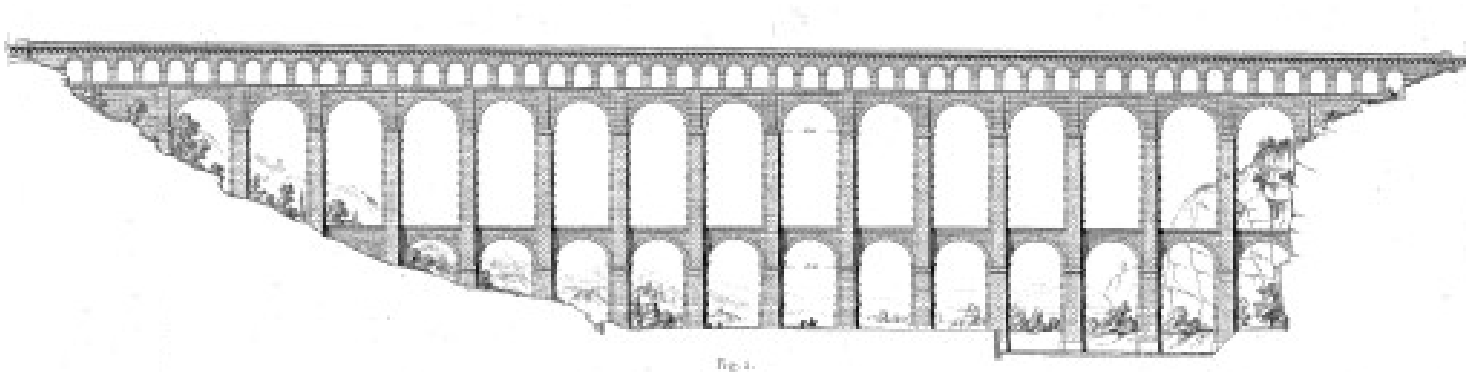
Model-Based Reliability Engineering

An Introduction from First Principles

Antoine B. Rauzy

PUBLISHED BY THE ALTARICA ASSOCIATION
ALTARICA-ASSOCIATION.ORG

First printing, January 2022 ISBN: 978-82-692273-2-1



Contents

Le Temps des Cerises

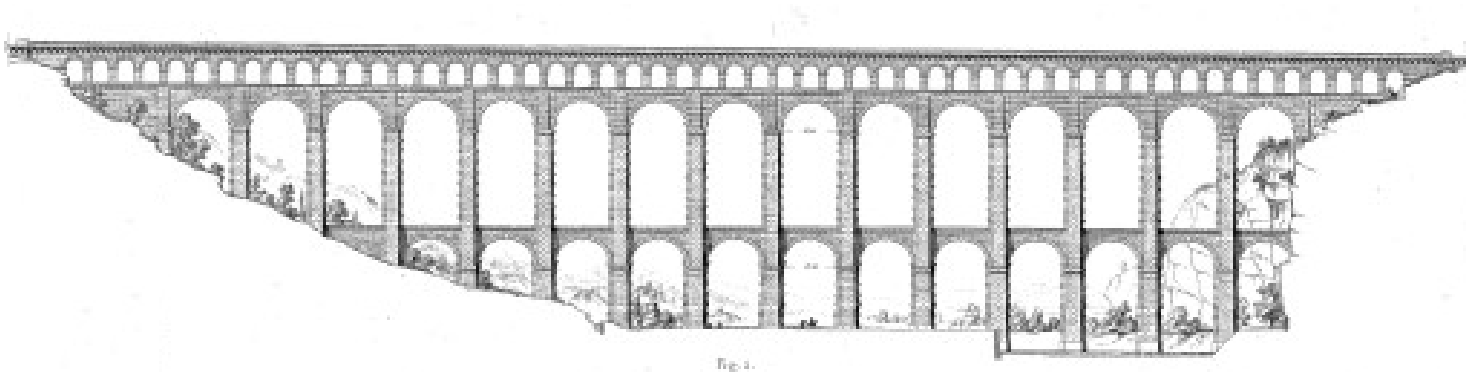
Quand nous chanterons le temps des cerises
Et gais rossignols et merles moqueurs
Seront tous en fête
Les belles auront la folie en tête
Et les amoureux du soleil au cœur
Quand nous chanterons le temps des cerises
Sifflera bien mieux le merle moqueur

Mais il est bien court le temps des cerises
Où l'on s'en va deux cueillir en rêvant
Des pendants d'oreilles
Cerises d'amour aux robes pareilles
Tombant sous la feuille en gouttes de sang
Mais il est bien court le temps des cerises
Pendants de corail qu'on cueille en rêvant

Quand vous en serez au temps des cerises
Si vous avez peur des chagrins d'amour
Évitez les belles
Moi qui ne crains pas les peines cruelles
Je ne vivrai pas sans souffrir un jour
Quand vous en serez au temps des cerises
Vous aurez aussi des chagrins d'amour

J'aimerai toujours le temps des cerises
C'est de ce temps-là que je garde au coeur
Une plaie ouverte
Et Dame Fortune, en m'étant offerte
Ne saura jamais calmer ma douleur
J'aimerai toujours le temps des cerises
Et le souvenir que je garde au coeur

Jean-Baptiste Clément



Preface

This book is an introduction to and a state-of-the-art of model-based reliability engineering. It is intended for students, engineers and researchers in all engineering disciplines interested in advanced developments of this domain. It results of more than twenty years of the author's industrial experience, academic researches, and teaching experience first at Ecole Centrale de Paris (Paris, France), then at the Norwegian University of Science and Technology (Trondheim, Norway).

Model-based reliability engineering is an emerging engineering domain at the confluence of two disciplines: systems engineering on the one hand, reliability engineering on the other hand.

Systems engineering aims at improving the design and the management of technical and socio-technical systems through their life-cycle (Walden et al. 2015). The systems engineering process covers actually two main activities. First, the analysis of the system under consideration from a holistic point of view and at a high level of abstraction. This activity aims at better understanding needs and constraints in order to support decision making. Second, the coordination of the other engineering disciplines contributing to the design and operation of the system under consideration. With that respect, the systems engineer plays in system design and management a role similar to the role of the architect in civil engineering.

The complexity of technical systems is steadily increasing. Consequently, the complexity of processes by which these systems are designed, produced, operated and eventually decommissioned is also steadily increasing. To tackle this complexity, the different engineering disciplines (mechanics, thermic, electric and electronic, software engineering...) rely more and more on models. Each complex system comes with dozens if not hundreds of models (Blanchard and Fabrycky 2008). This applies especially to systems engineering. We entered actually the era of *model-based systems engineering*.

Reliability engineering aims at assessing the performance of complex technical systems subject to uncertainties such as hazards, failures, operator errors and so on. It encompasses a wide range of activities, including the analysis of risks of operating the system under study, the assessment of its safety, the assessment of its expected production, or the optimization of its maintenance operations. Conversely to systems engineering, which was at its origin a document-centric activity, reliability engineering relies, from its inception, on models from which performance indicators can

be calculated (Andrews and Moss 2002).

Despite of their differences, systems engineering and reliability engineering can be seen as the two faces of the same medal. Although from different perspectives, they consider the system under study at the same level of abstraction. Systems engineering aims at determining what the system should do and should be, while reliability engineering aims at determining what can go wrong in its operation, to assess what is the likelihood and the consequences of something going wrong, and possibly how to remedy the potential problems.

Model-based reliability engineering results of a double movement. First, as already pointed out, systems engineering moves from a document-centric to a model-based set of activities, because of the steadily increasing complexity of technical systems. Second, still because of this increasing complexity, reliability engineering requires more structured models and more expressive modeling frameworks: it is not the same thing to assess the safety of a purely mechanical system and of a software intensive, highly reconfigurable one.

Models are thus an essential tool for both disciplines. They are also a means to better integrate them (Batteux, Prosvirnova, and Rauzy 2019b). They should therefore be considered as first class citizens and studied on their own.

This book aims thus not only at presenting state-of-the-art concepts, methods and tools of model-based reliability engineering, but also at putting these concepts, methods and tools in a broader perspective. It takes deliberately a foundational perspective, i.e. it studies models and modeling formalisms from first principles. It tries to give a formal status to each and every construct of these formalisms and to establish the relationships between these constructs. I believe that this approach is the only one able to capture, behind the apparent diversity of modeling formalisms, commonalities and differences between concepts at stake, and to organize these concepts in a systematic way.

Before presenting the actual content of the book, it is worth clarifying the notions of systems and models I shall deal with along these pages.

What is a system?

I shall adopt here the definition of INCOSE[1] as a working definition of the notion of system. Namely, a system is *“”*a combination of interacting elements that are organized so to achieve one or several established objectives (Walden et al. 2015). The book deals more specifically with intentionally designed technical systems, i.e. systems resulting of an engineering process. This includes typically cars, trains, aircraft, chemical plants, nuclear power plants, or significant parts of these products (like engines, gear systems, cooling systems…); I shall consider here neither natural systems (molecules, cells, trees, forests, volcanoes, rivers…), nor human systems not resulting of an explicit engineering process (cities, internet…), nor software systems although they result of an explicit engineering process. These systems are indeed extremely important and worth to study, but they are in general dealt with by means of other methods and tools that those presented here.

What is a model?

As for systems, it may seem hard, at least initially, to give a precise definition of the notion of model. At the time I am writing these lines, the English page of Wikipedia proposes no less than twenty-one high-level definitions of this term that are used in domains as varied as art, business, psychology, biology, physics and mathematical logic. In this sense, the category of models looks more and better defined by prototypical elements—a block-diagram is a model, a state-chart is a model, a system of differential equations is a model…—rather than by a predicate (would it be

[1] International Council on Systems Engineering, www.incose.org

a “fuzzy” one). This is not surprising as, as showed by Lakoff, this is the case for most of the categories of thought (Lakoff 1990).

Even if we restrict our attention to models used in systems engineering, giving a performative definition seems difficult, given the diversity of existing modeling principles, methods and tools. In the framework of this book, I shall however focus on computerized behavioral models, i.e. artifacts that can eventually be seen as arrangements of symbols (graphics, texts...) obeying a number of rules (a grammar) and that describe some aspect of the behavior of the system under study.

Understanding the whys and wherefores of the construction rules of models is of primary importance. It is however not sufficient. Models have actually no value in themselves. They are worth only in combination with the virtual experiments we perform on them and by the conclusions we can draw from these experiments. The term “*virtual experiments*” is to be taken here in a broad sense, ranging from collective brainstorming to complex calculations of indicators and advanced simulation procedures.

Models are a mediation between us and the system under consideration. They are used to perform experiments *in abstracto* and *in silico* that would be impossible to perform *in vivo*, because they would be too costly or to risky or simply because the system does not exist yet. This fantastic capacity of models does not come however for free, for at least three reasons. First, we must ensure that models are representative of the system, that they are well calibrated. Second, the design of models and virtual experiments is by no means an easy task. It results itself of an engineering process that requires its own expertise. This process has thus a cost that must balanced with the expected benefits. Third, we must ensure that the virtual experiments we want to perform are computationally feasible, which is not always the case.

In other words, we have to study not only the essence of models, but also their role in the design process and the processes by which they realize this role. The development and the deployment of model-based approach in systems engineering requires the development and the deployment of a science and an engineering of models.

How to learn to design models?

Although its content results of years of academic research and industrial experience, I wrote this book primarily for pedagogical purposes. Namely, I wanted to provide students in engineering curricula with an introduction to model-based reliability engineering. Here comes a common sense remark: to learn how to design models, one must design models. Just as for learning how to swim, to play the piano or to program, there is only one way of learning how to design model: practicing, practicing and practicing again.

Teaching by doing model-based reliability engineering faces however two difficulties.

First, apart in some very specific situations, it is not possible to deal with real case studies. Complex technical systems do not fit in a class room. Not only they are too big to be considered within the time frame of a course, but they are overloaded with irrelevant details that would blur the pedagogical message. We have thus to learn from realistic but not real case studies that aim at capturing this or that particular aspect of the problems at stake. Designing these case studies is challenging in itself. The risk is indeed to oversimplify things, letting the student the false impression that all of that is at best a bunch of academic exercises and at worst a pointless bla-bla. I tried to collect and to create as many realistic case studies as possible and to present them along these pages.

Second and more importantly, learning a particular modeling formalism, implemented in a particular modeling environment takes time. A lot of time. Not to speak about the availability of these environments on different operating systems (students are coming from all over the world and uses their own computer) and about their cost of acquisition (if they are commercial products) and maintenance (even though the tools themselves are free). As a consequence, it is not possible to

provide students with a panorama of modeling methods using available modeling environments. A unifying principle is needed. This applies probably not only to teaching, but also to industrial practice.

The S2ML+X paradigm

Here comes into the play S2ML+X paradigm (Rauzy and Haskins 2019). S2ML stands for system structure modeling language. S2ML gathers in a unified way structuring mechanisms stemmed from object-oriented and prototype-oriented programming (Batteux, Prosvirnova, and Rauzy 2018). The fundamental idea is that if modeling languages differ from their underlying mathematical frameworks (the X's) they can nevertheless share, at least to a large extent, their structuring mechanisms (S2ML) and consequently a syntax. To put it the other way round, given a X, i.e. mathematical framework, it is possible the design a rich domain specific modeling language on top of X, namely S2ML+X.

This book can be seen as a manifesto for the S2ML+X paradigm.

I decided thus, in order to illustrate the main modeling methods encountered in systems engineering, to design a series of domain specific modeling languages, all based on the S2ML+X principle, and to implement the corresponding assessment tools. Eventually, this led me to develop the AltaRica Wizard integrated modeling environment which supports two modeling languages:

- S2ML+SBE, where SBE stands for systems of Boolean equations. Systems of Boolean equations are the underlying mathematical framework to fault trees and reliability block diagrams, the two most widely used modeling techniques in reliability engineering. S2ML+SBE is the core language of XFTA, which is implements state-of-the-art assessment algorithms (Rauzy 2020).
- AltaRica 3.0, that can be seen as S2ML+GTS, where GTS stands for guarded transition systems (Batteux, Prosvirnova, and Rauzy 2019a). AltaRica 3.0 is the both the reference and the most advanced modeling language in the realm of model-based reliability engineering.

AltaRica Wizard is freely available and distributed by the non-profit AltaRica Association[2]. I strongly encourage the reader to download this environment and to install it. All of the examples, all the solutions to the exercises found in the book are distributed with the tools. It is worth, to fully understand what this book is about, to alternate readings and concrete experiments.

Organization of the book

Organizing this book has not been an easy task, as several quite different perspectives can be taken on model-based reliability engineering.

The first perspective consists in establishing a taxonomy of modeling languages according to the computational complexity of virtual experiments at stake (Rauzy 2018), and in organizing the discourse via this taxonomy. This is probably the most scientifically grounded approach, from a reliability engineering point of view. It is however quite challenging as it requires to introduce the taxonomy beforehand, while this taxonomy cannot be fully understood without concrete examples of modeling languages and virtual experiments. Moreover, it let aside systems engineering and does not emphasize enough the commonalities between modeling languages.

A second perspective consists in considering what models are used for, i.e. eventually to refer to the engineering disciplines contributing to the design of complex technical systems. With that respect, models can be classified into two categories: pragmatic models that aim primarily at supporting the communication among stakeholders and formal models that aim primarily at calculating indicators or performing simulations. The first version of my course on complex systems

[2] www.altarica-association.org

at École Centrale de Paris was more or less organized along this line. This organization was not fully satisfying however as it does not account for the mathematical differences and commonalities of modeling formalisms, nor for the computational complexity of assessment algorithms.

I decided eventually to split this book into two parts; a first part that introduces systems engineering and the fundamental concepts about models; and a second one that presents concrete modeling framework. In order to make the first part not too abstract, it includes case studies and exercises using on the modeling frameworks presented in full detailed only in the second part. Moreover, this part contains also the necessary discussion on the taxonomy of modeling languages that can only be fully understood by looking in details at the mathematical problems at stake, ... which are presented with the modeling frameworks. This means that reading this book cannot be a linear process. The reader should not hesitate to go back and forth, as her or his understanding of the domain progresses.

As explained above, Part I introduces the systems engineering and fundamental concepts about models. It is divided in four chapters.

Chapter 1 presents seven theses organizing the vision of model-based systems engineering and model-based reliability engineering I am promoting here. It can be seen an epistemic introduction to the book. Some readers may be surprised by these theses that differ significantly from the "*doxa*" of the domain. I ask these readers to go beyond their first impression and to be patient. They will understand step wisely, while progressing in the book, that these theses provide actually solid foundations for the science and engineering of models.

Chapter 2 introduces system architecture, i.e. the set of activities by which a technical system is analyzed and specified, taking a holistic point of view. System architecture relies on architecture frameworks, which aim at establishing common practices for creating, interpreting, analyzing and using architecture descriptions of systems. This chapter presents the Cube architecture framework that is used throughout the book.

Chapter 3 presents fundamental notions about models and modeling languages: those of syntax (and grammar), semantics and pragmatics. It explains why models should not be confused with diagrams, nor even with their graphical representations. It starts introducing AltaRica.

Finally, Chapter 4 presents the S2ML+X paradigm in details. It shows the power of object-oriented and prototype-oriented modeling, points out the commonalities and differences between these two approaches, their advantages and their drawbacks, and shows how to integrate them into a unique framework (S2ML). It pursues the introduction of AltaRica.

Part II is dedicated to concrete modeling environments of model-based reliability engineering. It is made of five chapters.

Chapter 5 recalls essential concepts, methods and tools of reliability engineering. It focuses more specifically on probabilistic risk and safety assessment. Although this chapter contains only well-known and accepted notions, it is important because the subsequent chapters refer extensively to these notions. It fixes also the vocabulary.

Chapter 6 presents systems of Boolean equations, which are the underlying mathematical framework of fault trees and reliability block diagrams. It introduces the language S2ML+SBE, presents the performance indicators that are usually computed from Boolean reliability models, as well as the XFTA tool (as integrated in the AltaRica Wizard environment).

Chapter 7 presents discrete event systems and more specifically guarded transition systems. As explained above, guarded transition systems are the underlying mathematical framework of the AltaRica 3.0 modeling language. Beyond, they are at the core of model-based reliability engineering. This chapter goes on the presentation of AltaRica and of the associated assessment tools.

Chapter 8 presents a collection of modeling patterns that are extremely useful to design AltaRica models. Modeling patterns are both a way to design models efficiently, but also a way

to document them. They are probably one of the most important contribution of model-based reliability engineering to both systems engineering and reliability engineering.

Finally, Chapter 9 introduces first a taxonomy of models that are looked at in the book. Then, it provides an abstract description of algorithms that are used to assess models of each class of this taxonomy. Finally, it recalls main results of computational complexity theory, gives the complexity of the described algorithms, and discusses why and how this complexity frames the whole systems engineering and reliability engineering processes.

The appendix contains the mathematical material necessary to defined formally the concepts introduced in the book. It is divided in four chapters. Appendix A recalls definitions and main properties of sets and relations. Appendix B provides a brief introduction to universal algebra. Appendix C recalls basics about probability theory. Finally, Appendix D describes the syntax and the semantics of expressions available in most of the domain specific languages used throughout this book.

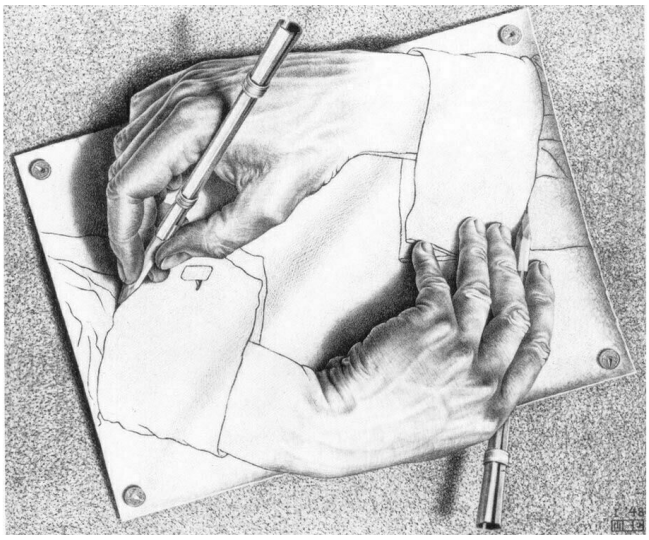
Acknowledgments

I would like to thank here all my colleagues and students who contributed, in a way or another, to make this book what it is.

Michel Batteux and Tatiana Prosvirnova deserve here a special mention. I have been lucky to collaborate with them for more than a decade. I could not have developed the theoretical and technological framework presented along these pages without the numerous exchanges we had, without their direct or indirect contributions. I am glad that Michel accepted to take over the course on model-based systems engineering in Centrale Pékin.

To finish, I would like to thank the Safran group which sponsored the chair Blériot-Fabre. Without the chair, all this fantastic scientific and pedagogical adventure would have never existed.

Antoine Rauzy
Trondheim, December 2021



About the Author

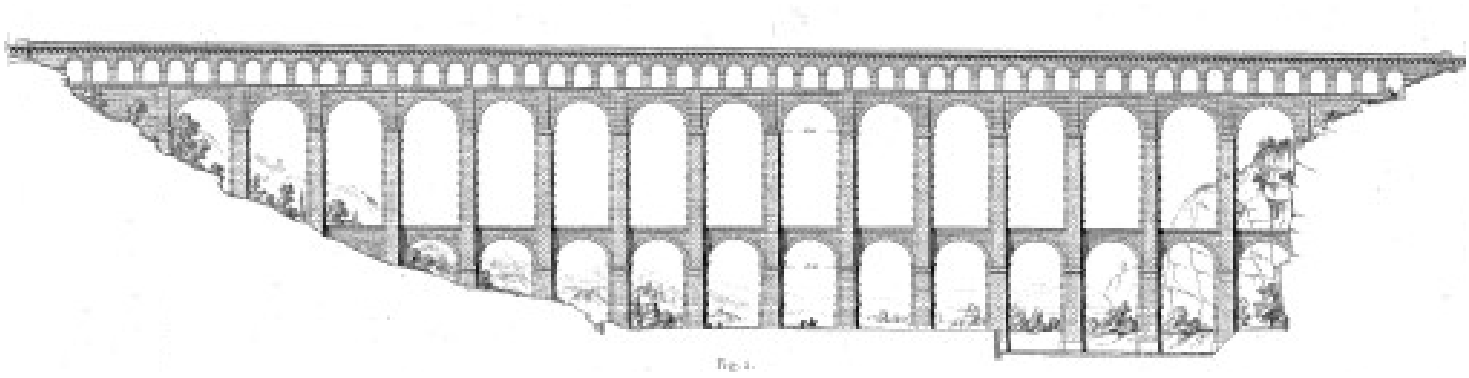
Professor Antoine B. Rauzy

Antoine B. Rauzy has currently a full professor position at Norwegian University of Science and Technology (NTNU, Trondheim, Norway), where he teaches model-based systems engineering, reliability engineering and computer science.

During his career, he moved forth and back from academia to industry, being notably researcher at French National Centrer for Scientific Research (CNRS), professor at Ecole Polytechnique and CentraleSupélec where he was the head of the chair Blériot-Fabre, sponsored by the group SAFRAN, at CentraleSupélec, CEO of the start-up company ARBoost Technologies he founded, and director of the R&D department on Systems Engineering at Dassault Systemes (largest French software editor).

Antoine B. Rauzy has a PhD and a tenure in computer science. He works in the reliability engineering and system safety field for more than 20 years. He extended his research topics to systems engineering for about 10 years. He published over 200 articles in international conferences and journals. He developed state-of-the-art software for probabilistic safety/risk analyses. He is also the main designer of the AltaRica modeling language and the president of the AltaRica Association.

Conceptual Framework



1. Synopsis

This chapter introduces the structuring ideas of the book. It presents them as a series of six theses. A first version of these theses has published in (Rauzy and Haskins 2019). I present here a slightly revised version that takes into account recent research developments.

In philosophy and rhetoric, a thesis is a statement, that may be summarized by means of a single sentence, but that results of an organized set of hypotheses, arguments and conclusions. It is thus the position of an author, a school, a doctrine or a movement on a given topics. This is exactly what I mean here. Through these theses, I aim at framing this introduction of model-based systems engineering.

This chapter may seem abstract for readers with not much experience in the field. They can come back to the ideas presented here after they acquired a better understanding of the problems at stake.

1.1 Diversity of Models

Models are working tools, not platonic ideals. A model is useful if and only if it is possible to draw useful conclusions from it, via the virtual experiments we perform on it. This has been nicely summarized by the statistician George Pellam Box, through his famous aphorism: "*All models are wrong, some are useful*" (G. P. Box 1979).

In other words, a model is not a goal that the system should reach, but a means to test hypotheses on the system. More:

Thesis 1 A model is part of a constructive and pragmatic proof.

The ultimate goal of systems engineering is actually to make a proof that there exists a system that meets a certain set of objectives (requirements). This proof is constructive in this sense that it is not made *in abstracto*, but by actually designing (or at least specifying) the system in question. The term constructive refers here to constructive mathematics, i.e. the branch of mathematics (or mathematical logic) which considers that to prove the existence of an object, one must provide an algorithm that generates an example of that object. This principle is known as the Curry–Howard

correspondence between proofs and algorithms, see e.g. (Girard, Lafont, and Taylor 1989). Of course, engineering is not mathematics and the proofs we are referring to cannot be fully formal ones. Rather, they are pragmatic, in this sense that they necessarily rely on a lot of implicit knowledge, which is shared by the stakeholders. Still, they must be as formal as possible, or to put it in other words, they must be formal although conditionally to hypotheses (axioms) coming from industrial practice. Systems engineering aims thus primarily at producing and organizing a number of arguments showing that the designed system meets its requirements. Models are formalized parts of this process.

A model is thus an abstraction of the system and is useful only because it is an abstraction. To use a well known metaphor, a model is like a map. It abstracts the system just as the map abstracts the territory. A map of the subway of Paris does not show the names of the streets and even less their altitude or the density of the population in the area. Such data would overload the map with useless information and make more complex the virtual experiment we do on a subway map, namely finding the best way to go from a point A to a point B in the city. In a short novel (Borges 1975), Jorge Luis Borges imagines a kingdom in which land surveyors are asked to design a one-to-one map of the territory: this map is extremely precise. . . but definitely useless.

The content and the level of abstraction of a model depends on what is to be proven, i.e. on objectives of virtual experiments to be performed on that model.

The meaning and practical consequences of this observation must be examined with care. First, it implies that it is not possible to design all of the models of a system within a unified framework. Each model presents a specific view on the system. The different views reflect dissimilar needs. Each engineering discipline has, and needs to have, its own modeling methodologies, formalisms and tools, dedicated to the specific purposes of that discipline. Hence our second thesis.

Thesis 2 The diversity of models is irreducible.

Consequently, models are not compositional: the set of models of a system is not a model.

With that respect, models designed by systems engineers are not different from models designed in other engineering disciplines. They reflect the vision of systems engineers and are dedicated for this very purpose. There cannot be such a thing as a unique model or even a master model of a complex system. We must live with heterogeneous models. Ensuring that these models are speaking about the same system cannot be decided *a priori*. It results necessarily of an organizational process, with all consequences this has in terms of mutual understanding of individuals and groups having different cultures, of power relationships and diverging interests intra- and inter-organizations and so on. This applies especially to the relationships between system architects and safety analysts. Although both consider systems at the same level of abstraction, their visions and purposes are different and must stay so. Improving the communication and the integration of engineering disciplines is of course of great interest, but attempts to merge their activities and their models are dangerous dreams.

1.2 Categories of Models

As I pointed out in the previous section, models are designed at different level of abstraction, for different purposes and in different modeling formalisms. Hundred of tools are available to support modeling processes. It is thus a priori difficult to design a taxonomy of modeling principles and methods. We can distinguish however two fundamental categories of models:

– Pragmatic models that aim primarily at supporting the communication among stakeholders;
– Formal models that aim primarily at calculating indicators or at performing simulations.

To explain the difference between these two categories in full details, I need to recall some notions of linguistic.

A sentence or a discourse stated in a natural language such as English can be analyzed at three levels: its *syntax*, its *semantics* and its *pragmatics* (Cruse 2011).

The syntax, or grammar, considers how words are assembled to create sentences and sentences to create the discourse. To be syntactically correct, a discourse must obey a number of rules, those of the grammar of the language the discourse is stated in.

The semantic of the discourse is the formal meaning of this discourse, i.e. what sentences literally mean. A sentence such as "the system must weight less than three kilograms" has a clear and well defined meaning. Note that a sentence may be perfectly correct from a syntactic point of view, while having no meaning. Noam Chomsky, the father of modern linguistic, illustrated this point with his famous sentence: "colorless ideas sleep furiously" (Chomsky 1957).

Now, it is often the case that a sentence or a discourse, although apparently meaningful *per se*, can only be understood in reference to the context in which it has been stated. Humor and metaphors provide countless examples of this phenomenon. Even the technical language is full of such "external" references. Safety standards, for instance, mention regularly that the likelihood of occurrence of this or that situation must be "as low as reasonably possible (ALARP)". What does this mean exactly can only be decided in the particular context where the standard is applied. Understanding a sentence or a discourse in its context is the third level of analysis, the pragmatics.

It could be argued that the pragmatics is just an extended semantics, i.e. that by adding the context to the sentence or the discourse, we would be able to apply usual rules deciding the semantics. However, this approach does not resist to the analysis. The main reason is that defining the context would require to make explicit a huge amount of implicit knowledge, which is just infeasible, at least in practice.

These explanations given, we can go back to our categories of models.

Let us first look at formal models, i.e. models that embed all what is necessary to perform the virtual experiments they are associated with, typically calculating the value of indicators or performing a simulation. These models essentially encode and organize a given type of mathematical equations, i.e. they have a well-defined semantics. This semantics associates each model with a mathematical object in a clear and unambiguous way. These models require thus making explicit every detail. They are typically designed in modeling formalisms such as State-Charts (Harel and Politi 1998), Petri nets (Murata 1989), Modelica (Fritzson 2015), or AltaRica (Batteux, Prosvirnova, and Rauzy 2019a).

The second category consists of pragmatic models, i.e. models that require referring to the context to be understood. They are essentially a communication means (Weilkiens et al. 2015). For this reason, they keep a lot of knowledge implicit and take usually a broad perspective on the system under study. They are often written in standardized notations such as BMPN (White and Miers 2008), SysML (Friedenthal, A. Moore, and Steiner 2011) or OPM (Dori 2002).

Note however that a model can be pragmatic while being written in a fully formal modeling formalism. Harel's state-charts (Harel 1987) for instance, are used both to design pragmatic models via SysML (Friedenthal, A. Moore, and Steiner 2011) and formal models in StateMate (Harel and Politi 1998).

Formal and pragmatic models do not differ in essence, but in the type of virtual experiments they are used for. There is a simple to test to distinguish them: just obfuscate your model, i.e. replace the names of elements by meaningless identifiers such as X, Y, Z... If you can still perform the virtual experiments for which you have designed your model, then it is formal, otherwise it is pragmatic.

This leads us to our third thesis.

Thesis 3 There is an epistemic gap between pragmatic and formal models.

This epistemic gap has important consequences on systems engineering processes.

First, as already said, pragmatic and formal models have radically different purposes. Both types of models are indeed useful, which means that systems engineering should rely on both.

Second, passing from pragmatic models to formal ones requires an engineering process. This process cannot be automated because it requires making explicit at least part of the implicit knowledge. Attempts to "decorate" pragmatic models with formal information in the hope of generating formal models end up into disappointing results: the source models are blurred and overloaded, losing their ability to support a seamless communication, while the generated formal models are incomplete and uselessly complex. This is the reason why, attempts to generate safety models (which are always formal) with from system architecture models (which are in most of the cases pragmatic) are essentially vain. Therefore, the central question is not to generate the former from the latter, or vice-versa, but to ensure their seamless coexistence.

Third, as pragmatic models are computerized and even written in formal modeling languages, we can design tools to process them. For instance, it is possible to trace impacts of changes in requirements on state machines describing the life-cycle of the system or on diagrams representing its functional architecture. These tools should however be designed without assigning models any pragmatic meaning. Models must be considered just as mathematical structures in which it is possible to navigate and on which some operations can be performed. Integration of engineering disciplines can be greatly eased by this kind of tools.

Pragmatic models written in standardized notations such as SysML are sometimes called "semi-formal". This term is misleading. First, either a modeling formalism has a well defined syntax and semantics, in which case it is formal, or it has not, in which case it is simply a notation. There is nothing in between. Second, it is the virtual experiments performed on the model that decide eventually whether this model is pragmatic or formal. Supporting the communication among stakeholders is a very important tasks in systems engineering. There is no need to pretend that it is semi- or nearly formal. As a communication process, it is informal, relies heavily on implicit knowledge and involves necessary gray zones.

1.3 Structure and Behavior

Models aim at reducing the complexity of systems so make them amenable for analyses. However, if the system under study is complex, we cannot expect the models of this system to be "just" simple. If it was the case, they would not capture relevant characteristics of the system. Modeling is a simplex operation in the sense of Berthoz (Berthoz 2012), i.e. it reduces the complexity without removing it totally. It is a way to tame the complexity, not a magic wand to make it vanish. Models of complex systems are thus complex, or simplex if one will. Consequently, they need to be structured, documented, configured, versioned through the life-cycle of the system.

To design a model, we need a modeling language/formalism – would this language/formalism be purely graphical – just as to design a computer program, we need a programming language. To some extent, models can be seen as declarative programs. Declarative programming is a programming paradigm in which programs describe what is to be computed but not how it should computed, conversely to imperative or functional programs that require to do both. The algorithm to execute the program is thus defined once for all and is not the responsibility of the programmer. Prolog (Colmerauer and P. Roussel 1996), constraint programming languages (van Hentenryck 1989) and database query languages such as SQL (Groff, Weinberg, and Oppel 2009) are prototypical examples of declarative programming languages.

Modeling languages/formalisms are more or less convenient to structure models, i.e. they provide more or less constructs to do so. The important point here is that these constructs can be studied on their own.

This leads us to our fourth thesis.

Thesis 4 Behaviors + structures = models

The above statement refers to the title of the famous book by Niklaus Wirth, one of the pioneers of structured programming, "Data structures + algorithms = programs" (Wirth 1976). It asserts that any modeling language is the combination of a mathematical framework to describe the behavior of the system under study and a structuring paradigm to organize the model.

The mathematical framework is for instance ordinary differential equations for Simulink and Modelica, Mealy machines for Lustre, guarded transitions systems for AltaRica... The choice of the appropriate mathematical framework for a model depends on which aspect of the system we want to study, i.e. eventually what kind of virtual experiment we want to perform on the model, e.g. multi-physic simulation with Modelica, generation of embedded controllers with Lustre or probabilistic safety analyses for AltaRica.

Structuring paradigms are to a large extent independent of the chosen mathematical framework. They can be studied on their own and applied to all mathematical frameworks.

Structuring paradigms for modeling languages are derived from those of programming languages. For the latter, object-orientation is dominant, if not hegemonic, in industrial practice. Reference monographs such as (Abadi and Cardelli 1998; Meyer 1988; Rumbaugh, Jacobson, and Booch 2005) influenced strongly, directly or indirectly, the design of most of modeling languages.

For many modeling languages, prototype-orientation seems particularly well-suited. Prototype-orientation is a variant of object-orientation in which objects can be defined either via the class/instance mechanism (as in strict object-orientation) or directly via the notion of prototype (Noble, Taivalsaari, and I. Moore 1999). Ideas behind prototype-orientation are stemmed from the important work by George Lakoff in cognitive science (Lakoff 1990). They echo also Hatchuel's CK-theory on how the knowledge is created (Hatchuel and Weill 2009). To summarize, prototypes are typically used when the system is analyzed with a top-down approach, i.e. the knowledge about the system is not stabilized yet, while the class/instance mechanism is typically used with bottom-up construction of models, i.e. when the knowledge about the system is sufficiently mature to develop extensive libraries of on-the-shelf modeling components. Reuse is also possible in the prototype/top down approach, but it is reuse of modeling patterns rather than reuse of modeling components, just as design patterns in software engineering (Gamma et al. 1994) are different from libraries of reusable classes like the Qt (Lazar and Penea 2016). With that respect, the system architecture and safety analysis processes are similar: they are essentially top-down approaches, relying on modeling patterns.

The modeling languages we shall use throughout this book implement the S2ML+X paradigm. S2ML stands for system structure modeling language (Batteux, Prosvirnova, and Rauzy 2018). S2ML is a versatile and organized set of structuring constructs stemmed from both object-orientation and prototype-orientation. It can be combined with any mathematical framework (the X) to give rise to a full modeling language.

1.4 Discrete versus Continuous Behavioral Descriptions

As discussed above, object-orientation and prototype-orientation provide good paradigms to structure models. Regarding the descriptions of behaviors of systems, system level models have to abstract away as many details as possible. When taking a holistic point of view on a system, one is actually not interested in the detailed evolution of the system, but rather in the big steps of this evolution. For instance, the description of a tank that is filled and emptied by some process discretizes typically the level of the tank into three states: within the acceptable range, too full and too empty. The physical equations ruling the evolution of the level are of no interest here. What is important to capture is that the tank is in one of three above mentioned states and that it changes of state under the occurrence of events such as operator actions, threshold crossings,

failures, reconfigurations... Physical equations can be implemented in detailed models designed by other engineering disciplines, but they would overload uselessly a system level model.

Hence our fifth thesis.

Thesis 5 At system level, behavioral descriptions are discrete.

We shall see that discrete behavioral descriptions can be split into three categories: combinatorial models, state automata and process algebras.

Many different types of virtual experiments can be performed on discrete description, from calculation and the optimization of some performance indicators to stochastic simulation and code generation. These techniques are part of the toolbox of the systems engineer.

It is sometimes hard for engineers coming from physical sciences or applied mathematics to acknowledge the discrete nature of system level behavioral descriptions. Their whole education and praxis rely on continuous mathematics and some of them, fortunately only a few, tend to equate "model" with "physical, i.e continuous, model". This whole book aim at showing that there is something between "more or less standardized drawings" and "systems of differential equations", and that the whole model-based systems engineering lies there.

1.5 Limits of Modeling

As already pointed out, a too detailed map would make useless. Similarly, a too detailed behavioral model would make virtual experiments computationally too costly and thus might put an additional strain on the capacity to check hypotheses on the system. Moreover, in early stages of system design, the behavior of the system is often known only at coarse grain. The role of the whole systems engineering process is precisely to specify the fine grain behavior from coarse grain specifications. This is a first, pragmatic argument to design models at the "right" level of detail and abstraction.

But in the case of models, there are more fundamental issues to be taken into account. Without entering into technical details, we can mention three of them:

- Some questions are undecidable, i.e. there provably exists no method to answer them.
- Some questions are intractable, i.e. there provably exists no efficient method to answer them.
- Some questions are extremely sensitive to initial conditions, i.e. there provably exists no method to answer them beyond a certain horizon.

This leads us to our sixth thesis.

Thesis 6 A model results always of a tradeoff between the accuracy of the description and the tractability of virtual experiments.

In other words, systems engineers must know the limits of modeling in order to make right modeling decisions. These issues, which are rooted in mathematical logic and computational complexity theory (Papadimitriou 1994), are examined Chapter 9.

As for the dichotomy continuous versus discrete, there is a cultural issue here. For many students and engineers, any problem must have a solution. The fantastic successes of science and technology as well as their whole education make them lean to think so: exercises given at school and at university have indeed always a solution. Heroic aphorisms such as "*they did not know it was impossible, so they did it*"[1] are here to sustain this belief. Unfortunately, in reality, most of the problems do not have solutions. Problems with a solution are the exception that proves the rule. Systems engineers must face this reality.

[1] Attributed, depending on the sources, to Mark Twain or Jean Cocteau.

1.6 Graphical Representations

According to the *doxa*, SysML is major asset of model-based systems engineering and it undoubtfully is. SysML is in essence a graphical notation (Friedenthal, A. Moore, and Steiner 2011). Beyond SysML, most of the models designed in systems engineering (and in other engineering disciplines) are authored via graphical modeling environments and many practitioners just refuse to write down a single line of code. It is therefore against the apparent evidence that we state our seventh and last thesis.

Thesis 7 Models should not be confused with their graphical representations.

This thesis may seem provocative. The reader should realize however that graphical modeling is mainly useful to describe structural parts of models and systems (Fuhrmann 2011). Some behavioral descriptions, like Petri nets or state-charts are also graphical, but authoring graphically a differential equation would be quite of a non-sense.

As soon as models get large, which is the case for most of industrial-scale systems, graphical representations cannot fit into any reasonable visualization space (computer screen or printed paper). The analyst must therefore look at them by parts. He or she must develop a global cognitive model to understand and navigate into local graphical representations. Moreover, many details are better described by a code (a text) than graphically.

In other words, models exist independently of their graphical representations. A model may have several partial graphical representations. These representations may be created dynamically or be persistent. Several alternative descriptions can be proposed for the same information. As an illustration, think to the model you are constantly navigating in: the hierarchy of folders and files in your favorite operating system. Several representations in one or two dimensions are available. Which one is the most convenient depends what you are looking at, i.e. on the virtual experiment you are performing. Except in simplistic cases, graphical representations cannot fully describe the model, even taken together.

Here again, an important lesson can be learned from software engineering. The architecture of a software is usefully represented by diagrams like those of UML (Rumbaugh, Jacobson, and Booch 2005). But programs exist independently of these representations and their code is the ultimate reference. Moreover, below a certain level of abstraction, the code gives a more compact, more precise, in a word more useful information than any drawing. At the end of the day, mankind invented writing to address the lack of precision of drawings.

This is the reason why all domain specific languages presented in this book are textual. Standardized graphical representations (close to those of SysML) are proposed to capture this or that aspect of models, but the code is the reference.

Another important lesson that can be learned from software engineering is that most of the programs are compiled (at least into sequences of instructions of a virtual machine) before being executed. In the modeling world, this translates into the difference between the model as designed and the model as assessed. The latter, for which efficient assessment algorithms exist, is obtained from the former, which is more human-friendly, by means of an automatic transformation process. This is what we call the model-as-script principle.

In summary, the code-centric approach we advocate here significant advantages. First, it clarifies the role of each and every constructs of modeling languages, how and when these constructs should be used. Second, it eases all operations on models, including verifications and transformations and therefore allow more efficient assessments. Third, it reduces the dependency to any particular tool or technology. In a word, it puts concepts first.

This sixth thesis concludes our synopsis of structuring ideas of this book. The next chapters will enter into the core of the discussion.

1.7 Further Readings

Some of the books and articles cited in this synopsis played an important role in structuring my vision of model-based systems engineering.

Books about object-oriented and prototype-oriented programming:

- Books by Meyer which capture the essence of object-orientation, e.g. (Meyer 1988).
- The rather theoretical but definitely important book by Abadi and Cardelli (Abadi and Cardelli 1998).
- The collection of articles on prototype-oriented programming (Noble, Taivalsaari, and I. Moore 1999).
- The reference book on design patterns (Gamma et al. 1994).

Books about computational complexity:

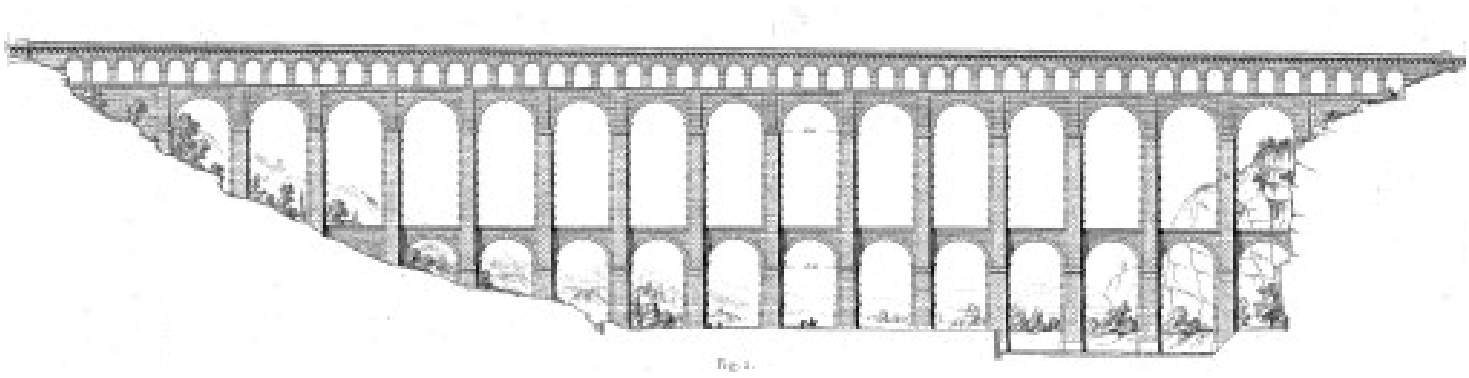
- The book Garey and Johnson on NP-completeness (Garey and Johnson 1979).
- The book by Papadimitriou which covers the domain (Papadimitriou 1994).

Books about graphical modeling and/or systems engineering:

- The reference book about UML (Rumbaugh, Jacobson, and Booch 2005).
- The reference book about SysML (Friedenthal, A. Moore, and Steiner 2011).
- The booklet by Krob on the CESAM method (Krob 2017).

Outside computer science and systems engineering, several books influenced my vision:

- The book by Lakoff about categories of thought which is undoubtedly one of the best book I ever read (Lakoff 1990).
- The book by Crozier and Friedberg which influenced deeply my understanding of organizations (Crozier and Friedberg 1980).
- The book by Hatchuel and Weill on CK-theory (Hatchuel and Weill 2009).
- The book by Berthoz on simplexity (Berthoz 2012).



2. Architecture of Systems

Key Concepts

- Systems Architecture
- Architecture frameworks
- Ontology
- Stakeholder
- Cube Architecture Framework: Sketch, Use Cases, Interface, Operation Modes, Functional Architecture, Physical Architecture
- UML, SysML
- Business Process Modeling and Notation
- Architectural patterns, redundant system, control system
- Functional chain
- Design Structure Matrix
- Needs, requirements

Before diving into model-based systems engineering, we need to recall fundamentals systems architecture. Systems architecture is the set of activities by which a technical system is analyzed and specified, taking a holistic point of view. It does not aim at providing technical solutions but rather at organizing and integrating technical solutions provided by other systems engineering disciplines.

This chapter introduces the key concepts and the vocabulary of systems architecture. We shall use these concepts each time we shall study a system either from a general or from a specific point of view, e.g. its performance, cost, safety…

The following presentation is strongly inspired by the work of Daniel Krob (Krob 2017). It does not aim at covering, even superficially, even partially, the subject of systems architecture. Rather, our objective here is to propose a pedagogical architecture framework that we can apply throughout the book. In depth justifications for the choice of this particular framework and comparison with alternative frameworks go beyond the scope of the book. We shall only mention here and there references to the relevant literature, when needed.

2.1 Introduction

For centuries, human beings designed rather complex products such as houses, bridges, vehicles and the like. Indeed, not all human beings were involved in the design process of these products. Rather, it was the affair of a small number of specialists, patiently trained since their youth by grown-ups who were themselves specialists. The knowledge and know-how hence accumulated generation after generation was for a good deal empirical, i.e. resulting from trials and errors processes. Nowadays engineering of technical systems still relies to a large extent on this incremental approach. However, the latter is no longer sufficient, for at least two categories of reasons.

First, the number of systems produced by industry as well as the pace at which new products appear on the market grew dramatically. Engineers cannot anymore specialized themselves into one type of systems and spend their whole career working on this type of systems. They constantly need to acquire quickly new concepts, methods and tools, to understand quickly the why and wherefores of an engineering domain. Consequently, informal and person to person transmission of knowledge is not sufficient anymore. One needs more formalized processes.

Second, the technical systems produced by industry grew not only in number, but also in complexity. Up to the point that not a single individual, would she or he be extremely smart, can master the whole required knowledge. The good old time of pioneers who did everything by themselves is definitely over. Nowadays, industrial projects are collective. Consequently, communication among stakeholders is a key issue. We call *stakeholder* an individual or collective actor (group or organization) who is actively or passively concerned by a decision or a project; i.e. whose interests can be affected positively or negatively following the execution (or non-execution) of the project.

Systems architecture is an emerging discipline that aims at making explicit and easy to communicate why a system is designed, for whom, which capacities it provides, which resources it needs to be operated, *et cætera*. In other words, systems architecture aims primarily at improving the communication among stakeholders.

Systems architecture relies on *architecture frameworks*, i.e. on corpuses of common practices for creating, interpreting, analyzing and using descriptions of systems within a particular domain of application or stakeholder community. Architecture frameworks rely themselves on ontologies of the domain they describe. In engineering, an *ontology* encompasses the naming and the definition of concepts, data and entities that substantiate the domain of discourse as well as of the properties and relations existing between these elements.

As the above considerations may look a bit vague and abstract to the reader, let us illustrate them on a concrete case study.

■ **Case Study 1 – Overflow Protection System.** Figure 2.1 shows a system in charge of controlling the level of liquid in the tank `T`. The liquid is entering into the tank via the pipe `source` and going out of the tank via the pipe `consumer`.

The overflow protection system is actually duplicated.

A first, regular, control line consists of the level sensor `LS1`, the controller `C` and the shutdown valve `SDV1`: if the sensor `LS1` detects that the level is above a certain threshold `tl1`, it sends a signal to the controller `C` which, if the signal is maintained sufficient long, sends the order to the shutdown valve `SDV1` to close.

The second control line consists of the level sensor `LS2`, the controller `C` and the shutdown valve `SDV2` and the discharge valve `DV`. It works in the same way as the first line, except that it is used only in case of an emergency, i.e. when the level of liquid goes above a certain threshold `tl2`. In this case, the controller sends the order to the shutdown valve `SDV2` to close and to the discharge valve `DV` to open.

■

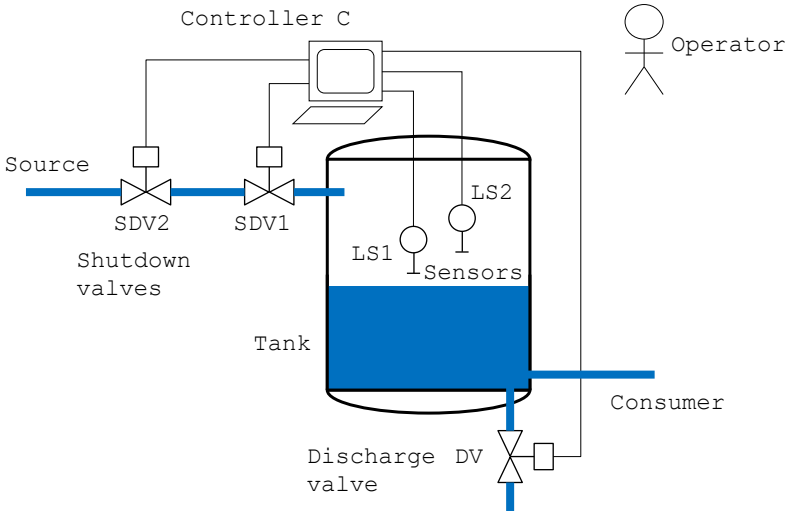


Figure 2.1: A system in charge of controlling the level of liquid in the tank.

The question is how to analyze this system, i.e. eventually to determine its architecture?

To answer this question, one should not only take into account technical aspects, but also economic, environmental and legal aspects as well as all what concerns the operation of the system, including issues regarding safety, maintenance, training of operators...

The system architecture process aims therefore at organizing logically the different point of views on the system and to monitor this organization through the whole life-cycle of the system. It is in essence a concurrent and collaborative process: different point of views are supported by different teams. Moreover, it is an iterative process: one cannot expect to find the "good" description upfront. The discovery of the "best" solutions results always from a process.

2.2 The Cube Architecture Framework

Throughout this book, we shall use the *Cube architecture framework*, hence called because it relies on the combination of six points of view on the system, like the six faces of a cube:

Sketch: When starting looking at a system, one needs a brief description of the system, a *sketch*, just like the one we gave for case study 1. This description is necessarily incomplete. It aims at making it possible to grasp at a glance the key features of the system.

Use Cases: One of the pitfalls of systems engineering is to reason too abstractly, hence producing abstract non-sense, or creating misunderstanding among stakeholders. The best way to understand how the system works (and may fail) is to write down scenarios of use. These scenarios are called *use cases*.

Interface: When analyzing a system, it is of primary importance to determine accurately what is in the system and what is out, i.e. to determine its *boundaries* and its *interfaces* with systems in its environment. In other words, the system never works in isolation. It provides services to external systems and requires in turn services from external systems to do so. The role of interface analysis is thus not only to determine the frontiers of the system, but also to characterize its interactions with external systems, to clarify the *contract* between the system and its environment.

Functional Architecture: As made explicit by the boundaries analysis, the system is designed to deliver some services to external systems. To do so, it implements some main functions or capacities. This main functions may require in turn the implementation of sub- and auxiliary functions, which themselves can be decomposed further and so on. In a word, the system has

a *functional architecture* that must be determined.

Operation Modes: Most, if not all, technical systems have not only a main operation mode, but also some subsidiary modes, e.g. shutdown, maintenance, failure... Functions and services from external systems required in these different modes may be different. It is thus of primary importance to determine the different *operation modes* of the system as well as the reasons for which the system switches from one mode to another one.

Physical Architecture: Last but not least, the system is made of a number of parts, which themselves may be made of sub-parts and so on. In other words, the system has a *physical architecture* that must be determined. The adjective "physical" must be understood here in a very broad sense: it can refer not only to physical components, but also to software and to organization resources (operators). Similarly, the physical architecture may group components according to their type, their location, the organization in charge of operating or maintaining them, or any other criteria relevant for the analysis.

