.7 shows Weibull distributions for different values of the parameter β ($\alpha = 2.0e4$, $t_0 = 0$).

The Open-PSA XML expression for a Weibull distribution with a scale parameter `alpha`, a shape parameter 3 and a time shift 0 is a follows.

```
<Weibull>
  <parameter name="alpha" />
  <float value="3.0" />
  <float value="0.0" />
</Weibull>
```

The S2ML+SBE version of this expression is built similarly:

```
Weibull(alpha, 3.0, 0)
```

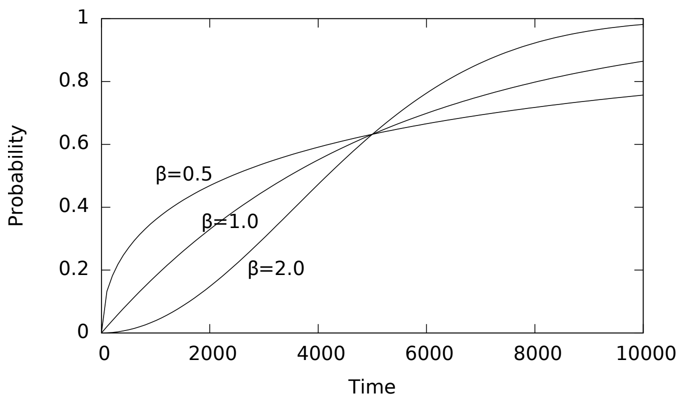


Figure 6.7: Weibull distributions for different values of the parameter β ($\alpha = 2.0e4$)

As for the exponential distribution, it is possible to give the mission time explicitely as the fourth argument.

6.4.1.5 Periodic Test Distribution

The distribution `periodic-test` is used to represent the state of periodically tested (and repaired) components. It exists under three forms, requiring respectively 3, 4 and 10 arguments.

Form with 3 arguments

The three arguments are the failure rate $\lambda > 0$, the delay $\tau > 0$ between two consecutive tests and the delay before the first test $\theta > 0$. The test is assumed to have no duration. The component is assumed as good as new after the test (or the maintenance). The delay before the first test is used to represent staggered tests. The definition of this distribution is as follows.

$$Q_t \overset{def}{=} \begin{cases} 1 - e^{-\lambda \cdot t} & \text{if} \quad t \leq \theta \\ 1 - e^{-\lambda \cdot ((t-\theta) \bmod \tau)} & \text{if} \quad t > \theta \end{cases} \tag{6.7}$$

Form with 10 arguments

The ten arguments are as follows.

1. The failure rate $\lambda > 0$ in operation.
2. The failure rate $\lambda^\star > 0$ during the test.
3. The repair rate $\mu > 0$ (once the test showed that the component is failed).
4. The delay $\tau > 0$ between two consecutive tests.
5. The delay $\theta > 0$ before the first test.
6. The probability of failure $\gamma \in [0,1]$ due to the (beginning of the) test.
7. The duration $\pi > 0$ of the test.
8. The indicator x of the component availability during the test (1 available, 0 unavailable).
9. The test covering $\sigma \in [0,1]$, i.e. the probability that the test detects the failure, if any.
10. The probability $\omega \in [0,1]$ that the component is badly restarted after a test or a repair.

The full explanation of this distribution is given Appendix B.

Form with 4 arguments

The four arguments are respectively the failure rate $\lambda > 0$, the repair rate $\mu > 0$, the delay $\tau > 0$ between two consecutive tests and the delay before the first test $\theta > 0$. This form is a restriction of the previous one, where:

– The duration of the test (π) is null, therefore the failure rate ($\lambda^\star$) during the test and the indicator of availability during the test (x) are meaningless.

– The test is assumed perfect, i.e. the probabilities γ, ω_1 and ω_2 are set to 0 and the test covering σ to 1.

The Open-PSA XML expression for a `periodic-test` distribution with a failure rate `lambda`, a delay between tests 4380 and a delay before the first test 2190 is a follows.

```
<periodic-test>
  <parameter name="lambda" />
  <float value="4380" />
  <float value="2190" />
</periodic-test>
```

The S2ML+SBE version of this expression is built similarly:

```
periodic-test(lambda, 4380, 2190)
```

6.4.1.6 Failure-on-demand Distribution

The `failure-on-demand` distribution is used in the specific context of time-dependent analyses, where unconditional failure intensities of components are involved, see Chapter 16. It takes two parameters: a constant probability of failure `q` and a constant unconditional failure intensity `w`.

The Open-PSA XML expression for a `failure-on-demand` distribution with a probability of failure 0.01, and unconditional failure intensity `w` is a follows.

```
<failure-on-demand>
  <float value="0.01" />
  <parameter name="w" />
</failure-on-demand>
```

The S2ML+SBE version of this expression is built similarly:

```
failure-on-demand(0.01, w)
```

6.4.2 Empirical Distributions

Empirical distributions are an alternative to parametric distributions. An empirical distribution consists in an ordered list of pairs made of a date and a probability. Between two successive dates, the value of the distribution is obtained by interpolation. The interpolation can be done in two ways: by considering that the probability does not change, which gives rise to stepwise distributions, or by computing a weighting mean between the two corresponding probabilities, which give rise to piecewise linear distributions.

Definition 6.4.1 — Stepwise distribution. A *stepwise distribution* consists in a (finite) list of points $(d_1, p_1), \ldots (d_n, p_n)$, $n \geq 1$ where the d_i's are increasing dates, $0 \leq d_1 < \ldots < d_n$ and the p_i's are probabilities, i.e. $0 \leq p_i \leq 1$, $i = 1, \ldots n$. It encodes the followig step function:

$$F(t) \overset{def}{=} \begin{cases} p_1 & \text{if } t < d_1 \\ p_i & \text{if } d_i \leq t < d_{i+1} \\ p_n & \text{otherwise} \end{cases}$$

Figure 6.8 shows a stepwise distribution.

Definition 6.4.2 — Piecewise linear distribution. A *piecewise linear distribution* consists in a (finite) list of points $(d_1, p_1), \ldots (d_n, p_n)$, $n \geq 1$ where the d_i's are increasing dates, $0 \leq$

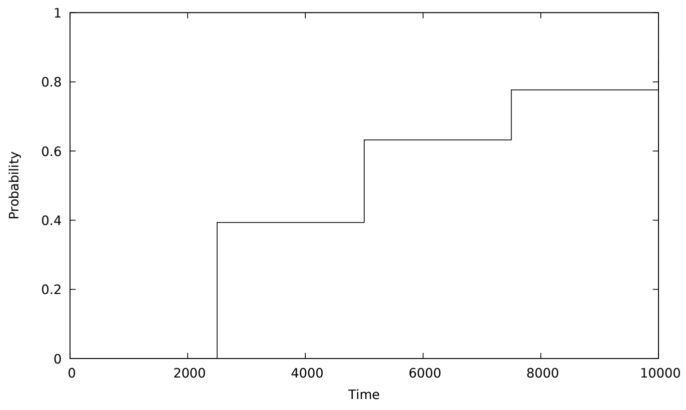


Figure 6.8: Stepwise distribution

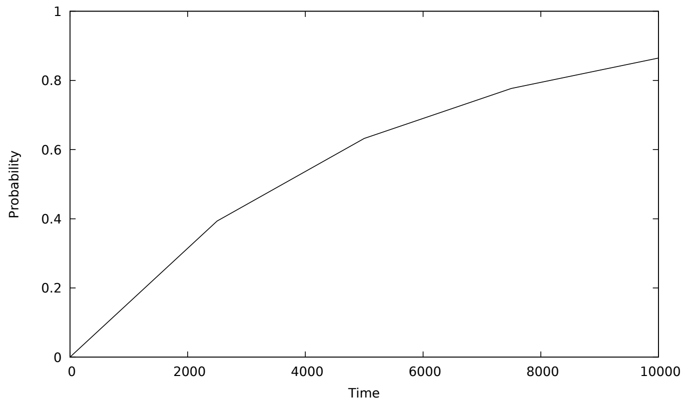


Figure 6.9: Piecewise linear distribution

$d_1 < \ldots < d_n$ and the p_i's are probabilities, i.e. $0 \leq p_i \leq 1$, $i = 1,\ldots n$. It encodes the following piecewise linear function:

$$F(t) \overset{def}{=} \begin{cases} p_1 & \text{if } t < d_1 \\ p_i + \frac{d_{i+1}-t}{d_{i+1}-d_i} \times (p_{i+1} - p_i) & \text{if } d_i \leq t < d_{i+1} \\ p_n & \text{otherwise} \end{cases}$$

Figure 6.9 shows a piecewise linear distribution.

Note that when the inverse of its transition rate λ is small compared to the mission time T, the exponential distribution is very close to a piecewise linear distribution made of a single point $(T, \lambda \times T)$.

From version 2.0.0 of XFTA, empirical distributions are described into separated text files. These text files have a very simple format (called TSV, for tabulation separated values):

– Each point is given on a line (no blank line is allowed).
– On each line, the date and the probability are separated with a tabulation character.

E.g.

```
0.0 0.0
100 0.1
1000  0.2
2000  0.05
10000 0.3
```

The Open-PSA XML expression for an `empirical-distribution` stored in the file pipe-distribution.tsv is a follows.

```
<empirical-distribution file="pipe-distribution.tsv"
   type="piecewise-linear" />
```

For stepwise distribution, the value of attribute type has to be changed to step.

The S2ML+SBE version of this expression is built similarly:

```
empirical-distribution("pipe-distribution.tsv", piecewise-linear)
```

Note that the name of the file must be surrounded with quotes.

6.4.3 Discussion

At the time we write these lines, the industry is moving from a situation where reliability data were scarce and difficult to access to a situation where data will be over-numerous and easy to access. This will induce considerable changes in the probabilistic risk assessment process, although it is admittedly still hard to see the premises of this (r)evolution.

As of today, reliability data are manually collected on the fields, then aggregated by experts (with high skills in statistics) who try to fit them into parametric distributions. The parameters of these distributions are then recorded into books like OREDA (SINTEF and NTNU 2015). Risk analysts use eventually these books to feed their models (fault trees).

This way of doing things was fine but looks now completely outdated.

First, manual processing of data will no longer be possible when these data will come from sensors continuously monitoring systems.

Second, going through books sounds weird at a time where billions and billions of digital data circulate on internet every second. Therefore, modeling environments should be soon directly connected to data bases.

Third, the main reason to use parametric distributions was they provide a compact way to store the information. However, the smallest image posted on social networks contains more information than required to describe hundreds of empirical probability distributions. Therefore, why going on using parametric distributions and not directly source data? Using directly source data would produce more accurate results. It could also make it possible to update data if not in real time, at least much more often than currently. Moreover, different treatments could be performed on data depending on the needs of the analysis.

In a word, reliability data that are used in probabilistic risk assessment are currently obtained via intermediations that will probably disappear in a near future.

6.5 Random Deviates

Random deviates are primitive to generate numbers at pseudo-random according to various distributions. They can be used in two ways: in a regular context they return a default value (typically their mean value). When used to perform sensitivity analyses via Monte-Carlo simulations (see Chapter 15), they return a number drawn at pseudo-random according their type (i.e. their distribution).

XFTA implements two types of random deviates: parametric random deviates and empirical random deviates in form of histograms.

Technically, all random deviates implemented in XFTA rely on the C++ <random> library. More specifically, on the Mersenne-Twister 64-bits random generator implemented by this library (Matsumoto and Nishimura 1998).

6.5.1 Parametric Random Deviates

Table 6.10 lists the parametric random deviates implemented in the current version of XFTA. The second column gives the arity (or possible arities) of the distribution, i.e. its number of arguments.

Table 6.10: Parametric random deviates

Tag	Arities	Arguments
`uniform-deviate`	2	lower and upper bounds
`loguniform-deviate`	2	lower and upper bounds
`normal-deviate`	2	mean and standard-deviation
`lognormal-deviate`	3	mean, error factor and confidence level
`triangular-deviate`	3	lower bound, upper bound and mode
`gamma-deviate`	2	shape and scale factors
`beta-deviate`	2	shape parameters α and β

In both the Open-PSA format and in S2ML+SBE, random deviates obey the same syntactic rules as arithmetic functions like `exp`, `log`, `pow`... Parameters are given in order.

6.5.1.1 Uniform Deviate

The *continuous uniform distribution*, or simply *uniform distribution*, takes two parameters, a and b, $a \leq b$, which are its minimum and maximum values.

The probability density function of the uniform distribution is:

$$f(z;a,b) = \frac{1}{z \times (b-a)} \tag{6.8}$$

Its cumulative distribution function is:

$$F(z;a,b) = \frac{z-a}{b-a} \tag{6.9}$$

The mean of the uniform distribution is $\frac{1}{2}(b-a)$ (it is used as default value), its median $\frac{1}{2}(b-a)$ and its variance $\frac{1}{12}(b-a)^2$.

The Open-PSA XML expression for a `uniform-deviate` with a lower bound 0.01 and an upper bound 0.02 is a follows.

```
<uniform-deviate>
  <float value="0.01" />
  <float value="0.02" />
</uniform-deviate>
```

The S2ML+SBE version of this expression is built similarly:

```
uniform-deviate(0.01, 0.02)
```

Figure 6.10 shows the shape the cumulative distribution function of the uniform distribution for the above values of its parameters.

6.5.1.2 Loguniform Deviate

The *loguniform distribution*, also known as reciprocal distribution, is characterized by its probability density function, within the support of the distribution, being proportional to the reciprocal of the variable. It takes two parameters, a lower bound a and an upper bound b, $0 < a < b$.

The probability density function of the loguniform distribution is:

$$f(z;a,b) = \frac{1}{z \cdot (\ln(b) - \ln(a))} \tag{6.10}$$

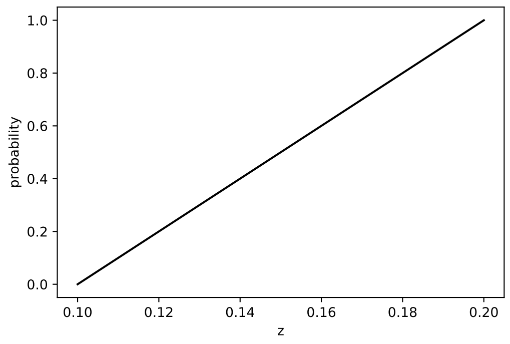


Figure 6.10: Cumulative distribution function of the uniform distribution of lower bound 0.01 and upper bound 0.02

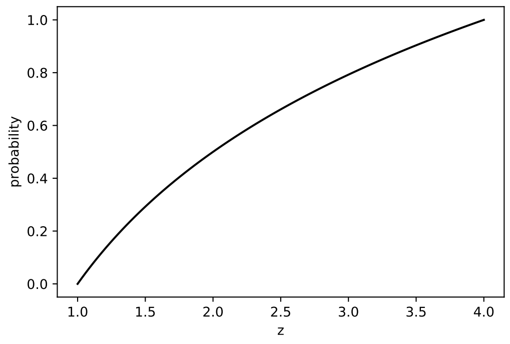


Figure 6.11: Cumulative distribution function of the loguniform distribution of lower bound 1 and upper bound 4

Its cumulative distribution function is:

$$F(z;a,b) = \frac{\ln(z)-\ln(a)}{\ln(b)-\ln(a)} \tag{6.11}$$

The default value of the loguniform deviate is set at $(a+b)/2$. When the random deviate is used to draw a number, z is picked-up uniformly at random between a and b.

The Open-PSA XML expression for a `loguniform-deviate` with a lower bound 1 and an upper bound 4 is a follows.

```
<loguniform-deviate>
  <float value="1" />
  <float value="4" />
</loguniform-deviate>
```

The S2ML+SBE version of this expression is built similarly:

```
loguniform-deviate(1, 4)
```

Figure 6.11 shows the shape the cumulative distribution function of the loguniform distribution for the above values of its parameters.

6.5.1.3 Normal Deviate

The *normal distribution*, also called (or *Gaussian* or *Gauss* or *Gauss-Laplace distribution*) is one of the most convenient to represent phenomena issued from several random sources. It is defined

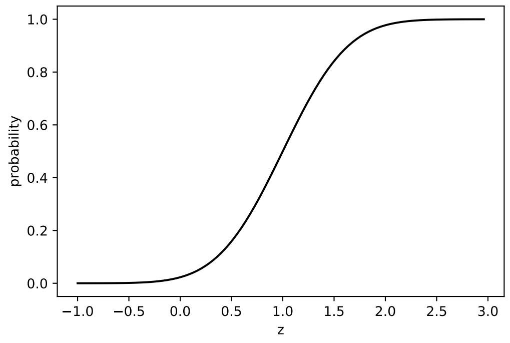


Figure 6.12: Cumulative distribution function of the normal distribution of mean 1 and standard-deviation 0.5

by means of two parameters: its mean, usually denoted as μ, and its standard-deviation, usually denoted as σ, (or equivalently its variance σ^2). It is denoted $\mathcal{N}(\mu,\sigma)$.

The probability density function of the normal distribution is:

$$f(z;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right) \tag{6.12}$$

Its cumulative distribution function is:

$$F(z;\mu,\sigma) = \frac{1}{2}\left(1+\operatorname{erf}\left(\frac{z-\mu}{\sigma\sqrt{2}}\right)\right) \tag{6.13}$$

The function $\operatorname{erf}(x)$ is the error function defined as follows.

$$\operatorname{erf}(x) \overset{def}{=} \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt \tag{6.14}$$

It gives the probability for a random variable with normal distribution of mean 0 and variance $1/2$ to fall in the interval $[-x,x]$.

The mean and median of the normal distribution are both equal to μ (this value is used as default value), and its variance σ^2.

The Open-PSA XML expression for a `normal-deviate` with a mean 1 and a standard-deviation 0.5 is a follows.

```
<normal-deviate>
  <float value="1" />
  <float value="0.5" />
</normal-deviate>
```

The S2ML+SBE version of this expression is built similarly:

```
normal-deviate(1, 0.5)
```

Figure 6.12 shows the shape the cumulative distribution function of the normal distribution for the above values of its parameters.

6.5.1.4 Lognormal Deviate

A *lognormal distribution* is a continuous probability distribution of a random variable whose logarithm is normally distributed. Thus, if the random variable X is normally distributed, then $Y = \ln X$ has a lognormal distribution. A lognormal process is the statistical realization of the multiplicative product of many independent random variables, each of which is positive. As for the normal distribution, it is characterized by the mean μ and standard-deviation σ of X.

The probability density function of the lognormal distribution is:

$$f(z;\mu,\sigma) = \frac{1}{z\sigma\sqrt{2\pi}}\exp\left(-\frac{(\ln z-\mu)^2}{2\sigma^2}\right) \tag{6.15}$$

Its cumulative distribution function is:

$$F(z;\mu,\sigma) = \frac{1}{2}\left(1+\operatorname{erf}\left(\frac{\ln z-\mu}{\sigma\sqrt{2}}\right)\right) \tag{6.16}$$

However, it is often more convenient to define lognormal distributions directly. This is the choice made in XFTA, where lognormal distributions are characterized by their mean, their error factor and the confidence level of this error factor.

In terms of μ and σ, the mean of a lognormal distribution is:

$$E[z] = \exp\left(\mu+\frac{\sigma^2}{2}\right) \tag{6.17}$$

The confidence interval $[X_{0.05}, X_{0.95}]$ associated with a 0.95 confidence level and the median $X_{0.50}$ are the following:

$$X_{0.05} = \exp(\mu - 1.645\sigma) \tag{6.18}$$

$$X_{0.95} = \exp(\mu + 1.645\sigma) \tag{6.19}$$

$$X_{0.50} = \sqrt{X_{0.05}\times X_{0.95}} = \exp(\mu) \tag{6.20}$$

The 0.95 error factor is then defined as follows.

$$EF_{0.95} = \sqrt{\frac{X_{0.95}}{X_{0.05}}} = \exp(1.645\sigma) \tag{6.21}$$

It is possible to proceed the reverse way to obtain μ and σ from $E[z]$ and $EF_{0.95}$ as follows.

$$\sigma = \frac{\log EF_{0.95}}{1.645}$$

$$\mu = \log E[z] - \frac{\sigma^2}{2}$$

Other confidence levels are available. They correspond to other constants, called standard scores, than 1.645. Table 6.11 reviews them.

The Open-PSA XML expression for a `lognormal-deviate` with a mean 0.001 and an error factor 3 and 0.99 confidence level is a follows.

```
<lognormal-deviate>
  <float value="0.001" />
  <float value="3" />
  <float value="0.99" />
</lognormal-deviate>
```

The S2ML+SBE version of this expression is built similarly:

```
lognormal-deviate(0.001, 3, 0.99)
```

Table 6.11: Available confidence levels and their standard scores

Confidence level	Standard scores
0.50000	0.6745
0.68269	1.0000
0.90000	1.6449
0.95000	1.9599
0.95450	2.0000
0.99000	2.5759
0.99730	3.0000
0.99900	3.2905
0.99990	3.8906
0.99993	4.0000
0.99999	4.4172

6.5.1.5 Triangular Deviate

The *triangular distribution* takes two parameters, a lower bound a, an upper bound b and a mode c, $a \leq c \leq b$.

The triangular distribution is typically used as a subjective description of a population for which there is only limited sample data. It is therefore often used in business decision making when not much is known about the distribution of an outcome (say, only its smallest, largest and most likely values).

The probability density function of the loguniform distribution is:

$$f(z;a,b,c) = \begin{cases} \frac{2(z-a)}{(b-a)(c-a)} & \text{if } a \leq z \leq c \\ \frac{2(b-z)}{(b-a)(b-c)} & \text{if } c < z \leq b \end{cases} \tag{6.22}$$

Its cumulative distribution function is:

$$F(z;a,b,c) = \begin{cases} \frac{(z-a)^2}{(b-a)(c-a)} & \text{if } a < z \leq c \\ 1 - \frac{(b-z)^2}{(b-a)(b-c)} & \text{if } c < z < b \end{cases} \tag{6.23}$$

The mean of the triangular distribution is $\frac{a+b+c}{3}$ (it is used as default value). When the random deviate is used to draw a number, z is picked-up uniformly at random between a and b.

The Open-PSA XML expression for a `triangular-deviate` with a lower bound 0.01, an upper bound 0.02 and a mode 0.0125 is a follows.

```
<triangular-deviate>
  <float value="0.01" />
  <float value="0.02" />
  <float value="0.0125" />
</triangular-deviate>
```

The S2ML+SBE version of this expression is built similarly:

```
triangular-deviate(0.01, 0.02, 0.0125)
```

Figure 6.13 shows the shape the cumulative distribution function of the triangular distribution with the parameters given above.

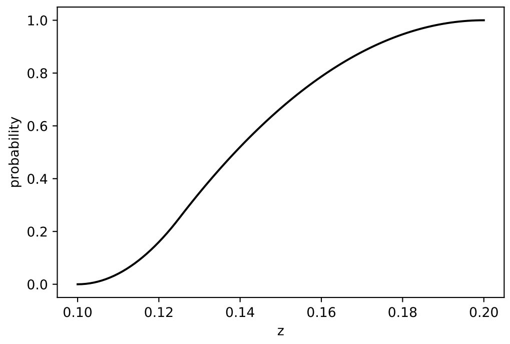


Figure 6.13: Cumulative distribution function of the triangular distribution of lower bound 0.01, upper bound 0.02 and mode 0.0125

6.5.1.6 Gamma Deviate

The gamma distribution is parameterized by a shape parameter k and a scale parameter θ.

The probability density function of the gamma distribution is:

$$f(z;k,\theta) \quad = \quad \frac{1}{\Gamma(k)\theta^k} z^{k-1} e^{-\frac{z}{\theta}} \tag{6.24}$$

where Γ is the gamma function, which defined via a convergent improper integral:

$$\Gamma(z) \quad = \quad \int_0^{+\infty} x^{z-1} e^{-x}\, dx$$

For all positive integer n, $\Gamma(n) = (n-1)!$.

The cumulative distribution function of the gamma distribution is:

$$F(z;k,\theta) \quad = \quad \frac{1}{\Gamma(k)} \gamma\left(k, \frac{z}{\theta}\right) \tag{6.25}$$

where γ is the lower incomplete gamma function, which defined via a convergent improper integral:

$$\gamma(s,x) \quad = \quad \int_0^{x} t^{s-1} e^{-t}\, dt$$

The Open-PSA XML expression for a `gamma-deviate` with shape parameter $k = 7.5$ and scale parameter $\theta = 1$ is a follows.

```
<gamma-deviate>
  <float value="7.5" />
  <float value="1" />
</gamma-deviate>
```

The S2ML+SBE version of this expression is built similarly:

```
gamma-deviate(7.5, 1)
```

6.5.1.7 Beta Deviate

The beta distribution takes two shape parameters, denoted by $\alpha > 0$ and $\beta > 0$ that appear as exponents of the random variable and control the shape of the distribution.

The probability density function of the beta distribution is:

$$f(z;\alpha,\beta) \quad = \quad \frac{z^{\alpha} \times (1-z)^{\beta}}{B(\alpha,\beta)} \tag{6.26}$$

where $B(\alpha,\beta)$ is defined as follows.

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha,\beta)}$$

The cumulative distribution function of the beta distribution is:

$$F(z;\alpha,\beta) = \int_0^z t^{\alpha-1} \times (1-t)^{\beta-1}\, dt \tag{6.27}$$

The Open-PSA XML expression for a `beta-deviate` with shape parameters $\alpha = 2$ and $\beta = 5$ is a follows.

```
<beta-deviate>
  <float value="2" />
  <float value="5" />
</beta-deviate>
```

The S2ML+SBE version of this expression is built similarly:

```
beta-deviate(2, 5)
```

6.5.2 Histograms

Histograms are to random deviates what empirical distributions are to failure models. A histogram consists in an ordered list of pairs made of a probability and a value. Between two successive probabilities, the value of the distribution is obtained by linear interpolation.

Definition 6.5.1 — Histogram. A *histogram* consists in a (finite) list of points $(p_1,v_1),\ldots(p_n,v_n)$, $n \geq 1$ where the p_i's are increasing probabilities, $0 \leq p_1 < \ldots < p_n \leq 1$ and the v_i's are any numerical values. It encodes the following piecewise linear function.

$$F(z) \overset{def}{=} \begin{cases} v_1 & \text{if } z < p_1 \\ v_i + \frac{p_{i+1}-z}{p_{i+1}-p_i} \times (v_{i+1}-v_i) & \text{if } p_i \leq z < p_{i+1} \\ v_n & \text{otherwise} \end{cases}$$

From version 2.0.0 of XFTA, histograms are described into separated text files. These text files have a very simple format (called TSV, for tabulation separated values):

– Each point is given on a line (no blank line is allowed).
– On each line, the probability and the value are separated with a tabulation character.

E.g.

```
0.0	0.0
0.1	100
0.2	1000
0.3	10000
0.4	1000
0.5	100
```

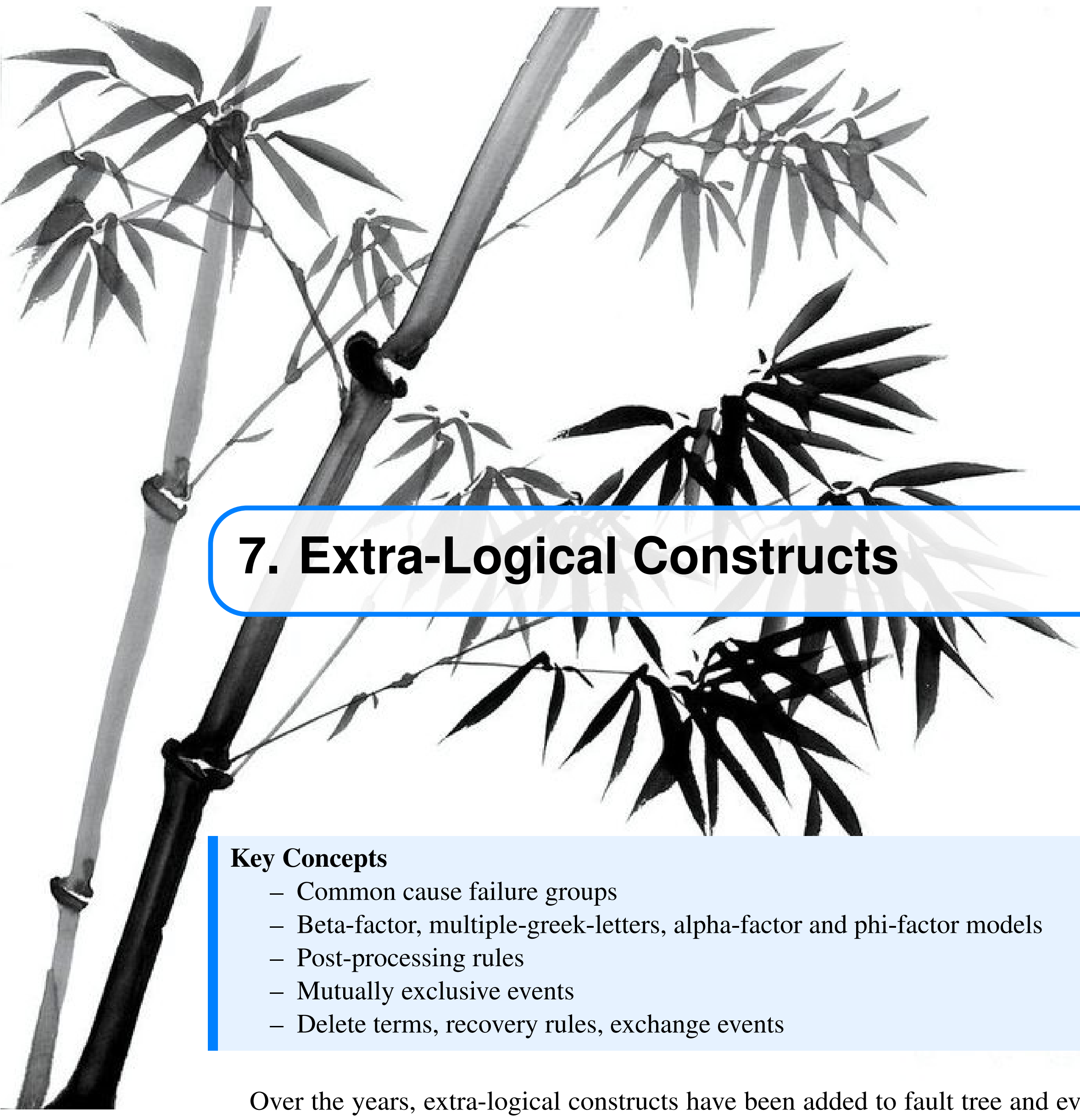
The Open-PSA XML expression for a `histogram` stored in the file `pipe-histogram.tsv` is a follows.

```
<histogram file="pipe-histogram.tsv" />
```

The S2ML+SBE version of this expression is built similarly:

```
histogram("pipe-histogram.tsv")
```

Note that the name of the file must be surrounded with quotes.



7. Extra-Logical Constructs

Key Concepts
- Common cause failure groups
- Beta-factor, multiple-greek-letters, alpha-factor and phi-factor models
- Post-processing rules
- Mutually exclusive events
- Delete terms, recovery rules, exchange events

Over the years, extra-logical constructs have been added to fault tree and event tree models in order to “simplify” the task of analysts. We put here simplify within quotes because these constructs are *ad-hoc* and tend to complexify significantly the semantics of models, and even more problematically, to make it tool-dependent. The extra-logical constructs implemented in tool *A* are not implemented in tool *B*, or they are, but with a different meaning. This hampers the portability of models and consequently the opportunities of cross-verification and peer reviews. Eventually, this degrades the confidence we can have in models. Nevertheless, as these constructs exist, the Open-PSA (and S2ML+SBE) format provides a way to encode them.

There are essentially two types of extra-logical constructs:
- Common cause failure groups;
- Post-processing rules for minimal cutsets.

The current version of XFTA implements the former, but not the latter. We shall see why and how to deal the latter.

7.1 Common Cause Failure Groups

7.1.1 Introduction

One of the basic assumptions of the fault tree technique is that basic events are statistically independent. However, some models include so-called common cause failure groups. *Common cause failure groups*, CCF groups for short, are groups of failure modes that may occur together due to a common cause, e.g. a shock, an environmental hazard, a manufacturing defect or an operator error.

Consider for instance a system that involves 3 calculators A, B and C. Assume that these calculators may fail independently or due to a common cause, e.g. strong electromagnetic fields, or repeated overheating within the technical room they are located in. The events (in the sense of probability theory, see Appendix A) are the following:

– The individual failures of the calculators A, B and C. In the sequel, we shall denote them respectively As, Bs and Cs.
– The failure of two or more of the calculators A, B and C, due to the common cause. In the sequel, we shall denote them respectively AB, AC, BC and ABC.

The idea behind the declaration of common cause failure groups is to transform automatically the model in which A, B and C are basic events into a model where they are internal events defined as the disjunction of basic events in which they occur:

$$\begin{aligned} \mathtt{A} &= \mathtt{As} \vee \mathtt{AB} \vee \mathtt{AC} \vee \mathtt{ABC} \\ \mathtt{B} &= \mathtt{Bs} \vee \mathtt{AB} \vee \mathtt{BC} \vee \mathtt{ABC} \\ \mathtt{C} &= \mathtt{Cs} \vee \mathtt{AC} \vee \mathtt{BC} \vee \mathtt{ABC} \end{aligned}$$

This principle generalizes to any number of components.

Let us look at the logical consequences of the above expansion.

Consider a minimal cutset $\mathtt{A} \cdot \pi$ of the model before expansion (such that neither B, nor C show up in π). This minimal cutset gives rise to 4 minimal cutsets in the expanded model: $\mathtt{As} \cdot \pi$, $\mathtt{AB} \cdot \pi$, $\mathtt{AC} \cdot \pi$ and $\mathtt{ABC} \cdot \pi$. This is fine, at least if the probabilities of the four basic events add up to the expected probability of A.

Now consider a minimal cutset $\mathtt{A} \cdot \mathtt{B} \cdot \rho$ of the model before expansion (such that C does not show up in ρ). Potentially, this minimal cutset gives rise to $4 \times 4 = 16$ cutsets in the expanded model. However, not all of these cutsets are minimal. The minimal ones are: $\mathtt{As} \cdot \mathtt{Bs} \cdot \rho$, $\mathtt{As} \cdot \mathtt{BC} \cdot \rho$, $\mathtt{AB} \cdot \rho$, $\mathtt{AC} \cdot \mathtt{Bs} \cdot \rho$, $\mathtt{AC} \cdot \mathtt{BC} \cdot \rho$ and $\mathtt{ABC} \cdot \rho$. The minimal cutset $\mathtt{As} \cdot \mathtt{BC} \cdot \rho$ may correspond to situations where the calculator A fails first, due to some internal reason, then both calculators B and C fail, due to the common cause. The minimal cutset $\mathtt{AC} \cdot \mathtt{BC} \cdot \rho$ is awkward: it assumes that calculators A and C on the one hand, B and C on the other hand fail for some common causes, but that these common causes are not the same (because this would mean that ABC occurs).

The case of a minimal cutset $\mathtt{A} \cdot \mathtt{B} \cdot \mathtt{C} \cdot \sigma$ is similar. It gives rise the following minimal cutsets: $\mathtt{As} \cdot \mathtt{Bs} \cdot \mathtt{Cs} \cdot \sigma$, $\mathtt{As} \cdot \mathtt{BC} \cdot \sigma$, $\mathtt{Bs} \cdot \mathtt{AC}.\sigma$, $\mathtt{Cs} \cdot \mathtt{AB} \cdot \sigma$, $\mathtt{AB} \cdot \mathtt{AC} \cdot \sigma$, $\mathtt{AB} \cdot \mathtt{BC} \cdot \sigma$, $\mathtt{AC} \cdot \mathtt{BC} \cdot \sigma$, and $\mathtt{ABC} \cdot \sigma$. Here again, interpreting the minimal cutset $\mathtt{AB} \cdot \mathtt{AC} \cdot \sigma$ is rather tricky.

The problem of interpreting minimal cutsets gets even trickier with common groups involving more basic events. This is the reason why it is assumed implicitly that no minimal cutsets of the model (before expansion) contains two or more events of the common cause group. Another solution consists in considering only individual failures of components and the failure of all components together.

It remains thus to associate probability distributions to the newly generated basic events. This is achieved by means of so-called *common cause failure models*, which are methods to distribute the probability of failure of the component onto the newly created basic events, i.e. eventually to associate a weight with each newly created basic event.

7.1.2 Models

The reliability engineering literature, see e.g. (Kumamoto and Henley 1996), proposes mainly three common cause failure models: the beta-factor model, the multiple Greek letters (MGL) model and the alpha-factor model. The two last ones differ only from the way the factors for each level (2 components failed, 3 components failed...) are defined. The OpenPSA format proposes in addition the so-called phi-factor model, which provides a more direct way to set factors.

7.1.2.1 Beta-Factor Model

The *beta-factor* model assumes that if a common cause failure occurs then all components of the group fail simultaneously. Components can nevertheless fail independently. Multiple independent failures are neglected. The beta-factor model assumes moreover that all of 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 `BE1`, `BE2`,... `BEn` be the n events of a common cause failure group with a probability distribution $Q(t)$ and a beta-factor β. Applying the beta-factor model on the fault tree consists in following operations.

1. Create new basic events `BEis` for each event `BEi`, $1 \leq i \leq n$, to represent the independent occurrences of `BEi`'s and a basic event `BEccf` to represent the occurrence of all `BEi`'s together.
2. Redefine each `BEi` as an intermediate event whose definition is $\texttt{BEi} = \texttt{BEis} \vee \texttt{BEccf}$.
3. Associate the probability distributions $(1-\beta) \times Q(t)$ with each `BEis` and $\beta \times Q(t)$ with `BEccf`.

7.1.2.2 Multiple Greek Letters Model

The *multiple Greek letters* (MGL) model generalizes the beta-factor model. It considers in addition the cases where sub-groups of 2..., $n-1$ components of the group fail together. This model is characterized by the probability distribution of failure of the components (which is again assumed to be the same for all components), and $n-1$ factors $\rho_2, \ldots \rho_n$. The factor ρ_k denotes the conditional probability that k components of the group are failed given that $k-1$ are failed.

Let `BE1`, `BE2`,... `BEn` be the n events of a common cause failure group, let $Q(t)$ the probability distribution of the group, and let $\rho_2, \ldots \rho_n$ its factors. 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. Redefine each event `BEi` as the disjunction of the newly created basic events that represent a group that contains `BEi` (as shown in the previous section).
3. Associate the following probability distribution with each newly created basic event representing a group of k components (with $\rho_{n+1} = 0$).

$$Q_k(t) \overset{def}{=} \frac{1}{\binom{n-1}{k-1}} \times \left(\prod_{i=2}^{k} \rho_i \right) \times (1 - \rho_{k+1}) \times Q(t) \tag{7.1}$$

Consider for instance a group of 4 events: `A`, `B`, `C` and `D`, the event `A` is redefined into the intermediate event:

$$\texttt{A} = \texttt{As} \vee \texttt{AB} \vee \texttt{AC} \vee \texttt{AD} \vee \texttt{ABC} \vee \texttt{ABD} \vee \texttt{ACD} \vee \texttt{ABDC}$$

The Q_k's are as follows.

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

7.1.2.3 Alpha-Factor Model

The *alpha-factor* model is the same as the MGL model except in the way the factors are given. Here n factors α_1, α_2,... α_n are given. The factor 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.

$$Q_k(t) \overset{def}{=} \frac{1}{\binom{n-1}{k-1}} \times \frac{\alpha_k}{\sum_{i=1}^{n} \alpha_i} \times Q(t) \tag{7.2}$$

7.1.2.4 Phi-Factor Model

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

The distribution associated with a group of size k is as follows.

$$Q_k(t) \overset{def}{=} \varphi_k \times Q(t) \tag{7.3}$$

Indeed, the φ_j's factor should sum to 1 (over the different basic events).

7.1.3 Encoding

The description of a common failure group consists of five elements:

1. The name of the group.
2. The common cause model chosen for the group.
3. The members of the group.
4. The factors, i.e. the parameters of the common cause model.
5. The probability distribution that serves as a basis.

The declarations of common cause failure groups follow the same pattern as other declarations in both the Open-PSA format and S2ML+SBE.

⚠ **Declaration of members of a common cause failure group**
Although they are eventually redefined as internal events, members of a common cause failure group must be declared as basic events and associated a (fake) probability distribution.

7.1.3.1 Open-PSA Format

The general form of declarations of common cause failure groups at the Open-PSA format is as follows.

```
<define-CCF-group name="group name" model="model name" >
 <members> members-of-the-group </members>
 <factors> parameters-of-the-model </factors>
 probability-distribution
</define-CCF-group>
```

The declaration is thus introduced by the tag `define-CCF-group` with the attributes `name` (that obeys usual rules regarding identifiers) and `model`. The latter must one of the available models, i.e. either `beta-factor`, `MGL`, `alpha-factor` or `phi-factor`. Members of the group must be references to basic events. The factors and the probability distribution must be stochastic expressions.

Members, factors and probability distribution must be given in this order. Factors must also be given in ascending order of levels, e.g. ρ_2, ρ_3,... ρ_n for the MGL model of a group of n events. The number of factors depend on the model.

Figure 7.1 shows the declaration of a common cause failure group at the Open-PSA format.

```
<define-CCF-group name="Calculators" model="MGL" >
  <members>
    <basic-event name="AFailed" />
    <basic-event name="BFailed" />
    <basic-event name="CFailed" />
    <basic-event name="DFailed" />
  </members>
  <factors>
    <float value="0.10" />
    <float value="0.20" />
    <float value="0.30" />
  </factors>
  <exponential>
    <parameter name="lambda" />
  </exponential>
</define-CCF-group>
```

Figure 7.1: Declaration of a common cause failure group at Open-PSA format

```
CommonCauseFailureGroupDeclaration ::=
   'CCFGroup' Path '=' CCFModel CCFMembers CCFFactors
       StochasticExpression ';'

CCFModel ::=
   'beta-factor' | MGL' | 'alpha-factor' | 'phi-factor'

CCFMembers ::=
   '[' BasicEventReference (',' BasicEventReference)* ']'

CCFFactors ::=
   '[' StochasticExpression (',' StochasticExpression)* ']'
```

Figure 7.2: EBNF grammar for S2ML+SBE declarations of common cause failure groups

Change in declarations of common cause failure groups at Open-PSA format
To make them coherent with the declarations of common cause failure groups in S2ML+SBE, declarations of common cause failure groups at Open-PSA format have been simplified. They obey now the above pattern, which is slightly different from the one of the original Open-PSA format.

7.1.3.2 S2ML+SBE

As for the other constructs, S2ML+SBE declarations of common cause failure groups follow the same pattern as declarations at the Open-PSA format. Figure 7.2 gives the EBNF grammar for these declarations.

The Open-PSA declaration of Figure 7.1 translates in S2ML+SBE as follows.

```
CCF-group Calculators = MGL
   [AFailed, BFailed, CFailed, DFailed]
   [0.10, 0.20, 0.30]
   exponential(lambda);
```

7.1.4 Implementation

XFTA implements the expansion of common cause failure groups as the first rewriting performed after the source model has been instantiated and flattened into a target model (see Chapter 10 for an overview of XFTA operation flow). The expansion applies directly on the target model.

The basic events generated by the expansion process are named after the group name. Consider again our common cause failure group.

```
CCF-group Calculators = MGL
    [AFailed, BFailed, CFailed, DFailed]
    [0.10, 0.20, 0.30]
    exponential(lambda);
```

The generated basic events are named `Calculators-`*ijkl* where *i*, *j*, *k*, *l* take the value 1 if the corresponding event is the represented subset and 0 otherwise. `Calculators-0100` represents thus the singleton {`BFailed`}, and `Calculators-1010` the pair {`AFailed`,`CCFailed`}.

The basic event `AFailed` is turned into an intermediate event whose declaration is as follows.

```
flow AFailed = Calculators-1000
    or Calculators-1100 or Calculators-1010 or Calculators-1001
    or Calculators-1110 or Calculators-1101 or Calculators-1011
    or Calculators1111;
```

Basic events are associated with probability distributions obtained according to the chosen common cause failure model. In the current version of XFTA, no simplification is performed on the resulting expressions:

```
basic-event Calculators-1000 = 1 * (1 - 0.1)
    * exponential(lambda);
basic-event Calculators-1100 = 0.333333 * 0.1 * (1 - 0.2)
    * exponential(lambda);
basic-event Calculators-1010 = 0.333333 * 0.1 * (1 - 0.2)
    * exponential(lambda);
basic-event Calculators-1001 = 0.333333 * 0.1 * (1 - 0.2)
    * exponential(lambda);
basic-event Calculators-1110 = 0.333333 * 0.1 * 0.2 * (1 - 0.3)
    * exponential(lambda);
basic-event Calculators-1101 = 0.333333 * 0.1 * 0.2 * (1 - 0.3)
    * exponential(lambda);
basic-event Calculators-1011 = 0.333333 * 0.1 * 0.2 * (1 - 0.3)
    * exponential(lambda);
basic-event Calculators-1111 = 1 * 0.1 * 0.2 * 0.3
    * exponential(lambda);
```

7.2 Post-Processing Rules for Minimal Cutsets

In this section, we shall discuss another type of extra-logical constructs, that are usually presented as post-processing rules for minimal cutsets. We shall first describe these constructs, then explain why the current version of XFTA does not implement any of them. Finally, we shall suggest some ways around.

7.2.1 Description

7.2.1.1 Delete Terms

Delete terms are groups of pairwise exclusive basic events. They are used to exclude configurations of the system under study that are impossible, due to physical or operational rules.

Typical examples of such configurations are:

– A calculator that can be either silent or outputting wrong values, but cannot indeed do both.
– Two redundant safety lines that may be unavailable either because they are failed, or because they are under maintenance, with the operational rule that the two lines are never maintained at the same time.

7.2.1.2 Recovery Rules

Recovery rules are an extension of delete terms. A *recovery rule* is a pair $(\mathscr{H}, E)$, where $\mathscr{H}$ is a set of basic events and E is a basic event. They are used to post-process minimal cutsets: if a minimal cutset π contains $\mathscr{H}$, then E is added to π.

Recovery rules are used to represent actions taken in some specific configurations to mitigate the risk (hence their name).

7.2.1.3 Exchange Events

An *exchange event (rule)* is a triple $(\mathscr{H}, E, E')$, where $\mathscr{H}$ is a set of basic events and E and E' are two basic events. Considered as a post-processing of minimal cutsets, such a rule is interpreted as follows.

If the minimal cutset π contains both $\mathscr{H}$ and E, then the basic event E' is substituted for E in the cutset.

Exchange events are also used to represent actions taken in some specific configurations to mitigate the risk, or situations in which the risk needs to be requalified. Although apparently similar to recovery rules, they turn out to be more difficult to handle.

7.2.2 Discussion

7.2.2.1 Preliminary Remarks

Post-processing rules are problematic because they rely on the assumption that all calculations of risk indicators are performed from minimal cutsets. In other words, they assume that the model is just a mean to obtain a bunch of minimal cutsets, from which real things are done. Although XFTA does actually work according to this scheme, it may be argued that this approach is too committing. For a start, it may exclude the use of alternative and promising assessment technologies such as binary decision diagrams (Rauzy 2008a). Moreover, it requires in some sense designing, therefore documenting, validating and maintaining, two models: the model from which minimal cutsets are generated and the set of post-processing rules that "correct" the generated set of minimal cutsets.

From a pragmatic point of view, it may be worth to proceed this way. The practical consequence is however that post-processing rules are *ad hoc*, tool-dependent, therefore hampering the mathematical soundness of calculations and the portability of models.

This raises three methodological and technical questions:

– Can post-processing rules be translated in terms of purely logical operations so to incorporate them seamlessly in models?
– Is there a generic notion of post-processing rule from which all particular rules currently implemented can be derived?
– Can processing rules be implemented efficiently on large models and huge sets of minimal cutsets?

7.2.2.2 Handling Exclusive Events

Conversely to the assessment algorithms implemented in most of the other freely or commercially available tools, the assessment algorithm of XFTA is fully and efficiently able to handle formulas with negations (non coherent models). This point will be discussed in further details Chapter 11.

As a consequence, there is no need to introduce extra-logical constructs to deal mutually exclusive events. It suffices to declare it in the model, by purely logical means. The principle is as follows.

Let *M* be a fault tree model and let `T` be the top event and finally let `A` and `B` be two mutually exclusive events of *M* (we do not require `A` and `B` to be basic events).

If we know, by construction of *M*, that `A` and `B` never occur together in a minimal cutset, then we can, at least as a first, possibly over pessimistic, approximation, simply ignore the fact that they are mutually exclusive. Problems really show up if they may occur together in a minimal cutset. In that case, it suffices to create a new flow variable `C`, representing the constraint that `A` and `B` are mutually exclusive, and a new top variable `T2` and-ing the previous one with that variable:

```
flow C = atmost 1 (A, B);
flow T2 = T and C;
```

This method generalizes immediately to groups of any number of mutually exclusive events and to any number of such groups, making delete terms useless. This is way preferable to *ad-hoc* solutions such as the one proposed in reference (Jung and Riley 2014), which is otherwise clever.

This method generalizes also to recovery rules: let (`{A,B},E`) be such a rule for a model *M* whose top event is `T`. The constraint we want to add is that if `A` and `B` are both present in a cutset, then `E` must be in that cutset too. This can be done as follows.

```
flow C = not (A and B) or E;
flow T2 = T and C;
```

Again, this method generalizes immediately to recovery rules with an number of events (basic or not) in the hypothesis and to any number of such rules.

The situation is however different for exchange events. Exchange events are actually impossible to represent logically. There are however probably ways around. We shall not dig further this subject here.

We can provide thus only a partial answer to at least two of three questions raised in the previous section: most of post-processessing rules can be implemented efficiently by purely logical means, thanks to XFTA assessment algorithm. Consequently, there is no need to introduce additional modeling constructs. This does not mean that future versions of XFTA will not implement some extra-logical post-processessing rules. But there must strong practical motivations to justify such a development, which is not the case yet.

7.2.2.3 Illustrative Example

To illustrate our discussion, consider a coolant supply system pictured in Figure 7.3, which consists in three identical trains working in parallel. Each train consists of three successive elements: a front pipe, a valve and a main pipe.

The system is failed if at least two out of its three trains are failed. Trains can be either in operation or in maintenance. In operation, a train may fail for two reasons: a large breach in the front pipe or a large breach in the main pipe. In maintenance, it may also fail for two reasons: a small breach in the front pipe (which, due to the high pressure, leads to a rupture of the pipe) or a spurious opening of the valve. Operation rules forbid to have more than one train maintained at a time.

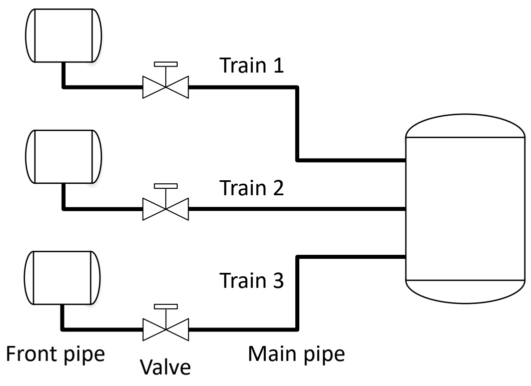


Figure 7.3: A coolant supply system

Modeling accurately such a system is quite problematic, due to the dependencies among components and functions:

– Operation and maintenance modes are exclusive one another.
– If there is a large breach in a pipe, there is *a fortiori* a small one.
– Operation rules make maintenance modes of trains exclusive one another.

Moreover, the system is actually a phased-mission system: operation and maintenance phases alternate, with different durations (maintenance phases are way shorter than operation phases).

Mathematically speaking, it does not make sense to calculate the probability of failure of the system by associating a probability or a frequency with each phase and then combining failures of the different phases: even though a risky phase has a short duration, the system goes through it with a probability 1. No one would accept to fly with an aircraft whose brakes of the landing gears have good chances to be broken, on the ground that they are used only during a tiny fraction of the flight duration.

Nevertheless, we may still want to design a fault tree model for our system, at least to get a first approximation of the risk. We shall thus try to make this approximation conservative enough so not to underestimate the risk. Our objective is then to design a model whose minimal cutsets represent the (minimal) configurations in which the system is failed, then to adjust probability distributions associated with basic events so to get the targeted approximation.

A train can potentially experience 9 different failures: small breach in the front pipe, either while in operation or while in maintenance; large breach in the front pipe, either while in operation or while in maintenance; spurious opening of the valve, only while in maintenance; small breach in the main pipe, either while in operation or while in maintenance; finally, large breach in the main pipe, either while in operation or while in maintenance.

But we are actually interested in only four of them: small breach in the front pipe while in maintenance; large breach in the front pipe while in operation; spurious opening of the valve while in maintenance; and large breach in the main pipe while in operation.

The idea is thus simply to describe locally the failure of a train as the disjunction of the above individual failures, then to describe the failure of the system as the conjunction of the failures of the trains, and finally to introduce constraints that represent mutually exclusive situation Figure 7.4 shows a model for the coolant supply system designed along this line.

The top event `CoolantSystem.failed` has 32 minimal cutsets, as expected.

Note that it is not necessary to add constraints to make exclusive, for each train, failures in operation and failures in maintenance. By construction of the model, the basic events representing these failures cannot show up simultaneously in a minimal cutset.