The faces of the cube are logically, not chronologically, related:

- Use cases involve functions, modes and physical components. It is thus questionable to have a function, a mode or a physical component that does not show up in any use case.
- Top level functions correspond to services the system delivers to external systems. It is thus questionable to have a top level function not related to any service or vice-versa a service not related to any function.
- Each mode of operation requires a certain subset of functions. It is thus questionable to have a function not used in any mode or vice-versa a mode that requires a function not described in the functional architecture.
- Each function is implemented by a certain subset of component, at least if it is not a service that is required from an external system. It is thus questionable to have a physical component not used in any function or vice-versa a function not allocated to any physical component.
- and so on...

Consequently, studying a technical system along the cube architecture framework is by no means a linear process in which one would first deal with one face of the cube, then with a second one and so on until the six faces have been completed. Rather, when the study starts, each face of the cube contains already a number of elements. These elements may be fuzzy and inconsistent one another. The description is most probably incomplete. However, it is a starting point. In current industry practice, it is extremely rare to design a system from scratch. It is for instance often said that a brand new model of car is made of at least ninety five percent of old components.

The role system architecture study is thus to start from the initial material and to progressively make the description concise, unambiguous, coherent and complete:

- A description is *concise* if contains all relevant information but no more, i.e. if it is not overloaded by irrelevant details.
- A description is *unambiguous* if it can be understood in the same way by all stakeholders, i.e. if as few room as possible is let for subjective interpretation.
- A description is *coherent* if no two of its elements contradict one another.
- Finally, a description is *complete* is all elements necessary to design and operate the system are in there, or at least it contains references to the places where these elements can be found.

Architecting system is thus in essence an iterative process, made of a number of steps. The number of steps necessary to make eventually the description of the system concise, unambiguous, coherent and complete depends indeed on the size and the complexity of the project. Sometimes, steps and their schedule can be decided upfront, most often not. There may be also big steps, medium steps and small steps or any other convenient categorization of steps. It is highly recommended to keep track of the state of the cube at each step, in other words to have some kind of a versioning system. Version control systems used in software engineering, such as Git (Loeliger and

Mccullough 2012), can be used for this purpose.

Always bear in mind our first thesis (Chapter 1, page 11): a model is part of a proof. The aim of the system architecture process is to prove that the system under design will meet its objectives. This in turn requires to make clear what is the system under design and what are its objectives. Systems architecture must by no means be turned into a bureaucratic process. It is goal oriented, or more precisely proof oriented. Concretely, if a face of the cube is empty because it does contribute to the proof that the system you are working on will meet its objectives, just let this face empty! Moreover, ask constantly yourself: which part of the proof this element (textual description, diagram or formal model) contributes to? Do not develop elements that do not contribute to the proof.

A last but not least recommendation: in all projects, in all organization, there are power relationships among stakeholders. Drive out Spinoza's *conatus* by the door, it comes back by the window. As Crozier pointed out, the power goes to whom masters the uncertainty (Crozier and Friedberg 1980). Information is power. System architecture is by no means neutral with that respect. You have to take that into account, especially if you want to get and to keep all stakeholders on board the project. The role of the system architect is to facilitate exchanges between stakeholders, not to impose his or her vision of the system, and even less to tell other engineers how they should do their work.

The remainder of this chapter is dedicated to get more details on each face of the cube. By necessity, the exposition is linear. But the reader must bear in mind that describing each face of the cube results of a concurrent process.

Before starting, let us just remark that the sketch should be kept simple, supported by a process and instrumentation diagram or a diagram such as the one pictured in Figure 2.1, using boxes and possibly icons to represent functions or physical components and wire to represent connections and relations. Recall that, if "a picture is worth a thousand words", a diagram and a text explaining and commenting the diagram is better than the diagram alone. Diagrams are seldom self-explanatory.

2.3 Use Cases

In software and systems engineering, a *use case* is a sequence of actions or steps defining the interactions between the system under study, or parts of it, and one or more external systems. External systems, whether they are human beings or not, are often called *actors*.

Use cases are of primary importance for at least three reasons:

- First, as already pointed out, they make it possible to reason concretely about the system, and to share with stakeholders scenarios of uses.
- Second, they help to ensure that key functionalities of the system are well understood and taken into account.
- Third, they prepare the tests to will be performed in order to validate the system, once built.

Use cases are probably both one of the most, if not the most important part of model-based systems engineering. Nevertheless, they are too often neglected.

As a general rule, for each feature introduced somewhere in one of the models, there must be at least one use case involving this feature.

2.3.1 Templates

The literature on use cases is rather scarce. The reference book is the one by Cockburn (Cockburn 2000).

Cockburn argues that use cases is in essence textual. He advocates however the use of templates to write them. In his view, templates consists in meta-data that help not only to structure use cases, but also to manage them. This includes, among others, the following.

Title: Giving the use case a name helps to understand its purpose.

Level: Cockurn defines a whole hierarchy of use case levels, ranging from very high summary to very low detail (like describing what a sub-sub-sub-function does).
Preconditions: Short description of what should be verified, typically which sub-systems are functioning or not, for the use case to be possible.
Postconditions: Short description of the consequences of the use case.
Trigger: Event that triggers the use case.
Primary Actor: The main external system involved.
Story: The real thing, i.e. the text of the use case.
Extensions: Possible extensions, variants of the use case.

In case several persons are working on use cases, one may add also information on the author(s) of the use case, the date of creation, the date of last modification, the maturity and so on.

Let us illustrate that on our overflow protection system.

■ **Example 2.1 – Use case: Closing of valve `SDV1`.**

Title: Closing of valve `SDV1`.
Level: Use goal.
Preconditions: The sensor `LS1`, the controller `C` and valve `SDV1` are functioning.
Postconditions: The upstream flow is stopped.
Trigger: Too much liquid flowing into the `Tank` via the `source` pipe.
Story:

1. Too much liquid is flowing in the `Tank` via the `source` pipe.
2. The sensor `LS1` detects that the liquid goes above the threshold level `tl1` in the `Tank`.
3. The sensor `LS1` sends a signal for more than x seconds in a row to the controller `C`.
4. The controller `C` make the decision to close the valve `SDV1`.
5. The controller `C` sends to the valve `SDV1` the order to close.
6. The valve `SDV1` closes.

■

The above example looks quite innocent. It raises however a number of questions that help to understand and to specify the system, e.g.

- Is it possible that too much liquid flows "naturally" in the `Tank`?
- If too much liquid flows in the `Tank`, is the `consumer` pipe insufficient to evacuate the surplus of liquid?
- In case the liquid goes above the threshold level `tl1` in the `Tank`, how long does the controller `C` "wait" before sending the order to close to the valve `SDV1`? In other words, what is the exact value of x in the above scenario?
- Does the valve `SDV1` close completely or is it possibly sufficient to close it partially?
- Does the controller `C` warn the `Operator` in case it sends the order to close to the valve `SDV1`?
- Does the controller `C` warn the `Operator` in case of the sensor `LS1` detects that the liquid goes above the threshold level `tl1` in the `Tank`? If so, after how long? x seconds? Another value y?
- How the system is put back in normal operation after the valve `SVD1` has been closed? Who is involved? The operator? Or is this done automatically when the liquid level goes back under the threshold `tl1`?
- How long does it take to put back the system in normal operation?
- ...

The above list of questions shows that the sketch we made was far from sufficient to get an

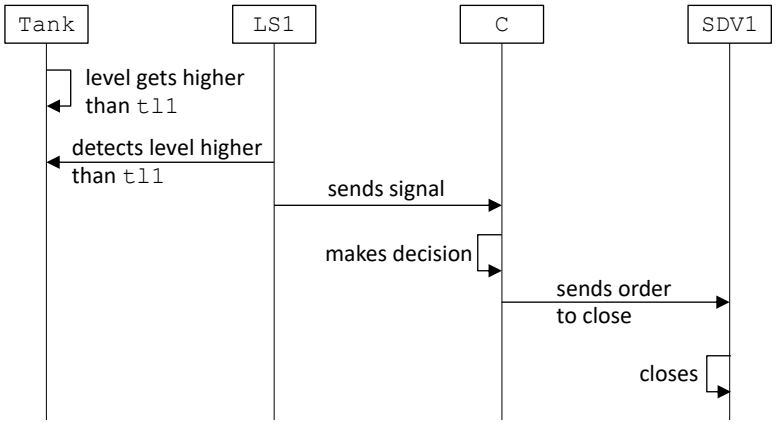


Figure 2.2: Sequence diagram representing the use case of example 2.1.

unambiguous and complete description. This is not a problem of our sketch. This simply tells us that the system is more complex that one may think at a first glance and that many features remain to be specified. The objective of writing use cases is precisely to help us in this task and to support the discussion among stakeholders.

Several other use cases are indeed necessary to have a better picture of how the overflow protection system operates. This is the subject of exercise 2.1.

2.3.2 Graphical Representations

Standardized diagrams like UML (Rumbaugh, Jacobson, and Booch 2005) and SysML (Friedenthal, A. Moore, and Steiner 2011) propose specific diagrams to represent use cases (they are actually almost the same in both cases). These diagrams are however far from convincing. Their use may be even counter-productive as they require any to be explained by some text.

Sequence diagrams, proposed by both UML and SysML, are probably more appropriate to represent use cases. A *sequence diagram* shows, as parallel vertical lines (called lifelines), different processes or objects that live simultaneously, and, as horizontal arrows, the messages (to be taken in a broad sense) exchanged between them, in the order in which they occur. This allows the specification of simple scenarios in a graphical manner. Figure 2.2 shows a sequence diagram representing the use case of example 2.1.

Figure 2.2 illustrates both the interest and drawbacks of sequence diagrams: they may help to grasp simple use cases, but reach quickly the limits of any graphical representation: when the number of actors increases or when the use gets a bit complex, the size of the diagram gets large. Consequently, it is not possible to display it onto a computer screen or to print out a page of paper at a readable scale.

Business Process Modeling and Notation (White and Miers 2008) are probably the best candidate for diagrammatic representation of use cases. They can be used in a much broader scope than the graphical representation of use cases. Moreover, they can be turned into executable scenarios.

Generally speaking, a *business process* is a structured collection of *activities*, also called *tasks*, that produces a specific service for one or more given clients. Business processes are often visualized by means of flow diagrams (flowcharts) as interleaved sequences of tasks and decision points.

The *Business Process Model and Notation*, BPMN for short, is a graphical formalism making it possible to describe the different steps of a business process as well as the exchanges of information between the stakeholders of this process.

BPMN diagram play an important role in the design of information systems:

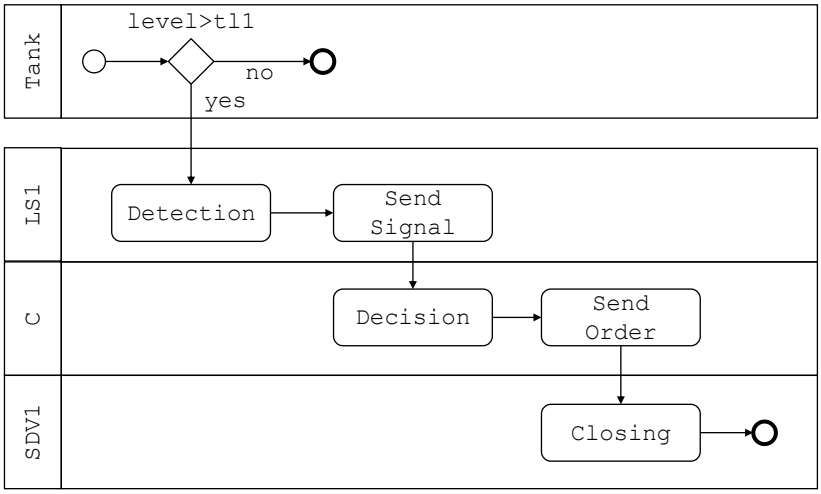


Figure 2.3: BPMN diagram representing the use case of example 2.1.

– They make it possible to describe business process in a uniform and standardized way, so that all members of an organization can understand each other with a minimum of ambiguity.
– They make it possible to create a bridge between business process and their implementation in information systems.
– The BPMN formalism is to a large extent independent of any modeling/analysis methodology of business processes.
– The BPMN formalism is an internationally accepted standard of OMG[1].

Figure 2.3 shows a possible BPMN diagram to represent the use case of example 2.1.

A BPMN diagram is made of a number of *swimlanes*, one per actor, represented by horizontal rectangles. Swim-lanes can be grouped into *pools*.

Swim-lanes contain essentially three types of objects:

– *Events* that represent points in time where something happen. Events are denoted by circles. Two types of events are of special interest: events that start a scenario (or set of scenarios) and events that end a scenario (or set of scenarios). The latter are denoted with a thicker line.
– *Activities* or *Task* that are denoted by rounded rectangles.
– *Gateways* that are denoted by diamonds. The BPMN standard defines a full menagerie of gateways to represent tests (as in our example), choices, parallel decomposition...

Interactions between objects (events, tasks and gateways), called *flows* are represented by arrows joining these objects. There are mainly two types of interactions:

– *Sequence flows*, denoted by plain arrows, that represent the order in which things happen.
– *Message flows*, denoted by dashed arrows, that represent exchanges of messages (in a broad sense) between objects.

The difference between sequence flows and message flows is that messages can take a certain time to reach their receivers, while sequences take no time (at least at the time scale one considers the system). This is the reason why in our example we did not use message flows even though the signal sent by the sensor `LS1` to the controller `C` and the order sent by the controller `C` to the valve `SDV1` are arguably messages.

This difference reflects the broader distinction between two types of process coordination:

– *Orchestration* that describes tasks performed by one actor or a group of actors sharing the same data/information (and typically belonging to the same entity). The activities of these different actors are participating to an orchestration are represented in lanes of a swimming

[1]The Object Management Group (OMG) is a computer industry standards consortium, see `www.omg.org`.

pool.
- *Choreography* which implies several separated interacting entities (e.g. client/supplier). It is then assumed that these entities interact by message exchanges but do not share their data/information. Activities are then represented in separated pools.

As it may be already clear from our example, one of the advantages of BPMN diagrams is that they make it possible to represent several scenarios, or more exactly several variants to a scenario, within the same diagram. This feature makes them extremely useful to specify systems. It remains that BPMN diagram are only graphical representation. They must be kept simple and readable as their interest stands in the communication among the stakeholders.

The reader is strongly encouraged to visit OMG web-site `www.bpmn.org` to learn more about BPMN as it proves to be a definitely useful tool in engineering and project management.

2.4 Interface Analysis

Interface analysis has two main objectives:

1. Determining what is in and what is out of the system, i.e. eventually what is in the perimeter of the system and what are the external systems the system interacts with.
2. Characterizing the interactions between the system and its environment, i.e. the external systems it interacts with.

Three methodological remarks can be made at this point.

First, determining the perimeter of the system is determining the scope of the study, and vice-versa. The perimeter of the system looks often "obvious" to engineers. They focus typically on technical parts, excluding almost always operators from the system they want to consider. There are however many cases where the operators play a central role in the system operation. In such cases, it may be wise to study the operated system, i.e. the technical system and its operators, rather than the technical system only.

Second, any technical system interact directly or indirectly with dozens of other systems. It is of primary importance, not to be overwhelmed by the amount of interactions to look at, to consider only the external systems the system interacts directly with. On the other hand, it is also necessary not to miss any external system. A paradoxical injunction. As a rule of thumb, the number of external systems a system interact with varies often between half a dozen and a dozen. If only very few external systems have been considered, there are good chances that something has been forgotten. If more than a dozen of external systems have been considered, there is probably a way to group some of them so to ease the study.

Third, the term “interactions” is to be taken in a broad sense: "dependencies" could be a reasonable alternative. The relation between the system and external systems is somehow symmetric: the system delivers a number of services to external systems, but requires to be delivered a number of services to do so. In other words, the specification of the system can be seen as a *contract*: if the system is delivered services A, B, C... then it will deliver services X, Y, Z... With that respect, the role of the boundaries analysis is to clarify the contract.

The notion of contract has been extensively studied in software engineering, notably by Bertrand Meyer (Meyer 1988; Meyer 2009).

Eventually, the system and its environment can be represented by means of an *environment diagram*.

■ **Example 2.2 – Environment diagram for the overflow protection system.** As an illustration, consider again the overflow protection system sketched in example 1. The environment diagram for this system could be as pictured in Figure 2.4.

We chose here to study the technical system only, i.e. the two sensors `LS1` and `LS2`, the controller `C` and the two valves `SDV1` and `SDV2`, without the operator. Note that we excluded the

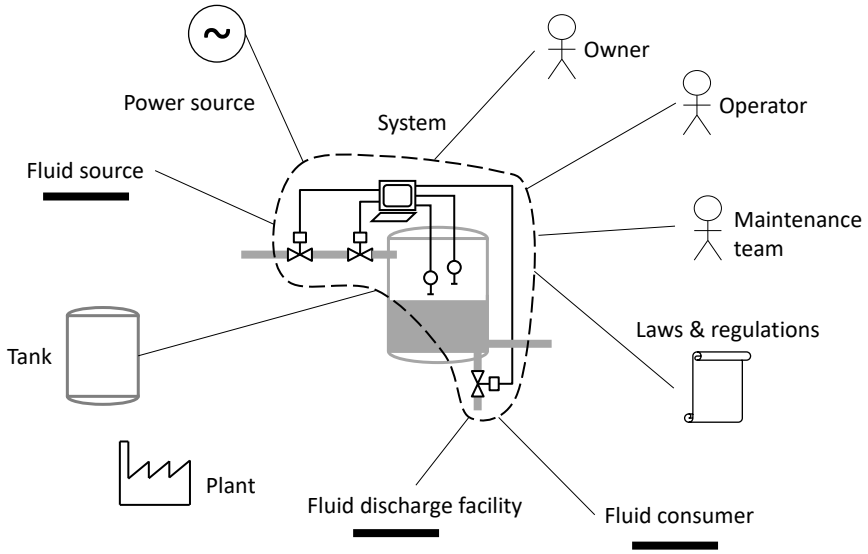


Figure 2.4: Environment diagram for the system of case study 1.

`Tank`, the output valve `OV`, the `source` and `consumer` pipes from the perimeter of the study. ■

An environment diagram such as the one of Figure 2.4 without a description of interactions between the system and its environment is of little use. To keep diagrams readable, it is however in general impossible to overload connections between boxes (or icons) representing the systems with meaningful labels. Interactions should be thus described separately, typically in a table or in a spreadsheet.

■ **Example 2.3 – Interactions of the overflow protection system with its environment.** Table 2.1 describes the interactions between the overflow protection system of case study 1 and its environment. ■

As already pointed out, the description of the contract between the system and its external systems, via the boundaries analysis, is done concurrently with the description of the other faces of the cube. At least at the end of the architecture study, services required and provided by the system should be reflected in use cases and in the functional architecture. No service should be "pending" without being referred to in these faces of the cube.

2.5 Functional Architecture

A technical system is designed and operated to deliver a number of services. These services can be seen as the top level functions of the system. To be implemented, these top level functions need in general to be decomposed into sub-functions, which can be in turned decomposed, and so on until a sufficiently small granularity is obtained. The degree of decomposition depends indeed on the system and the purpose of the analysis. The hierarchy of functions hence obtained is called the *functional architecture* of the system. Functional architectures are sometimes called *functional breakdown structures*.

Two important remarks must be made at this point.

First, the intuition of a tree like decomposition of functions is only partially correct. There may be sub-functions, typically support functions like power supply, that are shared by multiple higher level functions. Rather than as a tree, the decomposition is thus organized as a directed acyclic graph.

Second, not all of the sub-functions are implemented by the system. We can take, here again,

Table 2.1: Interactions between the overflow protection system and its environment.

External system	Services required from external systems
`source` pipe	Location of valves `SDV1` and `SVD2`
`consumer` pipe	Regular draining of the liquid
`Tank`	Location of sensors `LS1` and `LS2`
Power source	Power for the valves `SDV1` and `SVD2`, the controller `C` and sensors `LS1` and `LS2`.
Discharge facility	Location of the valve `DV` Draining of the overflow
Plant	Location of the controller `C`
Maintenance team	Maintenance of the system

External system	Services provided to external systems
`Tank`	Protection against overflow
Operator	Online monitoring of the level of liquid in the tank
`consumer` pipe	Control of the output flow
Owner	Demonstration that the system is efficient (in terms of availability, cost-effectiveness, energy consumption...) enough to be operated
Laws and regulations	Demonstration that the safe is safe enough to be operated

the example of the power supply: this service is most probably delivered by an external system. We have thus a situation where the top level functions correspond in general to services delivered to external systems and some of the bottom functions (leaves of the hierarchy) are services delivered by external systems.

We shall use massively functional architectures in the subsequent chapters of this book. For now, we can notice that there are mainly two ways of representing them graphically:

– By means of tree-like representations, with the proviso made above that they are not really trees, but rather directed acyclic graphs.
– By means of block-diagram like representations.

Both representations have their advantages and their drawbacks.

Before illustrating these points on our example, we shall discuss two architectural patterns that occur frequently in technical systems, those of redundant systems and control systems.

A *redundant system* is a system whose main function is duplicated, typically for the sake of safety. The simplest case of redundant systems is the case where the same function F is implemented by two or more physical mechanisms. In many systems however, it is not exactly the same functions that these mechanisms implement. Rather, they implement two different functions F_1 and F_2, both both being sufficient, independently of the other, to deliver the service the system is expected to deliver. If F_1 and F_2 are applied indifferently (and in general simultaneously), one speaks of *hot redundancy*. If F_1 is applied first, then F_2 in case F_1 is not available, one speaks of *cold redundancy*.

In our overflow protection system, the prevention of the overflow is implemented by means of two functions in cold redundancy: first, the level is controlled by the chain consisting of the level sensor `LS1`, the controller `C` and the shutdown valve `SVD1`. Then, in case this chain does not work properly, the overflow is drained by the chain consisting of the level sensor `LS2`, the controller `C`, the shutdown valve `SVD2`, and the discharge valve `DV`.

Both chains are control systems. Generally speaking, a *control system* is in charge regulating the behavior of some other device called the *process or equipment under control*. Control systems range

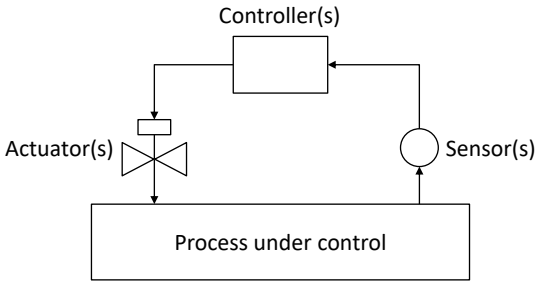


Figure 2.5: Generic structure of a control system.

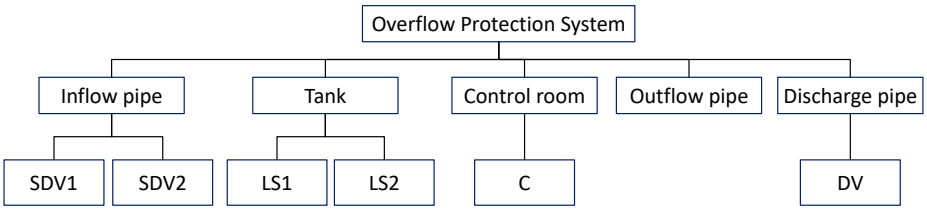


Figure 2.6: Tree-like representation of the functional architecture of the overflow protection system.

from home heating controller using a thermostat controlling a domestic boiler to large industrial systems used for controlling processes or machines. A control system implements a *control loop*, also called *feedback look* made of three functions, see Figure 2.5:

- A capacity to measure the physical quantity (or quantities) that will be used for the control. This capacity is usually implemented by means of *sensors*.
- A capacity to make decision based on the quantity measured. This capacity is often implemented by means of an electronic *controller*.
- Finally a capacity to act on the equipment under control. This capacity is implemented by means of *actuators* such as valves, pumps...

The notion of *functional chain* plays an important role in systems architecture. As pointed out in particular by Voirin (Voirin 2008), large technical systems are often looked at only via the functional chains they implement.

The notion of *architecture patterns* is even more central. Patterns are actually pervasive in engineering. They have been developed for instance in the field of technical system architecture (Maier 2009), as well as in software engineering (Gamma et al. 1994).

But let us come back to our example.

■ **Example 2.4 – Functional architecture of the overflow protection system.** We shall consider that the main service delivered by the overflow protection system of case study 1 is to prevent the overflow of the tank.

Figure 2.6 shows a tree-like representation of the functional architecture of the system. Note that we chose, for the sake of simplicity, to represent the power supply, which is a support function, at the level of the two control chains. We could have alternatively represented it either at the same level as these two chains or, in the opposite direction, at the level of base functions.

Figure 2.7 shows a block diagram representation of the functional architecture of the system.

■

Our example shows that tree-like representations are probably more appropriate to grasp the hierarchical decomposition. They are also better suited to represent the sharing of an element by several branches of the hierarchy. The block diagram representations are probably more appropriate to represent the chaining of elements, and thus functional chains.

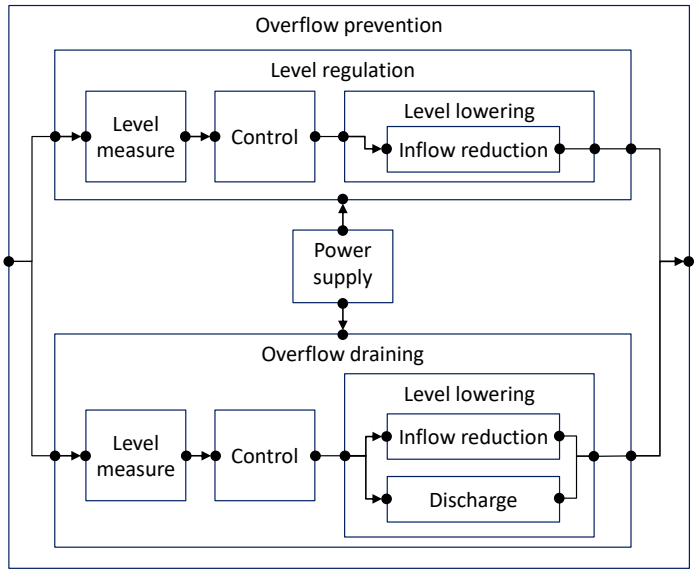


Figure 2.7: Block diagram representation of the functional architecture of the overflow protection system.

This illustrates our thesis stating that models should not be confused with their graphical representions.

2.6 Operation Modes

Technical systems have in general different *modes of operation*. The mode of operation may depend on the internal state of the system as well as on the state of the environment. Changes of modes can thus be triggered by internal actions as well as by external events. In each mode, the system may deliver different services to external systems and in turn may require different services of external systems to do so. This is the reason why the characterization of operation modes and their transition is worth to study and therefore is one of the face of the cube.

The chaining of operation modes is sometimes called the *life-cycle* of the system. This term is however confusing as it refers also to the chaining of the phases of the life of a product, like design, manufacturing, commissioning, operation, and decommissioning. These phases are indeed also worth to study, but we will not address them in the framework on this book, where we focus on the specification of technical systems.

Operation modes and transitions between these modes are often represented by means of hierarchical state automata called *state-charts*. State-charts have been introduced by David Harel (Harel 1987) in the framework of software specification and validation. These diagrams are available in both UML and SysML (under the name *state diagrams*).

■ **Example 2.5 – Operation modes of the overflow protection system.** Figure 2.8 shows the state-chart describing the modes of operation of the overflow protection system. We considered three major modes: operation, maintenance and failed. The operation is itself split into three minor modes: working (to represent the case where the system is working properly), regulating (to represent the case where the valve `SVD1` has be closed to regulate the level), draining (to represent the case where the valve `SVD2` has be closed and the valve `DV` has been open to drain the liquid).
■

The above example shows that a particular attention should be put on describing not only on how the system goes from their normal operation mode to degraded operation modes, but also on

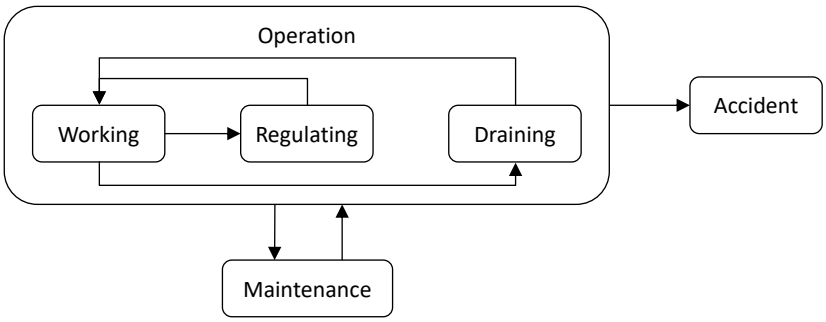


Figure 2.8: State-chart representing the operation modes of the overflow protection system.

the process by which they are put back in normal operation mode once degraded or stopped.

2.7 Physical Architecture

All technical systems consist of a number of physical components that are interconnected in some way. This is the reason why the Sketch face of the cube relies often on a *process and instrumentation diagram*, P&ID for short, or something similar.

In some cases, it is worth, for the purpose of the study, to decompose further the components showing up on the diagram.

It is often the case that, apparently at least, each component is in charge of implementing a specific function and that functional chains can be obtained by following the connection between components. As an illustration, in our overflow protection example, the sensor `LS1` implements a measurement capacity, the controller `C` a control capacity and the valve `SDV1` an actuation capacity. The physical chain `LS1-C-C` implements thus a functional capacity, namely a overflow protection capacity.

According to this remark, it may seem that it is not worth to distinguish between the physical and the functional architectures of the system as they are just two ways of naming the same thing. In industrial practice, many models designed for safety analyses mix actually functional and physical aspects. As a prototypical example, fault trees (that we shall study Chapter 6) rely on hierarchical decompositions of systems. The top part of the hierarchy consists of a description of the capacities of the system, or more exactly of the loss of these capacities, while the bottom part consists of a description of the physical components of the system, or more exactly of the failure modes of these components. Each intermediate node of the hierarchy connects its children nodes either by a or gate or by an and gate. The whole hierarchy describes eventually the combinations of failure modes of physical components that induce a loss of the main capacity of the system, i.e. the situations in which the system is not able to deliver the expected capacity.

There are however good reasons to separate functional and physical architectures.

First, a system architecture study does not aim only at describing a particular system, but also the possible variants of this system. In the earliest phases of the design of a technical system, it is often the case that some choices remain to make on the technologies to be used to implement some of the functions. The choice is usually made based on both technical and economic considerations. To make it, one has thus to distinguish clearly the functions to be implemented, which are required for the system to deliver the expected service, from the way they are implemented, which is the subject of the choice. This kind of analyses is referred to as *trade-off analyses* in the systems engineering literature, see e.g. (D. G. Ullman and Spiegel 2006). In the same vein, it may be the case that the chosen design has to be changed because it does meet some requirements in terms of cost, safety, weight, space occupation or any other measurable performance. In such an event, it is

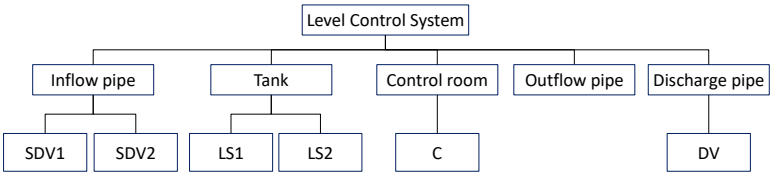


Figure 2.9: Zonal decomposition of the overflow protection system.

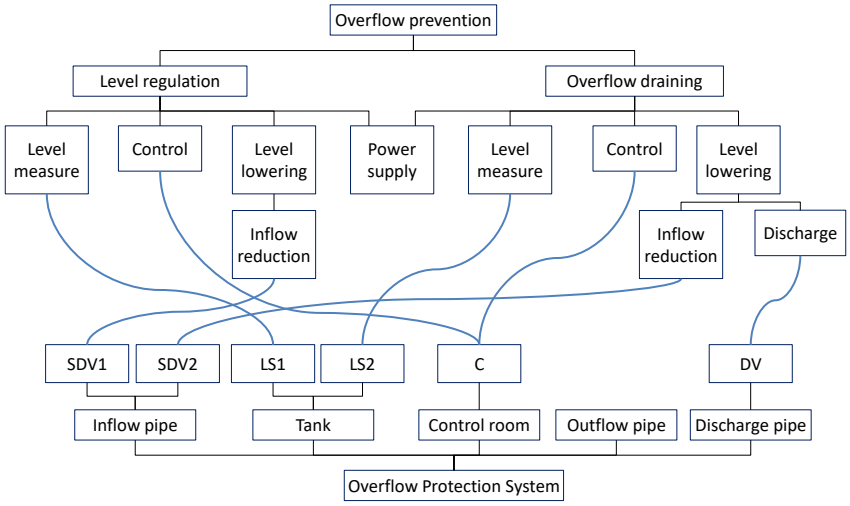


Figure 2.10: Allocation of functions onto physical components the overflow protection system.

thus worth to understand what can be changed and what cannot.

The second reason is more directly technical: looking at components via the functions they implement may lead to miss some important features of the system, like where are the components physically located? to which providers are they ordered? who is in charge of operating them? who is in charge of maintaining them? and so on. These questions may have strong impacts in terms of compliance to law and regulations, energy consumption, cost effectiveness, safety, maintainability...

Each of these aspects may deserve to organize components in a separate hierarchy. As for the functional architecture, tree-like and block diagram graphical representations can used to present the physical architecture, with the same advantages and drawbacks.

■ **Example 2.6 – Zonal decomposition of the overflow protection system.** Figure 2.9 shows a zonal decomposition of the overflow protection system of case study 1. Such a decomposition is of primary importance in safety analyses to detect potential common cause failures: components located in the same zone may be damaged simultaneously due to an impact, a fire, a flooding, electromagnetic waves...

■

Once the functional and the physical architectures established, it is possible to allocate functions onto physical components (or part).

■ **Example 2.7 – Allocation of functions on components of the overflow protection system.** Figure 2.10 shows the allocation of functions onto physical components determined by the functional and physical architectures of the overflow protection system.

■

Figure 2.10 shows that even for a very small system, the graphical representation of allocation

Table 2.2: Design Structure Matrices for the overflow protection System

(a) Before reorganization

	LS1	C	SDV1	PS	LS2	SDV2	DV
LS1				x			
C	x			x	x		
SDV1		x		x			
PS							
LS2				x			
SDV2		x		x			
DV		x		x			

(b) After reorganization

	LS1	LS2	C	LS2	SVD2	DV	PS
LS1							x
LS2							x
C	x	x					x
SDV1			x				x
SDV2			x				x
DV			x				x
PS							

link tends to look like a spaghetti plate. Allocation relations are thus better represented in a table or a spreadsheet.

Allocation links can be used to trace the impacts of an event in a zone, of a problem with a provider, ... They are also used to check that each function is actually allocated to a component (if it is not a service delivered by an external system, like the power supply in our example), and reciprocally that no component of the system is useless because it implements no function (or at least that the function implemented by this component has not been forgotten in the analysis).

2.8 Design Structure Matrices

It is sometimes the case that the "good" functional and physical architectures of a technical system are difficult to make emerge, due to the large number of components or functions. In such cases, *Design Structure Matrices*, DSM for short, can be a good help (Eppinger and Browning 2012).

The basic idea of DSM consists in organizing the n components (or functions) of the system is a $n \times n$ square matrix. The row i and the column i of the matrix represent the same component. The cell (i, j) of the matrix is filled with some information about the interactions between the component i and the component j. Typically, it contains 1 if the component i uses some output of the component j and 0 otherwise. Starting from any order of the components, rows and columns can be rearranged so make visual patterns emerge.

■ **Example 2.8 – Design Structure Matrices for the overflow protection System.** Table 2.2 shows the Design Structure Matrices for the overflow protection System before and after reorganization (and coloring). The right hand side matrix makes appear clearly the following.

- The power supply `PS` is a support function used by all other components.
- The controller `C` plays a pivot role: it is uses the outputs of a subset of components and its output is used by another subset of components, disjoint from the first one.

■

In the book cited above, Eppinger and Browning show how Design Structure Matrices can be use to detect clusters of components and functions.

When we visualize graphical representations such as functional and physical decompositions or design structure matrices, we use our short-term memory. The short-term memory can be seen as the working space of the brain. It is opposed to the long term memory which is the place where stabilized knowledge is stored. A number of cognitive science experiments have been performed that show that the short-term memory is able to distinguish seven plus or minus two items, so called chunks. This is called the Milner's rule (Miller 1970). *Mutatis mutandis*, the Miller's rule

is a good heuristic to group items so to obtain easy to read and easy to understand hierarchical representations: each node should contain seven plus or minus sub-nodes.

2.9 Requirements

The (implicit or explicit) outcomes of the systems architecture process can be formalized as sets of requirements. According to (IEEE 2005):

> A *requirement* $R(S)$ on a system S is a non ambiguous, testable and measurable property that expresses a characteristic of a constraint that the system should satisfy to be accepted by the stakeholders.

The characteristics of a good requirement are variously stated, see e.g. (Pohl and Rupp 2011). However, the following characteristics are commonly accepted.

- A requirement must address one and only one property, concisely stated.
- A requirement must be unambiguous, without recourse to technical jargon, acronyms.
- A requirement must be fully stated in one place with no missing information.
- A requirement must be objective and verifiable, not letting any room to interpretation.
- Two requirements cannot contradict one another.
- The importance of a requirement (ranging from mandatory to nice-to-have) must be clearly stated.
- The maturity of a requirement must be clearly stated.
- Last but not least, a requirement must come with the description of the means that will be used to check whether the system meets it or not.

In practice, requirements are in general a short sentence obeying typically a pattern in the form:

The «system» should «do something» with «this level of performance» in «that context».

Requirements are more and more often used as a contractual basis. Requirement management systems are available that make it possible to manage large sets of requirements, in version and in configuration.

■ **Example 2.9 – Requirements for the overflow protection system.** Here follows some requirements that could be written to specify the overflow protection system.

- The overflow protection system should prevent the level of liquid to be get to one 50 centimeters from the top of the tank with a probability higher than $1 - 10^{-6}$ per hour of use.
  Verification mean: on site test and safety analysis.
- The regulation system should prevent the level of liquid to be get to 100 centimeters from the top of the tank with a probability higher than $1 - 10^{-4}$ per hour of use.
  Verification mean: on site test and safety analysis.
- Once the regulation system has stop the incoming flow, it should take less than one hour for the operator to restart the production.
  Verification mean: on site test.
- ...

■

2.10 Further Readings

The reader interested in digging further on systems architecture may look at the following books.

Among the general descriptions of the systems architecture process, we can cite:

- The handbook edited by INCOSE (Walden et al. 2015).
- The guide of Daniel Krob (Krob 2017) already cited.

– The book by Blanchard and Fabricky (Blanchard and Fabrycky 2008).
– The book by Maier (Maier 2009).

On systems architecture based on standardized diagrams, we can cite:

– The reference book on UML (Rumbaugh, Jacobson, and Booch 2005) (dedicated to software systems).
– The reference guide on SysML (Friedenthal, A. Moore, and Steiner 2011).
– The reference book on BMPN (White and Miers 2008).
– Several books of interest, e.g. (Dori 2016; Feiler and Gluch 2015; Holt and Perry 2013; Weilkiens et al. 2015).

On a more cultural perspective, about system thinking, we can suggest the following books, which are easy to read:

– The book by the pioneer Ludwig von Bertalanffy (von Bertalanffy 2015).
– The books by the Nobel prize Herbert Simons, e.g. (H. Simon 1996).

2.11 Exercises and Problems

Exercise 2.1 – Overflow Protection System: Additional Use Cases. The objective of this exercise is to write down additional use cases for the overflow protection system (case study 1).

Question 1. Write a use case to explain why and how the valve `SDV2` is closed.

Question 2. Write a use case to explain how the system is put back in normal operation after either the valve `SDV1` or the valve `SDV2` has been closed.

Question 3. Write a use case to explain why the output valve `OS` may remain close while it was supposed to be open. The use case should explain what happens in this case.

Question 4. Write a use case that describes an accident.

Question 5. Write a use case that describes a maintenance operation.

■

Exercise 2.2 – Overflow Protection System: BPMN Diagrams. This exercise comes in complement of the previous one.

Question 1. Design BPMN diagrams for the use cases you defined in exercise 2.1.

■

Problem 2.3 – BPMN Diagram of a Loan Assessment Process. A lending agency wish to upgrade its information system. You are called as a IT consultant to help to specify the new system. To do so, you need to understand business processes of this company. One of the key point is to understand how a loan demand is assessed. Here is what you are explained:

> The sales representative enters the loan demand. The financial analyst assesses the demand. Contracts are managed by the contract department. For each demand, one needs either to create a contract or to update the existing one. The financial analyst must notify the answer to the sales representative so that he or she can inform the client.

Not very clear? All right, let us use BPMN to clarify things up.

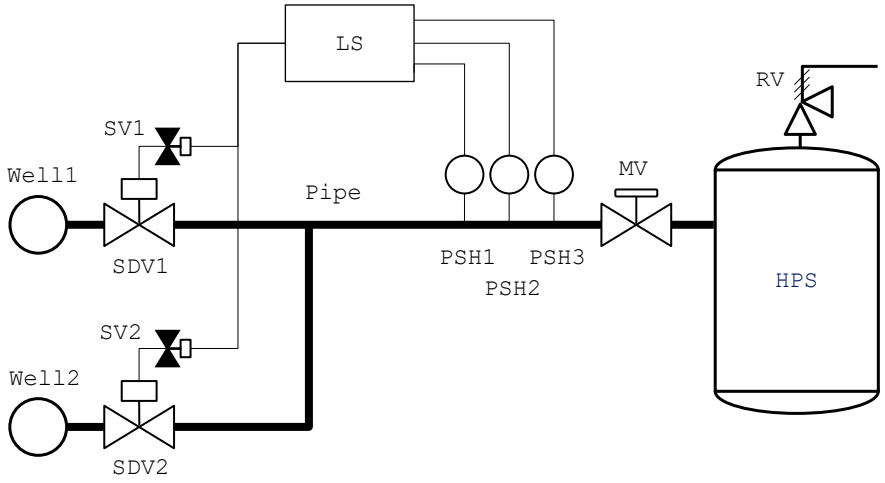


Figure 2.11: High pressure separator with relief valve.

Question 1. Design a BPMN diagram to represent the different scenarios of the loan assessment process.

■

Problem 2.4 – Order a Flight Ticket to an Online Travel Agency. The objective of this problem is to describe the process by which a client orders a flight ticket to an online travel agency.

Question 1. Figure out what are the actors of this business process as well as its activities.

Question 2. Design a BPMN diagram to represent the different scenarios of the process.

Question 3. Introduce in your diagram other services, e.g renting a car, booking a hotel room. . . .

■

■ **Case Study 2 – High Integrity Pressure Protection System.** Figure 2.11 shows a separation system as found in oil and gas industry. The mixture of oil, water and gas extracted from the wells `Well1` and `Well2` is sent to the high pressure separator `HPS` through the pipe `pipe`. To avoid damages to the separator caused by over-pressures, a relied valve `RV` is installed: when the pressure inside the vessel gets too high, this valve opens and releases the gas which is flared. Moreover, a manual valve `MV` makes it possible to stop the inflow, typically to be able to perform maintenance operations.

This protection is relatively cheap and efficient. However, it does not avoid any over-pressure. Repeated over-pressure in the vessel may eventually damage it. Moreover, restarting the process after a relief valve has been opened requires the operator intervention and takes time. Finally, flaming gas is not very environment friendly.

Consequently, another device is installed to prevent over-pressures by acting upfront. This device is called a *High Integrity Pressure Protection System*, HIPPS for short. Three redundant pressure sensors (`PSH1`, `PSH2` and `PSH3`) detect a possible over-pressure and send the information to a 2/3 logic solver `LS`. In case of a confirmed over-pressure, this logic solver activates the solenoid valves `SV1` and `SV2`, which in turn close the shutdown valves `SDV1` and `SDV2`. This removes eventually the over-pressure in the separator. ■

Exercise 2.5 – HIPPS: Use Cases. The objective of this exercise is to write down additional use

cases for the overflow protection system (case study 2).

Question 1. Write a use case to explain why and how the shutdown valves `SDV1` and `SDV2` are closed.

Question 2. Write a use case to explain how the system is put back in normal operation after valves `SDV1` and the valve `SDV2` have been closed.

Question 3. Write use cases to explain why the relief valve `RV` must to be open.

Question 4. Write a use case to explain how the system is put back in normal operation after the relief valve `RV` has been open.

Question 5. Write a use case that describes an accident.

Question 6. Write a use case that describes a maintenance operation on the high pressure separator `HPS`.

■

Exercise 2.6 – HIPPS: BPMN diagrams. This exercise comes in complement of the previous one.

Question 1. Design BPMN diagrams for the use cases you defined in exercise 2.5. Try to embed as many use cases as reasonable into a diagram.

■

Exercise 2.7 – HIPPS: Boundaries analysis. This exercise consists in doing a boundaries analysis for the HIPPS case study.

Question 1. Design an environment diagram for the protection system of the high pressure separator `HPS`.

Question 2. Complete your environment diagram with a table describing the interactions of the protection system with its environment.

■

Exercise 2.8 – HIPPS: Functional architecture. The objective of this exercise is to describe the functional architecture of the HIPPS case study.

Question 1. What are the functional chains of the protection system of the high pressure separator `HPS`?

Question 2. Design a tree-like representation for the functional architecture of this system.

Question 3. Design a block-diagram representation for the functional architecture of this system.

■

Exercise 2.9 – HIPPS: Operation Modes. The objective of this exercise is to determine the operation modes of the protection system of the separator `HPS` in the HIPPS case study.

Question 1. What are the operation modes of the protection system of the high pressure separator `HPS`? Take care of the location of the different components.

Question 2. Design a state-chart representing the operation modes defined in the previous question and the transitions between these modes.

Question 3. Describe in a separate table the events or actions that trigger the transitions between modes.

■

Exercise 2.10 – HIPPS: Physical architecture. The objective of this exercise is to describe the physical architecture of the HIPPS case study.

Question 1. Define the different zones in which the components of the protection system of the high pressure separator `HPS` are located.

Question 2. Design a tree-like representation for the physical architecture of this system.

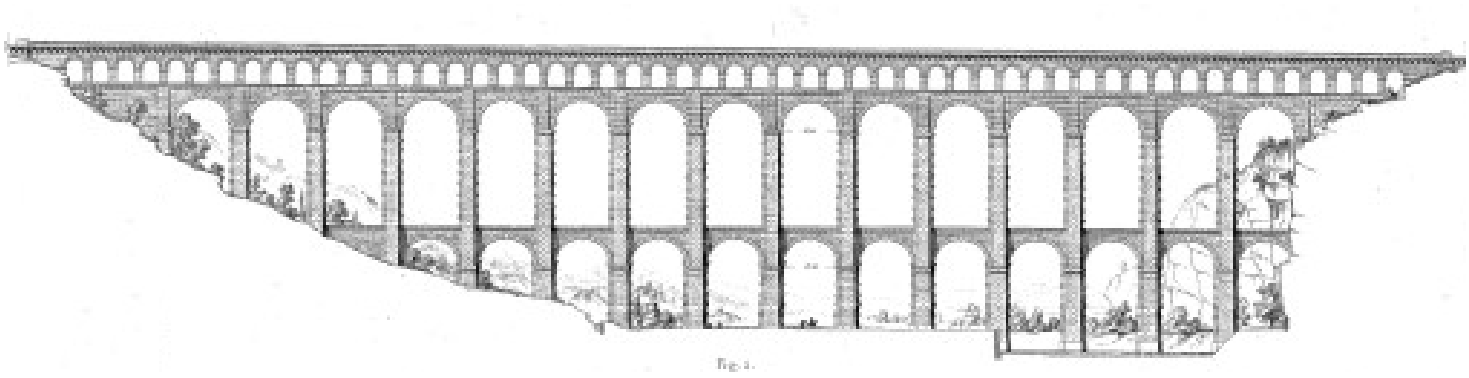
Question 3. Design a block-diagram representation for the physical architecture of this system.

■

Exercise 2.11 – HIPPS: Allocation of Functions to Physical Components. The objective of this exercise is to describe how of the protection system of the high pressure separator `HPS` of the HIPPS case study are allocated onto physical components.

Question 1. Using the functional architecture you designed at exercise 2.8 and the physical architecture you designed at exercise 2.10 describe in a table the allocation of functions onto physical components.

■



3. Fundamentals of Models

Key Concepts
- Diagrams and notations
- Models and modeling languages
- Syntax and grammar
- Extended Backus-Naur Form
- Semantics, denotational semantics, operational semantics
- Pragmatics
- Systems of symbolic equations, AltaRica 3.0

The model-based systems engineering literature proposes a wide variety of modeling formalisms and tools. These formalisms and tools are often named with "commercial" purposes in mind, which creates some confusion: more or less standardized notations are called modeling languages, no distinction is made between modeling formalisms and assessment tools, and so on. This chapter introduces fundamentals about models and modeling languages, i.e. those of syntax, semantics and pragmatics, so to clarify the panorama.

This chapter starts also introducing systems of data-flow equations and the AltaRica 3.0 modeling language.

3.1 Introduction

3.1.1 Objectives

In order to illustrate the concepts developed in this chapter, we shall consider again the case study presented Section 2.1 (page 20). Our objective is to design a simple model of this system that makes it possible to "play" use cases such as those we designed Section 2.3.

Our model should typically contain the following ingredients:

– A hierarchy of components that represent either concrete physical components, e.g. sensors, valves or computers or more abstract ones such as functions, locations or the level control system itself.

– Variables that represent the states of the components, e.g. a valve can be either open or closed, the fluid is coming or not into the tank. The values of some of these variables should be set directly by the analyst, e.g. the variable representing the state of a valve. The values of the others should be calculated from the values of the former, e.g. whether there is some fluid entering the tank depends on the state of valves `SDV1` and `SDV2`.

In other words, we shall consider components (and the system as a whole) as *transfer functions*, i.e. as mechanisms that calculate the values of their outputs from the values of their inputs and their internal states, see Figure 3.1 for an illustration. The behavior of the system is thus obtained by plugging outputs of some components onto inputs of some others.

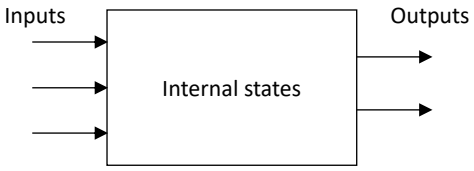


Figure 3.1: Transfer function

3.1.2 Making Things Concrete

To make things concrete, let us consider each component of our level control system in turn.

Tank

We must first consider the level of liquid in the tank. We can use a variable `Tank.filling` to do that. As we do not want to enter into the physics of the system, we can assume that `Tank.filling` can take four symbolic values: `EMPTY` in case there is no more liquid in the tank, `REGULAR` in case the level of liquid is below `tl1`, `HIGH` in case the level of liquid is between `tl1` and `tl2`, and finally `DANGEROUS` in case the level of liquid is above `tl2`.

In addition, we need variables to represent whether some liquid enters into the tank, and goes out the tank via the consumer pipe and the draining pipe. These variables can be Boolean variables, i.e. taking either the value `true` or the value `false`. Let us denote them respectively `Tank.inFlow`, `Tank.consumerOutFlow` and `Tank.dischargeOutFlow`.

Although we do not need to write the differential equations ruling the filling of the tank, we already write symbolic equations determining the values of variables `Tank.consumerOutFlow` and `Tank.dischargeOutFlow`:

$$\begin{aligned} \texttt{Tank.consumerOutFlow} &= \texttt{Tank.filling} \neq \texttt{EMPTY} \\ \texttt{Tank.dischargeOutFlow} &= \texttt{Tank.filling} \neq \texttt{EMPTY} \end{aligned}$$

Sensors

The two sensors `LS1` and `LS2` take the level of liquid in the tank as input and send or not a signal to the controller as output. We need thus Boolean variables `LS1.signal` and `LS2.signal` to represent the presence or the absence of these signals. Note that, at least if we consider that `LS1` and `LS2` are always functioning correctly, the values of `LS1.signal` and `LS2.signal` is fully determined by the value of `Tank.filling`:

$$\begin{aligned} \texttt{LS1.signal} &= \texttt{Tank.filling} = \texttt{HIGH} \vee \texttt{Tank.filling} = \texttt{DANGEROUS} \\ \texttt{LS2.signal} &= \texttt{Tank.filling} = \texttt{DANGEROUS} \end{aligned}$$

We shall release the assumption that sensors (and valves and the controller) work correctly in problem 3.11.

Valves

Valves can be either open or close. We need thus a Boolean variable to represent their state, e.g. `SDV1.closed`. Valves do not decide to open or close by themselves. They receive the order to

open or to close from the controller. We need thus another Boolean variable to represent this order, e.g. `SDV1.close`. Again assuming that everything works correctly, these two variables are linked by the following equation (in the case of valve `SDV1`).

$$\texttt{SDV1.closed} \;=\; \texttt{SDV1.close}$$

In addition, valves receive (or not) some liquid upstream and and emit some liquid downstream, when open. We can here again use Boolean variables, e.g. `SDV1.inFlow` and `SDV1.outFlow`. The value of `SDV1.outFlow` depends on the value of `SDV1.inFlow` and the value of `SDV1.closed`. Namely, it is ruled by the following equation.

$$\texttt{SDV1.outFlow} \;=\; \texttt{SDV1.inFlow} \wedge \neg\, \texttt{SDV1.closed}$$

Recall that $\wedge$ stands for "and", $\vee$ for "or" and $\neg$ for "not".

Controller

The controller `CTRL` receives signals from sensors `LS1` and `LS2`. Let us introduce two Boolean variables `CTRL.signal1` and `CTRL.signal2` to represent these signals. Depending on these signals, it switches into one of its three operation modes (see Figure 2.8 page 32): `OPEN`, `CLOSE` and `DRAIN`. We need thus a variable `CTRL.mode` to represent this mode. The value of this variable is ruled by the following equation.

$$\begin{array}{lcl}\texttt{CTRL.mode} & = & \texttt{if CTRL.signal2 then DRAIN}\\ & & \texttt{else if CTRL.signal1 then CLOSE}\\ & & \texttt{else OPEN}\end{array}$$

Depending on its mode, the controller sends orders to valves `SDV1`, `SDV2` and `DV`. Let us represent these orders by the Boolean variables `CTRL.close1`, `CTRL.close2` and `CTRL.close3`. We can thus write the following equations:

$$\begin{array}{lcl}\texttt{CTRL.close1} & = & \texttt{CTRL.mode} = \texttt{CLOSE} \vee \texttt{CTRL.mode} = \texttt{DRAIN}\\ \texttt{CTRL.close2} & = & \texttt{CTRL.mode} = \texttt{DRAIN}\\ \texttt{CTRL.close3} & = & \texttt{CTRL.mode} \neq \texttt{DRAIN}\end{array}$$

Putting things together

Figure 3.2 shows a block diagram representing our level control system. The connection between the blocks denote equations involving variables belonging to different components.

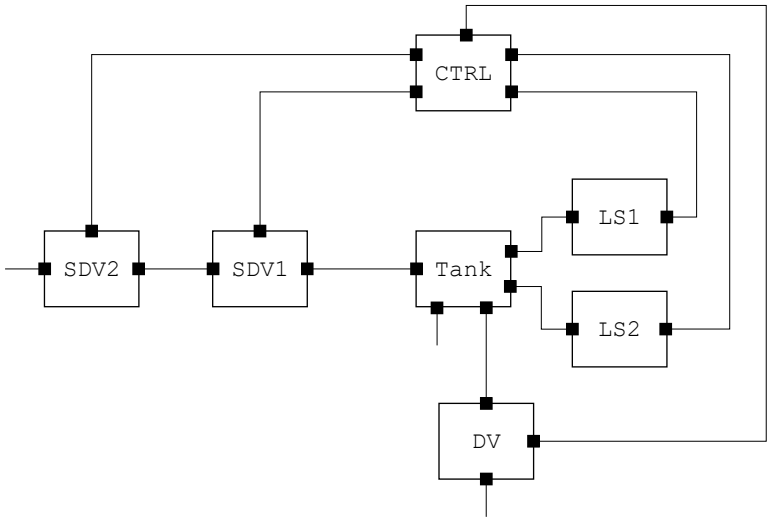


Figure 3.2: Block diagram for the level control system.

We can now complete these equations.

We need first to chain valves `SVD1` and `SVD2` and the tank along the input pipe:

$$
\begin{aligned}
\texttt{SDV2.inFlow} &= \texttt{true} \\
\texttt{SDV1.inFlow} &= \texttt{SDV2.outFlow} \\
\texttt{Tank.inFlow} &= \texttt{SDV1.outFlow}
\end{aligned}
$$

Similarly, we need to describe what goes on along the consumer pipe and to chain the tank and the valve `DV` along the discharge pipe:

$$
\texttt{DV.inFlow} = \texttt{Tank.dischargeOutFlow}
$$

Finally, we need to connect the controller to sensors on the one hand and to valves on the other hand:

$$
\begin{aligned}
\texttt{CTRL.signal1} &= \texttt{LS1.signal} \\
\texttt{CTRL.signal2} &= \texttt{LS2.signal} \\
\texttt{SDV1.close} &= \texttt{CTRL.signal1} \\
\texttt{SDV2.close} &= \texttt{CTRL.signal2} \\
\texttt{DV.close} &= \texttt{CTRL.signal3}
\end{aligned}
$$

Figure 3.3 shows the complete system of symbolic equations we have written for the level control system.

$$
\begin{aligned}
\texttt{SDV2.inFlow} &= \texttt{true} \\
\texttt{SDV2.outFlow} &= \texttt{SDV2.inFlow} \wedge \neg\, \texttt{SDV2.closed} \\
\texttt{SDV1.inFlow} &= \texttt{SDV2.outFlow} \\
\texttt{SDV1.outFlow} &= \texttt{SDV1.inFlow} \wedge \neg\, \texttt{SDV1.closed} \\
\texttt{Tank.inFlow} &= \texttt{SDV1.outFlow} \\
\texttt{LS1.signal} &= \texttt{Tank.filling} = \texttt{HIGH} \vee \texttt{Tank.filling} = \texttt{DANGEROUS} \\
\texttt{LS2.signal} &= \texttt{Tank.filling} = \texttt{DANGEROUS} \\
\texttt{CTRL.signal1} &= \texttt{LS1.signal} \\
\texttt{CTRL.signal2} &= \texttt{LS2.signal} \\
\texttt{CTRL.mode} &= \texttt{if CTRL.signal2 then DRAIN} \\
&\quad \texttt{else if CTRL.signal1 then CLOSE} \\
&\quad \texttt{else OPEN} \\
\texttt{SDV1.close} &= \texttt{CTRL.signal1} \\
\texttt{SDV1.closed} &= \texttt{SDV1.close} \\
\texttt{SDV2.close} &= \texttt{CTRL.signal2} \\
\texttt{SDV2.closed} &= \texttt{SDV2.close} \\
\texttt{DV.close} &= \texttt{CTRL.signal3} \\
\texttt{DV.closed} &= \texttt{DV.close} \\
\texttt{Tank.consumerOutFlow} &= \texttt{Tank.filling} \neq \texttt{EMPTY} \\
\texttt{Tank.dischargeOutFlow} &= \texttt{Tank.filling} \neq \texttt{EMPTY} \\
\texttt{DV.inFlow} &= \texttt{Tank.dischargeOutFlow} \\
\texttt{DV.outFlow} &= \texttt{DV.inFlow} \wedge \neg\, \texttt{DV.closed}
\end{aligned}
$$

Figure 3.3: System of symbolic equations for the level control system.

3.1.3 Discussion

To design the model sketched in the previous section, we made a number of modeling choices. For instance, we decided that the mode of the controller is calculated from its input signals, while the opening of the valves is decided by the analyst. These modeling choices are to some extent arbitrary. Our purpose here is more to illustrate the modeling concepts at stake than obtain a suitable model for our case study.

Similarly, we simplified the hierarchy of components by considering only a hierarchy two levels: the system level and the physical component level. A full fledged model would probably consist of a deeper hierarchy, as illustrated by the functional and physical architectures we designed for our case study in the previous chapter, see Figures 2.7 and 2.7.

Note finally that we represented the behavior of the system by means of simple symbolic equations. Here again, this choice is more driven by pedagogical purposes than by the will of representing faithfully the behavior of the level control system.

These remarks made, it is time to enter into more general considerations about models.

3.2 Notations versus Modeling Languages

The central question we shall debate in this chapter can be stated as follows.

What do we call a model in the framework of model-based systems engineering?

As pointed out in the preface of this book, the word "model" has many different meanings. Actually, much too many that are much too different to make possible a systematic study. Fortunately, at least if we restrict our attention to models developed in the context systems engineering, we can clarify things out.

3.2.1 Syntax and Semantics

To start with, we can remark that, in the previous chapter, we used a lot of diagrams. These diagrams actually been made using a software, e.g. Microsoft Powerpoint®, making it possible to draw geometrical forms—lines, rectangles, circles...—as well as to insert texts onto a canvas. The same software can be used to draw completely different things that do not look at all like our diagrams.

We saved our drawings into files, stored on the hard disk of our computer. These files are eventually sequences of 0's and 1's. Consequently, we can assimilate, without a loss of generality, any drawing made with our software to a sequence of 0's and 1's. The converse is however not true: our software is not able to read any sequence of 0's and 1's and to display it as a drawing. Some sequences are correct encoding of drawings, some are not. Our software simply rejects the latter.

This means that there is a set of rules that makes it possible for our software to determine whether a given sequence of 0's and 1's is a correct encoding of a drawing or not. Such set of rules is called a *syntax*, the syntax of computerized drawings in that case.

Formally, let us denote by $\{0,1\}^*$ the set of all finite sequences of 0's and 1's. The set of correct encodings of drawings is thus a subset $\mathscr{L}$ of $\{0,1\}^*$. $\mathscr{L}$ is called a *language*, the language of computerized drawings in that case.

Another software may rely on a different syntax for drawings, i.e. that files produced by our software may be impossible to read for this other software, and vice-versa. In other words, the language $\mathscr{L}$ recognized by our software may be different from the language $\mathscr{L}'$ recognized by the other software even though any drawing made with one software can also be made using the other one. Moreover, it may be the case that the sequence w of 0's and 1's that encodes a drawing made of a single rectangle in our software encodes a drawing made two circles in the other software. In such a case, although w belongs to both $\mathscr{L}$ and $\mathscr{L}'$, it does not encode the same drawing. In other words, it is not sufficient to distinguish between correct and incorrect sequences of 0's an 1's. There must be an additional set of rules telling us how to interpret a correct sequence of 0's and 1's as a drawing, i.e. as a set of geometric shapes with given sizes, colors, positions...

As the users of one or the other software, we are not really concerned by the way drawings are encoded. It can be anything convenient for the developers of these software.

Now, the diagrams presented in the previous chapter are not arbitrary drawings. They obey clearly stricter creation rules than usual drawings. Any diagram is a drawing, but the reverse is not true. As an illustration, consider again the block diagram pictured in Figure 3.2.

This diagram is by no means an arbitrary drawing. First, it obeys strict construction rules. Second, it is the graphical representation of a well-defined mathematical object. Namely, it represents a system of symbolic equations.

Note that the diagram of Figure 3.2 is only a partial view of the actual model: it does not show variables and equations. Adding them would just overload the diagram, hampering its readability. This is the reason why modeling languages are essentially textual. We shall come back to this point.

The set of graphical or textual construction rules a model must obey to be well-formed is called the *syntax* of that category of models.

The set of rules that describe how a well-formed model is *interpreted* as a well-defined mathematical object belonging to a certain mathematical framework is called the *semantics* of that category of models.

The target mathematical framework—systems of symbolic equations in the case of our diagram—defines not only mathematical objects, but also operations that can be made on these objects.

3.2.2 Notations

All engineering disciplines make a large use of more or less standardized diagrams. The question is however whether these diagrams obey a formally defined syntax, and if so, whether they are given a semantics. By *formally defined*, we mean here "computable" or equivalently "possible to check by means of computer program". The syntax of drawings is formally defined because a drawing software decides without ambiguity whether a sequence of 0's and 1's is a well-formed drawing or not. Moreover the answer is deterministic, i.e. is always the same for the same sequence. But what about diagrams? Is there a set of rules that defines what is a well-formed diagram and what is not? Moreover, is there a set of rules that associates a well-defined mathematical object, belonging to a well-defined mathematical framework, to each well-formed diagram?

We reach here a blind spot of the systems engineering literature: like other engineering disciplines, systems engineering relies for a good deal on diagrams, but the syntax of these diagrams, not to speak about their semantics, is formally defined nowhere. The difference between these diagrams and simple drawings, which is real and obvious, is nevertheless essentially a social convention. In the sequel, we shall call *notations* texts or diagrams obeying certain non-fully formalized rules or patterns. We reserve the term *model* to textual and possibly graphical descriptions obeying a formal syntax and semantics, i.e. written in some *modeling language*.

If diagrams are used to support the communication among stakeholders, the fact that they are not fully formalized is by no means a problem, as the communication itself is nothing but a social convention. But if we want to use them to perform experiments *in silico*, like calculating a performance indicator, then the lack of formal syntax and semantics is a show stopper. Computers, conversely to humans, have actually no capacity to "visualize" things, no imagination, no capacity for induction, in a word, no culture. They just apply mechanically sequences of instructions on data, i.e. on sequences of 0's and 1's. If we want our computer to interpret such a sequence in view of performing a certain calculation, then we have to define a formal syntax and a formal semantics for this sequence. There is no way around, no in between situations, i.e. no such as thing as a "semi-formal" syntax or semantics.

To conclude the above development, note that the syntax and semantics of computerized drawings is not defined on their visual aspect, but on sequences of 0's and 1's that encode them. This is by no means accidental: not only computers can only deal with such sequences, but more generally, it proves to be extremely difficult to define a formal syntax or a formal semantics on something else than a text. We shall discuss this point in further detail Chapter 9. It follows that if a

formal syntax and semantics for diagrams has to be defined, their visual aspect will be of little help.

3.2.3 Modeling Languages

At this point, let us reformulate some of our remarks.

First, in systems engineering as in all other scientific and engineering domains, mathematics are used to describe and to abstract real world objects and phenomena. As a paradigmatic example, the law of gravity captures in a single, relatively simple, equation an infinite variety of concrete situations.

The whole principle of model-based systems engineering is thus to describe and to abstract systems by means of mathematical objects obeying certain axioms and deduction rules. Mathematical objects used in this framework are essentially discrete. Abstract algebra and discrete mathematics provide methods and tools to create and to perform operations and calculations on these objects, see e.g. (O'Regan 2016; Pinter 2012) for introductory books.

Second, when the system under study is simple, the corresponding mathematical models are simple as well. They can thus be designed directly, *in cogito*, i.e. in the mind of the analyst, possibly with paper and pencil. But when it gets larger and more complex, such a process is no longer possible. The analyst needs the help of a computer to author models and to perform virtual experiments (that may be extremely computationally expensive). Models must thus be created *in silico*.

Third, a *computerized model* is eventually a sequence of symbols that the computer can process and store into a data structure. This sequence of symbols must obey certain construction rules, i.e. must have a formal *syntax*, and must encode a mathematical object, i.e. must have a formal *semantics*. Construction rules together with interpretation of data structures in terms of mathematical objects define (artificial, formal) *modeling languages*. Operations that are performed on data structures are justified by their mathematical counterparts.

Fourth, graphical user interfaces make it possible to use computers to carry out all sorts of drawings, including drawings obeying with some conventions, like diagrams. However, that these diagrams obey certain construction rules does not make them models. To deserve the "status" of model, a sequence of symbols, or an assembly of graphical constructs, must be designed within a modeling language, i.e.

- must obey strict construction rules, that make it possible to tell, without ambiguity, whether it is a well-formed model or not, i.e. it must have a syntax, and
- must be interpreted, without ambiguity, as a mathematical object obeying certain axioms and deduction rules, i.e. must have a semantics.

The existence of a clearly established syntax and semantics differentiate *modeling languages* from *notations*. Graphical notations can be very useful as a communication means, but they are what they are: drawings, not models.

Fifth, this said, it is very useful to use graphical notations to communicate about models. Graphical notations provide convenient views on models. They can be generated from the models or be persistent, i.e. exist aside the model. In this later case, there is however a risk to introduce a discrepancy between the model and its graphical views.

When the model gets large and complex—and this is necessarily the case of models of complex systems—it is impossible to represent it entirely graphically. Graphical representations of the model are then partial views on the model. The set of these partial views does not need to cover all features of the model. The experience shows that, on the contrary, there are many features that are better understood by looking directly at the text of the model, which is in any case the ultimate reference.

3.2.4 Pragmatics

Consider again the diagrams pictured in Figures 2.6 and 2.7 (pages 30 and 31). We wrote that these diagrams are a graphical representation of the functional architecture of the level control system we took as an example throughout the chapter. It is worth to come back on this statement.

From a mathematical standpoint, it is not difficult to define formally what is a hierarchy: fundamentally, it is a partial order relation. It is thus perfectly possible to design a model such that a possible graphical representation of this model is the tree-like diagram pictured in Figure 2.6 and another graphical representation of this model is the block-diagram pictured in Figure 2.7. Both graphical representations could even be generated automatically from the model.

So far, so good.

Now, we have a perfectly well defined mathematical object, a partial order relation, with well established properties, those of partial order relations, and we pretend that this mathematical object represents the functional architecture of our level control system. By doing so, we interpret a mathematical construct as a characteristics of the real world.

This interpretation is however definitely impossible to formalize. This would actually require to formalize notions like those of function (of a technical system), functional architecture and level control system, which, one thing after the other, would require to formalize a gigantic corpus of knowledge accumulated over the ages by human beings.

As an illustration, it would be meaningless to exchange labels `level regulation` and `level measure` in the diagram pictured Figure 2.6 (and thus in our model), even though this diagram would go on representing a partial order relation. We know, as a part of our engineering culture, that to regulate a level one needs to measure it, not the reverse.

In other words, not all of the partial order relations are legitimate candidates to represent the functional architecture of a level control system, even if we fix upfront the set of allowed labels. To put things differently, we cannot understand a model like the ones graphically represented Figures 2.6 and 2.7 without referring to the context.

Similarly, consider the block diagram obtained by switching the sensor `LS1` and the controller in the block diagram of Figure 3.2. This block diagram is well-formed and perfectly correct from a strict mathematical standpoint, but definitely meaningless from a pragmatic one.

The huge set of essentially non-formalisable rules by which we interpret a formal description as a property of the real world, is called the *pragmatics* of this description. In linguistics and semiotics, the pragmatics is a sub-field that studies the ways in which the context contributes to the meaning (Cruse 2011). This is exactly what we mean here, applied to the specific framework of model-based systems engineering.

The pragmatics of a model is thus fundamentally different from its semantics. The former stands in the "real" world, while the latter stands in the realm of mathematics. We said above that, for this very reason, the pragmatics of a model, or an engineering domain, is essentially non-formalisable. This does not mean however that nothing can be done to think about it and to organize it. The notion of *ontology* and more specifically of *pattern* proves on the contrary to be fruitful, see e.g. (Holt, Perry, and Brownsword 2016; Maier 2009). But one thing is to design methodologies to help the analyst during the modeling process, another is to design computerized algorithms to calculate indicators from models.

3.2.5 Notations or Modeling Languages?

Between notations and modeling languages, there is no hierarchy of value. Models are not "better" than diagrams because they are formal. Rather, diagrams and models, notations and modeling languages, serve different purposes. Diagrams aim at supporting the communication among stakeholders, while models are used to perform experiments *in silico*.

This said, notations and modeling languages are not symmetric neither. It is actually possible

to use a model to support the communication, while it is impossible to use a diagram to calculate an indicator, at least without giving to it a formal syntax and a formal semantics, i.e. without transforming it into a model.

The author believes that it would be highly beneficial for the systems engineering discipline to use more modeling languages and less notations. It is always better to know precisely what we are speaking about in engineering. This is the reason why book is not about diagram-based systems engineering, but about model-based systems engineering.

Indeed, it would be of little interest to try to formalize the diagrams we draw to document the sketch face of the cube, like the one pictured in Figure 2.1. Environment diagrams, like the one pictured Figure 2.4, are probably not worth to formalize neither. But all other diagrams presented in the previous chapter can be advantageously formalized. The remainder of this book is actually dedicated to this topic.

But before doing so, we need to dig further the concepts of syntax, semantics and pragmatics.

3.3 Syntax

To design a model, one needs thus a modeling language. There exists a wide variety of modeling languages. Some are textual, some are graphical and some are both (meaning that models can be designed in both form). Modeling languages have well defined a syntax. This section digs further the notions of syntax and grammar.

3.3.1 Textual Representation of Block Diagrams and Systems of Symbolic Equations

If we want to turn block diagrams and systems of symbolic equations into a textual modeling language, we need to design a syntax to describe them, i.e. we need to set the rules defining which texts represent well-formed models and which do not. The choice of a particular syntax is to some extent arbitrary.

Figures 3.4 and 3.5 show a possible model for our level control system, written in AltaRica 3.0.

The model (on Figure 3.4) starts by declaring two domains. In AltaRica, each *variable* comes with a *domain*, i.e. the set of values the variable can take. There are predefined domains: Boolean, integers and reals. There are also user-defined domains, which are sets of symbolic constants. The domain `LiquidLevel` is thus the set of the four symbolic constants `EMPTY`, `REGULAR`, `HIGH`, and `DANGEROUS`. Although this is not mandatory, we write always symbolic constants using capital letters.

The model declares then the outer block, which represents the level control system. In AltaRica 3.0, *blocks* are containers for declarations. In our restricted version of the language, blocks are thus containers for variables, equations and other blocks. Block declarations start with the keyword `block` (keywords are written in bold, blue font) and terminates with the keyword `end`. Right after `block`, the name of the block is given.

The body of block declarations consists of two parts: first declarations of variables and blocks, then declarations of equations, which are given after the keyword `assertion`.

Variable declarations start with the domain of the variable, then its name, then its attributes. *Attributes* are pairs consisting of a name and an expression. They characterize some features of variables (and other modeling elements). Attributes are given between parentheses and are separated with commas.

In AltaRica, there are two types of variables:

– *state variables*, that are declared with the attribute `init`, and

– *flow variables*, that are declared with the attribute `reset`.

For now, the reader can see state variables are those whose values can be set the analyst, while the values of flow variables are calculated by means of the equations given in the assertion part.

```
domain LiquidLevel {EMPTY, REGULAR, HIGH, DANGEROUS}
domain Mode {OPEN, CLOSE, DRAIN}

block LevelControlSystem
  block Tank
    LiquidLevel _filling(init = REGULAR);
    Boolean inFlow(reset = false);
    Boolean consumerOutFlow, dischargeOutFlow(reset = true);
    assertion
      consumerOutFlow := _filling!=EMPTY;
      dischargeOutFlow := _filling!=EMPTY;
  end
  block LS1
    Boolean signal(reset = false);
  end
  block LS2
    Boolean signal(reset = false);
  end
  block CTRL
    Mode mode(reset = OPEN);
    Boolean signal1, signal2(reset = false);
    Boolean close1, close2, close3(reset = false);
    assertion
      mode := if signal2 then DRAIN
              else if signal1 then CLOSE
              else OPEN;
      close1 := mode==CLOSE or mode==DRAIN;
      close2 := mode==DRAIN;
      close3 := mode!=DRAIN;
  end
  block SDV1
    Boolean closed, inFlow, outFlow, close(reset = false);
    assertion
      closed := close;
      outFlow := inFlow and not close;
  end
  block SDV2
    Boolean closed, inFlow, outFlow, close(reset = false);
    assertion
      closed := close;
      outFlow := inFlow and not close;
  end
  block DV
    Boolean closed, inFlow, outFlow, close(reset = false);
    assertion
      closed := close;
      outFlow := inFlow and not close;
  end
  ...
end
```

Figure 3.4: A first AltaRica model for the level control system (part 1)

```
block LevelControlSystem
  ...
  assertion
    SDV1.inFlow := true;
    SDV2.inFlow := SDV2.outFlow;
    Tank.inFlow := SDV1.outFlow;
    DV.inFlow := Tank.dischargeOutFlow;
    LS1.signal := Tank._filling==HIGH or Tank._filling==DANGEROUS;
    LS2.signal := Tank._filling==DANGEROUS;
    CTRL.signal1 := LS1.signal;
    CTRL.signal2 := LS2.signal;
    SDV1.close := CTRL.close1;
    SDV2.close := CTRL.close2;
    DV.close := CTRL.close3;
end
```

Figure 3.5: A first AltaRica model for the level control system (part 2)

To refer to a variable declared in a sub-block, one uses the dot notation. For instance, the variable `inFlow` of the block `Tank` is referred to as `Tank.inFlow` in the outer block `LevelControlSystem`.

In expressions, the symbol "==" stands for equal, while the symbol "!=" stands for different. Equations are written using the symbol ":=" which is truly an assignment, i.e. the value of the flow variable given as left-hand side is calculated by means of the expression given as the right-hand side. Equations are terminated with a semi-colon ";".

In the sequel, we shall call HSSE (hierarchical systems of symbolic equations) the subset of AltaRica 3.0 in which the model given in Figures 3.4 and 3.5 is written.

3.3.2 Grammars

When we communicate with other humans, we count on their education: we can thus assume that they know the rules governing the construction of block diagrams and that they will be able to understand what we mean even if what we wrote down does not fully obey these rules or if the rules themselves are fuzzy.

To communicate with computers, we need to make rules precise, unambiguous, because computers do not "understand" anything: again, they just apply mechanically sequences of instructions.

The set of rules governing a modeling language is called the *grammar* or the *syntax* of this language. The program, or the part of a program, that reads models at a particular syntax is called a *parser*.

Let Σ be a finite set of symbols called the *alphabet*. Strings are finite sequences of symbols of Σ. Namely, Σ together with the nullary "ε" (the empty string) and the concatenation operation "$\star$" form a *monoid* (see Appendix B). As $\star$ is associative, parentheses are not mandatory and the operator $\star$ itself can be omitted. E.g. $a \star (b \star a)$ can be simply written aba.

The set of strings over Σ is denoted $\Sigma^\star$. $\Sigma^\star$ can defined as the smallest set such that:

– The empty string ε belongs to $\Sigma^\star$.

– If a is a symbol of Σ and u is a string of $\Sigma^\star$, then ua is a string of $\Sigma^\star$.

A *language* L over Σ is a subset of $\Sigma^\star$.

A language can be the set of words of dictionary, but also the set of well-formed sentences of some sort, e.g. well-formed mathematical formulas or ... descriptions of block diagrams.

If the language is finite and small, it is possible to define it extensionally by enumerating the strings of the language. Most of the (natural or artificial) languages are infinite or at least too large

to be defined in this way. They are usually defined by a set of recursive rules forming together a generative grammar.

In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s:

Definition 3.3.1 – Generative grammar. A *generative grammar* G is a quadruple $\langle \Sigma, N, P, T \rangle$ where:
- Σ is a finite set (an alphabet) of terminal symbols.
- N is a finite set (and alphabet) of non-terminal symbols, that is disjoint from Σ.
- A finite set P of production rules, where each rule is in the form:
  $$uSv \quad \rightarrow \quad w$$
  where, $S \in N$ and $u, v, w \in (\Sigma \cup N)^\star$.
- T is a distinguished symbol of N, called the start symbol.

In the sequel, we shall say simply grammar instead of generative grammar.

A grammar $G : \langle \Sigma, N, P, T \rangle$ can be seen as a relation over $(\Sigma \cup N)^\star \times (\Sigma \cup N)^\star$:
- Let two strings $x, y \in (\Sigma \cup N)^\star$. Then $x \Rightarrow_G y$, if and only if there exist four strings $u, v, p, q \in (\Sigma \cup N)^\star$ and a rule $p \rightarrow q \in P$ such that $x = upv$ and $y = uqv$. If $x \Rightarrow_G y$, we say that y is derived in one step from x with the grammar G.
  The reflexive transitive closure of the relation $\Rightarrow_G$ is denoted by $\Rightarrow_G^\star$. If $x \Rightarrow_G^\star y$, we say that y is derived from x with the grammar G.
- The language of G, denoted by $\mathscr{L}(G)$ is the set of strings of $\Sigma^\star$ that can derived from the start symbol T:
  $$\mathscr{L}(G) \quad = \quad \{s \in \Sigma^\star ; T \Rightarrow_G^\star s\}$$

There are different types of grammars, depending on which conditions one puts on left and right members of rules. In 1956, Noam Chomsky defined a hierarchy, which is still used to classify grammars (Chomsky 1956). Grammars with only one non-terminal symbol as left-member of rules are said *context-free* because the possibility of a derivation depends only on the presence of this non-terminal symbol. Context-free grammars are less powerful than general grammars: some languages cannot be recognized by any context-free grammar. For instance, the language $\{a^n b^n c^n | n \geq 0\}$ is not. Natural languages are definitely not context-free neither. Context-free languages are much easier to parse than non context-free ones. This is the reason why most of the programming and modeling languages are either directly context-free, or easy to parse with a context-free grammar to which a second pass is added so to eliminate incorrect programs or models.

In practice, grammars are often specified using the so-called Extended Backus-Naur Form.

3.3.3 Extended Backus-Naur Form

Extended Backus–Naur Form (EBNF) is a notation technique for context-free grammars, often used to describe the syntax of not only programming and modeling languages but also document formats, instruction sets and communication protocols. It is used wherever exact descriptions of languages are needed: language specifications, manuals, and textbooks on programming language theory.
A EBNF specification is a set of derivation rules, written as:

$$\textit{symbol} \quad \texttt{::=} \quad \textit{expression}$$

where *symbol* is a non-terminal symbol and *expression* consists of one or more sequences of symbols.
EBNF expressions are built using the following conventions.
- Terminal symbols are usually surrounded with single quotes, e.g. `'block'`, `':='`.

– A sequence of expressions separated with white spaces (or tabulations) denote the concatenation of these expressions. For instance, the expression:

```
Identifier ':=' Expression ';'
```

denotes the concatenation of the non-terminal symbol `Identifier`, the terminal symbol "`:=`" the non-terminal symbol `Expression`, and the terminal symbol "`;`".
– The vertical bar symbol `|` denotes a choice. It has a lower priority than the concatenation, meaning that the expression:

```
VariableDeclaration | BlockDeclaration
```

denotes a choice between the two expressions `VariableDeclaration` and `BlockDeclaration`.
The order in which arguments of the choice are given matters. The first argument should be tried first. Then, in the case it is impossible to match the text with this argument, the second one should be try, and so on until a matching rule is found or it remains no argument to try.
– Parentheses are used for grouping expressions.
– Post-fixed operators of regular expressions `*`, `+` and `?` can used, i.e. $E\star$, $E+$ and $E?$ denote respectively any number of times, any positive number of times, and 0 or 1 time the expression E.
– The square brackets `[` and `]` are used to delimit categories of symbols such `0-9` (any digit), `a-z` (any lower case letter), and `A-Z` (any upper case letter). In particular `[ \t\n\r]` denotes any of the white space (" "), tabulation ("`\t`") and end of line symbol ("`\n`" or "`\r`").

Figure 3.6 gives the EBNF grammar for our HSSE language.

The grammar starts by describing models. Namely a model is any number of domain declarations followed by a block declaration.

Then, it defines domain declarations. A domain declaration starts with the keyword `domain`, followed by the name of the domain, followed by the set of symbolic constants constituting the domain. Constants of the set are given surrounded by curly braces and separated with commas.

The grammar defines then block declarations. A block declaration starts with the keyword `block`, followed by the name of the block, and ends with the keyword `end`. It consists in any number of variable and block declarations, possibly followed with an assertion.

And so on.

Rule line 43 asserts that an identifier (of a symbolic constant, a variable or a block) is any sequence of letters, digits, hyphen "–" and underscore "_" starting with a letter.

Keywords are reserved words and should not be used as identifiers.

Note that the EBNF grammar given in Figure 3.6 is not fully sufficient to describe well-formed reliability block diagrams. Additional checks must be performed, e.g.

– Variables and blocks declared within a block must have different names.
– Constant names must not be used for variables and blocks.
– Arguments of connectives `and`, `or` and `not`, as well as the condition of the `if-then-else` expression must be Boolean.
– There must be no loop in the definition of flow variables (a variable cannot depend on itself).
– ...

These checks are very difficult, if not impossible, to perform via the specification of the grammar of the language. They are nevertheless essential and performed in a second step.

```
Model ::=
    DomainDeclaration* BlockDeclaration

DomainDeclaration ::=
    'domain' Identifier '{' Identifier ( ',' Identifier )* '}'

BlockDeclaration ::=
    'block' Identifier
        ( VariableDeclaration | BlockDeclaration )* Assertion?
    'end'

VariableDeclaration ::=
    Domain Identifier (',' Identifier)* '(' Attribute ')' ';'

Domain ::=
    'Boolean' | Identifier

Attribute ::=
    ( 'init' | 'reset' ) '=' Atom

Assertion ::=
    'assertion' Equation+

Equation ::=
    Identifier ':=' Expression ';'

Expression ::=
        Atom
    |   Expression ( 'or' Expression )+
    |   Expression ( 'and' Expression )+
    |   'not' Expression
    |   'if' Expression 'then' Expression 'else' Expression
    |   Expression '==' Expression
    |   Expression '!=' Expression
    |   '(' Expression ')'

Atom ::=
    'false' | 'true' | Path

Path :=
    Identifier ( '.' Identifier )*

Identifier ::= [a-zA-Z][a-zA-Z0-9-_]*
```

Figure 3.6: EBNF grammar for the HSSE language

3.3.4 Grammars of Graphical Formalisms

Some popular modeling formalisms, such as SysML, are purely graphical. It would be interesting to design graphical grammars for these formalisms just as we can design textual grammars for textual formalisms. Unfortunately, it seems much more difficult to represent abstract objects graphically than textually. So far no convincing, widely accepted, way of describing graphical formalisms has been proposed. These formalisms are usually presented by means of examples and text discussing the examples. They are just notations, despite their "ML" suffix (which stands for

modeling language).

This is another argument in favor our thesis 7 that asserts that models should no be confused with their graphical representations.

3.4 Semantics

The difference between syntax and semantics is one of the most important notions of model-based systems engineering. We shall examine it in this section.

3.4.1 Preliminaries

Consider the following identity.

$$1+1 \quad = \quad 1 \tag{3.1}$$

This identity is obviously false in usual arithmetic. We have learnt since our very childhood that $1+1=2$ and that 2 is indeed different from 1. Consider however that 1 means "true" and that $+$ means "or". In this case, the identity becomes true.

In a word, the truth or falsity of the above identity depends on the way we interpret its constituents. This applies in general to any relation or set of relations.

The question here is twofold. First, we can study the truth formulas and relations independently of any particular interpretation. A whole branch of mathematical logic, model theory, is dedicated to this very purpose. Second, we can study the truth formulas and relations for a particular interpretation. This is typically what we are doing when we assign a semantics, i.e. a meaning, to computer programs.

Computer programs are artifacts. Imperative programs can be interpreted as mathematical functions transforming input data into output data. Parallel and reactive programs cannot be interpreted in this (relatively simple) way, but they can be still described as mathematical objects operating on mathematical objects. Everything that happens during the execution of the program is described by the program and stays in the realm of mathematics.

Systems engineering models are also artifacts. But they describe systems of the real world, not mathematical objects. This raises a number of difficulties. The main one stands indeed in the validation of models. The ultimate validation would be to perform experiments on the system itself, but we design models precisely to avoid performing experiments on systems. There is another, more subtle, difficulty. It stands in the confusion sometimes made by analysts between the intrinsic properties of the model, as a mathematical object, and the properties of the system echoed in the model. Analysts tend to overload the objective interpretation of the model (as a pure mathematical object) with their subjective understanding of the system, hence a dangerous source of ambiguity and potential misunderstanding between the stakeholders. This problem is worsened by the use of modeling formalisms with no clear semantics.

Fortunately, the concepts and methods developed to defined the semantics of programming languages, see e.g. (H. R. Nielson and F. Nielson 2007) for an introductory book, can be use for modeling languages as well.

There are many approaches to formal semantics, but they belong to two major classes:

- Denotational semantics, whereby each syntactic construct in the program is interpreted as a denotation, i.e. as a mathematical object inhabiting some a mathematical space.
- Operational semantics, whereby the execution of the program is described in terms of operations on an abstract machine.

Some authors consider axiomatic semantics as a third class. In the axiomatic semantics, one gives meaning to syntactic constructs by describing the logical properties that apply to them. It is often use to prove partial correctness of programs.

In any case, the distinctions between the two or three broad classes of approaches can sometimes be vague, but all known approaches to formal semantics use the above techniques, or some combination thereof. Apart from the choice between denotational or operational (or axiomatic approaches), most variations in formal semantic systems arise from the choice of the underlying mathematical formalism.

We shall apply both to define the semantics of our HSSE language. As said Section 3.2.1, block diagrams of our HSSE language are eventually interpreted as systems of symbolic equations. To define the semantics of the language, we shall proceed in three steps:

1. We shall define formally our mathematical framework, i.e. specify what is a system of symbolic equations.
2. Using operational semantics, we shall specify how to calculate the values of flow variables from the values of state variables.
3. Using denotational semantics, we shall specify how a HSSE model is translated into a system of symbolic equations.

3.4.2 Systems of Symbolic Equations

We shall first define expressions.

Definition 3.4.1 – Symbolic Expressions. Let $\mathscr{C}$ be a finite or denumerable set of symbols called constants that contains at least the two Boolean constants `false` and `true`. Let $\mathscr{V}$ be a finite or denumerable set of symbols called variables, such that $\mathscr{C} \cap \mathscr{V} = \emptyset$.

The set of *symbolic expressions* built over $\mathscr{C}$ and $\mathscr{V}$ is the smallest set such that:

- Constants of $\mathscr{C}$ and variables of $\mathscr{V}$ are symbolic expressions.
- If $e, e_1, \dots e_n$ are symbolic expressions, then so are $\texttt{or}(e_1, \dots e_n)$, $\texttt{and}(e_1, \dots e_n)$, $\texttt{not}(e)$, $\texttt{if}(e_1, e_2, e_3)$, $e_1 = e_2$ and $e_1 \neq e_2$.

The set of variables that occur in a expression e is denoted $\mathrm{var}(e)$. It can be defined by *structural induction*. Let $c \in \mathscr{C}$, $v \in \mathscr{V}$ and $e, e_1, \dots e_n$ be symbolic expressions. Then:

- $\mathrm{var}(c) = \emptyset$,
- $\mathrm{var}(v) = \{v\}$,
- $\mathrm{var}(\texttt{or}(e_1, \dots e_n)) = \mathrm{var}(\texttt{and}(e_1, \dots e_n)) = \bigcup_{i=1}^{n} \mathrm{var}(e_i)$,
- $\mathrm{var}(\texttt{not}(e)) = \mathrm{var}(e)$,
- $\mathrm{var}(\texttt{if}(e_1, e_2, e_3)) = \mathrm{var}(e_1) \cup \mathrm{var}(e_2) \cup \mathrm{var}(e_3)$,
- $\mathrm{var}(e_1 = e_2) = \mathrm{var}(e_1 \neq e_2) = \mathrm{var}(e_1) \cup \mathrm{var}(e_2)$.

We can now define systems of symbolic equations.

Definition 3.4.2 – Systems of symbolic equations. Let $\mathscr{C}$ be a finite or denumerable set of symbols called constants that contains at least the two Boolean constants `false` and `true`. A *system of symbolic equations* built over $\mathscr{C}$, is a pair $\langle V, E \rangle$ where:

- V is a finite set of symbols called *variables*. Each variable v of V takes its value into a finite subset of $\mathscr{C}$ called the domain of v and denoted $\mathrm{dom}(v)$.
- E is a finite set of equations, i.e. of pairs (v, e), where v is a variable of v and e is a symbolic expression built over $\mathscr{C}$ and V. The equation (v, e) is denoted $v := e$.

The set E must verify in addition that for all variable v of V, there must be at most one equation of E whose left hand side is the variable v.

Variables that show up only in right members of equations are called *state variables*. Variables that show up as the left member of an equation are called *flow variables*.

We can now define the dependency relation among variables.

Definition 3.4.3 – Dependent variables. Let $S : (V,E)$ be a system of symbolic equations, and let v and w be two variables of V. Then v *depends on* w in S if v is a flow variable, left member of the equation $v := e$ of E, and one of the two following conditions holds.
- $w \in \mathrm{var}(e)$.
- There exists a variable $u \in \mathrm{var}(e)$ such that u depends of w.

Eventually, we can define the data-flow property.

Definition 3.4.4 – Data-flow systems of symbolic equations. Let $S : (V,E)$ be a system of symbolic equations. Then S is *looped* if there is a variable v of V that depends on itself. It is *loop-free* or *data-flow* otherwise.

From now on, we shall only consider data-flow systems of symbolic equations, unless specified otherwise.

3.4.3 Operational Semantics

The *operational semantics* focuses on the successive steps of the execution of a program. In particular, the basic idea behind *structured operational semantics*, introduced by Plotkin (Plotkin 2004), is to define the behavior of a program in terms of the behavior of its parts, thus providing a structural, i.e. syntax-oriented and inductive, view on operational semantics. This is achieved by giving rules using the following notation, called a *sequent*.

$$R: \frac{C, s}{\langle I, s\rangle \longrightarrow t}$$

This sequent reads as follows: by rule R, if a certain condition C is verified in the current state s, then executing the instruction I in s transforms the state s into the state t.

We shall illustrate that onto our HSSE language. We need first to define variable assignments.

Definition 3.4.5 – Variable assignments. Let $S : (V,E)$ be a system of symbolic equations. A *variable assignment* of V is a function σ that associates a constant $\sigma(v)$ of $\mathrm{dom}(v)$ with each variable v of V.

A variable assignment σ is *partial* if associates a value only to a subset of the variables of V. For the sake of convenience, we shall write $\sigma(v) = \perp$ to denote that σ does not assign a value to v.

Variable assignments are lifted up into functions from symbolic expressions to constants as follows. Let V be a set of variables and let σ be a possibly partial assignment of V. Then,

$$\begin{aligned}
\sigma(c) &= c \text{ for all constants } c \in \mathscr{C} \\
\sigma(\mathtt{or}(e_1,\ldots e_n)) &= \begin{cases} \mathtt{true} & \text{if at least one of the } e_i \text{ is such that } \sigma(e_i) = \mathtt{true} \\ \mathtt{false} & \text{if all of the } e_i\text{'s are such that } \sigma(e_i) = \mathtt{false} \\ \perp & \text{otherwise} \end{cases} \\
\sigma(\mathtt{and}(e_1,\ldots e_n)) &= \begin{cases} \mathtt{false} & \text{if at least one of the } e_i \text{ is such that } \sigma(e_i) = \mathtt{false} \\ \mathtt{true} & \text{if all of the } e_i\text{'s are such that } \sigma(e_i) = \mathtt{true} \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

$$\sigma(\texttt{not}(e)) = \begin{cases} \texttt{false} & \text{if } \sigma(e) = \texttt{true} \\ \texttt{true} & \text{if } \sigma(e) = \texttt{false} \\ \bot & \text{otherwise} \end{cases}$$

$$\sigma(\texttt{if}(e_1,e_2,e_3)) = \begin{cases} \sigma(e_2) & \text{if } \sigma(e_1) = \texttt{true} \\ \sigma(e_3) & \text{if } \sigma(e_1) = \texttt{false} \\ \bot & \text{otherwise} \end{cases}$$

$$\sigma(e_1 = e_2) = \begin{cases} \texttt{false} & \text{if } \sigma(e_1) \neq \bot, \sigma(e_2) \neq \bot \text{ and } \sigma(e_1) \neq \sigma(e_2) \\ \texttt{true} & \text{if } \sigma(e_1) \neq \bot, \sigma(e_2) \neq \bot \text{ and } \sigma(e_1) = \sigma(e_2) \\ \bot & \text{otherwise} \end{cases}$$

$$\sigma(e_1 \neq e_2) = \begin{cases} \texttt{false} & \text{if } \sigma(e_1) \neq \bot, \sigma(e_2) \neq \bot \text{ and } \sigma(e_1) = \sigma(e_2) \\ \texttt{true} & \text{if } \sigma(e_1) \neq \bot, \sigma(e_2) \neq \bot \text{ and } \sigma(e_1) \neq \sigma(e_2) \\ \bot & \text{otherwise} \end{cases}$$

During the calculation of the value of flow variables, the current variable assignment is modified step wisely. We shall use the following notation.

Definition 3.4.6 – Extension of variable assignments. Let V be a finite set of variables, σ be a, possibly partial, assignment of V, v be a variable of V and finally c be a constant in $\text{dom}(v)$. Then, $\sigma[v = c]$ denotes the variable assignment such that:

$$\sigma[v = c](w) = \begin{cases} c & \text{if } w = v \\ \sigma(w) & \text{otherwise} \end{cases}$$

Moreover, $\sigma[v_1 = c_1, \ldots v_n = c_n]$ denotes $\sigma[v_1 = c_1] \cdots [v_n = c_n]$.

We can now use the structural operational semantics to describe how the values of flow variables are calculated from the values of state variables. In our case, the instructions of the "program" are the equations of our system $S : \langle V, E \rangle$ and its state is the current variable assignment σ. The main rule is as follows.

$$\text{Propagate: } \frac{v := e \in E \qquad \sigma(v) = \bot \qquad \sigma(e) \neq \bot}{\langle v := e, \sigma \rangle \longrightarrow \sigma[v = \sigma(e)]}$$

The above rule actually suffices: starting from the initial assignment, it is iterated until all flow variables that can be given a value are given a value, i.e. a *fixpoint* is reached. The following property holds.

Property 3.1 – Uniqueness of flow variable assignments. Let $S : (V, E)$ be a system of symbolic equations and let σ be a partial variable assignment that gives a value to all state variables but to no flow variable of V. Let τ be the variable assignment obtained by repeatedly applying the above rule Propagate until a fixpoint is reached. Then,

i) τ assigns a value to all flow variables.

ii) τ is unique, i.e. is the same whatever the order in which equations are considered.

Proof: The application of the rule propagate will consider first flow variables that depend only on state variables, then flow variables that depend on state variables and flow variables already calculated and so on. Thanks to the data-flow property, this process converges towards a unique variable assignment.

Note that it is possible to order the equations in such way that the calculation process consider them in turn. This is because the dependence relation is a partial order and any partial order can be turned into a total order. This is related to unit propagation mechanisms in constraint satisfaction

problems in general, in the SAT problem that we shall study in Chapter 9 in particular (Dowling and Gallier 1984).

Note also that the above developments assume that the system of symbolic equations is well typed, i.e. that for each equation $v := e$ and each total variable assignment σ, $\sigma(e) \in \mathrm{dom}(v)$. Moreover, expressions themselves must be well typed, e.g arguments of Boolean operations and of the condition of if-then-else expressions take either the value `false`, or the value `true` or are unassigned (take the value $\perp$).

The structural operational semantics is very well suited to describe dynamic behaviors, in particular behaviors of state automata that play a very important role in the framework of model-based systems engineering.

3.4.4 Denotational Semantics

The *denotational semantics* focuses on the effect of executing programs. It associates typically a function with each imperative program. This function is built by considering recursively each syntactic construct of the program and composing bottom-up these functions.

The denotational semantics approach aims thus at defining in a constructive way which mathematical object is associate to a program. It can be seen as a function that associates with each program P a function $[\![P]\!]$ (the notation $[\![.]\!]$ is characteristic of the approach).

As an illustration, we shall consider again our modeling language HSSE. The denotational semantics of HSSE describes how to associate a system of symbolic equations with a HSSE model.

We need first to introduce the prefix operation on identifiers.

Definition 3.4.7 – Prefix operation. Let p and q be two identifiers, then $p \odot q$ denotes the identifier $p.q$. The empty identifier is denoted ε and we pose $p \odot \varepsilon = \varepsilon \odot p = p$.

By extension, let p be an identifier, $c \in \mathscr{C}$, $v \in \mathscr{V}$ and $e, e_1, \ldots e_n$ be symbolic expressions. Then:

$$\begin{aligned}
p \odot c &\stackrel{def}{=} c \\
p \odot v &\stackrel{def}{=} p.v \\
p \odot \mathtt{or}(e_1, \ldots e_n) &\stackrel{def}{=} \mathtt{or}(p \odot e_1, \ldots p \odot e_n) \\
p \odot \mathtt{and}(e_1, \ldots e_n) &\stackrel{def}{=} \mathtt{and}(p \odot e_1, \ldots p \odot e_n) \\
p \odot \mathtt{not}(e) &\stackrel{def}{=} \mathtt{not}(p \odot e) \\
p \odot \mathtt{if}(e_1, e_2, e_3) &\stackrel{def}{=} \mathtt{if}(p \odot e_1, p \odot e_2, p \odot e_3) \\
p \odot e_1 = e_2 &\stackrel{def}{=} p \odot e_1 = p \odot e_2 \\
p \odot e_1 \neq e_2 &\stackrel{def}{=} p \odot e_1 \neq p \odot e_2
\end{aligned}$$

Now we shall write the rules describing how a declaration D transforms in the context p, i.e. with the prefix p, a system of symbolic equations $\langle V, E \rangle$ into an extended system of symbolic equations $\langle V', E' \rangle$. These rules will take the following form.

$$[\![D]\!]_p(\langle V, E \rangle) \quad = \quad \langle V', E' \rangle$$

For the sake of simplicity, we shall not consider the management of the domains of the variables.

The first rule regards indeed the declaration of blocks. It works as follows.

$$[\![\mathtt{block}\ n\ Ds\ A\ \mathtt{end}]\!]_p(\langle V, E \rangle) \quad = \quad [\![Ds\ A]\!]_{p \odot n}(\langle V, E \rangle) \tag{3.2}$$

In the above rule, n stands for the name of the block, Ds for its list of declarations of variables and blocks, and A for its assertion. Both Ds and A may be empty.

The rule for the declaration of variables is as follows.

$$[\![d\ n\ a;]\!]_p(\langle V,E\rangle) \quad = \quad \langle V \cup \{p \odot n\}, E\rangle \tag{3.3}$$

In the above rule, d stands for the domain of the variable, n for its name, and a for its attributes. The rule for the equations is as follows.

$$[\![v := e;]\!]_p(\langle V,E\rangle) \quad = \quad \langle V, E \cup \{p \odot n := p \odot e\}\rangle \tag{3.4}$$

Finally, we can give the rules to manage lists of declarations.

$$[\![]\!]_p(\langle V,E\rangle) \quad = \quad \langle V,E\rangle \tag{3.5}$$

$$[\![D\ Ds]\!]_p(\langle V,E\rangle) \quad = \quad [\![Ds]\!]_p([\![D]\!]_p(\langle V,E\rangle)) \tag{3.6}$$

Rule 3.5 gives the semantics of an empty list of declarations, while rule 3.6 gives the semantics of a list of declarations that contains at least one element.

With these five rules, the transformation of a HSSE model into a system of symbolic equations is fully and formally described.

3.4.5 Discussion

The semantics of a program or a model is not necessarily something as simple as what we have seen above. Much more complex mathematical objects can be involved. However, the above development gives a hint on the approach.

The key point here is that the formal definitions we gave in this section can be easily turned into computer programs. What these programs are doing is defined without ambiguity.

Although simple, systems of symbolic equations are sufficient to "play" use cases. In our example, it is easy to check that:

- If we set the variable `Tank._filling` to the value `REGULAR`, the sensors do not send any signal, consequently valves `SDV1` and `SDV2` are open and the valve `DV` is closed.
- If we set it to the value `HIGH`, then the sensor `LS1` sends a signal to the controller but not the sensor `LS2`, consequently the valves `SDV1` and `DV` are closed and the valve `SDV2` is open.
- If we set it to the value `DANGEROUS`, then both sensors send a signal to the controller, consequently the valves `SDV1` and `SDV2` are closed and the valve `DV` is open.
- Finally, If we set it to the value `EMPTY`, then the situation is the same as when it takes the value `REGULAR`.

When addition state variables are introduced, typically to represent the internal state of components, e.g. working or failed, it is possible to play more complex scenarios in which components do not react as they are expected to. This is the subject of problem 3.11.

Playing use cases is of special interest when the system under study has different modes of operation. Not only this makes it possible to check individually each mode of operation, but even more interestingly, this makes it possible to represent the dynamic of changes of modes.

3.5 Pragmatics

3.5.1 Pragmatic Competence

In linguistic, pragmatics studies how the context of a discourse contributes to its meaning. Unlike semantics, which examines the conventional meaning that is "encoded" in the discourse, pragmatics studies how the transmission of meaning depends not only on structural and linguistic knowledge of the speaker and listener, but also on the context in which the discourse is delivered. This study involves the analysis of the pre-existing knowledge of both the speaker and the listener, the intent of the speaker... With that respect, pragmatics attempts to explain how actors are able to overcome

apparent ambiguities. The ability to understand another speaker's intended meaning is called pragmatic competence.

Pragmatic competence plays a very important role in systems engineering as when stakeholders are communicating, they share in general a lot of knowledge about the system under study.

To illustrate this point, let us consider a very simple example.

■ **Example 3.1** Consider the following description.

> The parking lot contains three places numbered from 1 to 3. Three vehicles are parked there: a Jeep Cherokee, a Renault Zoe and a Toyota Prius. The Japanese car is parked to the left of the SUV and to the right of the electric car.

This description seems already clear and unambiguous: the Renault Zoe is parked on place 1, the Toyota Prius on place 2 and the Jeep Cherokee on place 3. It can be formalized using one of the many existing modeling languages.

Well, is this so clear and so unambiguous?

On the one hand, anybody with a minimum knowledge about cars understands the description and interprets it correctly. On the other hand, this "minimum knowledge" may be not that small after all. Let us review some of the implicit assumptions.

- Vehicles park on places and cannot park elsewhere.
- At most one vehicle can park on each place.
- If the place i is at the left of place j, then the place j is at the right of place i.
- Places are numbered from left to right, i.e. the place 1 is at the left of place 2 which is itself at the left of place 3.
- Places are aligned: the place 3 is not at the left of the place 1.
- If a vehicle is parked at place i, which is at the left (respectively right) of place j, then the vehicle is at the left (respectively right) of place j and the vehicle that may be parked on this place.
- A place or a vehicle cannot be at the left, or at the right, of itself.
- A Japanese car is a vehicle.
- A SUV is a vehicle.
- An electric car is a vehicle.
- A Renault Zoe is an electric car.
- A Jeep Cherokee is a SUV.
- A Toyota Prius is a Japanese car.
- A vehicle with an electric engine and a thermic engine is hybrid and therefore does not enter into the category of electric cars.
- Unless specified otherwise, a vehicle is not a Japanese car (respectively a SUV, an electric car).
- ...

■

The above example shows that, even a very simple description can involve many implicit assumptions, i.e. cannot be understood without referring to the context. Stakeholders are assumed to share an understanding of this context. In a word, the description is pragmatic.