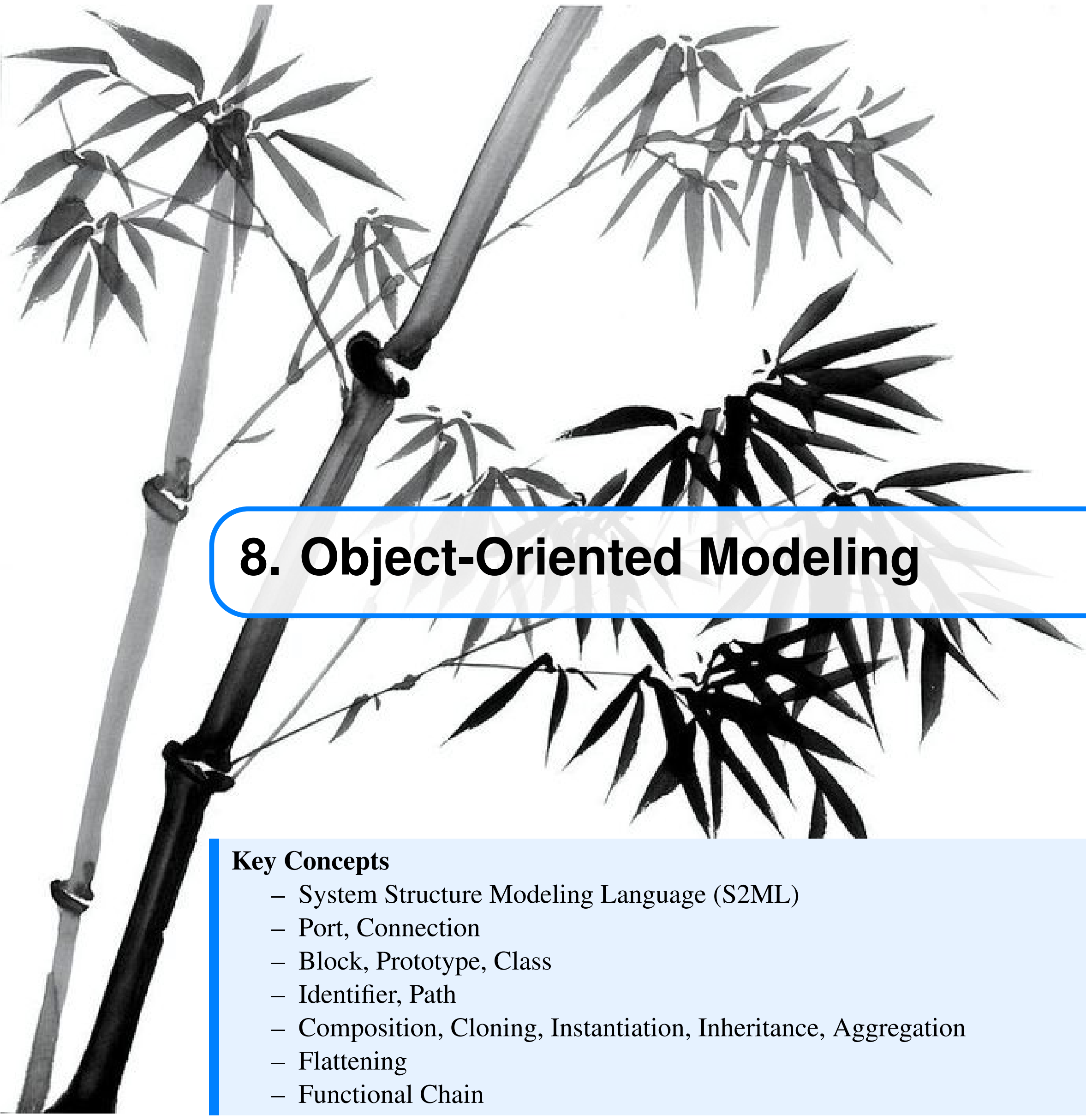
Note also that it is not necessary to consider large breaches of front pipes while in maintenance: we can adjust the probabilities associated with small breaches so that it accounts also for large

```
block CoolantSystem
   block TrainA
      flow failed = failedInOperation or failedInMaintenance;
      flow failedInOperation = _largeBreachInFrontPipe or
         _largeBreachInMainPipe;
      flow failedInMaintenance = _smallBreachInFrontPipe or
         _spuriousOpeningOfValve;
      state _smallBreachInFrontPipe = 1.0e-3;
      state _largeBreachInFrontPipe = 5.0e-4;
      state _spuriousOpeningOfValve = 3.0e-3;
      state _largeBreachInMainPipe = 8.0e-4;
   end
   clones TrainA as TrainB;
   clones TrainA as TrainC;
   flow unconstrainedFailed = atleast 2 (TrainA.failed, TrainB.failed,
      TrainC.failed);
   flow operationRule = atmost 1 (TrainA.failedInMaintenance,
      TrainB.failedInMaintenance, TrainC.failedInMaintenance);
   flow failed = unconstrainedFailed and operationRule;
end
```

Figure 7.4: A possible model for the coolant supply system

breaches.

This example concludes this chapter on extra-logical constructs and methods to handle them with XFTA. At this point, I can only repeat what I wrote before in this book and in my article with Olivier Nusbaumer (Nusbaumer and Rauzy 2013): negations and *a fortiori* extra-logical constructs must be handled with much care in Boolean reliability models. It is better not to use them, unless strictly necessary.



8. Object-Oriented Modeling

Key Concepts

– System Structure Modeling Language (S2ML)
– Port, Connection
– Block, Prototype, Class
– Identifier, Path
– Composition, Cloning, Instantiation, Inheritance, Aggregation
– Flattening
– Functional Chain

This chapter presents S2ML, the set of structuring constructs that make the input language of XFTA a full fledged object-oriented language.

8.1 Rational

Probabilistic risk assessment models such as fault trees are far from system architecture and specifications. Designing such a model requires a preliminary analysis of the system which, more than often, is captured nowhere but in the mind of the risk analyst. It is virtually impossible to retrieve an understanding of how the system works from the model. In other words, the process keeps implicit a lot of knowledge. As a consequence, models are hard to design, but even harder to share with stakeholders and to maintain through the life-cycle of systems.

Many dream thus to derive automatically probabilistic risk assessment models for models designed by systems designers, possibly by decorating the latter with some risk related information. In this way, not only the engineering process would be fluidified, but the information could be centralized. Unfortunately, this dream has little chance of success, if any. It is even counterproductive. First, although both types of models aim at representing the architecture of the system under study, their purposes are very different. Models designed by systems engineers are in general pragmatic while models designed by reliability engineers are almost exclusively formal, see (Rauzy and Haskins 2019) for a precise definition of these adjectives and a thorough discussion. Passing from pragmatic notations to formal models requires more than automated model transformation techniques can offer. Second, due to computational complexity of the calculation of performance indicators, probabilistic risk assessment models result always from a trade-off between the accuracy of the description and the tractability of the calculations. This tradeoff is often obtained by means of an iterative process. No such issue exists for communication-oriented models such as those of systems designers. Third, the system design process and the risk analysis process go at different paces and with different requirements (in terms of deliverables). Models tend thus to have different frontiers and maturity levels even thought they describe the same system at the same time. As a consequence, there is no such a thing as a one-to-one correspondence between the architecture of the system as seen by the system designers and the architecture of the system as seen by reliability engineers. Trying to merge models of both and just overload them and introduce of lot a bureaucratic work for very little results, if any.

In a word, the diversity of models is irreducible and we should live with.

Hence the need for object-oriented probabilistic risk assessment models. These models would present a number of advantages compared to current models:

– As they represent the architecture of systems, they are easier to share among stakeholders.
– They are also easier to synchronize with models of other system engineering disciplines, including of course system design.
– They make explicit at least part of the knowledge that is kept implicit by risk assessment models.
– They are easier to debug than lower level models written in lower level formalisms for it is easier to simulate them and to understand "what's going on".
– The same model can be used for several safety goals (risk assessment models tend to be specialized for a unique safety goal).

It remains that the design of such modeling formalisms, of the associated assessment tools and modeling methodologies is easier said than done.

Preliminary works in this direction started in France at the beginning of the nineties with the development of tools such as Fiabex (Gachet, Hutinet, and Mignot 1992) and Sofia (Sofreten 1992) and languages such as Figaro (Bouissou et al. 1991). Fiabex relied on ideas stemmed from assumption truth maintenance systems (de Kleer 1986) (in the early 1990's, so-called expert systems and rule-based reasoning raised a lot of interest, that has been eventually deceived). Sofia was an extension of reliability block diagrams via so-called Boolean polynomials. Figaro has been historically the first modeling language dedicated to probabilistic risk and safety analyses. Like Fiabex, it is rule-based.

Model-based safety assessment started however fully with the design of the AltaRica language, in the late nineties. We shall not retrace here the evolution of this language. We are actually mainly interested in its last version, AltaRica 3.0 (Batteux, Prosvirnova, and Rauzy 2019). AltaRica 3.0 can be described by the following equation.

$$\text{GTS} + \text{S2ML} = \text{AltaRica 3.0} \tag{8.1}$$

The above equation, which echoes the title of the famous book "Data structures + algorithms = programs", (Wirth 1976), summarizes an idea that goes much beyond risk and safety analyses and that can be stated as follows (Rauzy and Haskins 2019).

> Any behavioral description language is made of two parts: a mathematical framework to describe behaviors and a set of constructs to structure models.

In the case of AltaRica 3.0, the mathematical framework is the notion of guarded transitions systems (GTS) (Batteux, Prosvirnova, and Rauzy 2017; Rauzy 2008b) and the set of constructs to structure models is S2ML (Batteux, Prosvirnova, and Rauzy 2018). S2ML stands for system structure modeling language. It is the very topic of this chapter as the input language of XFTA (in either in XML or textual form), from version 2.0.0, is described by the following equation.

$$\text{SBE} + \text{S2ML} \quad = \quad \text{XFTA input language} \tag{8.2}$$

Where SBE stands for stochastic Boolean equations, i.e. the mathematical framework we described in the previous chapters.

Before diving into the presentation of S2ML, we need to make a few additional remarks.

The aim of the object-oriented approach in probabilistic risk assessment is to make the structure of a model reflect the functional and physical architecture of the system under study. There is however no such a thing as a one-to-one correspondence between system and model structures, for at least three reasons.

First, a model is always a specific point of view on the system. It aims at capturing a particular aspect or at evaluating a particular property of that system. Therefore, it abstracts away irrelevant parts of the system and is useful only because it does so.

Second, it is often necessary to model not only the system, but also its environment. Therefore, the frontiers of the system and the model are not always the same.

Third, the organization of a model has its own logic and its own constraints that can be quite different from those of the architecture of a technical system. It is clear that the choice of the right mathematical framework is of paramount importance to capture this or that physical aspect of the system, and to do it at the suitable level of abstraction. But the productivity of the modeling process stands also in the way models are structured. It is actually thanks to their structures that models can be versioned and configured throughout their life-cycle, that models of different engineering disciplines contributing to the design of a system can be synchronized, and that knowledge can be capitalized from projects to projects.

Constructs to structure models derive from those to structure computer programs. The main programming paradigms, such as object-oriented programming (see e.g. (Abadi and Cardelli 1998; Meyer 1988)), or functional programming (see e.g. (Abelson, G. J. Sussman, and J. Sussman 1996; Pierce 2002)) have their modeling counterparts. In programming, the choice of a paradigm is to some extent a matter of taste, although object-oriented programming is nowadays dominant in industrial practice.

The situation is rather different in the case of models. Their structuring paradigm is more independent of their behavioral content than in the case of computer programs. Moreover, prototype-orientation (Noble, Taivalsaari, and I. Moore 1999) seems to be the best-suited paradigm for models describing systems at least at the level of abstraction they are considered in systems engineering and reliability engineering.

S2ML gathers in a unified way constructs stemmed from both object- and prototype-oriented programming.

S2ML can eventually be seen from two different perspectives:

- As a domain specific language dedicated to the description of architectures of systems (see e.g. (Fowler 2010) for a reference textbook on domain specific languages);
- As a paradigm to structure models. A large class of actual modeling languages can be actually (re)constructed by plugging their behavioral constructs into S2ML.

8.2 S2ML in a Nutshell

Surprisingly enough, S2ML relies on only ten concepts: those of ports, connections, prototypes, classes, composition, cloning, instantiation, inheritance, reference and aggregation.

8.2.1 Basic Elements: Ports and Connections

Ports are basic objects of a model. In S2ML+SBE, parameters of probability distribution, state, flow and source variables, and common cause failure groups are ports.

Connections are relations, taken in a broad sense, that link ports. Connections capture the behavior of the system. In S2ML+SBE, equations defining parameters, state, flow and source variables, as well as definitions of common cause group failures are connections.

Ports and connections suffice to create a model. A fault tree is just a set of ports and connections. However, a model made only of ports and connections reflect only very indirectly the architecture of the system under study, hence the above mentioned difficulties.

8.2.2 Designing Hierarchical Models: Prototypes and Composition

To structure models, one needs containers for declarations of ports and connections (and other elements). The fundamental container is the *prototype*, a container with a unique occurrence in the model. In S2ML+SBE, prototypes are called blocks. When a container, a block, or any other type of container, contains an element, one says that the container *composes* this element. Composition is a fundamental relation between model elements, sometimes referred to as the *is-part-of* relation.

With ports, connections, and prototypes, it is already possible to design hierarchical models, i.e. to decompose the system under study into functional or physical subsystems, then these subsystems into subsubsystems, and so one until the wanted degree of granularity is reached.

Such models would lack however of two fundamental ingredients. First, a way to represent that two elements of the model describe similar parts of the system. This is especially of interest in reliability engineering where redundancy is a key element to ensure the required level of performance. Second, a way to connect elements located in different places in the hierarchy.

8.2.3 Duplicating Model Elements: Cloning, Classes, Instantiation and Inheritance

When two parts of the system under study are alike, e.g. if the system is made of two identical lines in hot redundancy, the description of the second line is the same as the description of the first one, up to the naming of elements. Once the first line described, the description of the second one can be obtained by a kind of copy-paste operation. This is however error prone and hides a fundamental information: the fact, precisely, that line 1 and line 2 are identical.

S2ML provides the concept of *cloning* to deal with such situations. Rather to copy-paste the prototype describing the first line to get the prototype describing the second one, one says that the second prototype is a clone of the first one. The assessment tool, XFTA in our case, is then in charge of performing the duplication.

Cloning makes it possible to duplicate modeling elements within a model, but not to reuse them from models to models. Moreover, considering the description of basic components, e.g. pumps or valves, the choice of the initial model element (from other similar model elements are obtained by cloning) is very arbitrary. The idea is therefore to create libraries of on-the-shelf modeling elements, outside any particular model, and to clone these modeling elements into the model, when needed.

In S2ML (and more generally in object-oriented programming), this is achieved by the concepts of classes and instances. A *class* is just a prototype declared outside the model. *Instantiation* is the operation by which a class is cloned into a model. The resulting prototype is called an *instance* of the class.

Now it is sometimes the case that a component is a particular type of a more general category of components, e.g. a solenoid valve is a particular type of valve. Most of the properties of the particular component are actually common to all components of the category, while some are specific. To represent that, it would be indeed possible to create a class for generic components then to instantiate this class into the class describing specific ones. This would lead however to awkward models: a solenoid valve is not part of a generic one. Rather, a solenoid valve *is-a* valve.

In S2ML (and more generally in object-oriented programming), capturing is-a relations is achieved by means of *inheritance*. When a prototype or class inherits from another prototype or a class, it means that all elements composed by the latter are composed by the former. It is then possible to modify the definitions of these elements or to add new ones to reflect the particular properties of the specific component.

8.2.4 Referring to Model Elements: Paths and Aggregation

The last ingredient we need to deploy fully object-oriented modeling is the possibility to refer to an element located somewhere in the hierarchy of prototypes from anywhere else in this hierarchy. The notion of *reference* is thus key.

In S2ML, referring to ports is achieved by means of *paths*. Within a block, each element is uniquely identified with a name, called its *identifier*. Two elements cannot have the same name, even though they are of different types. To refer to an element located in other blocks, one uses paths built with the dot notation and the two primitives `main` and `owner` (see also Section 4.6.2):

- `B.E` refers to the element `E` composed by the block `B` itself composed by the current block. Applying this principle recursively makes it possible to refer to any element located in the hierarchy rooted by the current block.
- `owner` refers to the parent block of the current block. Therefore, `owner.owner.B.E` refers to the element `E` composed by the block `B` itself composed by the grand-parent block of the current block. The primitive `owner` makes it possible to create relative paths referring to any element in the current hierarchy.
- `main` refers to the outermost block of the current hierarchy, i.e. the model itself. Therefore, `main.B.E` refers to the element `E` composed by the block `B` declared at the top-level. The primitive `main` makes it possible to create absolute paths referring to any element in the model.

There are cases where one needs to refer to not only an individual element, like a parameter or a variable, but a whole container, possibly itself composing sub-containers. In that case, using paths would be tedious, and error prone. The solution consists in the last concepts provided by S2ML, namely the *aggregation* of containers.

Let `A` and `B` be two containers located at different places in the same hierarchy. Let π`.B` the path (relative or absolute) to go from `A` to `B` in that hierarchy. To access an element `E` composed by `B` from `A`, one must normally use the path π`.B.E`. By aggregating in `A` the container `B` (actually the container π`.B`) under the name `C`, one makes possible to refer to `E` in `A` by means of the path `C.E`, instead of π`.B.E`. In some sense, this creates the alias `C` for the path π`.B` in `A`.

Aggregation should not be seen however only as a technical solution to create references. More fundamentally, it represents a *uses* relation. `A` uses `B` although `B` is not declared in the vicinity of `A`. We shall see application of this relation later on in this chapter.

8.2.5 Summary and Grammar

8.2.5.1 Blocks, Classes and Instances

We have already seen, Section 4.5.2, the grammar for block declarations. At the Open-PSA format, they are as follows.

```
<block name="ControlSystem">
  ... <!-- Declarations -->
</block>
```

Similarly, in S2ML+SBE they are as follows.

```
block ControlSystem
  ... // Declarations
end
```

Class declarations are similar, see Section 4.8.1. At the Open-PSA format, they are as follows.

```
<class name="Pump">
  ... <!-- Declarations -->
</class>
```

In S2ML+SBE, they are as follows.

```
class Pump
  ... // Declarations
end
```

Class declarations
In S2ML+SBE, classes are top-level elements: it is not possible to declare a class within a class or a block.

At the Open-PSA format, instance declarations are as follows.

```
<instance name="SV12" class="SolenoidValve">
  ... <!-- Declarations -->
</instance>
```

In S2ML+SBE, they can take two forms, depending on whether some elements of the instance need to be declared or redeclared.

```
SolenoidValve SV12;

SolenoidValve SV22
  ... // Declarations
end
```

8.2.5.2 Cloning, Inheritance and Aggregation

Cloning, inheritance and aggregation are described by means of specific directives, namely the directives `clones`, `extends` and `embeds`.

As seen Section 4.7.2, cloning at the Open-PSA format is as follows.

```
<clones name="Line2" block="Line1" >
  ... <!-- Declarations -->
</clones>
```

Cloning in S2ML+SBE is just as simple, with two variants depending on whether the clone declares or redeclares elements:

```
clones Line1 as Line2;

clones Line1 as Line3
  ... // Declarations
end
```

At the Open-PSA format, inheritance is declared as follows.

```
Model ::=
   ( BlockDeclaration | ClassDeclaration )*

BlockDeclaration ::=
   'block' Path ElementDeclaration* 'end'

ClassDeclaration ::=
   'class' ClassIdentifier ElementDeclaration* 'end'

ElementDeclaration ::=
      ParameterDeclaration | VariableDeclaration | CCFGroupDeclaration
   |  BlockDeclaration | InstanceDeclaration
   |  ExtendsDirective | EmbedsDirective

InstanceDeclaration ::=
      ClassIdentifier Path ';'
   |  ClassIdentifier Path ElementDeclaration* 'end'

ClassIdentifier ::=
   Identifier

ExtendsDirective ::=
      'extends' Path ';'

EmbedsDirective ::=
   'embeds' Path 'as' path ';'

Path ::=
      Identifier
   |  Identifier '.' Path
   |  'owner' '.' Path
   |  'main' '.' Path
```

Figure 8.1: EBNF S2ML+SBE grammar for S2ML constructs

```
<extends extented="Valve" />
```

In S2ML+SBE, it is similar:

```
extends Valve ;
```

Finally, at the Open-PSA format, the directive for aggregation is as follows.

```
<embeds embedded="main.Line1.LogicSolver" name="LogicSolver" />
```

In S2ML+SBE, it is similar:

```
embeds main.Line1.LogicSolver as LogicSolver;
```

8.2.5.3 EBNF Grammar

Figure 8.1 gives the EBNF grammar of S2ML declarations.

8.3 Models as Scripts

As explained Section 8.1, the objective is to design models that reflect the functional and physical architectures of the systems under study. No matter what these models are, assessment algorithms work on “flat” systems of stochastic Boolean equations. This means that the model “as designed” must to be transformed into a model “as assessed”. To be fully interesting, this transformation must:

– Be fully automated and transparent for the analyst;
– Preserve the semantics of the model;
– Be of low computational complexity.

The *flattening algorithm* that implements this transformation in XFTA fulfills these three criteria.

The new version of the Open-PSA format as well as S2ML+SBE illustrate the *model as script* principle. The idea is that the model “as designed” is seen as a script (or a program if one will) that is executed, thanks to the flattening algorithm, to obtain the model “as assessed”.

Technically, this operation is performed into two steps: the first step instantiates the source model to get a hierarchical model in which only blocks and stochastic Boolean equations remain, the second one flattens this hierarchical model into a pure set of stochastic Boolean equations. This is fully transparent for the user. Both steps are of quasi-linear complexity, meaning that they are performed in few seconds on average laptop computers, even for quite large models.

A key feature of the model as script principle is that it makes it possible to re-declare model elements, i.e. to change their initial definition. We saw this feature Section 4.7.1, but it is worth to present it again.

Consider the following S2ML+SBE model “as designed”.

```
1  block B
2     parameter lambda = 1.0e-4;
3     state BE1 = exponential(lambda);
4     state BE2 = exponential(2*lambda);
5     source HE = false;
6     flow Top = (HE and BE1) or (not HE and BE2);
7  end
8  parameter B.lambda = 1.23e-4;
9  source B.HE = true;
10 state B.BE1 = GLM(0.0, B.lambda, B.mu);
11 parameter B.mu = 1.0e-1;
```

Executing declarations in order creates model elements when they do not already exist and change their value otherwise. In this way,

– The value of the parameter `lambda` (of the block `B` is changed from `1.0e-4` to `1.23e-4` (line 8);
– The value of the house event `HE` is changed from `true` to `false` (line 9);
– The probability distribution associated with the basic event `BE1` is turned into a `GLM` distribution (line 10);
– A parameter `mu` composed by the block `B` is created outside this block (line 11).

Eventually, the following model “as assessed” is obtained:

```
#pure_model
parameter B.mu = 0.1;
gate B.Top = B.HE and B.BE1 or not B.HE and B.BE2;
house-event B.HE = true;
basic-event B.BE2 = exponential(2 * B.lambda);
basic-event B.BE1 = GLM(0, B.lambda, B.mu);
```

```
parameter B.lambda = 0.000123;
```

The directive `#pure_model` indicates that the model is a pure system of stochastic Boolean equations. It makes it possible to use identifiers with dots, e.g. `A.B.failed`.

The above principle which makes possible to clone prototypes and to instantiate classes and to change, in the resulting blocks, the definitions of parameters, basic events, house events and gates, and even to add new elements.

It works however slightly differently for containers (blocks or instances of classes). In this case, the redeclared block is not substituted for the previous block. Rather, declarations within the redeclared block are executed. Consider the following example.

```
block B
   block C
      state failed = exponential(lambda);
      parameter lambda = 1.0e-4;
   end
end
block B.C
   state failed = Weibull(alpha, beta);
   parameter alpha = 1.23e4;
   parameter beta = 3;
end
```

The redeclaration of the block `B.C` does not cancel its previous declaration. The redefinition of the probability distribution of the state variable `failed` does not delete the parameter `lambda`.

The flattened model is thus as follows.

```
#pure_model
state B.C.failed = Weibull(alpha, beta);
parameter B.C.lambda = 1.0e-4;
parameter B.C.alpha = 1.23e4;
parameter B.C.beta = 3;
```

In the remainder of this chapter, we shall illustrate S2ML constructs on a concrete case study.

8.4 Illustrative Case Study

8.4.1 Description

To illustrate our presentation, we shall consider the high integrity pressure protection system pictured in Figure 8.2.

This system is in charge of preventing an over pressure coming from wells `W1` and `W2` to damage the downstream equipment, namely the separator `S`.

A safety instrumented system is thus installed on each line. It consists of three pressure sensors (`PS11`, `PS12` and `PS13` for line 1), a logic solver (`LS1` for line 1), and two shutdown valve (`SDV11` and `SDV12` for line 1). The logic solver is made of two separate modules: an acquisition module directly connected to sensors (`AM1` for `LS1`) and a command module (`CM1` for `LS1`). Each shutdown valve is operated by means of a solenoid valve, which is directly connected to the logic solver (`SV11` for `SDV11`).

When two out of the three pressure sensors detect an overpressure, the logic solver sends the command to both solenoid valves to close their respective shutdown valves.

Each component of this system may fail. Reliability data for these components are given in Table 8.1. All components are continuously monitored but valves that are periodically maintained.

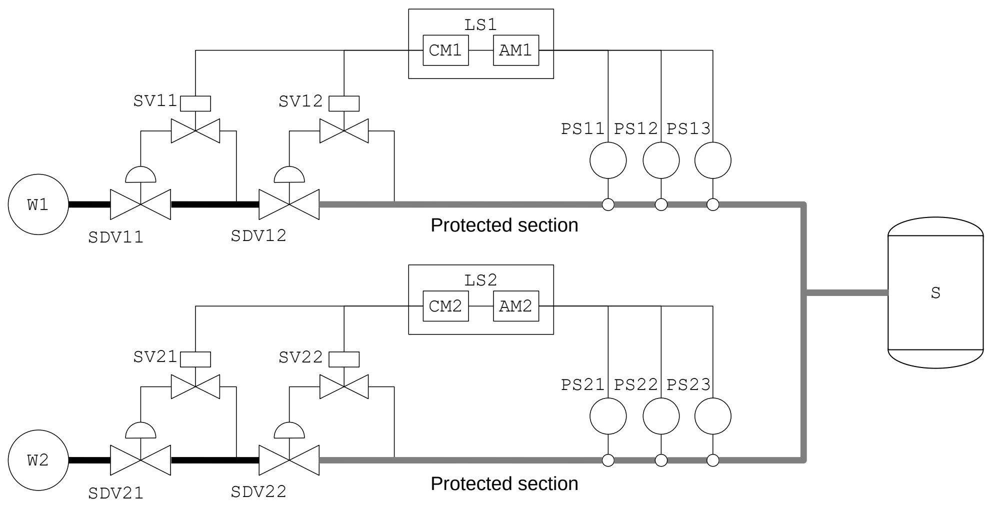


Figure 8.2: A high integrity pressure protection system

Table 8.1: Reliability data for the components of the high integrity pressure protection system pictured in Figure 8.2

Components	Distributions	Parameters
`PSij`'s	`exponential`	failure rate $\lambda = 1.23 \times 10^{-6} h^{-1}$
`AMi`'s	`Weibull`	scale parameter $\alpha = 2.56 \times 10^{7} h$, shape parameter $\beta = 3$
`CMi`	`Weibull`	scale parameter $\alpha = 6.47 \times 10^{7} h$, shape parameter $\beta = 3$
`SVij`'s	`periodic-test`	failure rate $\lambda = 2.89 \times 10^{-5} h^{-1}$, interval between tests $\tau = 4380 h$, date of the first test $\theta = 2190 h$ for valves `SVi1` and $\theta = 4380 h$ for valves `SVi2`
`SDVij`'s	`periodic-test`	failure rate $\lambda = 1.97 \times 10^{-4} h^{-1}$, interval between tests $\tau = 4380 h$, date of the first test $\theta = 2190 h$ for valves `SDVi1` and $\theta = 4380 h$ for valves `SDVi2`

Valves are assumed to be as good as new after a maintenance operation. The solenoid valve `SVij` and the shutdown valve `SDVij` are maintained at the same time. Moreover, maintenance operations for pairs of valves on the same line are staggered.

8.4.2 Building Blocks

We shall first prepare the ground for our model by designing a library of modeling components to describe components of our system: pressure sensor, acquisition modules, command modules, logic solvers, solenoid valves and shutdown valves.

8.4.2.1 Basic Blocks

Prior to designing the model for the system, we need to choose a modeling style, i.e. fault tree like, or reliability block diagram (or any other style). In our case study, the reliability block diagram style seems appropriate. All components will thus derive from a generic basic block, which can be described as follows.

```
class BasicBlock
   state failed = 0.0;
   flow in = false;
   flow out = in and not failed;
end
```

The basic block has an internal state `failed`, an input flow `in` and an output flow `out`. The probability distribution associated with the internal state and the definition of the input flow do not really matter here, because they will be redefined anyway.

Classes describing concrete components can be now derived from the class `BasicBlock`. For instance, the class for pressure sensors `PSi`'s is as follows.

```
class PressureSensor
   extends BasicBlock;
   state failed = exponential(lambda);
   parameter lambda = 1.23e-6;
   flow in = true;
end
```

The class `PressureSensor` inherits from the class `BasicBlock` and redefines the probability distribution associated with the internal state.

We can proceed in the same way for the classes `CommandModule`, `SolenoidValve` and `ShutdownValve` describing the corresponding components. Acquisition modules are slightly more complex, as we shall see now.

8.4.2.2 Macro-Components

The acquisition modules `AM`'s of logic solvers take three inputs rather than one, and combine these three inputs according to a 2-out-of-3 logic. The code for acquisition modules can thus be as follows.

```
class AcquisitionModule
   extends BasicBlock;
   state failed = Weibull(alpha, beta, 0);
   parameter alpha = 2.57e7;
   parameter beta = 3;
   flow in1 = false;
   flow in2 = false;
   flow in3 = false;
   flow in = atleast 2(in1, in2, in3);
end
```

Now, logic solvers combine in series an acquisition module and a command module. The code for logic solver can thus be as follows.

```
class LogicSolver
   AcquisitionModule AM;
   CommandModule CM;
   flow CM.in = AM.out;
end
```

We can proceed in the same way to define a class `ValveSystem` that combines in series an instance of `SolenoidValve` and an instance of `ShutdownValve`."

We have now a description for all basic components of our system.

8.4.3 Description of the Architecture of the System

So far, we worked bottom-up, i.e. we look for basic bricks that will help us to design our model. We shall now take a top-down approach to describe the architecture of the system.

We can first remark that our system is made of two identical lines working in parallel. The failure of one these two lines induces a failure of the system. The code for the HIPPS can thus be as follows.

```
block HIPPS
   block Line1;
     ... // To be defined
   end
   clones Line1 as Line2;
   flow failed = Line1.failed or Line2.failed;
end
```

Note that we gathered all declarations regarding the HIPPS within one top-level block `HIPPS`, for the sake of clarity of the model. It would be however possible to declared the blocks `Line1` and `Line2` as well as the variable `failed` directly in the model.

We have now to define the block `Line1`. We can remark that lines 1 and 2 are control system. As all control systems, they are made of three main parts: a sensor (or group of sensors), a controller and an actuator (or group of actuator). It is worth to reflect this functional architecture in our model.

We can also remark that in a reliability block diagram like model, the sensor, the controller and the actuator of a control system are arranged in series, i.e. the output of the sensor is plugged onto the input of the controller, and the output of the controller is plugged onto the input of the actuator. The control system is failed if there is nothing as output of the actuator.

Eventually, the code for `Line1` can be as follows.

```
block Line1
   block PSG // Pressure Sensor Group
      PressureSensor PS1;
      PressureSensor PS2;
      PressureSensor PS3;
   end
   LogicSolver LS;
   block VG // Valve Group
      flow in = true;
      ValveSystem VS1;
      ValveSystem VS2;
      flow VS1.SV.in = in;
      flow VS2.SV.in = in;
      flow out = VS1.SDV.out or VS2.SDV.out;
   end
   flow LS.AM.in1 = PSG.PS1.out;
   flow LS.AM.in2 = PSG.PS2.out;
   flow LS.AM.in3 = PSG.PS3.out;
   flow VG.in = LS.CM.out;
   flow failed = not VG.out;
end
```

The block `PSG` gathers pressure sensors. The block `VG` gathers valve systems. Redeclarations of input flow variables make it possible to plug components together.

We obtain in a rather simple way an easy to understand, easy to validate and easy to maintain model.

8.4.4 Sharing Model Elements

8.4.4.1 Reliability Data

It is often the case that reliability data are stored into data-bases outside the model, e.g. the OREDA data-base for the oil and gas industry (SINTEF and NTNU 2015). The version 2 of the Open-PSA standard provided a special container, `model-data`, to store reliability parameters (it was however not exactly the same notion of container as in the current version, as it did not act as a name space).

We may thus want to store reliability data for the elements of our model in a separate file, or at least in a separate block. This is possible thanks to directive `main` of paths.

The code for the data-base of reliability data could be as follows.

```
block ReliabilityData
   block Sensors;
      ...
      parameter lambdaPressureSensor = 1.23e-6;
      ...
   end
   ...
end
```

The reliability data stored into this data-base can then be referred to in the model. E.g.

```
block HIPPS
   block Line1
      block PSG // Pressure Sensor Group
         PressureSensor PS1
            parameter lambda =
               main.ReliabilityData.Sensors.lambdaPressureSensor;
         end
         ...
      end
      ...
   end
   ...
end
```

8.4.5 Functional Chains

Assume that line 1 and line 2 share the logic solver. The latter is thus made of two acquisition modules but only one command module that shutdown the production in case of an overpressure from either well. Its code could be as follows.

```
class LogicSolver
   AcquisitionModule AM1;
   AcquisitionModule AM2;
   CommandModule CM;
   flow CM.in = AM1.out and AM2.out;
end
```

We have to modify the model so to reflect this change in architecture. There is a problem however: it is impossible to have the same component, in this case the logic solver, composed by two different parent blocks (`Line1` and `Line2`). But it can be aggregated by two or more blocks.

The idea is thus to declare the logic solver outside the blocks `Line1` and `Line2`, then to aggregate it into these blocks (and to plug outputs of sensors onto the correct acquisition module):

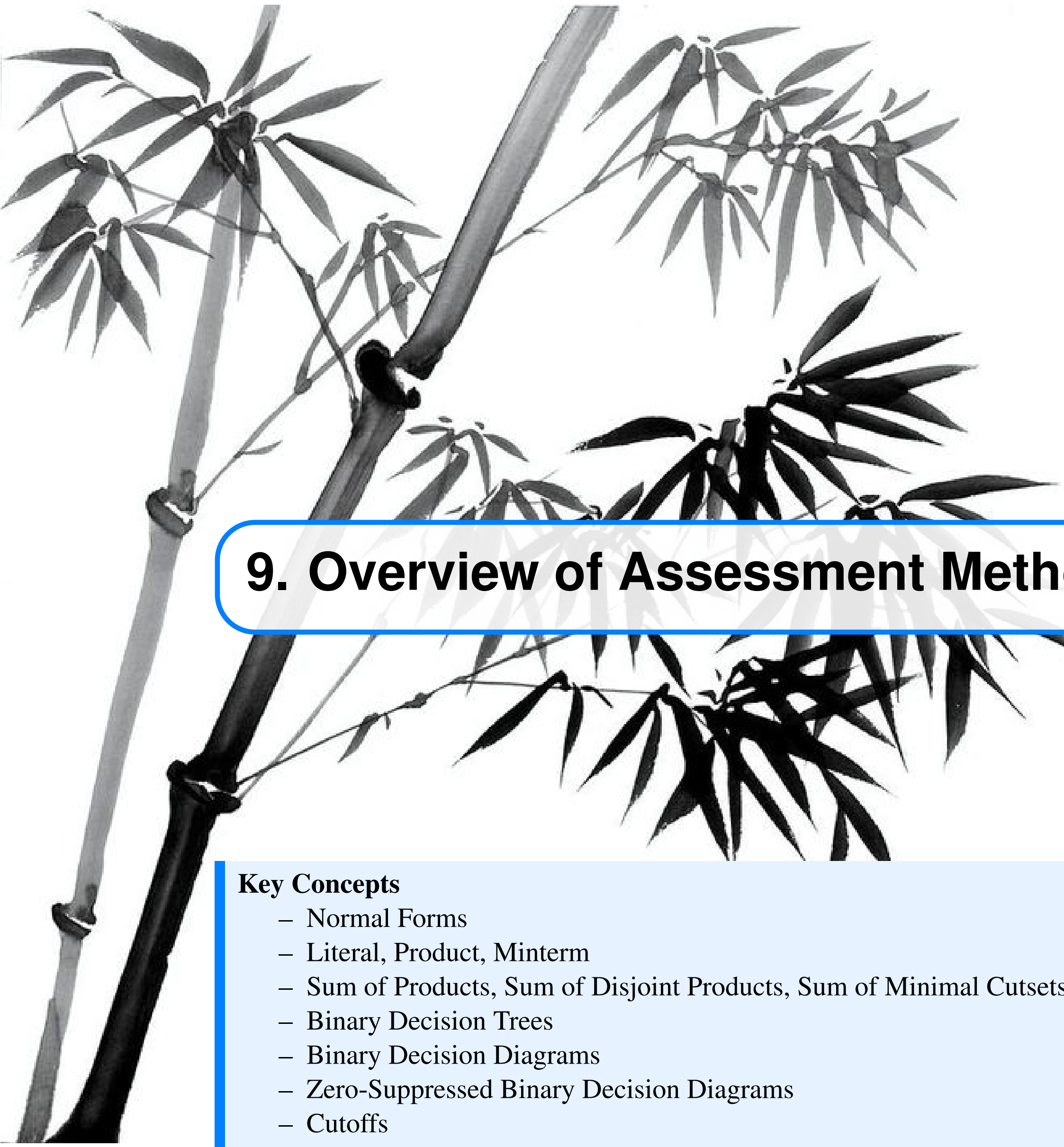
```
block HIPPS
   LogicSolver LS;
   block Line1
      ...
      embeds owner.LS as LS;
      ...
      flow LS.AM1.in1 = PSG.PS1.out;
      flow LS.AM1.in2 = PSG.PS2.out;
      flow LS.AM1.in3 = PSG.PS3.out;
      ...
   end
   clones Line1 as Line2
      flow LS.AM2.in1 = PSG.PS1.out;
      flow LS.AM2.in2 = PSG.PS2.out;
      flow LS.AM2.in3 = PSG.PS3.out;
      end
   flow failed = Line1.failed or Line2.failed;
end
```

Aggregation makes it possible to separate functional and physical views of the system: components are declared in the physical view and aggregated in the functional view. This is the fundamental idea behind the notion of *functional chains* (Batteux, Prosvirnova, and Rauzy 2019; Voirin 2008).

In the above example, `Line1` and `Line2` can be seen as functional chains. Here, we declared sensors and actuators within the chains, but we could have declared them outside as well.

III

Calculations



9. Overview of Assessment Methods

Key Concepts

- Normal Forms
- Literal, Product, Minterm
- Sum of Products, Sum of Disjoint Products, Sum of Minimal Cutsets
- Binary Decision Trees
- Binary Decision Diagrams
- Zero-Suppressed Binary Decision Diagrams
- Cutoffs
- Handles
- Model pruning
- Scripts, Commands
- Environment variables
- TSV files

This chapter aims at providing the reader with an overview of assessment methods implemented in XFTA.

Before assessing a system of stochastic Boolean equations, the structure function of its top event must be put in a normal form, and this normal form must be stored into a dedicated data structure. XFTA implements two normal forms, sums of disjoint products ans sums of minimal cutsets and three data structures to encode them, binary decision trees, binary decision diagrams and zero-suppressed binary decision diagrams. To use XFTA, it is important to have at least a shallow understanding its assessment flows, i.e. of the way data structures encoding normal forms are built then used for calculations of probabilistic risk indicators.

After a short introduction, this chapter gives mathematical definitions of normal forms and presents data structures to store them. It describes briefly the algorithms that build these data structures and those that calculate probabilistic risk indicators. In depth presentations will be given in the subsequent chapters.

9.1 Quick Start

9.1.1 Assessment Flows

XFTA sessions follow more or less always the same pattern:

– First, the source model is loaded from one or several files.
– Second, it is instantiated and flattened into a target model. Common cause failure groups are extended at this step.
– Third, one or more normal forms of the structure function of the top event are calculated and stored in a dedicated data structure.
– Fourth, at least the more significant minimal cutsets are printed out for model validation as well as for qualitative analyses purposes.
– Fifth, probabilistic risk indicators are calculated from the normal forms.

The fourth and fifth steps can be performed in any order.

A formula is said in *normal form* if it obeys certain syntactic conditions. Two normal forms are used in the realm of fault tree analysis: *sums of minimal cutsets* and *sums of disjoint products*. The current version of XFTA implements two assessment flows: the first one relies only of sum of minimal cutsets, the second one on both sums of disjoint products and sum of minimal cutsets. Figure 9.1 shows XFTA main assessment flows and gives the commands implementing these flows.

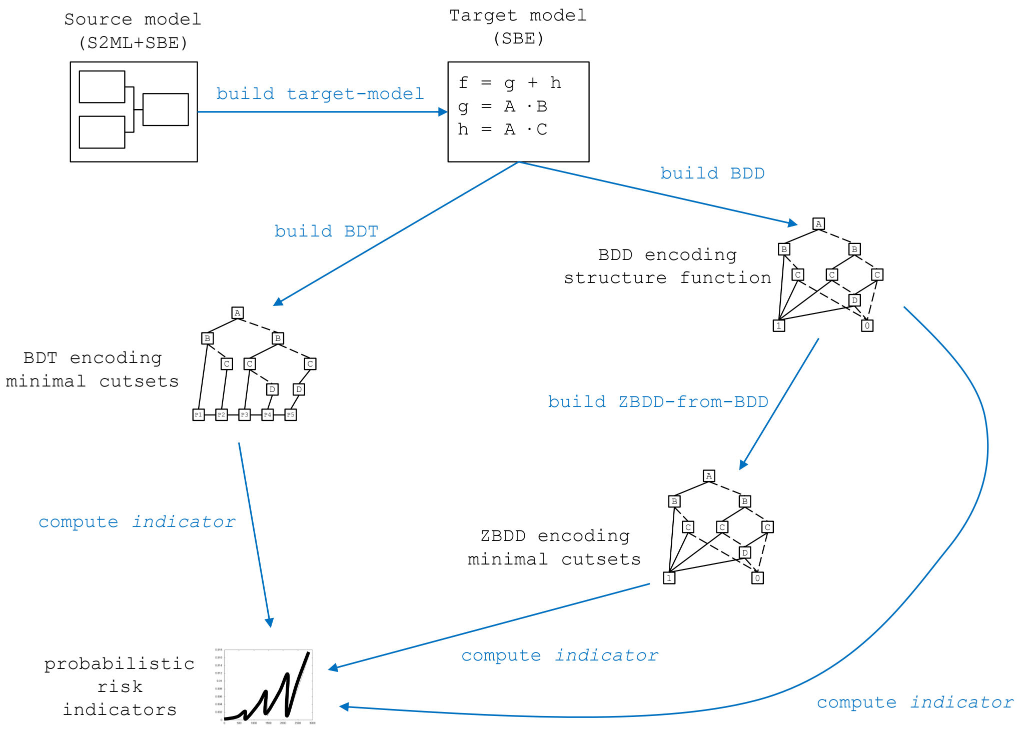


Figure 9.1: XFTA assessment flows

The first assessment flow consists in extracting minimal cutsets, storing them in a data structure called *binary decision trees*, and printing out the most significant minimal cutsets and calculating probabilistic risk indicators from this data structure.

The second assessment flow consists in calculating first a sum of disjoint products and storing it into a data structure called *binary decision diagrams*. Then, minimal cutsets are extracted from the binary decision diagram and stored in a data structure called *zero-suppressed binary decision diagrams*. Probabilistic risk indicators are calculated from binary decision diagrams. They could also be estimated from zero-suppressed binary decision diagrams, but this presents no real interest.

For the sake of convenience, we shall call these two assessment flows respectively the *minimal cutsets approach* and the *binary decision diagram approach*, although strictly speaking both involve the extraction of minimal cutsets and sums of minimal cutsets are a normal form while binary decision diagrams are a data structure.

At this point the reader may ask why implementing and maintaining two different assessment processes? It will take a large part of the remainder of this chapter and of the following chapters to answer properly this fundamental question. But we can already give the key ideas.

Binary decision trees *Binary decision trees* encode sums of minimal cutsets. The size of a binary decision tree is roughly proportional the size of the sum of minimal cutsets it encodes, i.e. to the sum of the sizes (called the orders) of the minimal cutsets. Minimal cutsets encoded by a binary decision tree can be sorted, e.g. in decreasing order of their probabilities. This makes it possible to maintain the set of the most interesting (probable) minimal cutsets.

Zero-suppressed binary decision diagrams *Zero-Suppressed binary decision diagrams* encode also sums of minimal cutsets. The size of a zero-suppressed binary decision diagram can much smaller that the size of the sum of minimal cutsets it encodes. Not only it can, but it often is. The downside of this compactness is that minimal cutsets encoded by a zero-suppressed binary decision diagram cannot be sorted.

Binary decision diagrams *Binary decision diagrams* encode sums of disjoint products, often in a very compact way.

Cutoffs Even medium size models can have huge numbers of minimal cutsets. If probabilities of basic events are not too high—an hypothesis that is verified most of the time in practice—, the probability of the top event is roughly equal to the sum of the probability of the minimal cutsets. In practice, this means that even if there is a huge number of minimal cutsets, only a tiny fraction of them has a significant probability and concentrates the risk.
This is the reason why, minimal cutsets extraction is performed using a *cutoff*, i.e. a minimum probability and/or a maximum order of the extracted minimal cutsets. In XFTA, cutoffs are adjusted dynamically so to keep only a certain number of minimal cutsets, usually the most probable ones.

Probabilistic risk indicators Sums of disjoint products make it possible to calculate the exact values of probabilistic risk indicators (including the most important of them, i.e. the probability of the top event), while only approximated values can be obtained from sums of minimal cutsets.

Quantification methods There exists several methods to approximate the values of probabilistic risk indicators from sums of minimal cutsets. XFTA implements three of them: the *rare event approximation*, the *mincut upper bound* and the *pivotal upper bound*. The first two ones have been proposed in the reliability engineering literature decades ago. The mincut upper bound gives more accurate results than the rare event approximation. The pivotal upper bound is original (I introduced it in 2020). It gives in general better estimates than the mincut upper bound and is faster to calculate.

Which approach to choose? For small and medium size models, the binary decision diagram approach outperforms the minimal cutsets approach. Not only it the assessment is faster and requires less computer memory (which is the decisive factor regarding the efficiency of assessment methods), but it provides exact values of probabilistic risk indicators.
For very large models however, it is not possible to build the binary decision diagram encoding the top event. The latter simply does not fit in the computer memory. The minimal cutsets approach provides then a very good alternative. As explained above, it usually the case in practice that only a tiny fraction of minimal cutsets concentrates the risk. By seeking these minimal cutsets (and ignoring the others), extraction algorithms act as *model pruners*. They do not aim at calculating exact values of risk indicators, but at obtaining reasonable

estimates for these values, using a necessarily limited amount of calculation resources.

9.1.2 Basic Script Implementing the Minimal Cutsets Approach

A XFTA session consists in executing a XFTA script. Scripts are stored into ordinary text files. Usually, we give these files the extension "`.xfta`". *Scripts* are sequences of *commands* that are executed in a row. They implement the processes described in the previous section.

■ **Example 9.1** Let us come back to the tracking system example presented Section 1.2. This model is small. It can thus be assessed by both the minimal cutsets (without cutoff) and the binary decision diagram approaches. Assume it is stored into a unique file "`TrackingSystem.sbe`". Assume finally that we want to estimate the probability of the top event at times 100, 1000 and 10000.

Figure 9.2 gives a basic script implementing the minimal cutsets approach. ■

```
load model "TrackingSystem.sbe";
build target-model;
build BDT TrackingSystem.failed;
print minimal-cutsets TrackingSystem.failed
   mission-time=1000
   output="TrackingSystem-mcs.tsv";
compute probability TrackingSystem.failed
   quantification-method=pivotal-upper-bound
   mission-time=[100, 1000, 10000]
   output="TrackingSystem-prb.tsv";
```

Figure 9.2: Basic script implementing the minimal cutsets approach

Commands are terminated with a semicolon. This makes it possible for a command to spread on several lines.

Commands have a name which is an action verb, e.g. `load`, `build`, `compute`, `print`. After the name comes the object on which the action is applied, e.g. `model`, `target-model`, `BDT`, `minimal-cutsets`. Then come possibly arguments, e.g. the name of the top event `TrackingSystem.failed`, and options, e.g. `output="TrackingSystem-mcs.tsv"`, `mission-time=[1000, 2000, 5000]`. Arguments are mandatory and consist of one value. Options are optional and consist of an identifier followed by the symbol =, itself followed by a value. Values can be atomic like an identifier, a string or a number, or complex expressions, including lists such as `[100, 1000, 10000]`.

Each command requires thus parameters that are specific to that command. These parameters are given either via arguments or via options (but never both). Some parameters are optional because XFTA manages a set of *environment variables*. These environment variables record default values for parameters of commands. We shall this point in more details Section 10.2.

The script given in Figure 9.2 starts with a command that loads the model from the file containing it, here the file "`TrackingSystem.sbe`".

It is possible to split the model into several files. In such a case, the command `load` must be called once on each file.

The names of files must be surrounded by quotes, so to be able to handle names containing spaces or starting with a digit. File names and directory names have however to be made of ASCII characters only.

The second command of this script builds the target model.

The third one extracts minimal cutsets of the variable `TrackingSystem.failed`. Minimal cutsets are always extracted for a certain cutoff. Cutoffs set up three threshold values: the maximum order (size) of minimal cutsets, their minimum probability, and the maximum number of minimal cutsets that can be extracted. Default values of these three threshold are defined by means of environment variables, respectively `maximum-order`, `minimum-probability` and `maximum-number`. The probabilities of minimal cutsets may depend on the mission time. The default value of the mission time is also defined by means of the environment variable `mission-time`.

The fourth command prints out the extracted minimal cutsets into the *TSV file* "`Tracking-System-mcs.tsv`". TSV files are ordinary text files that spreadsheet tools (like Excel®) can read. The extension `.tsv` stands for "tabulation separated value". Each line of the file encodes a row of the spreadsheet. Tabulations separate values, i.e. cells of the spreadsheet. By default, minimal cutsets are printed out together with their rank (their index once they have been sorted in descending order of their probabilities), their probability and their contribution, i.e. the probability of the cutset divided by the sum of the probabilities of the cutset. As for minimal cutsets extraction, the default value of the mission time at which probabilities are calculated is fixed by the environment variable `mission-time`. Eventually, the file "`TrackingSystem-mcs.tsv`" is as show in Figure 9.3. It contains one minimal cutset per line. The first three columns give respectively the rank, the probability and the contribution of the minimal cutset. The subsequent columns give its basic events. Figure 9.3 shows actually only the 10 most significant minimal cutsets, out of 44.

```
1 6.96776e-08 0.506864 TrackingSystem.LS.V1.failed
    TrackingSystem.LS.V2.failed
2 2.2333e-08 0.16246 TrackingSystem.M2.A1.failed
    TrackingSystem.M2.A2.failed TrackingSystem.M3.A1.failed
    TrackingSystem.M3.A2.failed
3 2.2333e-08 0.16246 TrackingSystem.M1.A1.failed
    TrackingSystem.M1.A2.failed TrackingSystem.M2.A1.failed
    TrackingSystem.M2.A2.failed
4 2.2333e-08 0.16246 TrackingSystem.M1.A1.failed
    TrackingSystem.M1.A2.failed TrackingSystem.M3.A1.failed
    TrackingSystem.M3.A2.failed
5 7.88519e-10 0.00573602 TrackingSystem.LS.D1.failed
    TrackingSystem.LS.D2.failed
6 2.16406e-13 1.57423e-06 TrackingSystem.LS.V1.failed
    TrackingSystem.M1.C2.failed TrackingSystem.M2.A1.failed
    TrackingSystem.M2.A2.failed
7 2.16406e-13 1.57423e-06 TrackingSystem.LS.V1.failed
    TrackingSystem.M2.A1.failed TrackingSystem.M2.A2.failed
    TrackingSystem.M3.C2.failed
8 2.16406e-13 1.57423e-06 TrackingSystem.LS.V2.failed
    TrackingSystem.M1.C1.failed TrackingSystem.M2.A1.failed
    TrackingSystem.M2.A2.failed
9 2.16406e-13 1.57423e-06 TrackingSystem.LS.V2.failed
    TrackingSystem.M2.A1.failed TrackingSystem.M2.A2.failed
    TrackingSystem.M3.C1.failed
10 2.16406e-13 1.57423e-06 TrackingSystem.LS.V1.failed
    TrackingSystem.M1.C2.failed TrackingSystem.M3.A1.failed
    TrackingSystem.M3.A2.failed
```

Figure 9.3: 10 minimal cutsets extracted for the variable `TrackingSystem.failed`

The fifth and last command compute the probability of the top event (here `TrackingSystem.-failed`) for different mission times. The results are printed out in the TSV file "`TrackingSystem--prb.tsv`". It could be as shown in Figure 9.4.

```
time PUB/BDT
100  7.03792e-10
1000 1.37465e-07
10000 0.00131658
```

Figure 9.4: Probability of the variable `TrackingSystem.failed` calculated at different mission times

9.1.3 Basic Script Implementing the Binary Decision Diagram Approach

Figure 9.5 gives a basic script implementing the binary decision diagram approach.

```
load model "TrackingSystem.sbe";
build target-model;
build BDD TrackingSystem.failed;
compute probability TrackingSystem.failed
   mission-time=[100, 1000, 10000]
   output="TrackingSystem-prb.tsv";
build ZBDD TrackingSystem.failed;
print minimal-cutsets TrackingSystem.failed
   mission-time=1000
   output="TrackingSystem-mcs.tsv";
```

Figure 9.5: A simple XFTA script implementing the binary decision diagram approach

This script is very similar to the previous one, although things are done in a slightly different order.

The first two commands are the same.

The third command builds the binary decision diagram encoding the structure function of the top event `TrackingSystem.failed`.

The fourth command computes the probability of the top event at times 100, 1000 and 10000 and stores the results in the file "`TrackingSystem-prb.tsv`". The probability is computed from the binary decision diagram.

The fifth command builds the zero-suppressed binary decision diagram encoding the minimal cutsets of the top event `TrackingSystem.failed` from the binary decision diagram encoding the structure function associated with this event.

The sixth and last command of the script prints out the minimal cutsets encoded in the zero-suppressed binary decision diagram built by the previous command into the file "`Tracking-System-mcs.tsv`". Note that here the minimal cutsets are not sorted.

9.1.4 Probabilistic Risk Indicators

Up to now, we considered only one probabilistic risk indicator: the probability of the top event, possibly calculated at different mission times. More details about the calculation of this indicator are given Chapter 13. XFTA implements several other indicators:

- Availability and mean availability, which are discussed Chapter 13.
- Importance measures, which are discussed Chapter 14.
- Sensitivity measures, which are discussed Chapter 15.
- Approximations of system reliability and safety integrity levels, which are discussed Chapter 16.

In the remainder of this chapter, we shall compare further the minimal cutsets and the binary decision diagram approaches, starting by defining formally normal forms and presenting in more details the data structures in which they are stored.

9.2 Normal Forms

9.2.1 Assessment of the Probability of the Top Event

We have seen in the previous part of this book that any S2ML+SBE model can be transformed in an equivalent system of stochastic Boolean equations and that a system of stochastic Boolean equations associates a Boolean function to each flow variable (intermediate event) of the system. Yet, this association is implicit. Consequently, it is in general impossible to calculate the probability of the top event directly from the system of stochastic Boolean equations.

This impossibility is illustrated by the following example.

■ **Example 9.2** Let $\mathscr{M}$ the following system of stochastic Boolean equations.

$$\begin{array}{rcl} T & = & G \vee H \\ G & = & 2/3(A,B,C) \\ H & = & 2/3(B,C,D) \end{array}$$

Although it is possible to calculate the probability of G and H directly (if we know the probabilities of A, B, C and D, it is not possible to calculate the probability of T from the probabilities of G and H as the events G and H are not disjoint (hence $p(G \wedge H) \neq 0$). ■

To be able to calculate the probability of the top event of a model, one needs to put the structure function of this top event in a normal form from which the probability can be estimated.

In general, a *normal form* is a formula obeying specific syntactic constraints. Regarding our problem, two normal forms have been looked at in the literature: sum of minimal cutsets and sums of disjoint products. We shall now introduce these notions.

9.2.2 Sum of Products, Sum of Disjoint Products, Sum of Minimal Cutsets

We shall start with some vocabulary and notations.

Definition 9.2.1 — Literal, Products, and Minterms. Let $\mathscr{V}$ be a finite set of Boolean variables.

A *literal* built over $\mathscr{V}$ is either a variable V of $\mathscr{V}$ or its negation $\overline{V}$. V is called the *positive literal* and $\overline{V}$ the *negative literal* of V. V and $\overline{V}$ are *opposite literals* (note that $\overline{\overline{V}} \equiv V$).

A *product* built over $\mathscr{V}$ is set of literals built over $\mathscr{V}$ interpreted as the conjunction of its elements. A product is said *essential* if it does not contain both literals of a variable and *positive* if it contains only positive literals.

Finally, a *minterm* built over $\mathscr{V}$ is a product built over $\mathscr{V}$ that contains a literal built over each variable of $\mathscr{V}$. The set of minterms that can be built over $\mathscr{V}$ is denoted $\text{minterms}(\mathscr{V})$.

From now, we shall say product instead of essential product, as non essential products are trivially equivalent to the constant 0 (false).

Products are sets interpreted as conjunctions. This dual vision of products is convenient. For instance to speak about product entailment.

Definition 9.2.2 — Subsumption. Let σ and π be two products built over a set of variables $\mathscr{V}$. Then, σ *subsumes* π if $\sigma \subseteq \pi$ or equivalently if $\sigma \models \pi$.

It is easy to verify that if $\mathscr{V}$ is finite and contains n variables, then 2^n distinct minterms can be built over $\mathscr{V}$.

This is by no means a coincidence as the following property holds.

Property 9.1 — Minterms versus Assignments. Let $\mathscr{V}$ be a set of Boolean variables, then minterms built over $\mathscr{V}$ one-to-one correspond with assignments of $\mathscr{V}$.

Proof: the minterm π one-to-one correspond with the assignment σ such that for all variable V of $\mathscr{V}$ $\sigma(V) = 1$ if and only if $V \in \pi$ (and thus $\sigma(V) = 0$ if and only if $\overline{V} \in \pi$.

Definition 9.2.3 — Sum of Products. Let $\mathscr{V}$ be a set of Boolean variables.

A *sum of products* built over $\mathscr{V}$ is a set of products built over $\mathscr{V}$ interpreted as the disjunction of its elements.

A sum of products φ is a *sum of disjoint products* if for any two products π and ρ of φ there exists at least one variable V of $\mathscr{V}$ that occurs positively in π and negatively in ρ, or vice-versa.

Sums of products are sometimes called formulas in *disjunctive normal form*.

A consequence of Property 9.1 is that for any formula f built over $\mathscr{V}$, there exists a unique sum of minterms φ that is equivalent to f. This sum of minterms one-to-one correspond with the set of assignments that satisfy f, i.e. with the Boolean function into which f is interpreted.

Note that a sum of minterms is by definition a sum of disjoint products. The reverse is not true, as illustrated by the following example.

■ **Example 9.3** Consider again the system of stochastic Boolean equations of Example 9.2. The structure function of the top event T is equivalent to the sum of the following minterms.

$$\begin{array}{lllll}
A\cdot B\cdot C\cdot D & A\cdot B\cdot C\cdot\overline{D} & A\cdot B\cdot\overline{C}\cdot D & A\cdot B\cdot\overline{C}\cdot\overline{D} & A\cdot\overline{B}\cdot C\cdot D \\
A\cdot\overline{B}\cdot C\cdot\overline{D} & \overline{A}\cdot B\cdot C\cdot D & \overline{A}\cdot B\cdot C\cdot\overline{D} & \overline{A}\cdot B\cdot\overline{C}\cdot D & \overline{A}\cdot\overline{B}\cdot C\cdot D
\end{array}$$

It is also equivalent to the sum of the following disjoint products.

$$A\cdot B \quad A\cdot\overline{B}\cdot C \quad \overline{A}\cdot B\cdot C \quad \overline{A}\cdot B\cdot\overline{C}\cdot D \quad \overline{A}\cdot\overline{B}\cdot C\cdot D$$

■

We shall postpone the formal definition of minimal cutsets to Chapter 11. For now, we can go with the intuitive definition:

– A *cutset* is a set of basic events which if they are realized together induce the top event.
– A cutset is *minimal* if none of its proper subset is a cutset, i.e. there is no other cutset that subsumes strictly it.

In our example, minimal cutsets of T are easily extracted by taking the union of minimal cutsets of G and H.

■ **Example 9.4** Consider again the system of stochastic Boolean equations of Example 9.2. The sum of minimal cutsets of the structure function of the top event T is made of the following products.

$$A\cdot B \quad A\cdot C \quad B\cdot C \quad B\cdot D \quad C\cdot D$$

■

9.2.3 Pivotal Decompositions

Sums of products are sets of products which are themselves sets of literals. Sums of products are also disjunctions of products which are themselves conjunctions of literals. It is actually sometimes convenient to see products and sums of products as sets and sometimes as formulas.

Along the following pages, we shall introduce a few operators, which are specific to sums of products.

Definition 9.2.4 — Extension of a Sum of Products. Let φ be a sum of products built over a set of variables $\mathcal{V}$ and let L be a literal (positive or negative) built over a variable of $\mathcal{V}$ that does not show up in φ. Then, the *extension* of φ with L, denoted $L \odot \varphi$, is defined as follows.

$$L \odot \varphi \overset{def}{=} \{L \cdot \pi ; \pi \in \varphi\}$$

Cofactoring is in some sense the reverse operation of extension. It applies to all formulas.

Definition 9.2.5 — Cofactor. Let f be a Boolean formula built over a set of variables $\mathcal{V}$, let E be a variable of $\mathcal{V}$ and let v be a Boolean value, i.e. either 0 or 1. Then, the *cofactor* of f for E and v, denoted $f\,|_{E=v}$, is the formula in which the value v has been substituted for all occurrences of E.

It is sometimes convenient to use literals rather than equalities, i.e. $f\,|_E$ denotes $f\,|_{E=1}$ and $f\,|_{\overline{E}}$ denotes $f\,|_{E=0}$.

Now in the case where f is a sum of positive products, we shall use cofactors to denote, by slight abuse, simplified formulas.

Definition 9.2.6 — Cofactor of a Sum of Positive Products. Let φ be a sum of positive products built over a set of variables $\mathcal{V}$ and let E be a variable of $\mathcal{V}$. Then, the *cofactor* of φ for E and $\overline{E}$, denoted $\varphi\,|_E$ and $\varphi\,|_{\overline{E}}$, are defined as follows.

$$\begin{aligned}\varphi\,|_E &\overset{def}{=} \{\pi ; E \cdot \pi \in \varphi\}\\ \varphi\,|_{\overline{E}} &\overset{def}{=} \{\pi \in \varphi ; E \notin \pi\}\end{aligned}$$

We can now state the pivotal decomposition theorems, on which binary decision trees, binary decision diagrams and zero-suppressed binary decision diagrams rely.

Theorem 9.2 — Shannon Decomposition. Let f a Boolean function built over a set of variables $\mathcal{V}$ and $E \in \mathrm{var}(f)$. Then the following equality holds.

$$f \equiv E \cdot f\,|_{E=1} + \overline{E} \cdot f\,|_{E=0}$$

The application of the Shannon decomposition theorem to the particular case of sums of products gives raise to the following corollary.

Corollary 9.3 — Pivotal Decomposition of Sum of Positive Products. Let φ be a sum of positive products built over a set of variables $\mathcal{V}$ and let E be a variable showing up in φ. Then, we can decompose φ according to E as follows.

$$\varphi = E \odot \varphi\,|_E \cup \varphi\,|_{\overline{E}}$$

9.2.4 Probability Assessment

The probability of a sum of disjoint products is simply the sum of the probabilities of its products (which are trivially calculated). Consequently, if we manage to get a sum of disjoint products equivalent to the Boolean function associated with the top event, we can calculate exact values of probabilistic risk indicators in an efficient way. The problem is indeed to get this sum of disjoint products.

As we shall explain in details Chapter 13, it is in theory possible to calculate the exact probability of a sum of minimal cutsets using the Sylvester-Poincaré development. However, the computational cost of this calculation would be prohibitive. Therefore, only approximated values are calculated.

Two approximations methods have proposed so far in the literature the rare event approximation and the mincut upper bound, see e.g. (Kumamoto and Henley 1996). XFTA implements both as well as a third, original, one called pivotal upper bound.

9.3 Data Structures

Sums of disjoint products and sums of minimal cutsets can be significantly larger than the systems of stochastic Boolean equations they are issued from. This is the reason why they have to be stored efficiently in the computer memory. This section reviews four data structures that can be used for this purpose.

9.3.1 Sparse Matrices

A first idea is to store products into a two dimensional matrix. Each row of the matrix encodes a product. Columns encode the variables. The cell (i, j) contains 1 if the product i contains positively the variable j, -1 if it contains it negatively and 0 if it does not contain it.

Storing sums-of-products into such matrices would be however over costly as the matrices would be extremely sparse, i.e. would contain 0's almost everywhere.

A solution consists then in using sparse matrices, which are very common in numerical analysis, see e.g. (Cormen et al. 2001) for a reference textbook on algorithms and data structures. In sparse matrices, each non empty cell is encoded by a separate object and cells are connected in row and in column as illustrated by the following example.

■ **Example 9.5** The sum of minimal cutsets of Example 9.4 can be encoded by the sparse matrix pictured in Figure 9.6. ■

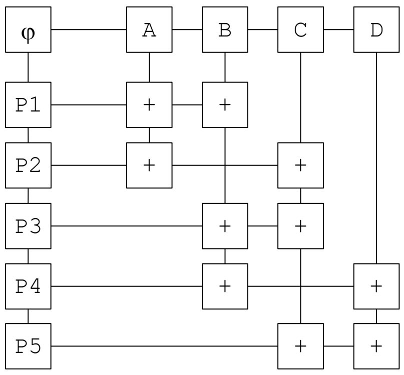


Figure 9.6: Sparse matrix encoding the sum of minimal cutsets of Example 9.4

Sparse matrices are thus much more efficient than plain matrices to store sums of products. Moreover, they make it possible to sort products (in view of printing them out) according to their probabilities or any other measure. They are however still very memory consuming. Moreover, most of the computations of probabilistic indicators require at least to visit once each cell of the matrix, which may be costly at the end.

9.3.2 Binary Decision Trees

Binary decision trees, BDT for short, are a more compact way (than sparse matrices) to encode sums of positive products. Applying the pivotal decomposition recursively, it is possible to decompose fully any sum of positive products. The resulting formula takes the form of a binary tree and can be implemented as such. We call this data structure a binary decision tree. Formally:

Definition 9.3.1 — Binary Decision Trees. Let be a set of variables $\mathcal{V}$. A *binary decision tree* over $\mathcal{V}$ is a tree with two types of nodes.

- Leaves that are labeled with numbers. Two leaves must be labeled with two different numbers.
- Internal nodes that are labeled with variables of $\mathcal{V}$ and that have two outedges pointing to subtrees in which the variable labeling the node does not show up. These two outedges are called respectively the *then-edge* and the *else-edge*. The else-edge may be point to no node.

The node pointed by the *then-edge* is called the *then-child*. Similarly, the pointed by the *else-edge*, if any, is called the *else-child*.

Let $\sqsubset$ be an order over the variables of $\mathcal{V}$ A binary decision tree is *ordered* if for any two internal nodes n_1 and n_2, labeled respectively with the variables E_1 and E_2 and such that n_2 is either the then-child or the else-child of n_1, we have $E_1 \sqsubset E_2$.

From now, we shall assume implicitly that binary decision trees are ordered.

■ **Example 9.6** The binary decision tree encoding the sum of minimal cutsets of Example 9.4 for the order $A \sqsubset B \sqsubset C \sqsubset D$ is pictured in Figure 9.7. Then-outedges are represented with plain lines. Else-outedges, when present, are represented with dashed lines. ■

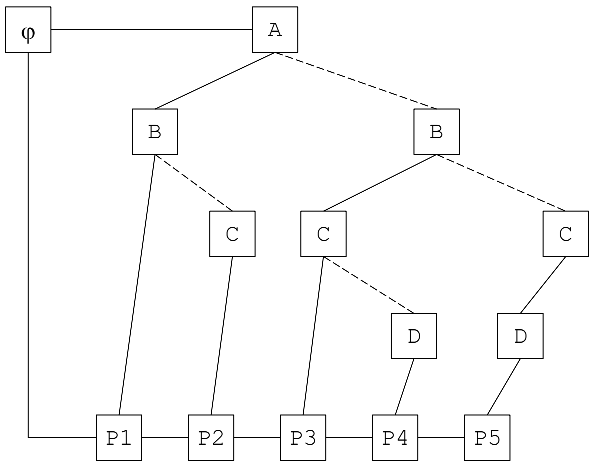


Figure 9.7: Binary decision tree encoding the sum of minimal cutsets of Example 9.4

A binary decision tree encodes a (fully decomposed) sum of positive products:
- Leaves encode the empty product (or equivalently the Boolean constant 1).
- An internal node labeled with a variable E encodes the sum of positive products $E \odot \varphi_1 \cup \varphi_0$, where φ_1 and φ_0 are the sums of positive products encoded by respectively its then-child and its else-child. If the node has no else-child, then $\varphi_0 = 0$.

Several important remarks at this point:
- Binary decision trees tend to be more compact encodings of sum of positive products than sparse matrices. However, the size of a binary decision tree is still roughly proportional to the size of the sum of positive products it encodes.
- Adding or removing products from a binary decision tree can be done efficiently, using top-down algorithms.
- Products encoded by a binary decision tree can be accessed either top-down, starting from the root node of the tree, or bottom-up starting from the leaves.
- Calculations of probabilistic risk indicators can thus be performed either top-down or bottom-up. We shall Chapter 13 that top-down calculations are much more efficient.

- As all products of the sum of products can be accessed via the leaves of the binary decision tree, it is possible to sort them in decreasing order of their probability (or any other convenient order). This, in turn, makes it possible to maintain efficiently the set of the n most important minimal cutsets, a feature upon which the minimal cutsets approach relies heavily.
- It is possible to encode sums of non-necessarily positive products by means of binary decision trees. The idea is simply to use nodes labeled with literals rather than with variables. The binary decision tree must then obey an additional construction rule preventing two opposite literals to show up along a branch of the tree.

For all these reasons, XFTA uses binary decision trees rather than sparse matrices to encode sum of minimal cutsets. It also uses zero-suppressed binary decision diagrams for the same purpose, as we shall see Section 9.3.4.

9.3.3 Binary Decision Diagrams

Binary decision diagrams, BDD for short, are a data structure to store and to perform operations on Boolean functions. Their modern implementation has been first described by Bryant & al. (Brace, Rudell, and Bryant 1990; Bryant 1986). They rely on the Shannon decomposition. As for binary decision tree, it is possible to apply this principle recursively so to encode any Boolean function into a binary tree. However, binary decision diagrams provide a much more efficient way to encode Boolean functions thanks to the sharing of equivalent subtrees and the removal of useless nodes. Formally,

Definition 9.3.2 — Binary Decision Diagrams. Let be a set of variables $\mathcal{V}$. A *binary decision diagram* over $\mathcal{V}$ is a directed acyclic graph with two types of nodes.

- Leaves that are either labeled with the Boolean constants 0 and 1. Two leaves must be labeled with two different numbers.
- Internal nodes that are labeled with variables of $\mathcal{V}$ and that have two outedges pointing to subgraphs in which the variable labeling the node does not show up. These two outedges are called respectively the *then-edge* and the *else-edge*.

The node pointed by the *then-edge* is called the *then-child*. Similarly, the pointed by the *else-edge*, if any, is called the *else-child*.

Let $\sqsubset$ be an order over the variables of $\mathcal{V}$ A binary decision diagram is *ordered* if for any two internal nodes n_1 and n_2, labeled respectively with the variables E_1 and E_2 and such that n_2 is either the then-child or the else-child of n_1, we have $E_1 \sqsubset E_2$.

A binary decision diagram is *reduced* if the following conditions hold.

1. It contains no node with identical children. Such a node is actually useless as it encodes the same function as its child ($E \cdot f + \overline{E} \cdot f \equiv f$).
2. It contains only one leaf labeled with 0 and one leaf labeled with 1.
3. It contains no two nodes labeled with the same variable and pointing to the same then- and else-children. As these two nodes would represent the same function, one is actually sufficient.

From now, we shall simply say binary decision diagram for reduced ordered binary decision diagram.

■ **Example 9.7** The binary decision tree encoding the sum of disjoint products of Example 9.3 for the order $A \sqsubset B \sqsubset C \sqsubset D$ is pictured in Figure 9.8. Then-outedges are represented with plain lines. Else-outedges are represented with dashed lines. ■

We shall discuss further the implementation of binary decision diagrams Chapter 12.

For now, two important remarks:

- For many practical models, binary decision diagrams are much more compact than the sums

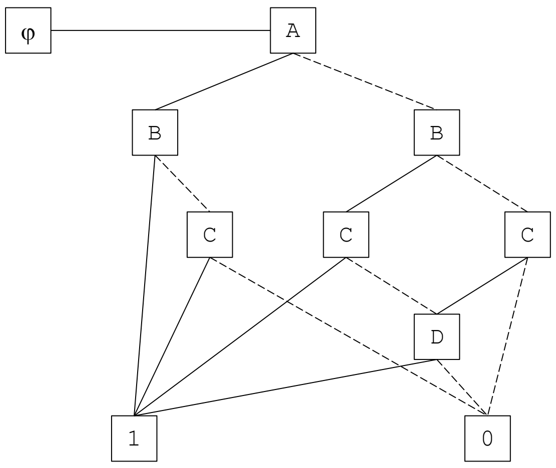


Figure 9.8: Binary decision diagram encoding the sum of disjoint products of Example 9.3

of disjoint products they encode. This said, there is no silver bullet. For very large models, it may be not possible to fit them in computer memory.

– As they encode in a compact way sums of disjoint products, binary decision diagrams make it possible to calculate very efficiently the exact value of the top event probability and beyond of all probabilistic risk indicators.

9.3.4 Zero-Suppressed Binary Decision Diagrams

Zero-suppressed binary decision diagrams, ZBDD for short, have been introduced by Minato (Minato 1993). They are just like binary decision diagrams except that they encode sums of positive products rather than sum of disjoint products. In other words, they rely on the pivotal decomposition (Corollary 9.3) rather than on the Shannon decomposition (Theorem 9.2).

Consequently, the first reduction rule is changed as follows (the two other ones stay the same).

1. A Zero-suppressed binary decision diagram is reduced if it contains no node whose then-child is the leaf 0 (hence the name zero-suppressed) as this node would be equivalent to its else-child: $E \odot \emptyset \cup \varphi = \varphi$.

■ **Example 9.8** The zero-suppressed binary decision diagram encoding the sum of minimal cutsets of Example 9.4 for the order $A \sqsubset B \sqsubset C \sqsubset D$ is pictured in Figure 9.8. Then-outedges are represented with plain lines. Else-outedges are represented with dashed lines.

Yes, this zero-suppressed binary decision diagram is the same as the binary decision diagram encoding the sum of disjoint products of Example 9.3. This happens quite frequently with coherent models. ■

Several important remarks.

– Zero-suppressed binary decision diagrams are often much more compact than binary decision trees encoding the same sum of minimal cutsets.
– As they are based on the same decomposition, top-down calculations of probabilistic risk indicators work alike on both data structures. They are however more efficient on zero-suppressed binary decision diagrams than on binary decision trees because the former are more compact than the latter.
– Conversely to binary decision trees, zero-suppressed binary decision diagrams do no encode products explicitly. Consequently, they cannot be used to sort products in decreasing order of their probabilities, or any other suitable measure.

9.3.5 Handles

As shown Figure 9.1, the assessment flows of XFTA may require to build several data structures. This is obviously the case for the binary decision diagram approach, but this also the case for the minimal cutsets approach as one may be interested in extracting minimal cutsets for different cutoffs, or different mission times. This means that several BDT, BDD and ZBDD may be associated with top events of models.

In XFTA, *handles* are names for data structures encoding formulas in normal forms, i.e. concretely to BDT, BDD and ZBDD associated with variables. More exactly a handle is a pair made of name and a reference to such a data structure. For instance, when minimal cutsets are extracted directly from the target model, by means of the command `build BDT`, the resulting binary decision tree is referred to by means of a handle. By default, this handle is named `BDT`. It is possible to change this default naming. Similarly, the binary decision diagram associated with the top event and the zero-suppressed binary decision diagram encoding its minimal cutsets are referred to by handles respectively named `BDD` and `ZBDD`.

Now assume for instance we want to extract, by the minimal cutsets approach, two sums of minimal cutsets, typically calculated at mission times 100 and 1000. To do so, we have to store these two sums of minimal cutsets in two different binary decision trees and to name these two BDT differently. This is achieved by means of the option `target-handle`, e.g:

```
build BDT TrackingSystem.failed
   mission-time=100
   target-handle=MCS2;
build BDT TrackingSystem.failed
   mission-time=1000
   target-handle=MCS3;
```

Now we can launch the calculation of probabilistic risk indicators from either sum of minimal cutsets. This is achieved by means of the option `source-handle`, e.g:

```
compute probability TrackingSystem.failed
   source-handle=MCS3
   quantification-method=pivotal-upper-bound
   mission-time=[100, 1000, 10000]
   output="TrackingSystem-prb.tsv";
```

For the sake of memory consumption, it is advised nevertheless not to multiply the number of stored sums of products, whether as BDT, BDD or ZBDD.

Handles of the target model can be printed out by means of the command `print handles`. E.g.

```
set option output="TrackingSystem-Handles.tsv";
set option mode write;
print handles;
```

The above command gives, for each handle of the model, the variable owning the handle, the name of the handle, its type and its size (the number of nodes of the binary decision tree or the binary decision diagram).

9.3.6 Complexity of Operations

As we shall see in the forthcoming chapters, XFTA implements quite a few calculations on binary decision trees, binary decision diagrams and zero-suppressed binary decision diagrams. As a general rule, a calculation which is implemented on one data structure is also implemented on the two others, at least if it makes sense. In particular, unless explicitly stated otherwise, calculations implemented on binary decision trees are implemented on zero-suppressed binary decision diagrams, and vice-versa, as both data-structures encode sums of minimal cutsets.

All implemented calculations work by traversing the data structure in some way. Nevertheless, they may have different complexities. In the sequel, we shall use the following notations.

Definition 9.3.3 — Sizes of Formulas. Let f be a Boolean formula built over a set of variables $\mathscr{V}$. Then,

- $|\mathrm{var}(f)|$ denotes the number of variables showing up in f.
- $|f|$ denotes the size of the formula f, i.e. the sum of number of occurrences of variables and the number of connectives use to write f.

Let φ be either a binary decision tree, or a binary decision diagram or a zero-suppressed binary decision diagram built over $\mathscr{V}$. φ encodes thus a certain sum of products $\mathrm{SoP}(\varphi)$. We can then distinguish two sizes:

- $|\varphi|$ which denotes the size, i.e. the number of nodes, of the data structure φ.
- $|\mathrm{SoP}(\varphi)|$ which denotes the size of, i.e. the number of literal occurrences showing up in, $\mathrm{SoP}(\varphi)$.

The following property holds that relates these measures.

Property 9.4 — Sizes of Formulas. Let f be a Boolean formula built over a set of variables $\mathscr{V}$ and let φ be either a binary decision tree, or a binary decision diagram or a zero-suppressed binary decision diagram built over $\mathscr{V}$ and encoding either the structure function (in case φ is a binary decision diagram) or the minimal cutsets (in case φ is either a binary decision tree or zero-suppressed binary decision diagram) of f. The following relations hold.

- The size of φ can be exponentially larger than the size of f (and *a fortiori* than the number of variables showing up in f, i.e. $|f| \ll |\varphi|$.
- The size of the sum of products encoded by a binary decision tree φ is roughly proportional to the size of that binary decision tree, i.e. $|\mathrm{SoP}(\varphi)| \approx |\varphi|$.
- The size of the sum of products encoded by a binary decision diagram or a zero-suppressed binary decision diagram φ can be exponentially larger than the size of that diagram, i.e. $|\varphi| \ll |\mathrm{SoP}(\varphi)|$.

To express the complexity of calculations, we shall use the big $\mathscr{O}$ notation (we give it here in a simplified form which sufficient for our purpose).

Definition 9.3.4 — Big $\mathscr{O}$ notation. Let f and g be two functions from $\mathbb{R}^+$ to $\mathbb{R}^+$. Then f is in big $\mathscr{O}$ of g, denoted as $f = \mathscr{O}(g)$, if and only if there exists a positive real number c and a real number x_0 such that:

$$\forall x \geq x_0,\ f(x) \leq c \times g(x)$$

In other words, saying that f is in big $\mathscr{O}$ of g is saying that $f(x)$ is roughly proportional to $g(x)$, at least asymptotically.

Putting things together, we shall say that:

- A calculation is in $\mathscr{O}(|\varphi|)$ if it requires a number of basic operations and therefore a computation time which is roughly proportional to the size of the diagram φ.
- A calculation is in $\mathscr{O}(|\mathrm{SoP}(\varphi)|)$ if it requires a number of basic operations and therefore a computation time which is roughly proportional to the size of the sum of products encoded by the diagram φ.

⚠ **Calculations in $\mathcal{O}(|\mathrm{SoP}(\varphi)|)$**
Calculations in $\mathcal{O}(|\mathrm{SoP}(\varphi)|)$ can be practically infeasible when performed on binary decision diagrams and zero-suppressed binary decision diagrams.

9.4 Assessment Methods

9.4.1 Two Approaches, Three Methods

XFTA implements the two main algorithmic approaches have been proposed to assess fault trees, namely the minimal cutsets and the binary decision diagrams approaches.

The minimal cutsets approach has been historically the first one to be developed. This started with the MOCUS method (Fussel and Vesely 1972), which is a top-down algorithm. Modern versions of this algorithm are implemented in RiskSpectrum® (Berg 1994) and in XFTA (Rauzy 2003; Rauzy 2012).

Note that the reliability engineering literature makes no mention of the data structures in which minimal cutsets are encoded. These data structures play however a central role in the efficiency of the approach, at least when the number of minimal cutsets to be consider is large (say bigger than a few thousands minimal cutsets). Although it just assembles otherwise well known ideas, the binary decision trees data structure, as described Section 9.3.2, is original.

Another approach to extract minimal cutsets has been proposed in 2004 by Jung & al. (Jung, Han, and Ha 2004). It works bottom-up by constructing the set of minimal cutsets of an intermediate event by combining the sets of minimal cutsets of the events showing up in its definition. Sums of minimal cutsets are encoded by means of zero-suppressed binary decision diagrams. This method is implemented in FTREX®. It is not implemented in the current version of XFTA.

As pointed out above, in the minimal cutsets approach, probabilistic risk indicators are estimated using either the rare event approximation or the mincut upper bound. Both give pessimistic results, i.e. the top event probability—and therefore the other indicators—is overestimated. XFTA implements a third quantification method, the pivotal upper bound, that gives in general better—but still pessimistic—estimates than the rare event approximation and the mincut upper bound.

The binary decision diagram approach has been introduced independently by Madre and Coudert (Coudert and Madre 1993) and the author (Rauzy 1993; Rauzy 2008a). Once the binary decision diagram encoding the top event built, it is possible to compute efficiently exact values of probabilistic risk indicators. It is also possible to build a zero-suppressed binary decision diagrams that encodes the minimal cutsets.

These various assessment methods can be compared along three lines:

- First, their efficiency, i.e. the amount of calculation resources they require. This is a key issue as fault tree quantification is a hard problem in the sense of computation complexity theory, see e.g. (Papadimitriou 1994) for an introductory textbook. The efficiency should be measured not only, nor even primarily, via the computation times, but rather by the computer memory consumption, which is the real limiting factor.
- Second, their accuracy. As explained above, the values of probabilistic risk indicators calculated from the minimal cutsets are only estimates. These estimates are pessimistic. This is acceptable from a safety point of view, as the risk is always overestimated. This may however be problematic if results are too pessimistic as it may lead to reject otherwise suitable designs and to put the emphasize on the wrong points.
- Third, the way they handle non-coherent models. A sum of minimal cutsets is actually always coherent. This means that the minimal cutsets extraction process transforms a possibly non-coherent model into a coherent one. The question is thus: what is the status of this transformation?

Table 9.1 makes a summary of approximation sources and effects of approximations in quantification methods. Note that the higher the probabilities at stake, the higher the effects in both directions.

Table 9.1: Sources and effects of approximations

Source of the approximation	Effect of the approximation
Coherent approximation of non-coherent models	Pessimistic
Cutoffs	Optimistic
Estimates of probabilistic risk indicators	Pessimistic

Before concluding this chapter, it is worth to discuss in further details the question of non-coherent models.

9.4.2 Non-Coherent Models

9.4.2.1 Non-Coherence of Systems versus Non-Coherence of Models

In fault trees and related models, basic events are supposed represent failed states of components of the system under study. As a general rule in all industrial systems, the more there are components failed, the more likely the system as a whole is failed. There may be exceptions to this rule, but they involve in general complex temporal dependencies. As a typical illustration consider a component monitored by some device that raises an alarm when the component fails. If the component fails before the device, the alarm is raised and the system goes to failed safe state. If, on the contrary, the device fails before the component, the system goes on its mission until the component fails, which remains undetected.

Fault trees and related models cannot be used to represent such dynamic behaviors. This is the reason why the underlying Boolean functions are expected to be monotonically increasing: if $f(\ldots,0,\ldots)=1$ then $f(\ldots,1,\ldots)=1$ as well. In other words, if in a given global state the top event is realized while a given basic event is not, then the top event is still realized in the state where the basic event is realized, *mutatis mutandis*. The reliability engineering literature uses the term *coherent* for this property, instead of the usual term monotone used in mathematics, see Section 5.7. This emphasizes that well designed models are expected to have this property.

There is a very simple syntactic conditions to ensure the coherence of a model: a model built only with connectives $\wedge$ (and), $\vee$ (or) and k/n (k-out-of-n) is always coherent. This condition is sufficient, but not necessary. We shall give a formal definition of coherence Chapter 11. For the current discussion, the intuitive definition is enough.

Now the question is: can models be non-coherent, i.e. involve negations in a way or another? In industrial practice, there are actually non-coherent models, although these are rather exceptional, see e.g. (Nusbaumer and Rauzy 2013) for a discussion.

The above question should actually be further decomposed into two questions, one about the systems under study and the other one about the models:

1. Is there systems showing a non-coherent behavior?
2. Can negations (and more generally non-monotone connectives) be used to simplify models, i.e. to design non-coherent models that will be made coherent by the assessment process?

I think we can answer negatively to the first question: there may be non-coherent behaviors in real systems, as illustrated above, but these behaviors cannot be represented in a purely Boolean logic, even using non-coherent models.

I must admit that I changed my mind regarding the second question. I think now that yes, such modeling “tricks” can be used, although this should be done with much care.

9.4.2.2 Using Non-Coherence Without Meaning It

As an illustration, consider again the coolant supply system presented in Section 7.2.2.3. We reproduce here the diagram for the sake of easiness.

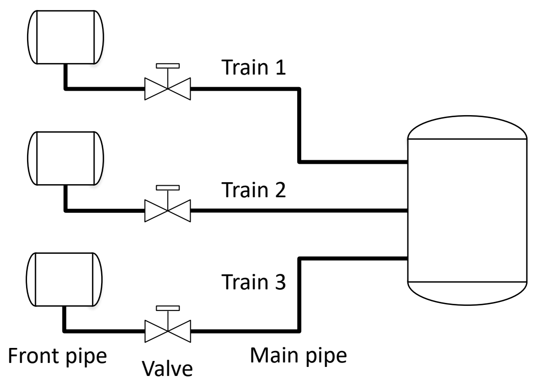


Figure 9.9: A coolant supply system

Eventually, this system can be represented by the following system of stochastic Boolean equations.

$$
\begin{array}{rclrclrcl}
\mathtt{SF} & = & \mathtt{SUF} \wedge \mathtt{OR} & \mathtt{SUF} & = & \mathtt{atleast}\,2(\mathtt{T1F}, \mathtt{T2F}, \mathtt{T3F}) & & & \\
\mathtt{T1F} & = & \mathtt{T1FO} \vee \mathtt{T1FM} & \mathtt{T1FO} & = & \mathtt{LBFP1} \vee \mathtt{LBMP1} & \mathtt{T1FM} & = & \mathtt{SBFP1} \vee \mathtt{SOV1} \\
\mathtt{T2F} & = & \mathtt{T2FO} \vee \mathtt{T2FM} & \mathtt{T2FO} & = & \mathtt{LBFP2} \vee \mathtt{LBMP2} & \mathtt{T2FM} & = & \mathtt{SBFP2} \vee \mathtt{SOV2} \\
\mathtt{T3F} & = & \mathtt{T3FO} \vee \mathtt{T3FM} & \mathtt{T3FO} & = & \mathtt{LBFP3} \vee \mathtt{LBMP3} & \mathtt{T3FM} & = & \mathtt{SBFP3} \vee \mathtt{SOV3} \\
\mathtt{OR} & = & \multicolumn{7}{l}{\mathtt{atmost}\,1(\mathtt{T1FM}, \mathtt{T2FM}, \mathtt{T3FM})}
\end{array}
$$

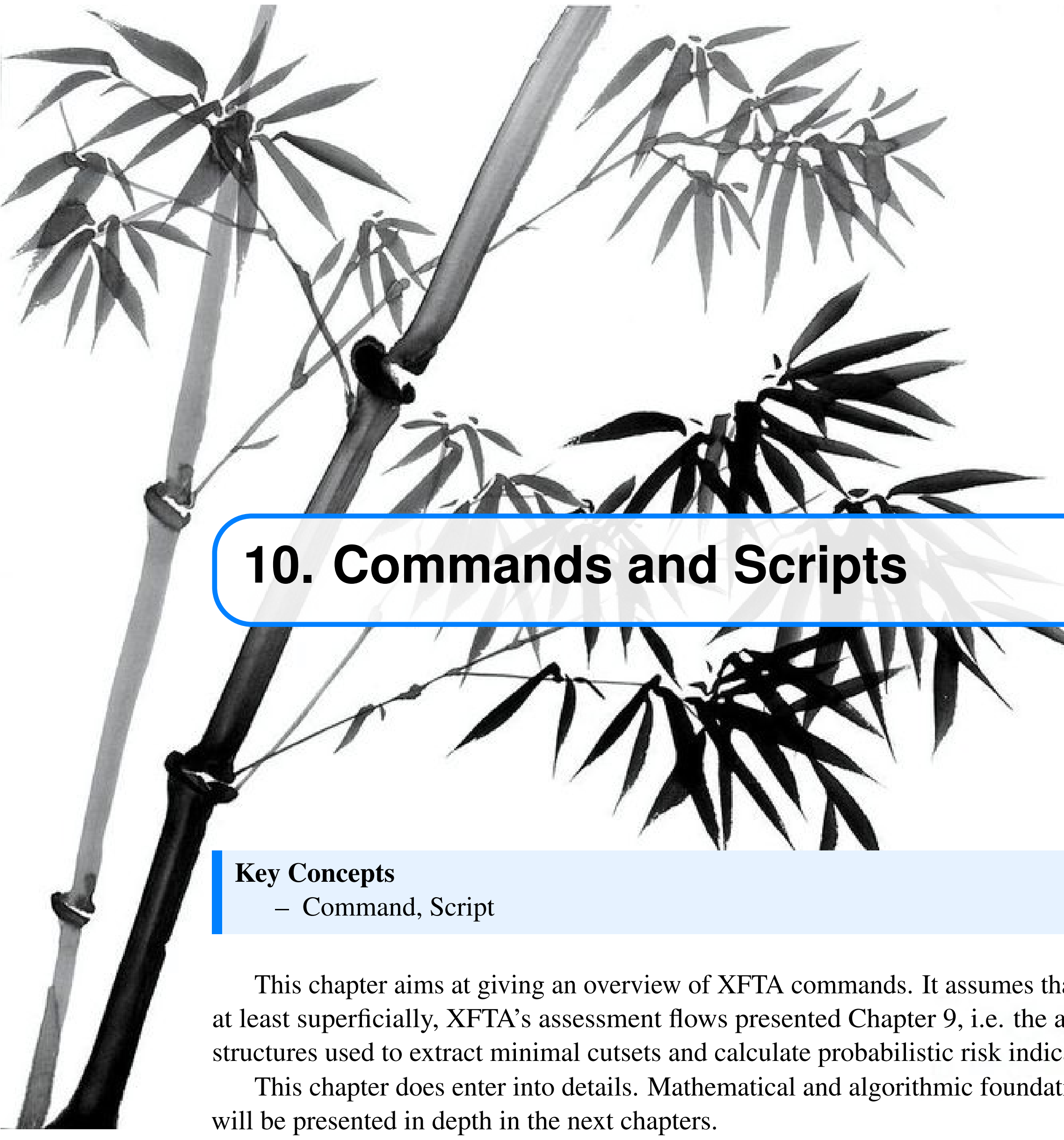
Where:
- `SF` stands for system failed.
- `SUF` stands for system unconditionally failed.
- `OR` stands for operation rules.
- `T`*`i`*`F` stands for train *i* failed.
- `T`*`i`*`FO` stands for train *i* failed in operation.
- `T`*`i`*`FM` stands for train *i* failed in maintenance.
- `SBFP`*`i`* stands for small breach in front pipe of train *i*.
- `LBFP`*`i`* stands for large breach in front pipe of train *i*.
- `LBMP`*`i`* stands for large breach in main pipe of train *i*.
- `SOV`*`i`* stands for spurious opening of valve of train *i*.

This model is made of two parts: a description of failure conditions of the system and a description of operation rules. The first part is coherent, as expected. The second part is not: operation rules prevent to have more than one train in maintenance at a time.

Each train can fail for four different reasons:
- small breach in the front pipe (while in maintenance): `SBFP`*`i`*,
- large breach in the front pipe (while in operation): `LBFP`*`i`*,
- spurious opening of the valve (while in maintenance): `LBMP`*`i`*,
- large breach in the main pipe (while in operation): `SOV`*`i`*.

As the system is failed if at least two of its three trains are failed. This gives potentially $4^3 = 64$ minimal cutsets. However, as only one train can be maintained at a time, due to operation rules, there are thus only 36 minimal cutsets.

Now the question is the following: should our non-coherent model be considered *per se*, or as a mean to generate the minimal cutsets? In terms of probabilities, should we consider the probability of the structure function of the top event, or should we consider the probability of the sum of its minimal cutsets? There is indeed no clear, universal, answer to the above question.



10. Commands and Scripts

Key Concepts
– Command, Script

This chapter aims at giving an overview of XFTA commands. It assumes that the reader knows, at least superficially, XFTA's assessment flows presented Chapter 9, i.e. the algorithms and data structures used to extract minimal cutsets and calculate probabilistic risk indicators.

This chapter does enter into details. Mathematical and algorithmic foundations of calculations will be presented in depth in the next chapters.

10.1 Scripts

10.1.1 Illustrative Example

To illustrate the use of commands and scripts, we shall use, throughout this chapter, the small monitoring system pictured in Figure 10.1.

This system is made of three identical lines `L1`, `L2` and `L3`, receiving information from a perfectly reliable source `S`. Each line consists of an acquisition module `A` and a calculator `C`. Results of calculations are sent to a voter `V` that works according to a 2 out of 3 logic.

Figure 10.2 gives a possible model of this system.

We shall assume that this model is stored in the file "`Models/MonitoringSystem.sbe`", i.e. in the file "`MonitoringSystem.sbe`" located in the sub-folder "`Models`" of the current folder.

10.1.2 Sessions

XFTA works by executing commands gathered into scripts. Scripts are stored into ordinary text files. Although this is not mandatory, script files have usually the extension `.xfta`.

Assume we create a script to assess the model given in Figure 10.2 and that we store this script in the file "`Scripts/MonitoringSystem.xfta`". To execute this script with XFTA, it suffices to call the executable `xftar.exe` with the name of the script file as argument:

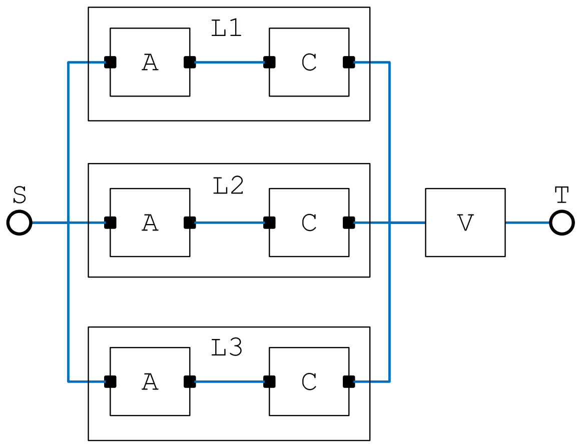


Figure 10.1: A monitoring system

```
xftar.exe Scripts/MonitoringSystem.xfta
```

10.1.3 Commands

A *script* is a sequence of *commands*. Commands of a script are executed in a row. Figure 10.3 shows a script implementing the minimal cutsets approach on our model.

Commands are terminated with a semicolon. This makes it possible for a command to spread on several lines.

Commands have a name which is an action verb, e.g. `load`, `build`, `compute`, `print`. After the name comes the object on which the action is applied, e.g. `model`, `target-model`, `BDT`, `minimal-cutsets`. Then come possibly arguments, e.g. the name of the top event, e.g. `Main.failed`, and options, e.g. `output="MonitoringSystem-mcs.tsv"`, `mission-time=[730, 4380, 8760]`. Arguments are mandatory and consist of one value. Options are optional and consist of an identifier followed by the symbol =, itself followed by a value. Values can be atomic like an identifier, a string or a number, or complex expressions, including lists such as `[730, 4380, 8760]`.

Each command requires thus parameters that are specific to that command. These parameters are given either via arguments or via options (but never both). Some parameters are optional because XFTA manages a set of *environment variables*. These environment variables record default values for parameters of commands. We shall this point in more details Section 10.2.

10.1.4 Loading and Printing Models

XFTA provides a unique command to load models, namely `load model`. Models can be written at two formats: the Open-PSA format and S2ML+SBE. When loading a file containing a model, the format of this file (`open-psa` or `s2ml-sbe`) is specified with the option `format`. E.g.

```
load model "MonitoringSystem.opsa" format=open-psa;
```

By default the format is `s2ml-sbe`. It is possible to change this default behavior by changing the value of the environment variable `format`.

A model can be stored into several files. In this case, the command `load model` must be called on each file of the model. E.g.

```
class BasicBlock
   state failed = 0;
   flow in = false;
   flow out = in and not failed;
end

class AcquisitionModule
   extends BasicBlock;
   state failed = Weibull(0, alpha, beta);
   parameter alpha = 1.0e+3;
   parameter beta = 3;
end

class Calculator
   extends BasicBlock;
   state failed = exponential(lambda);
   parameter lambda = 1.0e-4;
end

class Voter
   extends BasicBlock;
   state failed = exponential(lambda);
   parameter lambda = 1.0e-7;
   flow in1 = false;
   flow in2 = false;
   flow in3 = false;
   flow in = atleast 2(in1, in2, in3);
end

block Main
   source S = true;
   block L1
      AcquisitionModule A;
      Calculator C;
      flow C.in = A.out;
   end
   clones L1 as L2;
   clones L1 as L3;
   Voter V;
   flow L1.A.in = S;
   flow L2.A.in = S;
   flow L3.A.in = S;
   flow V.in1 = L1.C.out;
   flow V.in2 = L2.C.out;
   flow V.in3 = L3.C.out;
   flow failed = not V.out;
end
```

Figure 10.2: A possible model for the monitoring system pictured in Figure 10.1

```
load model "Components.sbe";
load model "MonitoringSystem.sbe";
load model "Models/MonitoringSystem.sbe";
build target-model;
build BDT Main.failed;
print minimal-cutsets Main.failed
   mission-time=8760
   output="Results/MonitoringSystem-mcs.tsv";
compute probability Main.failed
   quantification-method=pivotal-upper-bound
   mission-time=[730, 4380, 8760]
   output="Results/MonitoringSystem-prb.tsv";
```

Figure 10.3: Basic script implementing the minimal cutsets approach

File names and paths must be surrounded with quotes, e.g. `"../Models/FT1.opsa"`.

XFTA provides two forms of the command `print` to print models: `print model` that prints the source model, and `print target-model` that prints the model once instantiated, flattened and possibly preprocessed for one or more target variables.

Models can be printed at either OpenPSA or S2ML+SBE formats. The format (`open-psa` or `s2ml-sbe`) is specified by means of the option `format`. E.g.

```
print model output="src-model.sbe";
print target-model output="tgt-model.opsa" format=open-psa;
```

10.1.5 Loading and Executing Scripts

It is possible to split a script into several files, i.e. concretely to load a script by means of the command `load script`. E.g.

```
load script "Scripts/MCSApproach.xfta";
load script "Scripts/Calculations.xfta";
```

File names must be surrounded with quotes, as illustrated above.

10.1.6 Result Files

Commands `print` and `compute` print out data into TSV files. The name of the file, or more exactly its path, is specified using the option `output`. File names must be surrounded with quotes, e.g. `"../Results/FT42-mcs.tsv"`.

Note that on Windows® file paths can be written using an backslash “\” instead of a slash “/” as separator, e.g. “`..\Results\MonitoringSystem-prb.tsv`”.

Result files can be opened either in writing or in appending mode. In the first case, the file is overwritten it exists already, in the second the information is added at the end of the file. In both cases, a new file is created if no file with that name exists. The option `mode` is used to specify the mode. Its two possible values are `write` and `append`. E.g.

```
compute probability Main.failed
   quantification-method=rare-event-approximation
   mission-time=[730, 4380, 8760]
   output="Results/MonitoringSystem-prb.tsv"
   mode=write;
compute probability Main.failed
   quantification-method=pivotal-upper-bound
   mission-time=[730, 4380, 8760]
```

Table 10.1: Options to set default result file and writing mode

Option	Type	Default value
`output`	String	`result.txt`
`mode`	`write/append`	`write`

```
   output="Results/MonitoringSystem-prb.tsv"
   mode=append;
```

The first command overwrites the file "`Results/MonitoringSystem-prb.tsv`" (if it exists), while the second appends the results at the end of this file.

⚠ **Folders**
The current version of XFTA does not create folders, which means that attempting to write a file located in a folder that does not exists results in an error.

Default value for options `output` and `write` are as given table 10.1.

10.1.7 Comments

It is possible to insert comments into scripts. The syntax is the same as for S2ML+SBE, i.e.

– Two slashes "`//`" indicate that the remainder of the line is a comment, e.g.

```
load model "Models/Components.sbe"; // Library
load model "Models/MonitoringSystem.sbe"; // Model
```

– The text, possibly spreading over several lines, between "`/*`" and "`*/`" is a comment, e.g.

```
/*
 * Assessment of Monitoring System
 * Version: 1.2.3
 */
```

10.1.8 Errors

⚠ **Errors during the execution of scripts**
Some commands may result in an error, e.g. if one attempts to load a file that does not exists, or to save results into a unreachable file. In this case, one or more error messages are displayed and the execution of the script is stopped, which means that XFTA exits after having cleaned up its internal memory.

10.2 Options and Environment Variables

10.2.1 Arguments versus Options

As we have seen above, most of XFTA commands take arguments and options. E.g.

```
compute probability Main.failed
   quantification-method=rare-event-approximation
   mission-time=[730, 4380, 8760]
   output="Results/MonitoringSystem-prb.tsv"
   mode=write;
```

The above command (`compute`) has two arguments, `probability` and `Main.failed`, and four options, `quantification-method=...`, `mission-time=...`, `output=...`, and `mode=...`.

Arguments are mandatory. In the above command, the first argument `probability` specifies what to compute, name the probability, and the second one for which variable (top event) this probability must be computed.

The reason why options are... optional is that there is a mean to determine their value, even though this value is not specified by the command. Except for handles (discussed Section 9.3.5), default values of options are specified by means of environment variables.

10.2.2 Environment Variables

Environment variables have the same names as the corresponding options. The command `set option` is used to set default values of options. It takes two additional arguments: first, the name of the option, which is an identifier and its values, which is an expression. The value of that expression can be:

– A Boolean constant, i.e. either `true` or `false`.
– An integer.
– A floating point number.
– An identifier. Identifiers in XFTA script obey the same rules as in Open-PSA or S2ML+SBE. They start with a letter or an underscore "_" which is followed by any number of letters, digits, underscores and hyphens '-".
– A Path. Paths are references to model variables. They consists of any number of identifiers separated with dots "`.`".
– A string i.e. any text surrounded with quotes "`"`". Strings are used in particular to set file names.
– A list of values, in particular a list of mission times. Lists start with a left bracket "`[`" and end with a right bracket "`]`". Elements of a list are separated with commas "`,`".

Assume for example we want to change the default mission time to 8760 and the default output file so that we can accumulate results of calculations in that file. Then we can use the following commands.

```
set option mission-time 8760;
set option output "Results/MonitoringSystem-prb.tsv";
set option mode append;
```

Values of environment variables can be printed out by means of the commands `print option` that prints the value of the variable given as argument, and `print options` that prints the values of all environment variables. E.g.

```
print option print-minimal-cutset-order;
print option mission-time;
print options output="options.xfta";
```

Values of options are printed out as `set option` commands so that they can be modified and reloaded later.

Options for these two commands those given in Table 10.1.

10.3 Overview of XFTA Commands

The current version of XFTA relies on six commands: `build`, `compute`, `load`, `print`, `reset` and `set`.

This section reviews the different forms of these commands in turn (in alphabetic order). It does not aim at explaining in depth what they are doing, nor to list their options. Rather, it lists them and refers to the other parts of the guide in which they are presented in full details.

10.3.1 Command `build`

The command `build` builds data structures (BDT, BDD or ZBDD) encoding sums of minimal cutsets (for BDT and ZBDD) or sums of disjoint products (for BDD) associated with a variable (usually the top event) of the model.

`build target-model;`
: This command builds the target model from the (previously loaded) source model. The target model is a pure system of stochastic Boolean equations. To pass from the source model to the target model, object-oriented constructs are instantiated, the model is flattened and extra-logical constructs (common cause failure groups) are resolved.

`build BDD` *`variable Options`*`;`
: This command builds the binary decision diagram encoding the structure function of the given variable.
Its options are described Section 12.6.1 (page 184).

`build BDT` *`variable Options`*`;`
: This command builds the binary decision tree encoding the minimal cutsets of the given variable.
Its options are described Section 11.4.1 (page 168).

`build ZBDD-from-BDD` *`variable Options`*`;`
: This command builds the zero-suppressed binary decision diagram encoding the minimal cutsets of the given variable from the binary decision diagram encoding the structure function of this variable.
Its options are described Section 12.6.2 (page 185).

`build BDT-from-BDD` *`variable Options`*`;`
: This command builds the binary decision tree encoding the (most probable) minimal cutsets of the given variable from the binary decision diagram encoding the structure function of this variable.
It options are described Section 12.6.3 (page 186).

`build WBDD` *`variable Options`*`;`
: This command builds the binary decision diagram encoding the weighted structure function of the given variable.
Its options are described Section 12.6.4 (page 187).

`build WZBDD-from-BDD` *`variable Options`*`;`
: This command builds the zero-suppressed binary decision diagram encoding the weighted minimal cutsets of the given variable from the binary decision diagram encoding the structure function (weighted or not) of this variable.
Its options are described Section 12.6.5 (page 187).

`build BDD-from-BDT` *`variable Options`*`;`
: This command builds a binary decision diagram encoding the disjunction of the minimal cutsets of encoded by the source binary decision tree.
It options are described Section 12.6.6 (page 188).

`build BDD-from-ZBDD` *`variable Options`*`;`

This command builds a binary decision diagram encoding the disjunction of the minimal cutsets of encoded by the source zero-suppressed binary decision diagram.
It options are described Section 12.6.7 (page 188).

10.3.2 Command `compute`

The primary usage of the command `compute` is to compute probabilistic risk indicators. Quantitative indicators can be calculated as well. The first argument specifies which indicator must be computed. The second one specifies for which variable the indicator must be computed.

Indicators can be calculated from different data structures (BDT, BDD, ZBDD) encoding either sums of minimal cutsets or sums of disjoint products. As explained Section 9.3.5, data structures built for a variable are accessed via handles associated with this variable. Each handle has a name, which is unique. The option `source-handle` specifies from which handle the indicator must be calculated. If this option is not specified, XFTA looks for the first handle, in lexicographic order.

Results of calculations are printed out in TSV files. The option `output` specifies the output file. The option `mode` specifies whether the output file must be opened in `write` or in `append` mode.

Most of the commands described below take additional parameters and options, which are described in subsequent chapters.