Most of the above implicit assumptions are "obvious", e.g. if A is at the left of B, then B is at the right of A. Some are not, e.g. a Toyota Prius is not an electric car, despite it has an electric engine. Some are general truths, e.g. an object cannot be at the left of itself. Some are particular to this description, e.g. places are numbered from left to right. In a word, the context involves knowledge of very different levels of generality and abstraction.

3.5.2 The Danger of Pseudo-Formal Descriptions

In a very simple example as the one above, it is possible to make precise every detail so to get eventually a fully formal description, e.g. in form of a constraint satisfaction problem. But precisely: even for such simple example, making every detail precise is a process. This process has a cost. If the description serves only for communication purposes, it would be not only costly but also counter-productive to overload it by making precise all details. The bare description suffices. If, on the contrary, some calculation must be performed, e.g. to check that there is at least one way to park the vehicles that fulfills all of the constraints, then making precise all details is mandatory.

Nevertheless using mathematical formulas to describe constraints does not warranty to get a formal description. Consider, for instance, the following statement.

$$\forall p_i, p_j \ \text{place}(p_i) \wedge \text{place}(p_j) \wedge i < j \Rightarrow \text{left}(p_i, p_j) \tag{3.7}$$

This statement is an attempt to make formal the idea that places are numbered from left to right. It has however several flaws, e.g. the domain and the meaning of subscripts are kept implicit. But more importantly, the proof system used to make deductions on such formula is not defined, therefore there is no warranty that the statement can be used for any purpose.

In a word, pseudo-formalisation like statement 3.7 are useless. They are even dangerous, as they may give stakeholders the false impression that the description is mathematically proven, therefore undoubtedly true, while it is not the case at all.

3.5.3 Pragmatic Models

Assume we want to achieve a fully formal description of our problem. As way to do that would be to say that each object "car" has a number of characteristics including:

- Its origin {France, Germany, Japan, USA, ...}.
- Its brand {Fiat, Jeep, Renault, Toyota, Volkswagen, ...}.
- Its series {Golf, Prius, Zoe, ...}.
- Its category {microcar, compact, berline, minivan, SUV, ...}.
- Its motorization {thermic, electric, hybrid, ...}.
- Its location, i.e. the number of the place in the parking lot at which it is parked $\{1,2,3,\ldots\}$.

We have then some generic knowledge about cars, i.e. objects belonging to a certain set Cars = {Car1, Car2, ...}:

$$\forall c \in \text{Cars}, c.\text{brand} = \text{Toyota} \Rightarrow c.\text{origin} = \text{Japan}$$
$$\forall c \in \text{Cars}, c.\text{series} = \text{Zoe} \Rightarrow c.\text{brand} = \text{Renault} \wedge c.\text{motorization} = \text{electric}$$
$$\vdots$$

We have also some generic knowledge about (places in) the parking lot:

$$\forall p, q \in \text{Places}, \text{left}(p,q) \Leftrightarrow \text{right}(q,p)$$
$$\text{left}(1,2)$$
$$\text{left}(2,3)$$
$$\vdots$$

We have finally some specific knowledge about the way our three cars are parked:

$$\text{car1.brand} = \text{Jeep}, \text{car1.series} = \text{Cherokee}$$
$$\text{car2.brand} = \text{Renault}, \text{car2.series} = \text{Zoe}$$
$$\text{car3.brand} = \text{Toyota}, \text{car3.series} = \text{Prius}$$
$$\forall c1, c2 \in \{\text{car1}, \text{car2}, \text{car3}\}$$
$$c1.\text{origin} = \text{Japan} \wedge c2.\text{category} = \text{SUV} \Rightarrow \text{left}(c1.\text{location}, c2.\text{location})$$
$$c1.\text{origin} = \text{Japan} \wedge c2.\text{motorization} = \text{electric} \Rightarrow \text{right}(c1.\text{location}, c2.\text{location})$$

The above specification is formal. Every construct involved in the description is well defined. As all variables (the car characteristics) belong to finite sets, the description can be eventually translated into a propositional calculus formula. Then, one can look for variable valuations that satisfy this formula.

In our example, the problem is sufficiently simple to obtain a set of constraints with a unique solution. In general, it is not the case. There are two main reasons for that:

- Some constraints have been omitted, because it would be much to tedious and error-prone to add them all.
- The problem at stake is under-specified.

It remains that, even though the description is incomplete or under-specified, it may prove to be very useful in practice. The analysts know actually "*where to look for solutions*", something that a computer cannot do. The model is sufficient for a good communication among the stakeholder and for a "human" exploration of the solution space, although mathematically too loose to be automatically processed by a machine. In a word, it is *pragmatic*.

Note that pragmatic, in the sense we define this term here, is not necessarily opposed to formal. More precisely, there may be no difference in essence between a pragmatic and a formal model: they may be both written with the same modeling language. The difference stands in the way the two models are used and assessed. Formal models are designed to be automatically and completely assessed by a mechanized procedure implemented on a computer. Pragmatic models are designed to be assessed by humans.

This does not mean however that computers cannot be used to assess pragmatic models. Their use is however limited to some specific tasks. For instance, they can verify that the solution proposed by human analysts matches all of the constraints.

3.6 Further Readings

Some classical books about grammars and parsers:

- The book by Aho, Sethi and Ullman about compiler construction (Aho et al. 2006).
- The book by Hopcroft and Ullmann about automata theory, languages, and computation (Hopcroft, Motwani, and J. D. Ullman 2006).
- The book by Parr about the parser generator ANTLR (Parr 2013).

Some classical books about semantics of programs:

- The introduction by Meyer (Meyer 1990).
- The introduction by Nielson and Nielson (H. R. Nielson and F. Nielson 2007).

Some books about linguistic:

- The timeless "Syntactic Structures" (Chomsky 1957).
- A book about the semantics and the pragmatics of natural languages (Cruse 2011).

Finally,

- One of the very rare books about the pragmatics of modeling languages (Fuhrmann 2011).

3.7 Exercises and Problems

This section proposes three series of exercises and problems: the first series about syntax and semantics, the second one about systems of symbolic and numerical equations, the third one about architecture of systems.

3.7.1 Syntax and Semantics

Exercise 3.1 – Palindroms. A palindrome is a sentence that reads the same in both directions. E.g.

A man, a plan, a canal - Panama!
Madam, I'm Adam.
Never odd or even.
Was it a car or a cat I saw?

Question 1. Write an EBNF grammar that recognizes palindromes (and only these sentences).

■

Exercise 3.2 – Infix notation. The infix notation is used in some programming languages such as LISP or Scheme (Abelson, G. J. Sussman, and J. Sussman 1996). It consists in writing operations under the following form:

$$(Operator\ Operand_1 \cdots Operand_n)$$

Although a bit strange at a first glance, this notation proves to be easy to parse and solves issues related to priorities of operators. For instance, $(+\ 2\ (*\ 3\ 4))$ is interpreted without ambiguity as $2+(3*4)$ and not as $(2+3)*4$.

In this exercise, we shall describe the syntax and the semantics of a calculator relying on the infix notation. The calculator implements only two instructions, themselves in infix notation:

- (`set`*VariableExpression*), which possibly creates the variable and sets its value as the value of the given expression; and
- (`print`*Variable*), which prints the value of the given variable.

Question 1. Design an EBNF grammar for the input language of the calculator. The calculator must implement at least the usual arithmetic expressions (addition, subtraction...).

Question 2. Design an operational semantics for the calculator.

■

Problem 3.3 – Pizza Process. Figure 3.7 shows a diagram representing a business process, namely the one of ordering and eating a pizza at the restaurant.

This business process involves three actors: the customer, the waiter and the Chef. The diagram can be seen as a swimming pool where each actors "swims" into a swimming lane. The diagram describes tasks that are performed in sequence by these actors. Tasks are represented by rounded rectangles. Precedence between tasks, i.e. which task follows which other, are represented with arrows. Initial and final states of the process are represented by circles. Finally, diamonds represent binary choices. They are called choice gates. In the process described Figure 3.7, there is only one choice gate, which asks the question whether the customer got his pizza or not.

Diagrams such as the one of Figure 3.7 are very convenient to visualize business processes. However, we want to make a textual description of such diagrams.

Question 1. Design a textual grammar to describe business processes. The grammar should be made of commands, each command introducing an element of the business process. Give states, tasks and gates an identifier, e.g. `InitialState`, `T1`, `T2` ..., in addition to their label, e.g. `'Hungry'`, `'Order Pizza'`... Introduce a command to describe precedences

(arrows), e.g. `next InitialState T1`.

Question 2. Using your grammar, describe the business process represented Figure 3.7.

Question 3. Define the semantics of your language, as the set of possible executions.

■

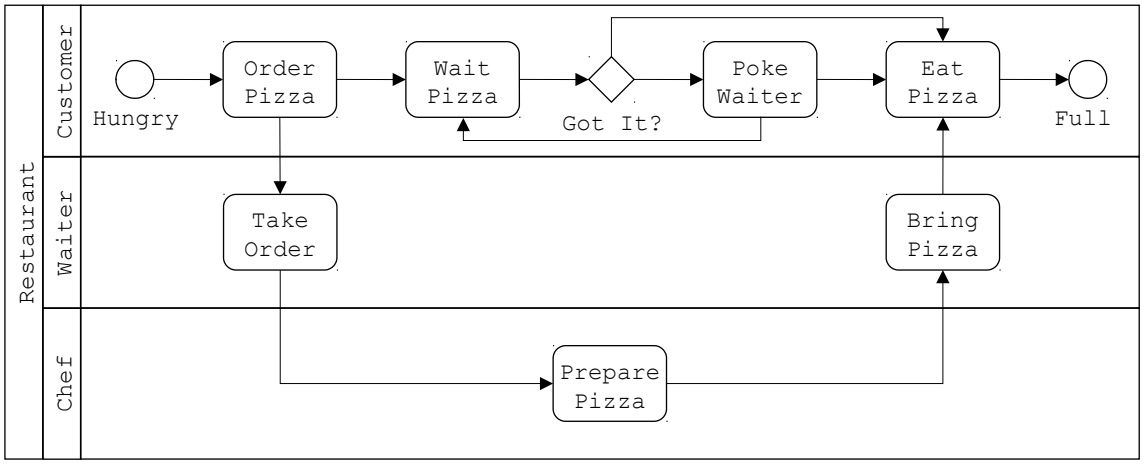


Figure 3.7: The pizza business process

Exercise 3.4 – Denotational semantics of queues. We shall consider queues, like the ones at the cashier in shops or administrations. Such queues work according to the first in, first out (FIFO) principle: the first client who arrives is served first. From an abstract point of view, three operations can be performed on queues:

- `ResetQueue`(Q) that creates an empty queue Q (or flushes it if it already exists).
- `EnQueue`(Q,c) that adds the client c at the end of the queue Q.
- `DeQueue`(Q) that removes the first client of the queue Q.

Using these three operations, we can create programs that create and operate queues. E.g.

```
1 ResetQueue(Q)
2 EnQueue(Q, c1)
3 EnQueue(Q, c2)
4 Dequeue(Q)
5 EnQueue(Q, c1)
```

This programs works as follows.

- The instruction line 1 creates a new queue named `Q`: `Q` $= []$.
- The instruction line 2 adds the client `c1` to the queue `Q`: `Q` $= [$`c1`$]$.
- The instruction line 3 adds the client `c2` to the queue `Q`, which contains now two clients in order: `Q` $= [$`c1`$,$`c2`$]$.
- The instruction line 2 removes the client `c1` from the queue: `Q` $= [$`c2`$]$.
- Finally, the instruction line 5 adds the client `c1` to the queue `Q`: `Q` $= [$`c2`$,$`c1`$]$.

Question 1. Propose an denotational semantics for the language made of the three above operations.

Question 2. Calculate the denotation of the following program.

```
ResetQueue(Q1) ResetQueue(Q2) ResetQueue(Q3)
EnQueue(Q1, c3) EnQueue(Q1, c2) EnQueue(Q1, c1)
DeQueue(Q1) EnQueue(Q3, c1)
DeQueue(Q1) EnQueue(Q2, c2)
DeQueue(Q3) EnQueue(Q2, c1)
DeQueue(Q1) EnQueue(Q3, c3)
DeQueue(Q2) EnQueue(Q1, c1)
DeQueue(Q2) EnQueue(Q3, c2)
DeQueue(Q1) EnQueue(Q3, c1)
```

■

Exercise 3.5 – Nim game. The Nim game is a two players games. The initial position is made of three heaps containing respectively 1, 3 and 5 stones. Each player takes in turn either 1, 2 or 3 stones in one of the heap. The player who takes the last stone loses the game.

Question 1. Describe all possible positions of this game as a Cartesian product.

Question 2. Design the EBNF grammar of a language making it possible to describe a game, i.e. the sequence of moves that lead from the initial to the final position.

Question 3. Define the operation semantics of your language.

Question 4. Describe lost positions as a unary relation lost(P) over positions, the find a recursive formulation of the winning and losing positions.

Question 5. Using this recursive formulation, determine whether there is a winning strategy for the first player.

■

Problem 3.6 – Robby and the Maze. Robby, your robot, should explore the maze pictured Figure 3.8 to collect the coins (represented circles labeled with a number) which are located in the different rooms. His exploration must obey the following rules.

1. He is in one cell at a time.
2. He can move east, south, west and north at will, but he cannot cross the walls.
3. He can collect an item if he is located in the same cell as this item.
4. He must collect coins in order.
5. The number of stars he has collected must be always greater than the number of coins he has collected.

You have to design a language to program Robby's quest.

Question 1. Design commands to describe the maze. Write the EBNF grammar of these commands. [Hints:] Each cell has two coordinates. Moreover, there may be walls to its east, south, west and north.

Question 2. Design commands to add objects in the maze. Write the EBNF grammar of these commands.

Question 3. Design commands to perform Robby's actions (movements and item collection). Write the EBNF grammar of these commands.

Question 4. Write the operational semantics of your language.

Question 5. Play a few scenarios using your language.

■

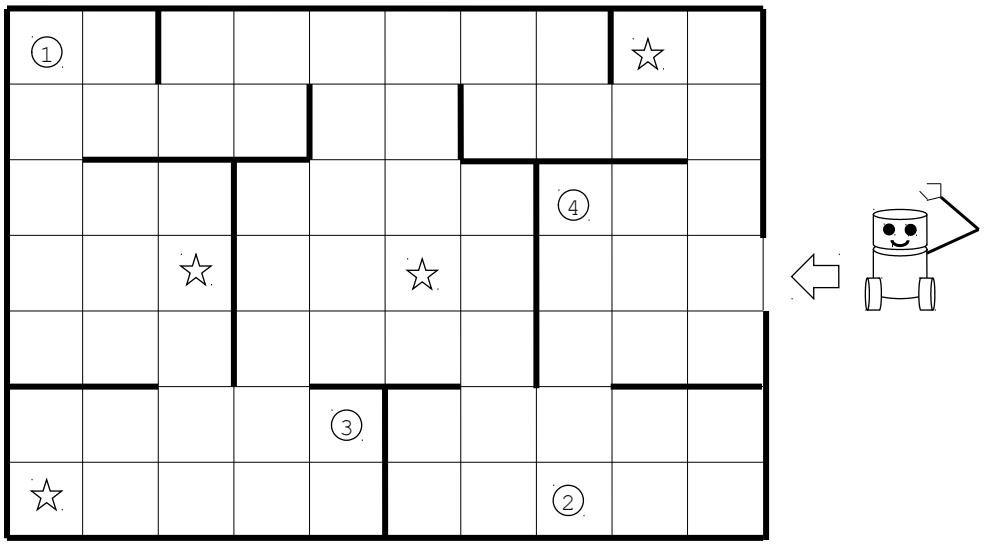


Figure 3.8: Robby and the Maze

3.7.2 Systems of Equations

With the following exercises and problems, we shall explore a bit more the expressive power of systems of equations. We shall generalize the systems of symbolic equations seen in this chapter by allowing numerical variables and arithmetic operations (addition, subtraction, multiplication, division, minimum, maximum...) with their usual syntax and semantics.

Exercises and problems proposed in this section should be solved using AltaRica 3.0 and the AltaRica Wizard integrated modeling environment.

Exercise 3.7 – Quadratic Equation. This little exercise aims at familiarizing you with the AltaRica 3.0 syntax and the use of AltaRica Wizard.

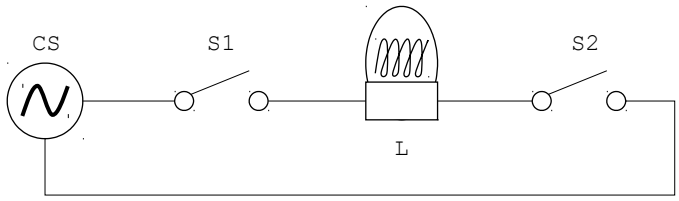
Question 1. Design a model that solves the quadratic equation $ax^2+bx+c=0$. The coefficients a, b and c will be state variables of your model. The model will have three outputs, i.e. it must calculate (at least) the value of three flow variables: `numberOfSolutions`, `solution1` and `solution2`, with their obvious meaning.

Question 2. Use the model designed in the previous question to calculate the solutions of the following equations:

- $x^2-4x-5=0$
- $-\frac{1}{2}x^2-\frac{11}{3}x-\frac{7}{6}=0$

■

Exercise 3.8 – Electric Circuit. Consider the simple electric circuit pictured below.



Question 1. Design an AltaRica model for this circuit.

Hint: Use Boolean variables to represent whether the current is circulating into the circuit. You will need two sets of variables: one to represent the circulation from left to right and another one to represent the circulation from from right to left.

Question 2. Assume that switches are automatically open and closed at the following times.

Switch/Hour	0:00	7:00	13:00	17:00	23:00
S1	closed	closed	open	closed	closed
S2	open	closed	closed	closed	open

Extend your model to take this situation into account. use your model when the light is on.

■

Exercise 3.9 – Attack Tree. Figure 3.9 shows an attack tree. Attack trees have been introduced by Schneier (Schneier 1999) to model threats against computer systems.

In the attack tree of the figure, one consider two types of attacks: those requiring a special equipment, and those that do not. Of course, if an attack is possible without special equipment, it is *a fortiori* possible with.

Each type of attack has a cost (some attacks are however impossible without a special equipment).

There are three types of nodes in the tree:

- Leaves, which represent basic attacks, may require or not a special equipment and have a cost.
- "Or" internal nodes (default), which represent the different possibilities to perform an attack. They may describe also an cost attached to attacks with and without special equipment.
- "And" internal nodes (pictured with an ellipse), which represent how threats can be combined to perform an attack. They may describe also an cost attached to attacks with and without special equipment.

Question 1. Design an AltaRica model to encode the attack tree of Figure 3.9.

Question 2. Use your model to calculate the minimum cost of an attack with and without special equipment.

■

Exercise 3.10 – Manifold. Figure 3.10 shows a (simplified) manifold, as one can find in oil and gas extraction fields.

Question 1. Design an AltaRica model for wells and valves. Assume that the flow rate of fluid (mix of oil, gas and water) coming from wells is a some real number.

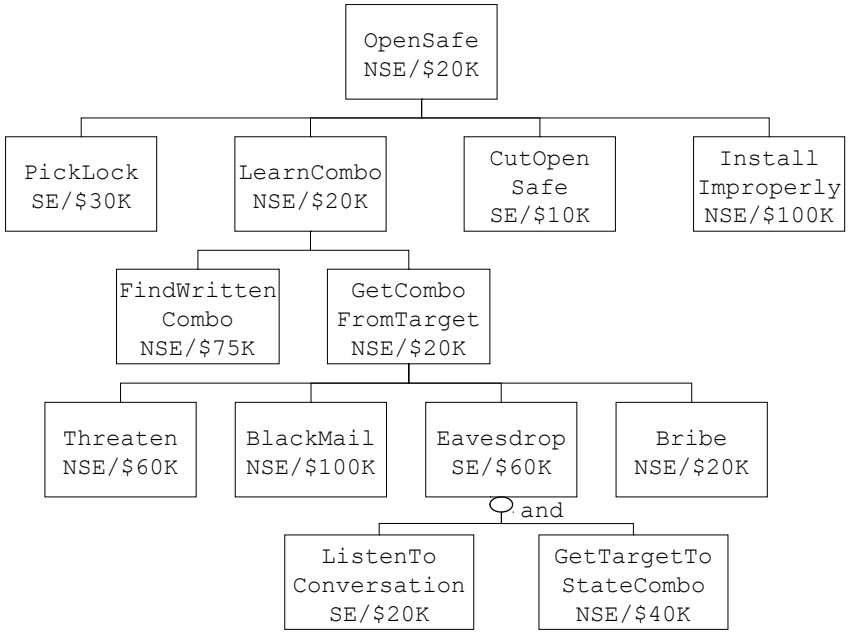


Figure 3.9: An attack tree

Question 2. Using the models designed in the previous section, design an AltaRica model to calculate the flow rate of fluid going from the wells to the production facility and to the test separator, depending on the states of the valves.

Question 3. Test your model using the AltaRica Wizard environment, by representing the following 24 hour schedule on the configuration of the manifold. In the following table, P stands for production and T for test. This means that the flow of fluid coming from well 1 is going to the production all the time but from 7:00 to 8:00 where it is diverted to the test separator.

Switch/Hour	0:00	7:00	8:00	15:00	16:00	23:00	24:00
Well 1	P	T	P	P	P	P	P
Well 2	P	P	P	T	P	P	P
Well 3	P	P	P	P	P	T	P

■

3.7.3 Architecture of Systems

Problem 3.11 – Level control system (complements). In the AltaRica model of the level control system proposed in Figure 3.4 and 3.5, we assumed that the sensors, the controller and the valves work correctly, which is indeed quite optimistic (and the reason why the draining line has been added to the system).

Question 1. Introduce an additional structuring layer, as in the functional architecture described in Figure 2.7 (page 31) or in the physical architecture described in Figure 2.9 (page 33).

Question 2. Introduce variables that describe the state (working or failed) of sensors, and modify assertions accordingly.

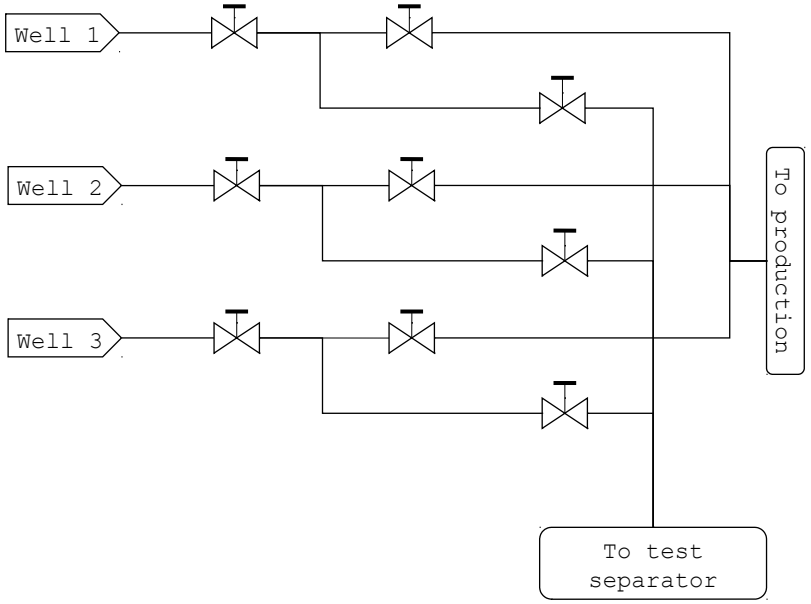


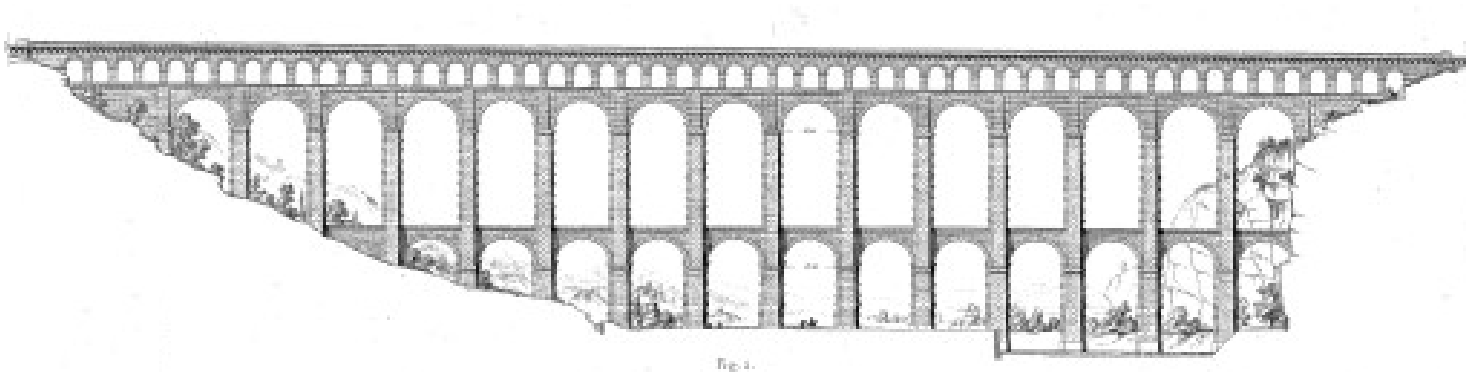
Figure 3.10: A simplified manifold

Question 3. Same question for valves.

Question 4. Same question for the controller. Assume that, when failed, the controller does not send any order to valve. Discuss your assumptions.

Question 5. Using your model, implement several use cases by changing the values of state variables.

■



4. Architecture of Models

Key Concepts

- Domain specific modeling language
- Blocks, ports and connections
- Expressions
- Prototypes and classes
- Instantiation and cloning
- Fundamental relations between elements of a model
  - "interacts-with" (connection)
  - "is-part-of" (composition)
  - "is-a" (inheritance)
  - "uses" (aggregation)
- Models as scripts, directives
- Polymorphism, parameters
- Absolute and relative references
- Functional chains
- Operators and Functions

This chapter discusses constructs to structure of models and to model structures. To do so, it introduces S2ML (system structure modeling language), a small modeling language that provides only structuring mechanisms (Batteux, Prosvirnova, and Rauzy 2018). S2ML can be seen in two ways. First, as a modeling tool on its own, dedicated to structural descriptions. With respect to this first vision, S2ML clarifies and generalizes constructs found in other formalisms with the same purpose, including the structural diagrams of SysML (internal block diagrams and block definition diagrams) (Friedenthal, A. Moore, and Steiner 2011). Second and more importantly, as a complete and versatile set of structuring constructs that can be applied to any mathematical framework. In other words, given any such mathematical framework X, it is possible to design a high level, object-oriented modeling language S2ML+X.

We shall use the S2ML+X paradigm in the whole book to design a family of domain specific

modeling languages. S2ML can be seen as a (re-)construction from first principles of what could/should be the structural part of modeling languages. A key idea is that models are not bare lists of declarations of objects. Rather they should be seen as scripts to generate such lists.

4.1 Working Example

Throughout this chapter, we shall illustrate the various constructs of S2ML by means of the working example pictured Figure 4.1. This figure presents a simplified process and instrumentation diagram of a high integrity pressure protection system (HIPPS). HIPPS are control systems used in oil & gas industry to prevent incidents and accidents due to over-pressure (*ISO/TR 12489:2013 Petroleum, petrochemical and natural gas industries – Reliability modelling and calculation of safety systems* 2013).

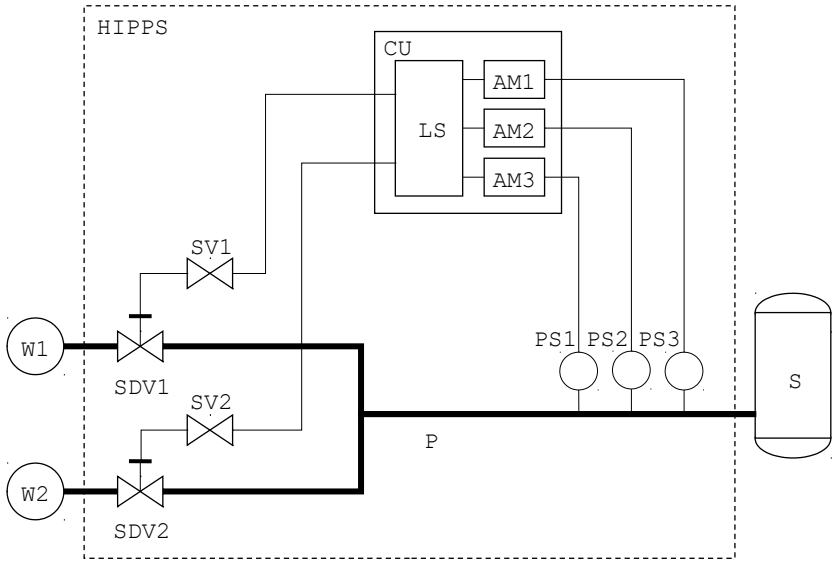


Figure 4.1: A high integrity pressure protection system

In our example, the HIPPS prevents the occurrence of an over-pressure in the separator `S`. This over-pressure is due to a too important flow rate of the mix of oil, gas and water coming from the wells `W1` and `W2` through the pipe `P`.

As most of control systems, this HIPPS is made of three main parts: some sensors, a controller and some actuators. The three pressure sensors `PS1`, `PS2` and `PS3` monitor the pressure into the pipe `P` and transmit respectively their measures to acquisition modules `AM1`, `AM2` and `AM3`. Sensors are triplicated to ensure a better protection against failures. The acquisition modules transmit a potential over-pressure detection to the logic solver `LS`. This logic solver works according to a 2-out-of-3 logic, i.e. it launches the over-pressure protection process when it receives an over-pressure message from at least 2 out of the 3 acquisition modules. This process consists in closing shutdown valves `SDV1` and `SDV2`. This valves are actuated respectively through solenoid valves `SV1` and `SV2`.

The control unit `CU` is made of the acquisition modules and the logic solver.

We shall consider, for the sake of simplicity, purely combinatorial models of the above system, i.e. models in which flows circulating in the network of components and internal states of components are described by means of data-flow equations. The corresponding language, S2ML+DFE (DFE stands for data-flow equations), is implemented in the tool `Janos`.

4.2 Blocks, Ports and Connections

4.2.1 Definition

The *block* is the basic S2ML modeling unit. In a physical decomposition, blocks are used to represent physical components. In a functional decomposition, they are used to represent functions. In a process description, they are used to represent activities. And so on.

In our working example, we shall use blocks to represent the sensors, the acquisition modules, the logic solver...

A model is thus made of blocks.

These blocks may interact. In our example, the sensor `PS1` interacts with the acquisition module `AM1`, which interacts with the logic solver `LS`. In S2ML, interactions are technically represented by means of ports and connections.

A *port* is something that has a name and that holds some atomic information. It is similar to variables of programming languages, but also of modeling languages such as Modelica or Lustre. Ports have usually a *type*, e.g. a port can be Boolean, integer, real or takes its value into a set of symbolic constants. The type of a port is called its *domain*. Note however that ports can also represent more abstract modeling elements such as events.

A *connection* describes a relation between ports. The connection, or relation "*interacts-with*", is one of the fundamental relations between elements of a model. Connections are used to describe behaviors. They are written using expressions involving constants, ports and various operators. Depending on the language, different types of ports, connections, constants and operators are available.

Languages of the S2ML+X family have a well-defined semantics but are neutral regarding their pragmatics. Ports and connections correspond in clear and unambiguous way to mathematical objects. However, it is up to the analyst to interpret these mathematical objects as features of the system under study. For instance, in a model representing the physical decomposition of a system, ports represent physical interfaces of components and connections physical connections between components. In a model representing the functional decomposition of the system, ports represent inputs and outputs of functions and connections dependencies between functions. In a model representing a business process, ports represent beginning and ends of activities and connections dependence relations between activities. And so on. Of course, a model can mixed these different aspects.

A block has thus a name and may contain ports, connections and other blocks (and, as we shall see, several other modeling elements). Blocks are thus *containers* for declarations of modeling elements.

4.2.2 Graphical Representation

Blocks are usually graphically represented with rectangles, delimited with light lines. Ports are represented by black filled squares and connections by bold lines.

Figure 4.2 (left) shows a normalized graphically representation for a valve.

It involves two ports: `close` and `flow`. They are of different natures: `close` is an action that can be applied to the valve while `flow` characterizes the flow of liquid going through the valve.

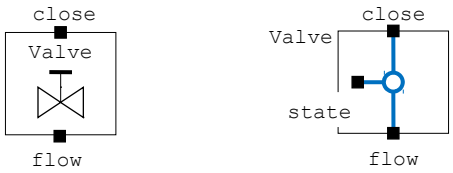


Figure 4.2: External and internal views of a block representing a valve

On this figure, all ports are represented on the border of blocks. This is by no means mandatory. Ports can be internal to a block. Figure 4.2 (right) shows for instance an internal view of the block with an internal port "`state`".

On this figure, ports are connected so to represent the interactions between the command `close`, the state `state` of the valve, and the flow `flow` of liquid going through the valve.

4.2.3 Textual Representation

Graphical representations, as those of Figures 4.1 and 4.2, are excellent communication means. However, they reach quickly their limits: as soon as the model ceases to be simple, they can only represent it partially. In other words, a model exists independently of its graphical representation. Moreover, as we shall see in the Section 4.3, there may be several graphical representations of the same concept, each having its own interest, depending on the context. That is the reason why languages of the S2ML+X family are primarily textual. The particular syntax of each these languages is not especially important. What is really important is to ensure a one to one correspondence between the concepts and the syntactic constructs of the language.

The S2ML+DFE code for the block representing the valve could be as shown Figure 4.3.

```
1  domain ValveState {WORKING, DEGRADED, FAILED}
2  domain FlowLevel {NULL, LOW, HIGH}
3
4  block ShutdownValve
5    ValveState state;
6    Boolean closed;
7    FlowLevel flow;
8    assertion
9      flow :=
10       if closed
11       then
12         if state==WORKING
13         then NULL
14         else
15           if state==DEGRADED
16           then LOW
17           else HIGH
18       else HIGH;
19 end
```

Figure 4.3: Code for a valve

This code declares first (line 1) the domains `ValveState`, which in this case is a set of three symbolic constants `WORKING`, `DEGRADED` and `FAILED`. Then, it declares the domain `FlowLevel`. Finally, it declares the block `Valve` (lines 4- 19).

As said above, a block is a container for declarations. These declarations are organized into two parts: first elements like ports and sub-blocks are declared, then the behavior of the component is described by means of connections.

In the code for the valve, three ports are declared: `state` whose domain is `ValveState`, `close` which is a Boolean, and `flow` whose domain is `FlowLevel`. The domain of a port is always declared in front of its name.

Connections are then declared. In S2ML+DFE, there is only one type of connections: data-flow equations. They are introduced by the keyword `assertion`.

In our example, there is only one equation, declared line 10, which updates the value of the port `flow` according to the values of the ports `state` and `close`.

The terms “block”, “port” and “connection” are borrowed to the SysML terminology (Friedenthal, A. Moore, and Steiner 2011), although they are used here with a slightly different meaning. In pure S2ML, ports and connections are interpreted by themselves, just as constants and function symbols in Herbrand’s interpretations of first order logic, see e.g. (Kleene 1967): whatever its syntax, a connection has no other meaning than “there is a relation between these ports”. In languages of the S2ML+X family, models have a richer semantics i.e. they are interpreted as elements of rich mathematical frameworks. For instance, a S2ML version of Modelica would interpret ports as continuous variables and connections as differential equations. A S2ML version of Petri nets would interpret ports as places and connections as transitions. And so on. Connections involve expressions.

Attributes

Attributes can be associated with ports so to add information. An *attribute* is a pair (name, expression), e.g.

```
1 ValveState state (init = WORKING);
2 Boolean close (reset = false);
3 FlowLevel flow (reset = NULL);
```

The above code associates the initial value `WORKING` to the port `state`, the default values `false` and `NULL` to the ports `close` and `flow`. Attributes are used not only to set initial and default values, but also to associated probability distributions with states, delays with events...

Figure 4.4 shows a normalized way of representing attributes graphically (and to given information about connections). Text of attributes are put in rectangles with folded bottom right corners. These rectangle are then attached to the element the attribute describes by means of a dotted line.

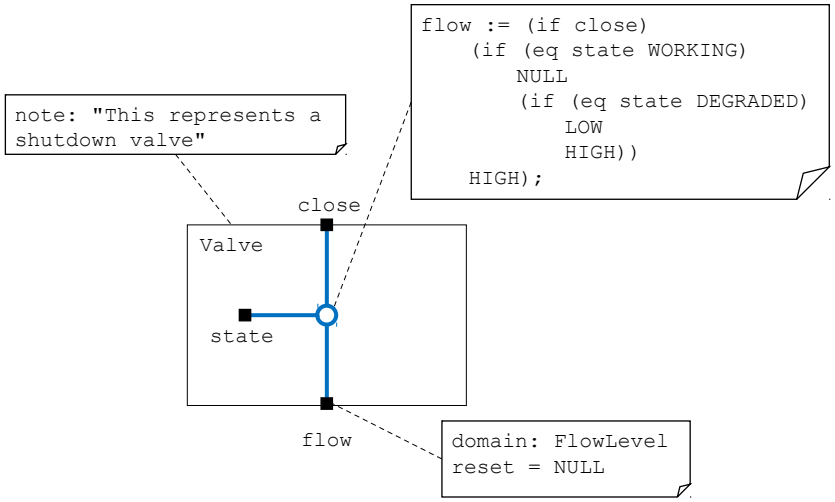


Figure 4.4: Graphical representation of attributes (and definition of connections)

4.2.4 Syntactic Elements and Notational Conventions

Identifiers and numbers

Ports, blocks and other constructs are uniquely identified with their name. Identifiers in the family of S2ML+X languages are those of most of the programming languages. Namely, an identifier

starts with a letter or an underscore character "_" followed by any number of letters, digits and underscore characters. Formally,

```
Identifier ::= [_a-zA-Z][_a-zA_Z0-9]*
```

The syntax of integers and floating point numbers is the same as in all computer tools. Examples of identifiers and numbers:

```
x       __A_long_identifier_of_39_characters__
12345   -1.23e-45
```

Expressions

In languages of the family S2ML+X, connections have a meaning and their writing involves expressions. Throughout this book, *expressions* are written in parenthesized infix form:

- Constants are written directly, e.g. `false`, `42`, `1.23e-5`, `OPEN`...
- References to ports are also written directly, e.g. `state`, `input`...
- Operations are parenthesized lists of items. The first item of the list is the operator to be applied. The subsequent items, if any, are the arguments of the operation. Arguments are themselves expressions. E.g. `(add x 1)`, `(or (not a) (not b))`... Symbols of operators are reserved words, i.e. that they cannot be used as identifiers for named elements such as ports and blocks.

The reason for using infix notation is twofold: first, it eases greatly the development of parsers; second, it is pedagogical in this sense that it forces to think about the difference between syntax and semantics. Note that, with both respect, we follow here the long tradition of functional programming languages started with LISP and continued for instance with Scheme (Abelson, G. J. Sussman, and J. Sussman 1996).

Appendix D describes the syntax and semantics of expressions.

Comments

Comments can be inserted everywhere in textual descriptions of models. There are two forms of comments:

- Comments spanning on one line. They are introduced by a two slashes "`//`". Everything between these two slashes and the end of the line is considered as a comment.
- Comments spanning on possibly several lines. They started with the two symbols "`/*`" and end with the two symbols "`*/`".

Examples of comments:

```
x := 42; // All roads lead to Roma.
/*
 This is a looooong comment.
 */
```

Notational conventions

Throughout this book, we use some notational conventions. These conventions are not required by the syntax of languages, but ease greatly the understanding of models.

- Identifiers of ports start with an underscore of with a lower case letter and are capitalized. E.g. `_state`, `failureDate`...
- Identifiers of containers start with an upper case letter and are capitalized. E.g. `P`, `RepairableComponent`...
- Symbolic constants are written using only upper case letters. Words are separated with underscores when necessary. E.g. `OPEN`, `FAILED_UNDETECTED`...

– Models are indented with two spaces or a tab character.

4.3 Composition

4.3.1 Definition

Figure 4.1 shows a hierarchical description. The controller `C` contains the logic solver `LS` and the three acquisition modules `AM1`, `AM2` and `AM3`. Similarly, the HIPPS contains the four valves `SV1`, `SV2`, `SDV1` and `SDV2`, the control unit `CU` and the three sensors `PS1`, `PS2` and `PS3`.

In object-oriented terminology, it is said that the HIPPS composes the valve `SV1` and that the valve `SV1` is part of the HIPPS.

The *composition*, or "*is-part-of*" relation, is the second fundamental relation between elements of a model.

In S2ML, to compose a block in another block, it suffices to declare the former inside the latter. The declaration of the block `CU` representing the control unit could be as shown Figure 4.5.

```
block CU // declaration of the controller
  block LS // declaration of the logic solver
    // body of the block LS
  end
  block AM1 // declaration of the acquisition module AM1
    // body of the block AM1
  end
  block AM2 // declaration of the acquisition module AM2
    // body of the block AM2
  end
  block AM3 // declaration of the acquisition module AM3
    // body of the block AM3
  end
  assertion
    LS.input1 := AM1.output;
    LS.input2 := AM2.output;
    LS.input3 := AM3.output;
end
```

Figure 4.5: Code for the controller (version 1)

In this code, the block `CU` composes the four blocks `LS`, `AM1`, `AM2` and `AM3`. It declares also three connections. The port `input1` of block `LS` is referred to as `LS.input1` as in most of the object-oriented languages.

In S2ML, it is possible to refer to a port (and more generally any modeling element) of a block nested at any hierarchical level, therefore "crossing the wall" of composing blocks. As connections can represent any type of relations, there is actually no reason to limit the reference mechanism as if ports were physical interfaces.

Note finally that each block is a name space: the blocks `AM1` and `AM2` can both declare a port `output`, but it is impossible to declare two elements with the same name within a block.

4.3.2 Graphical Representation

Composition can be graphically represented in several ways (think to various possibilities offered by Windows File Explorer® to represent graphically the hierarchies of files and folders).

Figure 4.6 shows a block-diagram representation for the whole HIPPS. This representation is very close to the process and instrumentation diagram of Figure 4.1. Note that on this figure,

components outside the system under study (the wells `W1` and `W2`, the pipe `P` and the separator `S`) are just represented as a "background image".

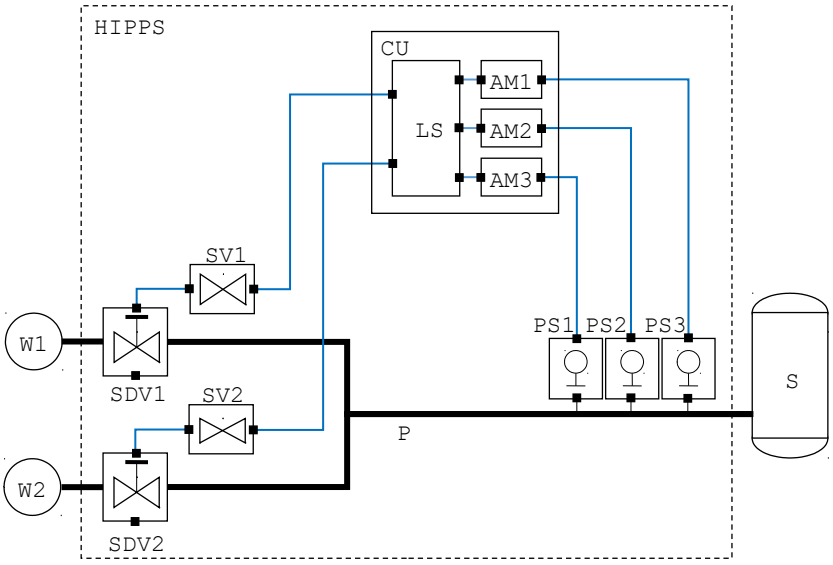


Figure 4.6: Block-diagram-like representation for the high integrity pressure protection system

Figure 4.7 shows a tree-like representation for the block `CU` that represents the control unit. One dimension tree (Figure 4.8 left) and tabular (spreadsheet) (Figure 4.8 right) representations are also possible. Which representation is the most convenient depends on the context.

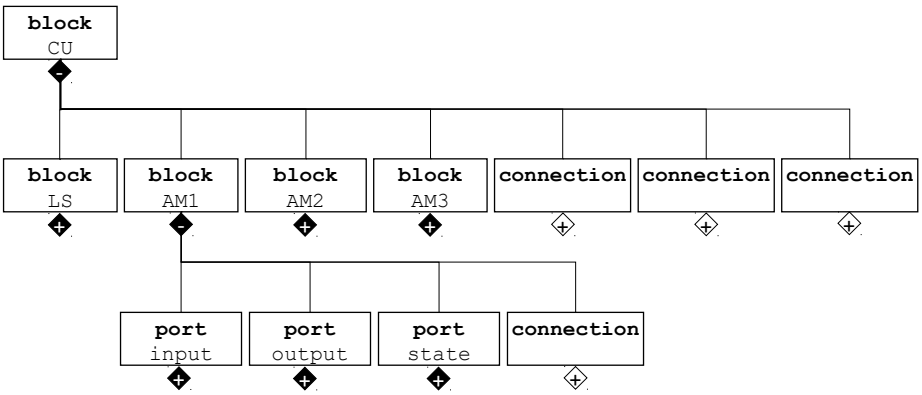
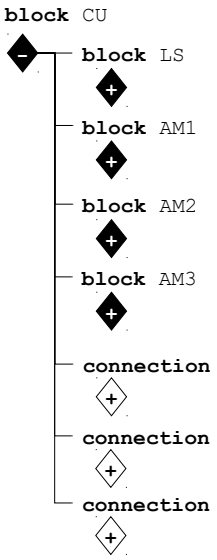


Figure 4.7: Tree-like representation of the control unit

The tree-like representation of Figure 4.7 is similar to SysML block definition diagrams (and more generally to UML class diagrams). We use black filled diamond to represent composition. We add to UML/SysML notation the possibility to fold and unfold parts of the tree, the "+" sign in the diamond meaning that the corresponding element can be expended, the "-" sign meaning that it can be folded.

The meaning of white filled diamonds will be explained Section 4.9.

Note that graphical representations such a tree-like representation can and probably should be to a large extent automatically generated from the model.



-	**block** CU		
		+	**block** LS
		+	**block** AM1
		+	**block** AM2
		+	**block** AM3
		+	**connection**
		+	**connection**
		+	**connection**

Figure 4.8: Alternative representations of the control unit

4.4 Prototypes and Clones

4.4.1 Definition

S2ML blocks are prototypes, in the sense of object- and prototype-oriented programming languages. A *prototype* is a modeling element, usually a container, with a unique occurrence.

There are cases however where the system under study embeds several identical components, or groups of components. In our working example, the two actuator lines `SV1`/`SDV1` and `SV2`/`SDV2` are probably identical, at least from a modeling standpoint. Moreover, it makes sense to gather the components `SV`*i* and `SDV`*i* under a block `Actuator`*i*. Once this grouping done, we have two identical blocks Actuator*i* in the model. Each time one of these blocks is modified, the other one must be modified as well. This is both error prone and confusing as the model does not reflect the fact that these two blocks are identical.

A solution to address this problem is create one block representing one of the actuator, e.g. `Actuator1`, then to clone this block. S2ML provides a directive to do so. This code for the HIPPS would be then as shown Figure 4.9.

```
1  block HIPPS
2    block Actuator1
3      block SV
4        // Body of the block SV
5      end
6      block SDV
7        // Body of the block SDV
8      end
9      assertion
10       SDV.close := SV.output;
11   end
12   clones Actuator1 as Actuator2;
13   // remainder of the model
14 end
```

Figure 4.9: Code for actuators

A *clone* is a copy of an existing block. The above code clones the block `Actuator1` to get the block `Actuator2`. Both blocks are local to the model. The cloning directive is extremely powerful in conjunction with aggregation and polymorphism mechanisms discussed later in this chapter.

4.4.2 Models as Scripts

Until this section, models were just (hierarchical) lists of declarations of ports, connections and blocks. With cloning, we introduced the notion of directive. A *directive* is a command that the tool in charge of loading models executes to create a piece of model. In other words, models that the analyst writes must be seen as *scripts* to generate models on which virtual experiments (calculations, simulations...) are performed.

The above code is thus transformed automatically in the one shown Figure 4.10.

```
block HIPPS
  block Actuator1
    block SV
      // Body of the block SV
    end
    block SDV
      // Body of the block SDV
    end
    assertion
      SDV.close := SV.output;
  end
  block Actuator2
    block SV
      // Body of the block SV
    end
    block SDV
      // Body of the block SDV
    end
    assertion
      SDV.close := SV.output;
  end
  // remainder of the model
end
```

Figure 4.10: Instantiated code for actuators

We shall come back on this transformation Section 4.10.

4.4.3 Graphical Representation

Figure 4.11 shows a normalized way of representing cloning graphically. The cloning block is linked to the cloned block with a dashed line ending with a white triangle.

4.5 Classes and Instances

4.5.1 Definition

There are many cases where not only a component, or group of components, has several occurrence in the model, but it is reused from model to model. In our example, it is probable that the description of sensors, acquisition modules, solenoid valves and shutdown valves enter into this category of reusable modeling components.

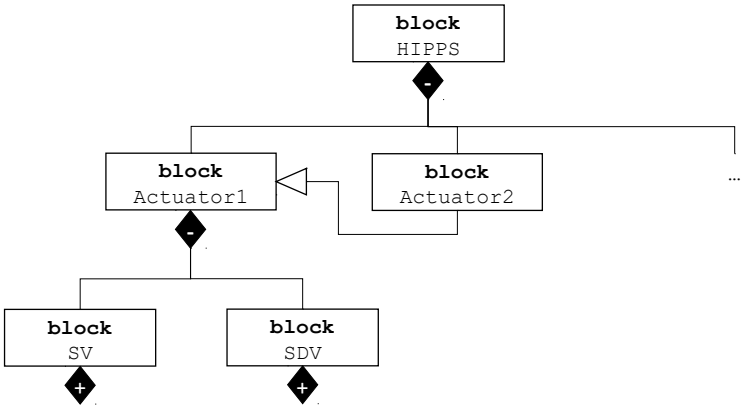


Figure 4.11: Graphical representation of cloning

This idea is captured by the notion of class. A *class* is a separate "on-the-shelf" block (with possibly a full hierarchy of blocks underneath) that can be reused in models via instantiation. *Instantiation* works like cloning, except that the cloned block does not belong, strictly speaking, to the model.

As an illustration consider the description of acquisition modules of the HIPPS. The idea is to create a class for these elements and to instantiate this class three times in the model. The code could be as shown Figure 4.12.

```
1  domain OnOff {ON, OFF}
2
3  class AcquisitionModule
4    OnOff state(init=ON);
5    Boolean input, output(reset=false);
6    assertion
7      output := if state==ON then input else false;
8  end
9
10 block CU
11   block LS
12     // body of the logic solver
13   end
14   AcquisitionModule AM1, AM2, AM3;
15   assertion
16     LS.input1 := AM1.output;
17     LS.input2 := AM2.output;
18     LS.input3 := AM3.output;
19 end
```

Figure 4.12: Code for the control unit

This code is equivalent to the one shown Figure 4.5 that embeds three identical blocks for acquisition modules `AM1`, `AM2` and `AM3`.

4.5.2 Graphical Representation

Figure 4.13 shows a normalized way of representing instantiation graphically. The instance is linked to the instantiated class with a dashed line ending with a white triangle. This representation is the same as the one of cloning as the two mechanisms are equivalent.

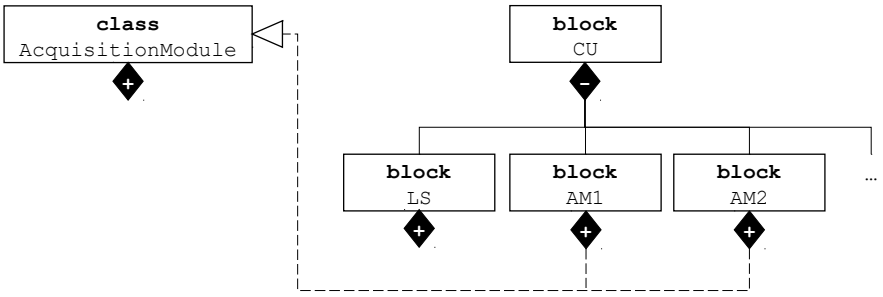


Figure 4.13: Graphical representation of instantiation

The superposition on a single figure of hierarchical relations and instantiation relations leads quickly to unreadable diagrams. For this reason, it is better to design separate diagrams, one for each type of information. It may be even better not to represent every graphically and to refer simply to the code...

4.6 Polymorphism

In programming languages and type theory, *polymorphism* is the capacity to handle entities of different types through the same interface. They are actually several types of polymorphisms, see e.g. (Abadi and Cardelli 1998). We shall not enter into too much details in the framework on this book as models make a much less extensive use of polymorphism than programs. Still, it is very convenient to define generic modeling components and to tune them for the specific needs of the model.

4.6.1 Parameters

The simplest way to achieve this goal is probably the use of parameters. A *parameter* is a port that has the particularity of keeping the same value in all executions of the model. This value is chosen once for all when the container that composes the parameter is instantiated.

Consider again the code for shutdown valves we introduced Section 4.2.3 and assume we want to make it a class. We may want to describe the stochastic delays between states `WORKING` and `DEGRADED` on the one hand, `DEGRADED` and `FAILED` on the other hand. Assume that these delays are both exponentially distributed, with respective rates λ_D and λ_F. It would be inconvenient to declare a new class each time we want to represent a shutdown valve with specific rates λ_D and λ_F. The idea is therefore to use parameters to represent these rates.

The code Figure 4.14 shows how it works.

When instantiating the class `ShutdownValve`, the value of parameters can be changed, e.g.

```
1  ShutdownValve SV1
2    parameter Real degradationRate = 1.0e-4;
3    parameter Real failureRate = 1.0e-2;
4    end
```

Note that the same construct applies if the component is cloned rather than instantiated, e.g.

```
1  clones SDV1 as SDV2
2    parameter Real failureRate = 2.0e-2;
3    end
```

```
class ShutdownValve
  ValveState state;
  Boolean closed;
  FlowLevel flow;
  Real timeToDegradation;
  Real timeToFailure;
  parameter Real degradationRate = 1.0e-3;
  parameter Real failureRate = 1.0e-3;
  assertion
    timeToDegradation := exponentialDeviate(degradationRate);
    timeToFailure := timeToDegradation + exponentialDeviate(failureRate);
    state :=
      if missionTime() < timeToDegradation
      then WORKING
      else
        if missionTime() < timeToFailure
        then DEGRADED
        else FAILED;
    flow :=
      if closed
      then
        if state==WORKING
        then NULL
        else
          if state==DEGRADED
          then LOW
          else HIGH
      else HIGH;
```

Figure 4.14: Code for a shutdown valve (extended)

Note also that, in virtue of the models-as-scripts principle, it is possible to change the values of parameters after instantiation or cloning, e.g.

```
  ShutdownValve SDV2;
  parameter Real SDV2.failureRate = 2.0e-2;
```

Note finally that parametric polymorphism is also called *genericity* in computer science.

4.6.2 Advanced Forms of Polymorphism

The models-as-scripts principle makes in theory possible to change any element after this named element has been declared. Languages studied in this book are however not so permissive: in most of them, only parameters can be modified.

It is however possible to compose new elements in a container. More advanced forms of polymorphism are available via the notion of inheritance (see Section 4.8) and aggregation (see Section 4.9).

4.7 Prototypes versus Classes

S2ML provides thus two types of containers: blocks, which are prototypes in the sense of prototype-oriented programming, and classes, see e.g. (Abadi and Cardelli 1998) and (Noble, Taivalsaari,

and I. Moore 1999) for discussions about these paradigms. Classes are well suited for "on-the-shelf", generic, stabilized knowledge. Prototypes are well suited for on-going work. The C-K theory of Hatchuel and Weill (Hatchuel and Weill 2009) formalizes this idea in the context of engineering design. Their dialectic applies to the model engineering as well: the model (made of block/prototypes) is the space C of concepts, i.e. a sandbox in which a new knowledge is maturing. Classes are the space K of stabilized knowledge.

The class/instance mechanism is without any doubt interesting to define "on-the-shelf", re-usable modeling components such as those for acquisition module in our working example. The richness of modeling languages such as Modelica or Matlab/Simulink stands for a great part in the wide choices of dedicated libraries.

This mechanism could actually be used systematically and describe all of the components, including the controller, of the HIPPS via classes. However, when authoring (graphically) the model, we may want to modify the code for the logic solver while editing the entire model. With a pure object-oriented language, this is not possible. Classes are flat: one cannot modify a class via its instance, even if this instance is unique. We may call that the "a box in a box in a box" issue.

Moreover, what can be re-used depends on the level of abstraction of the model. On the one hand, models designed with simulation languages such as Modelica or Matlab/Simulink, stand at physical component level. For these models, re-use means mainly re-use of modeling components. On the other hand, models standing at system level, typically designed with modeling languages such as SysML, BPMN (White and Miers 2008) or AltaRica (Prosvirnova, Batteux, et al. 2013) shows a rather different picture. At system level, each model is unique. Re-use can be obtained, but by cloning and adjusting models rather than by instantiating on-the-shelf components. In other words, there are prototypical systems/models, in the sense of Lakoff (Lakoff 1990). Nevertheless, no generic system/model from which other system/model could be derived just by setting some parameters. What can be re-used is thus modeling patterns (analogs of design patterns (Gamma et al. 1994) in programming) rather than modeling components.

4.8 Inheritance

4.8.1 Definition

As any object-oriented language, S2ML provides a construct to implement the "*is-a*" relationship, i.e. the *inheritance* mechanism: the "extends" directive. The inheritance is the third fundamental relation between elements of a model.

As an illustration, consider the sensors, the acquisition modules and the logic solver of our working example. They are all electronic components. They may share the property to be prone to failures whose probability obeys a Weibull distribution. The actual implementation for these components could therefore derive from the description of a generic electronic component. This derivation mechanism makes it possible to build step wisely the model. The code could be as shown Figure 4.15.

The class `Sensor` inherits, via the directive `extends`, of the class `ElectronicComponent`. This means that all members of the latter are members of the former, just as if its code was copied in place of the directive.

The class `Sensor` refine also the default values of the parameters `scaleParameter` and `shapeParameter` so to adjust them to the particular characteristics of sensors.

In the context of modeling languages, inheritance can be seen as a particular type of composition, where no prefix is added to the composed/inherited class or block. S2ML allows multiple inheritance, with all the usual warnings about it.

```
domain WF {WORKING, FAILED}

class ElectronicComponent
  WF state;
  Real timeToFailure;
  parameter Real scaleParameter = 1.0e-5;
  parameter Real shapeParameter = 3;
  assertion
    timeToFailure := WeibullDeviate(scaleParameter, shapeParameter);
    state := if (missionTime() < timeToFailure then WORKING else FAILED;
end

class Sensor
  extends ElectronicComponent;
  parameter Real scaleParameter = 1.0e-4;
  parameter Real shapeParameter = 10;
  Boolean input, output(reset = false);
  assertion
    output := input and state==CLOSED;
end
```

Figure 4.15: Code for valves with inheritance

4.8.2 Graphical Representation

Figure 4.16 shows a normalized way of representing inheritance graphically. The inheriting class is linked to the inherited class with a plain line ending with a white triangle.

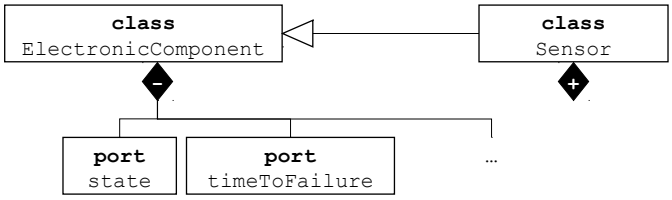


Figure 4.16: Graphical representation of inheritance

4.9 Paths and Aggregation

4.9.1 Paths

In the description of the HIPPS pictured Figure 4.6, The port `output1` of the logic solver `LS` should be linked to the port `input` of solenoid valve `SV1` via a connection. If this connection is declared at HIPPS level, it could be as follows.

```
SV1.input := C.LS.output1;
```

As already pointed out, the dot notation makes it possible to "cross the walls": The `SV1.input` denotes the port `input` of the block `SV1`, while `CU.LS.output1` denotes the port `output1` of the block `LS` of the block`CU`.

To put it differently, it is possible to refer the port `output1` of the block `LS` of the block`CU` within the parent block of `CU`, i.e. the block `HIPPS`. `CU.LS.output1` is the *path* making it possible to refer to the port `output1` within the block `HIPPS`.

It is sometimes useful to refer elements located not only in descendant blocks, but also parent or sibling block. This can be done in two ways: via absolute paths and via relative paths.

An *Absolute path* starts with the keyword `main`. `main` denotes the outermost block of the current hierarchy.

In the example, the outermost block is the block `HIPPS`. Therefore the above connection could be rewritten as follows.

```
main.SV1.input := main.CU.LS.output1;
```

The difference between the latter and the former forms is that the latter can be located anywhere in the model, for instance in the block `CU` or even in the block `PS1` (although it is highly recommended not to do so). The path `main.CU.LS.output1` denotes the port `output1` of the block `LS` of the block`CU` wherever in the model, i.e. within the block `HIPPS`.

A *relative path* uses the keyword `owner`, which denotes the parent block of the current block. For instance, within the block `CU`, `owner` denotes the block `HIPPS`. Within the block `CU`, the connection could thus be rewritten as follows.

```
owner.SV1.input := LS.output1;
```

Similarly, within the block `LS`, it could be rewritten as follows.

```
owner.owner.SV1.input := output1;
```

Absolute and relative paths are sometimes very useful, but they must be used with much care. The risk is indeed to transform your model into a spaghetti plate, with references going all over.

4.9.2 Aggregation: Definition

Composition describes a "is-part-of" relation. This relation assumes that a component cannot belong to two different super-components. There are cases however where the same component is used in several places.

As an illustration, consider again our working example. Figure 4.6 shows a physical breakdown of the system. This decomposition is fine, but other decompositions could be used as well. For instance, if we would adopt a bit more functional viewpoint, we could group solenoid valve `SV1` and shutdown valve `SDV1` together, as done Section 4.4. We could also group the pressure sensor `PS1` with the acquisition module `AM1` as these two components participate together to the over-pressure detection. `PS1` and `AM1` form together a *functional chain*. This functional chain could be completed by the power source in a more detailed model. The power source `PWS` would be then shared by, among other, the three functional chains `PWS-PS1-AM1`, `PWS-PS2-AM2` and `PWS-PS3-AM3`.

Functional chains are by no means a modeling gadget. When the model gets big, especially when it is authored concurrently by several systems engineers, it is edited via its functional chains (Voirin 2008). Functional chains represent logical units by which specialist contribute to the system design.

A solution to handle functional chains consists in designing several models, a "physical" model and one or more "functional" models. This solution has however several drawbacks, notably regarding the maintenance of the coherence of these different models.

Another solution consists in accepting that the same model contains different hierarchies. In our case, the model would be the one sketched in the previous section augmented with the description of the power source `PWS` and functional chains `SC`i, $i = 1,2,3$, made of the components `PWS`, `PS`i and `AM`i.

But going along this line raises a problem: if we use the composition, i.e. the "*is-part-of*" relation, components involved in functional chains would be declared and composed in several places.

The solution to this problem consists in introducing a new fundamental relation between elements of a model. This new relation is called *aggregation* in the object-oriented paradigm and can be described as a "*uses*" relation. The functional chain `SC1` uses the components `PWS`, `PS1` and `AM1` which are declared elsewhere, namely in the physical architecture.

The S2ML construct for aggregation is the "embeds... as" directive. For instance, the code for the two above functional chains could be as shown Figure 4.17.

```
block HIPPS
  // ...
  block CU
    AcquisitionModule AM1, AM2, AM3;
    // remainder of the description of the controller
  end
  // ...
  block PWS
    // description of the power source
  end
  // ...
  PressureSensor PS1, PS2, PS3;
  // ...
  block SC1 // sensor chain 1
    embeds main.PS1 as PS;
    embeds main.CU.AM1 as AM;
    embeds main.PWS as PWS;
    assertion
      AM.input := PS.output;
      AM.powerSupply := PWS.supply;
  end
  // ...
end
```

Figure 4.17: Code for the sensor functional chain

These functional chains declare only connections. They could declare blocks and ports as well.

In the code, the different aggregated components are referred to by means of absolute paths. It would be also possible to use relative paths. Aggregation and paths are actually tightly connected. Aggregation can be seen as a means of declaring aliases. Rather than to use absolute or relative paths for each reference to an element of the aggregated block, one aggregates the block as a whole.

The aggregation mechanism, seen as a "uses" relation, is extremely powerful. It makes it possible to embed into the same model different views, including functional and physical views. It can also be used to share universal objects (e.g. physical constants) in a clean way.

4.9.3 Aggregation: Graphical Representation

The aggregation is graphically represented as a white filled diamond, as shown Figure 4.18. Connections aggregate the ports they refer to. As for composition, S2ML adds a sign "+" or "-" to aggregation symbol so to fold and expand trees.

Note that connections use the ports they connect without declaring them. Connections aggregate thus their arguments.

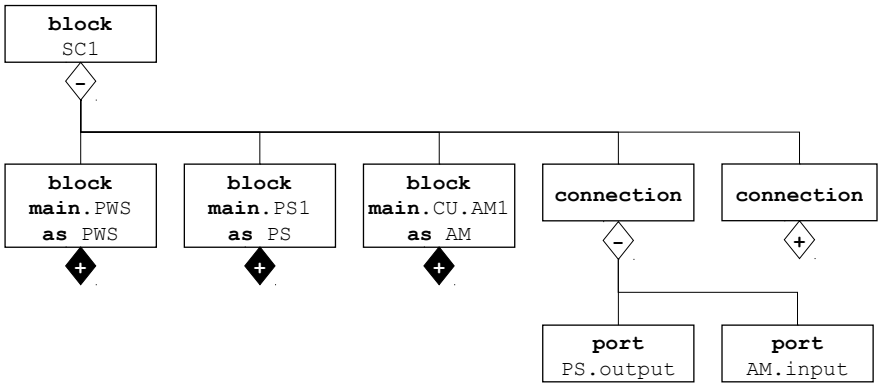


Figure 4.18: Graphical representation of aggregation

4.10 Instantiation and Flattening

By applying models-as-scripts principle, actual models are progressively built from source models. In order to assess a model, the tool performing the assessment (no matter what this assessment consists in), proceeds into two steps: first, it loads the model; then, it *instantiates* it, i.e. it transforms the source model into an actual one. This actual model is then made only of a hierarchy of blocks declaring only ports and connections.

As an illustration, consider again the block describing the controller of our example. The source code shown Figure 4.12 is transformed into the actual code shown Figure 4.5.

The instantiation process may be relatively complex when composition, inheritance, paths and aggregation are used together. Nevertheless, this process is performed automatically by assessment tools, so the analyst does not have to worry about it.

Note that the same term *instantiation* is used for the class/instance mechanism as for the process by which the source model is transformed into an actual model. These two mechanisms are actually very close one another, which justify to name them alike.

For some of the assessment tools, the instantiated form of the model is sufficient. For some others, it must be transformed again into a "flat" form, i.e. into a model made of single block declaring ports and connections.

In our controller example, the flat code would be as shown Figure 4.19.

```
1  block HIPPS
2    // ...
3    WF CU.AM1.state(init = WORKING);
4    Boolean CU.AM1.input, CU.AM1.output(reset = false);
5    // ...
6    assertion
7      //...
8      C.AM1.output := if C.AM1.state==WORKING then C.AM1.input else false;
9      // ...
10     C.LS.input1 := C.AM1.output;
11     C.LS.input2 := C.AM2.output;
12     C.LS.input3 := C.AM3.output;
13     // ...
14 end
```

Figure 4.19: Flattened code for the controller

This second operation is called *flattening* in the S2ML jargon. Together with instantiation, it transforms the source model into a pure behavioral model with no (explicit) structure. The key point here is that both transformations are purely syntactic and generic: they do not depend on the particular mathematical framework (the X) chosen to represent behaviors. This is the reason why the S2ML+X paradigm is so powerful.

4.11 Operators and Functions

Languages of the S2ML+X family provides a wide range of predefined operators, see Appendix D, and of structuring mechanisms, as illustrated in this chapter. Nevertheless, it is sometimes convenient to create new operators or new structuring mechanisms. This can achieve by means of user defined operators and functions.

4.11.1 Operators

A *user defined operator* has any positive number of formal parameters and a body, which is an expression. Calls to user defined operators work as calls to predefined operators, except that the body of their definition is substituted for the call and arguments of the call are substituted for formal parameters of the operator at instantiation.

Declarations of user defined operators are introduced by the keyword `operator`, followed by the name of the operator and the list of its formal parameter. This list is ended by the character "=". Then comes the expression. The declaration is terminated with a semicolon "`;`".

Calls to user defined operators have the same syntax as those to predefined operators.

■ **Example 4.1 – Operator AddFlowLevels.** Assume we want have different ways of combining flow of liquid in pipes. Then, it may worth declaring operators to represent the different possible combinations. This works as follows.

```
operator AddFlowLevels(f1, f2) =
  if f1==NULL
  then
    if f2==NULL then NULL else LOW
  else
    if f1==LOW
    then
      if f2==HIGH then HIGH else LOW
      if f2==NULL then LOW else HIGH
```

In the above operator declaration, the two formal arguments are `f1`, `f2` and the body of the operator is the expression `if ...`.

Hence declared, the operator can be called wherever one wants in the model. E.g.

```
flow := AddFlowLevels(SDV1.flow, SDV2.flow);
```

At instantiation, the following code is substituted for the above one.

```
flow :=
  if SDV1.flow==NULL
  then
    if SDV2.flow==NULL then NULL else LOW
  else
    if SDV1.flow==LOW
    then
      if SDV2.flow==HIGH then HIGH else LOW
      if SDV2.flow==NULL then LOW else HIGH;
```

■

4.11.2 Functions

User defined functions work along the same principles as operators, except that their body is made of declarations. They can be called everywhere the declarations contained in their body can.

Declarations of user defined functions are introduced by the keyword `function`, followed by the name of the operator and the list of its formal parameter. This list is ended by the character "=". Then comes the body which consists in any number of declarations. The declaration is terminated with the keyword `end`.

■ **Example 4.2 – Function.** Assume we want to the same distribution for probability distributions of a number of variables. Then, it may worth, for the sake of the clarity of the model, to create a dedicated function. This could works as follows.

```
function SetWeibullDistribution(state, scale, shape) =
  assertion
    state := WeibullDeviate(scale, shape) <= missionTime();
end

block Main
  ...
  parameter Real alpha = 1.0e-3;
  parameter Real beta = 3;
  SetWeibullDistribution(BE1.failed, alpha, beta);
  SetWeibullDistribution(BE2.failed, alpha, beta);
  ...
end
```

Calls to `SetWeibullDistribution` are replaced at instantiation by the following equations.

```
block Main
  ...
  parameter Real alpha = 1.0e-3;
  parameter Real beta = 3;
  assertion
    BE1.failed := WeibullDeviate(scale, shape) <= missionTime();
    BE2.failed := WeibullDeviate(scale, shape) <= missionTime();
  ...
end
```

Note that, in the above example, the body of the function contains only one equation. Nothing prevents to have more. ■

4.12 Further Readings

Here follows a list of books for those who want to go further.

Books about the object-oriented paradigm in programming.

- The book by Abadi and Cardelli (Abadi and Cardelli 1998) is a reference monograph regarding theoretical concepts behind object-oriented programming languages.
- The book by Meyer (Meyer 1988) is a reference monograph regarding object-oriented programming.
- The book edited by Noble and his colleagues (Noble, Taivalsaari, and I. Moore 1999) is one of the rare books dealing with prototype-oriented programming. It is a collection of articles but covers rather well the subject.

Books about graphical modeling formalisms in systems engineering.

- The book by "the three amigos" Rumbaugh, Jacobson and Booch (Rumbaugh, Jacobson, and Booch 2005) is the reference on UML (Unified Modeling Language). Although UML is targeted for software development, it had and still has a strong influence on modeling formalisms used in Systems Engineering.
- The book by Friedenthal, Moore and Steiner is the reference monograph on SysML (System Modeling Language), which is, at the time we write these lines, the most popular modeling formalism in Systems Engineering.

4.13 Exercises and Problems

Problem 4.1 – Overflow Protection System. The objective of this problem is to formalize functional and physical architectures of the overflow protection system presented in Chapter 2 (case study 1 page 20).

Question 1. Design a pure S2ML model to encode the functional architecture of the overflow protection system.

Question 2. Design a pure S2ML model to encode the physical architecture of the overflow protection system (apply a zonal decomposition).

Question 3. Merge the two models you designed in the previous section into a unique model encompassing both the functional and the physical architecture.
Hint: Use aggregation to embed the functional architecture onto the physical architecture.

■

Problem 4.2 – High Integrity Pressure Protection System. The objective of this problem is to formalize functional and physical architectures of the high integrity pressure protection system presented in Chapter 2 (case study 2 page 37).

Question 1. Design a pure S2ML model to encode the functional architecture of the high integrity pressure protection system.

Question 2. Design a pure S2ML model to encode the physical architecture of the high integrity pressure protection system (apply a zonal decomposition).

Question 3. Merge the two models you designed in the previous section into a unique model encompassing both the functional and the physical architecture.
Hint: Use aggregation to embed the functional architecture onto the physical architecture.

■

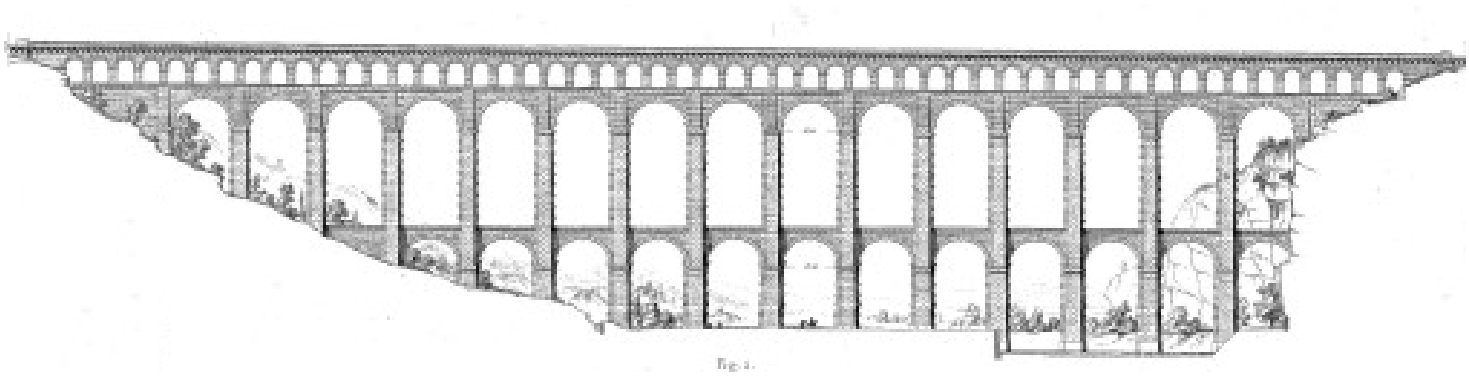
Problem 4.3 – BPMN for Pizze. Figure 3.7 page 65) shows a BPMN describing the business process of ordering and eating a pizza at the restaurant. The objective of this problem is to formalize the constructs of BPMN shows on the figure in the S2ML+X paradigm.

Question 1. Propose a syntax for basic BPMN objects: tasks, state, gateways and connections.

Question 2. Using S2ML constructs to represent swim-lanes and pools, encode the BPMN of the figure into your S2ML+X language.

■

II Modeling Environments



5. Reliability Engineering

Key Concepts

- Risk and hazard
- Epistemic versus aleatory uncertainties
- Risk matrices
- Safety barriers
- Probabilistic risk and safety assessment
- Reliability, availability, failure rate, mean time to failure
- Failure, fault, failure modes
- Cumulative distribution function, failure model
- Exponential, Weibull, Dirac, empirical distributions
- Failure mode and effects analysis
- Fault trees, event trees, reliability block diagrams
- Binary decision trees
- Markov chains
- Stochastic Petri nets

The operation of any technical system induces always risks for the system itself, its operators and its environment. These risks are impossible to avoid completely. The only question is therefore to assess whether they are socially acceptable. Aside safety-critical risks, the operation of all technical systems are subject to many uncertainties like changes in the environment of the system, failures of some of its parts, errors made by its operators and so on. Reliability engineering is the engineering discipline in charge of determining what can go wrong in a system, what is the likelihood that something goes wrong, what are the potential consequences if something goes wrong and possibly how to remedy the potential problems.

This chapter recalls fundamental concepts and tools of system reliability theory and presents briefly some of the modeling frameworks that are widely used in industry: failure mode and effects analysis, fault trees, reliability block diagrams, event trees, Markov chains and to a lesser extent stochastic Petri nets.

5.1 Risk Analysis

5.1.1 Risk, Hazard and Uncertainties

A *risk* is an event or 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 decrease it sufficiently to make it socially acceptable.

One should not confuse risks with hazards. A *hazard* is an event or series of events external to the system that impacts the system. For instance, a earthquake is a hazard that may impact a building. On the contrary, a risk, e.g. the collapse of a building, is an event or a series of events internal to the system. A hazard, here the earthquake, has also a certain likelihood and may impact more or less severely the system. Moreover, risks result often from hazards: a well designed and constructed building does not collapse without cause but it may collapse because of an earthquake. The difference between risks and hazards is that the designer and the operator of a system can hopefully do something to mitigate risks but in general nothing against hazards. This said, it is important to assess both dimensions of hazards and to integrate them into the assessment of risks. However, we shall not discuss this topic in this book.

Analyzing hazards and risks is in general subject to many uncertainties. There are actually two different types of uncertainties (Apostolakis 1990):

– *Epistemic uncertainties* that result from the lack of knowledge on the considered phenomena.
– *Aleatory uncertainties* that result from phenomena we know but that occur randomly.

Mechanical failures are typical examples of aleatory uncertainties. We know, possibly quite precisely, why they can occur and what are their consequences, but not when they will occur nor even if they will occur. Nevertheless, we can make statistics on real systems and computerized physical models to assess how their likelihood evolves through the time. Probabilistic risk and safety analyses deal essentially with such aleatory uncertainties, using stochastic models.

5.1.2 Risk Matrices

Safety regulations and standards classify the risk into *risk matrices* such has the one given in Table 5.1 which is extracted from IEC 61508 standard (*International IEC Standard IEC61508 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems (E/E/PE, or E/E/PES)* 2010).

Table 5.1: IEC 61508 risk matrix

	Consequences			
Likelihood	Negligible	Marginal	Critical	Catastrophic
Frequent	Undesirable	Unacceptable	Unacceptable	Unacceptable
Probable	Tolerable	Undesirable	Unacceptable	Unacceptable
Occasional	Tolerable	Tolerable	Undesirable	Unacceptable
Remote	Acceptable	Tolerable	Tolerable	Undesirable
Improbable	Acceptable	Acceptable	Tolerable	Tolerable
Incredible	Acceptable	Acceptable	Acceptable	Acceptable

The standard clarifies each term of the matrix. The categories of likelihood and severity of occurrences are characterized as on Tables 5.2 and 5.3.

Eventually, the standard classifies the risk as follows.

– Unacceptable: unacceptable in any circumstance.

Table 5.2: IEC 61508 characterization of the likelihood of a risk

Category	*Definition*	*Range (occurrences per year)*
Frequent	Many times in system lifetime	$> 10^{-3}$
Probable	Several times in system lifetime	10^{-3} to 10^{-4}
Occasional	Once in system lifetime	10^{-4} to 10^{-5}
Remote	Unlikely in system lifetime	10^{-5} to 10^{-6}
Improbable	Very unlikely to occur	10^{-6} to 10^{-7}
Incredible	Cannot believe that it could occur	$< 10^{-7}$

Table 5.3: IEC 61508 characterization of the severity of a risk

Category	*Definition*
Catastrophic	Multiple loss of life
Critical	Loss of a single life
Marginal	Major injuries to one or more persons
Negligible	Minor injuries at worst

– Undesirable: tolerable only if risk reduction is impracticable or if the costs are grossly disproportionate to the improvement gained.
– Tolerable: tolerable if the cost of risk reduction would exceed the improvement.
– Acceptable: acceptable as it stands, though it may need to be monitored.

Assessing the risk requires thus studying it along its two dimensions: likelihood and severity of 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) that followed the Three Mile Island nuclear accident. 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 for Development of Civil Aircraft and Systems* 2010)
– (*ISO26262 Functional Safety - Road Vehicle* 2012)

At this point, we can make two important remarks.

First, we shall speak in the sequel about risks in a rather broad sense, i.e. not only about risks with catastrophic consequences. We shall thus use the expression “probabilistic risk analysis” or equivalently “probabilistic risk assessment” to encompass processes as diverse as safety and reliability assessments, optimizations of maintenance policies, assessments of the expected production level of a plant over a given period and so on. In a word, we shall speak here about assessments of operational performance of technical systems subject to random events such as mechanical failures, operator errors...

Second, although everybody agrees that the likelihood of risks must be studied, many experts

and non-experts are 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 chapter, 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 the results to variations of the parameters of the model. In other words, the calculated probabilities do not need to exist outside the model, provided that they reflect some relevant aspects of the system.

5.1.3 Safety Systems and Barriers

Defense in Depth

To mitigate risks, one can reduce their likelihood or their consequences or both. With that respect, *defense in depth* is a key concept that is implicitly used in most of the industries and explicitly by the nuclear industry. This approach recognizes that imperfections, failures, and unanticipated events will occur and must be accommodated in the design, operation, and regulation of nuclear facilities. It consists in organizing the mitigation of risks into successive and as much as possible independent layers, including robust physical barriers, redundant and diverse safety systems, strong physical security, and emergency response readiness.

Figure 5.1 shows a classical illustration of the defense in depth concept.

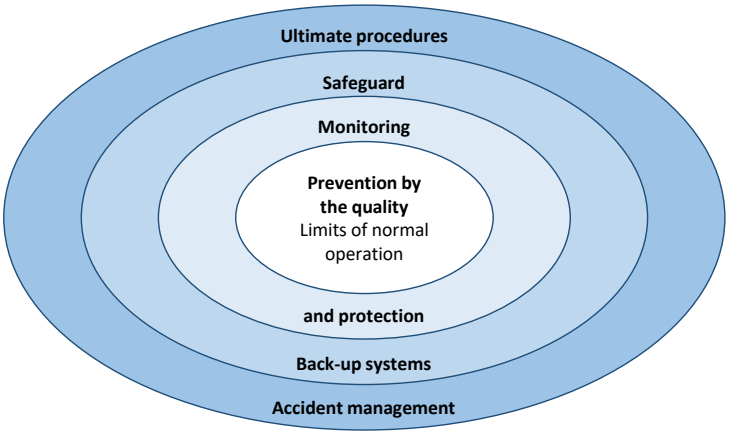


Figure 5.1: Defense in depth

The first layer of protection consists in preventing accidents by controlling the quality of the operations. This includes regulations, best practices, training of operators and more generally all means that can possibly ensure that the system is operated within the prescribed limits. The second layer of protection consists mostly in automated control systems. These systems are in charge of regulating operations so to prevent the system operation to drift outside its prescribed envelope. The third layer consists in *safety barriers* strictly speaking, i.e. of automated systems that put the system in a safe state after it drifted out its normal operation. The fourth layer consists in all possible means to mitigate the consequences of an accident, once it occurred.

Safety Instrumented Systems

In industrial processes presenting high risk, the third layer of protection is of primary importance. Safety systems, independent from systems in charge of regulating the operations, are put in place.

They are often termed *safety instrumented systems*, SIS for short.

The standard IEC 61508 (*International IEC Standard IEC61508 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems (E/E/PE, or E/E/PES)* 2010) is actually dedicated to safety instrumented system in process industries and is the mother standard for industry specific implementations, such as such as IEC 62061 (machinery systems), IEC 62425 (for railway signaling systems), IEC 61513 (for nuclear systems), and ISO 26262 (for road vehicles).

Safety instrumented systems played such an important role that safety studies are very often almost entirely dedicated to the assessment of their reliability.

5.1.4 Illustration

To illustrate the definitions and concepts proposed in this section, let us consider again the overflow protection system studied in Chapter 2 (case study 1, presented page 1). For the sake of readability, we shall recall basic about this case study.

Figure 5.2 shows the process and instrumentation diagram of the overflow protection system.

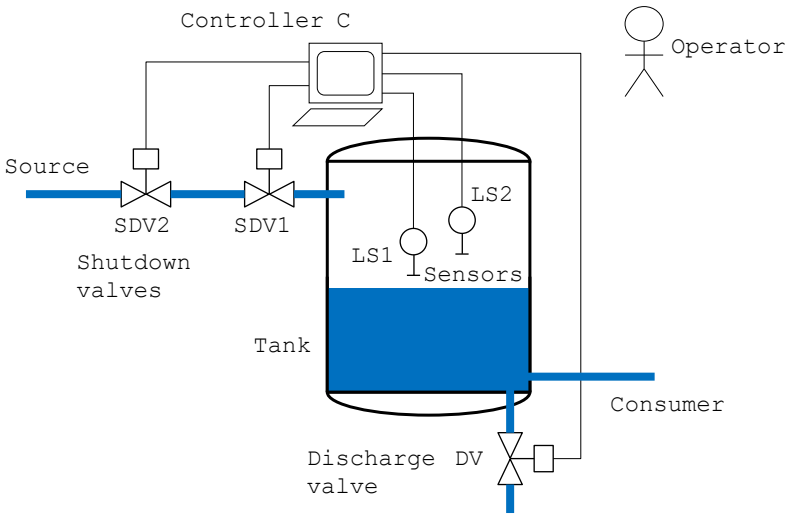


Figure 5.2: An overflow protection system

The equipment under study is a tank in which some chemical liquid is stored. The liquid comes by gravity into the tank from a source. It is regularly pumped out of the tank by a consumer. It may be the case however that too much liquid comes in or too few is pumped out. If such a situation would last, the liquid would overflow, which would causes serious problems. To prevent that, an overflow protection system is installed. It consists of two protection lines:

- The first line consists itself of the level sensor `LS1`, the calculator `C` and the shutdown valve `SDV1`. When the liquid reaches the level detected by `LS1`, this information is sent to `C`, which sends the order to `SDV1` to close.
- If this first line does not work, no matter the reason, a second protection line is activated. It consists on the level sensor `LS2`, the calculator `C`, the shutdown valve `SDV2` and the discharge valve `DV`. If the liquid reaches the level detected by `LS2`, this information is sent to `C`, which sends the order to `SDV2` to close and to `DV` to open.

As such, our overflow protection system is actually a safety instrumented system. It prevents the equipment under control, here the tank, from the risk of an overflow. This risk may follow several hazards, for instance the fact that the consumer stops pumping the liquid in the tank or that too much liquid is dumped to the tank. The severity of the consequences of an overflow in the tank depends indeed on what happens in case of such an event (explosion, breach, leakage...) and on the dangerousness of chemical products involved. However, we can assume that it is at

least critical, therefore needs imperatively to be mitigated, hence the installation of an overflow protection system.

The first two layers of the defense in depth concept are here realized by the operation policy of the facility. The overflow protection system belongs to the third layer. The latter consists actually of three successive barriers, as illustrated in Figure 5.3. The first two barriers are active. They consists of the two lines of the overflow protection system. The third one is passive. It is provided by the resistance of the tank to an overpressure.

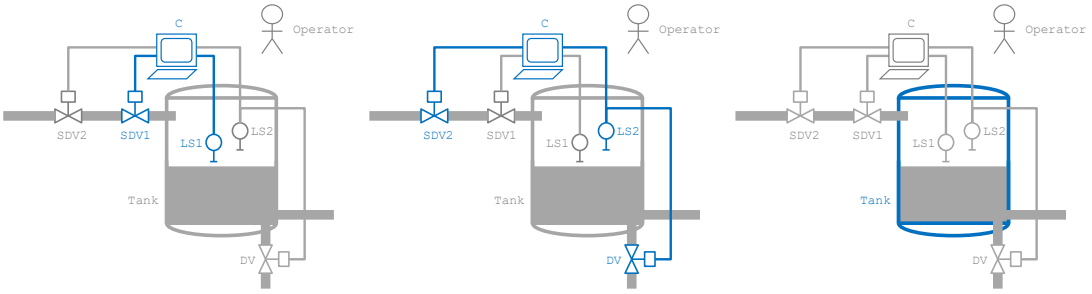


Figure 5.3: Safety barriers of the facility protected by the overflow protection system

Note that the two first barriers are not independent as they share the computer C. We shall come back on this point.

5.2 System Reliability Theory

5.2.1 Probabilistic Risk and Safety Assessment

As a first approximation, we can consider that all probabilistic risk assessment models describe incident or accident scenarios, i.e. combinations of degradations or failures of components of the system (including operator errors) that induce a degradation or a failure of the system as a whole. We shall make more precise in this section the terms “degradation”, “failure” and “combination” and correct the intuitive description we just gave.

But before doing so, we have to make three important methodological remarks.

First, probabilistic risk assessment models focus on how the system under study may degrade or fail, not on how it works. Ideally, it would be nice to obtain risk assessment models just by “decorating” models describing how the system works. In practice, this is impossible because of the combinatorial explosion of the number of combinations of possible degradations and failures. The risk analyst has only one solution to cope this combinatorial explosion: abstraction. Abstractions performed for probabilistic risk analyses are just specific to probabilistic risk analyses. No way around.

Second, it may seem at the first glance that the whole probabilistic risk assessment approach assumes that some of the components of the system must be degraded or failed for the system to be degraded or failed. This assumption (and in the same move probabilistic risk assessment) is strongly criticized by some authors, e.g. (Leveson 2012), who claim that many industrial accidents occurred, one should say emerged, in absence of degradation or failure of components. They are right about accidents, but wrong about probabilistic risk assessment: the notions of degradation and failure can be broadened so to encompass many practical problematic situations, e.g. lack of appropriate control action. Moreover, probabilistic risk analyses do not aim at assessing all possible issues a system may encounter when operated, but at assessing correctly those resulting of combinations of degradations and failures of components.

Third, scenarios of degradation or failure may involve not only degradations and failures of components, but also preventive and corrective maintenance actions, reconfigurations, changes of

life-cycle phase and many other events impacting the operation of the system. For the sake of the convenience, we shall speak here only of degradations and failures but this whole set of events should be understood.

We can now go back to our matter.

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 "discrepencies 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 fault states. These fault states 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.

Probabilistic risk assessment models are thus made of two main parts:

- The description of combinations of degradations or failures of components resulting in a degradation or failure of the system as whole.
- The description, for each component, of the probability that this component is degraded or failed.

There are indeed many ways combinations of degradations and failures of components can be described, from a purely combinatorial approach, like in fault trees, to a highly dynamic approach where the number of components and their interactions can change throughout the mission of the system, like in agent programming.

The description of probabilities of degradation or failure of components can also take different forms, from bare point estimates to complex distribution functions depending on the time. These probabilities are obtained by different means: physical models, statistical analyses on fleet on similar components operated in similar conditions, data mining, expert judgment... Methods to collect reliability data are industry dependent. As of today and despite decades of experience, they are still the Achilles' heel of probabilistic risk analyses, due mainly to the scarcity and the bad quality of data.

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).

They are often described by means of parametric distributions called failure models in the reliability engineering literature. Section 5.2.3 reviews the most important ones.

5.2.2 Fundamental Concepts

The reader has noticed that we made, at the end of the previous section, a conceptual shift which is by no means 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 C for a precise definition). Note that this is by no means 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 5.2.1 – 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,

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

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

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

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

$$\lim_{t \to 0} R_S(t) = 1 \tag{5.1}$$

$$\lim_{t \to \infty} R_S(t) = 0 \tag{5.2}$$

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

Definition 5.2.2 – 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.

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

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

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

Here follows some additional definitions.

Definition 5.2.3 – Failure density. The *failure density* $f_S(t)$ is the probability density function

of the law of T. It is the derivative of $F_S(f)$:

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

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 5.2.4 – Mean time to failure. The *mean time to failure* $MTTF_S(t)$ describes the expected time to the first failure, i.e.

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

Definition 5.2.5 – Failure rate. The *failure rate*, sometimes called *hazard function* $h_S(t)$ is defined as follows.

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

The following equalities hold from the definitions.

$$h_S(t) = \frac{f_S(t)}{R_S(t)} = \frac{f_S(t)}{1 - F_S(t)} \tag{5.3}$$

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.

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 a very important role in 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.

5.2.3 Failure Models

Parametric failure models are widely used in reliability engineering. They provide a simple and efficient way to define time-dependent probabilities. From a modeling language perspective, they can be seen as macro-expressions that are used to simplify the writing of probability distributions. Among parametric models, the exponential, the Weibull and the Dirac distributions are the most common.

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:

$$Q(t) \stackrel{def}{=} p \tag{5.4}$$

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.

Exponential distribution

The *exponential distribution* represents typically the life-span of a component without memory, aging nor wearing (Markovian hypothesis). The probability that the component is working at least $t+d$ hours knowing that it worked already t hours is the same as the probability that it works d hours after its entry into service. In other words, the fact that the component worked correctly for t hours does not change its expected life duration after this delay.

The exponential distribution is defined by means of a single parameter, the transition rate, usually denoted by λ. This transition rate is the inverse of the mean life expectation.

Table 5.4 gives the characteristics of the exponential distribution.

Table 5.4: Characteristics of the exponential distribution

Probability density function	$f(x) = \lambda e^{-\lambda x}$
Cumulative probability function	$F(x) = 1 - e^{-\lambda x}$
Mean	λ^{-1}
Median	$\lambda^{-1} \ln 2$
Variance	λ^{-2}

Figure 5.4 shows the shape the cumulative distribution function of the exponential distribution.

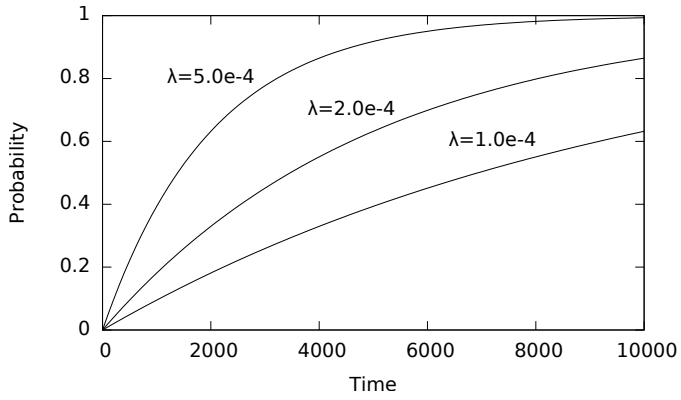


Figure 5.4: Cumulative distribution function of the exponential distribution

Weibull distribution

The exponential distribution assumes a constant failure rate over the time. This is not always realistic because of aging effects: at the beginning of its life the component has a decreasing failure rate, corresponding to debug (or infant mortality), then for a long while, its failure rate remains constant, then the wear-out period starts where the failure rate increases. This is the so-called bathtub curve.

This phenomenon is (piece wisely) captured by the *Weibull distribution* which takes two parameters: the shape parameter, usually denoted α, and the scale parameter, usually denoted β.

Table 5.5 gives the characteristics of the Weibull distribution. In this table, the Γ function is defined as follows.

$$\Gamma(z) \stackrel{def}{=} \int_0^\infty x^{z-1} e^{-x}\, dx \tag{5.5}$$

Table 5.5: Characteristics of the Weibull distribution

Probability density function	$f(x) = \frac{\beta}{\alpha}\left(\frac{x}{\alpha}\right)^{\beta-1} e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$
Cumulative probability function	$F(x) = 1 - e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$
Mean	$\alpha\Gamma\left(1+\frac{1}{\beta}\right)$
Median	$\alpha(\ln 2)^{\frac{1}{\beta}}$
Variance	$\alpha^2\left(\Gamma\left(1+\frac{2}{\beta}\right) - \left(\Gamma\left(1+\frac{1}{\beta}\right)\right)^2\right)$

Figure 5.4 shows the shape the cumulative distribution function of the Weibull distribution.

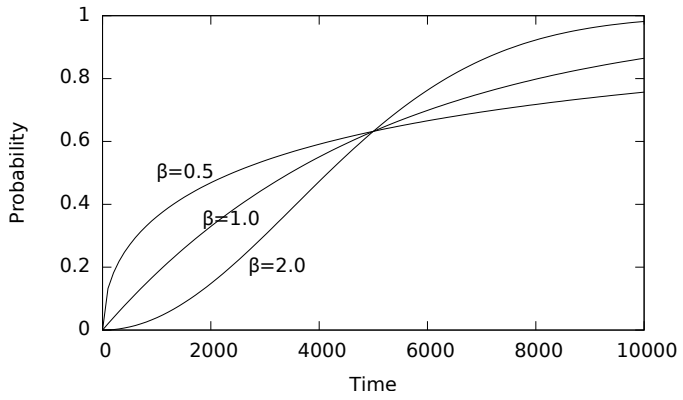


Figure 5.5: Cumulative distribution function of the Weibull distribution

Dirac distribution

The Dirac distribution is used to represent deterministic delays. It is thus characterized by a single parameter, the delay d:

Definition 5.2.6 – Dirac distribution. A *Dirac distribution* of *delay*, d, where d is a non-negative real number, is a cumulative distribution function verifying:

$$F(t;d) \stackrel{def}{=} \begin{cases} 0 & \text{if } t < d \\ 1 & \text{otherwise} \end{cases}$$

$$F^{-1}(z;d) \stackrel{def}{=} d$$

The Dirac distribution with a null delay is typically used to represent reconfigurations of the systems under study, e.g. if a main part fails, the spare part is attempted to start as a back-up.

The Dirac distribution with non-null delays is typically used to represent periodic actions, like preventive and corrective maintenance actions. In this case, they may be used both to represent the duration of the action and the delay between two actions.

Empirical distributions

Empirical distributions are given by a list of points $(x_1, y_1), \ldots, (x_n, y_n)$, $n \geq 2$, such that $x_1 < \ldots < x_n$ and, if one considers cumulative distribution functions, $y_1 \leq \ldots \leq y_n$. They are typically obtained via series of observations.

In between points, the value of the function is obtained by interpolation. There are basically two ways to do this interpolation:

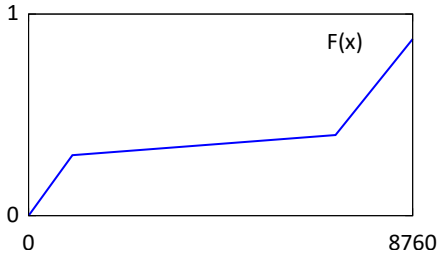


Figure 5.6: Cumulative distribution function of the piecewise uniform distribution

– To consider the distribution as a *stepwise distribution*, i.e.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases} \tag{5.6}$$

– To consider the distribution as a *piecewise uniform distribution*, i.e.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i + (y_{i+1} - y_i)\frac{x - x_i}{x_{i+1} - x_i} & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases} \tag{5.7}$$

Figure 5.6 shows a piecewise uniform distribution defined by 4 points: $(0,0)$, $(1000,0.3)$, $(7000,0.4)$ and $(8760,0.876)$.

Until recently, empirical distributions were mostly used when it was hard to fit them with any parametric distribution. For instance, the *Kaplan–Meier estimator* (Kaplan and Meier 1958), also known as the product limit estimator, is a non-parametric statistic used to estimate the survival function from lifetime data. In reliability engineering, Kaplan–Meier estimators may be used to measure the time-to-failure of machine parts. In medical research, they are often used to measure the fraction of patients living for a certain amount of time after treatment. This situation may generalize with easy internet communications: there is no more need to store data into a very compact form (the name of the parametric function and its parameters). Therefore, there is less and less reasons to spend time and energy in fitting empirical observations with parametric distributions, not to speak about the arbitrariness of the choice of the parametric distribution.

5.3 Failure Mode and Effect Analysis

5.3.1 Presentation

Failure mode and effects analysis (FMEA), sometimes written with "modes" in plural, is the process of reviewing as many components as possible to identify potential failure modes in a system and their causes and effects.

For each component, the failure modes and their resulting effects on the rest of the system are recorded in a specific FMEA worksheet. Table 5.6 shows a FMEA worksheet taken from (Rausand 2014).

Note that there are numerous variations of such worksheets.

Note also that the FMEA is sometimes extended into a *failure mode, effects, and criticality analysis* (FMECA) to indicate that criticality analysis is performed too.

5.3.2 Limitations

While a FMEA identifies important failures in a system, its results may not be comprehensive and the approach has thus strong limitations:

– FMEA may only identify major failure modes, having limited a validity when used in isolation.

Table 5.6: A FMEA worksheet

Description of unit			Description of failure			Effect of failure		Risk		Risk reducing measure	Responsible	Comment
Ref. no	Function	Operational mode	Failure mode	Failure cause	Detection of failure	On the subsystem	On the system function	Failure are (per hour)	Severity			
4.1	Close gas flow	Normal operation	Valve fails to close on demand	Spring broken	Proof test	Shutdown function failed	Unprotected	2.00×10^{-7}	HH	Periodic control of spring		
				Hydrates in valve Too high friction in actuator						Periodic operation of valve		
			Leakage through the valve	Erosion in valve seat	Proof test	Shutdown function degraded	Partly unprotected	8.00×10^{-7}	M	Improved startup control to prevent sand production		
				Sand between value seat and gate Sand between value seat and gate								
4.2	Open gas flow	Closed	Valve cannot be opened on command	Leakage in hydraulic system Too high friction in actuator	Immediately detected	Cannot start production	System cannot produce					

- FMEA cannot take into account multiple failures, which is highly problematic when studying the reliability of complex technical systems.
- The ranking of the severity of failures of components is problematic.
- The FMEA worksheet is hard to produce, hard to understand and read, as well as hard to maintain.

Eventually, failure mode and effect analyses turn often to be a bureaucratic activity, with little added value. They are performed because it is more or less required by regulations, but quickly forgotten right after and pile up into dusty bookshelves (or hard disks in their "modern" versions).

Nevertheless, the identification of failure modes and their probability is a mandatory step to design safety models.

5.4 Fault Trees, Reliability Block Diagrams and Event Trees

Fault trees, reliability block diagrams and event trees are the most popular methods to perform probabilistic risk assessment. They consists essentially at representing the failures of a system by means of a Boolean function of the failures of its (basic) components. They differ on the way the Boolean formula describing concretely this function is built.

Chapter 6 is dedicated to a thorough treatment of these methods. Consequently, we shall keep the presentation informal here.

5.4.1 Preliminaries

Before presenting fault trees, reliability block diagrams and event trees, we need to recall basics about Boolean algebras. The next chapter provides a more formal treatment. Appendix B.3 recalls also some fundamental properties.

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 ")":

- Boolean variables and constants are Boolean formulas.
- If f, $f_1 \dots f_n$ are Boolean formulas, then so are $f_1 \vee \dots \vee f_n$, $f_1 \wedge \dots \wedge f_n$, 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 $(A \wedge B) \vee ((\neg A) \wedge C)$.

Truth Assignments

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

Let $\mathscr{V}$, be a set of Boolean variables. A *truth assignment* of $\mathscr{V}$ is a mapping from $\mathscr{V}$ to $\{0,1\}$, i.e. to Boolean constants.

If $\mathscr{V}$ contains n variables, there are 2^n possible different such truth assignments.

Truth assignments are first lifted-up into mappings from Boolean formulas to Boolean constants using the well-known truth tables given in Table 5.7.

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

Table 5.7: 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

satisfy and falsify f are given in the following truth 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 $\mathscr{V}$ of Boolean variables can thus be interpreted as a mapping from truth assignments over $\mathscr{V}$ into $\{0,1\}$, i.e. eventually as a Boolean function over $\mathscr{V}$.

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

Probabilities

In probabilistic safety analyses, Boolean formulas are used to represent failures of the system under study. Namely, each variable represents the failure of a basic component and the formula represents the combinations of failures of basic components leading to a failure of the system as a whole.

From an operational point of view, the question is thus to design algorithms that, given a Boolean formula f and probabilities associated with the variables of f, calculate the probability of the formula.

One could think to use rules that students learn at high school, i.e. that given two events E and F, the following equalities hold.

$$p(E \cup F) = p(E) + p(F) - p(E \cap F) \tag{5.8}$$
$$p(E \cap F) = p(E) \times p(F) \tag{5.9}$$
$$p(\overline{E}) = 1 - p(E) \tag{5.10}$$

Unfortunately, this does not work, at least not directly. First, equality 5.10 holds only if E and F are independent (the occurrence of E gives no information on the occurrence of F, and vice-versa. Second, equality 5.9 requires to calculate, in a way or another, $E \cap F$.

Consequently, probabilistic indicators cannot be calculated directly from formulas. Formulas have to be transformed into other, preferably equivalent formulas, from which the direct computation is possible, possibly with some approximation. This will be one of the main topics of Chapter 6.

5.4.2 Fault Trees

Fault trees are usually introduced (and authored) using their graphical representation. Figure 5.7 shows a fault tree for our overflow protection system.

Intuitively, a *fault tree* is thus a Boolean circuit with several inputs (boxes labeled with `LS1failed`, `CFailed`...on the figure) and one output (the box labeled with `SISLost` on

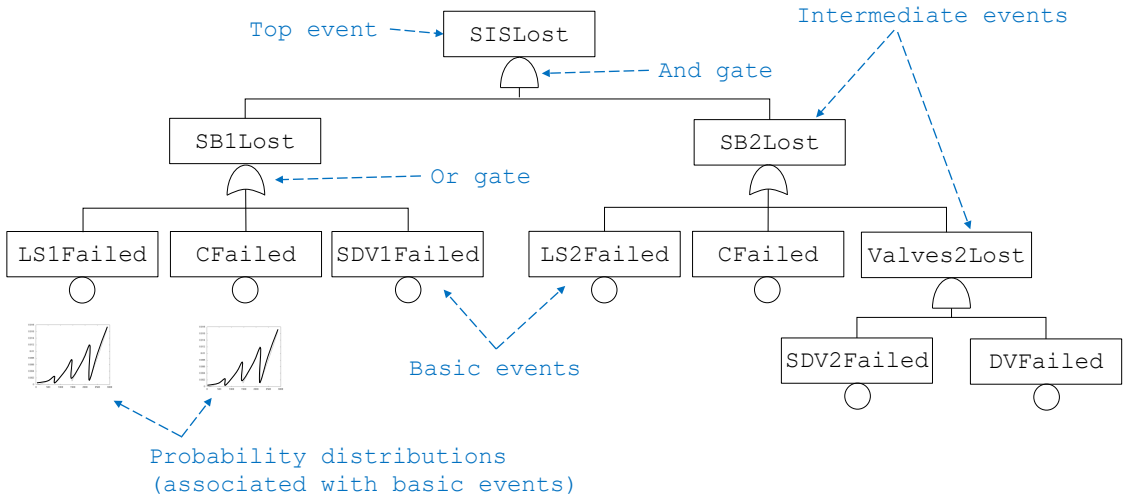


Figure 5.7: A fault tree for the overflow protection system

the figure). hence the name "tree", the inputs figuring the leaves and the output the root. It represents the combination of fault of components of the system under study that induce a fault of the system as a whole.