10.3.2.1 Probabilistic Risk Indicators

`compute probability` *`variable Options`*`;`

This command computes and prints out the probability of the variable given as argument.
The calculation of the probability of the top event is the topic of Chapter 13 for the most part and of Chapter 16 for what concerns safety integrity levels. Options of the command `compute probability` are described Section 13.2.1 (page 197) and 16.3.1 (page 228).

`compute values-of-parameters` *`variable Options`*`;`

This command computes and prints out the values of parameters involved in the definition of the given variable.
This command is described Section 13.2.2.1 (page 200).

`compute probabilities-of-basic-events` *`variable Options`*`;`

This command computes and prints out the probabilities of basic events (state variables) involved in the definition of the given variable.
This command is described Section 13.2.2.2 (page 200).

`compute importance-measures` *`variable Options`*`;`

This command computes and prints out importance measures of basic events (state variables) involved in the definition of the given variable.
Importance measures are discussed Chapter 14. Options of the command `compute importance-measures` are described Section 14.2.1 (page 210).

`compute group-importance-measures` *`variable group Options`*`;`

This command computes and prints out importance measures of the given group of basic events (state variables) involved in the definition of the given variable.
This command is described Section 14.2.2 (page 212).

`compute sensitivity` *`variable Options`*`;`

This command performs a sensitivity analysis on the probability of the variable given as argument and prints out statistics made during this analysis.
Importance measures are discussed Chapter 15. Options of the command `compute`

`sensitivity` are described Section 15.2.1 (page 220)

`compute failure-intensity` *`variable Options`*`;`
: This command calculates the failure intensity and related indicators for the variable given as argument and prints out results.
Safety integrity levels, which make use of failure intensity, are discussed Chapter 16. Options of the command `compute sensitivity` are described Section 16.3.2 (page 229)

`compute time-to-probability` *`variable probability Options`*`;`
: This command seeks for the mission time at which the probability of a given variable reaches a given value. Technically, this command implements a dichotomic search between a lower and an upper bound.
Options of the command `compute time-to-probability` are described Section 13.2.2.3 (page 200).

10.3.2.2 Qualitative Indicators

Aside probabilistic risk indicators, it is possible to compute various qualitative indicators on sums of disjoint products and sums of minimal cutsets.

`compute size-of-diagram` *`variable Options`*`;`
: This command computes and prints the size of diagram (BDT, BDD or ZBDD) φ associated with the variable given as argument. Its complexity is in $\mathscr{O}(|\varphi|)$.

`compute number-of-products` *`variable Options`*`;`
: This command computes and prints the number of products encoded by the diagram (BDT, BDD, ZBDD) associated with the variable given as argument.

`compute size-of-sum-of-products` *`variable Options`*`;`
: This command computes and prints the size, i.e. the number of literals, of the sum of products encoded by the diagram (BDT, BDD or ZBDD) φ associated with the variable given as argument. Its complexity is in $\mathscr{O}(|\mathrm{SoP}(\varphi)|)$.

`compute lowest-order` *`variable Options`*`;`
: This command computes and prints the lowest order of the products of the sum of products encoded by the diagram (BDT, BDD or ZBDD) φ associated with the variable given as argument. Its complexity is in $\mathscr{O}(|\varphi|)$.

`compute highest-order` *`variable Options`*`;`
: This command computes and prints the highest order of the products of the sum of products encoded by the diagram (BDT, BDD or ZBDD) φ associated with the variable given as argument. Its complexity is in $\mathscr{O}(|\varphi|)$.

`compute orders` *`variable Options`*`;`
: This command computes and prints the numbers of products of each orders of the sum of products encoded by the diagram (BDT, BDD or ZBDD) φ associated with the variable given as argument. Its complexity is in $\mathscr{O}(|\mathrm{SoP}(\varphi)|)$.

`compute numbers-of-occurrences` *`variable Options`*`;`
: This command computes and prints the numbers of occurrences basic events (state variables) showing up in the sum of products encoded by the diagram (BDT, BDD or ZBDD) φ associated with the variable given as argument. Its complexity is in $\mathscr{O}(|\mathrm{SoP}(\varphi)|)$.

Options of these commands are given in Table 10.2.

Table 10.2: Options of the commmands to print statistics about sum of products.

Option	Type	Default value
`source-handle`	Identifier	`BDT`
`output`	String	`result.txt`
`mode`	`write/append`	`write`

10.3.3 Command `load`

The command `load` loads models and scripts.

`load model` *`file Options`*`;`

This command loads a model (or part of a model) from the given file.
Options are described Section 10.1.4 of this chapter.

`load script` *`file`*`;`

This command loads a script from the given file and executes it. It has no option.
It is described Section 10.1.5 of this chapter.

10.3.4 Command `print`

10.3.4.1 Commands to Print Models

`print model` *`Options`*`;`

This command prints the (previously loaded) source model into a file.
Options are described Section 10.1.4 of the present chapter.

`print target-model` *`Options`*`;`

This command prints the target model, i.e. the model instantiated and possibly pre-processed obtained from the source model, into a file.
Options are described Section 10.1.4 of the present chapter.

`print model-statistics` *`variable Options`*`;`

This command prints statistics about the structure function of the given variable in the target model.
Options of this command are `output` that sets the name of the output file and `mode` that sets the opening mode (`write` or `append`).

`print top-events` *`Options`*`;`

This command prints the top events of the target model.
Options are described Section 10.1.4 of the present chapter.

10.3.4.2 Commands to Print Environment Variables

`print option` *`environment-variable Options`*`;`

This command prints the value of the given environment variable.
Options are described Section 10.2.2 of the present chapter.

`print options` *`Options`*`;`

This command prints the values of all environment variables.
Options are described Section 10.2.2 of the present chapter.

`print randomizer-seed` *`Options`*`;`

This command prints the seed of the random number generator. It is described Section 15.2.2.1 (page 222).

10.3.4.3 Commands to Print BDT and BDD Indices

`print BDD-index` *`variable Options`*`;`
: that prints the BDD index of the given variable.

`print BDD-indices` *`variable Options`*`;`
: that prints the BDD indices of all basic events involved in the structure function of the given variable.

`print BDT-index` *`variable Options`*`;`
: that prints the BDT index of the given variable.

`print BDT-indices` *`variable Options`*`;`
: that prints the BDT indices of all basic events involved in the structure function of the given variable.

Options of these commands are `output` that sets the name of the output file and `mode` that sets the opening mode (`write` or `append`).

Indices are printed as `set` commands so that they can be reloaded.

10.3.4.4 Commands to Print Statistics on BDD Tables

`print BDD-cache-size` *`Options`*`;`
: BDD operations are cached, so to optimize the performance of algorithms. Results of operations are actually managed in a hashcache. This command prints the size of this hashcache.

`print BDD-hashtable-size` *`Options`*`;`
: BDD nodes labeled with (the BDD index of) a given variable are accessed via a hashtable. This command prints the maximum size of hashtables.

`print BDD-page-size` *`Options`*`;`
: To optimize the memory management, BDD nodes are allocated by pages. This command prints the number of BDD nodes of each page.

`print BDD-table-size` *`Options`*`;`
: This command prints the maximum number of BDD nodes XFTA can allocate.

`print BDD-statistics` *`Options`*`;`
: This command prints statistics on BDD tables.

`print BDD-number-of-nodes` *`Options`*`;`
: prints the total number of nodes stored in the unique table.

`print BDD-number-of-pages` *`Options`*`;`
: prints the number of pages in the unique table.

`print BDD-number-of-active-nodes` *`Options`*`;`
: prints the total number of active nodes stored in the unique table, i.e. the number of nodes that are used in at least one binary decision diagram associated with a variable.

The management of BDD tables is described Section 12.7 (page 189).

10.3.5 Command `reset`

The command `reset` resets models and calculations.

`reset model;`
: This command resets the source model, the target model and all calculations that have been

possibly performed. In a word, `reset model` puts back XFTA as at beginning of a new session.

`reset target-model;`
: This command resets the target model and all the calculations that have been performed from this model, in particular all data structures (BDT, BDD and ZBDD) that have been built.

`reset source-handle` *`variable handle`*`;`
: This command deletes the given data structure (BDT, BDD and ZBDD) associated with the given variable. This form of the command `reset` is useful when one wants to free computer memory before building another data structure.

10.3.6 Command `set`

The command `set` sets environment variables as well as a number of parameters of XFTA.

10.3.6.1 Commands to Set Environment Variables

`set option` *`environment-variable value`*`;`
: This command sets the value of the given environment variable. It is described Section 10.2.2 of the present chapter.

`set randomizer-seed` *`value`*`;`
: This command sets the seed of the random number generator. It is described Section 15.1.3 (page 219) and Section 15.2.2.1 (page 222).

10.3.6.2 Commands to Set BDT and BDD Indices

`set BDD-index` *`variable index`*`;`
: that sets the BDD index of the given variable.

`set BDT-index` *`variable index`*`;`
: that prints the BDT index of the given variable.

Indices must be positive integers between 1 and 16384.

10.3.6.3 Commands to Set Sizes of BDD Tables

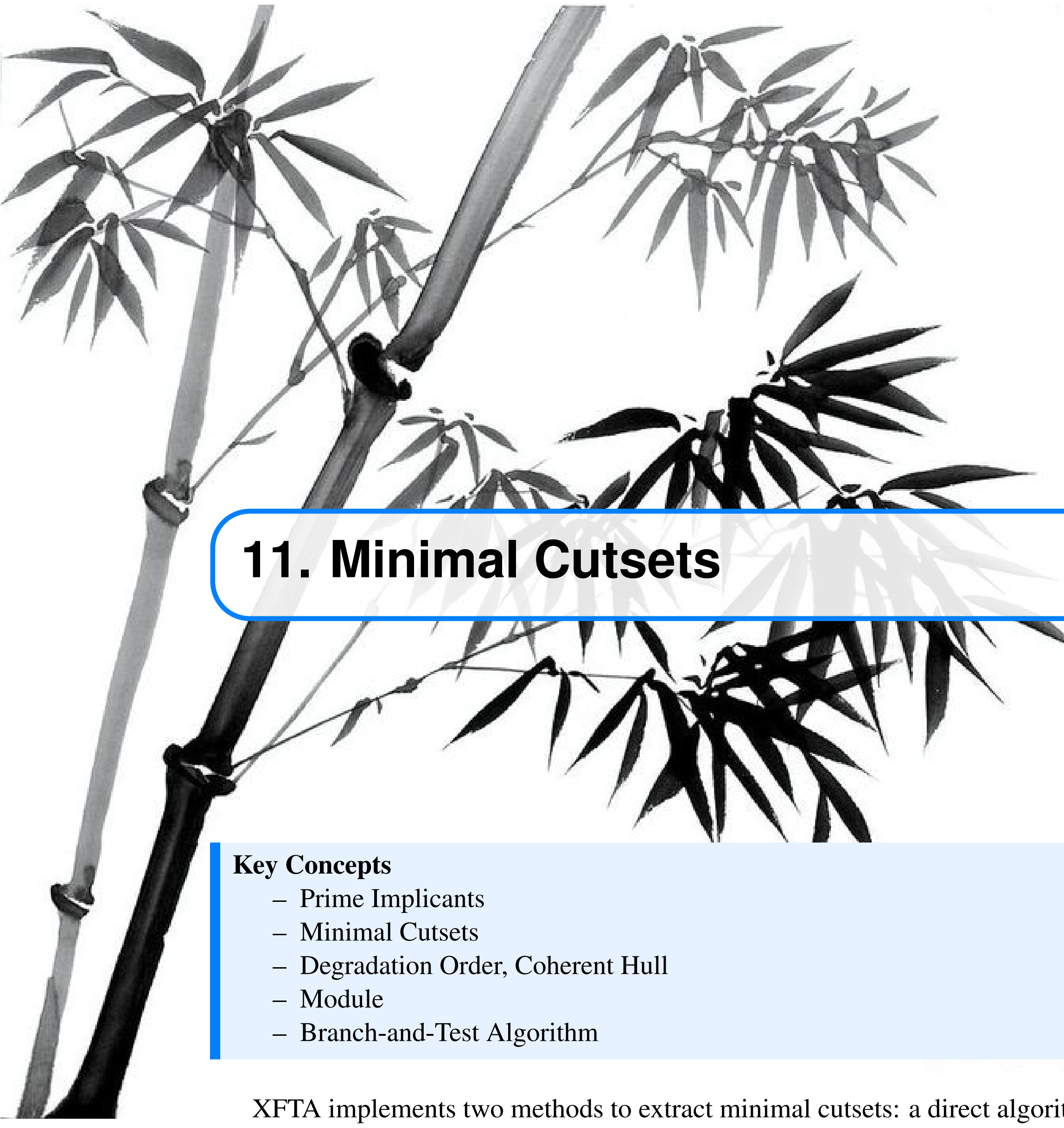
`set BDD-cache-size` *`size`*`;`
: BDD operations are cached, so to optimize the performance of algorithms. Results of operations are actually managed in a hashcache. This command sets the size of this hashcache.

`set BDD-hashtable-size` *`size`*`;`
: BDD nodes labeled with (the BDD index of) a given variable are accessed via a hashtable. This command sets the maximum size of hashtables.

`set BDD-page-size` *`size`*`;`
: To optimize the memory management, BDD nodes are allocated by pages. This command sets the number of BDD nodes of each page.

`set BDD-table-size` *`size`*`;`
: This command sets the maximum number of BDD nodes XFTA can allocate.

The management of BDD tables is described Section 12.7 (page 189).



11. Minimal Cutsets

Key Concepts

- Prime Implicants
- Minimal Cutsets
- Degradation Order, Coherent Hull
- Module
- Branch-and-Test Algorithm

XFTA implements two methods to extract minimal cutsets: a direct algorithm working from the system of stochastic Boolean equations and an indirect one working from the binary decision diagram encoding the structure function of the top event. The first algorithm encodes minimal cutsets by means of a binary decision tree, while the second one encodes them by means of a zero-suppressed binary decision diagram.

This chapter introduces first the mathematical foundations of the notion of minimal cutset, Section 11.1. Then, it presents XFTA direct algorithm Section 11.2. The indirect algorithm is presented Chapter 12, which is dedicated to the binary decision diagram approach. Finally, Section 11.4 presents the XFTA commands to implement the minimal cutsets approach, which relies on this direct algorithm. The reader not interested in theoretical aspects can thus skip the first two sections.

We shall not repeat here definitions of the notions of literal, product, minterm, sum-of-products and variable assignment. The reader is thus invited to refer to Section 9.2.2 for these definitions. Binary decision trees and their properties are presented Section 9.3.2.

11.1 Mathematical Framework

Intuitively, a minimal cutset represents a set minimum of failures of basic components that induces a failure of the system as a whole. This definition works fine for coherent models, but not for non-coherent ones.

Strangely enough, my 2001 article (Rauzy 2001) was the first one to give the correct mathematical definition of minimal cutsets, long after this notion was daily used.

Intuitively, a cutset is a set of basic events such that if these basic events are realized (set to true) and all other basic events are not (they are set to false), and the values of basic events are propagated up in the set of equations, then the top event is realized (it evaluates to true). A cutset is minimal if none of its proper subsets is a cutset. In this section, we shall formalize this intuition, starting with the notion of prime implicant.

11.1.1 Prime Implicants

Intuitively, minimal cutsets are minimal solutions of a model. In logic as well as in electronic circuit theory, the idea of minimal solution is captured via the notion of prime implicants.

Definition 11.1.1 — Prime Implicants. Let f be a Boolean formula built over a finite set of variables $\mathscr{V}$ and let π be a product built over $\mathscr{V}$. Then:

- π is an *implicant* of f, if $\pi \models f$, i.e. if any assignment σ of $\mathscr{V}$ that satisfies π satisfies f as well.
- π is a *prime implicant* of f if it is an implicant of f and none of its proper subset is.

The set of prime implicants of a formula f is denoted $\mathrm{PI}(f)$.

The above definition generalizes indeed to uniquely rooted, data-flow system of Boolean equations.

■ **Example 11.1** Let $f = A \cdot B + C$ and $g = A \cdot B + \overline{A} \cdot C$ built over $\{A, B, C\}$. Then,

$$\begin{aligned} \mathrm{PI}(f) &= \{A \cdot B, C\} \\ \mathrm{PI}(g) &= \{A \cdot B, \overline{A} \cdot C, B \cdot C\} \end{aligned}$$

■

As already discussed Chapter 7, negations, as the formula g in the above example, are, in reliability engineering, mostly used to exclude physically and operationally impossible configurations. This put aside, the more there are components failed, the more likely the system as a whole is failed. This is the reason why the notion of prime implicants, which produces minimal solutions with negations, is not fully satisfying in the reliability engineering context.

The notion of minimal cutsets that we shall introduce now differs from the one of prime implicants.

11.1.2 Formal Definition

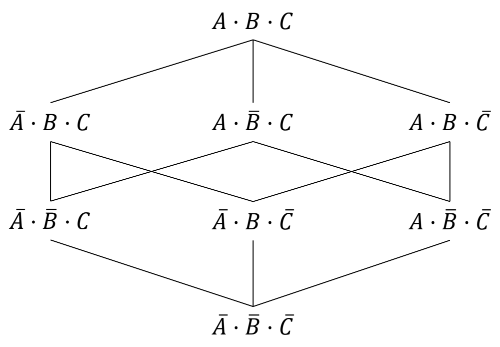
Minimal cutsets are sets of basic events, i.e. positive products. As already said, the intuition behind minimal cutsets is that if all basic events of the cutset are realized, then the system as a whole is failed. To put it more precisely, the system as a whole is failed in all basic events of the cutset are realized, even though none of the other basic events is. This precision is important because it makes it possible to go from a local state (the state of the basic events of the cutset) to a global one (the state of all basic events of the model).

Definition 11.1.2 — Least Minterm. Let π be a positive product built over a set of variables $\mathscr{V}$. The *least minterm* compatible with π, denoted by $\lfloor\pi\rfloor_{\mathscr{V}}$ or $\lfloor\pi\rfloor$ when $\mathscr{V}$ is clear from the context, is the minterm obtained by completing π with negative literals built over variables of $\mathscr{V}$ not showing up in π.

$$\lfloor\pi\rfloor_{\mathscr{V}} \overset{def}{=} \pi \cup \{\overline{V}; V \in \mathscr{V} \setminus \mathrm{var}(\pi)\}$$

We shall see in the next section why $\lfloor\pi\rfloor_{\mathscr{V}}$ is called the *least* minterm containing π.

We can now define formally minimal cutsets.



$A \cdot B \cdot C$
$\bar{A} \cdot B \cdot C$
$A \cdot \bar{B} \cdot C$
$A \cdot B \cdot \bar{C}$
$\bar{A} \cdot \bar{B} \cdot C$
$\bar{A} \cdot B \cdot \bar{C}$
$A \cdot \bar{B} \cdot \bar{C}$
$\bar{A} \cdot \bar{B} \cdot \bar{C}$

Figure 11.1: Hasse diagram for the lattice of minterms built over $\{A,B,C\}$.

Definition 11.1.3 — Minimal Cutsets. Let f be a Boolean formula built over a finite set of variables $\mathscr{V}$ and let π be a positive product built over $\mathscr{V}$. Then:

- π is a *cutset* of f, if $\lfloor\pi\rfloor_{\mathscr{V}} \models f$, i.e. π satisfies f.
- π is a *minimal cutset* of f if it is an cutset of f and none of its proper subset is.

The set of minimal cutsets of a formula f is denoted $\text{MCS}(f)$.

As for prime implicants, the above definition generalizes directly to uniquely rooted, data-flow system of Boolean equations.

■ **Example 11.2** Consider again the functions $f = A \cdot B + C$ and $g = A \cdot B + \overline{A} \cdot C$ of example 11.1. Then,

$$\begin{aligned} \text{MCS}(f) &= \{A \cdot B, C\} \\ \text{MCS}(g) &= \{A \cdot B, C\} \end{aligned}$$

■

To understand the relationship between prime implicants and minimal cutsets, we have to introduce the notion of degradation order.

11.1.3 Degradation Order

In Boolean risk assessment models, basic events represent failed states of components. This introduces a strong asymmetry between positive and negative literals:

- Positive literals represent the information of interest, namely what is failed, while negative literals represent what is not failed, or to put differently, components that are in their normal state.
- Moreover, as the systems under study are in general very safe, the probability p of the positive literal V is usually much smaller than the probability $1-p$ of its opposite $\overline{V}$.

Hence the idea of introducing a degradation order among minterms.

Definition 11.1.4 — Degradation Order. Let $\mathscr{V}$ be a finite set of Boolean variables and let π and ρ be two minterms built over $\mathscr{V}$. Then π is *less degraded* than ρ, which is denote $\pi \sqsubseteq \rho$, if each positive literal of π is a positive literal of ρ.

Minterms, equipped with the degradation order, form a lattice, which can be represented by means of a Hasse diagram (when the number of variables is small enough). Figure 11.1 shows such a diagram for minterms built over $\{A,B,C\}$. The least minterm (fully negative) is represented at the bottom, the biggest one at (fully positive) at the top.

The notion of least minterm gets now clear: least refers here to the degradation order. $\lfloor\pi\rfloor_{\mathscr{V}}$ is the smallest minterm built over $\mathscr{V}$ such that $\pi \subseteq \lfloor\pi\rfloor_{\mathscr{V}}$.

Using the degradation order, we can reformulate the condition for a model to be coherent:

Property 11.1 — Minterms of Coherent Models. Let f be a Boolean formula built over a finite set of variables $\mathcal{V}$. Then f is coherent if there is no two minterms π and ρ built over $\mathcal{V}$ such that:

- $\pi \sqsubset \rho$.
- $\pi \not\models f$ while $\rho \models f$.

■ **Example 11.3** Consider again the functions $f = A \cdot B + C$ and $g = A \cdot B + \overline{A} \cdot C$ of example 11.1. It is easy to verify that:

- f is coherent.
- g is not as the minterm $\overline{A} \cdot \overline{B} \cdot C$ satisfies g, the minterm $A \cdot \overline{B} \cdot C$ does not satisfy g, while $\overline{A} \cdot \overline{B} \cdot C \sqsubset A \cdot \overline{B} \cdot C$.

■

A non-coherent model can be made coherent be completing it with minterms that do not satisfy it, but that are greater than a minterm that satisfies it. This leads to the notion of coherent hull:

Definition 11.1.5 — Coherent Hull. Let f be a Boolean formula built over a finite set of variables $\mathcal{V}$. The *coherent hull* of f is the sum of minterms Φ built over $\mathcal{V}$ such that for each minterm π of Φ one of the two following conditions holds.

- π satisfies f.
- π does not satisfy f but there exists a minterm ρ built over $\mathcal{V}$ such that $\rho \sqsubset \pi$ and ρ satisfies f.

The coherent hull of a formula f is denoted $\llbracket f \rrbracket$.

■ **Example 11.4** Consider again the functions $f = A \cdot B + C$ and $g = A \cdot B + \overline{A} \cdot C$ of example 11.1. We have:

$$\begin{aligned} \llbracket f \rrbracket &= A \cdot B \cdot C + A \cdot B \cdot \overline{C} + A \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C \\ \llbracket g \rrbracket &= A \cdot B \cdot C + A \cdot B \cdot \overline{C} + \underline{A \cdot \overline{B} \cdot C} + \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C \end{aligned}$$

We underlined the (only) minterm that has been added to get $\llbracket g \rrbracket$. ■

The following property follows immediately from the definitions.

Property 11.2 — Coherent Hull of Formulas. Let f be a Boolean formula built over a finite set of variables $\mathcal{V}$. Then f is coherent if and only if $f \equiv \llbracket f \rrbracket$.

We can now state the central theorem of fault tree assessment.

Theorem 11.3 — Minimal Cutsets versus Prime Implicants. Let f be a Boolean formula built over a finite set of variables $\mathcal{V}$. Then,

$$\mathrm{MCS}(f) = \mathrm{PI}(\llbracket f \rrbracket)$$

In particular, if f is coherent then $\mathrm{MCS}(f) = \mathrm{PI}(f)$.

The formal proof of this theorem is given in Reference (Rauzy 2001). Intuitively, this proof is based on two remarks:

- Let π be a positive product such that $\lfloor \pi \rfloor$ satisfies f. Then, by definition, π is a cutset of f. It is also an implicant of $\llbracket f \rrbracket$ as, by construction, all minterms that contain π are in $\llbracket f \rrbracket$. It follows immediately that π is a minimal cutset of f if and only if it is a prime implicant of $\llbracket f \rrbracket$.

- Now, let π be a product containing at least one negative literal and let π^+ be the subset of its positive literals. By definition, π cannot be a cutset of f. It cannot be a prime implicant of $\lceil\!\lceil f \rceil\!\rceil$ neither, because if it was all the minterms containing it would satisfy $\lceil\!\lceil f \rceil\!\rceil$. In particular, the minterm $\lfloor \pi \rfloor = \lfloor \pi^+ \rfloor$. But in that case π^+ would be a cutset of f and therefore an implicant of $\lceil\!\lceil f \rceil\!\rceil$, which enters in contradiction with the fact that $\pi \supset \pi^+$ is a prime implicant of f.

Again, the above theorem generalizes directly to data-flow system of Boolean equations.

Historical remark:
I introduced the notion of degradation order, although under another name, in my 2001 article (Rauzy 2001). I generalized it in 2019 to models where variables take their values into finite domains (Rauzy and Yang 2019).

11.2 Algorithmic Framework

Designing efficient algorithms to extract minimal cutsets of a formula, or equivalently for a variable of a data-flow system of Boolean equations, is by no means easy. Although XFTA direct algorithm can be considered as a top-down method, it is only remotely inspired from the MOCUS method (Fussel and Vesely 1972). More concretely, it relies on techniques and data structures at work in SAT solvers implementing the Davis, Logeman and Loveland procedure (Davis, Logemann, and Loveland 1962), see e.g. (Kroening and Strichman 2017) for a general and recent introduction to algorithmic decision procedures.

The description of how all these mechanisms are implemented in XFTA goes beyond the scope of this guide. This section introduces however the key concepts to understand what is going on "under the hood". The reader interested in more details can refer to my 2012 article (Rauzy 2012).

11.2.1 Branch-and-Test Algorithm

The key principle of XFTA algorithm is to explore recursively all possible candidate cutsets. The execution of the algorithm can be seen as the traversal of a binary tree encoding a sum of disjoint products. The visit of each internal node of the tree consists first in testing whether the current product π must be kept or discarded and in the first case whether minimal cutsets can be extracted from π; Second, in *branching*, i.e. picking up a variable V not showing up in π and considering the two subcases where V is added to π and $\neg V$ is added to π. Extracted minimal cutsets are accumulated in a binary decision tree.

Products can be seen as partial assignments of variables. The value $\pi(f)$ of a formula f under a partial assignment π can be either 0, or 1 or undetermined. Here follows two definitions characterizing the status of a partial assignment in a system of Boolean equations.

Definition 11.2.1 — Contradictory and Closed Products. Let M be system of Boolean equations built over the set $\mathcal{V}$ be a set of Boolean variables and let π be a product (a partial assignment) built over $\mathcal{V}$. Then,

π is *contradictory* in M if there exists an equation $V = f$ of M such that either $\pi(V) = 1$ ($V \in \pi$) and $\pi(f) = 0$, or $\pi(V) = 0$ ($\neg V \in \pi$) and $\pi(f) = 1$.

π is *closed* in M if for all equations $V = f$ of M, either V does not show up in π and $\pi(f)$ is undetermined, or $\pi(V) = \pi(f)$.

Obviously, no cutset can be generated from contradictory partial assignments. On the contrary, closed partial assignments embed cutsets, as shown by the following property.

Property 11.4 — Closed Products and Cutsets. Let M be system of Boolean equations built over the set $\mathscr{V}$ be a set of Boolean variables, and let π be a product (a partial assignment) built over $\mathscr{V}$. Assume π is closed in M and that π contains (positively) the top event T of the model.

Then, the sub-product σ of π made of positive literals build over basic events of M showing up in π is a cutset of T.

Proof: By construction, the least minterm $\lfloor \sigma \rfloor$ must agree with π on all variables to which π gives a value. Moreover, π and thus $\lfloor \sigma \rfloor$ satisfy the top event T. Consequently, σ is a cutset of T.

Figure 11.2 gives the pseudo-code of the branch-and-test algorithm that builds the binary decision tree encoding the minimal cutsets of the top event of a system of stochastic Boolean equations.

```
1   BuildBinaryDecisionTree($M$, $T$, $p_{min}$):
2       // $M$: system of Boolean equations
3       // $T$: top event of $M$
4       // $p_{min}$: minimum probability of minimal cutsets
5       $\pi$ = $T$
6       BDT = new empty binary decision tree
7       BranchAndTest($M$, $T$, $\pi$, $p_{min}$, BDT)
8       return BDT
9
10  BranchAndTest($M$, $T$, $\pi$, $p_{min}$, BDT):
11      // $\pi$: product
12      if $\pi$ is contradictory in $M$:
13          return
14      if p($\pi$) $< p_{min}$:
15          return
16      if $\pi$ is closed in $M$:
17          $\sigma$ = product of positive literals built over basic events of $\pi$
18          TestAndInsertMinimalCutset($M$, $T$, $\sigma$, $p_{min}$, BDT)
19          return
20      Select a variable $V$ of $M$ not showing up in $\pi$
21      BranchAndTest($M$, $T$, $\pi \cdot V$, $p_{min}$, BDT)
22      BranchAndTest($M$, $T$, $\pi \cdot \overline{V}$, $p_{min}$, BDT)
```

Figure 11.2: XFTA branch-and-test algorithm to extract minimal cutsets

This code implements the traversal of a binary tree presented at the beginning of this section. In addition, it warranties that the extracted minimal cutsets have a probability greater than a given threshold p_{min} (line 18). Cutoffs will be discussed Section 11.2.3.

■ **Example 11.5** Consider the following set of Boolean equations.

$$\begin{array}{rclrcl} T & = & G_1 \vee G_2 & G_3 & = & C \vee D \vee E \\ G_1 & = & A \wedge G_3 \wedge G_4 & G_4 & = & F \wedge J \\ G_2 & = & B \wedge G_3 \wedge G_5 & G_5 & = & F \wedge K \end{array}$$

We want to extract the minimal cutsets of the top event T.

Step 1. We start with the partial assignment $T = 1$.

Step 2. As $T01$ is not closed, we select one a variable, e.g. G_1, and assign it first to 1. The current assignment is thus $T = 1, G_1 = 1$. Now the equation defining T is closed, but indeed not the one defining G_1.

Step 3. We select thus a new variable, e.g. A, and assign it first to 1. The current assignment is thus $T = 1, G_1 = 1, A = 1$.

Step 4. As equation defining G_1 is still not closed, we select another variable, e.g. G_3 and assign it to 1. The current assignment is thus $T = 1, G_1 = 1, A = 1, G_3 = 1$.

Step 5. Now, neither the equation defining G_1, nor the one defining G_3 is closed. We select thus a new variable, e.g. C, and assign it first to 1. The current assignment is thus $T = 1, G_1 = 1, A = 1, G_3 = 1, C = 1$.

Step 6. Now, the equation defining G_3 is closed, but still not the one defining G_1. We select thus a new variable, e.g. G_4, and assign it first to 1. The current assignment is thus $T = 1, G_1 = 1, A = 1, G_3 = 1, C = 1, G_4 = 1$.

Step 7. Now, the equation defining G_1 is eventually closed, but not the one defining G_4. We select thus a new variable, e.g. F, and assign it first to 1. The current assignment is thus $T = 1, G_1 = 1, A = 1, G_3 = 1, C = 1, G_4 = 1, F = 1$. This assignment is closed. We can thus test the candidate cutset $A \cdot C \cdot F$. This cutset is actually minimal. We can insert it in the BDT.

Step 8. Now, we come back on our last assignment, i.e. $F = 1$ and study the other value, i.e. $F = 0$. The current assignment is thus $T = 1, G_1 = 1, A = 1, G_3 = 1, C = 1, G_4 = 1, F = 0$. In this assignment, the equation defining G_4 is not closed. We select thus a new variable, e.g. J, and assign it first to 1. The current assignment is thus $T = 1, G_1 = 1, A = 1, G_3 = 1, C = 1, G_4 = 1, F = 0, J = 1$, which closed and gives another minimal cutset $A \cdot C \cdot J$.

Step 9. Backtring on the last assignment $J = 1$ and setting $J = 0$ leads to a contradictory assignment as $G_4 = 1$ but its definition is assigned to 0. We must thus backtrack again, til the assignment of G_4.

Step 10. And so on.

■

11.2.2 Extraction of Minimal Cutsets from Candidate Cutsets

The algorithm given in Figure 11.2 generates candidate cutsets. It remains to extract minimal cutsets out of these candidate cutsets. This is the role of the function `TestAndInsertMinimalCutset` (called line 18).

Figure 11.3 gives the pseudo-code of this function. This pseudo-code is a direct implementation of the definition of minimality of cutsets.

```
TestAndInsertMinimalCutset($M$, $T$, $\sigma$, $p_{min}$, BDT):
    isMinimalCutset = true
    forall variable $V$ in $\sigma$:
        $\rho$ = $\sigma \setminus \{V\}$
        if $\lfloor\rho\rfloor$ satisfies $T$:
            isMinimalCutset = false
            break
    if isMinimalCutset:
        InsertMinimalCutset(BDT, $\rho$, $p_{min}$)
```

Figure 11.3: Algorithm to test and insert a candidate minimal cutset

The function `InsertMinimalCutset` inserts the minimal cutset in the binary decision tree. It may modify the cutoff p_{min}, as we shall explain now.

11.2.3 Cutoffs

A *cutoff* is a condition minimal cutsets must verify not to be discarded. In algorithms given in Figures 11.2 and 11.3 the cutoff is only as a minimum probability for the extracted minimal cutsets.

In XFTA (and the minimal cutsets approach), cutoffs are actually defined by means of three thresholds:

- a maximum order (number of variables in the cutset),
- a minimum probability,
- the maximum number of minimal cutsets to be extracted.

The two first thresholds are local in this sense that they set a constraint on individual minimal cutset. The third one is global. The three thresholds are set by the analyst at the beginning of the calculation. However, while the maximum number of minimal cutsets to be extracted remains unchanged throughout the extraction, the maximum order and minimum probability of minimal cutsets are dynamically adjusted.

Recall that a binary decision tree maintains the list of the minimal cutsets it encodes sorted by decreasing order of probabilities (or increasing order of orders, if probabilities are not involved). When the number of extracted minimal cutsets reaches the predefined maximum number, the insertion of a new minimal cutset π is possible only if the probability of π is higher than the probability of the minimal cutset ρ of lowest probability already encoded in the binary decision tree. In this case, π is actually inserted, while ρ is removed. The cutoff can thus been dynamically adjusted so to that into account that from this point, a forthcoming minimal cutset needs to have a probability higher than the one of ρ to be inserted.

Large probabilistic safety assessment models of the nuclear industry involve thousands of basic events and even more gates. No one knows how many minimal cutsets these models have. The numbers of minimal cutsets are for sure astronomical. In a Japanese model I could study in details (Epstein and Rauzy 2005), which was relatively small (for a nuclear power plant model), one of the described accident sequences had more than 10^{40} minimal cutsets! It was possible to encode them in a relatively compact way, thanks to the binary decision diagram technology, but indeed not to inspect them one by one.

However, the key issue did not stand in the number of minimal cutsets, but in their individual probabilities. Fortunately, nuclear power plants do not experience an accident every morning (even though such accidents happen). This translates into models in general and translated into that model in particular. The calculated probability of the accident sequence under scrutiny was less than 10^{-6} (per year). In other words, the cumulated probability of 10^{40} minimal cutsets was less than 10^{-6}. This simply meant that the probabilities of most of these cutsets were extremely low. Lower, for instance, than the probability that a cosmic ray changed a bit in the memory of the computer on which calculations have been done.

The above discussion explains why cutoffs are used for the extraction of minimal cutsets. By discarding minimal cutsets whose probabilities are lower than a given threshold, not only one alleviates dramatically the computational cost (we shall come back to that Section 11.3), but also one gets rid of meaningless information. Factually, direct minimal cutsets extraction algorithms are first and foremost *model pruners*: their primary goal is to extract the relevant information from the model. And they do. Experiments on American nuclear probabilistic safety assessment models showed than only a tiny fraction (sometimes less than 5%) of the basic events of these models showed up in the extracted minimal cutsets (Epstein, Rauzy, and Wakefield 2006).

This raises indeed a methodological question: why developing models at significant cost, if it is to discard large parts of them when performing calculations?

Moreover, this leads to what I called the *refinement paradox* in my 2018 article (Rauzy 2018).

As an illustration, assume we decide to set up the cutoff at 1.00×10^{-10}, i.e. to discard minimal cutsets whose probabilities are lower than this threshold. Assume moreover we have a first version M_1 of our model and that in this version we have a minimal cutset $E.\pi$ whose probability is 2.00×10^{-10}, the probability of basic event E being 2.00×10^{-3}. Assume finally that we decide, in order to be more accurate, to refine our model which leads us to design a new version M_2 in which the basic event E is decomposed into four basic events E_1, E_2, E_3 and E_4 whose probabilities are equal to 6.00×10^{-4}. Note that by doing so, we are slightly more pessimistic that in the first version as the probability of the disjunction of the E_i's (namely 2.40×10^{-3}) is slightly higher than the estimated probability of E (namely 2.00×10^{-3}). In M_2, the minimal cutset $E.\pi$ gives rise to four minimal cutsets $E_1.\pi, \dots E_4.\pi$ whose probabilities are all 6.00×10^{-11}, i.e. below the cutoff threshold. This means that these minimal cutsets are discarded.

Here stands the paradox: by refining the model, we decreased the probability of the top event, even though our refinement is supposedly conservative. At the limit, we could make the risk vanish just by refining the model enough.

11.2.4 Preprocessing

Another key ingredient for the efficiency of minimal cutsets extraction algorithms stands in the preprocessing of models. Preprocessing rules have two main purposes: detecting modules and rewriting the model so to make basic operations of the core algorithm more efficient and to help heuristics.

11.2.4.1 Detecting Modules

Intuitively, a module is a part of a model which is independent from the rest of the model. The variables that show up in the module do not show up elsewhere in the model. Formally, modules are defined as follows.

Definition 11.2.2 — Module. Let S be a uniquely rooted data-flow system of Boolean equations. Let G be a flow variable of this system. Then G is a *module* of S, if for any other flow variable H of S one of the three following conditions hold.

- H is a descendant of G ($H \in \text{var}(G)$).
- H is an ancestor of G ($G \in \text{var}(H)$).
- G and H share no descendant ($\text{var}(H) \cap \text{var}(G) = \emptyset$).

Modules can be seen as macro basic events, and they can be considered as such during the extraction of minimal cutsets.

■ **Example 11.6** Consider again the model of Example 11.5. In this model, G_3 is a module as variables C, D and E occur only under G_3. During the extraction of minimal cutsets of T, we can thus consider G_3 as a macro basic event, i.e.

- Extract its minimal cutsets separately, here we get the three singletons C, D and E.
- Extract the minimal cutsets of T as if G_3 was a basic event.
- Expand eventually the minimal cutsets containing G_3 by means of its own minimal cutsets.

■

In practice, the modularization of models alleviates often considerably the cost of extraction of minimal cutsets. Modules can be detected in linear time with respect to the size of the model (Dutuit and Rauzy 1996), which makes this preprocessing very interesting.

⚠ **Modules**
XFTA detects modules before extracting minimal cutsets and performs extraction on the modularized model. However, in the current version, once minimal cutsets are extracted, they are demodularized so to be able to perform seamlessly all probabilistic calculations. It is thus not a good idea to "count" on the modularization process to reduce the number of extracted minimal cutsets.

11.2.4.2 Rewritings

It is often the case that, by rewriting the model under study into an equivalent one, one can simplify dramatically the extraction of minimal cutsets.

■ **Example 11.7** Consider again the model of Example 11.5 and assume that G_3 is not a module.

One thing we can do is to factorize G_3 in G_1 and G_2. We get thus new equations (that replace those defining G_1 and G_2:

$$\begin{array}{rclcrcl} T & = & G_3 \wedge G_6 & & G_6 & = & G_7 \vee G_8 \\ G_7 & = & A \wedge G_4 & & G_8 & = & B \wedge G_5 \end{array}$$

In this way, G_3 is expanded only once. ■

Rewritings can also help the heuristics that selects the next variable to consider.

■ **Example 11.8** Consider again the model of Example 11.5. Assume that the basic events F and J have low probabilities so that both $p(A) \times p(F)$ and $p(A) \times p(J)$ are lower than the minimal probability.

Then, rewriting the definition of G_1 as $G_1 = A \wedge G_4 \wedge G_3$, may help the variable selection heuristic to select G_4 first (rather than G_3). Do so avoids the exploration of the different ways to satisty G_3 which is useless since there is no way to satisfy A and G_4. ■

The above examples show that rewritings rely heavily on rules of thumb. Moreover, to be interesting, they should be of low computational cost, otherwise if would be moving from the frying pan into the fire. For this reason, XFTA implements a set of relatively simple rewritings.

11.3 Computational Complexity Issues

Computational complexity theory is the branch of theoretical computer science interested in the computation cost of solving problems. By problem, we mean here . Calculating the probability of the top event of a fault tree and extracting the minimal cutsets of a fault tree are examples of such problems.

In this section, we shall recall some fundamental computational complexity results about problems issued from reliability engineering. The following presentation is very sketchy as a whole book or at least several chapters would be necessary to present the topic in depth.

The reader interested in a thorough treatment should refer to reference textbooks and articles, e.g. (Arora and Barak 2009; Garey and Johnson 1979; Papadimitriou 1994; Valiant 1979) and, for complements, to my articles (Rauzy 2001; Rauzy 2018).

11.3.1 Testing Satisfiability and Related Problems

The simplest problem we can look at is to determine, given a data-flow system of Boolean equations S built over a set of variables $\mathscr{V} = \text{var}(S)$ and with unique root R, whether there exists an assignment of the variables of $\mathscr{V}$ that satisfies R. If the model S is coherent, the problem is trivial: it suffices to assign the value 1 to all variables and we get a satisfying assignment of R. If the model S is non-coherent, then the problem is equivalent to SAT, i.e. is NP-complete.

Now consider the problem of determining whether a product π built over $\mathscr{V}$ is a prime implicant of R. Again, we must distinguish two cases:

– If S is coherent, then we have to check first π is a cutset of R, i.e. that $\lfloor \pi \rfloor$ satisfies f which can be done in $\mathscr{O}(|S|)$, where $|S|$ denotes the size of S; then that no proper subset of π is a cutset of S, which can be done in $\mathscr{O}(|\pi| \times |S|)$. The problem is easy.
– S is non-coherent, then we have to solve a series $(|\pi| + 1)$ of SAT instances: first, we have to test first whether π is an implicant of R, i.e. to verify that $\pi \wedge \neg R$ is not satisfiable; then to

verify that no proper subset of π is an implicants R. Strictly speaking this problem is thus coNP-complete.

As for satisfiability testing, there is thus a huge difference between coherent and non-coherent models regarding the problem of testing whether a product π is a prime implicant of R or not. Note that testing whether a positive product π is a minimal cutset of R or not has the same complexity, i.e. in $\mathscr{O}(|\pi| \times |S|)$, whether the model is coherent or not (as the procedure described above applies in both cases).

11.3.2 Counting and Reliability Problems

We can turn now to the problem of counting the number of satisfying assignments of R. This problem has been shown $\sharp P$-complete by Valiant in his second 1979 article (Valiant 1979). Problems in the class $\sharp P$ are strongly believed to be intractable as, by Toda's theorem (Toda 1991), a Turing machine with a $\sharp P$ oracle captures the whole polynomial hierarchy.

One of very surprising results obtained by Valiant is that MONOTONE-SAT is also $\sharp P$-complete, i.e. counting the number of satisfying assignments of a coherent model is as difficult as counting the number of satisfying assignments of a non-coherent one.

Assume we would have an efficient procedure to calculate the exact probability of R (given the probabilities of basic events of $\mathscr{V}$. Then, we could apply this procedure for the case where all basic events have the probability $1/2$. By multiplying the result by 2^n, where n is the number of basic events of $\mathscr{V}$, we would have thus an efficient procedure to count the number of satisfying assignments of R. This simple reasoning, done by Valiant, shows that calculating the exact probability of R is a $\sharp P$-hard problem, whether the model is coherent or not.

This result is quite disappointing: it shows that even the most basic problem encountered in reliability engineering is intractable, at least if one is interested only in exact results (or warrantied approximations).

11.3.3 Extracting Prime Implicants and Minimal Cutsets

So far, we looked at problems whose solutions have small sizes (a floating point number in the worst case). Things are indeed completely different when we look at the problems of extracting prime implicants and minimal cutsets. Chandra and Makowski showed that a Boolean formula can have up to $\mathscr{O}(n \times 3^n)$ prime implicants, where n denotes the number of variables of this formula (Chandra and Markowsky 1978). The former number reduces to $\mathscr{O}(\sqrt{n} \times 2^n)$ in case the formula is monotone (coherent), but is still very, very large. In practice, storing all of the minimal cutsets of a model (not to speak about its prime implicants) proves to be unfeasible, even if compact data structures such as Minato's zero-suppressed binary decision diagrams (Minato 1993) are used.

Now assume that we are only interested in prime implicants and minimal cutsets of order k or less. The number prime implicants of order k or less is at most $\sum_{i=1}^{k} 2^k \binom{n}{k}$, while the number of minimal cutsets of order k or less is at most $\sum_{i=1}^{k} \binom{n}{k}$. In both cases, it is thus in $\mathscr{O}(n^k)$, assuming k constant and $k \ll n$.

At this point we are back to our discussion about the problem of checking whether a product π is a prime implicant or a minimal cutset of R.

Extracting all prime implicants of order k or less is at least as difficult as checking that a particular product π (of order k or less) is a prime implicant of R. This means that this problem is coNP-hard, which means intractable for practical purposes.

The situation is very different with minimal cutsets: As checking whether a positive product π is a minimal cutset of R can be done in $\mathscr{O}(|S|)$ and the number of minimal cutsets of order k or less is in $\mathscr{O}(n^k)$, it follows that extracting all minimal cutsets of order k or less is in $\mathscr{O}(|S| \times n^k)$, i.e. of polynomial worst case complexity, the degree of the polynomial being k.

Now we can remark that extracting minimal cutsets whose probability is greater than a given value is no more difficult as extracting minimal cutsets of order k or less. Probabilities act here as weights of variables, the weight of a positive product being the sum of the weights of its variables. To see that it suffices to pose $w(E) = -\log(p(E))$ for each basic event E ($w(E) \geq 1$ if $p(E) > 0.37$, which is a reasonable assumption). We have then $w(\pi) = \sum_{E\in\pi} w(E)$. In other words, for a given threshold probability p_{min}, the number of minimal cutsets whose probability is greater than p_{min} is in $\mathscr{O}\left(n^{-\log(p_{min})}\right)$.

This explains why both the algorithm using approximate binary decision diagrams (Rauzy 2001) and the XFTA algorithm are of polynomial worst case complexity.

The calculation of risk indicators are of linear complexity with respect to the size of the sum of products (or binary decision diagram) they are calculated from. It follows that assessing systems of Boolean equations via minimal cutsets calculation can be done in polynomial time, provided we accept approximated results. Indeed, these results cannot be warrantied. There exists always the risk that a huge number of minimal cutsets of tiny probabilities passes under the radar and that taken together these minimal cutsets make the probability of the top event. The analyst should be aware of this problem, which brings us back to the refinement paradox discussed Section 11.2.3.

This discussion concludes the first part of this chapter, dedicated to the mathematical and algorithmic frameworks of minimal cutsets extraction. The remainder of the chapter is dedicated to XFTA commands to handle minimal cutsets.

11.4 Commands to Handle Minimal Cutsets

This section presents XFTA commands to handle minimal cutsets.

11.4.1 Extracting Minimal Cutsets

11.4.1.1 Command `build BDT`

The base command to extract minimal cutsets of the variable `Top` and store them into a binary decision tree is as follows.

```
build BDT Top;
```

In the above example, the variable is called `Top`. In general, the extraction of minimal cutsets is performed for the root variable (a top event) of the model. However, the extraction of minimal cutsets can be performed for any variable of the model.

It happens sometimes that, after preprocessing, the definition of the variable reduces to a constant (either true or false). In such as case, the command `build BDT` returns an error and the execution of the script stops.

The extraction of minimal cutsets is always performed for a given cutoff. In the above command, the cutoff is implicitly chosen via the default values of the corresponding options. We shall come back to that in a moment.

The above command performs thus the following actions:

1. It preprocesses the model for the given variable.
2. It extracts minimal cutsets for the given variable and cutoffs, and stores them into a binary decision tree.

The binary decision tree in which minimal cutsets are stored is accessible via a *handle*, see Section 9.3.5.

The command `build BDT` has thus several options making it possible to define the cutoff and the name of the handle. Table 11.1 summarizes these options, and gives their types and default values.

Table 11.1: Options of the command `build BDT`

Option	Type	Default value
`maximum-number`	Positive integer	1,000,000
`maximum-order`	Positive integer	100
`minimum-probability`	Real in $[0,1]$	0.0
`mission-time`	Positive real	0.0
`target-handle`	Identifier	`BDT`
`variable-ordering-heuristic`	Identifier	`DFLM`

11.4.1.2 Cutoffs

The cutoff with which the extraction of minimal cutsets is performed is specified by means of the following options.

- The maximum order of the extracted minimal cutsets, set via the option `maximum-order`;
- The minimal probability of the extracted cutsets probability, set via the option `minimum-probability`.
- The mission time at which the probability is calculated, set via the option `mission-time`;
- The maximum number of extracted cutsets, set via the option `maximum-number`;

The value of these options can be given:

- Either directly with the command, e.g.

```
build BDT Top minimum-probability=1.0e-6 mission-time=8760;
```

- Or by modifying their default values via the command `set option` (see Section 10.3.6), e.g.

```
set option minimum-probability 1.0e-6;
set option mission-time 8760;
build BDT Top;
```

In order to avoid memory allocation problems, there is a limit to the number of minimal cutsets XFTA extracts. The extraction is stopped when this limit is reached (no matter the probabilities of these minimal cutsets or the number of minimal cutsets that remain potentially to extract). This limit is set via the option `maximum-number`. Its default value is rather low (1,000,000) but can be safely increased on most of the computers. Recall however that the memory occupation of a sum of minimal cutsets is roughly proportional to the number of occurrences of variables in the sum.

11.4.1.3 Handles

Each variable manages a list of handles. This list is empty in most of the cases. Two different events can be associated with handles with the same name. However, all handles associated with a variable must have different names.

By default, the handle for the binary decision tree encoding minimal cutsets is named `BDT`.

Commands to print information on sums of minimal cutsets and commands to calculate risk indicators are all taking the name of the handle from which these operations should be performed as an option, e.g.

```
print minimal-cutsets Top source-handle=MCS2 output="mcs.tsv";
compute probability Top source-handle=MCS2 output="prb.tsv";
```

By default, they are thus performed on the handle named `MCS` (unless the default handle name is changed).

If the command `build BDT` is called with a handle that already exists, the previously extracted sum of minimal cutsets is deleted, i.e. the new sum of minimal cutsets replaces the old one.

As already pointed out, binary decision tree encoding minimal cutsets may occupy a lot of computer memory. It may be thus useful to cleaned up those that are no longer used before starting the building of a new one. This is done by means of the command `reset source-handle` (see Section 10.3.5), e.g.

```
reset source-handle Top MCS2;
```

11.4.1.4 Variable Ordering Heuristics

Building a binary decision tree requires defining a total order over the variables showing up in the tree. This is achieved by assigning an index to each variable and sorting variables according to their indices (two variables cannot have the same index).

Although much less dramatic than in the case of binary decision diagrams, the chosen variable order may have a small influence on the size of the binary decision tree. As there is no way to predict what is the best ordering, nor even a good one, heuristics are used. Heuristics to order variables in binary decision trees are the same than those used for binary decision diagrams. We report thus the reader to Section 12.2 for a discussion and a presentation of heuristics available in the current version of XFTA.

The variable ordering heuristic can be chosen via the option `variable-ordering-heuristic` or the corresponding environment variable. Its default value is `DLFM`.

```
build BDT Top variable-ordering-heuristic=SCG-OAF;
```

11.4.2 Printing Minimal Cutsets

Once extracted, minimal cutsets can be printed out into a text file at the TSV format. The command `print minimal-cutsets` is used to do so. E.g.

```
print minimal-cutsets Top source-handle=MCS2 mission-time=8760
    output="mcs8760.txt";
```

Minimal cutsets are printed sorted in decreasing order of their probabilities (and by order in case no probability is computable). Basic events inside minimal cutsets are separated by tabulations. They are printed out sorted accorded to their index. Consequently, the order is consistent from one minimal cutset to the next one. However, indices are calculated internally and there is no way (in the current version at least) to influence them. To sort basic events inside minimal cutsets, an external tool such as a spreadsheet tool should be used.

The following indicators can be printed out in front of each minimal cutset (separated with tabulations).

- The *rank* of the minimal cutset, i.e. its index in the sum of minimal cutsets sorted in decreasing order of their probabilities;
- The *order* of the minimal cutset, i.e. the number of basic events that show up in the minimal cutset;
- The *probability* of the minimal cutset calculated at a given mission time;
- The so-called *contribution* of the minimal cutset, i.e. the probability of the minimal cutset divided by the sum of the probabilities of the minimal cutsets.
- The *unconditional failure intensity* of the minimal cutset, which is defined as follows.

$$\omega_\pi \overset{def}{=} \sum_{E\in\pi} \omega_E \times \mathrm{MIF}_{\pi,E} \tag{11.1}$$

The mathematical framework of failure intensity is discussed Chapter 16. We shall thus give here only the above definition.

Options of the command `print minimal-cutsets` are given in Table 11.2.