The (graphical representation of a) fault tree is a made of several elements:

- A set of *basic events*. Basic events represent fault of components of the system under study. They are the leaves of the tree and graphically represented by rectangles with a small circle under. The fault tree pictured in Figure 5.7 involves six basic events: `LS1failed`, `LS2failed`, `CFailed`, `SDV1Failed`, `SDV2Failed` and `DVFailed`.
- A set of *intermediate events*. Intermediate events represent Boolean combination of basic events (and other intermediate events). They are intermediate nodes of the tree and graphically represented by rectangles. The fault tree pictured in Figure 5.7 involves four intermediate events: `SISLost`, `SB1Lost`, `SBD2Lost` and `Valves2Lost`.
- A set of *logical gates*, one per intermediate event. They are represented using the usual symbol of digital circuits. In the fault tree pictured in Figure 5.7:
  - The intermediate event `SISLost` is associated/defined by an "and" gate whose inputs are the events `SB1Lost` and `SBD2Lost` as the capacity provided by the safety instrumented system is lost of both safety barriers are lost.
  - The intermediate event `SB1Lost` is associated/defined by an "or" gate whose inputs are the basic events `LS1Failed`, `CFailed` and `SDV1Failed` as the first safety barrier is lost if either the sensor `LS1` is failed, or the computer `C` is failed or the shutdown valve `SDV1` is failed.
  - Similarly, the intermediate event `SB2Lost` is associated/defined by an "or" gate whose inputs are the events `LS2Failed`, `CFailed` and `Valves2Lost`.
  - Finally, the intermediate event `Valves2Lost` is associated/defined by an "and" gate whose inputs are the events `SDV2Failed` and `DVFailed`.
- A unique intermediate event, called the *top event*, here `SISLost`, which is the only event that is not input of any gate. The top event represents the fault of the system as a whole.

It is important to understand that top-, intermediate- and basic events are events in the sense of probability theory, i.e. out-comes of an experience. They are not events in the common English sense, i.e. something that happens. They describe states of the system, not transitions between these states.

Basic events are associated with probability distributions representing the probability of failure

Table 5.8: Minimal cutsets of the fault tree pictured in Figure 5.7

1	CFailed	
2	LS1Failed	LS2Failed
3	LS1Failed	SDV2Failed
4	LS1Failed	DVFailed
5	SDV1Failed	LS2Failed
6	SDV1Failed	SDV2Failed
7	SDV1Failed	DVFailed

of the component. These probability distributions, or failure models, are obtained by experience feedback, expert judgment or calculated from lower level models.

Once a fault tree designed, essentially two types of analyses can be made:

– *Qualitative analyses* that consist in extracting scenarios of failure, i.e. combination of basic events that imply the top event. For instance, in our example, the capacity provided by the safety instrumented system is lost if both sensors LS1 and LS2 are failed. These are called *cutsets* of the fault tree. These cutsets are actually *minimal cutsets* as one cannot remove any of their member while keeping the property to be a cutset. Table 5.8 gives the complete list of minimal cutsets.

– *Quantitative analyses* that consist in calculating probabilistic indicators, such as the probability of the top event.

Note that, fault trees may not be tree at all, but rather directed acyclic graphs. For instance, in the fault tree pictured in Figure 5.7 the leaf CFailed is shared by two branches (those rooted by intermediate events SB1Lost and SB2Lost).

5.4.3 Reliability Block Diagrams

A *reliability block diagram* is a directed acyclic graph. The nodes, called *basic blocks*, of the graph represent basic components of the system. Each basic block has a Boolean input flow and a Boolean output flow. Intuitively, reliability block diagrams are interpreted by means of the following rules.

– The output flow is true if and only if the input flow is true and the component is working.

– The system as the whole works if there is an operating path from its source node to its target node.

Figure 5.8 shows the graphical representation of a reliability block diagram.

This reliability block diagram consists of the following elements.

– The *source node* S. In reliability block diagrams, the source node is *a priori* unique. In case it is not, one can always create a unique source node and connect it to the previous source nodes.

– The *target node* T. The target node is always unique.

– The *basic blocks* LS1, LS2, C, SDV1, SDV2 and DV.

– Arcs connecting the nodes and blocks.

Candidate operating S-T paths, called *path sets*, are: LS1-C-SDV1, LS2-C-SDV2, and LS2-C-DV.

Another, equivalent way, to interpret a reliability block diagram is to take a dual vision:

– The output of a basic block is false if and only if either its input is false or the component represented by the block is failed.

– The system as whole is failed if and only if all inputs of its target node are false (assuming all the output of its source node is true).

As expected, the dual of the notion of (minimal) path set is the notion of *(minimal) cutset*, which are (minimal) set of basic blocks whose combined failures disconnect the source and target

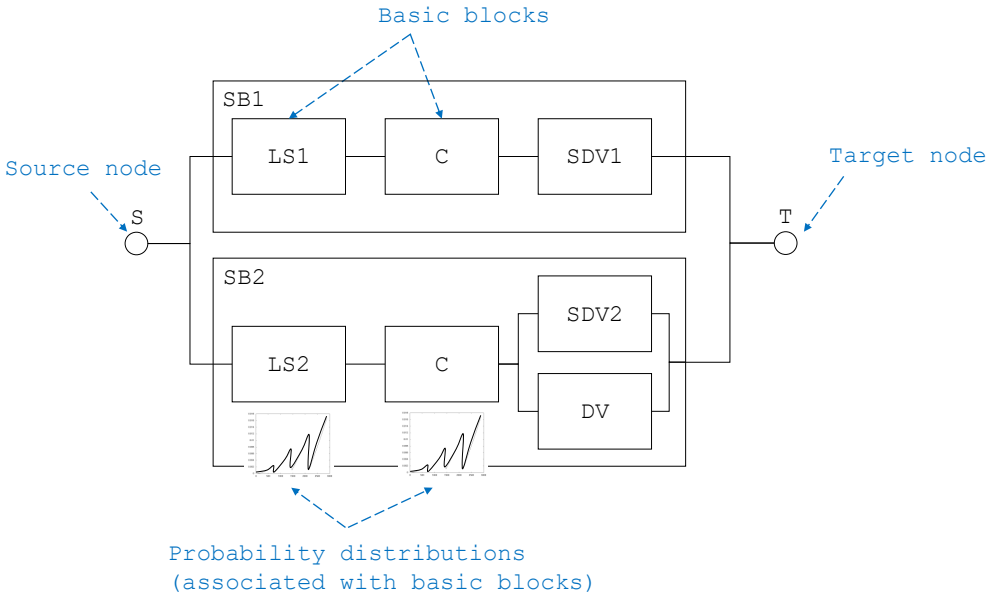


Figure 5.8: A reliability block diagram for the overflow protection system

nodes. In our example, a minimal cutset is again the combined faults of `LS1` and `LS2`. The minimal cutsets obtained from the reliability block diagram pictured 5.8 are the same as those given in Table 5.8.

As for fault trees, both *qualitative analyses* (extraction of minimal cutsets) and *quantitative analyses* (calculation of probabilistic indicators) are performed on reliability block diagrams.

5.4.4 Event Trees

Event trees are also Boolean models, most of the time used in combination with fault trees. Intuitively, an *event tree* represents the reaction the system to an *initiating event*, usually a hazard. Successive safety barriers are applied to prevent an accident. Each barrier may succeed or fail, leading to a branching in the tree. Each branch of the tree represents thus a possible accident sequence and may end with different *consequences* ranging from coming back to normal operations to a severe accident.

Figure 5.9 shows the graphical representation of an event tree for the overflow protection system.

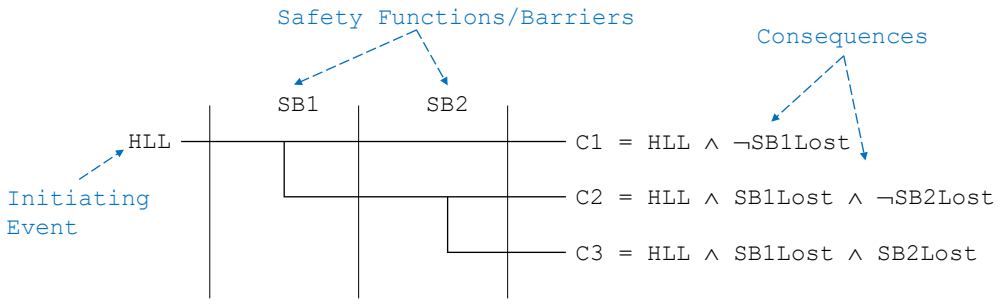


Figure 5.9: An event tree for the overflow protection system

This event tree reads from left to right and consists of the following elements.

– The *initiating event* `HLL` (high level of liquid), which is the starting point of all sequences.

- Two *functional events* SB1 and SB2. Functional events describe the success or the failure of safety barrier or the availability or the unavailability of some subsystem, usually some support system. In an event tree, they are organized in columns, as shown on the figure.
- Nodes of the tree. Each node corresponds to a decision regarding a functional event. It has two out-edges. The upper out-edge represents the success of the corresponding safety barrier, or the availability of the corresponding subsystem. The lower out-edge represents its failure.
- Finally, consequences C1 to C3, i.e. where the different scenarios of evolution of the system ends.

In the event tree pictured in Figure 5.9, the consequence C1 corresponds to the occurrence of the initiating event HLL, then a success of the first safety barrier (functional event SB1). As, in event trees like in fault trees, we are interested in what does not work, this corresponds to the formula:

$$\texttt{C1} = \texttt{HLL} \wedge \neg\texttt{SB1Lost}$$

Usually the failures of safety barriers, i.e. functional events, are themselves described by means of fault trees. Hence, we have a fault tree rooted by the event SB1Lost representing the causes of loss of the safety barrier represented by the functional event SB1. This fault tree is the same as the one given in Figure 5.7. Similarly for the loss of the second safety barrier represented by functional event SB2 is described by the fault tree rooted by the intermediate event SB2Lost.

Consider now the branch of the tree leading to the consequence C2. This scenario consists of the occurrence of the initiating event HLL, then the loss of the first safety barrier and the success of the second one. This corresponds to the formula:

$$\texttt{C2} = \texttt{HLL} \wedge \texttt{SB1Lost} \wedge \neg\texttt{SB2Lost}$$

The same principle applies to all sequences.

Note that in practice, negations of functional events are often dropped off, for reasons that will be clarified in Chapter 6. With such simplification, the formulas associated with consequences would be as follows.

$$\texttt{C1} = \texttt{HLL}$$
$$\texttt{C2} = \texttt{HLL} \wedge \texttt{SB2Lost}$$
$$\texttt{C3} = \texttt{HLL} \wedge \texttt{SB1Lost} \wedge \texttt{SB2Lost}$$

Event trees are assessed by building a fault tree, usually called the *master fault tree* that create the disjunction of the scenario leading to an accident. In our example, only the consequence C3 could represent such scenarios. The top event TOP of the master fault tree would then be defined as follows.

$$\texttt{TOP} = \texttt{C3}$$

Once the master fault tree built from the source event tree, both *qualitative analyses* (extraction of minimal cutsets) and *quantitative analyses* (calculation of probabilistic indicators) are performed.

Event trees are mostly used in nuclear industry. Over the year, software have been developed that make the event tree framework significantly more complex than what we presented above. One of the reason is that different treatments are applied on each branch of the tree, like setting the values of some variables or reliability parameters. The problem is indeed that these treatments have not been normalized and are specific to each tool. Consequently, models and their semantics tend to be heavily tool dependent.

5.4.5 Assessment Algorithms

Assessment algorithms for Boolean reliability models will be discussed in Chapter 6. These highly efficient algorithms cannot however be implemented within few lines of codes (nor even few thousands). It is worth to know how to assess the most important quantitative indicator, namely the probability of the top event of a fault tree, given the probabilities of its basic events.

As explained in Section 5.4.1, this cannot be done directly from the fault tree. An intermediate representation has to be built. Truth tables are such an intermediate representation. Unfortunately, they suffer from the exponential blow up of their size (2^n rows and n columns for a fault tree with n basic events). Binary decision trees provide an interesting alternative.

A *binary decision tree* associated with the top event of a fault tree is a directed binary tree such that:

- Each node of the decision tree is labeled with a basic event of the fault tree and has two out-edges: A then out-edge corresponding to the case where the basic event takes the value true, i.e. it occurred at the considered mission time, and an else out-edge corresponding to the case where the basic event takes the value false, i.e. it did not occur at the considered mission time.
- The leaves of the tree are labeled with either the constant 1, which corresponds to the case where the top event of the fault tree has occurred at the given mission time, or the constant 0, which corresponds to the case where it did not occur.
- A basic event occurs at most once in a branch from the root node to a leaf.

Figure 5.10 shows a possible binary decision tree for the fault tree pictured in Figure 5.7 (and also the reliability block diagram pictured in Figure 5.8).

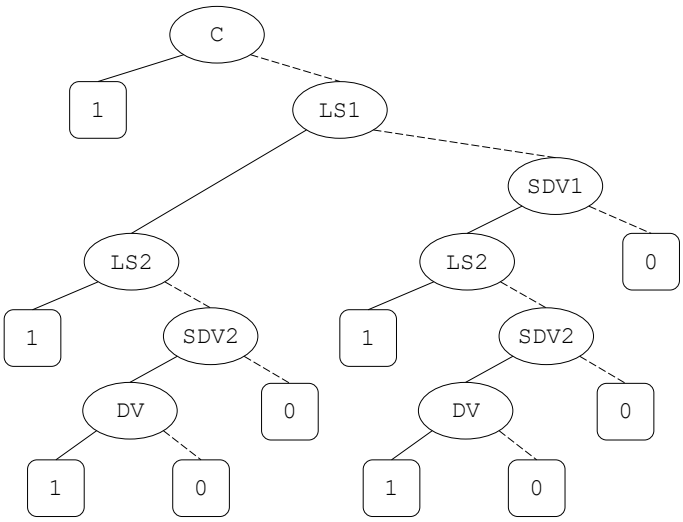


Figure 5.10: Binary decision tree for the fault tree pictured in Figure 5.7

Then out-edge are represented with plain lines, while else out-edges are represented with dashed lines. The value of the top event for a given variable assignment can be read by going down the corresponding branch (trees of computer scientists are growing downward, for cultural reasons). Binary decision trees can be seen as a compact encoding of truth tables.

Now, the whole point is that it is easy to calculate the probability of the top event of fault tree from a binary decision tree encoding that fault tree, thanks to the following property.

Property 5.1 – Shannon decomposition. Let f be a Boolean function and let E a variable of

f. Then, the following equality holds.

$$p(f) \quad = \quad p(E) \times p(f|E=1) + (1-p(E)) \times p(f|E=0)$$

To calculate the probability p of a node of a binary decision tree labeled with the variable E, it suffices thus:

- To calculate the probability p_1 of its then child;
- To calculate the probability p_0 of its else child;
- To calculate p as $p = p(E) \times p_1 + (1-p(E)) \times p_0$.

Indeed the probability of a leaf 1 is 1 and the probability of a leaf 0 is 0.

5.5 Markov Chains

Introduced by the Russian mathematician Andrei Markov in the early 20th century, Markov chains are nowadays pervasive in science and engineering. Everywhere a process with some uncertainty is under study, Markov chains are in the background. Markov chains are thus a fundamental modeling tool, if not directly, at least as a underlying conceptual framework. This applies especially to reliability engineering.

There are actually two "species" of Markov chains: discrete time Markov chains and continuous time ones. We shall review them in turn, although in the context of reliability engineering, one is mostly interested in continuous time Markov chains.

5.5.1 Discrete-Time Markov Chains

In probability theory, a *discrete-time Markov chain (DTMC)* is a sequence of random variables X_0, $X_1 \ldots$, known as a stochastic process, in which the value of the next variable depends only on the value of the current variable, and not any variables in the past:

$$Pr(X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) \quad = \quad Pr(X_{n+1} = x \mid X_n = x_n) \tag{5.11}$$

In practice, discrete-time Markov chains are often described as directed graphs, i.e. as pairs $\langle S, T \rangle$, where

- S is a finite set of *states*.
- T is a finite set of *transitions*, i.e. of triples $\langle s, p, t \rangle$, where s and t are states called respectively the source and the target states of the transition and p is a probability, i.e. a real number between 0 and 1. For the sake of clarity, transitions $\langle s, p, t \rangle$ are denoted $s \xrightarrow{p} t$.

In addition, the transitions of T must verify that for all states, the probabilities of out-transitions sum to 1, i.e.

$$\forall s \in T, \sum_{s \xrightarrow{p} t \in T} p = 1$$

As an illustration, consider the evolution of the level of liquid in the tank of the system pictured in Figure 5.2. During regular operations of the system, the level of liquid at time $t + dt$ depends on the level at time t and the amount of liquid flowing in and out the tank during the time interval dt. According to the description of the system, the latter quantities may vary randomly.

Figure 5.11 shows a discrete time Markov chain that could possibly represent the evolution of the level of liquid in the tank. The level has been here discretized in 7 levels, not necessarily evenly distributed, ranging from 0 (dry tank) to 6 (overflow), passing by intermediate levels, notably the level 4 where the sensor `LS1` raises an alarm and the level 3 where the sensor `LS2` raises an alarm.

Probabilities to go from the state i to the state j have been calculated for a reasonable time interval dt, for instance 1 minute.

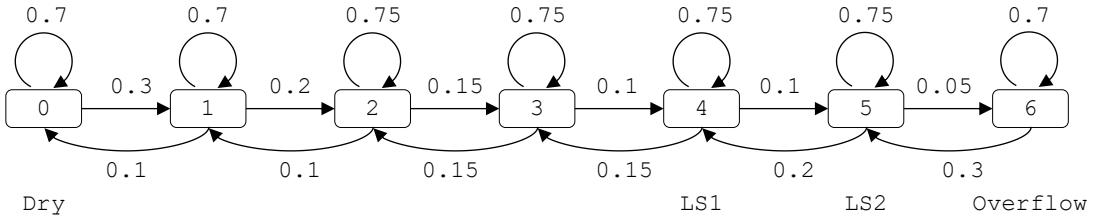


Figure 5.11: Discrete-time Markov chain representing the evolution of the level of liquid in the tank

Let us denote by $p_n(l)$, the probability of being at level l at step n. Assume the level is initially 2. then $p_0(2) = 1$ while $p_0(l) = 0$ for all $l \neq 2$. Now at step 1, we have:

$$\begin{aligned} p_1(1) &= 0.3 \times p_0(0) + 0.7 \times p_0(1) + 0.1 \times p_0(2) = 0.0 + 0.0 + 0.1 = 0.1 \\ p_1(2) &= 0.2 \times p_0(1) + 0.75 \times p_0(2) + 0.15 \times p_0(3) = 0.0 + 0.75 + 0.0 = 0.75 \\ p_1(3) &= 0.15 \times p_0(2) + 0.75 \times p_0(3) + 0.15 \times p_0(4) = 0.0 + 0.0 + 0.15 = 0.15 \end{aligned}$$

For all other states $p_1(l) = 0$. We can iterate this process at will.

It is relatively intuitive to see that, on the long run, the probability will distribute over all states. This is actually the case for all chains without source and sink states (states without respectively in- and out-transition).

Some chains are ergodic, i.e. that alternate eventually among a finite number of distributions. Some other show steady states, i.e. the probabilities of states tend toward a unique distribution.

Although seldom used in reliability engineering, discrete-time Markov chains are an essential mathematical tool.

5.5.2 Continuous-Time Markov Chains

A *continuous-Time Markov chain (CTMC)* is a stochastic process, i.e. a random variable $\{X(t), 0 \leq t \leq \infty\}$. The values assumed by $X(t)$ are called states and belong to a finite set. Moreover, $X(t)$ must satisfies the so-called memoryless property or Markovian assumption, i.e. for all integers n and for any sequence $t_0 < t_1 < \ldots < t_n < t$, the following equality holds.

$$Pr(X(t) = s \mid X(t_0) = s_0, X(t_1) = s_1, \ldots, X(t_n) = s_n) = Pr(X(t) = s \mid X(t_n) = s_n) \quad (5.12)$$

where $Pr(E)$ denotes the probability of the event E. The above property is called memoryless because the fact that the system was in state s_0 at t_0, s_1 at t_1 and so on is irrelevant.

When the probabilities of transitions out of the state $X(t)$ do not depend on the time t, the continuous-time Markov chain is said *homogeneous*. In the sequel, we shall simply say Markov chain for continuous-time homogeneous Markov chain.

Markov chains can be represented as graphs. Edges are labeled with transition rates. The rate λ_{ij} of a transition from state s_i to state s_j is defined as the limit when dt tends to 0 of the conditional probability $Pr(X(t+dt) = s_j \mid X(t) = s_i)$.

As an illustration, consider again the overflow protection system pictured in Figure 5.2. We shall now make the following assumptions:

- Level sensors have two modes of failure: a complete loss of the sensor and a drift which makes it send an erroneous information to the controller `C`. Complete losses are detected by the calculator (if it is working), but not drifts. When a complete loss is detected, the controller sends an alarm and puts the whole process in a fail safe mode.
- The loss of the sensors `LS1` and `LS2` are exponentially distributed with respective loss rates $\gamma_{\texttt{LS1}}$ and $\gamma_{\texttt{LS2}}$. Their drifts are also exponentially distributed, with respective drift rates $\lambda_{\texttt{LS1}}$ and $\lambda_{\texttt{LS2}}$.

– The failure of the controller C is exponentially distributed with a failure rate λ_{C}.
– The failure of the shutdown valve SDV1 is exponentially distributed with a failure rate λ_{V1}.
– The second safety barrier is degraded if either the shutdown valve SDV2 or the discharge valve DV is failed and all other components of the barrier are working properly. In this situation, the safety barrier still provides the intended capacity.
– The failure of the shutdown valve SDV2 and the discharge valve DV is exponentially distributed with a failure rate λ_{V2}.

Figure 5.12 shows a continuous-time Markov chain for the overflow protection system under the above assumption.

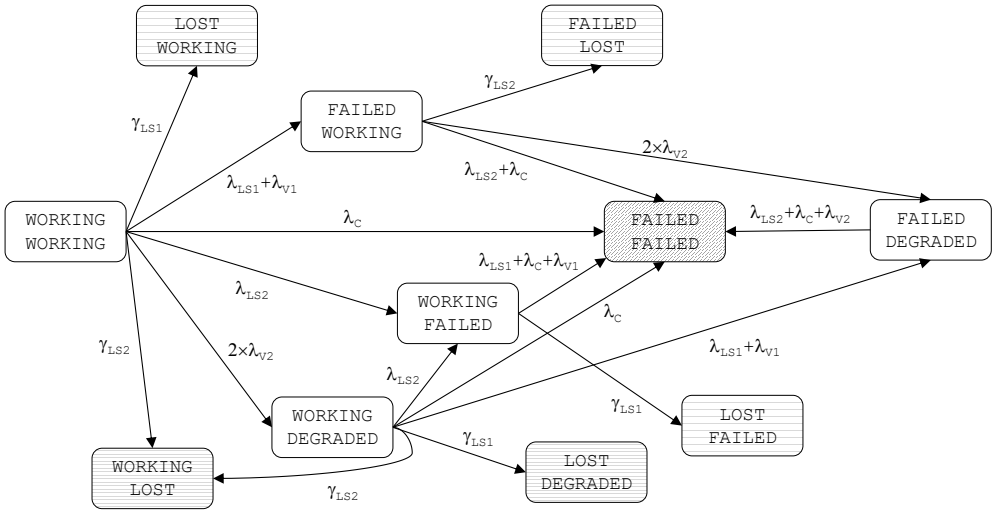


Figure 5.12: Continuous time Markov chain for the overflow protection system

States of the chain are characterized by the states of the two safety barriers.

The first one is either working, lost or failed. It is working if the sensor LS1, the controller C and the shutdown valve SDV1 are all working. It is lost if the sensor LS1 is lost, and the controller C and the shutdown valve SDV1 are working. Finally, it is failed if either the sensor LS1 is drifting, or the controller C is failed, or the shutdown valve SDV1 is failed.

The second one is either working, lost, degraded or failed. It is working if the sensor LS2, the controller C, the shutdown valve SDV2 and the discharge valve DV are all working. It is lost if the sensor LS2 is lost, the controller C is working and either the shutdown valve SDV2 or the discharge valve DV are working. It is degraded if the sensor LS2 and the controller C are working and either the shutdown valve SDV2 or the discharge valve DV are working while the other valve is failed. Finally, it is failed if either the sensor LS2 is drifting, or the controller C is failed, or both the shutdown valve SDV2 and the discharge valve DV are failed.

Transient probabilities

Let $X(t)$ be a Markov chain. According to the memoryless property, $Pr(X(t+d) = j \mid X(t) = i)$ does not depend on t, so we can write it simply as as $p_{ij}(d)$. If S is the set of all possible states of X, the following equality holds for all values of $i \in S$ and $d \in \mathbb{R}^{+}$.

$$\sum_{j \in S} p_{ij}(d) \quad = \quad 1 \tag{5.13}$$

Formally, the rate q_{ij} of the transition of the transition from state i to state j (at time any t) is defined as follows.

$$q_{ij} \stackrel{def}{=} \lim_{\Delta t \to 0} \left(\frac{p_{ij}(\Delta t)}{\Delta t} \right) \quad \texttt{for} \quad i \neq j \tag{5.14}$$

It is easy to verify that the following equality holds for all $i \in S$.

$$q_{ii} = - \sum_{j \in S, j \neq i} q_{ij} \tag{5.15}$$

The matrix square Q whose ijth element is q_{ij} is called the *infinitesimal generator* matrix or transition rate matrix for the Markov chain. Similarly, the matrix $P(\Delta t)$ whose ijth element is $p_{ij}(\Delta t)$ is called the transition probability matrix. In terms of matrices, the following equality holds.

$$Q = \lim_{\Delta t \to 0} \left(\frac{P(\Delta t) - I}{\Delta t} \right) \tag{5.16}$$

where I denotes for the unity matrix.

It can be shown, applying the Chapman-Kolmogorov equations, that the following equality holds.

$$P'(t) = P(t) \times Q \tag{5.17}$$

The solution to this equation is given by a matrix exponential:

Let $\pi(t)$ be the vector of probabilities to be in state i at time t. $\pi(t)$ is called the vector of *transient probabilities* (a time t). $\pi(0)$ denotes thus the vector of initial probabilities. It can be shown, from equality 5.17, that the following equality holds.

$$\pi(t) = \pi(0) \times \exp(t \times Q) \tag{5.18}$$

Steady state probabilities

For some applications, one is interested in *steady state probabilities*, i.e. on the probability distribution, if any, to which the process converges for large values of t. This can be done by solving the following equation.

$$\pi \times Q = 0 \tag{5.19}$$

with the additional constraint that:

$$\sum_s \pi_s = 1$$

In the framework of probabilistic risk and safety assessment, steady state probabilities are of much lower interest than transient ones. The reason is that, according to the famous quote of John Maynard Keynes: "*In the long run, we are all dead*". Any industrial system will eventually age in such way that it is no longer working.

Illustration

As an illustration, consider the Markov chain pictured in Figure 5.12. Assume that we merge all fail safe states, i.e. all states in which one of the two safety barriers is lost. Let order the states as follows.

1	2	3	4	5	6	7
WORKING WORKING	WORKING DEGRADED	WORKING FAILED	FAILED WORKING	FAILED DEGRADED	FAILED FAILED	LOST

Then, the infinitesimal generator of the Markov chain is as follows.

$$Q = \begin{pmatrix} -\mu_1 & 2\times\lambda_{V2} & \lambda_{LS2} & \lambda_{LS1}+\lambda_{V1} & 0 & \lambda_C & \gamma_{LS1}+\gamma_{LS2} \\ 0 & -\mu_2 & \lambda_{LS2}+\lambda_{V2} & \lambda_{LS1}+\lambda_{V1} & 0 & \lambda_C & \gamma_{LS1}+\gamma_{LS2} \\ 0 & 0 & -\mu_3 & 0 & 0 & \lambda_{LS1}+\lambda_C+\lambda_{V1} & \gamma_{LS1} \\ 0 & 0 & 0 & \mu_4 & 2\times\lambda_{V2} & \lambda_{LS2}+\lambda_C & \gamma_{LS2} \\ 0 & 0 & 0 & 0 & \mu_5 & \lambda_{LS2}+\lambda_C+\lambda_{V2} & \gamma_{LS2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where,

$$\begin{aligned} \mu_1 &= -\gamma_{LS1}-\lambda_{LS1}-\lambda_C-\lambda_{V1}-\gamma_{LS2}-\lambda_{LS2}-2\times\lambda_{V2} \\ \mu_2 &= -\gamma_{LS1}-\lambda_{LS1}-\lambda_C-\lambda_{V1}-\gamma_{LS2}-\lambda_{LS2}-\lambda_{V2} \\ \mu_3 &= -\gamma_{LS1}-\lambda_{LS1}-\lambda_C-\lambda_{V1} \\ \mu_4 &= -\gamma_{LS2}-\lambda_{LS2}-\lambda_C-2\times\lambda_{V2} \\ \mu_5 &= -\gamma_{LS2}-\lambda_{LS2}-\lambda_C-\lambda_{V2} \end{aligned}$$

The above matrix illustrates a very common property to matrices involved in Markov chains. Namely, these matrices are sparse, i.e. they contain a very large number of zeros. The larger the matrix, the higher the proportion of zeros.

5.5.3 Assessment Algorithms

To solve analytically equation 5.18, one can use Laplace transforms, see e.g. (Kreyszig 2011) (Chapter 6). This method is very important from a theoretical view point but almost useless in practice as it is practicable only for very small Markov chains (with less than 10 states at best).

In practice, Markov chains are solved numerically (Stewart 1994).

Numerical solutions rely on the development the matrix exponential according to the Taylor series:

$$\exp(t\times Q) = \lim_{n\to\infty}\sum_{k=0}^{n}\frac{(t\times Q)^k}{k!} \tag{5.20}$$

However, the above equation cannot be applied directly, for two reasons. First, calculation powers of matrices makes them loose their sparsity. As already pointed out, the infinitesimal generators of Markov chains resulting from practical applications are sparse (contain mostly 0's). This property can be used to design efficient encoding of the matrices. Namely, matrices are stored as lists of cells rather than as two-dimensional arrays. This reduces dramatically their memory occupation, making it possible to deal with models with up to several millions of states

and transitions. Second, due to the special encoding of floating point numbers on computers, coefficients must be kept between 0 and 1 because of rounding errors.

Two ideas are thus applied.

The first idea consists in discretizing the calculation, i.e. in splitting the mission time t into small time intervals dt ($t = n \times dt$).

$$\pi(0) \times \exp(t.Q) = (\cdots((\pi(0) \times \exp(dt \times Q)) \times \exp(dt \times Q)) \times \cdots \times \exp(dt \times Q)) \quad (5.21)$$

By choosing dt sufficiently small, one keeps coefficients of the matrix $dt \times Q$ between 0 and 1.

The second idea consists in calculating progressively the factors $\frac{(t \times Q)^k}{k!}$ by performing only products of vectors by matrices.

$$\pi \times \frac{(dt \times Q)^k}{k!} = \frac{1}{k} \times \left(\pi \times \frac{(dt \times Q)^{k-1}}{(k-1)!} \right) \times (dt \times Q) \quad (5.22)$$

Note that the matrix $dt \times Q$ can be calculated once for all and that both calculations can be stopped when they are "stabilized".

Several algorithms can be designed based on the above ideas, see e.g. (Rauzy 2004) for a practical comparison.

5.6 Stochastic Petri Nets

Introduced by Carl Adam Petri in his PhD thesis (Petri 1962), Petri nets are a simple but powerful mathematical model to describe distributed systems. They belong to the wide class of discrete event systems.

Over the years, Petri nets acquired an incredible popularity which, in author's opinion, is not fully deserved. Nevertheless, Petri nets are an important element in the toolbox of both the system engineer and the reliability engineer, which means that students working in these domains should be familiar with them.

5.6.1 Definition

A *Petri net* is directed graph with two types of nodes, called respectively *places* and *transitions*. Technically the graph is bi-partite, i.e. arcs go from a place to a transition or vice versa, but never between places or between transitions. A place P with an out-arc (directed egde) going to a transition T is called an *input place* of T. Symmetrically, a place P with a in-arc coming from a transition T is called an *output place* of T.

Places of a Petri net contains a certain number of *tokens*. A *marking* of the Petri net is a function that associates a number of tokens to each place.

A transition is *enabled* if all its input places contain at least one token. *Firing* the transition consists in:

- Removing a token in each input place of the transition;
- Adding a token in each output place of the transition.

Timed Petri nets is a Petri net in which delays are associated with transitions, i.e. if a transition gets enabled at the date t, it is actually fired at date $t + d$, where d is the delay associated with the transition.

Stochastic Petri nets are timed Petri nets in which transitions are associated with stochastic delays. In some variants of stochastic Petri nets, deterministic delays are allowed as well.

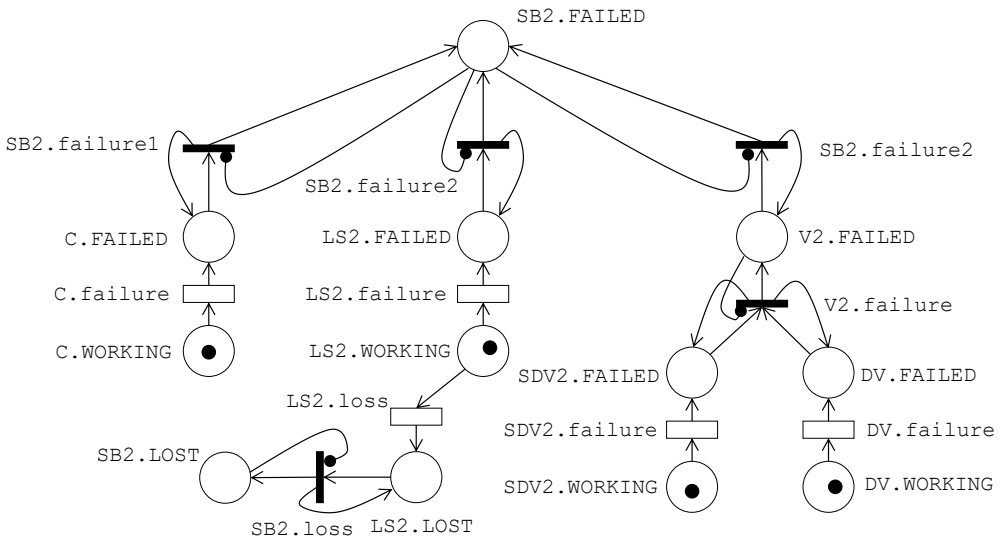


Figure 5.13: A Petri net

Figure 5.13 shows a stochastic Petri net representing the behavior of the second safety barrier of the overflow protection system.

Places are represented by circles, transitions by rectangles. Hence, `C.WORKING` and `C.FAILED` are places and `C.failure` and `SB2.failure1` are transitions. The unique input place of the transition `C.failure` is the place `C.WORKING`. Its unique output place is the transition `C.FAILED`. This reflects simply that the only condition for the controller `C` to fail is that it is working.

Similarly, the two input places of the transition `SB2.failure1` are `C.FAILED` and `SB2.FAILED`. These two places are also the output places of the transition. The arc joining `SB2.FAILED` to `SB2.failure1` is terminated with a circle instead of an arrow. This indicates that is this arc is an *inhibitor arc*. It says that the transition is enabled only if there is no token in its source place `SB2.FAILED` (and of course, no token is withdrawn from this place when the transition is fired).

The two transitions `C.failure` and `SB2.failure1` are represented with different rectangles.

Transitions represented by thick, white rectangles are *stochastic transitions*. They are associated with a cumulative probability distribution that associates a random delay with the transition. Conversely to Markov chains, delays associated with transitions in stochastic Petri nets can be any (inverse of) a cumulative probability distribution (Weibull distribution, censored normal distribution, empirical distribution...).

Transitions represented by thin, black rectangles are *immediate transitions*, i.e. they are associated with a null delay and are thus fired as soon as they are enabled. In the case of the transition `SB2.failure1`, this reflects the fact that the transition that the safety barrier `SB2` is failed as soon as the controller `C` is.

Tokens are represented by small black dots in the places. In the marking pictured in Figure 5.13, places `C.WORKING`, `LS2.WORKING`, `SDV2.WORKING` and `DV.WORKING` contain a token, while the others are empty. This reflects the initial state of the safety barrier.

In this marking, four transitions are thus enabled: `C.failure`, `LS2.failure`, `SDV2.failure` and `DV.failure`. At time $t = 0$, a delay is thus drawn at random for these transitions (but according to their respective probability distributions). Assume that these delays are respectively 3043.5, 2876.2, 1065.3 and 2777.1.

As the transition `SDV2.failure` has the smallest delay, it is fired first, at time $t = 0+$

$1065.3 = 1065.3$. The firing this transition removes the token from the place `SDV2.WORKING` and adds a token in the place `SDV2.FAILED`. The transition `SDV2.failure` is no longer enabled. The transition `V2.failure` is not enabled neither because its input place `DV.FAILED` is still empty.

We go thus to the next scheduled transition, i.e. `DV.failure` at time $t = 0 + 2777.1 = 2777.1$. This transition is fired, removing the token in the place `DV.WORKING` and adding one in the place `DV.FAILED`.

Now the transition `V2.failure` is enabled: there is a token in places `SDV2.FAILED` and `DV.FAILED` and no token in place `V2.FAILED`. Firing the transition `V2.failure` removes a token from the places `SDV2.FAILED` and `DV.FAILED` and adds one in the places... `SDV2.FAILED` and `DV.FAILED`. In other words, nothing changes for these two places. However, firing the transition `V2.failure` also adds a token in the place `V2.FAILED`. This has two consequences: first, the transition `V2.failure` stops to be enabled. Second, the transition `SB2.failure` becomes enabled. As it is an immediate transition, it is fired (still at time $t = 2777.1$). Its firing adds eventually a token in the place `SB2.FAILED` reflecting the fact that the safety barrier `SB2` is lost if the two valves `SDV2` and `DV` are lost.

In practice, properties of stochastic Petri nets are assessed by means of Monte-Carlo simulation.

5.6.2 Discussion

Let us put something first: stochastic Petri nets are an important tool for reliability engineering.

Stochastic Petri nets are a completely formal model. A stochastic Petri net is a mathematical object, with a clear syntax and a semantics. Some authors are disdaining Petri nets pretending that equations, with analytic solutions would have some mathematical superiority. These authors just show their ignorance about... mathematics. Most of the systems of differential equations do not have analytical solutions. Those which have are just like little islands in a vast ocean. In practice, casting at all force problems into analytically solvable systems of differential equations just lead to abstract non-sense and over approximations of the reality. In contrast, Petri nets and similar models are (relatively) easy to design and make it possible to represent (relatively) faithfully the phenomena under study.

Moreover, Petri nets and similar models make it possible to distinguish the description of the problem, the model, from its resolution, the assessment algorithm. This clear separation makes in turn possible to develop highly optimized assessment algorithms and to apply possibly different algorithms on the same model, something which is impossible to do when the model and its assessment algorithm are mixed up into, for instance, a Matlab® script.

Stochastic Petri nets present another advantage: conversely to Markov chains for instance, they represent implicitly the set of possible executions. This property is a *sine qua non* condition to describe complex systems.

This said, the author's opinion is that the popularity of Petri nets rely on a series of misunderstandings.

First, Petri nets are attractive because their graphical representation. It is possible to animate a Petri net on a computer screen, with tokens moving from places to places. Such graphical animations are however limited to very simple nets that are much to simple to represent any real system. Petri nets representing real systems contain dozens if not hundreds of places and transitions. Their global visualization is just impossible, hence smashing the readability argument. The stochastic Petri net pictured in Figure 5.13 is already quite hairy. In reality, Petri nets tend to be a "write-only" formalism: once written, a Petri net model is unreadable, therefore impossible to maintain. The same applies, no more, no less, to any graphical representation of state automata.

Second, the generic term "Petri net" covers an incredible variety of extensions of the initial concept, some very far from it. Colored Petri nets (Jensen 2014) for instance have a much higher

expressive power, leaving the realm of state automata to jump into the one of process algebras. Even if we restrict our attention to stochastic Petri nets, those studied in the book by Ajmone-Marsan & alii (Ajmone-Marsan et al. 1994) are quite different than those studied in the book by Signoret and Leroy (Signoret and Leroy 2021), both books being otherwise excellent. In a word, no one knows exactly what one is speaking about when using the term (stochastic) Petri nets.

Third, the original idea of Carl Adam Petri was to use linear algebra to study the property of his net. Determining whether a given marking is reachable from an initial marking is actually a decidable, although its algorithmic complexity is very high. Alas, this nice decidability property is lost in any useful extension of the mathematical model (Esperza 1998). For instance, the simple introduction of inhibitor arcs smashes it out. The same applies for the use of linear algebra which is possible only because strong restrictions are put on the pre- and post-conditions of transitions. The guarded transition systems model, which we shall study in Chapters 7 and 8 is strictly more expressive than the one of stochastic Petri nets, even widely extended as suggested in (Signoret and Leroy 2021), without adding any algorithmic complexity.

5.7 Limits

Probabilistic risk and safety assessment is a great tool, nowadays widely used in industry. However, it has limits that safety analysts should be aware of. The objective of this section is to discuss some of these limits.

The first limit is indeed the epistemic uncertainty which results from the lack of knowledge on the phenomena at stake. One cannot expect correct information popping out from an incorrect model, so to say by magic. To put it crudely, "garbage in, garbage out". In most of the case however, models capture a significant part of the problems at stake. But they may miss something, typically because this something results from an extremely rare combination of events. This is what is behind the metaphor of "black swan" defended by Taleb (Taleb 2010): for centuries, it has been widely believed that all swans were white. Black swans were thus considered as phantasmagoric creatures... until dutch explorers saw some in western Australia. Systems engineers do their best to develop robust systems. It may be hard for them to foresee why their systems may fail.

Another source of epistemic uncertainty is the lack of reliability data. By definition, incidents and accidents are rare. Industrial systems like chemical plants, nuclear plants or airplanes are designed to be run for years without the least incident. Consequently, experience feedback is often insufficient to get a clear picture of what can happen. This is the reason why, despite the hype on the subject at the time I am writing these line, machine learning has little chance to replace more traditional modeling techniques. This does not mean indeed that it cannot be useful in complement and in coordination with these techniques.

Aside epistemic uncertainty, another limits of probabilistic risk and safety analysis is the computational complexity of the calculations at stake. This question will be extensively discussed in Chapter 9.

Finally, one should never neglect organizational and economical constraints. Obviously, a safer design may be more costly than a less safer one, both in terms of production and operation. But economical constraints apply to probabilistic risk and safety analyses themselves. Unless they are strictly required by laws and regulations, they are often seen as a cost (with reason). The question is then to make the best possible analysis within a limited budget and not the best possible analysis that could be done. The experience shows that the willingness to invest time and money on safety analysis decreases with the time after a significant accident.

To apply the modeling frameworks we shall study in the next Chapters, the analyst must be familiar with a number of concepts, methods and tools. Even though on the medium and long term, training analysts is profitable, on the short term, it has a cost. As of today, this initial cost is

probably the main limitation of the full scale deployment of state of the art probabilistic risk and safety analysis methods and tools.

5.8 Further Readings

Here follows a few of references of interest.

The book by Andrews and Moss (Andrews and Moss 2002) is a very good introduction to reliability engineering. The book by Kumamoto and Henley (Kumamoto and Henley 1996), although now a bit dated, is a very good introduction to probabilistic risk assessment in nuclear industry. Finally, the book by Signoret and Leroy (Signoret and Leroy 2021) is a reference regarding probabilistic safety assessment in oil and gas industry, written by two influential (for good reasons) practitioners.

The reference book on numerical assessment of Markov chains is the book by Stewart (Stewart 1994).

The article by Murata (Murata 1989) is an excellent introduction to Petri nets and their applications. Books by Ajmone-Marsan & alii (Ajmone-Marsan et al. 1994) and Signoret and Leroy (Signoret and Leroy 2021), are very good presentation of stochastic Petri nets.

5.9 Exercises and Problems

The two following problems illustrate how probability distributions can be associated with failure modes from experience feedback. To answer the questions of these problems, it is highly recommended to use a software, e.g. Excel®, Matlab®, R, Python...

Problem 5.1 – Failure Distribution of Pumps. A company operates in similar conditions a fleet of 200 identical pumps. Each time a pump fails, its failure date is recorded. After one year (8760 hours) of operation, records are as indicated in Table 5.9. Pumps that are not mentioned in the table have worked without failure during the full operation year.

Question 1. Assuming that these failure dates obey an exponential distribution,

a) Find a failure rate that matches the data;
b) Using this failure rate calculate the probabilities of failure at $t = 730$ (1 month), $t = 1460$ (2 months), and so on until one year;
c) Draw this failure distribution (from $t = 0$ to $t = 8760$).

Question 2. Recall that the *Kaplan-Meier estimator* is built by considering the dates of failure in increasing order. At the first failure date 1 out of the 200 pumps are failed. At the second failure date, 2 are failed. And so on. One can build in this way an empirical distribution.

a) Build the Kaplan-Meier estimator for the data of Table 5.9;
b) Using this estimator, calculate the probabilities of failure at $t = 730$ (1 month), $t = 1460$ (2 months), and so on until one year;
c) Draw this failure distribution (from $t = 0$ to $t = 8760$).

Question 3. Let us define the distance between the two distributions at date d as the absolute value of the difference of the probabilities calculated at this date.

a) Using the probabilities calculated in the previous questions, estimate the average distance between the two distributions.
b) What can you conclude?

■

Table 5.9: Failure dates of pumps

Pump code	Failure date	Pump code	Failure date	Pump code	Failure date
2	895	16	8386	24	2085
29	303	53	3717	62	4431
64	7167	73	5563	83	7309
91	292	92	279	105	2423
115	6141	134	6165	137	3151
146	2610	152	2723	161	6875
164	3617	166	7481	170	1222
176	3650	185	8433	198	2179

Table 5.10: Failure dates of sensors

Sensor code	Failure date	Sensor code	Failure date	Sensor code	Failure date
2	3802	16	8015	24	5040
29	2650	34	8260	53	6111
62	6479	64	7606	73	6990
83	7656	91	2618	92	2579
97	8153	105	5298	115	7224
134	7233	137	5783	146	5431
152	5509	161	7501	164	6056
166	7715	170	4218	171	8657
176	6074	177	8264	185	8029
198	5114				

Problem 5.2 – Failure Dates of Sensors. The same company as in problem 5.1 has installed a vibration sensor on each pump. The 200 sensors are identical and thus operated in similar conditions. As for the pumps, each time a sensor fails, its failure date is recorded. After one year (8760 hours) of operation, records are as indicated in Table 5.10. Sensors that are not mentioned in the table have worked without failure during the full operation year.

Question 1. Assuming that these failure dates obey an exponential distribution,

a) Find a failure rate that matches the data;
b) Using this failure rate calculate the probabilities of failure at $t = 730$ (1 month), $t = 1460$ (2 months), and so on until one year;
c) Draw this failure distribution (from $t = 0$ to $t = 8760$).

Question 2. We can go for the Kaplan-Meier estimator.

a) Build the Kaplan-Meier estimator for the data of Table 5.10;
b) Using this estimator, calculate the probabilities of failure at $t = 730$ (1 month), $t = 1460$ (2 months), and so on until one year;
c) Draw this failure distribution (from $t = 0$ to $t = 8760$).

Question 3. Let us define the distance between the two distributions at date d as the absolute value of the difference of the probabilities calculated at this date.

a) Using the probabilities calculated in the previous questions, estimate the average distance between the two distributions.
b) What can you conclude?

■

Problem 5.3 – Binary Decision Trees. Consider the fault tree pictured in Figure 5.14. Assume that the basic events of this fault tree are respectively associated with the following probability distributions.

Basic event	Distribution
A	exponential distribution, with failure rate $1.23 \times 10^{-6}\ h^{-1}$
B	Weibull distribution, with scale parameter $9.87 \times 10^{4}\ h$ and shape parameter 3
C	Weibull distribution, with scale parameter $7.96 \times 10^{4}\ h$ and shape parameter 3
D	exponential distribution, with failure rate $4.32 \times 10^{-6}\ h^{-1}$

Question 1. Build the truth table for the top event of this fault tree. It is recommended to use a spreadsheet tool to do so.

Question 2. Use this truth table to calculate the probability of the top event at $t = 8760\ h$.

Question 3. Design a binary decision tree for the fault tree using the variable order $\mathtt{A} < \mathtt{B} < \mathtt{C} < \mathtt{D}$.

Question 4. Use this binary decision tree to calculate the probability of the top event at $t = 8760\ h$.

Question 5. Find a more compact binary decision tree to encode the top event of the fault tree.

Question 6. Use this binary decision tree to calculate the probability of the top event at $t = 8760\ h$.

For the braves:

Question 4. Design a program, in your favorite programming language or spreadsheet, to build binary decision trees and to calculate their probability at different mission time.

Question 5. Use this program to calculate to calculate the probability of the top event at $t = 730, 1460, 2190, \ldots 8760\ h$.

■

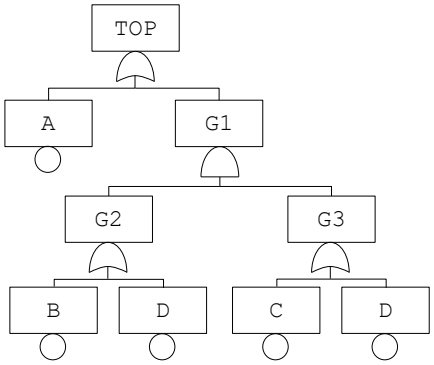


Figure 5.14: A fault tree

Problem 5.4 – 2-out-of-4 System. Consider a 2-out-of-4 system, i.e. a system that is working if at least 2 out of its 4 identical components are working. Assume that the failures of its components are exponentially distributed with a failure rate $1.14 \times 10^{-4}\ h{-}1$.

Question 1. Build the truth table for this system. It is recommended to use a spreadsheet tool to do so.

Question 2. Use this truth table to calculate the probability of failure at $t = 8760\ h$.

Question 3. Design a binary decision tree for the system.

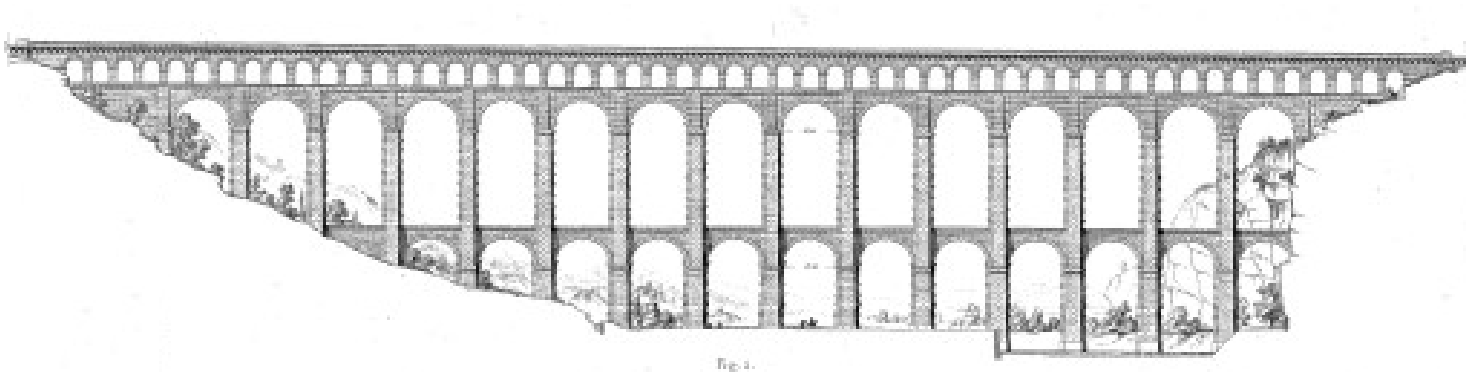
Question 4. Use this binary decision tree to calculate the failure at $t = 8760\ h$.

For the braves:

Question 4. Design a program, in your favorite programming language or spreadsheet, to build binary decision trees and to calculate their probability at different mission time.

Question 5. Use this program to calculate to calculate the probability of the top event at $t = 730, 1460, 2190, \ldots 8760\ h$.

■



6. Systems of Boolean Equations

Key Concepts

- Boolean formulas and functions
- Systems of Boolean equations
- Fault Trees and Reliability Block Diagrams
- Basic-event, intermediate event, top event, house event
- State variable, flow variable, root variable, source variable
- Failure model, probability distribution
- Literal, product, minterm
- Prime implicants, Minimal cutsets
- Normal forms
- Sum of minimal cutsets, sum of disjoint products
- Binary decision trees, binary decision diagrams
- Availability, top event probability
- Shannon decomposition
- Rare Event Approximation, MinCut Upper Bound, Pivotal Upper Bound
- Critical states, importance measures

This chapter deals with Boolean reliability models such as fault trees and reliability block diagrams. More exactly, it deals with systems of stochastic Boolean equations, which are the mathematical framework in which these models are actually developed. It presents S2ML+SBE, the corresponding language of the S2ML+X family. It describes also the main calculations that are performed on these models: extraction of minimal cutsets, calculation of the top event probability and importance measures... All these notions are formally introduced.

Throughout the chapter, we shall use the calculation engine XFTA as integrated into the AltaRica Wizard environment to perform all experiments. Due to space and scope limitations, this chapter cannot however be a full introduction to XFTA. The interested reader can refer to the XFTA book (Rauzy 2020) for an in-depth presentation[1].

[1] Both AltaRica Wizard and the XFTA book can be freely downloaded from author's website

6.1 Preliminaries

Before entering into the formal definition, this section gives some intuitions about systems of (stochastic) Boolean equations.

6.1.1 Case Study

To illustrate our presentation, we shall consider the following case study inspired from (Signoret and Leroy 2021).

■ **Case Study 3 – Pumping System.** Figure 6.1 shows a pumping system, in charge of delivering pressurized water in case of an emergency, typically of a fire outburst in a plant.

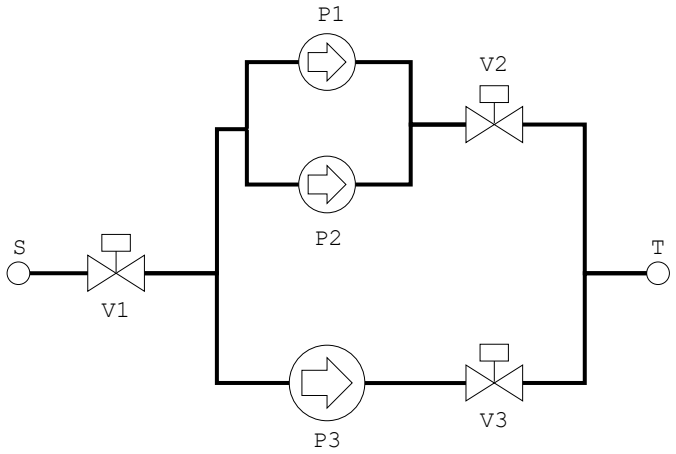


Figure 6.1: A simple pumping system

The system consists of the following elements.
– A valve `V1` that isolates the system when it is not in operation.
– The upper pump train which consists of two small pumps `P1` and `P2` and a valve `V2`. `V2` isolates the upper train when the system is not in operation.
– The lower pump train which consists in the big pump `P3` and the valve `V3`. `V3` isolates the lower train when the system is not in operation.

The three pumps are located in the same room. Valves are located outside this room.
All its elements may fail:
– Valves may fail to open when the system is put in service.
– Pumps may fail in operation.

For now, we shall assume the following reliability data for failures of pumps and valves.

Component	Failure model
Primary Valve (`V1`)	Point estimate probability 0.01
Secondary Valves (`V2`, `V3`)	Point estimate probability 0.01
Small pumps (`P1`, `P2`)	Exponential distribution, failure rate: $5.67 \times 10^{-3}\ h^{-1}$
Large pump (`P3`)	Exponential distribution, failure rate: $4.21 \times 10^{-5}\ h^{-1}$

Both pump trains are sufficient in isolation to pump out the required amount of water. ■

6.1.2 Fault Trees and Reliability Block Diagrams

As explained in the previous chapter, fault trees and reliability block diagrams are the most popular formalisms to design probabilistic risk assessment models. We shall propose a methodology to

design such models in the next section. But for now, the system is simple enough to allow a direct approach.

Figure 6.2 shows a fault tree describing the failures of our pumping system.

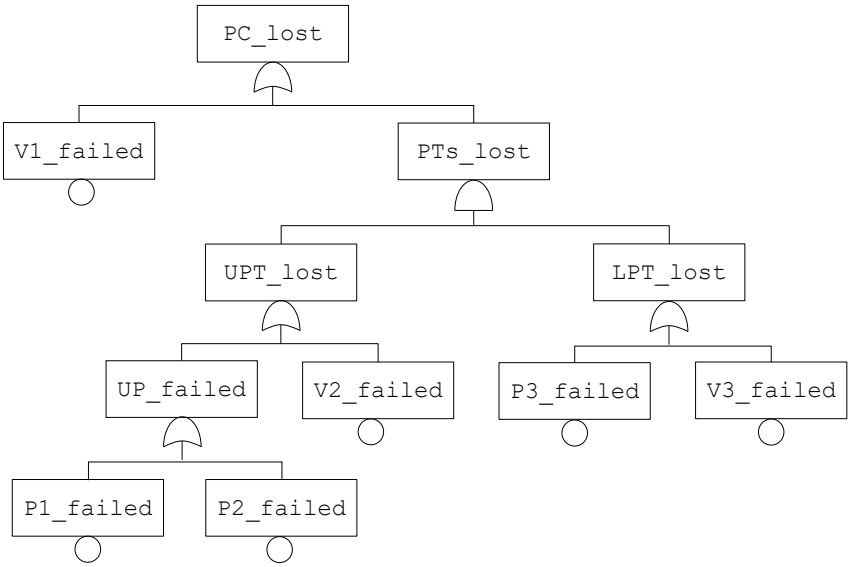


Figure 6.2: A fault tree describing the failures of the pumping system pictured in Figure 6.1

The logic of construction of this fault tree is the following.

- The pumping capacity is lost (top event `PC_lost`) if either the valve `V1` is failed closed (basic event `V1_failed`) or both pumping trains are lost (intermediate event `PTs_lost`).
- The upper pump train is lost (intermediate event `UPT_lost`) if either the upper pump group is failed (intermediate event `UP_failed`) or the valve `V2` is failed closed (basic event `V2_failed`).
- The upper pump group is failed, if either the upper pump `P1` is failed (basic event `P1_failed`) or the upper pump `P2` is failed (basic event `P2_failed`).
- Finally, the lower pump train is lost (intermediate event `LPT_lost`) if either the lower pump `P3` is failed (basic event `P3_failed`) or the valve `V3` is failed closed (basic event `V3_failed`).

Figure 6.3 shows a hierarchical reliability block diagram describing the functioning of our pumping system.

This model follows closely the physical architecture of the system. It is actually very close to the process and instrumentation diagram of Figure 6.1. Note however a subtlety: although both pumps `P1` and `P2` are required to make the upper train available, they are represented as if they were in parallel on the reliability block diagram of Figure 6.3, as they are physically operating in parallel.

6.1.3 Intuitive Definition

Now, graphical models of Figures 6.2 and 6.3 represent the same system, at the exact same level of abstraction. Both are Boolean models, i.e. that they consider that each component of the system is either working or failed, and describe the failures of the systems as a function of the failures of its components. In a word, they are equivalent. They must be equivalent.

The question is thus how to show this equivalence?

The answer to this question can only come from the definition of a clear syntax and a sound semantics for both types of models, a topic that is unfortunately ignored by textbooks of reliability

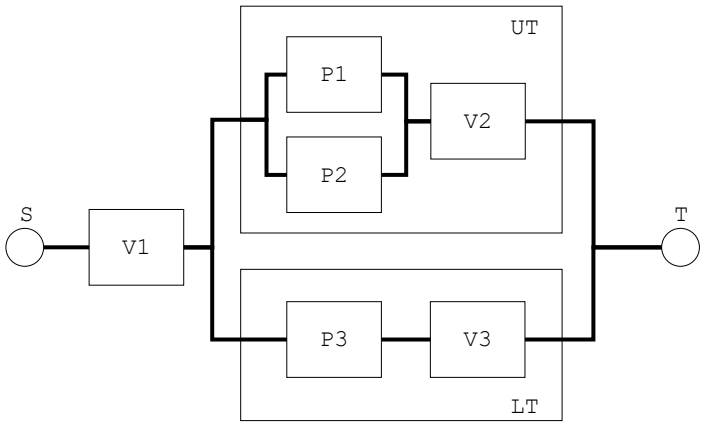


Figure 6.3: A reliability block diagram describing the functioning of the pumping system pictured in Figure 6.1

engineering.

Let us start with the fault tree pictured in Figure 6.2. This fault tree is the graphical representation of the system of Boolean equations given in Figure 6.4.

$$
\begin{aligned}
\texttt{PC\_lost} &= \texttt{V1\_failed} \vee \texttt{PTs\_lost} \\
\texttt{PTs\_lost} &= \texttt{UPT\_lost} \wedge \texttt{LPT\_lost} \\
\texttt{UPT\_lost} &= \texttt{UP\_failed} \vee \texttt{V2\_failed} \\
\texttt{UP\_failed} &= \texttt{P1\_failed} \vee \texttt{P2\_failed} \\
\texttt{LPT\_lost} &= \texttt{P3\_failed} \vee \texttt{V3\_failed}
\end{aligned}
$$

Figure 6.4: The system of Boolean equations graphically represented by the fault tree pictured in Figure 6.2

Each intermediate event of the fault tree is uniquely defined by, i.e. is the left member of, an equation. Right hand side formulas of equations are definition of their left hand side and follow the logic of construction of the fault tree.

Basic events show up only in right hand side of equations. The top event is the only intermediate event that does not show up in a right member of an equation. Finally, the system of equations is data-flow, i.e. no intermediate event depends eventually on itself.

Now, consider the reliability block diagram pictured in Figure 6.3. Obviously, its logic of construction differs from the one of the fault tree. To start with, it is in some sense dual: the former represents how the system works, while the latter how it may fail. Still the former as the latter are graphical representation of two equivalent systems of Boolean equations

Let us first consider basic blocks. These blocks have an input and an output and represent the failure of a certain component `C`. The relation that links the two is as follows.

$$\texttt{C\_output} = \texttt{C\_input} \wedge \neg\texttt{C\_failed}$$

In other words, there is a flow in output of the component if there is a flow in input and the component is working (not failed). Once this point established, it suffices to connect inputs and outputs of blocks according to the logic of the diagram. Eventually, this gives the system of Boolean equations given in Figure 6.5 (in the figure, the two sets of equations are separated, for the sake of clarity).

$$
\begin{aligned}
\texttt{V1\_output} &= \texttt{V1\_input} \wedge \neg\texttt{V1\_failed}\\
\texttt{V2\_output} &= \texttt{V2\_input} \wedge \neg\texttt{V2\_failed}\\
\texttt{V3\_output} &= \texttt{V3\_input} \wedge \neg\texttt{V3\_failed}\\
\texttt{P1\_output} &= \texttt{P1\_input} \wedge \neg\texttt{P1\_failed}\\
\texttt{P2\_output} &= \texttt{P2\_input} \wedge \neg\texttt{P2\_failed}\\
\texttt{P3\_output} &= \texttt{P3\_input} \wedge \neg\texttt{P3\_failed}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{S\_output} &= \texttt{true}\\
\texttt{V1\_input} &= \texttt{S\_output}\\
\texttt{UT\_input} &= \texttt{V1\_output}\\
\texttt{P1\_input} &= \texttt{UT\_input}\\
\texttt{P2\_input} &= \texttt{UT\_input}\\
\texttt{V2\_input} &= \texttt{P1\_output} \wedge \texttt{P2\_output}\\
\texttt{UT\_output} &= \texttt{V2\_output}\\
\texttt{LT\_input} &= \texttt{V1\_output}\\
\texttt{P3\_input} &= \texttt{LT\_input}\\
\texttt{V3\_input} &= \texttt{P3\_output}\\
\texttt{LT\_output} &= \texttt{V3\_output}\\
\texttt{T\_input} &= \texttt{UT\_output} \vee \texttt{LT\_output}
\end{aligned}
$$

Figure 6.5: The system of Boolean equations graphically represented by the reliability block diagram pictured in Figure 6.3

Yet, we are interested in characterizing the failures of the pumping system, not its functioning. To do so, we have to take the dual of the above equations. This is achieved by applying de Morgan's law, see appendix B.3.

For basic components, we thus rewrite the equations as follows, by replacing $\neg$`C_input` by `C_disconnected` and $\neg$`C_output` by $\neg$`C_lost`.

$$
\begin{aligned}
\texttt{C\_output} &= \texttt{C\_input} \wedge \neg\texttt{C\_failed}\\
&\downarrow\\
\neg\texttt{C\_output} &= \neg\texttt{C\_input} \vee \texttt{C\_failed}\\
&\downarrow\\
\texttt{C\_lost} &= \texttt{C\_disconnected} \vee \texttt{C\_failed}
\end{aligned}
$$

Applying the same ideas to connections gives system of Boolean equations shown in Figure 6.6.

What is important here is that the above transformation preserves the semantics: The two systems of Boolean equations are strictly equivalent.

Still preserving the semantics, we can simplify the system of equations of the reliability block diagram: constants can be propagated and intermediate variables replaced by their definitions. Doing so, reordering equations and renaming `T_disconnected` into `PC_lost`, we can get the equivalent system given in Figure 6.7.

It is now easy to check that, although the above system is not the same as the one we wrote for the fault tree, these two systems are equivalent: any basic event valuation that satisfies `PC_lost`

```
          V1_lost  =  V1_disconnected ∨ V1_failed
          V2_lost  =  V2_disconnected ∨ V2_failed
          V3_lost  =  V3_disconnected ∨ V3_failed
          P1_lost  =  P1_disconnected ∨ P1_failed
          P2_lost  =  P2_disconnected ∨ P2_failed
          P3_lost  =  P3_disconnected ∨ P3_failed

           S_lost  =  false
V1_disconnected  =  S_lost
UT_disconnected  =  V1_lost
P1_disconnected  =  UT_disconnected
P2_disconnected  =  UT_disconnected
V2_disconnected  =  P1_lost ∨ P2_lost
          UT_lost  =  V2_lost
LT_disconnected  =  V1_lost
P3_disconnected  =  LT_disconnected
V3_disconnected  =  P3_lost
          LT_lost  =  V3_lost
 T_disconnected  =  UT_lost ∧ LT_lost
```

Figure 6.6: Dual system of Boolean equations graphically represented by the reliability block diagram pictured in Figure 6.3

```
          PC_lost  =  UT_lost ∧ LT_lost
          UT_lost  =  V2_disconnected∨ V2_failed
V2_disconnected  =  P1_lost ∨ P2_lost
          P1_lost  =  V1_failed ∨ P1_failed
          P2_lost  =  V1_failed ∨ P2_failed
          LT_lost  =  P3_lost ∨ V3_failed
          P3_lost  =  V1_failed ∨ P3_failed
```

Figure 6.7: Simplified version of the system of Boolean equations of Figure 6.6

in the former satisfies it in the latter, and vice-versa.

The fault tree representation may look much more direct and simple at a first glance. Nevertheless, in practice, it may be of interest to reflect the (physical) architecture of the system in the model. The transformations we have made are not done by the analyst, but by the processing tool, XFTA for the matter. These transformations are computationally efficient (their complexity is nearly linear in the size of the model). The two approaches can thus be taken, none of them being superior to the other in terms of computational complexity.

6.2 Formal Definitions

It is now time to enter into formal definitions.

6.2.1 Boolean Formulas

As said in the previous chapter, 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 ")".

Definition 6.2.1 – 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 $f, f_1 \ldots f_n$ 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, $E_1 \ldots$, and Boolean formulas by lower case letters, possibly with subscripts, e.g. f, g, $f_1 \ldots$. 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 $\mathrm{var}(f)$. 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 $\mathrm{var}(f)$ set of Boolean variables occurring in f is the smallest set such that:
- $\mathrm{var}(0) = \mathrm{var}(1) = \emptyset$.
- $\mathrm{var}(E) = \{E\}$ for any variable E.
- If $f, f_1, \ldots f_n$ are formulas, then $\mathrm{var}(f_1 \vee \cdots \vee f_n) = \mathrm{var}(f_1) \cup \cdots \cup \mathrm{var}(f_n)$, $\mathrm{var}(f_1 \wedge \cdots \wedge f_n) = \mathrm{var}(f_1) \cup \cdots \cup \mathrm{var}(f_n)$ and $\mathrm{var}(\neg f) = \mathrm{var}(f)$.
- Finally, if f is a formula, then $\mathrm{var}(\,(f)\,) = \mathrm{var}(f)$.

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

6.2.2 Systems of Boolean Equations

As we have seen in Section 6.1, although presenting fault trees and reliability block diagrams as Boolean formulas is a convenient abstraction, in reality they are systems of Boolean equations.

Definition 6.2.2 – 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:

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

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

The notation $\mathrm{var}(f)$ 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 $\mathrm{var}(S)$ is the union of the set of variables

showing up in equations:

$$\mathrm{var}(S) \stackrel{def}{=} \bigcup_{V=f\in S} \{V\} \cup \mathrm{var}(f)$$

A variable V of $\mathrm{var}(S)$ 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 6.2.2 does not prevent loops in the definitions of variables.

Let S be a system of Boolean equations built over a sets $\mathscr{V}$ 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 6.2.3 – Data-Flow Systems. Let S be a system of Boolean equations over the set $\mathscr{V}$ 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.

6.2.3 Semantics

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

Definition 6.2.4 – Truth assignment. Let $\mathscr{V} = \{V_1, \ldots V_n\}$, $n \geq 1$, be a set of Boolean variables. A *truth assignment* of $\mathscr{V}$ is a mapping from $\mathscr{V}$ 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 $\mathscr{V}$ and f, f_1,... f_n be Boolean formulas built over $\mathscr{V}$, 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 $\mathscr{V}$ and S be a system of Boolean equations built

over $\mathcal{V}$. The σ is *valid* if the following condition holds.

$$\forall V = f \in S \; \sigma(V) = \sigma(f) \tag{6.1}$$

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.

Property 6.1 – Uniqueness of values of flow variables. Let S be a data-flow system of Boolean equations built over a set of Boolean variables $\mathcal{V} = \text{var}(S)$ 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 given in Table 5.7, page 109. Truth tables work however only for binary operators.

Definition 6.2.5 – Satisfaction, falsification. The truth assignment σ *satisfies* the formula f if $\sigma(f) = 1$, it *falsifies* f otherwise, i.e. if $\sigma(f) = 0$.

As an illustration, see example 5.1, page 108.

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

Definition 6.2.7 – Semantics of Boolean formulas. 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 6.2.8 – Semantics of systems of Boolean equations. 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.

6.2.4 Properties

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

Definition 6.2.9 – Entailment and equivalence. Let f and g be two Boolean formulas built over a set of Boolean variables $\mathscr{V}$. Then,
- f *entails* g, which is denoted by $f \models g$ if any truth assignment over $\mathscr{V}$ 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 $\mathscr{V}$).

■ **Example 6.1** 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 6.2 – 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.

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

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

6.2.5 Substitutions

Substitutions play an important role in assessment algorithms.

Definition 6.2.10 – 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.

$$\mathrm{var}(f[V \leftarrow g]) \quad = \quad (\mathrm{var}(f) \setminus \{V\}) \cup \mathrm{var}(g) \tag{6.2}$$

where $\setminus$ stands for set difference.

The following property holds.

Property 6.3 – Substitutions of flow variables by their definition. Let S be a uniquely rooted, data-flow system of Boolean equations built over a set $\mathscr{V}$ of Boolean variables and let $V = f \in S$. Finally, let S' be the system of Boolean equations built over $\mathscr{V} \setminus \{V\}$ obtained from S by substituting V for f, i.e.

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

Then, $S \equiv S'$.

By applying property 6.3 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, see exercise 6.5. This 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.

6.2.6 Additional Connectives

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

Definition 6.2.11 – Additional connectives. Let f, g and h be two Boolean formulas built over a set of Boolean variables $\mathscr{V}$. 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
$f \overline{\wedge} g$	which is equivalent to	$\neg f \wedge g$	nand, also called Sheffer's stroke
$f \overline{\vee} g$	which is equivalent to	$\neg f \vee g$	nor, also called Quine's dagger
$f ? g : h$	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 6.2.12 – 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.

$$f_1 \vee \ldots \vee f_n \equiv \texttt{atleast}\ 1\ (f_1, \ldots, f_n) \tag{6.3}$$

$$f_1 \wedge \ldots \wedge f_n \equiv \texttt{atleast}\ n\ (f_1, \ldots, f_n) \tag{6.4}$$

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.

$$\texttt{atmost}\ k\ (f_1, \ldots, f_n) \equiv \neg\texttt{atleast}\ n-k\ (f_1, \ldots, f_n) \tag{6.5}$$

$$\texttt{atleast}\ k\ (f_1, \ldots, f_n) \equiv \neg\texttt{atmost}\ n-k\ (f_1, \ldots, f_n) \tag{6.6}$$

$$\texttt{cardinality}\ l, h\ (f_1, \ldots, f_n) \equiv \texttt{atleast}\ l\ (f_1, \ldots, f_n) \wedge \texttt{atmost}\ h\ (f_1, \ldots, f_n) \tag{6.7}$$

6.3 The S2ML+SBE Modeling Language

In S2ML+SBE, S2ML stands as before for System Structure Modeling Language and SBE for Systems of Boolean Equations. We have already presented the structured constructs gathered into S2ML in Chapter 4. This section focus thus on SBE constructs.

A S2ML+SBE model consists actually of the following categories of constructs.

- The S2ML constructs like blocks, classes, instances, clones and extends directives and so on.
- The SBE constructs that describe Boolean equations (logical constructs).
- The SBE constructs that describe the probability distributions associated with basic events (stochastic constructs).
- The SBE constructs that describe common cause failures (extra-logical constructs).

We shall review the SBE constructs in turn.

6.3.1 Logical Constructs

In S2ML+SBE declarations of elements obey the following (meta-)grammatical rule.

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

Where,

- `Type` is the type of the declared element, e.g. `gate` for intermediate events of fault trees;
- `Path` is the path of the declared element considered from the current container. Usually, this path consists a unique identifier when the element is declared for the first time, but may be a full path when the element is redeclared.
- Finally, `Term` is an expression that defines the elements, e.g. a Boolean formula in case of an intermediate event of a fault tree, or a stochastic expression encoding a probability distribution in case of a basic event.

As an illustration, consider the system of Boolean equations given in Figure 6.4. Figure 6.8 shows our first S2ML+SBE model, which relies on this system of equations.

```
1  gate PC_lost = V1_failed or PTs_lost;
2  gate PTs_lost = UPT_lost and LPT_lost;
3  gate UPT_lost = UP_failed or V2_failed;
4  gate UP_failed = P1_failed or P2_failed;
5  gate LPT_lost = P3_failed or V3_failed;
6  basic-event V1_failed = failureProbabilityPrimaryValve;
7  basic-event V2_failed = failureProbabilitySecondaryValve;
8  basic-event V3_failed = failureProbabilityValve;
9  basic-event P1_failed = exponential(failureRateSmallPump);
10 basic-event P2_failed = exponential(failureRateSmallPump);
11 basic-event P3_failed = exponential(failureRateLargePump);
12 parameter failureProbabilityPrimaryValve = 0.01;
13 parameter failureProbabilitySecondaryValve = 0.02;
14 parameter failureRateSmallPump = 5.67e-3;
15 parameter failureRateLargePump = 4.21e-5;
```

Figure 6.8: System of Boolean equations of Figure 6.4 at the S2ML+SBE syntax

The declarations of *intermediate events* are introduced by the keyword `gate`. The left member of the declaration is the identifier (more exactly the path) of the declared event. Its right member is a Boolean formula involving intermediate, basic and house events.

House events are a special type of intermediate events that can be defined only by constants (`true` or `false`). House events are used to configure the model. It is probably not a good modeling habit to use house events. Nevertheless, some industrial models make a rather extensive usage of this modeling technique, especially in the nuclear industry. This is the reason why they have been introduced as a separate syntactic category in S2ML+SBE. However, assessment algorithms consider them as intermediate events.

The declarations of *basic events* are introduced by the keyword `basic-event`. As for intermediate events (and house events and parameters), the left member of the declaration is the identifier (more exactly again the path) of the declared event. Its right member is a stochastic expression describing the failure model of the basic event, i.e. a stochastic expression involving possibly parameters.

The declarations of *parameters* are introduced by the keyword `parameter`. The right member of such declaration is a stochastic expression describing the value of the parameter. This stochastic

Table 6.1: Logical connectives implemented by S2ML+SBE

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.

expression may involve other parameters. Parameters can thus depend on other parameters. Similarly to logical equations, the system of stochastic equations defining basic events and parameters must be data-flow, i.e. a parameter cannot depend eventually on itself.

Table 6.1 reviews the logical connectives implemented by S2ML+SBE. The first column gives the keyword 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.

The EBNF grammar of the logical constructs of S2ML+SBE is given in Figure 6.9.

The model given in Figure 6.8 is "flat" in this sense that it does not involve any S2ML construct. To encode faithfully the reliability block diagram pictured in Figure 6.3, we do need these constructs.

Figures 6.10 and 6.11 show the corresponding model. Figure 6.10 shows the declarations of classes that describe the valves and the pumps. Figure 6.11 shows the block describing the system as a whole.

This model uses different keywords to declare intermediate and basic events, more in the "S2ML spirit". In S2ML+SBE,

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

It is however wise not to mix the two terminologies.

Note also that this model makes a rather extensive use of redeclarations. For instance, the flow variable `UpperTrain.input` is first declared in its parent block, then redeclared in the main block to plug it on the flow variable `S`. Similarly, the parameter `failureRate` is first declared in the base class `BasicBlock`, then redeclared in the derived classes `Valve`, `SmallPump` and `LargePump`. These redeclarations are mandatory to build the model by pieces.

6.3.2 Stochastic Expressions

Stochastic expressions are used to define probability distributions associated with basic events as well as to define values of parameters. S2ML+SBE provides a wide set of operators, including:

- The usual arithmetic operators "+", "-", "*" and "/";
- Some predefined functions like "`exp`", "`log`", "`pow`", and "`sqrt`";
- Parametric distributions like "`exponential`" and "`Weibull`";

```
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 6.9: EBNF grammar for S2ML+SBE declarations of systems of Boolean equations (logical constructs)

```
class BasicBlock
  state failed = 0.0;
  flow input = false;
  flow output = input and not failed;
end

class Valve
  extends BasicBlock;
  state failed = failureProbability;
  parameter failureProbability = 0.02;
end

class Pump
  extends BasicBlock;
  state failed = exponential(failureRate);
  parameter failureRate = 1.0e-3;
end
```

Figure 6.10: Classes describing valves and pumps of the pumping system

– Constructs to describe piece-wise and linear empirical distributions;
– Constructs to create random deviates, like "`uniform`", "`normal`" and "`lognormal`".

Here follows a few examples. In all these examples, we use parameters with explicit names. Of course, other names could be used as well as numerical values or complex expressions.

Point estimate probabilities can be given directly:

```
block PumpingSystem
  flow S = true;
  Valve V1
    parameter failureProbability = 0.01
    end
  block UpperTrain
    flow input = false;
    Pump P1
      parameter failureRate = 5.67e-3;
      end
    Pump P2
      parameter failureRate = 5.67e-3;
      end
    Valve V2;
    flow P1.input = input;
    flow P2.input = input;
    flow V2.input = P1.output and P2.output;
    flow output = V2.output;
  end
  block LowerTrain
    flow input = false;
    Pump P3
      parameter failureRate = 4.21e-5;
      end
    Valve V3;
    flow P3.input = input;
    flow V3.input = P3.output;
    flow output = V3.output;
  end
  flow UpperTrain.input = S;
  flow LowerTrain.input = S;
  flow output = UpperTrain.output or LowerTrain.output;
  flow failed = not output;
end
```

Figure 6.11: Block describing the pumping system as a whole, from the reliability block diagram pictured in Figure 6.3

```
basic-event pumpFailure = 0.05; // probability
```

An exponential distribution:

```
basic-event pumpFailure = exponential(1/meanTimeToFailure);
parameter meanTimeToFailure = 1.36e+4; // in h^-1
```

A Weibull distribution:

```
basic-event sensorFailure = Weibull(scaleFactor, shapeFactor);
parameter scaleFactor = 3.72e+4; // in h
parameter shapeFactor = 3;
```

A distribution to represent a repairable component:

```
basic-event pumpFailure = GLM(probabilityOnFailureDemand,
  failureRate, repairRate);
parameter probabilityOnFailureDemand = 0.02;
parameter failureRate = 1.36e-4; // in h^-1
parameter repairRate = 2.65e-1; // in h^-1
```

A distribution to represent a periodically tested component:

```
basic-event pumpFailure = periodic-test(failureRate,
  timeBetweenTests, dateFirstTest);
parameter failureRate = 1.36e-4; // in h^-1
parameter timeBetweenTests = 4380; // in h
parameter dateFirstTest = 2190; // in h
```

Parametric probability distributions are convenient. For some of them, it would be of course possible to write the corresponding expressions directly. e.g.

```
basic-event pumpFailure = 1 - exp(-1/meanTimeToFailure * mission-time);
parameter meanTimeToFailure = 1.36e+4; // in h^-1
```

The special construct `mission-time` returns the mission time chosen for the calculations. It is used implicitly in parametric and empirical distributions.

The reader is invited to consult the XFTA Book (Rauzy 2020) for a complete description of stochastic expressions in S2ML+SBE.

6.3.3 Extra-Logical Constructs

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, S2ML+SBE provides a way to encode them.

The current version of XFTA implements mostly *common cause failure* groups as there are workarounds for the other extra-logical constructs. One of the basic assumptions of the fault tree and reliability block diagram techniques is that basic events are statistically independent. 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 C) are the following:

- The intrinsic failures of the calculators `A`, `B` and `C`. In the sequel, we shall denote them respectively `Ai`, `Bi` and `Ci`.
- 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:

```
A = Ai ∨ AB ∨ AC ∨ ABC
B = Bi ∨ AB ∨ BC ∨ ABC
C = Ci ∨ AC ∨ BC ∨ ABC
```

This principle generalizes to any number of components. It raises however significant mathematical issues, see (Rauzy 2020).

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.

We shall develop this point further here. Again, the reader is invited to consult the XFTA Book (Rauzy 2020) for a complete description of these constructs.

6.4 Design Methodology

6.4.1 Existing Methodologies

The fault tree pictured in Figure 6.4 as the reliability block diagram pictured in Figure 6.5 are designed more or less intuitively, by looking at the description of the pumping system. If such an approach works relatively well for small systems, it is not systematic enough for larger one.

The literature contains a few guidelines to design fault trees and reliability block diagrams. For fault trees, these guidelines consist essentially in suggesting a recursive descendant approach:

- The top event is considered first. Its possible causes are analyzed, the corresponding intermediate events are created and connected to the top event by the suitable gate.
- Then, each of the intermediate events is considered in turn, as if it was a new top event.
- And so on until the suitable degree of decomposition is reached.

For each event, which more or less describes the possible fault of a sub-system or a component, one must consider:

- The intrinsic failures of the sub-system or the component;
- The external failures that induce a failure the sub-system or the component, e.g. the loss of power supply;
- The control problems that may hamper the good functioning of the sub-system or the component.

This approach has its merits. Nevertheless, models designed in this way tend to be very specific, i.e. very far from the specifications of the system. They tend also to be very "personal": two analysts end up in general with two quite different models.

6.4.2 The System Architecture Approach

To overcome the above difficulties, an innovative approach consists in using... system architecture frameworks. The Cube framework presented in Chapter 2 is indeed a good candidate.

As an illustration, let us apply it to our case study. We shall perform here only a light study, for pedagogical purposes.

Functional Architecture

Figure 6.12 shows a simplified functional architecture of the pumping circuit.

This functional architecture puts immediately the focus on the key elements we have to take into account for the probabilistic safety assessment:

- The role of the pumping circuit is to deliver a pumping capacity.
- This capacity is ensured by two redundant trains (the upper and the lower trains).

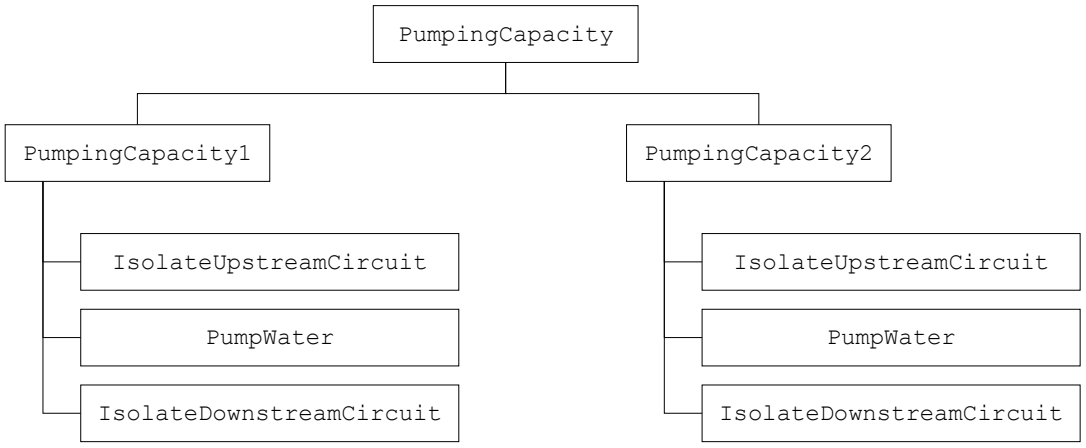


Figure 6.12: Simplified functional architecture of the pumping system pictured in Figure 6.1

– Each train must implement three main functions: indeed pumping the water, but also isolate the line from the upstream and downstream circuits.

Note here that applying system architecture principles helps to raise some good questions. For instance, does the pumping water function require some power supply? Is this power supply common to the two trains? Who is in charge of the isolation of the upstream and downstream circuits? And so on...

Use Cases and Operation Modes

The main objective of the system is to deliver pressurized water in case of an emergency. We can thus design a core use case from safety use cases will be derived.

Title: Core use case pumping system.

Preconditions: Because of a fire outburst in the plant, pressurized water is required to extinguish it.

Postconditions: The pumping system is in operation.

Trigger: Fire outburst in the plant.

Primary Actor: Operator.

Story: The scenario can be the following.

1. There is a fire outburst in the plant. Pressurized water is needed.
2. The operator opens valves `V1`, `V2` and `V3` that isolate the pumps from upstream and downstream water circuit.
3. The operator starts pumps `P1`, `P2` and `P3` to put in operation both upper and lower train.

Extensions: None for now.

This simple use case makes it possible to clarify an number of points.

First, the system is used on-demand, i.e. it has at least two very different operation modes: most of the time, it is in standby and from time to time, it is in operation. This means that procedures are applied to make it pass from standby to operation and vice-versa.

For this very reason, components (pumps and valves) can experience at least three different failure modes: dormant failures, i.e. failures while in standby, failures on demand or bad (re)configurations, and finally, failures while in operation.

In our case study, both pumps and valves can experience dormant failures. However, as the system is a safety system, they are probably regularly inspected and tested.

Pumps may fail on demand (any engine that must be start may). Valves are less much susceptible to fail on demand, but the operator, who is probably under stress in emergency circumstances, may

forget to open them, or not open them correctly.

Finally, it is reasonable to assume that only pumps can fail while in operation. It is actually hard to imagine a valve deciding suddenly to close by itself.

The above reasoning explains why failures of valves are associated with point estimate probabilities (which gather the probability of a dormant failure and the probability of a bad configuration by the operator), while failure pumps are associated with exponential distributions, neglecting dormant failures and failures on demand. This choice is indeed an approximation and as such, should be discussed.

In any case, dormant failures, failures on demand and bad (re)configuration induce the need for extensions to the above use case.

Physical Architecture

Studying the physical architecture of the pumping system may make us realize that things are not as simple as we may have thought at a first glance. Figure 6.13 shows a possible physical architecture of the pumping circuit.

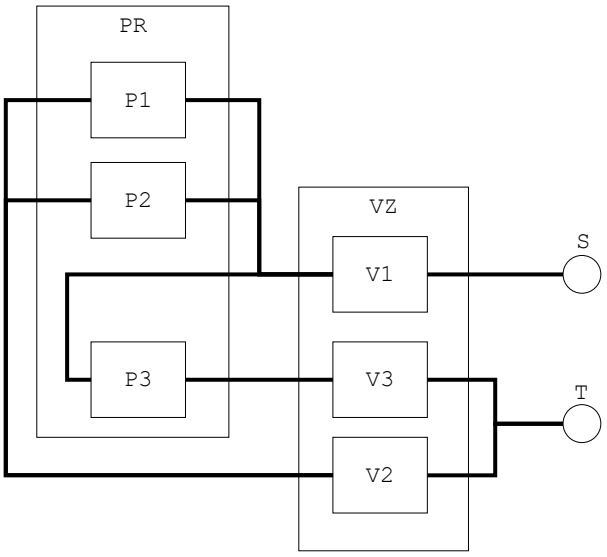


Figure 6.13: Possible physical architecture of the pumping system pictured in Figure 6.1

As incidentally said in the description of the case study, Section 6.1.1, the three pumps are located in the same room called `PR` on the figure. We may have found that the three valves are also located in the same zone, called `VZ` on the figure. Both facts are of importance is common cause failures are to be taken into account: in case of an emergency, are the valves reachable by the operator? Can the three pumps fail for the same reason (recall the Fukushima nuclear accident)? And so on...

Putting Things Together

To design the probabilistic safety assessment model, one can start from the functional architecture of the system, or from its physical architecture. To some extent, this is a matter of taste.

There is one important rule however: top events represent almost always a loss of capacity, i.e. a functional event. On the contrary, basic events represent always faults (failure modes) of physical components. This rule applies not only to fault trees, but also to reliability block diagrams, and even to discrete event and process algebra models.

As an illustration, assume that we decide to go for a fault tree for our case study. The construction works as follows.

1. Start from the functional architecture pictured in Figure 6.12.
2. Allocate functions/capacities onto physical components/parts.
3. Associate each internal node of the resulting decomposition a logical gate, telling what combination of its child nodes the node needs to work. For instance, the (global) pumping capacity is available if either the first pumping capacity is available, or the second one is. In turn, the first pumping capacity is available, if its all its three sub-capacities are. And so on...
4. Finally, take the dual of then constructed system of Boolean equations.

Figure 6.14 shows the resulting fault tree.

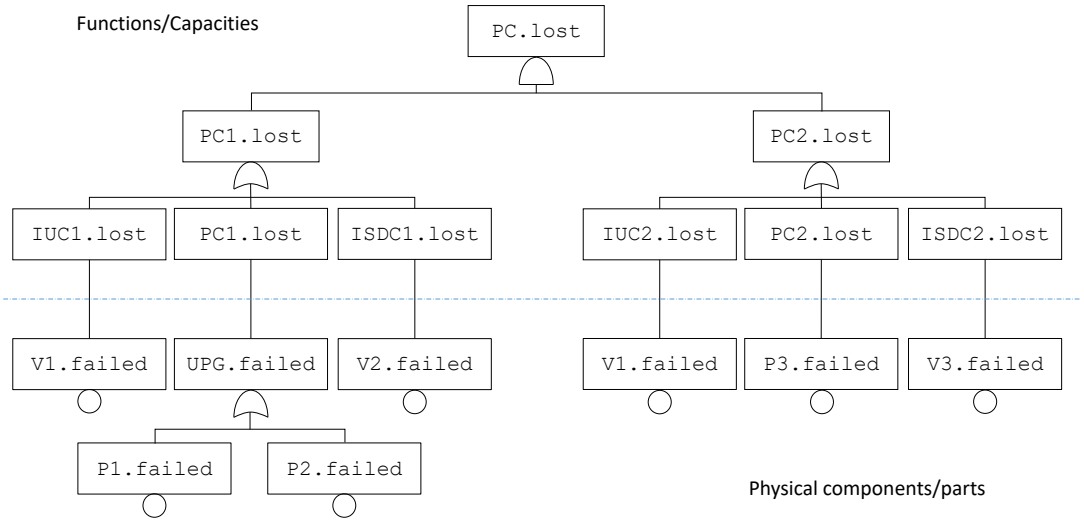


Figure 6.14: Fault tree resulting from the allocation of the functions/capacities onto physical components/parts

The upper part of the fault tree deals with loss of functions/capacities, while the lower part deals with failures of physical components/parts.

S2ML+SBE equations implementing this fault tree can be written directly, as we have done in Figure 6.8 for the intuitively designed fault tree pictured in Figure 6.2.

We can also take advantage of S2ML constructs to create a model that reflects both functional and physical architecture of the system under study. The idea is twofold: first, create a description of the functional architecture of the system, aside the description of its physical architecture. Second, use `embeds` directives to represent allocations.

Figure 6.15 gives a possible description of the functional architecture of the pumping system, assuming its physical architecture is described as in Figure 6.11.

Of course, in our example, this description of the functional architecture is to a large extent superfluous, as it comes in addition to the description of physical architecture from which the probabilistic safety analysis can already be performed.

Our objective here is to demonstrate a general methodology to design reliability models from system architecture models. With that respect, reflecting the latter into the former is of interest.

Note that the hierarchical decomposition rooted by the block `PumpingCapacity` embeds a fault tree structure, or more exactly is the dual of a fault tree structure, via the `available` flow variables.

Note also that our extended model could be use to study capacities that are not possible to study directly from the physical architecture.

```
block PumpingCapacity
  block PumpingCapacity1
    block IsolateUpstream
      embeds main.PumpingSystem.V1 as V1;
      flow available = not V1.failed;
    end
    block PumpWater
      embeds main.PumpingSystem.UpperTrain.P1 as P1;
      embeds main.PumpingSystem.UpperTrain.P2 as P2;
      flow available = not P1.failed and not P2.failed;
    end
    block IsolateDownstream
      embeds main.PumpingSystem.UpperTrain.V2 as V2;
      flow available = not V2.failed;
    end
    flow available = IsolateUpstream.available and PumpWater.available
      and IsolateDownstream.available;
  end
  block PumpingCapacity2
    block IsolateUpstream
      embeds main.PumpingSystem.V1 as V1;
      flow available = not V1.failed;
    end
    block PumpWater
      embeds main.PumpingSystem.LowerTrain.P3 as P3;
      flow available = not P3.failed;
    end
    block IsolateDownstream
      embeds main.PumpingSystem.LowerTrain.V3 as V3;
      flow available = not V3.failed;
    end
    flow available = IsolateUpstream.available and PumpWater.available
      and IsolateDownstream.available;
  end
  flow available = PumpingCapacity1.available
    or PumpingCapacity2.available;
  flow failed = not available;
end
```

Figure 6.15: Description of the functional architecture of the pumping system, assuming its physical architecture is described as in Figure 6.11

6.5 A Glimpse at Assessment Algorithms

Once the (uniquely rooted, data-flow) system of Boolean equations designed, two types of analyses can be performed:

- *Qualitative analyses*, that consist in extracting minimal cutsets, i.e. minimal sets of basic events that entail the top event.
- *Quantitative analyses*, that consist in calculating various probabilistic performance indicator, starting from the probability of the top event given the probabilities of basic events.

As already explained in Section 5.4.5, probabilistic performance indicators cannot be assessed directly from the system of Boolean equations. Prior to any probabilistic assessment, the structure function of the top event must be put in a normal form, and this normal form must be stored into a dedicated data structure.

6.5.1 Normal Forms, Binary Decision Trees and Binary Decision Diagrams

A Boolean formula is said in *normal form* if it obeys certain syntactic conditions. Two normal forms are used in the realm of reliability analysis: *sums of minimal cutsets* and *sums of disjoint products*. We shall give now an intuitive definition of these concepts. Formal definitions will come later in this chapter.

Literals, Products, Sums of Products

We shall start with some vocabulary and notations.

Definition 6.5.1 – 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 $\neg V$. V is called the *positive literal* and $\neg V$ the *negative literal* of V. V and $\neg V$ are *opposite literals* (note that $\neg\neg V \equiv V$).

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 6.5.2 – 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 6.4 – Minterms versus Truth Assignments. Let $\mathscr{V}$ be a set of Boolean variables, then minterms built over $\mathscr{V}$ one-to-one correspond with truth assignments of $\mathscr{V}$.

Proof: the minterm π one-to-one correspond with the truth 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 $\neg V \in \pi$.

Definition 6.5.3 – 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 6.4 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 indeed not true.

Sums of Minimal Cutsets

Sum of minimal cutsets are sum of positive products. Intuitively, a cutset is a set of basic events whose conjunction entails the top event. A cutset is minimal if one cannot remove any of its member

while preserving the property to be a cutset. Under certain conditions, a formula is equivalent to the sum of its minimal cutsets.

In our case study, the sum of minimal cutsets equivalent to the structure function of the top event `PC_lost` is the following.

```
   V1_failed
∨  P1_failed∧P3_failed
∨  P1_failed∧V3_failed
∨  P2_failed∧P3_failed
∨  P2_failed∧V3_failed
∨  V2_failed∧P3_failed
∨  V2_failed∧V3_failed
```

Now assume we want to encode a sum of minimal cutsets. We can indeed encoded as a list of minimal cutsets, each minimal cutset being itself encoded as a list of basic events. This type of encoding is called a *sparse matrix* in the software engineering literature (Cormen et al. 2001). But there are more compact and more suitable encodings. Binary decision trees are one of those. They are based on the following decomposition.

Property 6.5 – Decomposition of sums of minimal cutsets. Let φ be a sum of minimal cutsets and let V be a variable showing up in φ. Then, the following equality holds.

$$\varphi \;=\; V \odot \bigvee_{V \wedge \pi \in \varphi} \pi \;\vee \bigvee_{\pi \in \varphi, V \notin \pi} \pi$$

Where, for any variable V and sum of products ψ, $V \odot \psi$ is defined a follows.

$$V \odot \psi \;\stackrel{def}{=}\; \bigvee_{\pi \in \psi} V \wedge \pi$$

This decomposition is nothing but the declension of the Shannon decomposition (property 5.1, page 114) to sums of minimal cutsets. For this very reason, we can use binary decision trees, introduced Section 5.4.5, to encode sums of minimal cutsets.

Figure 6.16 shows a *binary decision tree* encoding the minimal cutsets of the top event of the models we designed for the pumping system.

A *binary decision tree* that encodes a sum of minimal cutsets is a directed binary tree such that:

- Each node of the decision tree is labeled with a variable V and has two out-edges: A then out-edge pointing out the subtree that encodes the products that contain V, and a else out-edge pointing out the subtree that encodes the products that do not contain V.
- The leaves of the tree are labeled either by in which case they encode no product, or by an index of the product encoded by the branch. Leaves encoding products are chained, to make it possible to sort them, e.g. by decreasing order of probability.
- A variable occurs at most once in a branch from the root node to a leaf.

The ability to sort minimal cutsets encoded by a binary decision tree makes it possible to maintain the sum of the most probable minimal cutsets. When the maximum number of minimal cutsets is reached, a new minimal cutset is added to the sum only if its probability is higher than the probability of the minimal cutset with the lowest probability already encoded. In this case, the latter is removed from the binary decision tree.

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. Even medium size models can actually have huge numbers of minimal cutsets.

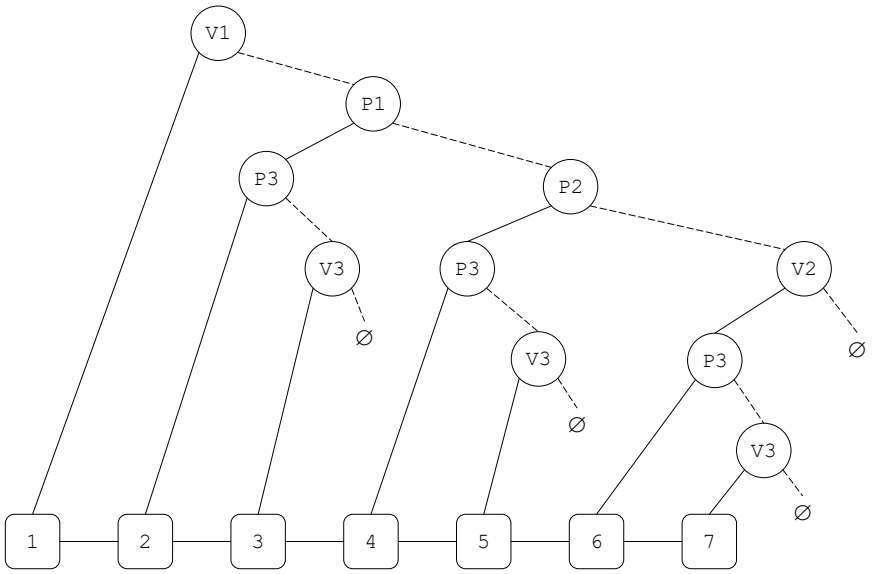


Figure 6.16: Binary decision tree encoding the minimal cutsets of the pumping system

A way to encode very large sum of minimal cutsets is to share not only the prefixes, as in binary decision trees, but also the suffixes. This gives raise to zero-suppressed binary decision diagrams.

Figure 6.17 shows a zero-suppressed binary decision diagram encoding the minimal cutsets of the top event of the models we designed for the pumping system.

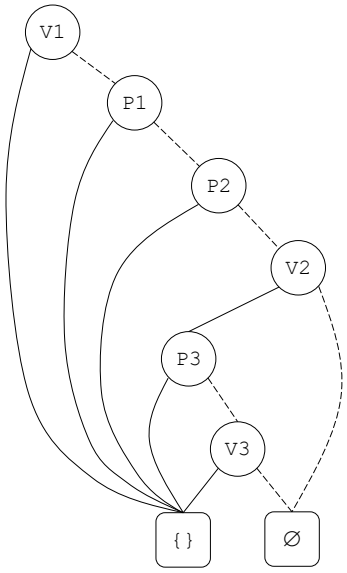


Figure 6.17: Zero-suppressed binary decision diagram encoding the minimal cutsets of the pumping system

Zero-suppressed binary decision diagrams can be much smaller than the sums of the minimal cutsets they encode. Their downside is that they do not make it possible to sort the minimal cutsets they encode.

Sums of Disjoint Products

A drawback of sums of minimal cutsets is that they do not allow the calculation of exact values of probabilistic performance indicators. The reason is that minimal cutsets are not exclusive one

another.

Sums of disjoint products on the contrary make it easy to calculate exact values for these indicators. The probability of a sum of disjoint products is actually simply the sum of the probabilities of its products.

In our case study, the sum of disjoint products equivalent to the structure function of the top event `PC_lost` is the following.