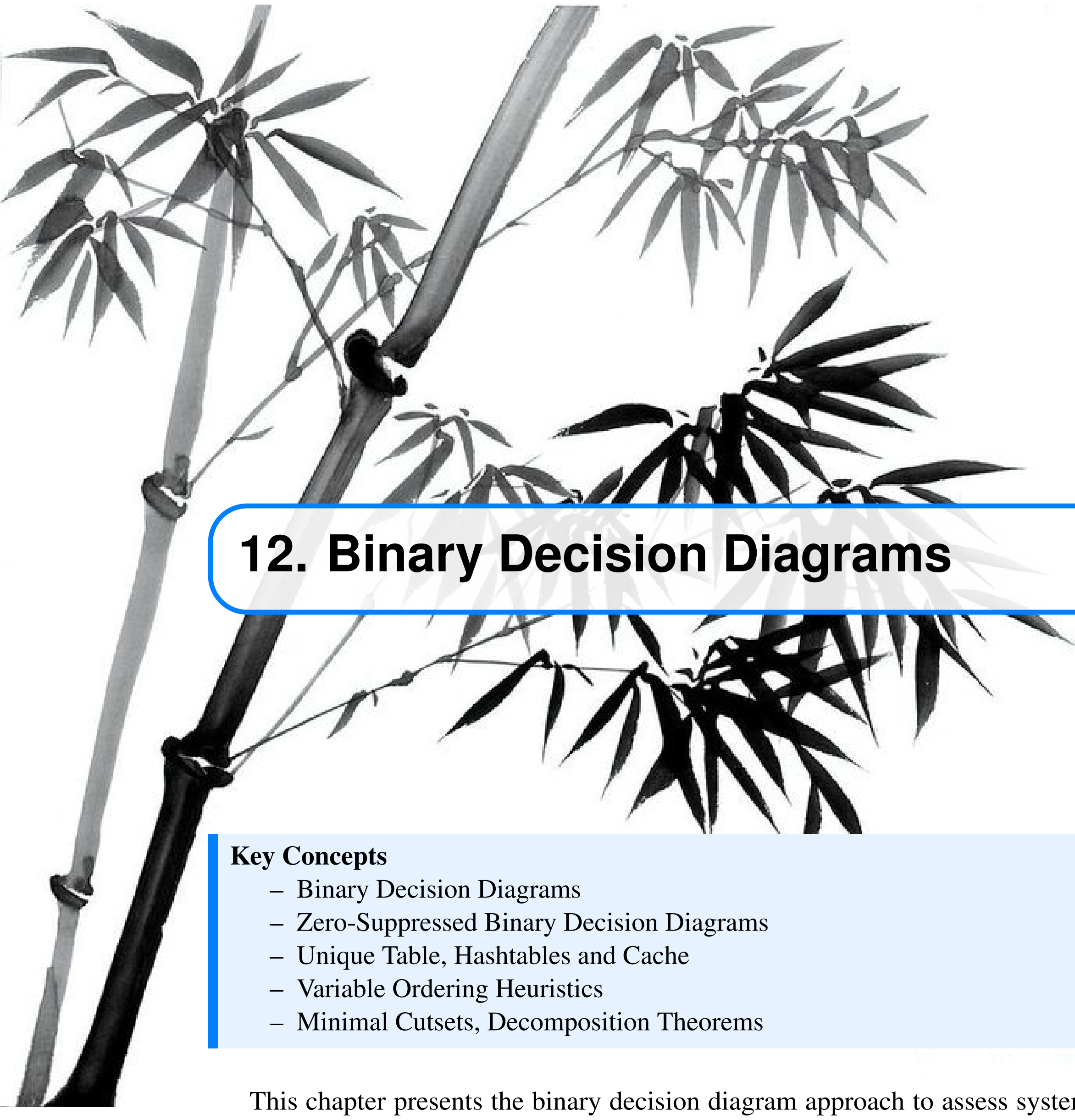
Table 11.2: Options of the command `print minimal-cutsets`

Option	Type	Default value
`top-event`	Identifier	None[1]
`source-handle`	Identifier	`BDT`
`mission-time`	Positive real	0.0
`print-minimal-cutset-rank`	Boolean	`true`
`print-minimal-cutset-order`	Boolean	`false`
`print-minimal-cutset-probability`	Boolean	`true`
`print-minimal-cutset-contribution`	Boolean	`true`
`print-minimal-cutset-failure-intensity`	Boolean	`false`
`output`	String	`result.txt`
`mode`	`write/append`	`write`

11.4.3 Commands to Compute Statistics about Minimal Cutsets

The command `compute` makes it possible to print out various statistics about minimal cutsets, see Section 10.3.2.2.

[1]Only for XML grammar, as this is an argument of the command



12. Binary Decision Diagrams

Key Concepts
- Binary Decision Diagrams
- Zero-Suppressed Binary Decision Diagrams
- Unique Table, Hashtables and Cache
- Variable Ordering Heuristics
- Minimal Cutsets, Decomposition Theorems

This chapter presents the binary decision diagram approach to assess systems of stochastic Boolean equations. Binary decision diagrams have been already presented Chapter 9. The reader is thus invited to check Chapter 9 before diving into this one. The objective of this chapter is not to present the binary decision diagram technology in full details, but rather to emphasize its important features. The reader interested in an in-depth presentation can refer to Bryant's article (Brace, Rudell, and Bryant 1990; Bryant 1992) as well as my articles dealing on the use of this technology in the context of reliability engineering (Rauzy 2001; Rauzy 2008a).

12.1 Creation of Binary Decision Diagrams

12.1.1 Bottom-Up Construction

In Chapter 9, we presented binary decision diagrams as a data structure to encode Boolean functions (which they are) without explaining how they are obtained. Their construction plays however a key role in the efficiency of the technology.

Binary decision diagrams are built bottom-up. To build the binary decision diagram of a formula $f = g \vee h$, one builds first the binary decision diagrams encoding g and h, then one composes these two binary decision diagrams to get the one encoding h. This applies indeed to all logical connectives.

Combining binary decision diagrams is possible thanks to the orthogonality of the Shannon decomposition with the usual logical connectives.

Property 12.1 — Orthogonality of the Shannon Decomposition with Usual Logical Connectives. Let $f = E \cdot f_1 + \overline{E} \cdot f_0$ and $g = E \cdot g_1 + \overline{E} \cdot g_0$ be two Boolean formulas, where E is a Boolean variable and f_1, f_0, g_1 and g_0 are Boolean formulas in which E does not show up. Then, the following equivalences hold.

$$\begin{aligned} f+g &\equiv E\cdot(f_1+g_1)+\overline{E}\cdot(f_0+g_0) \\ f\cdot g &\equiv E\cdot(f_1\cdot g_1)+\overline{E}\cdot(f_0\cdot g_0) \\ \overline{f} &\equiv E\cdot\overline{f_1}+\overline{E}\cdot\overline{f_0} \end{aligned}$$

The combination of binary decision diagrams can thus be performed top-down, from their root nodes to their leaves. The pseudo-code of the algorithm building the disjunction of two binary decision diagrams is given Figure 12.1.

```
1  BuildOr(F, G):
2     if F==1 or G==1:
3        return 1
4     if F==0:
5        return G
6     if G==0:
7        return F
8     R = LoopUpCache(or, F, G)
9     if R!=null:
10       return R
11    if F.index<G.index:
12       R1 = BuildOr(F.thenSon, G)
13       R0 = BuildOr(F.elseSon, G)
14       R = NewBDDNode(F.index, R1, R0)
15    else if F.index>G.index:
16       R1 = BuildOr(F, G.thenSon)
17       R0 = BuildOr(F, G.elseSon)
18       R = NewBDDNode(G.index, R1, R0)
19    else:
20       R1 = BuildOr(F.thenSon, G.thenSon)
21       R0 = BuildOr(F.elseSon, G.elseSon)
22       R = NewBDDNode(F.index, R1, R0)
23    InsertInCache(or, F, G, R)
24    return R
```

Figure 12.1: Pseudo-code of the function that builds the disjunction of two BDD

This pseudo-code is directly derived from Property 12.1. It manages in addition the terminal cases and the cases where the two binary decision diagrams are not rooted with the same variable. It introduces also a caching mechanism, which is the topic of the next section.

12.1.2 Caching

The combination principle we sketched in the previous section would be of exponential worst case complexity and practically infeasible without caching.

The principle of *caching*, also called *tabulation* and *memoization*, consists in storing results of operations into a table called a *cache*. When an operation must be performed, the cache is looked up. If the operation has been already performed and its result is stored in the cache, then the result is returned (line 8 of the pseudo-code given in Figure 12.1). Otherwise, the operation is performed and its results is inserted in the cache (line 23 of the pseudo-code given in Figure 12.1).

Caching makes the logical combination of binary decision diagrams of polynomial worst case complexity (Brace, Rudell, and Bryant 1990), at least in theory. In practice, it is impossible to store the results of all operations. This would consume much too much computer memory. The cache is thus a table of fixed size, managed as a hashtable. The larger the cache, the more efficient the caching.

In XFTA, it is possible to tune sizes of binary decision diagram tables, so to achieve better performance, see Section 12.7.

12.1.3 k-out-of-n and Related Connectives

So far, we considered classical connectives $\vee$, $\wedge$ and $\neg$, which are unary or binary (or associative). Systems of stochastic Boolean equations involve also non-traditional connectives such as k-out-of-n and related connectives.

In my article (Dutuit and Rauzy 2001), I showed that these connectives can be handled efficiently thanks to dynamic programming techniques (which are nothing but a specific implementation of caching).

They key idea is to use the following property.

Property 12.2 — Decomposition of k-out-of-n. Let f_1, f_2,... f_n be n Boolean formula and k be an integer. Then, the following equivalence holds.

$$\texttt{atleast}\; k\,(f_1,\ldots,f_n) \equiv \begin{cases} 1 & \text{if } k \leq 0 \\ 0 & \text{if } k > n \\ f_1 \wedge r_1 \vee r_0 & \text{otherwise} \end{cases}$$

where,

$$\begin{aligned} r_1 &= \texttt{atleast}\; k-1\,(f_2,\ldots,f_n) \\ r_0 &= \texttt{atleast}\; k\,(f_2,\ldots,f_n) \end{aligned}$$

It is easy to modify this property to handle `atmost` and `cardinality` connectives.

12.2 Variable Ordering Heuristics

12.2.1 Importance of the Variable Ordering

The size of the binary decision diagram encoding a function may depend dramatically on the chosen variable ordering.

■ **Example 12.1** Figure 12.2 shows the binary decision diagrams encoding the function $(\texttt{A1} \wedge \texttt{B1}) \vee (\texttt{A2} \wedge \texttt{B2}) \vee (\texttt{A3} \wedge \texttt{B3})$ for the variable ordering $\texttt{A1} < \texttt{A2} < \texttt{A3} < \texttt{B1} < \texttt{B2} < \texttt{B3}$ and $\texttt{A1} < \texttt{B1} < \texttt{A2} < \texttt{B2} < \texttt{A3} < \texttt{B3}$. ■

We can generalize example 12.1 to n layers, i.e. consider the binary decision diagrams encoding a function:

$$\varphi_n \stackrel{def}{=} \bigvee_{i=1}^{n} A_i \wedge B_i \tag{12.1}$$

for the orderings $A_1 < \ldots < A_n < B_1 < \ldots < B_n$ and $A_1 < B_1 \ldots < A_n < B_n$.

In the first case, the size of the binary decision diagrams is exponential in n because before making a decision on the value of the function, one gives a value to all the A_i's and each assignment of the A_i's leads to a different function. Consequently the upper part of the binary decision diagram is complete binary tree, which contains thus $2^n - 1$ nodes.

In the second case, the size of the binary decision diagram is simply $2n+2$.

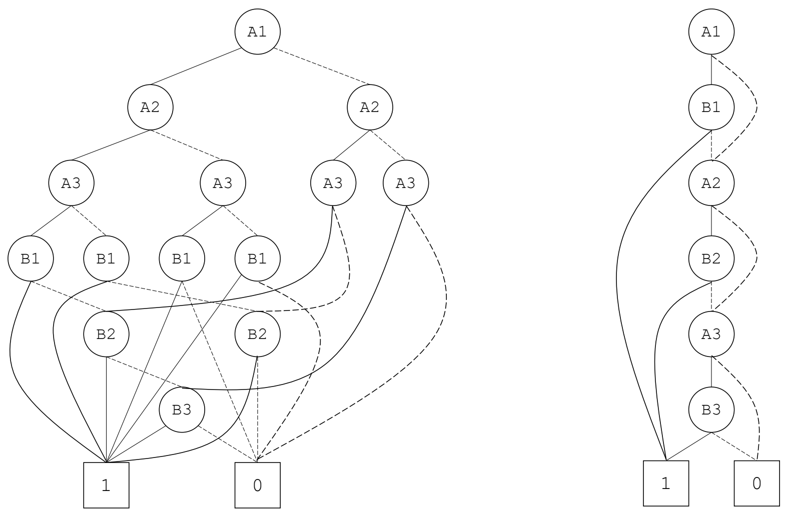


Figure 12.2: Illustration of the importance of variable ordering

It is thus of primary importance to choose the best possible variable ordering. Unfortunately, there is no mean to know in advance whether an ordering is good or not, not to speak about finding the best one. The only way to know whether an ordering is good or not, is to build the binary decision diagram and to look at its size.

Weneger showed that the problem of finding the best variable ordering is NP-complete (Wegener 2000). This result assumes however that at least one binary decision diagram encoding the function has been built. But this diagram can be of exponential size with respect to the number of the variables, or equivalently the size of the formula (or system of Boolean equations).

12.2.2 Literature Review

The impossibility to know *a priori* whether a variable ordering will be good or not leads to the design of heuristics. There exists a vast literature on this topic.

Variable ordering heuristics can be ranged into two categories:

– Static heuristics that choose the variable order once for all, prior to the construction of the binary decision diagram.
– Dynamic heuristics that start with a static heuristics then reorder the binary decision diagram while building it. This technique, introduced by Rudell, is called *sifting* (Rudell 1993).

Sifting and similar techniques make it possible to reduce very significantly the size of binary decision diagrams. They are however extremely time consuming. If they give good results in some domains, like electronic circuit verification, their performance on reliability models is rather poor. This is the reason why the current version of XFTA implements only static heuristics.

Most, if not all, static heuristics aim at keeping close in the ordering variables that are related in the formula. This idea is illustrated by example 12.1. The variables A_i and B_i, $i = 1, \ldots n$ are strongly related. Consequently, they must be close in the order. We can see them as a kind of macro-variable, an independent sub-function. In our example, this is what they really are, as the conjunction $(A_i \wedge B_i)$ forms a module, see Section 11.2.4.1. The same idea applies in the cases where although not forming modules, "close" variables tend to influence simultaneously the value of the function.

Static heuristics can themselves be ranged into two categories:

– Those based on a depth-first left-most traversal of the formula.
– Those based on other principles.

Heuristics of the first category work actually in two steps: first, they reorder arguments of connectives according to some metrics; then, they traverse the formula (or the system of Boolean equations) in a depth-first left-most manner and order variables accordingly.

The big advantage of these heuristics is that they tend to preserve the locality of variables: variables that are "close" in the formula tend to be close in the variable ordering. The initial reordering of arguments of connectives may improve, or on the contrary degrade, significantly the performance of the heuristic. As I showed in my 2008 article (Rauzy 2008c), there is unfortunately no silver bullet, i.e. no way to know *a priori* whether a metric used to reorder arguments will produce good or bad results.

12.2.3 Available Heuristics

The current version of XFTA implements several variable ordering heuristics, primarily for research purpose. Two of these heuristics are made available:

`DFLM`
: This heuristic order state variables thanks to a depth-first left-most traversal of the system of Boolean equations, hence its name. It ranks also the all variables (state and flow) in post-order of this traversal, and sorts arguments of connectives in increasing order of this rank.

`SCG-OAF`
: This heuristic is also based on a depth-first left-most traversal of the system of Boolean equations. However, arguments of connectives are sorted before being traversed. This family of heuristics is introduced by the prefix `SCG` that stands for "sort children and go". `OAF` stands for "order atoms first", i.e. arguments of connectives are sorted as follows.
  - State variables come first.
  - Then modules.
  - Then other flow variables, sorted in decreasing order of their numbers of parents.

  On many models, this heuristic has a better performance than the pure depth-first left-most traversal.

More heuristics will be made available in the next versions of XFTA.

The variable ordering can also be defined "by hand" using the command `set BDD-index`, see Section 12.6.8.

12.3 Extracting All Minimal Cutsets

If we want to extract all the minimal cutsets of a top event, one way to proceed consists in first building the binary decision diagram encoding the structuring function of the top event, then in building, from this binary decision diagram, the zero-suppressed binary decision diagram encoding the minimal cutsets. This process relies on a decomposition theorem I introduced in my 1993 article (Rauzy 1993) and later refined in my 2001 article (Rauzy 2001).

Before presenting it, we need to introduce the binary operator without that applies on sums of positive products.

Definition 12.3.1 — Without Operator. Let φ and ψ be two sums of positive products built over a set of variables $\mathscr{V}$. The sum of products φ without ψ, denoted $\varphi \div \psi$, is defined as follows.

$$\varphi \div \psi \quad \overset{def}{=} \quad \{\pi \in \varphi; \nexists \rho \in \psi; \rho \subseteq \pi\}$$

$\varphi \div \psi$ is thus obtained from the sum of products φ by removing all products that are subsumed by a product of ψ.

The following decomposition theorem shows how to implement the calculation of $\varphi \div \psi$ when both φ and ψ are encoded by zero-suppressed binary decision diagrams.

Theorem 12.3 — Decomposition Theorem for the Without Operator. Let $\varphi = E \odot \varphi_1 \cup \varphi_0$ and $\psi = E \odot \psi_1 \cup \psi_0$ be two sums of positive products over a set of variables $\mathscr{V}$, where E is a variable of $\mathscr{V}$, φ_1, φ_0, ψ_1 and ψ_0 are sums of positive products in which E does not show up. Then, the following equality holds.

$$\varphi \div \psi \quad = \quad E \odot ((\varphi_1 \div \psi_1) \div \psi_0) \cup (\varphi_0 \div \psi_0)$$

The proof follows from the definition: a product π of φ that does not contain E, i.e. a product of φ_0 can only be subsumed by a product of ψ that does not contain E, i.e. a product of ψ_0. Now, a product $E \cdot \pi$ of φ, i.e. a product π of φ_1 can be subsumed either by a product $E \cdot \rho$ of ψ, in which case ρ belongs to ψ_1, or by a product ρ of ψ that does not contain E, i.e. a product ρ of ψ_0.

We can now state the decomposition theorem for minimal cutsets.

Theorem 12.4 — Decomposition Theorem for Minimal Cutsets. Let $f = E \cdot f_1 + \overline{E} \cdot f_0$ be a Boolean formula built over a set of variables $\mathscr{V}$, where E is a variable of $\mathscr{V}$ and f_1 and f_0 are two Boolean formulas in which E does not show up. Then, the following equality holds.

$$\text{MCS}(f) \quad = \quad E \odot (\text{MCS}(f_1) \div \text{MCS}(f_0)) \cup \text{MCS}(f_0)$$

Proof: Clearly, minimal cutsets of f that do not contain E must be minimal cutsets of f_0, and reciprocally, minimal cutsets of f_0 are also minimal cutsets of f.

Now, a minimal cutset π of f_1 can be turned into a cutset $E \cdot \pi$ of f. π is minimal if there is no minimal cutset ρ of f_0 that subsumes π. Reciprocally, if $E \cdot \pi$ is minimal cutset of f, then π must be a minimal cutset of f_1 as it cannot be a cutset of f_0.

Theorems 12.3 and 12.4 give the core for the two recursive functions that extract minimal cutsets from a binary decision diagram and to encode them in a zero-suppressed binary decision diagram. Figure 12.3 and 12.4 give the pseudo-code of these two functions. Terminal cases for these functions are straightforward, e.g. $\varphi \div \emptyset = \varphi$, $\varphi \div \{\} = \emptyset$ and so on. Note that, as for the calculation of logical operators, both functions make an extensive use of caching.

Extraction of huge numbers of minimal cutsets
The process described in this section, which is implemented by the command `build ZBDD-from-BDD`, extracts all of the minimal cutsets of the considered top event. It does so without looking at each minimal cutset individually. This makes possible the extraction of huge numbers of minimal cutsets within reasonable computation times and memory occupations. However, extracting millions of minimal cutsets may be simply useless, even though they are encoded in a compact way: calculations of probabilistic risk indicators are better performed from the binary decision diagram and scaning that many minimal cutsets, without the possibility to sort them by decreasing order of probability, is just infeasible.

12.4 Extracting the Most Probable Minimal Cutsets

As pointed out in the warning concluding the previous section, it may be wise, once the binary decision diagram encoding the structure function of the top event, to extract a reasonable number of minimal cutsets that can be looked at by the analyst. Of course, if we do so, we want to get the 100, 1 000 or even 1 000 000 most probable ones, preferably in order. Unfortunately, this is not possible, at least not directly, using zero-suppressed binary decision diagrams. The latter have the drawbacks

```
Without(S, T):
   if S==0 or T==1:
      return 0
   if T==0:
      return S
   if S==1:
      return 1
   R = LoopUpCache(without, S, T)
   if R!=null:
      return R
   if S.index<T.index:
      R1 = Without(S.thenSon, T)
      R0 = Without(S.elseSon, T)
      R = NewZBDDNode(S.index, R1, R0)
   else if S.index>T.index:
      R = Without(S, T.elseSon)
   else:
      R0 = Without(S.elseSon, T.elseSon)
      R2 = Without(S.thenSon, T.thenSon)
      R1 = Without(R2, T.elseSon)
      R = NewZBDDNode(S.index, R1, R0)
   InsertInCache(without, S, T, R)
   return R
```

Figure 12.3: Pseudo-code of the function that calculates the without operator between two ZBDD

of their advantages (and vice-versa): they can encode huge numbers of cutsets in a compact way, but they cannot be used to sort these cutsets.

A solution consists then in buiding a binary decision tree, rather than a zero-suppressed binary decision diagram. binary decision trees have also the drawbacks of their advantages (and vice-versa): they make it possible to sort minimal cutsets, but they force to consider minimal cutsets one by one, which limits indeed the number of minimal cutsets that can be looked at. Moreover, their size is roughly in $\mathcal{O}(|\mathrm{SoP}(\varphi)|)$, where φ is the sum of minimal cutsets.

The algorithm that extract minimal cutsets from a binary decision diagram and stores them into a binary decision tree relies on a recursive traversal of the binary decision diagram. All paths from the root to the leaf 1 are successively explored. Positive literals are collected along the way. Their conjunction is a candidate cutset that can be checked for minimality and possibly inserted into the binary decision tree. Cutoffs are used to stop the exploration of branches. Figure 12.5 gives the pseudo-code of this algorithm.

Note that, as in the branch-and-test algorithm, the cutoff is dynamically adjusted so to keep only the *n* most probable minimal cutsets, see Section 11.2.3 for a discussion.

The binary decision diagram is used to check minimality of candidate cutsets. The idea is to go down along the branch encoding the candidate cutset and to check along the way that no proper sub-product of the candidate cutset is itself a cutset. The pseudo-code for this algorithm is given in Figure 12.6.

The algorithms described in this section are implemented by the command `build BDT-from-BDD`.

The extraction mechanism presented in this section has however a flaw: it may explore an enormous number of branches, that encode to minimal cutsets, before finding those encoding minimal cutsets. This is the reason why it is mandatory to initiate the algorithm with the suitable cutoffs. An option has been introduced for the command `build BDT-from-BDD` to do so.

```
MCS(F):
   if F==0
      return 0
   if F==1:
      return 1
   R = LoopUpCache(mcs, F)
   if R!=null:
      return R
   R0 = MCS(F.elseSon)
   R2 = MCS(F.thenSon)
   R1 = Without(R2, R0)
   R = NewZBDDNode(F.index, R1, R0)
   InsertInCache(mcs, F, R)
   return R
```

Figure 12.4: Pseudo-code of the function that builds a ZBDD encoding the minimal cutsets of a BDD

12.5 Weighted Boolean Algebras

Another approach to extract only the most significant minimal cutsets consists in working with binary decision diagrams and zero-suppressed binary decision diagrams in weighted Boolean algebras. I introduced this idea in 2001 article (Rauzy 2001) and later refined it in an unnoticed conference communication (Rauzy 2010).

12.5.1 Mathematical Framework

The key idea consists in associating a weight to each basic event, then in defining the weight of a product as the sum of the weights of the positive literals showing up in this product. Formally,

Definition 12.5.1 — Weights. Let $\mathcal{V}$ be a set of Boolean variables. A *weighting function* or simply a *weight* is a function w from $\mathcal{V}$ to $\mathbb{R}^+$, i.e. a function that associates a positive real number $w(V)$ to each variable V of $\mathcal{V}$.

Weights are then lifted up to literals and products as follows.

$$\begin{aligned} w(\neg V) &\overset{def}{=} 0 \\ w(\pi) &\overset{def}{=} \sum_{L\in\pi} w(L) \end{aligned}$$

We can use weights to define cutoffs, i.e. maximum weights of extracted minimal cutsets:

– If we want a cutoff on order of minimal cutsets, we can simply set the weight of all basic events to 1.
– If we want a cutoff on the probability of minimal cutsets, we can define the weights of basic events as follows.

$$w(V) \overset{def}{=} -\log_\alpha(p(V)) \tag{12.2}$$

where α is a scale factor (we shall see its interest in the next section).

So far, we did nothing special, just a reformulation of the notion of cutoff.

Now, we can look at what it means, from a mathematical standpoint, to discard minimal cutsets whose weights are larger than a given maximum w_{max}. What it actually means is that we want to consider only minterms whose weights are lower than w_{max}. This set of minterms is called the care set.

```
BuildBDTFromBDD(F, $p_{min}$):
   // F: BDD Node
   // $p_{min}$: minimum probability of minimal cutsets
   $\pi$ = 1
   BDT = new empty binary decision tree
   BuildBDTFromBDD(F, $\pi$, $p_{min}$, F, BDT)
   return BDT

BuildBDTFromBDD(F, $\pi$, $p_{min}$, T, BDT):
   // $\pi$: candidate cutset
   if p($\pi$) < $p_{min}$:
      return
   if F=Leaf(0):
      return
   if F=Leaf(1):
      if IsMinimalCutset(T, $\pi$):
         Insert(BDT, $\pi$, $p_{min}$)
      return
   // F is an internal node
   BuildBDTFromBDD(F.thenSon, $\pi$·F.variable, $p_{min}$, T, BDT)
   BuildBDTFromBDD(F.elseSon, $\pi$, $p_{min}$, T, BDT)
```

Figure 12.5: Algorithm to extract minimal cutsets from a BDD and to store them into a BDT

Definition 12.5.2 — Care Set. Let $\mathscr{V}$ be a set of Boolean variables, let w a weight defined over $\mathscr{V}$, and let finally w_{max} a positive real number. The *care set* of $\mathscr{V}$ for w and w_{max}, denoted as $\text{CareSet}_{\mathscr{V},w}(w_{max})$, is the set of minterms defined as follows.

$$\text{CareSet}_{\mathscr{V},w}(w_{max}) \overset{def}{=} \{\sigma \in \text{minterms}(\mathscr{V})\,;\; w(\sigma) \leq w_{max}\}$$

In contrast, the set of minterms whose weights are larger than w_{max} is called the *don't care set*.

In the sequel, we shall interpret $\text{CareSet}_{\mathscr{V},w}(w_{max})$ as the disjunction of its elements, just as any other set of products. With this interpretation, we can use $\text{CareSet}_{\mathscr{V},w}(w_{max})$ to filter out minterms of interest, i.e. to interpret any Boolean formula f built over $\mathscr{V}$ as $f \cdot \text{CareSet}_{\mathscr{V},w}(w_{max})$.

The following property shows the orthogonality of this filtering with the Shannon decomposition.

Property 12.5 — Orthogonality of Care Sets with the Shannon Decomposition. Let $\mathscr{V}$ be a set of Boolean variables. Let $f = E \cdot f_1 + \overline{E} \cdot f_0$ be a Boolean formula built over $\mathscr{V}$, where E is a variable of $\mathscr{V}$, and f_1 and f_0 are Boolean formulas built over $\mathscr{V}$ in which E does not show up. Finally, let w a weight defined over $\mathscr{V}$, and w_{max} a positive real number. Then, the following equivalence holds.

$$f \cdot \text{CareSet}_{\mathscr{V},w}(w_{max}) \equiv E \cdot r_1 + \overline{E} \cdot r_0$$

where,

$$\begin{aligned} r_1 &= f_1 \cdot \text{CareSet}_{\mathscr{V},w}(w_{max} - w(E)) \\ r_0 &= f_0 \cdot \text{CareSet}_{\mathscr{V},w}(w_{max}) \end{aligned}$$

The above property provides us with a recursive principle to filter out binary decision diagrams. It would be however cumbersome to first build binary decision diagrams then filter them out.

```
IsMinimalCutset(F, $\pi$):
   if F=Leaf(0):
      return false
   if F=Leaf(1):
      return true
   if F.variable $\notin \pi$:
      return IsMinimalCutset(F.elseSon, $\pi$)
   return IsMinimalCutset(F.thenSon, $\pi$) $\wedge$ $\neg$IsCutset(F.elseSon, $\pi$)

IsCutset(F, $\pi$):
   if F=Leaf(0):
      return false
   if F=Leaf(1):
      return true
   if F.variable $\notin \pi$:
      return IsCutset(F.elseSon, $\pi$)
   return IsCutset(F.thenSon, $\pi$)
```

Figure 12.6: Algorithm to check whether a candidate cutset is minimal

Fortunately, care sets can be applied while creating binary decision diagrams, thanks to the following property (which is just a consequence of the previous one).

Property 12.6 — Orthogonality of Care Sets with Logical Operators. Let $\mathcal{V}$ be a set of Boolean variables. Let $f = E \cdot f_1 + \overline{E} \cdot f_0$ and $g = E \cdot g_1 + \overline{E} \cdot g_0$ be a Boolean formula built over $\mathcal{V}$, where E is a variable of $\mathcal{V}$, and f_1, f_0, g_1 and g_0 are Boolean formulas built over $\mathcal{V}$ in which E does not show up. Finally, let w a weight defined over $\mathcal{V}$, and w_{max} a positive real number. Then, the following equivalence holds.

$$\begin{aligned}
(f+g) \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max}) &\equiv E \cdot r_1 + \overline{E} \cdot r_0 \\
(f \cdot g) \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max}) &\equiv E \cdot s_1 + \overline{E} \cdot s_0 \\
\overline{f} \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max}) &\equiv E \cdot u_1 + \overline{E} \cdot u_0
\end{aligned}$$

where,

$$\begin{aligned}
r_1 &= (f_1+g_1) \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max} - w(E)) \\
r_0 &= (f_0+g_0) \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max}) \\
s_1 &= (f_1 \cdot g_1) \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max} - w(E)) \\
s_0 &= (f_0 \cdot g_0) \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max}) \\
u_1 &= \overline{f_1} \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max} - w(E)) \\
u_0 &= \overline{f_0} \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max})
\end{aligned}$$

The same idea applies to the decomposition theorem for minimal cutsets.

Theorem 12.7 — Care Sets and Decomposition Theorem for Minimal Cutsets. Let $f = E \cdot f_1 + \overline{E} \cdot f_0$ be a Boolean formula built over a set of variables $\mathcal{V}$, where E is a variable of $\mathcal{V}$ and f_1 and f_0 are two Boolean formulas in which E does not show up. Moreover, let w a weight defined over $\mathcal{V}$, and w_{max} a positive real number. Then, the following equality holds.

$$\mathrm{MCS}\left(f \cdot \mathrm{CareSet}_{\mathcal{V},w}(w_{max})\right) = E \odot (\varphi_1 \div \varphi_0) \cup \varphi_0$$

where,

$$\begin{aligned}\varphi_1 &= \mathrm{MCS}\left(f_1 \cdot \mathrm{CareSet}_{\mathscr{V},w}\left(w_{max} - w(E)\right)\right)\\ \varphi_0 &= \mathrm{MCS}\left(f_0 \cdot \mathrm{CareSet}_{\mathscr{V},w}\left(w_{max}\right)\right)\end{aligned}$$

Note that it is not necessary to apply the filtering on the without operation as both of its arguments are already filtered.

12.5.2 Algorithmic Framework

Properties 12.6 and 12.7 provide us with the recursive principles to build binary decision diagrams encoding weighted structure functions, as well as to extract weighted minimal cutsets and store them into zero-suppressed binary decision diagrams. There is a problem however: how to perform caching? Clearly, the caching must take into account maximum weights as the results of calculations depend on them. However, there are little chances that the same operation is called twice with exactly the same maximum weight.

To solve this problem, the idea is to approximate, for caching, the current maximum weight by the integer immediately bigger. Figure 12.7 shows the implementation of this principle for the function that filters a binary decision diagram with a given maximum weight. The same principle applies for the functions calculating logical operators and extracting minimal cutsets. We see now the importance of the scale factor α in Equation 12.2: the bigger α, the more efficient the caching, but meanwhile the more approximated the extraction. By default, XFTA sets the value of α to e, i.e. define $w(V)$ as $-\log(p(V))$.

```
Filter(F, $w_{max}$):
   if $w_{max} < 0$ or F==0
      return 0
   if F==1:
      return 1
   R = LoopUpCache(filter, F, $\lceil w_{max} \rceil$)
   if R!=null:
      return R
   R1 = Filter(F.thenSon, $w_{max} - w($F.variable$)$)
   R0 = Filter(F.elseSon, $w_{max}$)
   R = NewBDDNode(F.index, R1, R0)
   InsertInCache(filter, F, $\lceil w_{max} \rceil$, R)
   return R
```

Figure 12.7: Pseudo-code of the function that filters a BDD with a weight

The commands `build WBDD` and `build WZBDD-from-BDD` implement respectively the construction of the weighted binary decision diagram and of the weighted zero-suppressed binary decision diagram encoding the minimal cutsets.

12.6 Commands Implementing the Binary Decision Diagram Approach

The developments of this chapter complete the picture sketched 9. Figure 12.8 shows the actual assessment flows of the current version of XFTA and the commands that implement them.

The remainder of this section describes these commands and their options.

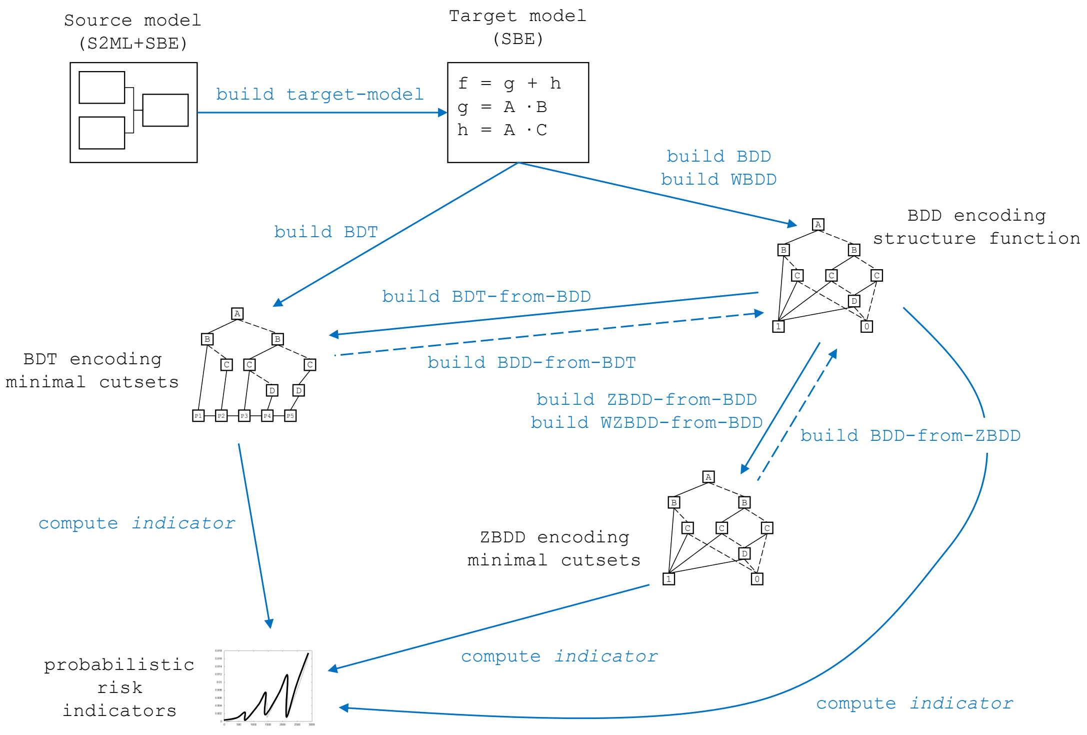


Figure 12.8: Completed assessment flows

12.6.1 Command `build BDD`

The main command to build the binary decision diagram encoding the structure function of a variable is `build BDD`, see Section 12.1. Its base form is as follows (assuming the variable `Top` is the variable were are interested in).

```
build BDD Top;
```

Options of the command `build BDD` are summarized in Table 12.1.

Table 12.1: Options of the command `build BDD`

Option	Type	Default value
`target-handle`	Identifier	`BDD`
`variable-ordering-heuristic`	Identifier	`DFLM`
`keep-intermediate-BDD-handles`	Boolean	`true`

12.6.1.1 Target Handle

By default, the binary decision diagram encoding the structure function of variable is stored in a handle named `BDD`.

The option `target-handle` makes it possible to give another name to the handle. E.g.

```
build BDD Top target-handle=BDD4;
```

12.6.1.2 Variable Ordering Heuristics

As explained Section 12.2, the sizes of binary decision diagrams and therefore the efficiency of the whole technology

Building a binary decision tree requires defining a total order over the variables showing up in the tree. This is achieved by assigning an index to each variable and sorting variables according to their indices (two variables cannot have the same index).

Although much less dramatic than in the case of binary decision diagrams, the chosen variable order may have a small influence on the size of the binary decision tree. As there is no way to predict what is the best ordering, nor even a good one, heuristics are used. Heuristics to order variables in binary decision trees are the same than those used for binary decision diagrams. We report thus the reader to Section 12.2 for a discussion and a presentation of heuristics available in the current version of XFTA.

The variable ordering heuristic can be chosen via the option `variable-ordering-heuristic` or the corresponding environment variable. Its default value is `DLFM`.

```
build BDT Top variable-ordering-heuristic=SCG-OAF;
```

12.6.1.3 Intermediate Handles

To build the binary decision diagram encoding the structure function of the target variable, one needs to build the binary decision diagrams of all descendants of the target variable, i.e. all the state, source and flow variables involved in the definition of the target variable.

By default, these intermediate binary decision diagrams are stored in handles. By default, the name of these handles is `BDD`. This default behavior can be changed, as explained in the previous section.

If intermediate binary decision diagrams are of no interest, it is not worth no to keep them, as they may occupy a very significant amount of memory. The Boolean option `keep-intermediate-BDD-handles` and the corresponding environment variable can be used to tell XFTA to keep or not to keep intermediate handles. E.g.

```
set option keep-intermediate-BDD-handles false;
build BDD Top;
```

12.6.2 Command `build ZBDD-from-BDD`

The command `build ZBDD-from-BDD` builds a zero-suppressed binary diagram encoding the minimal cutsets of the (function encoded by) the binary decision diagram associated with a given target variable, see Section 12.3. Its base form is as follows.

```
build ZBDD-from-BDD Top;
```

Its options are given in Table 12.2, i.e. takes only two options:

– `source-handle` that specifies the name of the BDD from which the calculation is performed. By default, the first BDD handle (in lexicographic order) is chosen.
– `target-handle` that specifies the name of the ZBDD in which the minimal cutsets are stored, `ZBDD` is chosen by default.

Table 12.2: Options of the command `build ZBDD-from-BDD`

Option	Type	Default value
`source-handle`	Identifier	None
`target-handle`	Identifier	`ZBDD`

12.6.3 Command `build BDT-from-BDD`

The command `build BDT-from-BDD` builds a binary decision tree encoding the (most probable) minimal cutsets of (the function encoded by) a binary decision diagram associated with a given variable, see Section 12.4. Its base form is as follows.

```
build BDT-from-BDD Top;
```

Its options are given in Table 12.3.

Table 12.3: Options of the command `build BDT-from-BDD`

Option	Type	Default value
`maximum-number`	Positive integer	1,000,000
`maximum-order`	Positive integer	100
`minimum-probability`	Real in $[0,1]$	0.0
`minimum-contribution`	Real in $[0,1]$	0.0
`mission-time`	Positive real	0.0
`source-handle`	Identifier	None
`target-handle`	Identifier	`BDT`
`variable-ordering-heuristic`	Identifier	`DFLM`

The options describing the cutoff are:
- `maximum-number` that sets the maximum number of extracted minimal cutsets;
- `maximum-order` that sets the maximum order, i.e. the maximum number of variables, of extracted minimal cutsets;
- `minimum-probability` that sets the minimum probability of extracted minimal cutsets;
- `minimum-contribution` that sets the minimum contribution of extracted minimal cutsets, where the contribution is defined as the ratio of the probability of the cutset with the probability of the structure function, computed from the binary decision diagram,
- `mission-time` that sets the mission time at which probabilities of basic events are calculated.

These options are discussed in details Section 11.4.1.1. We shall thus not reproduce this discussion here.

As for the command `build ZBDD-from-BDD`, the following options set the names of source and target data structures.
- `source-handle` that specifies the name of the BDD from which the calculation is performed. By default, the first BDD handle (in lexicographic order) is chosen.
- `target-handle` that specifies the name of the BDT in which the minimal cutsets are stored, `BDT` is chosen by default.

Finally, building a binary decision tree requires selecting an order for the variables. This order is chosen with a heuristic that is specified with the option `variable-ordering-heuristic`. Available heuristics are the same as for binary decision diagrams, see Sections 12.2 and 12.6.1.2.

⚠ **Command `build BDT-from-BDD`**
To avoid unending calculations, it is mandatory to call the command `build BDT-from-BDD` with a cutoff, maximum order, minimum probability or minimum contribution, making it possible to stop as early as possible the exploration of the branches of the binary decision diagram. The minimum contribution is probably the easiest way to ensure fast executions of this command.

12.6.4 Command `build WBDD`

The command `build WBDD` builds a binary decision diagram encoding the weighted structure function of the given variable. Its base form is as follows.

```
build WBDD Top;
```

The options of this command are a subset of the options of the command `build BDT` summarized in Table 11.1.

The first role of these options is to describe the cutoff, i.e. the maximum weight, namely:

– `maximum-order` that sets the maximum order, i.e. the maximum number of variables, of extracted minimal cutsets;
– `minimum-probability` that sets the minimum probability of extracted minimal cutsets;
– `weight-scale-factor` that sets the scale factor α used to calculate weights from probabilities;
– `mission-time` that sets the mission time at which probabilities of basic events are calculated.

Recall that the maximum weight is defined either as a maximum order or as a minimum probability, but not both. Recall also that the option `maximum-number` is meaningless here.

The three other options are the following.

– `target-handle` that specifies the name of the WBDD in which the minimal cutsets are stored, `BDD` is chosen by default.
– `variable-ordering-heuristic` that specifies the variable ordering heuristic to be used, `DFLM` by default value.
– `keep-intermediate-BDD-handles` that specifies whether intermediate BDD must be kept or not, `true` by default.

These options are described in further details in Section 12.6.1.

12.6.5 Command `build WZBDD-from-BDD`

The command `build WZBDD-from-BDD` builds a zero-suppressed binary decision diagram encoding the weighted minimal cutsets from a binary decision diagram encoding the structure function of the given variable. Its base form is as follows.

```
build WZBDD-from-BDD Top;
```

The options of this command have two roles: defining the cutoff and setting the source and target handles.

Options describing the cutoff, i.e. the maximum weight, are the following:

– `maximum-order` that sets the maximum order, i.e. the maximum number of variables, of extracted minimal cutsets;
– `minimum-probability` that sets the minimum probability of extracted minimal cutsets;
– `minimum-contribution` that sets the minimum contribution of extracted minimal cutsets, see Section 12.6.3;
– `weight-scale-factor` that sets the scale factor α used to calculate weights from probabilities;
– `mission-time` that sets the mission time at which probabilities of basic events are calculated.

Recall that the maximum weight is defined either as a maximum order or as a minimum probability, but not both. Recall also that the option `maximum-number` is meaningless here.

The three other options are the following.

– `source-handle` that specifies the name of the BDD from which the calculation is performed. By default, the first BDD handle (in lexicographic order) is chosen.
– `target-handle` that specifies the name of the WZBDD in which the minimal cutsets are stored, `ZBDD` is chosen by default.

Note that the source binary decision diagram does not need to be a weighted binary decision diagram.

12.6.6 Command `build BDD-from-BDT`

The command `build BDD-from-BDT` builds a binary decision diagram encoding the disjunction of the minimal cutsets of encoded by the source binary decision tree. Its base form is as follows.

```
build BDD-from-BDT Top;
```

It has two options:
– `source-handle` that specifies the name of the BDT from which the calculation is performed. By default, the first BDT handle (in lexicographic order) is chosen.
– `target-handle` that specifies the name of the target BDD, `BDD` is chosen by default.

12.6.7 Command `build BDD-from-ZBDD`

The command `build BDD-from-ZBDD` builds a binary decision diagram encoding the disjunction of the minimal cutsets of encoded by the source zero-suppressed binary decision diagram. Its base form is as follows.

```
build BDD-from-ZBDD Top;
```

It has two options:
– `source-handle` that specifies the name of the ZBDD from which the calculation is performed. By default, the first ZBDD handle (in lexicographic order) is chosen.
– `target-handle` that specifies the name of the target BDD, `BDD` is chosen by default.

12.6.8 Commands to Set and Print BDD Indices

Variable ordering heuristics work by assigning a different index to each basic event and module. The indices chosen for binary decision trees are not necessarily the same as for binary decision diagrams and zero-suppressed binary decision diagrams.

It is possible to print these indices by using following commands,

`print BDD-index` *`variable Options`*`;`
that prints the BDD index of the given variable.

`print BDD-indices` *`variable Options`*`;`
that prints the BDD indices of all basic events involved in the structure function of the given variable.

`print BDT-index` *`variable Options`*`;`
that prints the BDT index of the given variable.

`print BDT-indices` *`variable Options`*`;`
that prints the BDT indices of all basic events involved in the structure function of the given variable.

Options of these commands are `output` that sets the name of the output file and `mode` that sets the opening mode (`write` or `append`).

Indices are printed as `set` commands so that they can be reloaded. The command `set` makes it possible to set by hand BDD and BDT indices as follows.

`set BDD-index` *`variable index`*`;`
  that sets the BDD index of the given variable.

`set BDT-index` *`variable index`*`;`
  that prints the BDT index of the given variable.

Indices must be positive integers between 1 and 16384.

⚠ **BDD and BDT indices of variables**
Different basic events must be given different BDD and BDT indices. Moreover, it is recommended to chose low values for indices as high values would require to enlarge uselessly the BDT and BDD entry tables.

12.7 Fine Tuning of BDD Tables

The BDD package implemented in XFTA involves several tables to store or to access BDD nodes. These tables are pictured in Figure 12.9.

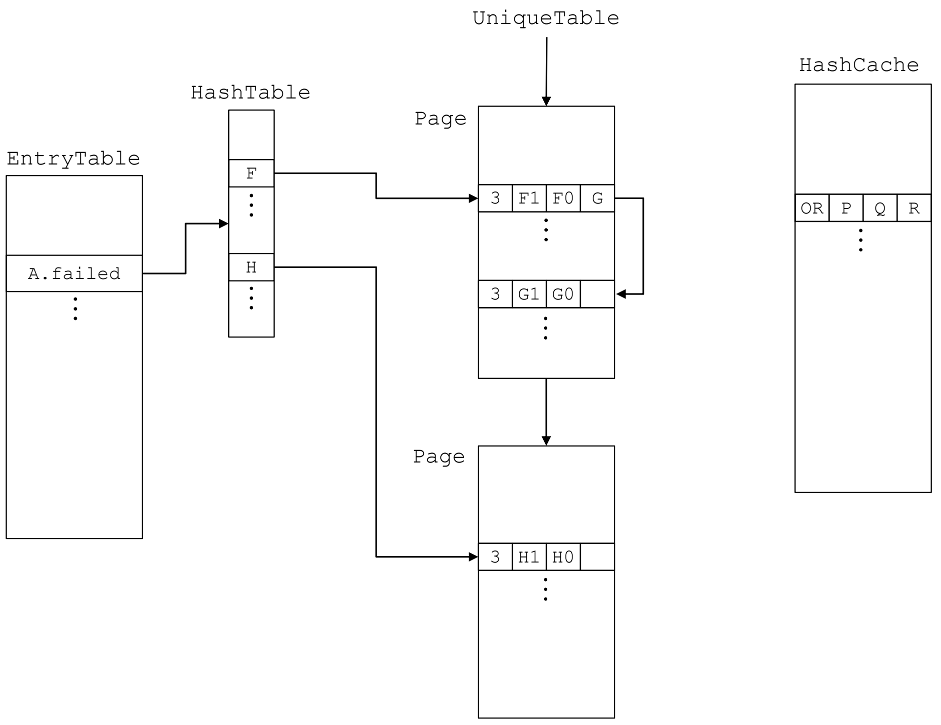


Figure 12.9: Binary decision diagram tables

Their roles are the following:

– The *entry-table* contains one entry per state variable (or module). It manages also a hashtable of references to BDD nodes.
– The BDD nodes are manages into the *unique-table*. In order to optimize the management of the memory and to avoid allocating too many contiguous memory cells, the unique table manages actually a list of pages.

– Each *page* contains a certain number of BDD nodes (and is linked to the next page).
– *Hashtables*, one per variable, that make it possible to access efficiently nodes labeled with that variable, via collision chains.
– The *hashcache* stores results of calculations.

By default, these tables are given sizes so to optimize the treatment of middle and large size models. Handling very large size models requires to tune their sizes. This is the reason why XFTA provides commands to set (and to print) the sizes of the BDD tables.

Sizes of BDD tables
The sizes of BDD tables must be set prior any calculations involving binary decision diagrams.

For internal reasons, default sizes of BDD tables are power of 2. Table 12.4 recalls a number of power of 2 that can be used to define size (if your computer has enough memory).

Table 12.4: Powers of two

.

10	11	12	13	14	15	16
1 024	2 048	4 096	8 192	16 384	32 768	65 536
17	18	19	20	21	22	23
131 072	262 144	524 288	1 048 576	2 097 152	4 194 304	8 388 608

The command `print BDD-statistics` prints out statistics on sizes and usage of BDD tables. E.g.

```
print BDD-statistics output="BDD-stats.txt";
```

Command `print BDD-statistics`
The command `print BDD-statistics` may take a rather long time, as it traverses BDD tables, including collision chains.

The following commands give indicators that are faster to calculate.
– `print BDD-number-of-nodes` prints the total number of nodes stored in the unique table.
– `print BDD-number-of-pages` prints the number of pages in the unique table.
– `print BDD-number-of-active-nodes` prints the total number of active nodes stored in the unique table, i.e. the number of nodes that are used in at least one binary decision diagram associated with a variable.
Note that the printed number may be overestimated: to get the actual number of active nodes, it would be necessary to launch a garbage collection first.

12.7.1 Unique-Table

A BDD node is encoded onto 4 words in a 64 bits machines, i.e. onto $32 = 2^5$ bytes. The overhead due to the management of the unique table and the pages is negligible. This means that the maximum memory occupation of the unique table (in bytes) is about 32 times the maximum number of BDD nodes. If this maximum number is set to 1 million nodes, the unique table occupies, 32 millions bytes, i.e. 32 megabytes.

The maximum number of BDD nodes can be printed out by means of the command `print BDD-table-size` and modified by the command `set BDD-table-size`. E.g.

```
set option mode append;
print BDD-table-size output="BDD-stats.txt";
set BDD-table-size 100000;
```

By default, the maximum size of the unique table is set to $2^{24} = 16\,777\,216$ nodes, which occupies a bit more than half gigabyte.

Note that when the unique table is full, a garbage process is applied to free all unused nodes.

In the current version of XFTA, it is also possible to print and to modify the size, i.e. the number of nodes, of pages. This is achieved by means of the commands `print BDD-page-size` and `set BDD-page-size`.

```
print BDD-page-size output="BDD-stats.txt" mode=append;
set BDD-page-size 65536;
```

Pages are allocated on-demand, until the maximum number of nodes is reached. Allocating large pages reduces the number of memory allocations, which means that pages should be as large as possible. On the other hand, allocating very large pages may occupy memory uselessly and fail due to the lack of large enough available contiguous memory zones. There is thus a trade-off to find. By default, the size of pages is set to $2^{15} = 32\,768$ nodes. In other words, by default XFTA can allocate $2^9 = 512$ pages, which is a reasonable trade-off.

12.7.2 Hashtables

Hashtables make possible to check efficiently whether a BDD node already exist in the unique table. When several nodes have the same hashcode, the hashtable manages them in a collision chain, i.e. in a list. The longer the chain, the less efficient the checking process. Consequently, the larger the hashtable, the more efficient the checking process, under the condition that the hashing distributes more or less equally the BDD nodes on the entries of the hashtable. This said, the number of nodes referred by a hashtable may differ dramatically from one variable to the other. Intuitively, relatively few nodes are labeled with (the indices of) variables located at the top and the bottom of the BDD, while many nodes are labeled with (the indices of) variables located in the middle of it. BDD have thin heads and legs, but fat bellies. This is the reason why hashtables associated with variables are dynamically resized depending the numbers of BDD nodes they contain.

The commands `print BDD-hashtable-size` and `set BDD-hashtable-size` make it possible to print and set the maximum size of hashatbles. E.g.

```
print BDD-hashtable-size output="BDD-stats.txt" mode=append;
set BDD-hashtable-size 8192;
```

By default, this maximum size is set to $2^{16} = 65\,536$. This is normally sufficient, including to deal with rather large models. As a hashtable is essentially a table of references to nodes, it occupies roughly 8 times its size bytes on a 64 bits machine. This means that a hashtable of size $2^{16} = 65\,536$ occupies around a quater megabyte. This is not much, but recall that each variable has its own hashtable.

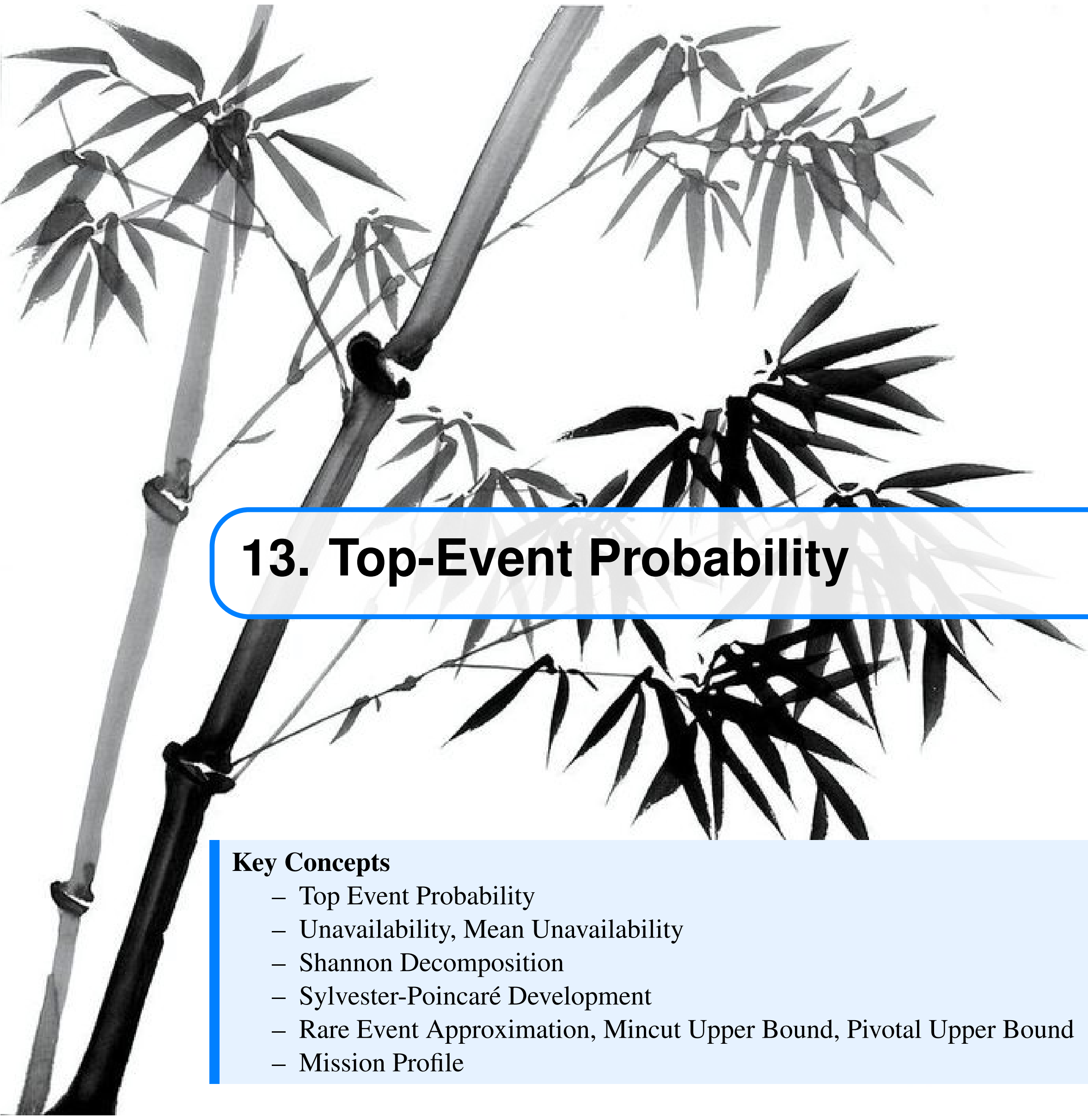
12.7.3 Hashcache

The hashcache plays a central role in the efficiency of the BDD technology. With that respect, the larger the better. However, there is a trade-off to find as the memory occupied by the hashcache cannot be occupied by BDD nodes. Each entry of the hashcache is encoded onto 4 words in a 64 bits machines, i.e. onto 32 bytes. This means that a hashcache containing 1 million entries occupies 32 megabytes.

The size of the hashcache can be printed out by means of the command `print BDD-cache--size` and modified by the command `set BDD-cache-size`. E.g.

```
print BDD-cache-size output="BDD-stats.txt" mode=append;
set BDD-cache-size 524288;
```

By default, this maximum size of hashcache is set to $2^{17} = 131\,072$. To handle very large models, it is recommended to enlarge it significantly, e.g. to $2^{23} = 8\,388\,608$ entries. This requires however that there is a free, contiguous memory zone of about a quarter gigabyte.



13. Top-Event Probability

Key Concepts
- Top Event Probability
- Unavailability, Mean Unavailability
- Shannon Decomposition
- Sylvester-Poincaré Development
- Rare Event Approximation, Mincut Upper Bound, Pivotal Upper Bound
- Mission Profile

This chapter describes XFTA algorithms and commands to assess the probability of the top event of a model, given the probabilities of its basic events. The probability of the top event can be calculated for different mission times, so to get an idea of its evolution. From the empirical distribution of the top event probability, it is possible to calculate its empirical average value.

13.1 Mathematical and Algorithmic Framework

13.1.1 Unavailability, Mean Unavailability and their Distributions

Recall that the *availability* $A_S(t)$ of a system S at time t is the probability that S is working at time t, given it was working at time 0. (see Section 3.1.2). Its *unavailability* is then defined simply as:

$$Q_S(t) \overset{def}{=} 1 - A_S(t)$$

By calculating the probability of the top event of a model, we are actually assessing the unavailability of the system under study.

By calculating this probability for sufficiently many points in time, we can thus get an empirical distribution of the unavailability of the system.

The mean unavailability of a system is defined as follows.

Definition 13.1.1 — Mean Unavailability. The *mean unavailability* of a system S at time t is its mean value from time 0 to time t:

$$Q_S^{avg}(t) \overset{def}{=} \int_0^t Q_S(t)\, dt$$

By calculating this probability for sufficiently many points in time, we can thus also get an empirical distribution of the mean unavailability of the system.

It remains to determine “sufficiently many points in time” so that the calculated empirical distributions of the unavailability and the mean unavailability are close to their actual distributions.

XFTA provides mechanisms to:

– Define a set of dates $t_1 < t_2 < \ldots < t_n$ at which the computation must be performed. This set of dates can be defined either by giving the list of the dates or by constructing it iteratively from an initial date t_{min} to a terminal date t_{max} with a step s. In this latter case, dates are $t_{min}, t_{min}+s, t_{min}+2s\ldots$.
– Add automatically *singular points*, which are determined from the probability distributions of basic events. Singular points are typically the dates at which the maintenance of a periodically tested component starts and stops. These dates create discontinuities in the unavailability distribution. It is thus important to add them.
– Smoothen the distribution, i.e. add intermediate dates in case the values calculated at t_i and t_{i+1} are too different.

We shall see in the next section how to use these mechanisms.

13.1.2 Quantification Methods

In XFTA, the assessment of the probability of the top event of a model is performed either via sums of disjoint products (encoded by means of binary decision diagrams) or sums of minimal cutsets (encoded either by binary decision trees or by zero-suppressed binary decision diagrams), see Section 9 for explanations.

In case the probability of the top event is assessed from a sum of disjoint products:

– What is truly assessed is the probability of the structure function associated with the top event.
– The calculated value is exact.

In case the probability of the top event is assessed from a sum of minimal cutsets:

– What is truly assessed is the probability of this sum of minimal cutsets.
– The calculated value is only estimated, as calculating the exact probability would be too resource consuming.

13.1.2.1 Calculating the Top Event Probability from a Sum of Disjoint Products

In XFTA, *sums of disjoint products* are encoded by means of *binary decision diagrams*. Recall that the latter rely on the *Shannon decomposition* of Boolean formulae. Let φ a Boolean function built over a set of variables $\mathcal{V}$ and $E \in \mathrm{var}(\varphi)$. Then the following equality holds.

$$\varphi \equiv E\cdot\varphi(E=1) + \overline{E}\cdot\varphi(E=0)$$

The Shannon decomposition is easily extended to probability calculation (the following property is actually the original formulation of the decomposition given by Claude Shannon).

Property 13.1 — Shannon Decomposition (Probability). Let φ a Boolean function built over a set of variables $\mathcal{V}$ and $E \in \mathrm{var}(\varphi)$. Then the following equality holds.

$$p(\varphi) \equiv p(E)\times p(\varphi \mid E) + (1-p(E))\times p\left(\varphi \mid \overline{E}\right)$$

The above property makes it possible to calculate the exact value of the top event probability from the probabilities of basic events and the binary decision diagram encoding the top event.

Thanks to caching, this calculation is performed in linear time with respect to the size of the binary decision diagram (Rauzy 1993).

13.1.2.2 Calculating the Top Event Probability from a Sum of Minimal Cutsets

The exact probability of a product is simply the product of the probabilities of the literals of the product. This applies indeed to minimal cutsets.

The probability of a sum of products is however more difficult to assess, as two products of the sum are in general non independent, i.e. there is at least one minterm that satisfies both.

The Sylvester-Poincaré development, that generalizes the well-known formula $p(A \cup B) = p(A) + p(B) - p(A \cap B)$, makes it possible, at least in theory, to compute the exact probability of a sum of products.

Definition 13.1.2 — Sylvester-Poincaré Development. Let $\varphi = \pi_1 + \cdots + \pi_n$ be a sum of products built over a set of variables $\mathcal{V}$. Then the probability of φ can be calculated by means of the following equation.

$$p(\varphi) \overset{def}{=} \sum_{1 \leq i \leq n} p(\pi_i) - \sum_{1 \leq i_1 < i_2 \leq n} p(\pi_{i_1} \cdot \pi_{i_2}) + \sum_{1 \leq i_1 < i_2 < i_3 \leq n} p(\pi_{i_1} \cdot \pi_{i_2} \cdot \pi_{i_3}) \cdots$$

In practice, applying this method is infeasible but for very small number of products. The number of terms of the development is actually $2^n - 1$.

To overcome this problem, several approximation methods have been proposed.

The first one, so-called *rare event approximation* consists in calculating only the first term of the development.

Definition 13.1.3 — Rare Event Approximation. Let $\varphi = \pi_1 + \cdots + \pi_n$ be a sum of products built over a set of variables $\mathcal{V}$. The rare event approximation REA_φ of $p(\varphi)$ is defined as follows.

$$\text{REA}_\varphi \overset{def}{=} \sum_{1 \leq i \leq n} p(\pi_i)$$

In XFTA, *sums of minimal cutsets* are encoded by means of *binary decision trees* and *zero-suppressed binary decision diagrams*. Both rely on the so-called pivotal decomposition of sum of positive products, see Section 9.3.2. Let φ be a sum of positive products built over a set of variables $\mathcal{V}$ and let E be a variable showing up in φ. Then, we can decompose φ according to E as follows.

$$\varphi \equiv E \cdot \varphi(E=1) + \varphi(E=0)$$

In the above equation, we denoted by slight abuse:

- $\varphi(E=1)$ the sum of positive products $\sum_{E \cdot \pi \in \varphi} \pi$.
- $\varphi(E=0)$ the sum of positive products $\sum_{E \notin \pi \wedge \pi \in \varphi} \pi$.

The rare event approximation can thus be calculated recursively:

Property 13.2 — Rare Event Approximation (Decomposition). Let φ be a sum of positive products built over a set of variables $\mathcal{V}$ and let E be a variable showing up in φ. Then the rare event approximation of $p(\varphi)$ can be calculated by means of the following equation.

$$\text{REA}_\varphi \equiv p(E) \times \text{REA}_{\varphi(E=1)} + \text{REA}_{\varphi(E=0)}$$

The above property makes it possible to assess REA_φ is linear time with respect to the size of the binary decision tree or the zero-suppressed binary decision diagram that encodes φ.

The rare event approximation is accurate when the probability of basic events are low. When they are not, it may give over pessimistic results (it may even exceeds 1).

To circumvent this problem, the so-called *mincut upper bound*, has been proposed from the very preliminary works on probabilistic risk assessment, e.g. in the famous WASH 1400 report (Rasmussen 1975).

Definition 13.1.4 — Mincut Upper Bound. Let $\varphi = \pi_1 + \cdots + \pi_n$ be a sum of products built over a set of variables $\mathscr{V}$. The mincut upper bound MCUB_φ of $p(\varphi)$ is defined as follows.

$$\mathrm{MCUB}_\varphi \overset{def}{=} 1 - \prod_{1 \leq i \leq n} 1 - p(\pi_i)$$

The mincut upper bound generalizes thus the formula $p(A \cup B) = 1 - p\left(\overline{A \cup B}\right) = 1 - p\left(\overline{A} \cap \overline{B}\right) = 1 - (1 - p(A)) \times (1 - p(B))$. It provides a better approximation than the rare event approximation. In particular, it never exceeds 1. It has however its own drawbacks. The main one is that it is not possible to calculate it recursively, as for the rare event approximation. Consequently, the complexity of its calculation is linear in the size of the sum of products and not in the size of the data structure encoding this sum.

In a 2010 article (Vaurio 2010), Vaurio proposed to use the Sylvester-Poincaré development in a recursive way, so to stop developing the terms when the products have a too low probability. This idea may work, but has the same drawback as the mincut upper bound regarding efficient encoding of minimal cutsets.

In addition to the two above approximations, XFTA implements a third one, so-called *pivotal upper bound*, which is original and gives in general better results that the two others.

Definition 13.1.5 — Pivotal Upper Bound. Let $\varphi = \pi_1 + \cdots + \pi_n$ be a sum of positive products built over a set of variables $\mathscr{V}$. The pivotal upper bound PUB_φ of $p(\varphi)$ is defined as follows.

If φ is reduced to a constant, then is value is simply the probability:

$$\begin{aligned} \mathrm{PUB}_0 &\overset{def}{=} 0 \\ \mathrm{PUB}_1 &\overset{def}{=} 1 \end{aligned}$$

Otherwise, let E be a variable of $\mathrm{var}(\varphi)$. Then:

$$\mathrm{PUB}_\varphi \overset{def}{=} p(E) \times P_1 + P_0 - p(E) \times P_1 \times P_0$$

where,

$$\begin{aligned} P_1 &= \mathrm{PUB}_{\varphi(E=1)} \\ P_0 &= \mathrm{PUB}_{\varphi(E=0)} \end{aligned}$$

It is easy to verify that PUB_φ is always comprised between 0 and 1 as for any two values $0 \leq x, y \leq 1$, $0 \leq max(x,y) \leq x + y - x \times y \leq 1$.

Moreover, as for the rare event approximation, its is possible to calculate PUB_φ in linear time with respect to the size of the binary decision tree or the zero-suppressed binary decision diagram that encodes φ.

⚠ **Pivotal upper bound**
The pivotal upper bound has however a drawback: its result is sensitive to the order of variables chosen to build the binary decision tree or the zero-suppressed binary decision diagram. Variations are not very big, but they are not null neither.

Table 13.1: Options of the command `compute probability`

Option	Type	Default value
`source-handle`	Identifier	None
`quantification-method`	Identifier	`pivotal-upper-bound`
`mission-time`	List descriptor	0.0
`add-singular-points`	Boolean	`false`
`smooth-curve`	Boolean	`false`
`smoothing-ratio`	Real in $]0,1[$	0.1
`print-unavailability`	Boolean	`true`
`print-mean-unavailability`	Boolean	`false`
`print-safety-integrity-level`	Boolean	`false`
`output`	String	`results.txt`
`mode`	`write`, `append`	`write`

13.2 Commands to Assess the Top Event Probability

This section presents the XFTA commands to compute the probability of the top event.

13.2.1 Main Command

The command to compute the probability of a variable of a model is `compute probability`. Its base form is as follows (assuming the variable `Top` is the variable were are interested in).

```
compute probability Top;
```

Options of the command `compute probability` are summarized in Table 13.1.
The last two options are now quite familiar:
– `output` sets the path to the output file.
– `mode` sets the output mode, i.e. either (`write` or `append`).
They can be used directly or via the corresponding environment variables. E.g.

```
compute probability Top
   output="Results/BridgePrb.tsv";
set option output "Results/BridgePrb.tsv";
set option mode append;
compute probability Top;
```

We shall review the other options of the command in turn.

13.2.1.1 Source Handle and Quantification Methods

When there is only one handle associated with the top event, the probability is assessed from this handle (see Section 9.3.5 for explanations on handles). When there are several, XFTA selects the first one in lexicographic order. The option `source-handle` selects the handle from which the probability is computed. E.g.

```
compute probability Top source-handle=BDT;
```

If the selected handle encodes a sum of disjoint products, only one quantification method can be applied. If, on the contrary, it encodes a sum of minimal cutsets, then three quantification methods are available. The option `quantification-method` (or the corresponding environment variable) selects the quantification method:

– `rare-event-approximation` or equivalently `REA`,

- `mincut-upper-bound` or equivalently `MUCB`,
- `pivotal-upper-bound` or equivalently `PUB`.

E.g.

```
compute probability Top source-handle=ZBDD
   quantification-method=mincut-upper-bound;
set option quantification-method REA;
compute probability Top source-handle=ZBDD;
```

13.2.1.2 Mission Times

The mission time(s) can be given in one of the three following forms.

- A single real value, e.g.

```
compute probability Top mission-time=8760;
```

- A list of real values surrounded with square brackets “`[`” and “`]`” and separated with commas “`,`”, e.g.

```
compute probability Top mission-time=[0, 2190, 4380, 8760];
```

- A range iterator with a minimum value, a maximum value and a step (in this order), e.g.

```
compute probability Top mission-time=range(0, 8760, 2190);
```

The above iterator produces the list `[0, 2190, 4380, 6570, 8760]`

In addition to the list of mission times defined as explained above, XFTA provides two other ways to complete the list of dates at which the indicators are calculated:

- By adding singular points.
- By smoothing the curve.

Singular points are determined by looking at probability distributions of basic events. As explained Section 13.1.1, they are typically the dates at which maintenance operations of a periodically tested component start and stop. These dates create discontinuities in the unavailability distribution and should therefore be added. E.g.

```
compute probability Top mission-time=[0, 8760]
   add-singular-points=true;
```

XFTA looks for singular points located in between the given lowest and the highest dates, here `2190` and `8760`.

Smoothing the curve is done by adding an intermediate date $t_m = (t_1 + t_2)/2$ in the middle of each pair (t_1, t_2) of successive dates. The value $Q_c = Q_S(t_m)$ is compared to the interpolated value $Q_i = (Q_S(t_1) + Q_S(t_2))/2$. If the relative difference $|Q_c - Q_i|/Q_c$ is greater than the smoothing ratio, t_m is added to the list of dates and the pairs (t_1, t_m) and (t_m, t_2) are considered in turn. To smooth the curve, the Boolean option `smooth-curve` must be turned on. The smoothing ratio is specified by means of the option `smoothing-ratio`. E.g.

```
set option mission-time range(0, 8760, 730);
compute probability Top
   add-singular-points=true
   smooth-curve=true
   smoothing-ratio=0.15;
```

13.2.1.3 Time Distributions

There are three Boolean options to define which indicators must be printed out:

- `print-unavailability` to print the unavailability (top event probability).
- `print-mean-unavailability`[1] to print the mean unavailability.
- `print-safety-integrity-level`. This option is discussed Chapter 16.

By default, the environment variables `print-unavailability` and `print-mean-unavailability` are set respectively to `true` and `false`. This is the reason why the command `compute probability` calculates the top event probability.

Assume that we calculate the top event probability at time $t_1, t_2 \ldots t_n$. Then the mean unavailability at time t_k is calculated as the empirical mean of the first k values, i.e.

$$Q_S^{avg}(t_k) \quad = \quad \frac{1}{k} \times \sum_{i=1}^{k} Q_S(t_i)$$

To get significant values of the mean unavailability, it is thus recommended to calculate the unavailability at sufficiently many mission times. E.g.

```
set option output "Results/Bridge03.tsv";
set option mode write;
set option mission-time range(0, 8760, 730);
compute probability Top
   add-singular-points=true
   smooth-curve=true
   print-unavailability=false
   print-mean-unavailability=true;
```

Computations of time distributions

Computations of time distributions make sense only if at least two mission times are given, e.g.

```
set option mission-time [0, 8760];
```

This applies in particular to options `print-mean-unavailability` and `smooth-curve`. The current version of XFTA *does not* include automatically the mission time 0 in the list of mission times.

13.2.1.4 Complexity of Calculations

The complexity in time of the command `compute probability` depends on the quantification method and the number m of mission times at which the probability is calculated. It is in:

- $\mathcal{O}(m \times |\varphi|)$ if φ is a sum of disjoint products (encoded by a binary decision diagram);
- $\mathcal{O}(m \times |\varphi|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the rare event approximation or the pivotal upper bound;
- $\mathcal{O}(m \times |\mathrm{SoP}(\varphi)|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the mincut upper bound.

13.2.2 Auxiliary Commands

XFTA provides two commands in addition to `compute probability`. These two commands are mostly used for verification purposes.

[1]This option was called `print-average-unavailability` in previous versions of XFTA

Table 13.2: Options of the command `compute time-to-probability`

Option	Type	Default value
`source-handle`	Identifier	None
`quantification-method`	Identifier	`pivotal-upper-bound`
`probability-margin`	Floating point number	0.1
`time-lower-bound`	Floating point number	0
`time-upper-bound`	Floating point number	8760
`time-grain`	Floating point number	1
`output`	String	`results.txt`
`mode`	`write`, `append`	`write`

13.2.2.1 Command `compute values-of-parameters`

The command `compute values-of-parameters` calculates the values of parameters at a given mission time. This command takes the top event as second argument. It collects recursively all parameters involved in the definition of that top event and calculates their values. E.g.

```
set option output "Results/Bridge04.tsv";
set option mode write;
set option mission-time 8760;
compute values-of-parameters Top;
```

Note that, in general, values of parameters do not depend on the mission time.

The complexity in time of the command `compute values-of-parameters` is nearly linear in the number of parameters involved in the structure function of the given top event.

13.2.2.2 Command `compute probabilities-of-basic-events`

The command `compute probabilities-of-basic-events` calculates the probabilities of basic events at a given mission time. This command takes the top event as second argument. It collects recursively all basic events involved in the definition of that top event and calculates their probabilities. E.g.

```
set option output "Results/Bridge04.tsv";
set option mode append;
set option mission-time 8760;
compute probabilities-of-basic-events Top;
```

The complexity in time of the command `compute probabilities-of-basic-events` is nearly linear in the number of parameters involved in the structure function of the given top event.

13.2.2.3 Command `compute time-to-probability`

The command `compute time-to-probability` seeks for the mission time at which the probability of a given variable reaches a given value. Technically, this command implements a dichotomic search between a lower and an upper bound. Its base form takes two arguments (in addition to the first one `time-to-probability`): the name of the target variable and the target value for the probability. E.g.

```
compute time-to-probability Top 1.0e-3;
```

Options of the command `compute time-to-probability` are summarized in Table 13.2.

The following options are familiar:

– `source-handle` sets the data structure from which the probabilities are computed. By default, XFTA selects the first available handle in lexicographic order.
– `quantification-method` sets the quantification method (if the data structure encodes a sum of minimal cutsets).
– `output` sets the path to the output file.
– `mode` sets the output mode, i.e. either (`write` or `append`).

The command looks actually a mission time t, $t_{min} \leq t \leq t_{max}$, such that:

$$p_{obj} - \varepsilon \times p_{obj} \quad \leq \quad Q_S(t) \quad \leq \quad p_{obj} + \varepsilon \times p_{obj}$$

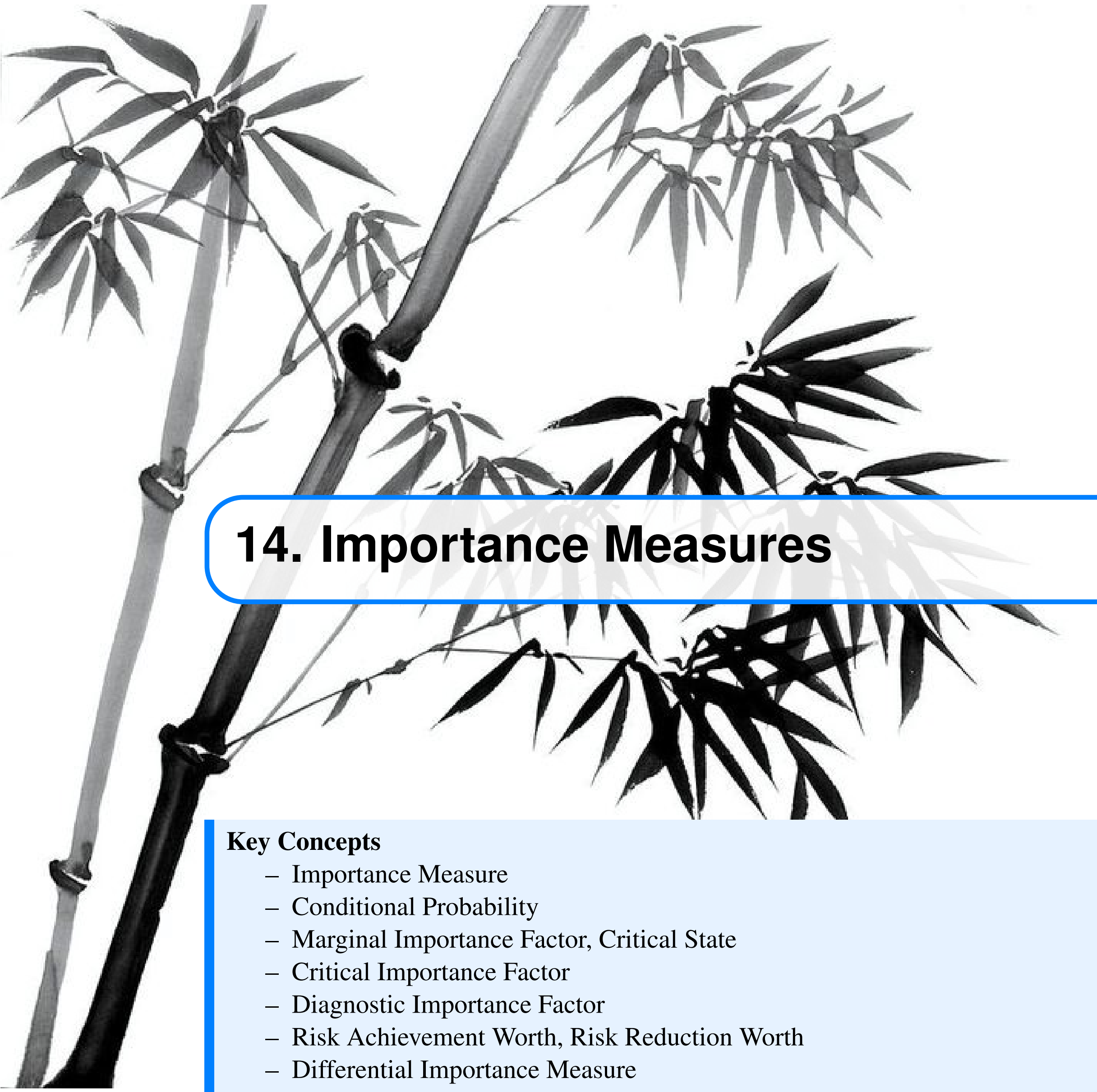
where:
– p_{obj} is the objective probability, which is given as third argument of the command.
– S is the target variable, which is given as second argument of the command.
– t_{min} is the lower bound of the mission time, given via the option `time-lower-bound`.
– t_{max} is the lower bound of the mission time, given via the option `time-upper-bound`.
– ε is the probability margin, given via the option `probability-margin`.

To avoid to perform too many recursive calls, a time grain τ is set by means of the option `time-grain`. If at certain a certain point, the algorithm looks for the target mission time t between two bounds t_{low} and t_{high} such that $t_{high} - t_{low} < \tau$, then the recursion is stopped and $t = \frac{t_{high}+t_{low}}{2}$ is returned even though $Q_S(t)$ does not belong to the target interval.

Command `compute time-to-probability`
The command `compute time-to-probability` assumes that $Q_S(t)$ is monotonically increasing in the time interval $[t_{min}, t_{max}$. No check that this condition is fulfilled is performed.

Computational Complexity

The complexity in time of the command `compute time-to-probability` depends on the quantification method and the recursive calls of the algorithm. It is in:
– $\mathcal{O}\left(\frac{\log_2(t_{max}-t_{min})}{\varepsilon} \times |\varphi|\right)$ if φ is a sum of disjoint products (encoded by a binary decision diagram);
– $\mathcal{O}\left(\frac{\log_2(t_{max}-t_{min})}{\varepsilon} \times |\varphi|\right)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the rare event approximation or the pivotal upper bound;
– $\mathcal{O}\left(\frac{\log_2(t_{max}-t_{min})}{\varepsilon} \times |\mathrm{SoP}(\varphi)|\right)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the mincut upper bound.



14. Importance Measures

Key Concepts
- Importance Measure
- Conditional Probability
- Marginal Importance Factor, Critical State
- Critical Importance Factor
- Diagnostic Importance Factor
- Risk Achievement Worth, Risk Reduction Worth
- Differential Importance Measure
- Barlow-Proschan Factor

Importance measures are probabilistic indicators calculated for individual basic events or groups of basic events of a fault tree. These indicators aim at assessing the relative contributions of the different components of the system to the overall risk. Throughout this chapter, we assume that the top event T of the fault tree represents the failure of the system under study and that each basic event E denotes the failure of a component of this system.

An importance measure is therefore an indicator $\mathrm{IM}_{S,E}$. The reliability engineering literature distinguishes five main importance measures, see e.g. (Kumamoto and Henley 1996):

- The marginal importance factor $\mathrm{MIF}_{T,E}$ often called Birnbaum importance factor.
- The critical importance factor $\mathrm{CIF}_{T,E}$.
- The diagnostic importance factor $\mathrm{DIF}_{T,E}$ also called Fussel-Vesely importance factor.
- The risk achievement worth $\mathrm{RAW}_{T,E}$ also called risk increase factor.
- The risk reduction worth $\mathrm{RRW}_{T,E}$ also called risk decrease factor.

In addition, two other importance measures are of interest.

- The differential importance measure $\mathrm{DIM}_{T,E}$.
- The Barlow-Proschan factor $\mathrm{BPF}_{T,E}$.

Theses indicators are strongly related one another. Therefore, XFTA provides a unique command to compute all of them. In the current version of XFTA, importance measures are systematically calculated for all basic events.

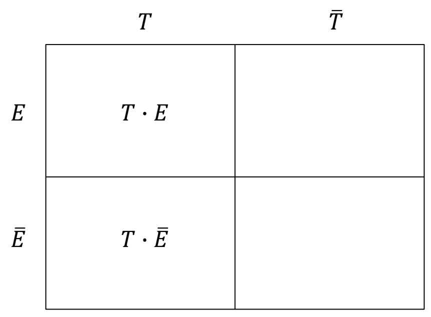


Figure 14.1: Sets of minterms characterized by $p(T \mid E)$ and $p\left(T \mid \overline{E}\right)$

It is also possible to calculate importance measures for groups of components.

For the sake of simplicity, all definitions of this chapter are given without referring to the time t at which the indicators are calculated. However, t must be implicitly understood, i.e. $\text{MIF}_{T,E}$ must be read $\text{MIF}_{T,E}(t)$, $\text{CIF}_{T,E}$ must be read $\text{CIF}_{T,E}(t)$ and so on.

14.1 Mathematical and Algorithmic Framework

14.1.1 Conditional Probabilities

14.1.1.1 Definition

Let T be the top event of the model and E be one of its basic events. Technically speaking, all importance measures for E are calculated from the probability $p(T)$ of T and the conditional probabilities $p(T \mid E)$ and $p\left(T \mid \overline{E}\right)$.

To understand importance measures, it is worth to look at which sets of minterms they characterize (Dutuit and Rauzy 2013). Figure 14.1 shows the sets of minterms characterized by $p(T \mid E)$ and $p(T \mid \overline{E})$, namely those of functions $T \cdot E$ and $T \cdot \overline{E}$.

This characterization relies on the well known property of conditional probability:

Property 14.1 — Conditional Probability. Let f and g be two events over the same probability space. Then, the following equality holds.

$$p(f \cdot g) \quad = \quad p(f \mid g) \times p(g)$$

Applied to T and E, we have thus:

$$\begin{aligned} p(T \mid E) &= \frac{p(T \cdot E)}{p(E)} \\ p\left(T \mid \overline{E}\right) &= \frac{p(T \cdot E)}{p\left(\overline{E}\right)} \end{aligned}$$

Hence the graphical representation of Figure 14.1.

14.1.1.2 Calculation

Depending on which normal form, quantification measure and data structure are used to encode the structure function of T, there are different ways of assessing $p(T \mid E)$ and $p\left(T \mid \overline{E}\right)$. However, the simplest is to perform the calculation of $p(T)$ in which the probability of E has been set to respectively 1 and 0.

14.1.2 Marginal Importance Factor

14.1.2.1 Definition

The marginal importance factor is also called Birnbaum importance factor in the reliability engineering literature, as it has been originally proposed by Zygmund Birnbaum (Birnbaum 1969).

Definition 14.1.1 — Marginal Importance Factor. Let T be the top event of a model and E be a basic event of that model. Then, the *marginal importance factor* of E in T, denoted by $\mathrm{MIF}_{T,E}$, is defined as follows.

$$\mathrm{MIF}_{T,E} \quad \overset{def}{=} \quad \frac{\partial p(T)}{\partial p(E)}$$

Using the Shannon decomposition (property 13.1), we can decompose $p(T)$ as follows.

$$p(T) \quad = \quad p(E) \times p(T \mid E) + (1 - p(E)) \times p\left(T \mid \overline{E}\right) \tag{14.1}$$
$$= \quad p(E) \times \left[p(T \mid E) - p\left(T \mid \overline{E}\right)\right] + p\left(T \mid \overline{E}\right) \tag{14.2}$$

The following property follows from equality 14.2 and the calculation of derivatives of linear functions.

Property 14.2 — Marginal Importance Factor. Let T be the top event of a model and E be a basic event of that model. Then,

$$\mathrm{MIF}_{T,E} \quad = \quad p(T \mid E) - p\left(T \mid \overline{E}\right)$$

By definition, $\mathrm{MIF}_{T,E}$ can be interpreted as the effect of a small variation of the probability of E on the probability of T, *mutatis mutandis*.

By property 14.2, it can be interpreted as the probability to be in a critical state. Formally:

Definition 14.1.2 — Critical States. Let T be the top event of a model and E be a basic event of that model. Then the minterm $E \cdot \sigma$ is *critical* if the following conditions hold.
- $E \cdot \sigma \models T$,
- $\overline{E} \cdot \sigma \models \overline{T}$.

The set (sum) of critical states of T with respect to E is denoted $\mathrm{CriticalStates}_{T,E}$.

The following property holds.

Property 14.3 — Critical States. Let T be the top event of a coherent model and E be a basic event of that model. Then,

$$p\left(\mathrm{CriticalStates}_{T,E}\right) \quad = \quad p(E) \times \mathrm{MIF}_{T,E}$$

The set of minterms satisfying T can be split into four subsets:
- The subset C_1 of minterms $E \cdot \sigma$ such that $\overline{E} \cdot \sigma \models \overline{T}$,
- The subset NC_1 of minterms $E \cdot \tau$ such that $\overline{E} \cdot \tau \models T$,
- The subset C_0 of minterms $\overline{E} \cdot \sigma$ such that $E \cdot \sigma \models \overline{T}$,
- The subset NC_0 of minterms $\overline{E} \cdot \tau$ such that $E \cdot \tau \models T$.

By definition $C_1 = \mathrm{CriticalStates}_{T,E}$.
If the model is coherent, we have $C_0 = \emptyset$. Consequently,

$$p(NC_0) \quad = \quad p\left(T \cdot \overline{E}\right) \;=\; (1 - p(E)) \times p\left(T \mid \overline{E}\right)$$

Minterms of NC_1 one-to-one corresponds with minterms of NC_0. Consequently,

$$p(NC_1) \quad = \quad p(E) \times p\left(T \mid \overline{E}\right)$$

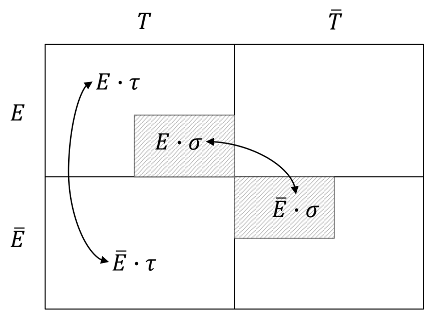


Figure 14.2: Graphical illustration of $\text{CriticalStates}_{T,E}$

These three facts are illustrated in Figure 14.2. The set $\text{CriticalStates}_{T,E}$ is shaded.
We have thus:

$$\begin{aligned}
p\left(\text{CriticalStates}_{T,E}\right) &= p\left(T\right) - p\left(NC_1\right) - p\left(NC_0\right) \\
&= p\left(E\right) \times p\left(T \mid E\right) + \left(1 - p\left(E\right)\right) \times p\left(T \mid E\right) \\
&\quad - p\left(E\right) \times p\left(T \mid \overline{E}\right) + \left(1 - p\left(E\right)\right) \times p\left(T \mid \overline{E}\right) \\
&= p\left(E\right) \times \left(p\left(T \mid E\right) - p\left(T \mid \overline{E}\right)\right) \\
&= p\left(E\right) \times \text{MIF}_{T,E}
\end{aligned}$$

QED

14.1.2.2 Calculation

As for conditional probabilities, there are several ways of calculating $\text{MIF}_{T,E}$, depending on which normal form, quantification measure and data structure are used to encode the structure function of T.

A simple way consists in applying property 14.2, in calculating $\text{MIF}_{T,E}$ as $p\left(T \mid E\right) - p\left(T \mid \overline{E}\right)$. This is the one used in the current version of XFTA for the calculation of importance measures.

Another way consists in applying the definition of $\text{MIF}_{T,E}$, i.e. in calculating $\text{MIF}_{T,E}$ by numerical differentiation.

A third and direct algorithm can be designed to calculate $\text{MIF}_{T,E}$ from the binary decision diagram that encodes T (Dutuit and Rauzy 2013).

The mathematical developments of this section assume that the model is coherent. In case the model is non-coherent, $\text{MIF}_{T,E}$ can be negative and the the interpretation in terms of critical states does not hold anymore. To cope with this problem, alternative definitions of the marginal importance factor have been proposed, see e.g. (Beeson and Andrews 2003). They are however not implemented in the current version of XFTA.

14.1.3 Critical Importance Factor

14.1.3.1 Definition

As recalled at the beginning of this chapter, one of the main goals of importance measures is to rank components of the system under study according to their contribution to the risk. With that respect, the marginal importance factor has an important drawback: it does not take into account the probability of the basic event. $\text{MIF}_{T,E}$ stays the same, whether $p\left(E\right)$ is low or high.

The critical importance factor, introduced by Lambert (Lambert 1975), aims at circumventing this problem.

Definition 14.1.3 — Critical Importance Factor. Let T be the top event of a model and E be a basic event of that model. Then, the *critical importance factor* of E in T, denoted by $\text{CIF}_{T,E}$, is defined as follows.

$$\mathrm{CIF}_{T,E} \overset{def}{=} \frac{p(E) \times \mathrm{MIF}_{T,E}}{p(T)}$$

It is clear that $\mathrm{CIF}_{T,E}$ depends on $p(E)$.

Note that, with respect to the ranking of components, the denominator $p(T)$ does not play any role, as it is the same for all components. Rather, it should be seen as a normalization factor making it possible to compare results from models to models.

Except for that, $\mathrm{CIF}_{T,E}$ is simply the probability of critical states with respect to E.

$$\mathrm{CIF}_{T,E} = p(\mathrm{CriticalStates}_{T,E})$$

$\mathrm{CIF}_{T,E}$ characterizes thus the same minterms as $\mathrm{MIF}_{T,E}$.

14.1.3.2 Calculation

The calculation of $\mathrm{CIF}_{T,E}$ is performed according to its definition.

14.1.4 Diagnostic Importance Factor

14.1.4.1 Definition

The diagnostic importance factor does not attempt to measure the criticality of components but rather to determine which component should be looked at first when the system is failed. This notion is of interest in harsh environments where sending a robot, or even worse a human operator, may present serious difficulties. So, the faster one finds the problem the better.

Definition 14.1.4 — Diagnostic Importance Factor. Let T be the top event of a model and E be a basic event of that model. Then, the *diagnostic importance factor* of E in T, denoted by $\mathrm{DIF}_{T,E}$, is defined as follows.

$$\mathrm{DIF}_{T,E} \overset{def}{=} p(E \mid T)$$

The diagnostic importance factor has been introduced by Fussel (Fussel 1975).

Applying property 14.1, we can rewrite $\mathrm{DIF}_{T,E}$ as follows.

$$\mathrm{DIF}_{T,E} = \frac{p(E \cdot T)}{p(S)} \tag{14.3}$$

$$= \frac{p(E) \times p(T \mid E)}{p(S)} \tag{14.4}$$

The above equality means that, except for the normalization factor $p(S)$, $\mathrm{DIF}_{T,E}$ measures eventually the probability of $T \cdot E$, i.e. of the set of minterms represented by the upper left rectangle in Figure 14.1.

14.1.4.2 Calculation

In XFTA, $\mathrm{DIF}_{T,E}$ is calculated using equation 14.4.

The diagnostic importance factor is often called the Fussel-Vesely importance factor after its authors. In some textbooks, however, it is defined as follows.

$$\mathrm{FVF}_{T,E} \overset{def}{=} \frac{p\left(\bigvee_{E\cdot\pi \in \mathrm{MCS}(T)} E \cdot \pi\right)}{p\left(\bigvee_{\pi \in \mathrm{MCS}(T)} \pi\right)} \tag{14.5}$$

The reason is that, if we consider the rare event approximation, then:

$$\begin{aligned} p(E\cdot T) &\approx \sum_{E\cdot\pi\in\text{MCS}(T)} p(\pi) \\ p(T) &\approx \sum_{\pi\in\text{MCS}(T)} p(\pi) \end{aligned}$$

There is however a fundamental flaw in definition 14.5, as illustrated by the following example (taken from (Dutuit and Rauzy 2013)).

■ **Example 14.1** Let $T = A\cdot B\cdot C + A\cdot B\cdot D + C\cdot D$. Then:

$$\begin{aligned} \text{CriticalStates}_{T,A} &= A\cdot B\cdot C\cdot\overline{D} + A\cdot B\cdot\overline{C}\cdot D \\ A\cdot T &= A\cdot B\cdot C\cdot D + A\cdot B\cdot C\cdot\overline{D} + A\cdot B\cdot\overline{C}\cdot D + A\cdot\overline{B}\cdot C\cdot D \\ \bigvee_{A\cdot\pi\in\text{MCS}(T)} A\cdot\pi &= A\cdot B\cdot C\cdot D + A\cdot B\cdot C\cdot\overline{D} + A\cdot B\cdot\overline{C}\cdot D \end{aligned}$$

■

In other words, $\text{CIF}_{T,E} < \text{FVF}_{T,E} < \text{DIF}_{T,E}$.

The problem is that minimal cutsets have no direct logical status. Consequently, considering the disjunction of a subset of minimal cutsets leads to ackward definition, messing up the mathematical definition and the calculation method.

14.1.5 Risk Achievement Worth and Risk Reduction Worth

14.1.5.1 Definition

The two other main importance factors, widely used in nuclear probabilistic safety analyses, are the risk achievement worth and the risk reduction worth, also called respectively *risk increase factor* and *risk decrease factor* (Berg 1994).

Definition 14.1.5 — Risk Achievement Worth and Risk Reduction Worth. Let T be the top event of a model and E be a basic event of that model. Then, the *risk achievement worth* and the *risk reduction worth* of E in T, denoted respectively by $\text{RAW}_{T,E}$ and $\text{RRW}_{T,E}$, are defined as follows.

$$\begin{aligned} \text{RAW}_{T,E} &\overset{def}{=} \frac{p(T \mid E)}{p(T)} \\ \text{RRW}_{T,E} &\overset{def}{=} \frac{p\left(T \mid \overline{E}\right)}{p(T)} \end{aligned}$$

$\text{RAW}_{T,E}$ measures the increase in system failure probability assuming that the component is failed. It is an indicator of the importance of maintaining the current level of reliability for the component (Cheok, Parry, and Sherry 1998). In reference (Wall and Worledge 1996), it is argued that $\text{RAW}_{T,E}$ should be used with care, for it is rather rough. As $\text{DIF}_{T,E}$, $\text{RAW}_{T,E}$ measures eventually the probability of $T\cdot E$, i.e. of the set of minterms represented by the upper left rectangle in Figure 14.1.

$\text{RRW}_{T,E}$ represents the maximum decreasing of the risk it may be expected by increasing the reliability of the component. Consequently, this quantity may be used to select components that are the best candidates for efforts leading to improving system reliability. Note that $\text{RRW}_{T,E}$ is sometimes defined as $\frac{p(T)}{p\left(T\mid\overline{E}\right)}$, i.e. the inverse of the above definition, e.g. in RiskSpectrum (Berg 1994). Taking one definition or the other does change anything but the presentation of the results. $\text{RRW}_{T,E}$ measures eventually the probability of $T\cdot\overline{E}$, i.e. of the set of minterms represented by the lower left rectangle in Figure 14.1.

Note that, as for $\text{CIF}_{T,E}$ and $\text{DIF}_{T,E}$, the numerator $p(T)$ can be seen as a normalization factor.

14.1.5.2 Calculation

The calculations of $\mathrm{RAW}_{T,E}$ and $\mathrm{RRW}_{T,E}$ are performed according to their definition.

14.1.6 Differential Importance Measure

14.1.6.1 Definition

The differential importance measure has been introduced by Borgonovo and Apostolakis (Borgonovo and Apostolakis 2001). According to its authors, this indicator responds to the need of the analyst to get information about the importance of proposed changes that affect component properties and multiple basic events. Unlike the previous indicators, this indicator is actually additive by construction, i.e. that the differential importance measure of a group of basic events is simply the sum of the differential importance measures of individual basic events of the group.

Definition 14.1.6 — Differential Importance Measure. Let T be the top event of a model built over a set of basic events $\mathscr{V}$ and E be a basic event of $\mathscr{V}$. Then, the *differential importance measure* of E in T, denoted by $\mathrm{DIM}_{T,E}$, is defined as follows.

$$\mathrm{DIM}_{T,E} \overset{def}{=} \frac{\mathrm{MIF}_{T,E}}{\sum_{E\in\mathscr{V}} \mathrm{MIF}_{T,E}}$$

If $\mathscr{G}$ is a subset of $\mathscr{V}$, then:

$$\mathrm{DIM}_{T,\mathscr{G}} \overset{def}{=} \frac{\sum_{E\in\mathscr{G}} \mathrm{MIF}_{T,E}}{\sum_{E\in\mathscr{V}} \mathrm{MIF}_{T,E}}$$

14.1.6.2 Calculation

The calculation of $\mathrm{DIM}_{T,E}$ (and $\mathrm{DIM}_{T,\mathscr{G}}$) is performed according to its definition.

14.1.7 Barlow-Proschan Factor

14.1.7.1 Definition

The Barlow-Proschan factor has been introduced by these two authors in 1975 (Barlow and Proschan 1975). Unlike the other indicators presented in this chapter, it addresses the (un)reliability, or more exactly the failure intensity, of the system rather than its (un)availability. The mathematical framework for this indicator will be discussed in Chapter 16. Consequently, we give here only the definition of the Barlow-Proschan factor, without further explanations.

Definition 14.1.7 — Barlow-Proschan Factor. Let T be the top event of a model built over a set of basic events $\mathscr{V}$ and E be a basic event of $\mathscr{V}$. Then, the *Barlow-Proschan factor* of E in T, denoted by $\mathrm{BPF}_{T,E}$, is defined as follows.

$$\mathrm{BPF}_{T,E} \overset{def}{=} \frac{\omega_E \times \mathrm{MIF}_{T,E}}{\sum_{E\in\mathscr{V}} \omega_E \times \mathrm{MIF}_{T,E}}$$

where ω_E denotes the *unconditional failure intensity* of the component E.

14.1.7.2 Calculation

The calculation of $\mathrm{BPF}_{T,E}$ is performed according to its definition.

14.1.8 Importance Measures for Groups of Basic Events

14.1.8.1 Definition

Some software, e.g. RiskSpectrum®, provides some (limited) features to calculate importance measures of groups of basic events. The idea is to be able to assess the importance of groups of similar components, e.g. pumps, valves... However, extending importance measures discussed in this section to groups of basic events (rather than to individual events) is far from trivial, see e.g. (Dutuit and Rauzy 2015) for an in-depth discussion. The key issue stands in the definition of the probability of occurrence of the group. As there is no real satisfying answer to this issue, one must make quite arbritray choices. The current version of XFTA makes the following ones.

Definition 14.1.8 — Importance Measures of Groups of Components. Let T be the top event of a model built of over a set of variables $\mathscr{V}$ and let $\mathscr{G} = \{E_1, \ldots E_k\}$ be a subset of $\mathscr{V}$. Then, the definitions of importance measures are extended as follows.

$$\begin{aligned}
p(\mathscr{G}) &\overset{def}{=} \prod_{E \in \mathscr{G}} p(E) \\
p(T \mid \mathscr{G}) &\overset{def}{=} p(T \mid E_1, \ldots E_k) \\
p\left(T \mid \overline{\mathscr{G}}\right) &\overset{def}{=} p\left(T \mid \overline{E_1}, \ldots \overline{E_k}\right) \\
\mathrm{MIF}_{T,\mathscr{G}} &\overset{def}{=} p(T \mid \mathscr{G}) - p\left(T \mid \overline{\mathscr{G}}\right) \\
\mathrm{CIF}_{T,\mathscr{G}} &\overset{def}{=} \frac{p(\mathscr{G}) \times \mathrm{MIF}_{T,\mathscr{G}}}{p(T)} \\
\mathrm{DIF}_{T,\mathscr{G}} &\overset{def}{=} \frac{p(\mathscr{G}) \times p(T \mid \mathscr{G})}{p(T)} \\
\mathrm{RAW}_{T,\mathscr{G}} &\overset{def}{=} \frac{p(T \mid \mathscr{G})}{p(T)} \\
\mathrm{RRW}_{T,\mathscr{G}} &\overset{def}{=} \frac{p\left(T \mid \overline{\mathscr{G}}\right)}{p(T)} \\
\mathrm{DIM}_{T,\mathscr{G}} &\overset{def}{=} \frac{\sum_{E \in \mathscr{G}} \mathrm{MIF}_{T,E}}{\sum_{E \in \mathscr{V}} \mathrm{MIF}_{T,E}} \\
\mathrm{BPF}_{T,\mathscr{G}} &\overset{def}{=} \frac{\sum_{E \in \mathscr{G}} \omega_E \times \mathrm{MIF}_{T,E}}{\sum_{E \in \mathscr{V}} \omega_E \times \mathrm{MIF}_{T,E}}
\end{aligned}$$

The above definitions takes only into account the extreme cases where either all components of the group are working, or all of them are failed, but none of the intermediate situations. This reflects in the definition of conditional probabilities $p(T \mid \mathscr{G})$ and $p\left(T \mid \overline{\mathscr{G}}\right)$. $p(T \mid \mathscr{G})$ represents the case where all components of the group are failed, while $p\left(T \mid \overline{\mathscr{G}}\right)$ represents the case where all components of the group are working.

14.1.8.2 Calculation

Calculations of all indicators are performed according to their definitions. Results depend indeed on the normal form and the quantification method that are used.

14.2 Commands to Assess Important Measures

14.2.1 Main Command

The main command to compute importance measures of basic events involved in the structure function of a top event is `compute importance-measures`. In its base form, it is a follows (assuming the variable `Top` is the top event of the model).

```
compute importance-measures Top;
```

Options of the command `compute importance-measures` are summarized in Table 14.1. Several of these options have been presented Section 13.2.1.

Table 14.1: Options of the command `compute importance-measures`

Option	Type	Default value
`source-handle`	Identifier	None
`quantification-method`	Identifier	`PUB`
`mission-time`	List descriptor	0.0
`print-probability`	Boolean	`true`
`print-conditional-probability-1`	Boolean	`false`
`print-conditional-probability-0`	Boolean	`false`
`print-marginal-importance-factor`	Boolean	`true`
`print-critical-importance-factor`	Boolean	`true`
`print-diagnostic-importance-factor`	Boolean	`true`
`print-risk-achievement-worth`	Boolean	`true`
`print-risk-reduction-worth`	Boolean	`true`
`print-differential-importance-measure`	Boolean	`false`
`print-Barlow-Proschan-factor`	Boolean	`false`
`output`	String	`results.txt`
`mode`	`write`, `append`	`write`

– `output` sets the path to the output file.
– `mode` sets the output mode, i.e. either (`write` or `append`).
– `source-handle` sets the handle (BDT, BDD or ZBDD) from which the quantification is performed.
– `quantification-method` sets the quantification method in case the handle encodes a sum of minimal cutsets.

We shall now review the other options of the command in turn.

14.2.1.1 Mission Times

The mission time(s) can be given in one of the three following forms.

– A single real value, e.g.

```
compute importance-measures Top mission-time=8760;
```

– A list of real values surrounded with square brackets "`[`" and "`]`" and separated with commas "`,`", e.g.

```
compute importance-measures Top mission-time=[0, 2190, 4380, 8760];
```

– A range iterator with a minimum value, a maximum value and a step (in this order), e.g.

```
compute importance-measures Top mission-time=range(0, 8760, 2190);
```

The above iterator produces the list `[0, 2190, 4380, 6570, 8760]`

14.2.1.2 Importance Measures

As discussed in the previous section, importance measures are essentially computed from conditional probabilities. Consequently, they can be printed out or not at will, once conditional probabilities have been computed. This is the role of `print-`*`importance-measure`* options, with their obvious meaning.

Assume for instance we want to print out $\mathrm{RAW}_{T,E}$ and $\mathrm{RRW}_{T,E}$ and only these two indicators. This can be done as follows.

```
set option print-probability false;
set option print-conditional-probability-1 false;
set option print-conditional-probability-0 false;
set option print-marginal-importance-factor false;
set option print-critical-importance-factor false;
set option print-diagnostic-importance-factor false;
set option print-risk-achievement-worth true;
set option print-risk-reduction-worth true;
set option print-differential-importance-measure false;
set option print-Barlow-Proschan-factor false;
compute importance-measures Top
   source-handle=BDT
   quantification-method=mincut-upper-bound;
```

14.2.2 Command to Compute Importance Measures of Group of Components

The command to compute importance measures of a group of basic events is `compute group-importance-measures`. Assuming that the top event is `Top` and the group is made of basic events `E1`, `E2` and `E3`, its base form is as follows.

```
compute group-importance-measures Top [E1, E2, E3];
```

The list of basic events is thus surrounded with square brackets "`[`" and "`]`". Identifiers of basic events are separated with commas '`,`".

Options of the command `compute group-importance-measures` are the same as those of the command `compute importance-measures`, i.e. those explained in the previous section and summarized in Table 14.1.

14.2.3 Complexity of Calculations

If the number of basic events is n, the number of mission times at which the importance measures are computed is m, and the sum of products from which they are computed is φ, then the complexity of the command `compute importance-measures` is in:

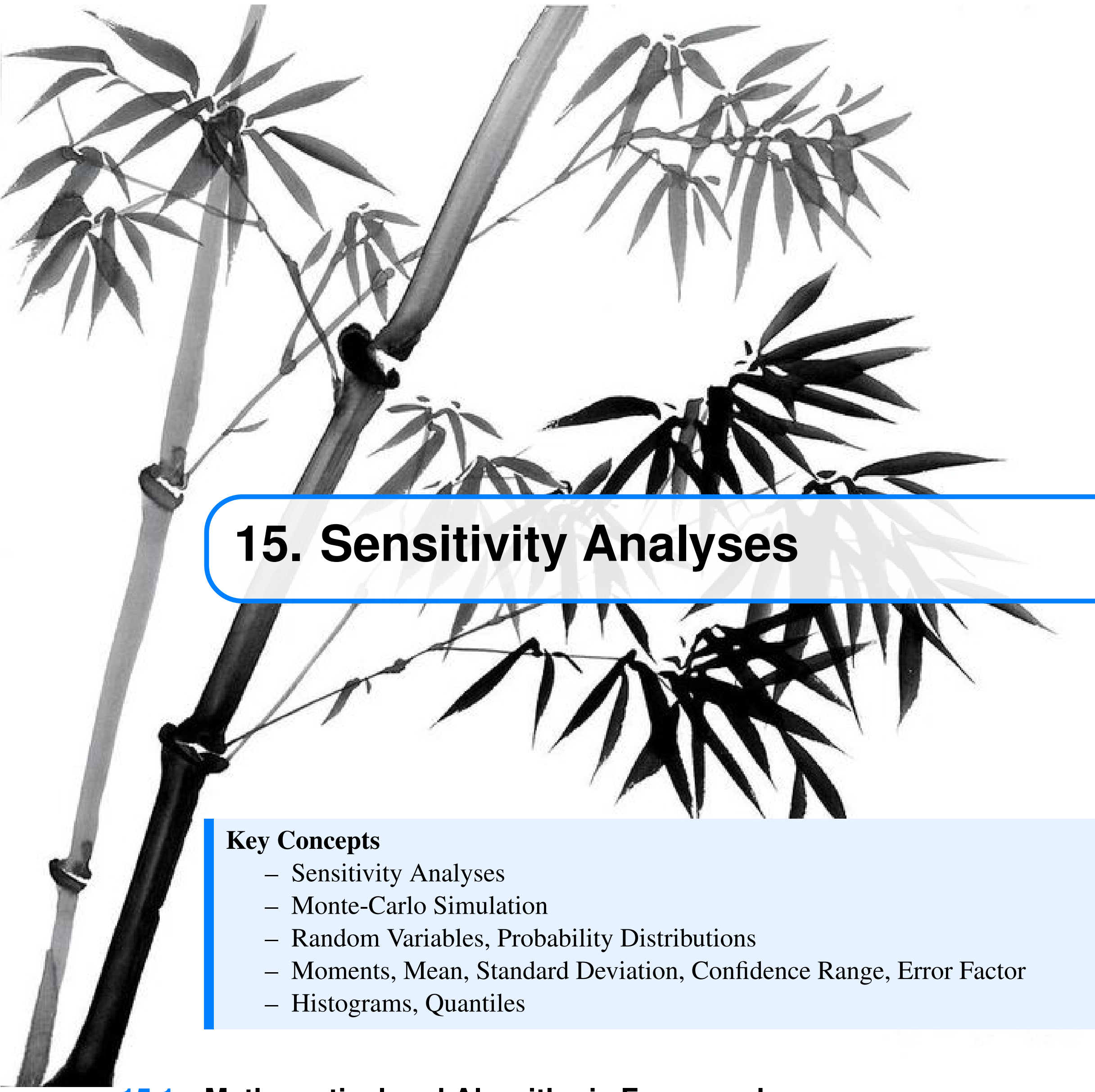
- $\mathcal{O}(n \times m \times |\varphi|)$ if φ is a sum of disjoint products (encoded by a binary decision diagram);
- $\mathcal{O}(n \times m \times |\varphi|)$ if φ is a sum of minimal cutstes (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the rare event approximation or the pivotal upper bound;
- $\mathcal{O}(n \times m \times |\mathrm{SoP}(\varphi)|)$ if φ is a sum of minimal cutstes (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the mincut upper bound.

If the number of basic events in the group is n, the number of mission times at which the importance measures are computed is m, and the sum of products from which they are computed is φ, then the complexity of the command `compute group-importance-measures` is in:

- $\mathcal{O}(n \times m \times |\varphi|)$ if φ is a sum of disjoint products (encoded by a binary decision diagram);
- $\mathcal{O}(n \times m \times |\varphi|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the rare event approximation or the pivotal upper bound;
- $\mathcal{O}(n \times m \times |\mathrm{SoP}(\varphi)|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the mincut upper bound.

In case the differential importance measure or the Barlow-Proschan factor are calculated, n be be taken as the number of basic events involved in the structure function of the top event rather than as the number of basic events in the group.

⚠ **Mission times and importance measures**
Although XFTA makes it possible to calculate importance measures at different mission times, this functionality must be used with much care as it may involve enormous amount of calculations. This is the reason why options `add-singular-point` and `smooth-curve` that are available for the assessment of the probability of the top event are not available for the assessment of importance measures.



15. Sensitivity Analyses

Key Concepts
- Sensitivity Analyses
- Monte-Carlo Simulation
- Random Variables, Probability Distributions
- Moments, Mean, Standard Deviation, Confidence Range, Error Factor
- Histograms, Quantiles

15.1 Mathematical and Algorithmic Framework

15.1.1 Monte-Carlo Simulation

Reliability parameters are often known only up to an uncertainty. Therefore, it is of interest to study the sensitivity of the probabilistic risk indicators to (small) variations of values of reliability parameters reliability parameters. Sensitivity analyses are performed by means of Monte-Carlo simulations.

In this context, a *Monte-Carlo simulation* consists in repeating the following operations, a sufficient number of times:

- First, the value of parameters is drawn according to their probability distribution (itself defined through random deviates).
- Second, the probability of basic events is calculated.
- Third, the probability of the top event is calculated and stored (or at least accumulated) for future statistical treatment.

Each iteration is called a *try*.

At the end of the simulation, statistics are performed on the computed values. Moments (mean, standard deviation, confidence range and error factor) of the distribution of the probability of the top event are the calculated and printed out. It is also possible to print out quantiles and a histogram of this distribution. Quantiles are computed on the fly. Therefore, their values are approximated, although in general very close to actual values. The same remark holds for histograms. Finally, it is also possible to print out moments of basic events and parameters distributions.

Monte-Carlo simulation are calculation resource intensive. This is the reason why XFTA implements sensitivity analyses only for the top event probability and only for a given mission time.

15.1.2 Statistics

15.1.2.1 Moments

In statistics, a *moment* is a specific quantitative measure of the shape of a function. Formally,

Definition 15.1.1 — n-th Moment. The n-th *moment* μ_n of a real-valued continuous function f of a real variable about a value c is defined as follows.

$$\mu_n \overset{def}{=} \int_{-\infty}^{\infty} (x-c)^n f(x)\, dx$$

The moment of a function, without further details, refers usually to the above expression with $c = 0$. The mathematical concept of moment is closely related to the concept of moment in physics. Note that the n-moment does not necessarily exist (because the above integral may be infinite).

If the function is a probability distribution, then the first moment is called the *mean*, the second moment is called the *variance*, the third moment is called the *skewness*, and the fourth moment is called the *kurtosis*. The *standard-deviation* is the square root of the variance.

Definition 15.1.2 — Mean. Let X be a random variable with a probability density function of $f(x)$, The *mean* of X, also called its *(mathematical) expectation* and denoted $E[X]$, is defined as the following Lebesgue integral.

$$E[X] \overset{def}{=} \int_{\mathbb{R}} x f(x)\, dx$$

The mean of a random variable X is also denoted $\mu(X)$.

Definition 15.1.3 — Variance. Let X be a random variable. The *variance* of X, denoted $\text{var}(X)$, is the expectation of the squared deviation of X from its mean $E[X]$:

$$\text{var}(X) \overset{def}{=} E\left[(X - E[X])^2\right]$$

Intuitively, $\text{var}(X)$ measures how far a set of (random) numbers are spread out from their average value.

Definition 15.1.4 — Standard-Deviation. Let X be a random variable. The *standard-deviation* of X, denoted $\sigma(X)$, is the square root of its variance.

$$\sigma(X) \overset{def}{=} \sqrt{\text{var}(X)}$$

15.1.2.2 Sample Mean and Standard Deviation

The mean of a probability distribution is thus the long-run arithmetic average value of a random variable having that distribution. In practice, it is in general impossible to calculate it. Consequently, it is approximated by the average value of a sample of data obtained from some experiment, a Monte-Carlo simulation in our case.

Definition 15.1.5 — Sample Mean. Let $S = \{x_1, x_2, \ldots, x_n\}$ be a set of real valued data. The the *sample mean*, usually denoted by $\overline{\mu}(S)$, is the sum of the values divided by the number of values in the data set:

$$\overline{\mu}(S) \overset{def}{=} \frac{\sum_{i=1}^{n} x_i}{n}$$

$\overline{\mu}(S)$ of a sample S must be distinguished from the mean $\mu(X)$ of the underlying random variable X. However, the law of large numbers ensures that the larger the size of the sample, the more likely it is that its mean is close to the actual population mean, see Appendix A for more details.

Note that in practice, it may not be possible to store all of the x_i's of the sample. It is however always possible to calculate their sum "on-the-fly", i.e. each time a new value is obtained, it is added to the current sum. Then, when the mean is to be calculated, it suffices to divide the accumulated sum by the number of values in the sample.

As for the mean, we need to estimate the variance and the standard-deviation from the sample. The expression defining the variance can be expanded:

$$\begin{aligned}
\mathrm{var}(X) &\overset{def}{=} E\left[(X - E[X])^2\right] \\
&= E\left[X^2 - 2XE[X] + E[X]^2\right] \\
&= E\left[X^2\right] - 2E[X]E[X] + E[X]^2 \\
&= E\left[X^2\right] - E[X]^2
\end{aligned}$$

The naïve algorithm to estimate the variance $\mathrm{var}(X)$ (and the standard deviation $\sigma(X)$) from a sample S consists simply in applying the above formula, i.e. in calculating the mean of the squares and the square of the mean of the values in S, and in dividing their difference by the number of values. To do so, it suffices to accumulate the sum of the squares of the values and the sum of the values, as for the mean.

Definition 15.1.6 — Sample Variance and Standard-Deviation. Let $S = \{x_1, x_2, \ldots, x_n\}$ be a set of real valued data, then the *sample variance* and *sample standard-deviation*

$$\begin{aligned}
\overline{\mathrm{var}}(S) &\overset{def}{=} \frac{\sum_{i=1}^{n} x_i^2}{n} - \overline{\mu}(S)^2 \\
\overline{\sigma}(S) &\overset{def}{=} \sqrt{\overline{\mathrm{var}}(S)}
\end{aligned}$$

In case the two components of the right hand side of the above equation are similar in magnitude, this algorithm may suffer from biases and numerical instability. Fortunately, there exist better algorithms to handle these cases. A first correction is provided by the Bessel's formula, which consists in multiplying the variance obtained by the naïve algorithm by a factor $\frac{n}{n-1}$. This does not solve however numerical instability. The Welford's online algorithm makes it possible to get a good estimate of the variance "on-the-fly", see e.g. (Knuth 1998).

In most of practical cases however, the naïve algorithm is good enough. This is the approach used in XFTA.

15.1.2.3 Confidence Intervals and Error Factors

A *point estimate* is a single value given as the estimate of a population parameter of interest, for example the mean. In contrast, an *interval estimate* specifies a range within which the parameter is estimated to lie. *Confidence intervals*, also called *confidence ranges*, are commonly reported along with point estimates of the same parameters, to show the reliability of the estimates. A confidence interval comes with a confidence level, which specifies the probability that the actual value of the parameter lies in the given interval. For a same sample, the smaller the confidence range, the smaller the confidence level, and vice-versa, the larger the confidence level the larger the confidence range. Most commonly, the 95% confidence level is used. However, other confidence levels are also used, for example, the 90% and the 99% confidence levels.

The *margin of error* or *error factor* is usually defined as the “radius” (or half the width) of a confidence interval.

Definition 15.1.7 — Error Factor. Let S be a sample of n real values. The *error factor* corresponding to a confidence level α, denoted $\mathrm{EF}_\alpha(x)$, is defined as follows.

$$\mathrm{EF}_\alpha(S) \overset{def}{=} t_\alpha \times \frac{\overline{\sigma}(S)}{\sqrt{n}}$$

The *confidence factor* t_α is obtained, assuming a normal distribution of the values, by looking at the table defining the normal law. Typical values chosen for t_α are $t_{90\%} \approx 1.64$, $t_{95\%} \approx 1.96$ and $t_{99\%} \approx 2.58$.

Definition 15.1.8 — Confidence Interval. Let S be a sample of n real values. The *confidence interval* corresponding to a confidence level α, denoted $\mathrm{CI}_\alpha(x)$, is defined as follows.

$$\mathrm{CI}_\alpha(x) \overset{def}{=} [\overline{\mu}(x) - \mathrm{EF}_\alpha(x), \overline{\mu}(x) + \mathrm{EF}_\alpha(x)]$$

15.1.2.4 Histograms and Quantiles

Moments give a usually a good summary of the phenomena at stake. In some cases however, one can be interested in a finer grain analysis of the probability distributions. This can be achieved by means of the extraction of histograms and the calculation of quantiles. We shall look at both in turn.

Histograms

Consider a sample of values $S = \{x_1, x_2 \ldots x_n\}$. A *histogram* H for X is built over a series of *bins*, i.e. contiguous intervals: $H = \{[h_1,h_2],[h_2,h_3]\ldots[h_{k-1},h_k]$ such that $h_j < h_{j+1}$ and all $x_i's$ are comprised between h_1 and h_k. Extracting the histogram consists then in counting the number of $x_i's$ lying in each bin. Although it is not strictly necessary, it is in general assumed that bins are of equal size.

Histograms can be very interesting as they summarize how the x_i's are distributed. They are a convenient abstraction of the sample.

In principle, extracting a histogram from a sample is thus rather simple. The extraction of the histogram is done in three steps:

1. The minimum and maximum values of the sample are determined (by scanning all x_i's).
2. The interval between the minimum value and the maximum value is split into k sub-intervals of equal size.
3. By scanning again all x_i's, one determines how many values lie in each interval.

An approximate discrete cumulative distribution can be obtained from the histogram by summing the number of values in intervals preceding the considered intervals.

Note that it is not necessary to store the values of the sample in computer memory to extract a sample. It suffices to scan them twice: once to determine the bins (by extracting minimum and maximum values of the sample), and once to count the number of values lying in each bin. This assumes however that values are stored somewhere, which is not always possible: in some experiments, values are just calculated but not stored. Histograms have thus to be extracted on-the-fly . A solution to this problem consists in choosing *a priori* the minimum and maximum values (and therefore the intervals) in such way that all values lie between them and that they are not too far from the actual minimum and maximum.

The main problem of extracting histograms stand however elsewhere, namely in the fact that values of the sample may be very unevenly distributed between the minimum and maximum values. In such cases, considering bins of equal size does not provide meaningful results.

Quantiles

An alternative approach consists in calculating quantiles.

Quantiles are cut points dividing the range of a probability distribution into successive intervals with equal probabilities, or dividing the observations in a sample in the same way to estimate their values. Common quantiles have special names: *quartiles* (when the probability distribution is divided in 4), *deciles* (when it is divided in 10), *centiles* (when it is divided in 100). The *median* is the point such that half of the values in the sample are below and half are above, i.e. the sample is divided into two sub-intervals.

In principle, estimating quantiles is rather simple. Let $x_1,\ldots, x_n$ the n numerical values in the sample. The extraction of quantiles is done in three steps:

1. The x_i's are sorted in ascendant order.
2. The sorted list of the x_i's is split into q sub-list of equal size (where q depends on which quantiles one wants to obtain).
3. The i-th q-quantile is the highest value in the i-th sorted sub-list.

Making the above principle to work on-the-fly, i.e. without recording all the x_i's is rather tricky: only approximated values can be obtained. A full presentation of algorithms that perform such on-the-fly calculation goes beyond the scope of this guide. The main idea is to maintain successive bins of equal probabilities. Values accumulated in each bin are considered as random variables, for which a mean, a standard-deviation as well as extremum values can be calculated on-the-fly.

15.1.3 Random Number Generator

Monte-Carlo simulation requires generating numbers at random. A *random-number generator* is a device that generates a sequence of numbers that cannot be reasonably predicted better than by a random chance. This definition is somehow circular as it relies on the concept of random chance, but the intuitive idea is there. A full mathematical treatment of the subject goes much beyond the scope of this guide.

Random number generators can be hardware random-number generators, which generate genuinely random numbers, or pseudo-random number generators, i.e. algorithms which generate series of numbers which look random, but are actually deterministic. The former are not very convenient, at least in the context of computer-based experiments, for they require to connect computers with such devices. The latter present not only the advantage of being cheaper, but also that series of numbers generated by the algorithm can be reproduced at will.

It remains to find "good" generation algorithms, i.e. algorithms that mimic as much as possible "true" randomness, while being not too expensive from a computational view point. Algorithmic generators can actually suffer from several defects:

– Lack of uniformity of distribution for large quantities of generated numbers;
– Correlation of successive values;
– Poor dimensional distribution of the output sequence;
– The distances between where certain values occur may be distributed differently from those in a "true" random sequence distribution;

Fortunately, all these problems are now solved, due to the generalization of 64 bits machines, and more importantly to the introduction of techniques based on linear recurrences on the two-element field. With that respect, the invention of the Mersenne Twister generator in 1997 (Matsumoto and Nishimura 1998) was a major step forward. This generator (or a successor of thereof) is nowadays implemented in random number generation libraries of programming languages, including of course C++. This is the one which is used in XFTA.

Table 15.1: Options of the command `compute sensitivity`

Option	Type	Default value
`source-handle`	Identifier	None
`quantification-method`	Identifier	`PUB`
`mission-time`	List descriptor	0.0
`number-of-tries`	Integer	10000
`print-mean`	Boolean	`true`
`print-standard-deviation`	Boolean	`true`
`print-confidence-range`	Boolean	`true`
`print-error-factor`	Boolean	`false`
`number-of-quantiles`	Integer	0
`number-of-bins`	Integer	0
`print-parameter-sensitivity`	Boolean	`false`
`print-basic-event-sensitivity`	Boolean	`false`
`output`	String	`results.txt`
`mode`	`write`, `append`	`write`

15.2 Commands to Perform Sensitivity Analyses

15.2.1 Main Command

The command to perform a sensitivity analysis on the probability of a variable is `compute sensitivity`. In its base form, it is a follows (assuming the variable `Top` is the top event of the model).

```
compute sensitivity Top;
```

However, performing a sensitivity analysis means performing a Monte-Carlo simulation, which in turn requires to select a number of tries. This is achieved by means of the option `number-of-tries` or the corresponding environment variable. E.g.

```
compute sensitivity Top number-of-tries=10000;
set option number-of-tries 20000;
compute sensitivity Top;
```

Options of the command `compute sensitivity` are summarized in Table 15.1.
Several of these options have been presented Section 13.2.1.

- `output` sets the path to the output file.
- `mode` sets the output mode, i.e. either (`write` or `append`).
- `source-handle` sets the handle (BDT, BDD or ZBDD) from which the quantification is performed.
- `quantification-method` sets the quantification method in case the handle encodes a sum of minimal cutsets.

We shall now review the other options of the command in turn.

15.2.1.1 Mission Times

The mission time(s) can be given in one of the three following forms.

- A single real value, e.g.

```
compute sensitivity Top mission-time=8760;
```

– A list of real values surrounded with square brackets "`[`" and "`]`" and separated with commas "`,`", e.g.

```
compute sensitivity Top mission-time=[0, 2190, 4380, 8760];
```

– A range iterator with a minimum value, a maximum value and a step (in this order), e.g.

```
compute sensitivity Top mission-time=range(0, 8760, 2190);
```

The above iterator produces the list `[0, 2190, 4380, 6570, 8760]`

15.2.1.2 Statistics

Other options set up which statistics should be performed on the sample generated by the Monte-Carlo simulation.

The following Boolean options are used to set up which moments should be printed out.

- `print-mean` to print the sample mean.
- `print-standard-deviation` to print the sample standard-deviation.
- `print-confidence-interval` to print the sample confidence interval.
- `print-error-factor` to print the sample error factor.

By default, XFTA prints the 95It is possible to change this default behavior, see Section **??**.

XFTA makes it possible to print out quantiles in two forms: either the maximum values (quantiles strictly speaking) of each bins are printed out, or mean values. The number of quantiles is set up by the integer option `number-of-quantiles` in the first case, and `number-of-bins` in the second one. E.g.

```
compute sensitivity Top number-of-tries=10000
   number-of-quantiles=20;
set option number-of-bins 100;
compute sensitivity Top number-of-tries=10000;
```

Note that quantiles are computed on-the-fly. Their values are thus only approximated.

In addition to statistics on the probability of the top event, it is possible to print out statistics on the values of the parameters and the probabilities of basic events. The Boolean options to do so are the following.

- `print-parameter-sensitivity` to print statistics on values of parameters.
- `print-basic-event-sensitivity` to print statistics on values of basic events.

Only moments are printed out for parameters and basic events (not quantiles). The printed moments are the same as those printed out for the probability of the top event.

15.2.1.3 Complexity of Calculations

The complexity in time of the command `compute sensitivity` depends on the quantification method the number of mission times m at which the simulation is performed and the number n of tries of each simulation. It is in:

- $\mathcal{O}(n \times m \times |\varphi|)$ if φ is a sum of disjoint products (encoded by a binary decision diagram);
- $\mathcal{O}(n \times m \times |\varphi|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the rare event approximation or the pivotal upper bound;
- $\mathcal{O}(n \times m \times |\mathrm{SoP}(\varphi)|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the mincut upper bound.

Mission times and sensitivity analyses
Although XFTA makes it possible to perform sensitivity analyses at different mission times, this functionality must be used with much care as it may involve enormous amount of calculations. This is the reason why options `add-singular-point` and `smooth-curve` that are available for the assessment of the probability of the top event are not available for the command `compute sensitivity`.

15.2.2 Auxiliary Commands

15.2.2.1 Random Number Generator Seed

As explained above, Monte-Carlo simulations require to generate numbers at pseudo-random. The Mersenne-Twister random number generator used in XFTA works by calculating successive numbers $z_1 = MS(z_0)$, $z_2 = MS(z_1)$, $z_3 = MS(z_2)$ and so on. z_0 is called the *seed* of the sequence.

This process is fully deterministic: starting from the seed z_0, the generator produces always the same sequence $z_1, z_2\ldots$

This is a very important property that makes experiments reproducible.

The seed of the random generator is set arbitrarily when the program is launched. It is possible to set the seed "by hand" before starting a sensitivity analysis by means of the command `set randomizer-seed`. E.g.

```
set randomizer-seed 234561;
```

The current value of the seed can be printed out using the command `print randomizer-seed`. E.g.

```
print randomizer-seed output="seed.xfta";
```

15.2.2.2 Confidence Factor

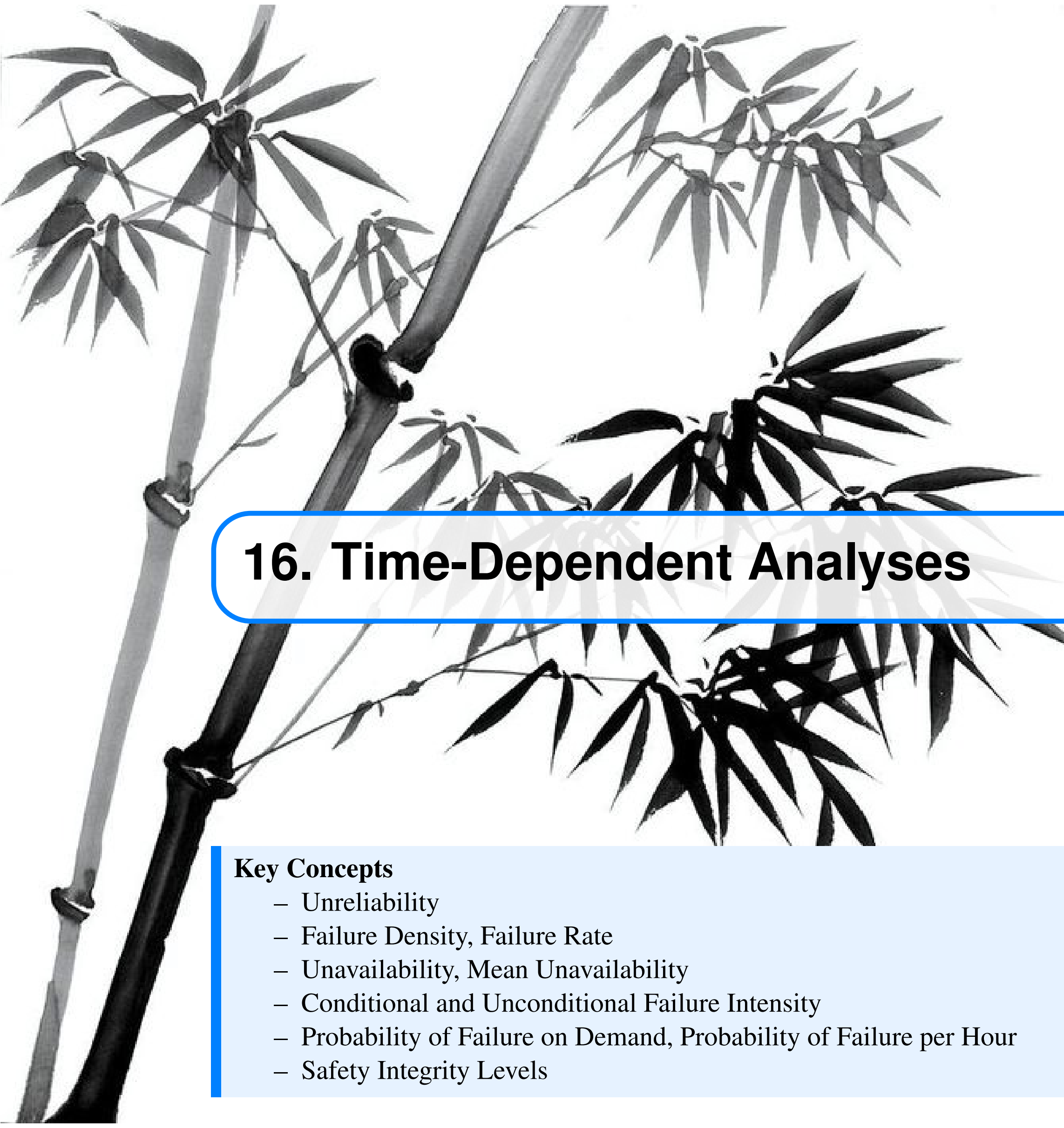
By default, XFTA prints out 95It is possible to change this default behavior by setting the environment variable `confidence-factor`, i.e. the t_α of definition 15.1.7 (which by default is thus close to 1.96).

This is achieved by the command `set option confidence-factor`. E.g.

```
set option confidence-factor 2.575829303549; // 99% confidence factor
```

The current value of the confidence factor can be printed out by means of the command `print option`. E.g.

```
print option confidence-factor output="CF.xfta";
```



16. Time-Dependent Analyses

Key Concepts

– Unreliability
– Failure Density, Failure Rate
– Unavailability, Mean Unavailability
– Conditional and Unconditional Failure Intensity
– Probability of Failure on Demand, Probability of Failure per Hour
– Safety Integrity Levels

This chapter presents XFTA commands to calculate failure intensities, to approximate unreliability and other indicators related to so-called safety integrity levels.

16.1 Introduction

Safety Integrity Levels play a central role in IEC 61508 and IEC 61511 standards and daughters.

– *International IEC Standard IEC61508 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems (E/E/PE, or E/E/PES)* 2010,
– *International IEC Standard IEC61511 - Functional safety - Safety instrumented systems for the process industry sector* 2020.

The concept of *Development Assurance Level* of DO-178C and ARP4761 standards has similar goals.

– *Software Considerations in Airborne Systems and Equipment Certification* 2012,
– *Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment* 2004.

The *safety integrity level* is defined as a relative level of risk-reduction provided by a safety function, or to specify a target level of risk reduction. In simple terms, the safety integrity level is a measurement of performance required for a safety related system. IEC 61508 standard identifies two modes of functioning: safety related system working in low demand modes of operation, and those working in high demand or continuous modes. These two kinds of modes are arbitrarily split according to the demand frequency (lower or higher than once per year). The calculation of the average value of the so-called *probability of failure on demand* is required for those in the first category. The calculation of the so-called *probability of failure per hour* is required for those in the second category.

The mode of operation should actually be determined by comparing demand and proof test frequencies: when the demand frequency is low compared to the test frequency, a failure occurring during the test interval is likely to be detected and repaired before the occurrence of a demand. Consequently, the safety related system behaves almost independently from the protected installation. An accident happens if the safety related system is unable to respond (i.e. unavailable) when a demand occurs. The concept of probability of failure on demand is therefore identical to the traditional concept of unavailability (which is computed by XFTA through the command `compute probability`.

As the frequency of demand increases (compared to the frequency of tests), the probability that the failure of the safety related system is detected and repaired decreases. It even reaches 0 in genuine continuous mode. In this case, the safety related system and the protected installation become tightly linked. If the safety related system is the only protection system of the operation process, an accident is likely to happen as soon as the former experiences its first failure (i.e. is unreliable). The concept of probability of failure per hour is therefore strongly connected to the traditional concept of unreliability. The command `compute failure-intensity`, introduced in this chapter is dedicated to this type of calculations.

In both case, it is worth to observing the evolution of the value of these indicators throughout a sufficiently large period of time: test intervals of each (periodically tested) components strongly influence the value of the indicator and there is nothing as an asymptotic value (at least if components are assumed to be almost as good as new after their maintenance).

XFTA provides commands to evaluate the availability or to approximate the reliability of a system throughout a period of time. This chapter describes the mathematical foundations as well as the XFTA commands to perform these calculations.

16.2 Mathematical and Algorithmic Framework

Several of the definitions below have been already given in preceding chapters. We recall them in this section however, for the sake of convenience.

16.2.1 Availability and Related Indicators

The availability of the system S at time t is the probability that S is working at time t, given it was working at time 0.

Definition 16.2.1 — Availability and Unavailability. The *availability* of S at time t, denoted $A_S(t)$, is:

$$A_S(t) \overset{def}{=} p(S \text{ is working at time } t \mid S \text{ was working at time } 0)$$

Its *unavailability*, denoted $Q_S(t)$, is then defined simply as:

$$Q_S(t) \overset{def}{=} 1 - A_S(t)$$

The following well known property follows from the definition.

Property 16.1 — Unreliability versus Unavailability. If the system under study is not repairable, then $F_S(t) = Q_S(t)$. In any case, $F_S(t) \geq Q_S(t)$.

The conditional failure intensity refers to the probability that the system fails per unit time at time t, given it was working at time 0 and it is working at time t. Formally,

Definition 16.2.2 — Conditional Failure Intensity. The *conditional failure intensity* of S at time t, denoted $\lambda_S(t)$, is:

$$\lambda_S(t) \overset{def}{=} \lim_{dt \to 0} \frac{p(S \text{ fails between } t \text{ and } t+dt \mid S \text{ is working at } t)}{dt}$$

The *average conditional failure intensity* of S at time t, denoted $\lambda_S^{avg}(t)$, is:

$$\lambda_S^{avg}(t) \overset{def}{=} \frac{1}{t} \times \int_{u=0}^{t} \lambda_S(u)\, du$$

$\lambda_S(t)$ is thus an indicator of how the system is likely to fail.

The unconditional failure intensity refers to the probability that the system fails per unit of time at time t, given it was working at time 0. Formally,

Definition 16.2.3 — Unconditional Failure Intensity. The *unconditional failure intensity* of S at time t, denoted $\omega_S(t)$, is:

$$\omega_S(t) \overset{def}{=} \lim_{dt \to 0} \frac{p(S \text{ fails between } t \text{ and } t+dt \mid S \text{ was working at } 0)}{dt}$$

The following property holds, thanks to the definition of conditional probability.

Property 16.2 — Conditional Failure Intensity. $\lambda_S(t)$ and $\omega_S(t)$ obey the following equality.

$$\lambda_S(t) \overset{def}{=} \frac{\omega_S(t)}{A_S(t)}$$

Recall that the marginal importance factor of a basic event is defined as follows (see Chapter 14 for more details).

Definition 16.2.4 — Marginal Importance Factor. The *marginal importance factor* of S with respect to an event E at time t, denoted $\mathrm{MIF}_{S,E}(t)$, is:

$$\mathrm{MIF}_{S,E}(t) \overset{def}{=} \frac{\partial p(T)(t)}{\partial p(E)(t)}$$

$\mathrm{MIF}_{S,E}(t)$ is the probability that S is in a critical state with respect to E, i.e. in a state such that of E is realized then the system is failed and if E is not realized then S is working.

Considering all basic events of the model representing the failures of S, we can estimate $\omega_S(t)$ by means of the following property.

Property 16.3 — Unconditional Failure Intensity. Let S be the top event of a system of stochastic Boolean equations and let $\mathrm{var}(S)$ be the set of basic events involved in the structure function of S. Then, the following equality holds.

$$\omega_S(t) \overset{def}{=} \sum_{E \in \mathrm{var}(S)} \omega_E(t) \times \mathrm{MIF}_{S,E}(t)$$

where $\omega_E(t) = \lambda_E(t) \times A_E(t)$.

16.2.2 Reliability and Related Indicators

Let S denote the system under study. Let T denote the date of the first failure of S. We shall assume that T is a random variable. T is called the *lifetime* of S. We shall assume moreover that components of S were as good as new at time 0 and that they are as good as new after a repair.

Definition 16.2.5 — Reliability and Unreliability. The *reliability* of S at time t, denoted $R_S(t)$, is the probability that S experiences no failure during time interval $[0,t]$, given it was working at time 0. Formally,

$$R_S(t) \overset{def}{=} p(t < T)$$

The *unreliability*, denoted $F_S(t)$ is just the complement to 1 of the reliability.

$$F_S(t) \overset{def}{=} 1 - R_S(t)$$

$F_S(t)$ is a cumulative distribution function.

The failure density refers to the probability density function of T.

Definition 16.2.6 — Failure Density. The *failure density* of S at time t, denoted $f_S(t)$, is the derivative of $F_S(t)$.

$$f_S(t) \overset{def}{=} \frac{dF_S(t)}{dt}$$

For sufficiently small δt's, the product $f_S(t) \times \delta t$ expresses the probability that the system fails between t and $t+\delta t$, given it was working at time 0.

The failure rate or hazard rate is the probability the system fails for the first time per unit of time at age t. Formally,

Definition 16.2.7 — Failure Rate. The *failure rate*, also called *hazard rate* of S at time t, denoted $r_S(t)$, is:

$$r_S(t) \overset{def}{=} \lim_{dt\to 0} \frac{p(t \le T \le t+dt)}{dt}$$

The following property follows directly from the definitions.

Property 16.4 — Unreliability and Failure Rate. The following equality holds.

$$F_S(t) \overset{def}{=} 1 - \exp\left(-\int_{u=0}^{t} r_S(u)\, du\right)$$

The very principle of fault tree analysis (and beyond assessment of systems of stochastic Boolean equations) is to consider the state of the system at time t, disregarding how the system evolved from time 0 to time t. To put it differently, fault tree assessment consists in taking a snapshot of the state of the system at time t. This means that the only availability can be assessed, not reliability.

Nonetheless, reliability can be approximated. An idea, initially proposed by W. E. Vesely then explored by Y. Dutuit and myself (Dutuit and Rauzy 2005), consists in substituting $\lambda_S(t)$ for $r_S(t)$ in the calculation of $F_S(t)$, i.e.

$$\widetilde{F}_S(t) \overset{def}{=} 1 - \exp\left(-\lambda_S^{avg}(t) \times t\right) \tag{16.1}$$

Table 16.1: Safety Integrity Levels

(a) Low demand safety related systems

SIL	$\text{PFD}_S^{avg}(t)$
1	10^{-1}–10^{-2}
2	10^{-2}–10^{-3}
3	10^{-3}–10^{-4}
4	$\leq 10^{-4}$

(b) Continuous mode safety related systems

SIL	$\text{PFH}_S(t)$
1	10^{-5}–10^{-6}
2	10^{-6}–10^{-7}
3	10^{-7}–10^{-8}
4	$\leq 10^{-8}$

In the cited article, we showed that $\widetilde{F_S}(t)$ gives often a reasonably good approximation of $F_S(t)$, i.e.

$$F_S(t) \quad \approx \quad \widetilde{F_S}(t)$$

This approximations requires to calculate $\lambda_S(t) = \omega_S(t) \times (1 - Q_S(t))$ at sufficiently many intermediate dates between time 0 and the chosen mission time.

16.2.3 Safety Integrity Levels

The IEC 61508 standard introduces the following indicators:

- Probability of failure on demand,
- Average of probability of failure on demand,
- Probability of failure per hour.

From the above development, these indicators can be defined as follows.

Definition 16.2.8 — Probability of Failure on Demand. The *probability of failure on demand* of S at time t, denoted $\text{PFD}_S(t)$, is simply its unavailability:

$$\text{PFD}_S(t) \quad \stackrel{def}{=} \quad Q_S(t)$$

The *average probability of failure on demand* of S at time t, denoted $\text{PFD}_S^{avg}(t)$, is thus:

$$\text{PFD}_S^{avg}(t) \quad \stackrel{def}{=} \quad \frac{\int_{u=0}^{t} Q_S(u)\, du}{t}$$

Definition 16.2.9 — Probability of Failure per Hour. The *probability of failure per hour* of S at time t, denoted $\text{PFH}_S(t)$, is derived from its unreliability:

$$\text{PFH}_S(t) \quad \stackrel{def}{=} \quad \frac{F_S(t)}{t}$$

Safety integrity levels, SIL for short, are discrete numerical indicators, ranging from 1 to 4. They are defined separately for low demand and continuous mode safety related systems.

- Fow low demand mode, they are defined via the average probability of failure on demand.
- For high demand mode, they are defined via the probability of failure per hour.

Table 16.1 gives definitions of safety integrity levels as defined by IEC 61508 standard for low demand and continuous mode safety related systems.

The calculation of $\text{PFD}_S^{avg}(t)$ requires assessing $\text{PFD}_S(t) = Q_S(t)$ at sufficiently many dates between 0 and t. Even if it is the case, taking a mean value as an indicator may be problematic: as aircraft pilots use to say take-off is optional, but landing is mandatory. Having the safety related system available on average does not mean that it does not go through short periods in which the probability that is out of order is very high.

As explained in the previous section, it is not possible to assess the unreliability, and consequently the probability of failure per hour, from a combinatorial model. However, following the line of equation 16.1, we can approximate it as follows.

$$\mathrm{PFH}_S(t) \approx \omega_S(t) \tag{16.2}$$

16.3 Commands to Calculate Safety Integrity Levels and Related Indicators

There are two commands to calculate safety integrity levels and related indicators: `compute probability` and `compute failure-intensity`. We shall now review them in turn.

Time-dependent analyses
Most of time dependent analyses involve the numerical integration of an indicator over the time. For this to make sense, at least two mission times must be given, e.g.

```
set option mission-time [0, 8760];
```

The current version of XFTA *does not* include automatically the mission time 0 in the list of mission times.

16.3.1 Low Demand Safety Related Systems

The command `compute probability` makes it possible to assess safety related system with a low demand. This command is documented Section 13.2.1. Its options are given Table 13.1.

As explained above, to get accurate values of the average probability of failure on demand $\mathrm{PFD}_S^{avg}(t)$, i.e. the mean unavailability $Q_S^{avg}(t)$, one needs to calculate the unavailability $Q_S(t)$ i.e. eventually the probability of the top event $p(S)$ at sufficiently many dates between 0 and the chosen mission time. It is in particular recommended to add all singular points (via the option `add-singular-points`) as well as to smooth the curve (via the option `smooth-curve`).

Two indicators can then be displayed for each date involved in the calculation:

- The unavailability $Q_S(t)$, via the option `print-probability` which is set by default to `true`.
- The mean unavailability $Q_S^{avg}(t)$, via the option `print-mean-unavailability` which is set by default to `false`.

If the option `print-safety-integrity-level` is set to `true`, the sojourn times in each safety integrity level are displayed (computed at the last date). E.g.

```
set option output "Results/hipps1.tsv";
set option mode write;
compute probability top
   mission-time = range(0, 43800, 730)
   add-singular-points = true
   smooth-curve = true
   print-unavailability = true
   print-mean-unavailability = true
   print-safety-integrity-level = true;
```

Table 16.2 shows what the result of the SIL calculation, once loaded in a spreadsheet tool, looks like.

Table 16.2: Result of SIL calculation for a low demand safety related system

top-event	top			
source-handle	BDD			
quantification-method	Pr			
Qavg	0.000106211			
SIL	lower-bound	upper-bound	sojourn-time	ratio
0	0.01	1	0	0
1	0.001	0.01	0	0
2	0.0001	0.001	21026.3	0.480053
3	1e-05	0.0001	22763.3	0.519709
4	0	1e-05	10.4138	0.000237759

Table 16.3: Options of the command `compute failure-intensity`

Option	Type	Default value
`source-handle`	Identifier	None
`quantification-method`	Identifier	`PUB`
`mission-time`	List descriptor	0.0
`add-singular-points`	Boolean	`false`
`smooth-curve`	Boolean	`false`
`smoothing-ratio`	Real in $]0,1[$	0.1
`print-conditional-failure-intensity`	Boolean	`true`
`print-mean-conditional-failure-intensity`	Boolean	`false`
`print-unreliability`	Boolean	`false`
`print-probability-of-failure-per-hour`	Boolean	`false`
`print-safety-integrity-level`	Boolean	`false`
`output`	String	`results.txt`
`mode`	`write`, `append`	`write`

16.3.1.1 Complexity of Calculations

Assume that that indicators are calculated from a sum of products (sum of disjoint products or sum of minimal cutsets) φ associated with given target variable S and that they are calculated at m mission times. The command `compute probability` consists thus essentially in assessing the unavailability $Q_S(t)$ at each mission time t. Its complexity is thus in:

– $\mathcal{O}(m \times |\varphi|)$ if φ is a sum of disjoint products (encoded by a binary decision diagram);
– $\mathcal{O}(m \times |\varphi|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the rare event approximation or the pivotal upper bound;
– $\mathcal{O}(m \times |\mathrm{SoP}(\varphi)|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the mincut upper bound.

16.3.2 Continuous Mode Safety Related Systems

16.3.2.1 Command

The command `compute failure-intensity` makes it possible to assess safety related system with a low demand. Its options are given Table 16.3.

Several of these options have been presented Section 13.2.1.

– `output` sets the path to the output file.

– `mode` sets the output mode, i.e. either (`write` or `append`).
– `source-handle` sets the handle (BDT, BDD or ZBDD) from which the quantification is performed.
– `quantification-method` sets the quantification method in case the handle. encodes a sum of minimal cutsets.
– `mission-time` sets the mission times at which indicators are calculated.
– `add-singular-points` adds singular points to the set of the mission times at which indicators are calculated.
– `smooth-curve` smooth the curve of the conditional failure intensity, by adding mission times.
– `smoothing-ratio` sets the smoothing ratio.

We shall now review the other options of the command `compute failure-intensity` in turn.

16.3.2.2 Indicators

Option `print-conditional-failure-intensity`
If this option is set to `true`, the conditional failure intensity $\lambda_S(t)$ is calculated and printed for each mission time.

Option `print-mean-conditional-failure-intensity`
If this option is set to `true`, the conditional failure intensity $\lambda_S(t)$ is calculated for each mission time. Moreover, for each mission time its average value $\frac{\sum_{i=1}^{k}\lambda_S(t_i)}{k}$ over the preceding mission times is calculated and printed.

Option `print-unreliability`
If this option is set to `true`, the approximation of the reliability $\widetilde{F_S}(t)$ is calculated and printed for each mission time.

Option `print-probability-of-failure-per-hour`
If this option is set to `true`, the probability of failure per hour $\mathrm{PFH}_S(t)$, i.e. eventually the unconditional failure intensity $\omega_S(t)$ is calculated and printed for each mission time.

Option `print-safety-integrity-level`
If this option is set to `true`, the safety integrity level in high demand mode of the system for the whole mission is calculated, via the calculation of $\mathrm{PFH}_S(t)$ (and thus $\omega_S(t)$) at each mission time. More exactly, XFTA estimates the sojourn time of the system at each safety integrity level and print them out.

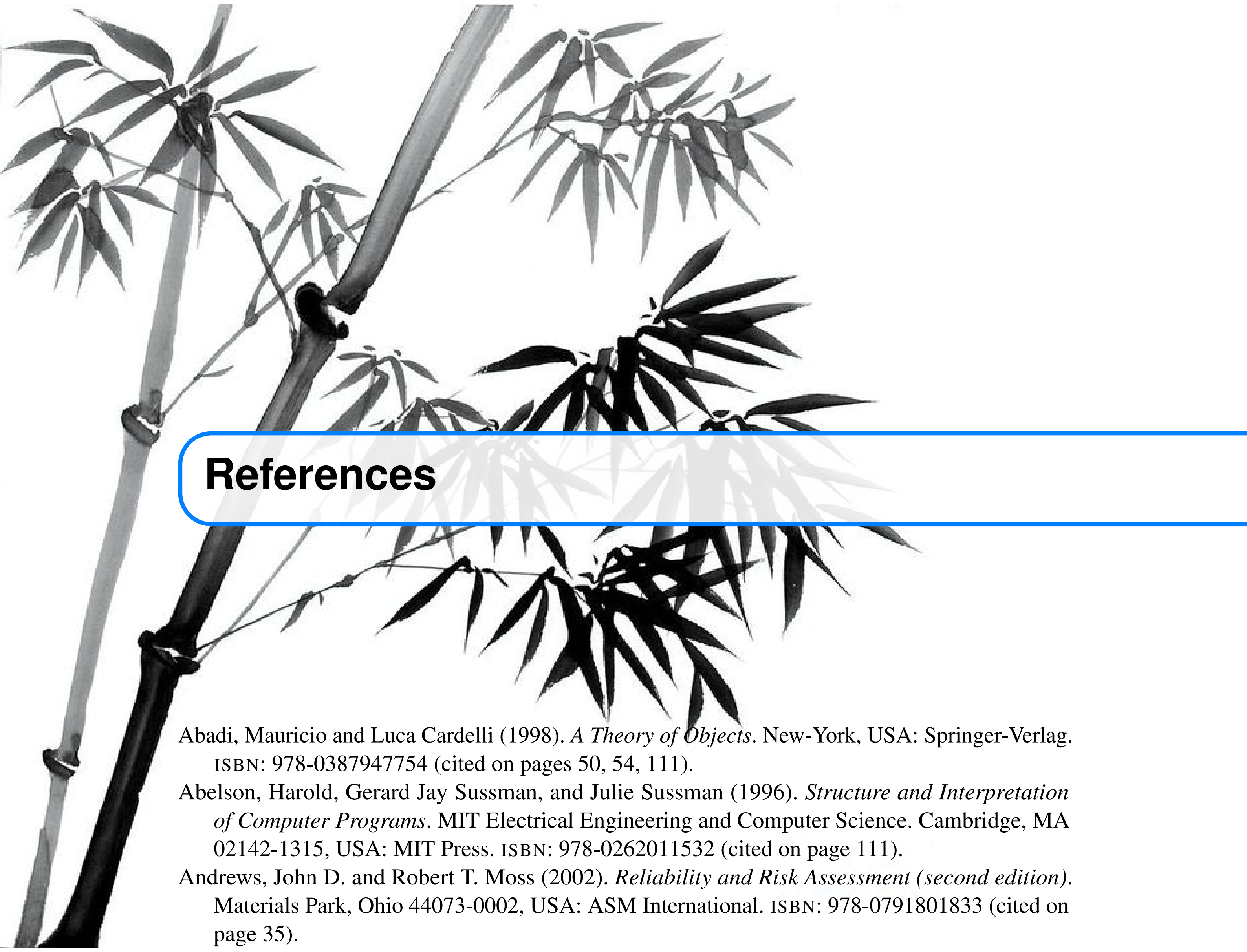
16.3.2.3 Complexity of Calculations

Assume that the structure function of the given target variable S involves n basic events, that indicators are calculated from a sum of products (sum of disjoint products or sum of minimal cutsets) φ, and finally that they are calculated at m mission times.

The calculation of the unconditional failure intensity $\omega_S(t)$ requires the calculation of the marginal importance factor $\mathrm{MIF}_{S,E}(t)$ for each basic event E involved in the structure function of S. The calculations of all above indicators involve essentially the calculation of $\omega_S(t)$, or to put it differently, only a few additional operations in addition to the calculation of $\omega_S(t)$. The complexity of the command `compute failure-intensity` is thus in:
– $\mathcal{O}(m \times n \times |\varphi|)$ if φ is a sum of disjoint products (encoded by a binary decision diagram);
– $\mathcal{O}(m \times n \times |\varphi|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the rare event approximation or the pivotal upper bound;

– $\mathscr{O}(m \times n \times |\mathrm{SoP}(\varphi)|)$ if φ is a sum of minimal cutsets (encoded by a binary decision tree or a zero-suppressed binary decision diagram) and the quantification method is either the mincut upper bound.

This complexity analysis concludes the present chapter and the book.



References

Abadi, Mauricio and Luca Cardelli (1998). *A Theory of Objects*. New-York, USA: Springer-Verlag. ISBN: 978-0387947754 (cited on pages 50, 54, 111).

Abelson, Harold, Gerard Jay Sussman, and Julie Sussman (1996). *Structure and Interpretation of Computer Programs*. MIT Electrical Engineering and Computer Science. Cambridge, MA 02142-1315, USA: MIT Press. ISBN: 978-0262011532 (cited on page 111).

Andrews, John D. and Robert T. Moss (2002). *Reliability and Risk Assessment (second edition)*. Materials Park, Ohio 44073-0002, USA: ASM International. ISBN: 978-0791801833 (cited on page 35).

Arora, Sanjeev and Boaz Barak (2009). *Computational Complexity, a Modern Approach*. New-York, NY, USA: Cambridge University Press. ISBN: 978-0-521-42426-4 (cited on page 166).

Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment (July 2004). Standard. Warrendale, Pennsylvania, USA: Society of Automotive Engineers (cited on pages 36, 223).

Barlow, Richard E. and Franck Proschan (1975). “Importance of System Components and Fault Tree events”. In: *Stochastic Processes and their Applications* 3, pages 153–173. DOI: 10.1016/0304-4149(75)90013-7 (cited on page 209).

Batteux, Michel, Tatiana Prosvirnova, and Antoine Rauzy (Sept. 2017). “AltaRica 3.0 Assertions: the Why and the Wherefore”. In: *Journal of Risk and Reliability* 231.6, pages 691–700. DOI: 10.1177/1748006X17728209 (cited on page 110).

— (Oct. 2018). “From Models of Structures to Structures of Models”. In: *IEEE International Symposium on Systems Engineering (ISSE 2018)*. Best paper award. Roma, Italy: IEEE. DOI: 10.1109/SysEng.2018.8544424 (cited on pages 7, 9, 44, 110).

— (2019). “AltaRica 3.0 in 10 Modeling Patterns”. In: *International Journal of Critical Computer-Based Systems* 9.1–2, pages 133–165. DOI: 10.1504/IJCCBS.2019.098809 (cited on pages 9, 110, 122).

Beeson, Sally and John D. Andrews (2003). “Importance Measures for Non-Coherent System Analysis”. In: *IEEE Transactions on Reliability* 52.3, pages 301–310. DOI: 10.1109/TR.2003.816397 (cited on page 206).

Berg, Ulf (Apr. 1994). *RISK SPECTRUM, Theory Manual*. RELCON Teknik AB (cited on pages 140, 208).

Birnbaum, Zygmunt William (1969). “On the importance of different components and a multicomponent system”. In: *Multivariable analysis II*. Edited by P. R. Korishnaiah. Academic Press, New York, pages 581–592 (cited on page 205).

Borgonovo, Emanuele and George E. Apostolakis (2001). “A New Importance Measure for Risk-Informed Decision Making”. In: *Reliability Engineering and System Safety* 72.2, pages 193–212. DOI: 10.1016/S0951-8320(00)00108-3 (cited on page 209).

Bouissou, Marc et al. (1991). “Knowledge modelling and reliability processing: presentation of the FIGARO language and of associated tools”. In: *Proceedings of SAFECOMP’91 – IFAC International Conference on Safety of Computer Control Systems*. Edited by Johan F. Lindeberg. Trondheim, Norway: Pergamon Press, pages 69–75. ISBN: 0-08-041697-7 (cited on page 110).

Box, George Pellam (1979). “Robustness in the strategy of scientific model building”. In: *Robustness in Statistics*. Edited by Robert L. Launer and Graham N. Wilkinson. Academic Press, pages 201–236. ISBN: 9781483263366 (cited on page 36).

Brace, Karl S., Richard L. Rudell, and Randal S. Bryant (1990). “Efficient Implementation of a BDD Package”. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. Orlando, Florida, USA: IEEE, pages 40–45. ISBN: 0-89791-363-9. DOI: 10.1145/123186.123222 (cited on pages 136, 173, 175).

Bryant, Randal S. (Aug. 1986). “Graph Based Algorithms for Boolean Fonction Manipulation”. In: *IEEE Transactions on Computers* 35.8, pages 677–691 (cited on page 136).

— (Sept. 1992). “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams”. In: *ACM Computing Surveys* 24, pages 293–318 (cited on page 173).

Chandra, Ashok Kumar and George Markowsky (1978). “On the number of prime implicants”. In: *Discrete Mathemathics* 24.1, pages 7–11. DOI: 10.1016/0012-365X(78)90168-1 (cited on page 167).

Cheok, Michael C., Gareth W. Parry, and Richard R. Sherry (1998). “Use of importance measures in risk informed regulatory applications”. In: *Reliability Engineering and System Safety* 60.3, pages 213–226 (cited on page 208).

Clément, Emmanuel, Thierry Thomas, and Antoine Rauzy (Oct. 2014). “Arbre Analyste : un outil d’arbres de défaillances respectant le standard Open-PSA et utilisant le moteur XFTA”. In: *Actes du congrès Lambda-Mu 19 (actes électroniques)*. Dijon, France: Institut pour la Maitrise des Risques. ISBN: 978-2-35147-037-4 (cited on page 9).

Cormen, Thomas H. et al. (2001). *Introduction to Algorithms (Second ed.)* Cambridge, MA, USA: The MIT Press. ISBN: 0-262-03293-7 (cited on page 134).

Coudert, Olivier and Jean-Christophe Madre (Jan. 1993). “Fault Tree Analysis: 10^{20} Prime Implicants and Beyond”. In: *Proceedings of the Annual Reliability and Maintainability Symposium, ARMS’93*. Edited by T.J. Donovan. Atlanta NC, USA: IEEE, pages 240–245. DOI: 10.1109/RAMS.1993.296849 (cited on page 140).

Davis, Martin D., George Logemann, and Donald W. Loveland (1962). “A Machine Program for Theorem Proving”. In: *Communications of the ACM* 5, pages 394–397. DOI: 10.1145/368273.368557 (cited on pages 8, 161).

de Kleer, Johan (Mar. 1986). “An assumption based TMS”. In: *Artificial Intelligence* 278.2 (cited on page 110).

Software Considerations in Airborne Systems and Equipment Certification (Jan. 2012). Standard. European Organisation for Civil Aviation Equipment (cited on page 223).

Dutuit, Yves and Antoine Rauzy (1996). “A Linear Time Algorithm to Find Modules of Fault Trees”. In: *IEEE Transactions on Reliability* 45.3, pages 422–425. DOI: 10.1109/24.537011 (cited on page 165).

— (2001). “New insights in the assessment of k-out-of-n and related systems”. In: *Reliability Engineering and System Safety* 72.3, pages 303–314. DOI: `10.1016/S0951-8320(01)00024-2` (cited on page 175).

— (2005). “Approximate estimation of system reliability via fault trees”. In: *Reliability Engineering and System Safety* 87.2, pages 163–172. DOI: `10.1016/j.ress.2004.02.008` (cited on page 226).

— (Dec. 2013). “Importance Factors of Coherent Systems: a Review”. In: *Journal of Risk and Reliability* 228.3, pages 313–323. DOI: `10.1177/1748006X13512296` (cited on pages 204, 206, 208).

— (Oct. 2015). “On the Extension of Importance Measures to Complex Components”. In: *Reliability Engineering and System Safety* 142, pages 161–168. DOI: `10.1016/j.ress.2015.04.016` (cited on page 210).

Epstein, Steven, Olivier Nusbaumer, et al. (2007). “A Modest Proposal: A Standard PSA Model Representation Format”. In: *Proceedings of the conference Nuclear Energy for New Europe, 2007*. Edited by I. Jencic and M. Lenosek. Portoroz, Slovenia: INIS. ISBN: 978-961-6207-28-7 (cited on page 9).

Epstein, Steven and Antoine Rauzy (2005). “Can We Trust PRA?” In: *Reliability Engineering and System Safety* 88.3, pages 195–205. DOI: `10.1016/j.ress.2004.07.013` (cited on pages 8, 164).

— (Apr. 2008). *Open-PSA Model Exchange format, version 2.0d.* available at www.altarica-association.org (cited on pages 8, 9, 43).

Epstein, Steven, Antoine Rauzy, and Donald Wakefield (2006). “Can We Trust PRA: Take 3”. In: *Proceedings of he 8th International Conference on Probabilistic Safety Assessment and Management, PSAM 2006*. New Orleans, USA: IAPSAM. ISBN: 9780791802441 (cited on pages 8, 164).

Epstein, Steven, Mark Reinhart, and Antoine Rauzy (June 2008a). “The open PSA initiative for next generation probabilistic safety assessment”. In: *Proceeding of International Topical Meeting on Probabilistic Safety Assessment and Analysis, PSA 2008*. Volume 1. Knoxville, USA: American Nuclear Society, pages 409–419 (cited on page 9).

— (June 2008b). “The open PSA initiative for next generation probabilistic safety assessment”. In: *Proceeding of 9th International Conference on Probabilistic Safety Assessment and Management 2008, PSAM 2008*. Volume 1. Hong-Kong, China: IAPSAM, pages 542–550. ISBN: 978-162276577-5 (cited on page 9).

— (June 2010a). “The Open PSA Initiative for next generation probabilistic safety assessment”. In: *Kerntechnik* 74.3, pages 101–105. DOI: `10.3139/124.110020` (cited on page 9).

— (June 2010b). “Validation Project for the Open-PSA Model Exchange using RiskSpectrum and CAFTA”. In: *Proceeding of the PSAM’10 Conference*. Edited by B.P. Hallbert. (also in E. Fadier ed., Actes du congrès LambdaMu’10). Seatle, USA. ISBN: 9781622765782 (cited on page 9).

Epstein, Steven, Donald Wakefield, and Antoine Rauzy (Oct. 2002). “Very Large Nuclear Risk Models and Binary Decision Diagrams”. In: *Proceedings of PSA 02, International Topical Meeting on Probabilistic Safety Assessment*. Edited by Illinois American Nuclear Society La Grange Park. Detroit, Michigan, USA, pages 80–84 (cited on page 8).

Fowler, Martin (Oct. 2010). *Domain Specific Languages*. Boston, MA 02116, USA: Addison-Wesley. ISBN: 978-0321712943 (cited on page 111).

Friedenthal, Sanford, Alan Moore, and Rick Steiner (2011). *A Practical Guide to SysML: The Systems Modeling Language*. San Francisco, CA 94104, USA: Morgan Kaufmann. The MK/OMG Press. ISBN: 978-0123852069 (cited on page 57).

Fussel, Jerry B. (1975). “How to hand-calculate system reliability characteristics”. In: *IEEE Transactions on Reliability* R-24.3, pages 169–174 (cited on page 207).

Fussel, Jerry B. and W. E. Vesely (June 1972). “A New Methodology for Obtaining Cut Sets for Fault Trees”. In: *Transactions of American Nuclear Society* 15, pages 262–263 (cited on pages 8, 140, 161).

Gachet, Bruno, Tony Hutinet, and Jean Mignot (1992). “FIABEX modeling Mars rover’s localization unit”. In: *Proceedings of the Safety and Reliability conference, ESREL’92*. Edited by Elsevier Applied Science. Copenhagen, Denmark, pages 10–22. ISBN: 1-85166-875-6 (cited on page 110).

Garey, Michael R. and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Fransisco, CA, USA: Freeman. ISBN: 978-0716710455 (cited on page 166).

Hibti, Mohamed, Thomas Friedlhuber, and Antoine Rauzy (June 2012a). “Overview of The Open PSA Platform”. In: *Proceedings of International Joint Conference PSAM’11/ESREL’12*. Edited by Reino Virolainen, pages 2798–2807. ISBN: 978-162276436-5 (cited on page 9).

— (June 2012b). “Variant Management in a Modular PSA”. In: *Proceedings of International Joint Conference PSAM’11/ESREL’12*. Edited by Reino Virolainen, pages 2808–2818. ISBN: 978-162276436-5 (cited on page 9).

International electrotechnical vocabulary - Part 192: Dependability (2015). Standard. Geneva, Switzerland: International Electrotechnical Commission (cited on page 38).

International IEC Standard IEC61508 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems (E/E/PE, or E/E/PES) (Apr. 2010). Standard. Geneva, Switzerland: International Electrotechnical Commission (cited on pages 27, 36, 38, 223).

International IEC Standard IEC61511 - Functional safety - Safety instrumented systems for the process industry sector (Feb. 2020). Standard. Geneva, Switzerland: International Electrotechnical Commission (cited on page 223).

IEC 61709:2017 - Electric components - Reliability - Reference conditions for failure rates and stress models for conversion (Aug. 2017). Standard. Geneva, Switzerland: International Electrotechnical Commission (cited on page 39).

ISO/TR 12489:2013 Petroleum, petrochemical and natural gas industries – Reliability modelling and calculation of safety systems (Nov. 2013). Standard. Geneva, Switzerland: International Organization for Standardization (cited on page 36).

ISO26262 Functional Safety - Road Vehicle (2012). Standard. Geneva, Switzerland: International Standardization Organization. URL: `http://www.iso.org/iso/home.html` (cited on page 36).

Jung, Woo Sik, Sang Hoon Han, and Jaejoo Ha (2004). “A fast BDD algorithm for large coherent fault trees analysis”. In: *Reliability Engineering and System Safety* 83.3, pages 369–374. DOI: `10.1016/j.ress.2003.10.009` (cited on page 140).

Jung, Woo Sik and Jeff Riley (2014). “Strength of ZBDD Algorithm for the Post Processing of Huge Cutsets in Probabilistic Safety Assessment”. In: *Proceedings of Probabilistic Safety Assessment and Management, PSAM 12*. Edited by Todd Paulos and Curtis Lee Smith. Volume 4. Honolulu, Hawaii, USA: Createspace Independent Publishing Platform, pages 156–163. ISBN: 9781540671554 (cited on page 106).

Knuth, Donald E. (1998). *The Art of Computer Programming, volume 2: Seminumerical Algorithms (3rd edn)*. Boston, MA, USA: Addison-Wesley. ISBN: 978-0201896848 (cited on page 217).

Kolmogorov, Andrei N. (1933). *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Berlin, Germany: Springer. ISBN: 978-3540061106 (cited on page 249).

Kroening, Daniel and Ofer Strichman (2017). *Decision Procedures : An Algorithmic Point of View*. Theoretical Computer Science. Berlin, Germany: Springer. ISBN: 978-3662504963 (cited on pages 8, 161).

Kumamoto, Hiromitsu and Ernest J. Henley (1996). *Probabilistic Risk Assessment and Management for Engineers and Scientists*. Piscataway, N.J., USA: IEEE Press. ISBN: 978-0780360174 (cited on pages 35, 43, 100, 134, 203).

Lambert, Howard E. (1975). "Measures of importance of events and cut sets in fault trees". In: *Reliability and Fault Tree Analysis*. Edited by Richard E. Barlow, Jerry B. Fussel, and N.D. Singpurwalla. Philadelphia, PA, USA: SIAM Press, pages 77–100 (cited on page 206).

Matsumoto, Makoto and Takuji Nishimura (1998). "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Transactions on Modeling and Computer Simulation* 8.1, pages 3–30. DOI: `10.1145/272991.272995` (cited on pages 90, 219).

Meyer, Bertrand (1988). *Object-Oriented Software Construction*. MIT Electrical Engineering and Computer Science. Upper Saddle River, New Jersey, USA: Prentice Hall. ISBN: 978-0136290490 (cited on page 111).

Minato, Shin-Ichi (1993). "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems". In: *Proceedings of the 30th ACM/IEEE Design Automation Conference, DAC'93*. Dallas, Texas, USA: IEEE, pages 272–277. ISBN: 0-89791-577-1. DOI: `10.1145/157485.164890` (cited on pages 137, 167).

Naur, Peter (May 1985). "Programming as theory building". In: *Microprocessing and Microprogramming* 15.5, pages 253–261. DOI: `10.1016/0165-6074(85)90032-8` (cited on page 11).

Noble, James, Antero Taivalsaari, and Ivan Moore (1999). *Prototype-Based Programming: Concepts, Languages and Applications*. Berlin and Heidelberg, Germany: Springer-Verlag. ISBN: 978-9814021258 (cited on pages 54, 111).

Nusbaumer, Olivier and Antoine Rauzy (June 2013). "Fault Tree Linking versus Event Tree Linking Approaches: a Reasoned Comparison". In: *Journal of Risk and Reliability* 227.3, pages 315–326. DOI: `10.1177/1748006X13490260` (cited on pages 108, 141).

Papadimitriou, Christos H. (1994). *Computational Complexity*. Boston, MA 02116, USA: Addison Wesley. ISBN: 978-0201530827 (cited on pages 140, 166).

Pierce, Benjamin C. (2002). *Types and Programming Languages*. Cambridge, MA 02142-1315, USA: MIT Press. ISBN: 978-0262162098 (cited on page 111).

Rasmussen, Norman C. (Oct. 1975). *Reactor Safety Study. An Assessment of Accident Risks in U.S. Commercial Nuclear Power Plants*. Technical report WASH 1400, NUREG-75/014. Rockville, MD, USA: U.S. Nuclear Regulatory Commission (cited on pages 36, 196).

Rauzy, Antoine (1993). "New Algorithms for Fault Trees Analysis". In: *Reliability Engineering and System Safety* 05.59, pages 203–211. DOI: `10.1016/0951-8320(93)90060-C` (cited on pages 8, 140, 177, 195).

— (Dec. 2001). "Mathematical Foundation of Minimal Cutsets". In: *IEEE Transactions on Reliability* 50.4, pages 389–396. DOI: `10.1109/24.983400` (cited on pages 157, 160, 161, 166, 168, 173, 177, 180).

— (2003). "Towards an Efficient Implementation of Mocus". In: *IEEE Transactions on Reliability* 52.2, pages 175–180. DOI: `10.1109/TR.2003.813160` (cited on page 140).

— (2008a). "BDD for Reliability Studies". In: *Handbook of Performability Engineering*. Edited by Krishna B. Misra. Amsterdam, the Netherlands: Elsevier, pages 381–396. ISBN: 978-1-84800-130-5 (cited on pages 8, 9, 105, 140, 173).

Rauzy, Antoine (2008b). “Guarded Transition Systems: a new States/Events Formalism for Reliability Studies”. In: *Journal of Risk and Reliability* 222.4, pages 495–505. DOI: `10.1243/1748006XJRR177` (cited on page 110).

— (2008c). “Some Disturbing Facts about Depth First Left Most Variable Ordering Heuristics for Binary Decision Diagrams”. In: *Journal of Risk and Reliability* 222.4, pages 573–582. DOI: `10.1243/1748006XJRR174` (cited on page 177).

— (Oct. 2010). “Diagrammes Binaires de Décision pondérés : vieilles idées, nouvelle mise en œuvre”. In: *Actes du congrès LambdaMu’17 (actes électroniques)*. Edited by Elie Fadier (cited on page 180).

— (June 2012). “Anatomy of an Efficient Fault Tree Assessment Engine”. In: *Proceedings of International Joint Conference PSAM’11/ESREL’12*. Edited by Reino Virolainen. Helsinki, Finland, pages 3333–3343. ISBN: 978-162276436-5 (cited on pages 140, 161).

— (2018). “Notes on Computational Uncertainties in Probabilistic Risk/Safety Assessment”. In: *Entropy* 20.3. DOI: `10.3390/e20030162` (cited on pages 40, 41, 164, 166).

Rauzy, Antoine and Cecilia Haskins (2019). “Foundations for Model-Based Systems Engineering and Model-Based Safety Assessment”. In: *Journal of Systems Engineering* 22, pages 146–155. DOI: `10.1002/sys.21469` (cited on pages 7, 9, 44, 109, 110).

Rauzy, Antoine and Liu Yang (2019). “Finite Degradation Structures”. In: *Journal of Applied Logics – IfCoLog Journal of Logics and their Applications* 6.7, pages 1471–1495 (cited on page 161).

Ray, Erik T. (2003). *Learning XML*. Sebastopol, CA 95472, USA: O’Reilly Media, Inc. ISBN: 978-0596004200 (cited on page 43).

Reliability Prediction of Electronic Equipment: MIL-HDBK-217F. (1995). Technical report. U.S. Department of Defense (cited on page 39).

Rudell, Richard L. (Nov. 1993). “Dynamic Variable Ordering for Ordered Binary Decision Diagrams”. In: *Proceedings of IEEE International Conference on Computer Aided Design, ICCAD’93*. Edited by Michael Lightner and Jochen A. G. Jess. Santa Clara, CA, USA: IEEE, pages 42–47. ISBN: 0-8186-4490-7 (cited on page 176).

Rumbaugh, James, Ivar Jacobson, and Grady Booch (2005). *The Unified Modeling Language Reference Manual*. Boston, MA 02116, USA: Addison Wesley. ISBN: 978-0321267979 (cited on page 57).

SINTEF, prepared by and NTNU, editors (2015). *OREDA Handbook – Offshore Reliability Data, volume 1 and 2, 6th edition* (cited on pages 39, 90, 121).

Sofreten (1992). *SOFIA Spécifications Générales*. Technical report SOFRETEN.24501.005. Sofreten (cited on page 110).

Toda, Seinosuke (Oct. 1991). “PP is as Hard as the Polynomial-Time Hierarchy”. In: *SIAM Journal on Computing* 20.5, pages 865–877. DOI: `10.1137/0220053` (cited on page 167).

Valiant, Leslie G. (1979). “The Complexity of Enumeration and Reliability Problems”. In: *SIAM Journal of Computing* 8.3, pages 410–421. DOI: `10.1137/0208032` (cited on pages 40, 166, 167).

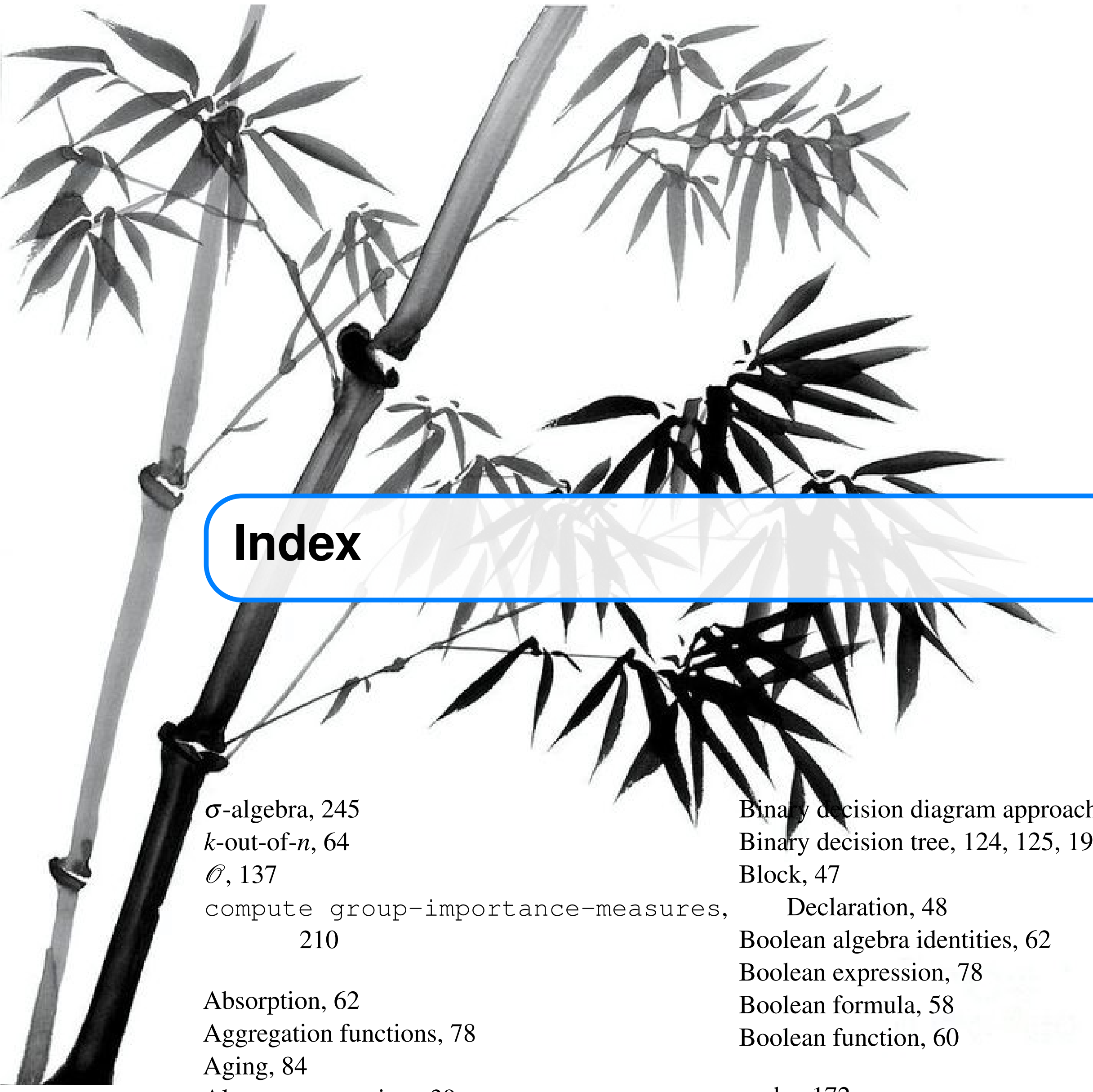
Vaurio, Jussi K. (2010). “Ideas and Developments of Importance Measures and Fault Tree Techniques for Reliability and Risk Analysis”. In: *Reliability Engineering and System Safety* 95.2, pages 95–107 (cited on page 196).

Voirin, Jean-Luc (June 2008). “Method and Tools for Constrained System Architecting”. In: *Proceedings 18th Annual International Symposium of the International Council on Systems Engineering (INCOSE 2008)*. Utrecht, The Netherlands: Curran Associates, Inc., pages 775–789. ISBN: 978-1605604473 (cited on page 122).

Wall, I. B. and Daniel H. Worledge (1996). “Some perspectives on risk importance measures”. In: *Proceedings of the international conference on Probabilistic Safety Assessment, PSA’96,* *Moving Toward Risk Based Regulation*. Park City, Utah, USA: American Nuclear Society Inc., pages 203–207. ISBN: 0-89448-621-7 (cited on page 208).

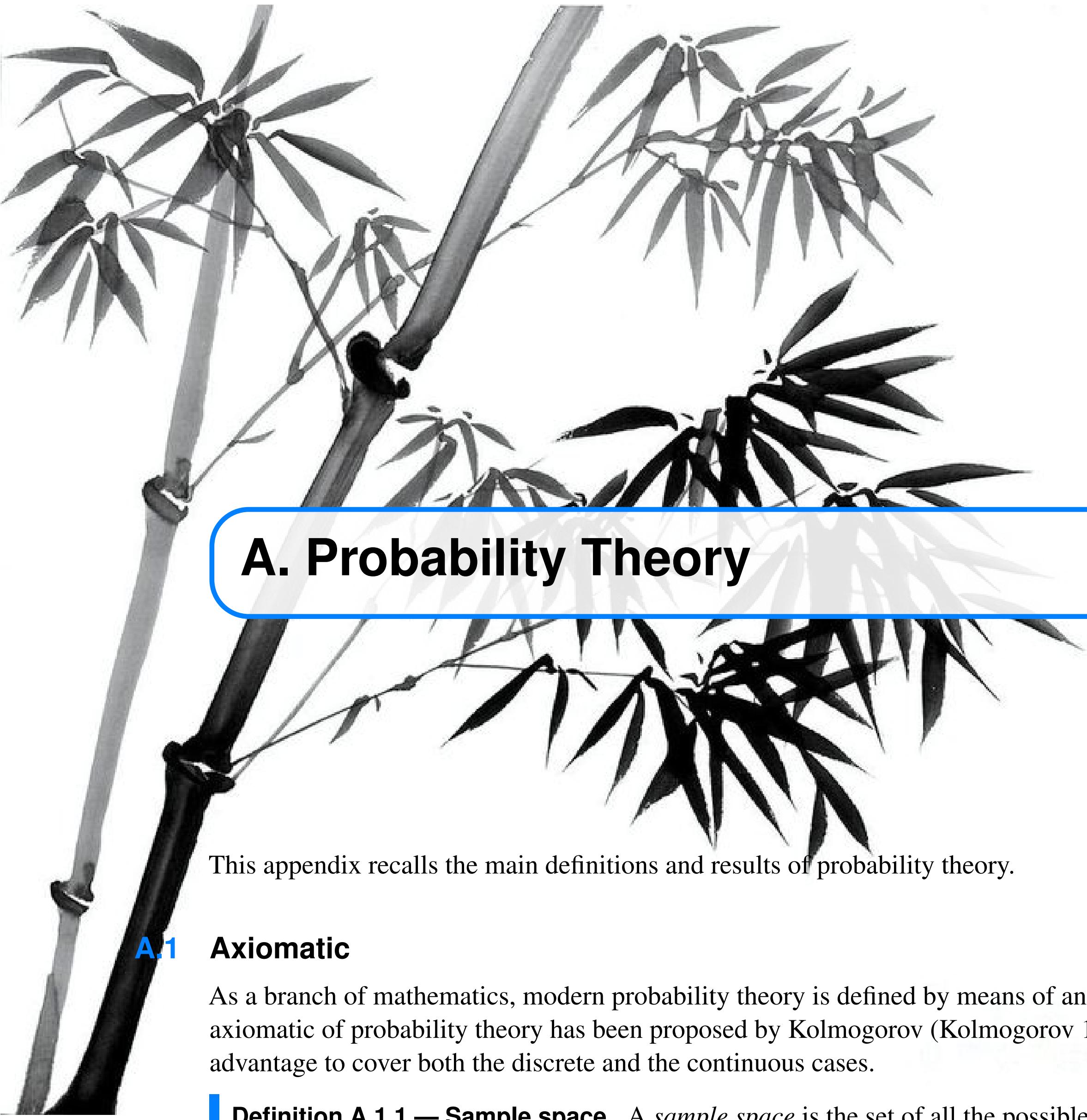
Wegener, Ingo (2000). *Branching Programs and Binary Decision Diagrams - Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications. Philadelphia, PA, United States: SIAM. ISBN: 0-89871-458-3 (cited on page 176).

Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Upper Saddle River, New Jersey 07458, USA: Prentice-Hall. ISBN: 978-0130224187 (cited on page 110).



Index

IV Appendix



A. Probability Theory

This appendix recalls the main definitions and results of probability theory.

A.1 Axiomatic

As a branch of mathematics, modern probability theory is defined by means of an axiomatic. The axiomatic of probability theory has been proposed by Kolmogorov (Kolmogorov 1933). It has the advantage to cover both the discrete and the continuous cases.

Definition A.1.1 — Sample space. A *sample space* is the set of all the possible outcomes of a non-deterministic experiment. An experiment is *deterministic* if it gives always the same result in the same condition and *non-deterministic* otherwise.

In probability theory, the sample space is usually called Ω.

Definition A.1.2 — Event. An *event* over a sample space Ω is a subset of Ω. An event gather all possible outcomes of the experience that fulfill a certain property.

For the probability theory to be well defined, the set $\mathscr{A}$ of events should have a particular mathematical structure, namely it should be a Borel σ-algebra, or σ-algebra for short.

Definition A.1.3 — σ-algebra. Let Ω be a set. A σ*-algebra* over Ω is a set $\mathscr{A} \subseteq 2^\Omega$, where 2^Ω denotes the *power set*, i.e. the set of subsets of $\mathscr{A}$, such that:

$\mathscr{A} \neq \emptyset$	$\mathscr{A}$ is not empty.
$\forall A \in \mathscr{A}, \Omega \setminus A \in \mathscr{A}$	$\mathscr{A}$ is closed by complementation.
If $\forall n \in \mathbb{N}$, $B_n \in \mathscr{A}$, then $\bigcup_{n\in\mathbb{N}} B_n \in \mathscr{A}$	$\mathscr{A}$ is closed by countable union.

The above definition implies that:
- The empty event $\emptyset$ and the total event Ω belong to $\mathscr{A}$.
- $\mathscr{A}$ is closed by countable intersection.

The pair $(\Omega, \mathscr{A})$ is a probabilisable space, i.e. it is possible to define a probability measure on it.

Definition A.1.4 — Probability measure. Let Ω be a set and $\mathscr{A}$ be a σ-algebra over Ω. A *probability measure* p is a function from $\mathscr{A}$ to the real interval $[0,1]$ such that:

1. $\forall A \in \mathscr{A}, 0 \leq p(A) \leq 1$ (*positivity*).
2. $p(\Omega) = 1$ (*unitary mass*).
3. If $\forall n \in \mathbb{N}$, $A_n \in \mathscr{A}$ and if moreover $\forall i, j \in \mathbb{N}$, $i \neq j$, $A_i \cap A_j = \emptyset$, then $p\left(\bigcup_{n\in\mathbb{N}} A_n\right) = \sum_{n\in\mathbb{N}} p(A_n)$ (*additivity*).

A.2 Additional Definitions and Properties

The above definition induces a number of well-known properties.

Proposition A.1 — Basic properties of probability measures. Let p be a probability measure over the space $(\Omega, \mathscr{A})$. Then, the following equalities hold for all $A, B \in \mathscr{A}$.

$$\begin{aligned} p(\emptyset) &= 0 \\ p(\Omega \setminus A) &= 1 - p(A) \\ p(A \cup B) &= p(A) + p(B) - p(A \cap B) \end{aligned}$$

The above third equality is extensible to finite unions of events via the so-called Sylvester-Poincaré development.

Proposition A.2 — Sylvester-Poincaré development. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A_1, A_2, \ldots A_n \in \mathscr{A}$. Then, the following equality holds.

$$p\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{k=1}^{n} \left((-1)^{k-1} \sum_{1 \leq i_1 < i_2 < \ldots < i_n \leq n} p(A_{i_1} \cap \ldots \cap A_{i_k}) \right)$$

The notion of independence plays an important role in probabilistic risk analysis. It is defined as follows.

Definition A.2.1 — Independent events. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$. Then, A and B are *independent* if $p(A \cap B) = p(A) \times p(B)$.

The notion of conditional probability captures the idea of measuring the probability of occurrence of an event, given that another event occurred

Definition A.2.2 — Conditional probability. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$ such that $p(B) \neq 0$. The *conditional probability* of A given B, denoted as $p(A \mid B)$, is defined as follows.

$$p(A \mid B) \overset{def}{=} \frac{p(A \cap B)}{p(B)}$$

It follows immediately from the definitions that if A and B are independent events, the following equality holds.

$$p(A \mid B) = p(A) \tag{A.1}$$

We shall conclude this section by the very useful Bayes's theorem, also called theorem about the probability of causes.

Theorem A.3 — Bayes's theorem. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$. Then the following equality holds.

$$p(A \mid B) \quad = \quad \frac{p(B \mid A) \times p(A)}{p(B)}$$

A.3 Random Variables

Often, some numerical value calculated from the result of a non-deterministic experiment is more interesting than the experiment itself. These numerical values are called random variables. We shall introduce here only real-valued random variables, but the notion of random variables applies to any measurable set.

Definition A.3.1 — Random variable. Let p be a probability measure over the space $(\Omega, \mathscr{A})$. A (real-valued) *random variable* X is a function from Ω into $\mathbb{R}$ such that:

$$\forall r \in \mathbb{R}, \{\omega \in \Omega : X(\omega) \leq r\} \in \mathscr{A}$$

The cumulative distribution function of a random variable is the probability that this random variable takes a value less or equal to a certain threshold.

Definition A.3.2 — Cumulative distribution function. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let X be a random variable built over p. The *cumulative distribution function* of X is the function F_X from $\mathbb{R}$ into $[0,1]$ defined as follows (for all $r \in \mathbb{R}$).

$$F_X(r) \quad \overset{def}{=} \quad p(\{\omega \in \Omega : X(\omega) \leq r\})$$

Intuitively, the *expected value* a random variable X, also called the *expectation* of X, is the long-run average value of repetitions of the same experiment X represents.

In the finite or denumerable case, this is formalized as follows.

Definition A.3.3 — Expected value (finite or denumerable case). Let X be a random variable with a countable set of finite outcomes $x_1, x_2, \ldots,$ occurring with probabilities p_1, p_2, $\ldots$, respectively, such that the infinite sum $\sum_{i=1}^{\infty} |x_i|\, p_i$ converges. The *expected value* of X, denoted $E[X]$, is defined as following series.

$$E(X) \quad \overset{def}{=} \quad \sum_{i=1}^{\infty} x_i \times p_i$$

In the infinite uncountable case, the formal definition is as follows.

Definition A.3.4 — Expected value (uncountable case). Let X be a random variable whose cumulative distribution function admits a density $f(x)$, then the *expected value* of X is defined as the following Lebesgue integral.

$$E(X) \quad \overset{def}{=} \quad \int_{\mathbb{R}} x f(x)\, dx$$

Here follows a basic property of expected value.

Proposition A.4 — Basic property of expected value. Let X and Y be two random variables and $c \in \mathbb{R}$ be a constant. Then,

$$\begin{aligned} E[X+Y] &= E[X]+E[Y] \\ E[cX] &= cE[X] \end{aligned}$$

It is often of interest to know how closely a distribution is packed around its expected value. The variance provides such a measure.

Definition A.3.5 — Variance. Let X be a random variable. The *variance* of X, denoted $\mathrm{Var}(X)$, is the expectation of the squared deviation of X from its expected value.

$$\mathrm{Var}(X) \overset{def}{=} E\left[(X-E[X])^2\right]$$

The standard deviation has a more direct interpretation than the variance because it is in the same units as the random variable. It is defined as follows.

Definition A.3.6 — Standard deviation. Let X be a random variable. The *standard-deviation* of a random variable X, denoted $\sigma(X)$, if the square root of its variance.

$$\sigma(X) \overset{def}{=} \sqrt{\mathrm{Var}(X)}$$

A.4 Laws of Large Numbers and Theorem Central Limit

A.4.1 Laws of large numbers

The frequentist interpretation of probability states that if an experiment is repeated a large number of times under the same conditions and independently, then the relative frequency with which an event E occurs and the probability of that event E should be approximately the same.

A mathematical formulation of this interpretation is the law of large numbers, which exists under two forms: the weak law and the strong law.

The weak law of large numbers states that the sample average converges in probability towards the expected value.

Theorem A.5 — Weak law of large numbers. Let $X_1, X_2 \ldots$ a denumerable family of random variables identically distributed with expected value $E(X_1) = E(X_2) = \ldots = \mu$. Let $\overline{X}_n$ be the average of the sample made of the n first variables, i.e.

$$\overline{X}_n \overset{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

Then, for any positive number ε,

$$\lim_{n\to\infty} Pr\left(\left|\overline{X}_n - \mu\right| > \varepsilon\right) = 0$$

The strong law of large numbers states that the sample average converges almost surely to the expected value.

Theorem A.6 — Strong law of large numbers. Let $X_1, X_2 \ldots$ a denumerable family of random variables identically distributed with expected value $E(X_1) = E(X_2) = \ldots = \mu$. Let $\overline{X}_n$ be the average of the sample made of the n first variables, i.e.

$$\overline{X}_n \overset{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

Then,

$$\lim_{n\to\infty} Pr\left(\overline{X}_n = \mu\right) = 1$$

The weak law states that for a specified large n, the average of the sample $\overline{X}_n$ is likely to be near μ. Thus, it leaves open the possibility that $\left|\overline{X}_n - \mu\right| > \varepsilon$ happens an infinite number of times, although at infrequent intervals.

The strong law shows that this almost surely will not occur. In particular, it implies that with probability 1, for any $\varepsilon > 0$, there exists a n_0 such that for all $n > n_0$, $\left|\overline{X}_n - \mu\right| < \varepsilon$ holds.

There are special cases, for which the weak law is verified but not the strong one.

A.4.2 Theorem central limit

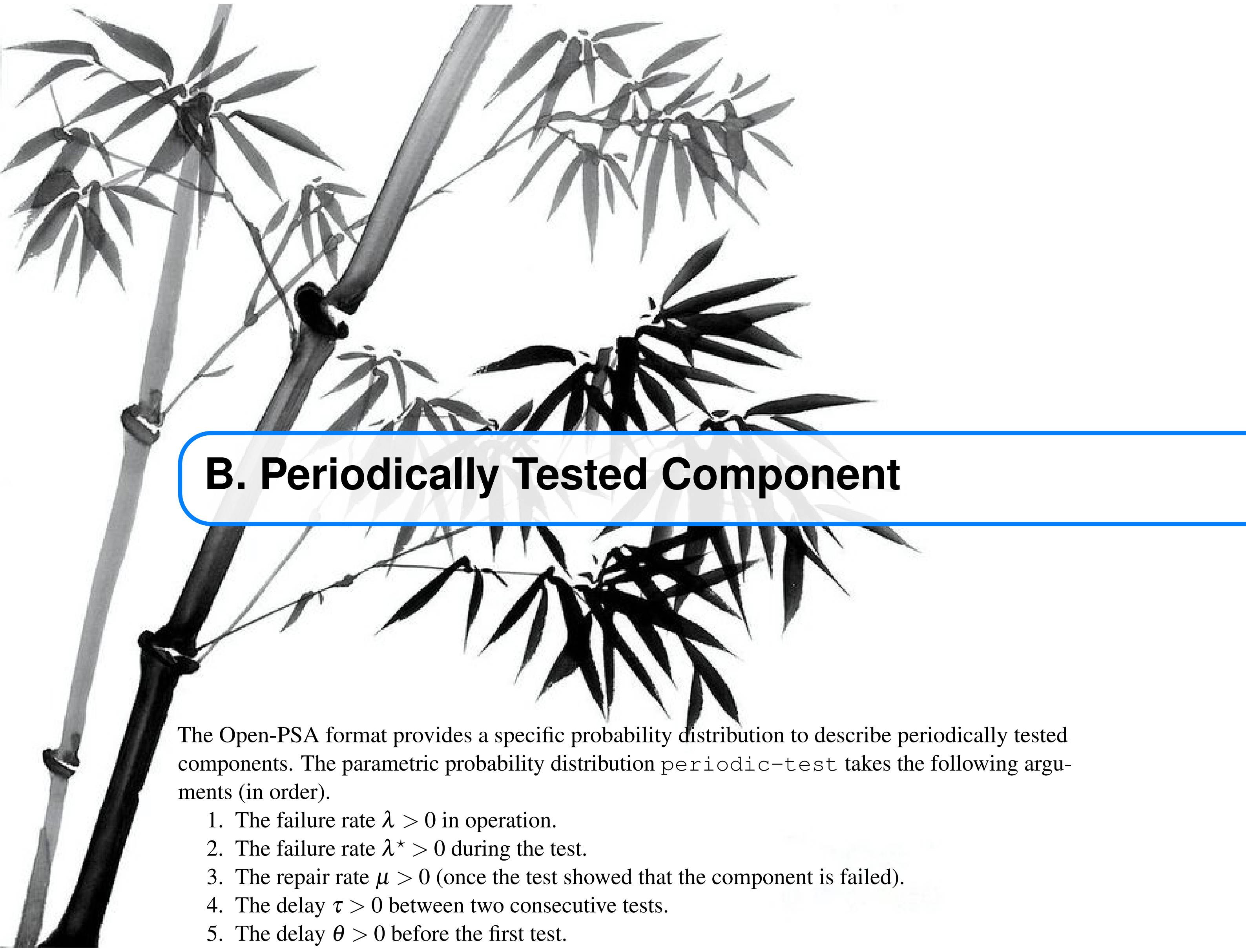
The central limit theorem states that, in some situations, when independent random variables are added, their properly normalized sum tends toward a normal distribution even if the original variables themselves are not normally distributed. This theorem plays a central role in probability theory because it implies that probabilistic and statistical methods that work for normal distributions can be applicable to many problems involving other types of distributions.

Theorem A.7 — Theorem Central Limit. Let $X_1,\ldots,X_n$ be independent random variables having a common distribution with expectation μ and variance σ^2. Let $\overline{X}_n$ and Z_n defined as follows.

$$\overline{X}_n \overset{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$
$$Z_n \overset{def}{=} \frac{\overline{X}_n - \mu}{n/\sqrt{n}}$$

Let $\Phi(z)$ be the distribution function of the normal law $\mathcal{N}(0,1)$, i.e. the normal law of mean 0 and variance 1. Then for all $z \in \mathbb{R}$,

$$\lim_{n\to\infty} Pr\left(Z_n \leq z\right) = \Phi(z)$$



B. Periodically Tested Component

The Open-PSA format provides a specific probability distribution to describe periodically tested components. The parametric probability distribution `periodic-test` takes the following arguments (in order).

1. The failure rate $\lambda > 0$ in operation.
2. The failure rate $\lambda^\star > 0$ during the test.
3. The repair rate $\mu > 0$ (once the test showed that the component is failed).
4. The delay $\tau > 0$ between two consecutive tests.
5. The delay $\theta > 0$ before the first test.
6. The probability of failure $\gamma \in [0,1]$ due to the (beginning of the) test.
7. The duration $\pi > 0$ of the test.
8. The indicator x of the component availability during the test (1 available, 0 unavailable).
9. The test covering $\sigma \in [0,1]$, i.e. the probability that the test detects the failure, if any.
10. The probability $\omega \in [0,1]$ that the component is badly restarted after a test or a repair.

Figure B.1 illustrates the meaning of the parameters τ, θ and π.

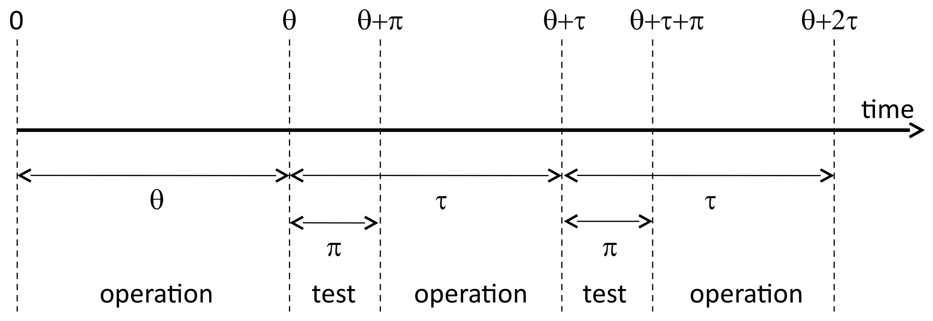


Figure B.1: Meaning of parameters τ, θ and π of the probability distribution `periodic-test`.

There are thus 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.

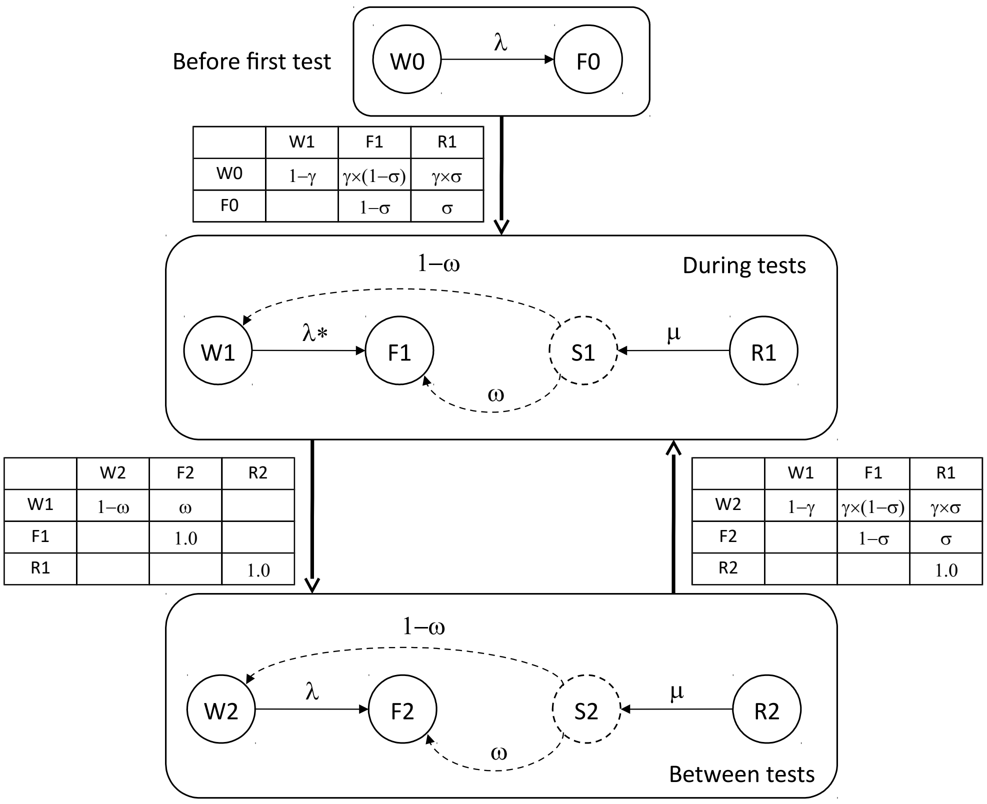
The second phase is the test phase. It spreads from times $\theta + n \cdot (\tau + \pi)$ to $\theta + n \cdot (\tau + \pi) + \pi$, where n is a any positive integer.

The third phase is the operation phase. It spreads from times $\theta + n \cdot (\tau + \pi) + \pi$ from $\theta + (n+1) \cdot (\tau + \pi)$.

The Markov chains for these three phases as well as the transitions matrices between them are given in Figure B.2. `Wi`'s, `Fi`'s, `Ri`'s states correspond respectively to states where the component is working, failed and in repair. Dashed lines correspond to immediate transitions. In the first phase, the distribution is a simple exponential law of parameter λ. The test phase and the operation phase differ only with respect to the failure rate of the component (which is $\lambda^{\star}$ for the former and λ for the latter). In both phases, there is a probability ω to restart badly the component after a repair. Hence the immediate state `Si` ($i = 1, 2$) and the two immediate transitions leaving this state. Transition matrices describe the transitions between phases. At the beginning of the test, there are three cases:

- The component is working. In this case, there is a probability $1 - \gamma$ to break it (due to the test) and then a probability σ not to detect this failure.
- The component is failed. In this case, there is the same probability σ not to detect this failure. Finally,
- The component was under repair; in this case it remains so.

At the end of the of the test, there is a probability ω not to restart the component correctly.

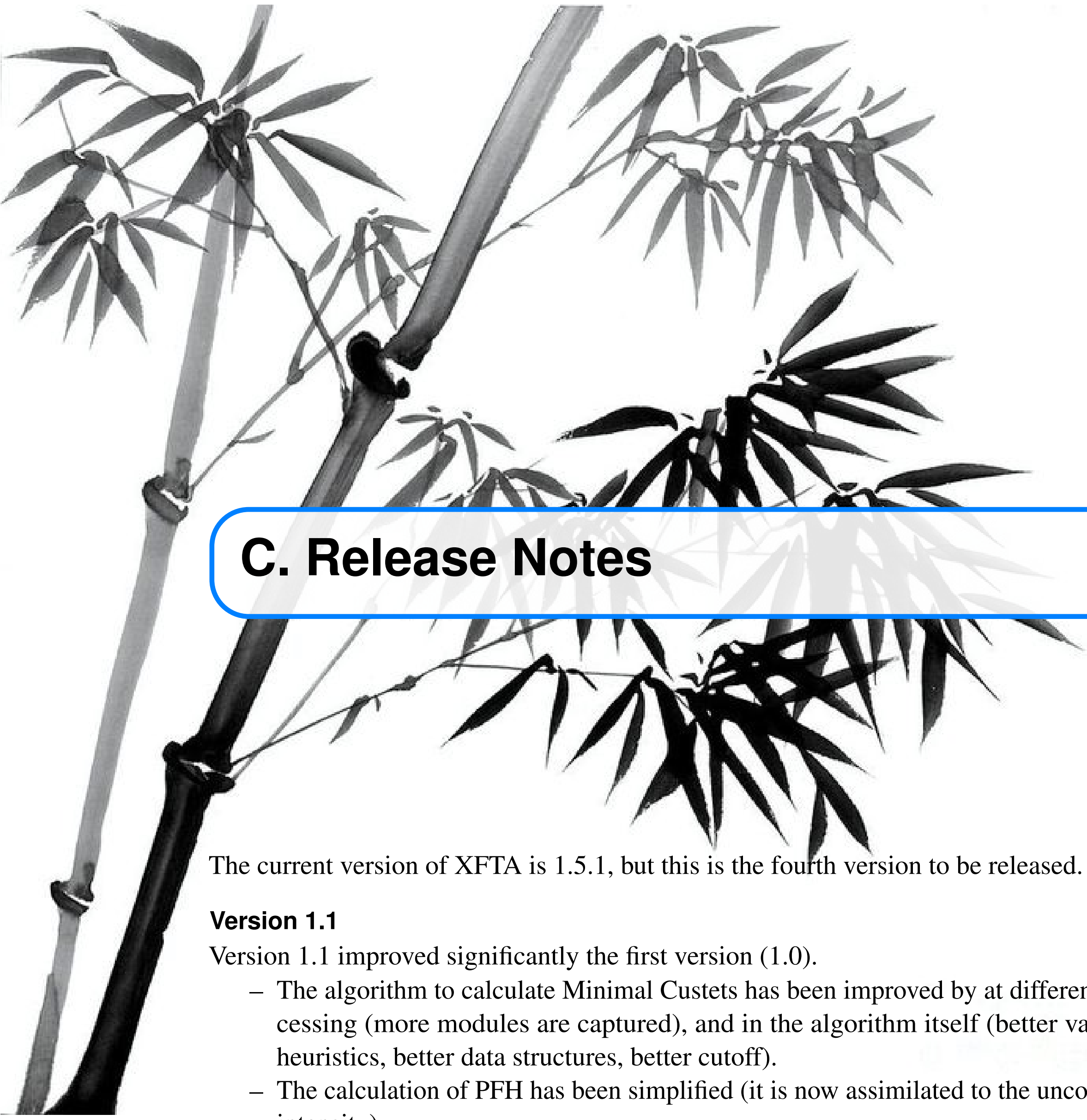


	W1	F1	R1
W0	$1-\gamma$	$\gamma\times(1-\sigma)$	$\gamma\times\sigma$
F0		$1-\sigma$	σ

	W2	F2	R2
W1	$1-\omega$	ω	
F1		1.0	
R1			1.0

	W1	F1	R1
W2	$1-\gamma$	$\gamma\times(1-\sigma)$	$\gamma\times\sigma$
F2		$1-\sigma$	σ
R2			1.0

Figure B.2: Multiphase Markov chain for the probability distribution `periodic-test`.



C. Release Notes

The current version of XFTA is 1.5.1, but this is the fourth version to be released.

Version 1.1

Version 1.1 improved significantly the first version (1.0).

– The algorithm to calculate Minimal Custets has been improved by at different levels: preprocessing (more modules are captured), and in the algorithm itself (better variable selection heuristics, better data structures, better cutoff).
– The calculation of PFH has been simplified (it is now assimilated to the unconditional failure intensity).
– A few bugs have been fixed.

Versions 1.2, 1.3.1 and 1.3.2

From version 1.2 to version 1.3.2, we made only minor evolutions and bug fixes, at least from a user perspective (internally, we performed a very important refactoring of the code).

Versions 1.4.X

Versions 1.4.X were experimental. They have not been released.

Version 1.5.1

Version 1.5.1 is a major release. An important refactoring has been performed. Moreover:

– The license has been modified.
– The present manual (which is more complete than previous versions) has been written.
– The so-called mincut-upper-bound and pivotal-upper-bound methods for the calculations of probabilistic indicators have been introduced.
– The calculation algorithm for the marginal importance factor has been modified.
– The way to specify dates (mission times) at which calculations are performed has been made more flexible and general.
– Parametric probability distributions `exponential`, `GLM` and `Weibull` can be used with or without specifying the mission time. If the mission time is not specified, the default mission time is used. These distributions can thus now be given with respectively 1 or 2, 3 or

4 and 2 or 3 arguments.

– The mission time should not be specified with the parametric probability distribution `periodic-test`. The default mission-time is systematically used. This distribution can thus be used with 3, 4 or 10 arguments.

The AltaRica Association

This book is a companion for the XFTA software. Simply put, XFTA is a highly efficient calculation engine for fault trees, the most widely used method to assess the risk in industrial systems such as nuclear power plants, airplanes, oil and gas facilities, chemical plant, cars, trains…

XFTA is however much more than that.

First, XFTA provides a full-fledged object-oriented language to design models: S2ML+SBE. S2ML+SBE is the combination of S2ML, which stands for system structure modeling language and SBE, which stands for stochastic Boolean equations. Systems of stochastic Boolean equations are the underlying mathematical framework of fault trees and reliability block diagrams. S2ML is a coherent and versatile set of object-oriented constructs that help to design and to structure models. XFTA instantiates thus the model-based safety assessment paradigm to the specific case of systems of stochastic Boolean equations.

Second, XFTA implements state-of-the-art algorithms and data structures to assess fault trees, making it probably the most efficient calculation engine available as of today. Namely, XFTA implements both a highly efficient direct algorithm to extract minimal cutsets and the whole binary decision diagram technology. Concretely, XFTA provides commands to calculate the most widely used probabilistic risk indicators, including the top event probability, importance measures, sensitivity analyses, time-dependent analyses, and safety integrity levels.

This book introduces the mathematical and algorithmic frameworks of underlying XFTA calculations. It provides also a complete description of XFTA commands, as well as numerous examples of application.

Antoine Rauzy is currently professor at the Norwegian University of Science and Technology (NTNU, Trondheim, Norway). He is also the head of the chair Blériot-Fabre, sponsored by the group SAFRAN, at CentraleSupélec (Paris, France). During his career, he moved back and forth from academia to industry, being notably senior researcher at French National Centre for Scientific Research (CNRS), associate professor at Universities of Bordeaux and Marseille, professor at Ecole Polytechnique and Ecole Centrale Paris (now CentraleSupélec), CEO of the start-up company ARBoost Technologies, and director of the R&D department on Systems Engineering at Dassault Systemes (largest French software editor).

Prof. Antoine B. Rauzy started his career in computer science. He works in the reliability engineering field for more than 20 years. He extended his research topics to systems engineering more recently. He published over 200 articles in international scientific conferences and journals. He is on the advisory boards of several international conferences and journals and is regularly invited to deliver keynote talks in international conferences.

He renewed mathematical foundations and designed state-of-the-art algorithms of probabilistic safety assessment. He developed safety assessment software that are daily used in industry (Aralia, XFTA, MarkXPR). He is also the main designer of the AltaRica modeling language and is a prominent contributor to model-based approach in reliability engineering. He managed numerous collaborations between academia and industry, in Europe, in the USA and in Japan, and has been the adviser of fifteen PhD theses.