```
   V1_failed
∨  ¬V1_failed∧P1_failed∧P3_failed
∨  ¬V1_failed∧P1_failed∧¬P3_failed∧V3_failed
∨  ¬V1_failed∧¬P1_failed∧P2_failed∧P3_failed
∨  ¬V1_failed∧¬P1_failed∧P2_failed∧¬P3_failed∧V3_failed
∨  ¬V1_failed∧¬P1_failed∧¬P2_failed∧V2_failed∧P3_failed
∨  ¬V1_failed∧¬P1_failed∧¬P2_failed∧V2_failed∧¬P3_failed
   ∧V3_failed
```

As sums of minimal cutsets, sums of of disjoint products can be encoded by means of binary decision trees and binary decision diagrams. They are based on the following decomposition.

Property 6.6 – Decomposition of sums of disjoint products. Let φ be a sum of disjoint products and let V be a variable showing up in φ. Then, the following equality holds.

$$\varphi \quad = \quad V \wedge \varphi(V \leftarrow 1) \ \vee \ \varphi(V \leftarrow 0)$$

Figure 6.18 shows a binary decision diagram encoding the above sum of disjoint products.

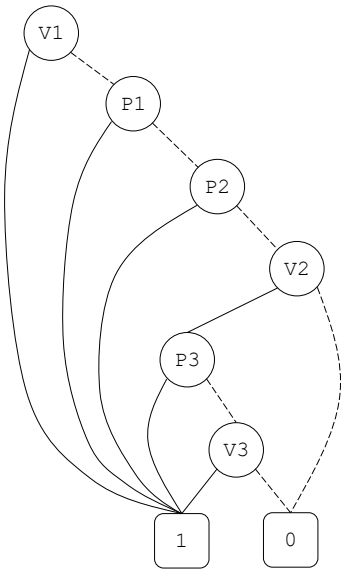


Figure 6.18: Binary decision diagram encoding the sum of disjoint products of the pumping system

6.5.2 Assessment Flows

As of today, two assessment flows are implemented in fault tree analysis tools, including XFTA: 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 6.19 shows these two assessment flows.

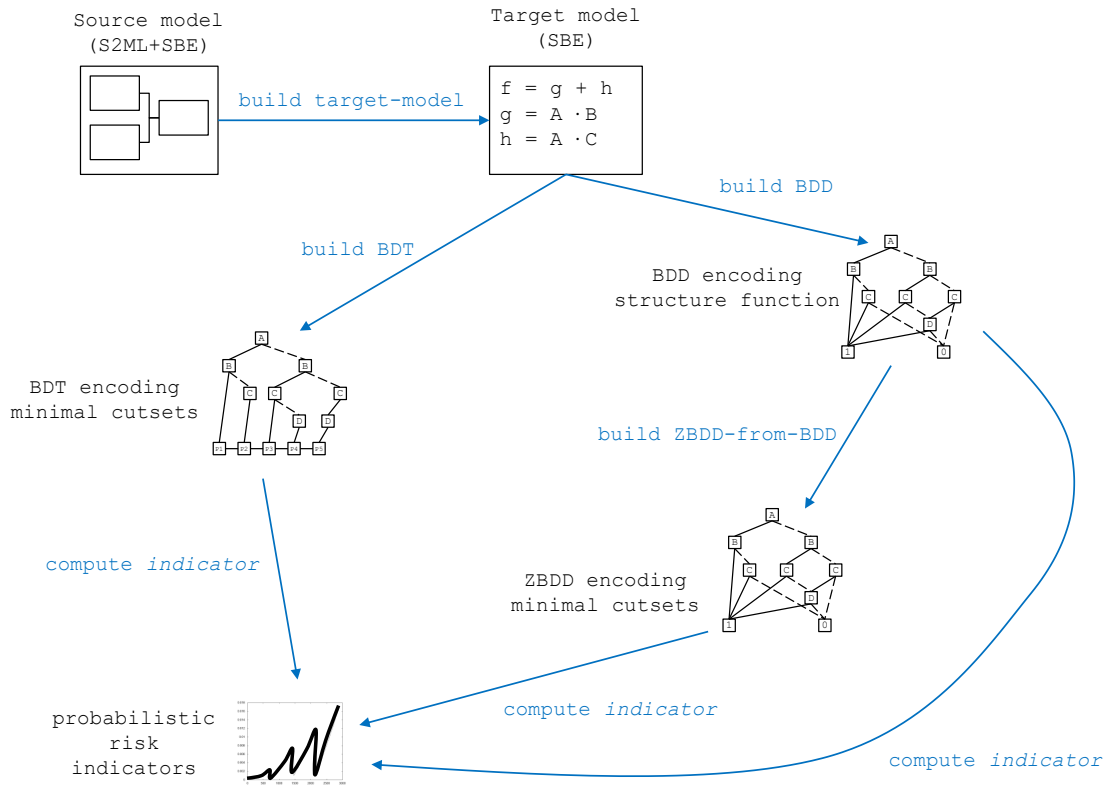


Figure 6.19: XFTA assessment flows

The first assessment flow consists in extracting the most significant minimal cutsets, storing them into a suitable data structure, namely *binary decision trees* in the case of XFTA, and printing them out 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, namely *binary decision diagrams* in the case of XFTA. Then, minimal cutsets are extracted from the binary decision diagram and stored in a data structure, namely a *zero-suppressed binary decision diagrams* in the case of XFTA. 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. Note also that the main difficulty is this process is to obtained the binary decision diagram. In most of the case, the construction of the zero-suppressed binary decision diagram is not really an issue.

As the ready has probably already guessed, if there are persistently two assessment flows, it is because each of them has advantages and drawbacks.

Even medium size models can actually 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, namely the most probable ones.

The minimal cutsets assessment flow has thus a major advantage: it works on very large models, has it is able to focus on the most important minimal cutsets discarding the others along the way. This advantage is also a drawback: the value of probabilistic performance indicators can only be

approximated, not only because only the most significant minimal cutsets are kept, but also because sums of minimal cutsets do not allow the calculation of exact values.

Binary decision diagrams encode sums of disjoint products, often in a very compact way. Sums of disjoint products make it possible to calculate the exact values of probabilistic performance indicators. Moreover, binary decision diagrams make these calculations extremely efficient (linear in the size of the diagram for most of them). Zero-Suppressed binary decision diagrams encode sums of minimal cutsets. The size of a zero-suppressed binary decision diagram is often much smaller that the size of the sum of minimal cutsets it encodes. The downside of this compactness is that minimal cutsets encoded by a zero-suppressed binary decision diagram cannot be sorted.

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.

6.5.3 Historical Notes

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 2003b; Rauzy 2012; Rauzy 2020). As of today, XFTA algorithm outperforms all other existing ones.

This approach has been the only existing one for many years. This is quite unfortunate because many authors, still today, ignore that an alternative approach exists. Would it be only a matter of algorithmic efficiency, this would not be a too serious issue. The problem is that many academic textbooks and industrial standards or guidelines define incorrectly probabilistic performance indicators via the minimal cutsets, taking substituting approximate *ad hoc* formulas for actual mathematical definitions.

Binary decision diagrams have been developed in the context of electronic circuits validation (Brace, Rudell, and Bryant 1990; Bryant 1986; Bryant 1992). Same thing for zero-suppressed binary decision diagrams (Minato 1993). Their use in the context of fault tree analysis has been proposed independently by Madre and Coudert (Coudert and Madre 1993) and the author (Rauzy 1993; Rauzy 2008a; Rauzy 2020). For this approach too, the XFTA implementation outperforms, as of today, all other existing ones.

The reader interested by an in-depth discussion on these algorithms can refer to XFTA book (Rauzy 2020), which gives references to advanced research articles.

6.5.4 Get It Applied with XFTA

S2ML+SBE are written into text files. Although this is not required, it is recommended to use the extension "`.sbe`" for files containing S2ML+SBE models. To load a model into XFTA and to perform calculations on this model, we need to write a XFTA script. As for models, scripts are written into text files. Again, although this is not required, it is recommended to use the extension "`.xfta`" for XFTA script files.

Scripts are sequences of commands to be executed by XFTA. There are commands to load models, to instantiate them, to build data structures encoding normal forms, to calculate probabilistic

performance indicators and to print out information. The XFTA book (Rauzy 2020) details XFTA commands. We shall see a number of these commands in the remaining of this chapter. However, if you are using the integrated modeling environment AltaRica Wizard, you do not need to remember the syntax of the commands: AltaRica Wizard helps you to build step by step your scripts.

Assume that the model give in Figure 6.8 is stored into the file "`PumpingSystemFT.sbe`". Figure 6.20 shows a script to assess our model.

```
load model "PumpingSystemFT.sbe";
build target-model;
build BDD PC_lost;
build ZBDD-from-BDD PC_lost;
compute probability PC_lost
    source-handle=BDD
    mission-time=[0, 730, 1460, 2190, 2920, 3650, 4380, 5110,
        5840, 6570, 7300, 8030, 8760, 8760]
    output="prb.csv"
    mode=write;
print minimal-cutsets PC_lost
    source-handle=ZBDD
    mission-time=8760
    print-minimal-cutset-order=true
    print-minimal-cutset-probability=true
    print-minimal-cutset-contribution=true
    output="mcs.csv"
    mode=write;
```

Figure 6.20: A script to assess the model given in Figure 6.8

This script consists of six commands. 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 "`PumpingSystemFT.sbe`".

The second command builds the target model. As S2ML+SBE is an object-oriented language, 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 builds the binary decision diagram (BDD) encoding the structure function associated with the top event `PC_lost`. Data structures such a binary decision diagrams, zero-suppressed binary decision diagrams and binary decision trees are managed via handles. A *handle* consists, among other data, of a name and a reference to the data structure. In the command, the name is set implicitly to `BDD`

The fourth command extracts minimal cutsets from this BDD and encodes them into as zero-suppressed binary decision diagram (ZBDD). This command involves two data structures, therefore two handles: the source binary decision diagrama, here `BDD`, and the target binary decision diagram, managed via a handle called by default `ZBDD`.

The fifth command computes the probability of the top event at different mission-time, 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. This command involves the following options:

- `source-handle`, here set to `BDD`, that specifies the data structure from which the probabilities are computed.
- `mission-time` that specifies the list of mission times at which the probability is computed.
- `output` that specifies in which file the calculated probabilities is printed out, here "`prb.tsv`".
- `mode` that specifies whether, if the file already exists, its content is erased (value `write`), or if the calculated probabilities are appended to it (value `append`).

In our case, after the execution of the script, the file "`prb.tsv`" is as showed in Figure 6.21.

```
variable    PC_lost
source-handle   BDD
quantification-method   Pr
time    Q
0   0.010396
4   0.0112669
8   0.0121131
12  0.0129356
16  0.013735
20  0.0145122
24  0.0152677
28  0.0160024
32  0.0167169
36  0.0174119
40  0.018088
44  0.0187458
48  0.019386
52  0.0200091
56  0.0206157
60  0.0212064
64  0.0217818
68  0.0223422
72  0.0228883
```

Figure 6.21: Probabilities of the top event, as calculated by the script given in Figure 6.20

The probability of top event `top` calculated at time $t = 72h$ is thus 2.29×10^{-2}.

The sixth and last command prints out the minimal cutsets into the TSV file "`mcs.tsv`". This command involves the following options:

- `source-handle`, here set to `ZBDD`, that specifies the data structure from which the minimal cutsets are printed out.
- `mission-time` that specifies the mission time at which the probability of minimal cutsets is computed.
- `print-minimal-cutset-probability` that specifies to print the probabilities of minimal cutsets.
- `print-minimal-cutset-contribution` that specifies to print the contributions of minimal cutsets, i.e. its probability divided by the sum of the probabilities of the minimal cutsets.
- `output` that specifies in which file the calculated probabilities is printed out, here "`mcs.tsv`".
- `mode` is a described previously.

After the execution of the script, the file "`mcs.tsv`" is as showed in Figure 6.22.

```
variable    PC_lost
source-handle   ZBDD
order   probability contribution    minimal-cutsets
time    72
rare-event-approximation    0.0258967
1   0.01    0.38615 V1_failed
2   0.00101446  0.0391734   P1_failed   P3_failed
2   0.00670361  0.25886 P1_failed   V3_failed
2   0.00101446  0.0391734   P2_failed   P3_failed
2   0.00670361  0.25886 P2_failed   V3_failed
2   6.05322e-05 0.00233745  V2_failed   P3_failed
2   0.0004  0.015446    V2_failed   V3_failed
```

Figure 6.22: Minimal cutsets, as extracted and printed out by the script given in Figure 6.20

6.6 Prime Implicants and Minimal Cutsets

This section aims at providing a formal definition of the notion of minimal cutsets, which we have postponed so far. In logic as well as in electronic circuit theory, the idea of minimal solution is captured by the notion of prime implicants. In most of the practical cases, the two notions coincide. They differ actually on non-coherent models, which are seldom used in reliability engineering. We shall thus introduce the notion of coherence, then the one of prime implicants and eventually the one of minimal cutsets. Finally, we shall present the XFTA commands for build data structures encoding minimal cutsets, and to display minimal cutsets from these data structures.

6.6.1 Coherence

In Boolean reliability models, basic events (state variables) represent fault of physical components, human errors and so on.

A priori, the more there are such faults and human errors, the more likely the system as a whole is failed (or its main capacity lost). 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 realized (the more likely the root variable takes the value true).

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

Definition 6.6.1 – Coherence. Let f be a Boolean formula built over a set of Boolean variables $\mathscr{V}$. Moreover let V be a variable of $\mathrm{var}(f)$.

Then f is *coherent* if for two variable assignments σ and σ' of $\mathscr{V}$ such that $\sigma(V) = 0$, $\sigma'(V) = 1$ and $\sigma(W) = \sigma'(W)$ for all variables $W \neq V$, the following implication holds.

$$\sigma(f) = 1 \;\rightarrow\; \sigma'(f) = 1$$

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'.

The following property is easily verifiable.

Property 6.7 – Syntactic coherence. Let f be a Boolean formula built over the two constants 0 and 1, a set of variables $\mathscr{V}$, but only using connectives $\vee$, $\wedge$ and k-out-of-n. Then f is coherent.

By Property 6.7, all models we have seen so far in this chapter are coherent.

It is possible to write coherent formulas using the other connectives than those of Property 6.7, but this must be done cautiously.

Here follows another example of coherent formula and an example of non-coherent formula

that we shall use throughout this section.

■ **Example 6.2** Let $f = (A \wedge B) \vee C$ and $g = (A \wedge B) \vee (\neg A \wedge C)$ built over $\{A, B, C\}$. Then,
- f is coherent by Property 6.7.
- g is non-coherent as the minterm $\sigma = \neg A \wedge \neg B \wedge C$ satisfies g, while the minterm $\sigma' = A \wedge \neg B \wedge C$ falsifies g.

■

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. One of the algorithmic approach of XFTA relies on the extraction of minimal cutsets of the model. In this approach, 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 source model is coherent or not. Consequently, non coherent models can be used, but in very controlled way.

6.6.2 Prime Implicants

In logic as well as in electronic circuit theory, the idea of minimal solution of a formula is captured via the notion of prime implicants.

> **Definition 6.6.2 – 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 6.3** Consider again the formulas $f = (A \wedge B) \vee C$ and $g = (A \wedge B) \vee (\neg A \wedge C)$ built over $\{A, B, C\}$. Then,

$$\begin{aligned} \mathrm{PI}(f) &= \{A \wedge B, C\} \\ \mathrm{PI}(g) &= \{A \wedge B, \neg A \wedge C, B \wedge C\} \end{aligned}$$

The product $B \wedge C$ is a prime implicant of g because whatever the value of A, g is satisfied: if A is true, then $A \wedge B$ is true and consequently g is true; now, if A is false, then $\neg A \wedge C$ is true and consequently g is true. ■

In reliability engineering, negations are 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 this context.

The notion of minimal cutsets that we shall introduce now differs from the one of prime implicants.

6.6.3 Minimal Cutsets

Minimal cutsets are 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 6.6.3 – 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.

$$\lfloor\pi\rfloor_{\mathscr{V}} \stackrel{def}{=} \pi \cup \{\neg V; V \in \mathscr{V} \setminus \mathrm{var}(\pi)\}$$

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.

Definition 6.6.4 – 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 $\mathrm{MCS}(f)$.

As for prime implicants, the above definition generalizes directly to uniquely rooted, data-flow system of Boolean equations.

■ **Example 6.4** Consider again the functions $f = (A \wedge B) \vee C$ and $g = (A \wedge B) \vee (\neg A \wedge C)$ of example 6.3. Then,

$$\begin{aligned} \mathrm{MCS}(f) &= \{A \wedge B, C\} \\ \mathrm{MCS}(g) &= \{A \wedge B, C\} \end{aligned}$$

■

To understand the relationship between prime implicants and minimal cutsets, we have to introduce the notion of degradation order.

6.6.4 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 $\neg V$.

Hence the idea of introducing a degradation order among minterms.

Definition 6.6.5 – 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 6.23 shows such a diagram for minterms built over $\{A, B, C\}$. Negative literals are denoted with a bar over the name of the variable. The least minterm (fully negative) is represented at the bottom, the biggest one at (fully positive) at the top.

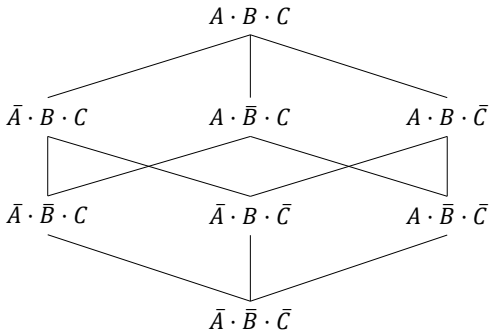


Figure 6.23: Hasse diagram for the lattice of minterms built over $\{A, B, C\}$.

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 6.8 – Minterms of Coherent Models. Let f be a Boolean formula built over a finite set of variables $\mathscr{V}$. Then f is coherent if there is no two minterms π and ρ built over $\mathscr{V}$ such that:
- $\pi \sqsubset \rho$.
- $\pi \not\models f$ while $\rho \models f$.

■ **Example 6.5** Consider again the functions $f = (A \wedge B) \vee C$ and $g = (A \wedge B) \vee (\neg A \wedge C)$ of example 6.3. It is easy to verify that:
- f is coherent.
- g is not as the minterm $\neg A \wedge \neg B \wedge C$ satisfies g, the minterm $A \wedge \neg B \wedge C$ does not satisfy g, while $\neg A \wedge \neg B \wedge C \sqsubset A \wedge \neg B \wedge C$.

■

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 6.6.6 – Coherent Hull. Let f be a Boolean formula built over a finite set of variables $\mathscr{V}$. The *coherent hull* of f is the sum of minterms Φ built over $\mathscr{V}$ 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 $\mathscr{V}$ such that $\rho \sqsubset \pi$ and ρ satisfies f.

The coherent hull of a formula f is denoted $[\![f]\!]$.

■ **Example 6.6** Consider again the functions $f = (A \wedge B) \vee C$ and $g = (A \wedge B) \vee (\neg A \wedge C)$ of example 6.3. We have:

$$\begin{aligned} [\![f]\!] &= A \wedge B \wedge C \vee A \wedge B \wedge \neg C \vee A \wedge \neg B \wedge C \vee \neg A \wedge B \wedge C \vee \neg A \wedge \neg B \wedge C \\ [\![g]\!] &= A \wedge B \wedge C \vee A \wedge B \wedge \neg C \vee \underline{A \wedge \neg B \wedge C} \vee \neg A \wedge B \wedge C \vee \neg A \wedge \neg B \wedge C \end{aligned}$$

We underlined the (only) minterm that has been added to get $[\![g]\!]$. ■

The following property follows immediately from the definitions.

Property 6.9 – Coherent Hull of Formulas. Let f be a Boolean formula built over a finite set of variables $\mathscr{V}$. Then f is coherent if and only if $f \equiv [\![f]\!]$.

We can now state the central theorem of fault tree assessment.

Theorem 6.10 – Minimal Cutsets versus Prime Implicants. Let f be a Boolean formula built over a finite set of variables $\mathscr{V}$. Then,

$$\mathrm{MCS}(f) \quad = \quad \mathrm{PI}(\llbracket f \rrbracket)$$

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 $\llbracket f \rrbracket$ neither, because if it was all the minterms containing it would satisfy $\llbracket f \rrbracket$. 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 $\llbracket f \rrbracket$, 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.

6.6.5 Get It Applied With XFTA

There are two main commands to extract minimal cutsets in XFTA. The first one corresponds to the sum of minimal cutsets approach, the second one the sum of disjoint products approach. We shall review in turn the main forms of these commands, as they can be called via AltaRica Wizard. The reader interested by a detailed description should refer to the XFTA book.

The creation of a XFTA script via AltaRica Wizard is performed by filling successive forms. Each form corresponds to a specific calculation. The first form "`Algorithm`" is dedicated to the choice of the approach. It is the only mandatory one.

Sum Of Disjoint Products Approach

Extracting the minimal cutsets using the sum of disjoint products approach is performed into two steps: first, the binary decision diagram encoding the structure function of the top event is built; second, the zero-suppressed binary decision diagram encoding the minimal cutsets is built from this binary decision diagram.

The first step is implemented by means of the command `build BDD`:

```
build BDD Top;
```

Then second step is implemented by the command `build ZBDD-from-BDD`:

```
build ZBDD-from-BDD Top;
```

We gave above the base forms of these commands. For a detailed description, the reader should refer to the XFTA book.

In AltaRica Wizard, the choice of the algorithm `BDD+ZBDD` puts in place these two commands.

Sum Of Minimal Cutsets Approach

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 AltaRica Wizard, the choice of the algorithm `BDT` puts in place this command.

The extraction of minimal cutsets (using this algorithm) is usually performed for a given cutoff. The command `build BDT` has thus several options making it possible to define the cutoff. Table 6.2 summarizes these options, and gives their types and default values.

Table 6.2: Options of the command `build BDT`

Option	Type	Default value
`mission-time`	Positive real	0.0
`maximum-number`	Positive integer	1,000,000
`maximum-order`	Positive integer	100
`minimum-probability`	Real in $[0,1]$	0.0

The cutoff with which the extraction of minimal cutsets is performed is specified by means of the following options.

- The mission time at which probabilities of minimal cutsets calculated, set via the option `mission-time`;
- The maximum number of extracted cutsets, set via the option `maximum-number`;
- The maximum order, i.e. the maximum number of variables, 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 value of these options are given with the command, e.g.

```
build BDT Top minimum-probability=1.0e-6 mission-time=8760;
```

Printing Minimal Cutsets

The command `print minimal-cutsets` makes it possible to print out minimal cutsets. In AltaRica wizard, the form "`Minimal Cutsets`" is dedicated to this command. (Some of) its options are given in Table 6.3.

Table 6.3: Options of the command `print minimal-cutsets`

Option	Type	Default value
`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`

The rank of the minimal cutset makes only sense if minimal cutsets are sorted, i.e. extracted with the sum of minimal cutsets approach.

The order of the minimal cutset is its number of elements.

The contribution of the minimal cutset is the ratio of its probability and the sum of the probability of all minimal cutsets. Probabilities are calculated at a given mission time, set by the option `mission-time`.

Minimal cutsets are printed out in a file, set by the option. The option `mode` tells what to do with the data contained into the file, if it was already existing.

Figure 6.22 shows minimal cutsets of the pumping system case study. These minimal cutsets are obtained by means of the sum of disjoint products approach. The order, the probability and the contribution is printed in addition to the variables of the minimal cutsets.

Figure 6.24 shows the minimal cutsets of the same case study, obtained by means of the sum of minimal cutsets approach with a probability cutoff set to 0.001.

```
1  variable    PC_lost
2  source-handle   BDT
3  rank    order   probability contribution    minimal-cutsets
4  time    72
5  rare-event-approximation    0.0254362
6  1   1   0.01    0.38615 V1_failed
7  2   2   0.00670361  0.25886 P1_failed   V3_failed
8  3   2   0.00670361  0.25886 P2_failed   V3_failed
9  4   2   0.00101446  0.0391734   P2_failed   P3_failed
10 5   2   0.00101446  0.0391734   P1_failed   P3_failed
```

Figure 6.24: Minimal cutsets extracted with the sum of minimal cutsets approach and a 0.001 probability cutoff

The ranks, the orders, the probabilities and the contributions of minimal cutsets are printed out. Minimal cutsets are sorted by decreasing probability order.

The probability cutoff discards the two minimal cutsets `V2_failed`$\wedge$`P3_failed` and `V2_failed`$\wedge$`V3_failed`.

6.7 Top-Event Probability

The top event probability is the main probabilistic indicator that can be assessed from the probabilities of basic events and the structure function associated with the top event.

If the sum of disjoint products approach is used, the structure function is encoded by means of a binary decision diagram. The Shannon decomposition (property 5.1, page 114) can then be used to calculate the exact value of the top event probability, and more generally of most of the probabilistic performance indicators.

If the sum of minimal cutsets approach is used, the value of the top event probability can only be approximated. XFTA implements three different approximation methods, as we shall see now.

6.7.1 Approximation Methods

The exact probability of a minimal cutset is simply the product of the probabilities of its variables. The probability of a sum of minimal cutsets is however more difficult to assess, as two cutsets 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 6.7.1 – Sylvester-Poincaré Development. Let $\varphi = \pi_1 \vee \cdots \vee \pi_n$ be a sum of products built over a set of variables $\mathscr{V}$. Then the probability of φ can be calculated by means of the following equation.

$$p(\varphi) \stackrel{def}{=} \sum_{1 \le i \le n} p(\pi_i) - \sum_{1 \le i_1 < i_2 \le n} p(\pi_{i_1} \wedge \pi_{i_2}) + \sum_{1 \le i_1 < i_2 < i_3 \le n} p(\pi_{i_1} \wedge \pi_{i_2} \wedge \pi_{i_3}) \cdots$$

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 6.7.2 – Rare Event Approximation. Let $\varphi = \pi_1 \vee \cdots \vee \pi_n$ be a sum of products built over a set of variables $\mathscr{V}$. The rare event approximation REA_φ of $p(\varphi)$ is defined as follows.

$$\mathrm{REA}_\varphi \stackrel{def}{=} \sum_{1 \leq i \leq n} p(\pi_i)$$

In XFTA, sums of minimal cutsets are encoded by means of *binary decision trees* and *zero-suppressed binary decision diagrams*. Let φ be a sum of positive products built over a set of variables $\mathscr{V}$ and let E be a variable showing up in φ. Then, we can decompose φ according to E as follows.

$$\varphi \equiv E \wedge \varphi(E=1) \vee \varphi(E=0)$$

In the above equation, we denoted by slight abuse:

- $\varphi(E=1)$ the sum of positive products $\sum_{E \wedge \pi \in \varphi} \pi$.
- $\varphi(E=0)$ the sum of positive products $\sum_{E \notin \pi \wedge \pi \in \varphi} \pi$.

The rare event approximation can thus be calculated recursively:

Property 6.11 – Rare Event Approximation (Decomposition). Let φ be a sum of positive products built over a set of variables $\mathscr{V}$ 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.

$$\mathrm{REA}_\varphi \equiv p(E) \times \mathrm{REA}_{\varphi(E=1)} + \mathrm{REA}_{\varphi(E=0)}$$

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 6.7.3 – Mincut Upper Bound. Let $\varphi = \pi_1 \vee \cdots \vee \pi_n$ 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.

$$\mathrm{MCUB}_\varphi \stackrel{def}{=} 1 - \prod_{1 \leq i \leq n} 1 - p(\pi_i)$$

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 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.

Table 6.4: 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`

Definition 6.7.4 – Pivotal Upper Bound. Let $\varphi = \pi_1 \vee \cdots \vee \pi_n$ be a sum of minimal cutsets 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:

$$\begin{aligned} \mathrm{PUB}_0 &\stackrel{def}{=} 0 \\ \mathrm{PUB}_1 &\stackrel{def}{=} 1 \end{aligned}$$

Otherwise, let E be a variable of $\mathrm{var}(\varphi)$. Then:

$$\mathrm{PUB}_\varphi \stackrel{def}{=} p(E) \times P_1 + P_0 - p(E) \times P_1 \times P_0$$

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 $0 \leq x,y \leq 1$, $0 \leq max(x,y) \leq x+y-x \times y \leq 1$.

It is often the case that this approximation is much more accurate than the two others. 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 that encodes φ.

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.

6.7.2 Get it Applied with XFTA

The command to compute the top probability of a variable of a model is as follows.

```
1 compute probability Top;
```

In AltaRica Wizard, the form "`Top event probability`" is dedicated to this command. Its options are summarized in Table 6.4.

We shall not describe all of these options here. The reader is invited to consult the XFTA book for a precise description.

A few key points:

– The probability of the top event can be calculated as different mission times;

– Some additional mission times can be automatically added to smooth the curves;
– Additional probabilistic indicators can be calculated via this command.

Figure 6.25 shows the probability distribution of the top event of the pumping system calculated from the binary decision diagram (exact probability) and from the binary decision tree with the pivotal upper bound approximation.

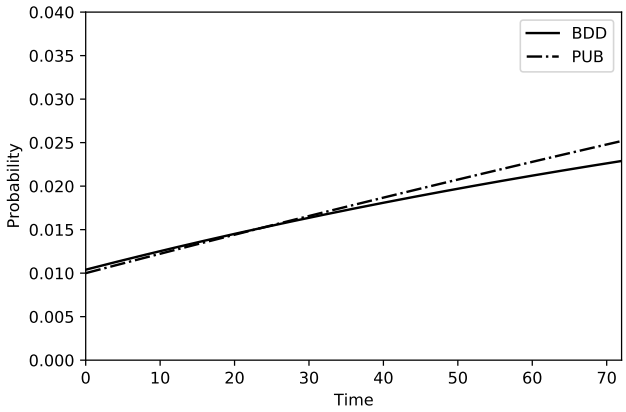


Figure 6.25: Probability distribution of the top event calculated with different approximation methods

In this example, the other approximations (rare event approximation and minimal upper bound) give values that are very similar, although slightly bigger, than the pivotal upper bound.

On the figure, we see that for low values of t, the pivotal upper bound is slightly optimistic. This is due to the cutoff at 0.01 that has been used to calculate the minimal cutsets.

Note that, to be fully consistent, the minimal custets should be recalculated at each considered date. On large models, this is not feasible within reasonable computation times.

For higher values of t, the pivotal upper bound is slightly pessimistic, although it does take into account all of the minimal cutsets. This is in general the case. The higher the probabilities of basic events and of the top event, the more pessimistic approximations.

6.8 Importance Measures

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 section, we assume that the top event T of the fault tree represents the failure of the system under study, that f is the structure function associated with T, and that each basic event E denotes the failure of a component of this system.

An importance measure is therefore an indicator $\mathrm{IM}(f,E)$. The reliability engineering literature distinguishes five main importance measures, see e.g. (Kumamoto and Henley 1996):

– The marginal importance factor $\mathrm{MIF}(f,E)$ often called Birnbaum importance factor.
– The critical importance factor $\mathrm{CIF}(f,E)$.
– The diagnostic importance factor $\mathrm{DIF}(f,E)$ also called Fussel-Vesely importance factor.
– The risk achievement worth $\mathrm{RAW}(f,E)$ also called risk increase factor.
– The risk reduction worth $\mathrm{RRW}(f,E)$ also called risk decrease factor.

In addition to these measures, we need to consider the two conditional probabilities $p(f \mid E)$ (the probability of f given E) and $p(f \mid \neg E)$ (the probability of f given $\neg E$).

We shall review them in turn.

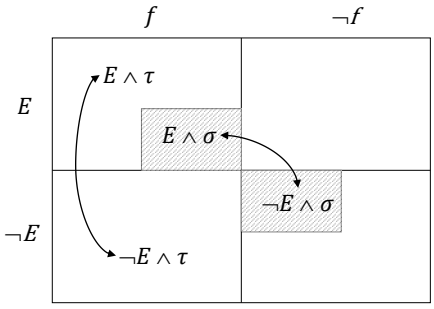


Figure 6.26: Graphical illustration of sets of minterms relevant for importance measures

6.8.1 Conditional Probabilities, Critical States and Minterms

The set of minterms built over var(f) can be divided into four subsets, as illustrated in Figure 6.26:

- The set of minterms that satisfy both f and E (upper left quadrant), i.e. that satisfy $f \wedge E$.
- The set of minterms that satisfy f and but not E (lower left quadrant), i.e. that satisfy $f \wedge \neg E$.
- The set of minterms that satisfy E and but not f (upper right quadrant), i.e. that satisfy $\neg f \wedge E$.
- The set of minterms that satisfy neither f nor E (lower right quadrant), i.e. that satisfy $\neg f \wedge \neg E$.

Now we have the central property of conditional probabilities.

Property 6.12 – Conditional Probability. Let f and g be two events over the same probability space. Then, the following equality holds.

$$p(f \wedge g) = p(f \mid g) \times p(g)$$

Applied to f and E, we have thus:

$$\begin{aligned} p(f \mid E) &= \frac{p(f \wedge E)}{p(E)} \\ p(f \mid \neg E) &= \frac{p(f \wedge E)}{1 - p(E)} \\ p(\neg f \mid E) &= \frac{p(\neg f \wedge E)}{p(E)} \\ p(\neg f \mid \neg E) &= \frac{p(\neg f \wedge E)}{1 - p(E)} \end{aligned}$$

Each quadrant (set of minterms) of Figure 6.26 represents thus a conditional probability.

Consider now a minterm $E \wedge \sigma$ of f. There are two cases: either $\neg E \wedge \sigma$ is also a minterm of f, or it is a minterm of $\neg f$. This remarks leads to the following notion of critical states.

Definition 6.8.1 – Critical States. Let f be a structure function and E be a basic event of f. Then the minterm $E \wedge \sigma$ is *critical* if the following conditions hold.

- $E \wedge \sigma \models f$,
- $\neg E \wedge \sigma \models \neg f$.

The set (sum) of critical states of f with respect to E is denoted CriticalStates(f, E).

The grayed rectangle of Figure 6.26 represents critical states.

Now, assume that f is coherent. In this case, if $\neg E \wedge \tau$ is a minterm of f, then so is $\neg E \wedge \tau$. In other words, non critical states $E \wedge \tau$ of f correspond one to one with minterms $\neg E \wedge \tau$ of f.

6.8.2 Definitions

We have now all what we need to define and to discuss importance measures.

Marginal Importance Factor

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 6.8.2 – Marginal Importance Factor. Let f be a structure function and E be a basic event of f. Then, the *marginal importance factor* of E in f, denoted by $\mathrm{MIF}(f,E)$, is defined as follows.

$$\mathrm{MIF}(f,E) \stackrel{def}{=} \frac{\partial p(f)}{\partial p(E)}$$

Using the Shannon decomposition (property 5.1), we can decompose $p(f)$ as follows.

$$p(f) = p(E) \times p(f \mid E) + (1 - p(E)) \times p(f \mid \neg E) \quad (6.8)$$
$$= p(E) \times (p(f \mid E) - p(f \mid \neg E)) + p(f \mid \neg E) \quad (6.9)$$

The following property follows from equality 6.9 and the calculation of derivatives of linear functions.

Property 6.13 – Marginal Importance Factor. Let f be a structure function and E be a basic event of f. Then,

$$\mathrm{MIF}(f,E) = p(f \mid E) - p(f \mid \neg E)$$

By definition, $\mathrm{MIF}(f,E)$ can be interpreted as the effect of a small variation of the probability of E on the probability of f, *mutatis mutandis*.

The following property holds.

Property 6.14 – Critical States. Let f be a coherent structure function and E be a basic event of f. Then,

$$p(\mathrm{CriticalStates}(f,E)) = p(E) \times \mathrm{MIF}(f,E)$$

There is thus a strong relationship between the probability of critical states and the marginal importance factor.

Critical Importance Factor

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. $\mathrm{MIF}(f,E)$ is the same whether $p(E)$ is low or high.

The critical importance factor, introduced by Lambert (Lambert 1975), aims at circumventing this problem.

Definition 6.8.3 – Critical Importance Factor. Let f be a structure function and E be a basic event of f. Then, the *critical importance factor* of E in f, denoted by $\mathrm{CIF}(f,E)$, is defined as follows.

$$\mathrm{CIF}(f,E) \stackrel{def}{=} \frac{p(E) \times \mathrm{MIF}(f,E)}{p(f)}$$

It is clear that $\mathrm{CIF}(f,E)$ depends on $p(E)$.

Note that, with respect to the ranking of components, the denominator $p(f)$ 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, $\text{CIF}(f,E)$ is simply the probability of critical states with respect to E.

$$\text{CIF}(f,E) \quad = \quad p(\text{CriticalStates}(f,E))$$

$\text{CIF}(f,E)$ characterizes thus the same minterms as $\text{MIF}(f,E)$.

Diagnostic Importance Factor

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 6.8.4 – Diagnostic Importance Factor. Let f be a structure function and E be a basic event of f. Then, the *diagnostic importance factor* of E in f, denoted by $\text{DIF}(f,E)$, is defined as follows.

$$\text{DIF}(f,E) \quad \stackrel{def}{=} \quad p(E \mid f)$$

The diagnostic importance factor has been introduced by Fussel (Fussel 1975).

Applying property 6.12, we can rewrite $\text{DIF}(f,E)$ as follows.

$$\text{DIF}(f,E) \quad = \quad \frac{p(E \wedge f)}{p(f)} \tag{6.10}$$

$$\quad = \quad \frac{p(E) \times p(f \mid E)}{p(f)} \tag{6.11}$$

The above equality means that, except for the normalization factor $p(f)$, $\text{DIF}(f,E)$ measures eventually the probability of $f \wedge E$, i.e. of the set of minterms represented by the upper left quadrant in Figure 6.26.

Risk Achievement Worth and Risk Reduction Worth

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 6.8.5 – Risk Achievement Worth and Risk Reduction Worth. Let f be a structure function and E be a basic event of f. Then, the *risk achievement worth* and the *risk reduction worth* of E in f, denoted respectively by $\text{RAW}(f,E)$ and $\text{RRW}(f,E)$, are defined as follows.

$$\text{RAW}(f,E) \quad \stackrel{def}{=} \quad \frac{p(f \mid E)}{p(f)}$$

$$\text{RRW}(f,E) \quad \stackrel{def}{=} \quad \frac{p(f \mid \neg E)}{p(f)}$$

$\text{RAW}(f,E)$ 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 $\text{RAW}(f,E)$ should be used with care, for it is rather rough. As $\text{DIF}(f,E)$, $\text{RAW}(f,E)$ measures eventually the probability of $f \wedge E$, i.e. of the set of minterms represented by the upper left rectangle in Figure 6.26.

Table 6.5: Options of the command `compute importance-measures`

Option	Type	Default value
`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`

RRW(f,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 RRW(f,E) is sometimes defined as $\frac{p(f)}{p(f|\neg E)}$, 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. RRW(f,E) measures eventually the probability of $f \wedge \neg E$, i.e. of the set of minterms represented by the lower left rectangle in Figure 6.26.

Note that, as for CIF(f,E) and DIF(f,E), the numerator $p(f)$ can be seen as a normalization factor.

6.8.3 Get it Applied with XFTA

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).

```
2 compute importance-measures Top;
```

Options of the command `compute importance-measures` are summarized in Table 6.5.

These options are available via the form `Importance Measures` of AltaRica Wizard.

Figure 6.27 shows the values of importance measures, obtained from the binary decision diagram at $t = 72$.

The marginal importance factor of the valve `V1` is close to 1 as this component is extremely critical. For the same reason, its risk reduction worth is also very high.

The marginal importance factors and the risk reduction worth of the pump `P3` and the valve `V3` are also high because a failure of any of these two components increase very significantly the probability of failure of the system as a whole (as they induce the loss of the lower train).

The critical importance factor of the valve `V3` is even higher than the one of the valve `V1`, because the probability that the former is not properly open is much higher than the probability that the latter is not properly open. This different in their respective probability explains also why the risk achievement worth of the valve `V3` is higher than the one of the valve `V1`.

The diagnostic importance factors of the pumps `P1` and `P2` on the one hand, and the valve `V3` on the other hand are high. The reason is indeed that the probabilities of failure of these components are quite high. Taken together, the minimal cutsets `P1_failed`$\wedge$`V3_failed` and `P2_failed`$\wedge$`V3_failed` contribute significantly to the overall risk, eventually even more than

```
variable PC_lost
source-handle BDD
quantification-method Pr
time Q
72 0.0228883
variable PC_lost
source-handle BDD
quantification-method Pr
variable Pr MIF CIF DIF RAW RRW
V1_failed 0.01 0.986982 0.431218 0.436905 43.6905 1.75814
P1_failed 0.335181 0.0148133 0.216929 0.479399 1.43027 1.27702
P2_failed 0.335181 0.0148133 0.216929 0.479399 1.43027 1.27702
V2_failed 0.02 0.0100491 0.00878106 0.0286054 1.43027 1.00886
P3_failed 0.00302661 0.549963 0.0727239 0.0755304 24.9554 1.07843
V3_failed 0.02 0.559488 0.488887 0.499109 24.9554 1.95651
```

Figure 6.27: Importance measures of basic events calculate from the binary decision diagram at $t = 72$

the failure of the valve `V1` alone. These four components are those to look at in priority to repair the system.

6.9 Further Readings

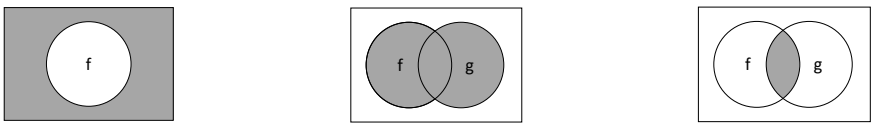
Fault trees and reliability block diagrams are introduced in nearly all the reliability engineering textbooks. Among these books, we can suggest those of Andrews and Moss (Andrews and Moss 2002), Kumamoto and Henley (Kumamoto and Henley 1996), and Signoret and Leroy (Signoret and Leroy 2021).

Regarding assessment algorithms, the XFTA book (Rauzy 2020) presents a state of the art, as of fall 2021. It is not fully complete but it gives references to the state of the art articles for the subjects it does not cover.

In this chapter, we covered some of the most important concepts of fault tree analysis. We did not cover however sensitivity analyses, approximation of reliability, and safety integrity levels. These are however important subjects that are covered in the XFTA book.

6.10 Exercises and Problems

Exercise 6.1 – Venn Diagrams. Truth tables of logical connectives can be graphically represented by means so-called *Venn diagrams*. Venn diagrams of connectives of $\neg$, $\vee$ and $\wedge$ are as follows.



Question 1. Create truth tables for connectives $f \Rightarrow g$, $f \Leftrightarrow g$, $f \oplus g$, $f \overline{\wedge} g$, $f \overline{\vee} g$ and $f\,?\,g : h$.

Question 2. Draw Venn diagrams for connectives $f \Rightarrow g$, $f \Leftrightarrow g$, $f \oplus g$, $f \overline{\wedge} g$, $f \overline{\vee} g$ and $f\,?\,g : h$.

■

Exercise 6.2 – Boolean Bases. A *base* of the propositional calculus is a set of connectives which is sufficient to represent any Boolean function.

Question 1. Show that $\{\neg, \vee, \wedge\}$ is a base of the propositional calculus.

Question 2. Show that it is not possible to remove any of these connectives while staying a base of the propositional calculus.

Question 3. Using the above results, show that $\{\oplus\}$ and $\{\overline{\wedge}\}$ are also bases of the propositional calculus.

■

Exercise 6.3 – Boolean Functions. This exercise looks at Boolean functions.

Question 1. Enumerate all possible Boolean functions from $\{0,1\}$ to $\{0,1\}$

Question 2. Same question with Boolean functions from $\{0,1\}^2$ to $\{0,1\}$

Question 3. More generally, how many Boolean functions from $\{0,1\}^n$ to $\{0,1\}$ are there?

■

Exercise 6.4 – Set Formulas. The set of set formulas over a set S of elements is the smallest set such that:

- The empty set $\emptyset$ is a set formula.
- If $e_1, \ldots e_k$ $k > 0$ are elements of O the $\{e_1, \ldots e_k\}$ is a set formula over O.
- If f, f_1,... f_1 are set formulas over O, then so are $f^{\complement}$, $f_1 \cup \ldots \cup f_n$, $f_1 \cap \ldots \cap f_n$ and (f).

Parentheses are used to avoid ambiguity.

Question 1. Assuming operators $\complement$, $\cup$ and $\cap$ have their usual meaning, show by structural induction that any set formula f over a set O can associated with a subset $[\![f]\!]$ of O.

Question 2. Let $P = \{p_1, \ldots, p_n\}$ be a set of n propositions. Define a morphism ϕ from $(O, \{\emptyset, O, \complement, \cup, \cap\})$ into $(P, \{0, 1, \neg, \vee, \wedge\})$. What can you say about these two algebraic structures?

■

Exercise 6.5 – Formulas versus Systems of Equations. Systems of Boolean equations make it possible to reflect the architecture of the system in the model by defining intermediate gate that correspond to failure of subsystems or to flows at certain points of the network of components.

Aside this fundamental advantage over pure formulas, they also make it possible to have compact representations of formulas that would otherwise be very large.

Question 1. Write the formula equivalent to the definition of T in the following system of Boolean equations.

$$\begin{aligned} T &= F \vee G \\ F &= A \wedge H \\ G &= B \wedge H \\ H &= C \vee D \end{aligned}$$

Question 2. Generalize the above example to get a parametric system of Boolean equations such that the formula corresponding to the top event is exponentially larger than the system itself.

■

Problem 6.6 – Overflow Protection System (intuitive analysis). Consider again the overflow protection system we studied extensively in Chapter 2. The process and instrumentation diagram for this system is pictured in Figure 2.1, page 21.

Question 1. Design a fault tree for this system, applying the intuitive method proposed in Section 6.1.2.

Question 2. Write down the corresponding system of Boolean equations.

Question 3. Design a reliability block diagram for this system, applying the intuitive method proposed in Section 6.1.2.

Question 4. Write down the corresponding system of Boolean equations.

Question 5. Rewrite and simplify this system of Boolean equations as done in Section 6.1.3.

Question 6. Verify that this latter system of Boolean equations is equivalent to the one of question Question 2..

■

Problem 6.7 – HIPPS (intuitive analysis). Consider again the high integrity pressure protection system studied in exercises of Chapter 2, case study 2 page 37.

Question 1. Design a fault tree for this system, applying the intuitive method proposed in Section 6.1.2.

Question 2. Write down the corresponding system of Boolean equations.

Question 3. Design a reliability block diagram for this system, applying the intuitive method proposed in Section 6.1.2.

Question 4. Write down the corresponding system of Boolean equations.

Question 5. Rewrite and simplify this system of Boolean equations as done in Section 6.1.3.

Question 6. Verify that this latter system of Boolean equations is equivalent to the one of question Question 2..

■

Exercise 6.8 – k-out-of-n expansion. Formulas involving only connectives $\wedge$ and $\vee$ can be written using only the `atleast` k connectives. Similarly, formulas involving connectives $\wedge$, $\vee$ and $\neg$ connectives can be rewritten using only the `cardinality` l,h, connectives. But how to do the reverse, i.e. to encode `atleast` $k\ (e_1,\ldots e_n)$, respectively `cardinality` $l,h\ (e_1,\ldots e_n)$, using only $\wedge$ and $\vee$ connectives, respectively $\wedge$, $\vee$ and $\neg$ connectives?

A naïve approach consists in rewriting `atleast` $k\ (e_1,\ldots e_n)$ as the disjunction of all products of k e_i's. Similarly, one could rewrite `cardinality` $l,h\ (e_1,\ldots e_n)$ as the disjunction of all products of p positive literals and q negative ones such that $p \geq l$ and $q \geq n-h$. The problem with this approach is indeed that the number of subsets of k elements of a set of n elements is the binomial $\binom{n}{k}$ which grows very fast with n and k.

Dynamic programming provides an elegant solution to this problem (Dutuit and Rauzy 2001).

It is based on the following equivalences.

$$\texttt{atleast}\ k\ (e_1,\dots e_n) \equiv \begin{array}{ll} & \texttt{atleast}\ k\ (e_1,\dots e_{n-1}) \\ \vee & e_n \wedge \texttt{atleast}\ k-1\ (e_1,\dots e_{n-1}) \end{array} \tag{6.12}$$

$$\texttt{cardinality}\ l,h\ (e_1,\dots e_n) \equiv \begin{array}{ll} & \neg e_n \wedge \texttt{cardinality}\ l,h\ (e_1,\dots e_{n-1}) \\ \vee & e_n \wedge \texttt{cardinality}\ l-1,h-1\ (e_1,\dots e_{n-1}) \end{array} \tag{6.13}$$

Question 1. Use equivalence to rewrite $top = \texttt{atleast}\ 4\ (e_1,\dots e_6)$ into an equivalent system of Boolean equations.

Question 2. Determine the asymptotic size of the system of Boolean equations encoding the equation $top = \texttt{atleast}\ k\ (e_1,\dots e_n)$ using the above scheme.

Question 3. Draw the binary decision diagram encoding $top = \texttt{atleast}\ 4\ (e_1,\dots e_6)$. What do you observe?

Question 4. Use equivalence to rewrite $top = \texttt{cardinality}\ 2,4\ (e_1,\dots e_6)$ into an equivalent system of Boolean equations.

Question 5. Determine the asymptotic size of the system of Boolean equations encoding the equation $top = \texttt{cardinality}\ l,h\ (e_1,\dots e_n)$ using the above scheme.

Question 6. Draw the binary decision diagram encoding $top = \texttt{cardinality}\ 2,4\ (e_1,\dots e_6)$. What do you observe?

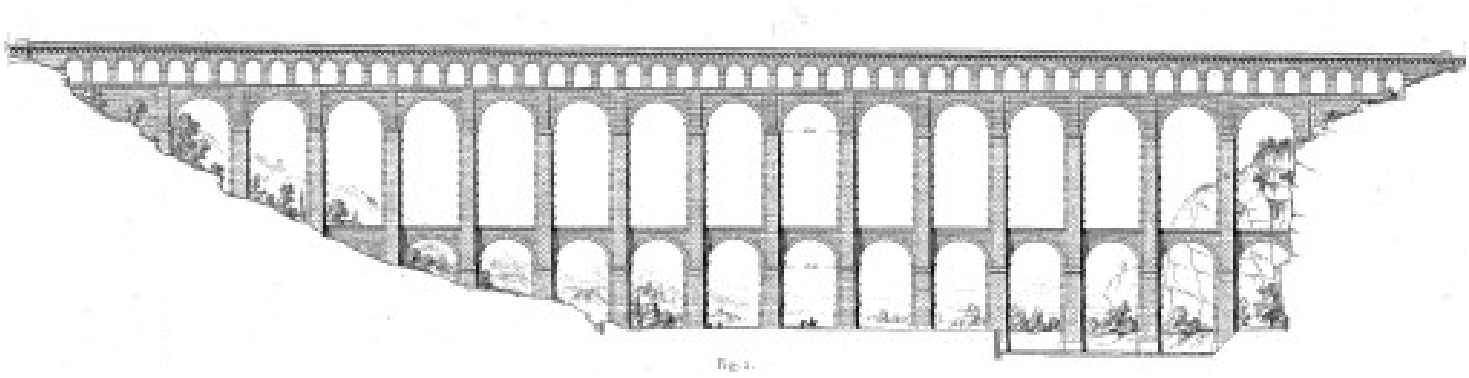
■

Exercise 6.9 – 4-out-of-6. Consider a 4-out-of-6 system, i.e. a system made of 6 components and that is failed if at least 4 out of the 6 components are failed. Assume moreover that the 6 components are identical and that their failures are exponentially distributed with a failure rate $\lambda = 5.00 \times 10^{-6}$.

Question 1. Use XFTA to calculate the exact probability of failure of the system at $t = 1000$, 2000,... 10000.

Question 2. Same question using the rare event approximation, the mincut upper bound and the pivotal upper bound. What do you observe?

■



7. Discrete Event Systems

Key Concepts
- Finite state automata
- Discrete event systems
- Guarded transition systems
- Expressions and instructions
- Executions, traces
- Reachable states, reachability graphs
- Data-flow versus looped models

This chapter introduces discrete event systems. Discrete event systems form a large class of mathematical frameworks that are used the dynamic behavior of natural, social, and technical systems. Beyond their differences, these frameworks share the same principle: they consider that the system under study changes of states under the occurrence of events and stays in the same state in between two consecutive events. The time elapsing between two events varies. It is in most of the case stochastic. Stochastic discrete event systems provide probably the most suitable tool to represent changes over the time of complex technical systems.

More specifically, this chapter presents guarded transition systems introduced by the author (Batteux, Prosvirnova, and Rauzy 2017a; Rauzy 2008b). Guarded transition systems, GTS for short, are at the core of the AltaRica 3.0 modeling language that is described by the equation:

$$\text{GTS} + \text{S2ML} \quad = \quad \text{AltaRica 3.0}$$

This chapter aims thus at presenting the mathematical and algorithmic framework of AltaRica 3.0. It is complemented by Chapter 8 that presents key modeling patterns.

The first three sections introduce the different constructs of the language by means of examples, in the same spirit as the introductory article (Batteux, Prosvirnova, and Rauzy 2019a). The reader is encouraged, once she or he gets a bit familiar with AltaRica 3.0, to read Section 7.5, before going back to examples. Section 7.5 presents the AltaRica interactive simulator that makes it possible to execute models, which makes it much easier to understand them.

7.1 Case Study

To illustrate our presentation, we shall consider throughout this chapter the gas production facility pictured in Figure 7.1. The gas coming from the well `W` is first separated from oil and water in the separation line `S`. Then, it is dehydrated in the dehydration line `D`. Then, it is compressed in the compression line `C`. Finally, it is sent in the pipeline `P`.

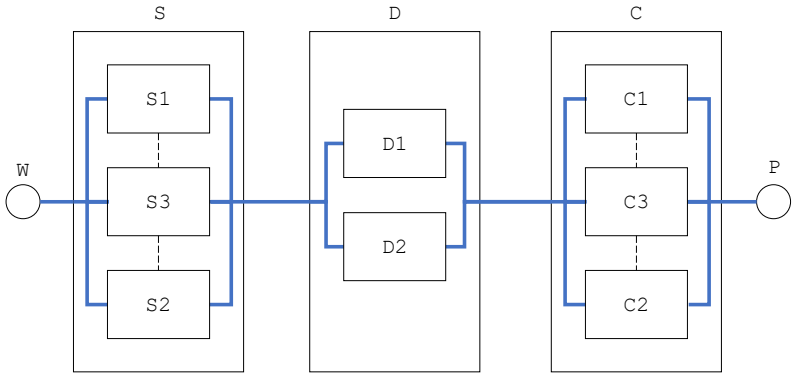


Figure 7.1: A gas production facility

The separation line `S` consists of three separators `S1`, `S2` and `S3`. The separator `S3` is in cold redundancy for the two others, i.e. if either `S1` or `S2` fails, `S3` is used instead. Each of these separators can fulfill 50% of the production.

The dehydration line `D` consists of two dehydrators `D1` and `D2` is hot redundancy. Each of these dehydrators can fulfill 85% of the production.

Finally, the compression line `C` consists of three compressors `C1`, `C2` and `C3`. The compressor `C3` is in cold redundancy for the two others. Each of these compressors can fulfill 60% of the production.

We shall assume, unless stated otherwise, that each unit of the system may fail and be repaired, and that they are as-good-as new after a repair. We shall assume moreover that their failures and repairs obey exponential distributions. Table 7.1 gives their respective failure and repair rates.

Table 7.1: Reliability data of the gas production facility

Unit	Failure rate	Repair rate
Separators `S1`, `S2` and `S3`	$8.50 \times 10^{-5}\ h^{-1}$	$2.50 \times 10^{-3}\ h^{-1}$
Dehydrators `D1` and `D2`	$3.00 \times 10^{-5}\ h^{-1}$	$3.00 \times 10^{-3}\ h^{-1}$
Compressors `C1`, `C2` and `C3`	$3.50 \times 10^{-5}\ h^{-1}$	$5.00 \times 10^{-3}\ h^{-1}$

7.2 Guarded Transition Systems

This section introduces guarded transition systems by means of examples. It presents the main syntactic constructs of the AltaRica 3.0 language. For a complete description, the reader can refer to language specification (Batteux, Prosvirnova, and Rauzy 2017b) available on the AltaRica Association website `www.altarica-association.com`.

7.2.1 States and Transitions

As said above, stochastic discrete event systems describe the evolution of the system under study as sequences of transitions between states. They can be seen and conveniently represented graphically

as stochastic state automata. They are subtle differences between the two formalisms, but we shall not enter into these details now.

To start with, let us thus consider the state automaton pictured in Figure 7.2 that can be used to represent units of our case study.

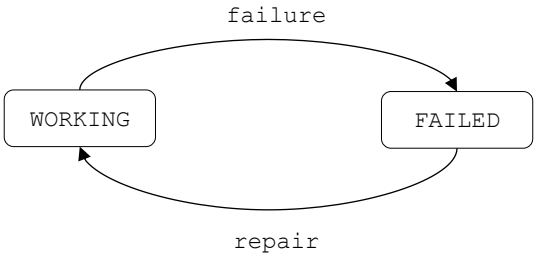


Figure 7.2: State automaton describing a repairable unit

This automaton is made of two states, `WORKING` and `FAILED` and two transitions `failure` and `repair`. We can moreover assume that these two transitions are exponentially distributed, with respective rates $1.00 \times 10^{-5}\ h^{-1}$ (failure rate) and $1.00 \times 10^{-3}\ h^{-1}$ (repair rate). In other words, this state automaton is a continuous time Markov chain (see Section 5.5.2).

The AltaRica code for the corresponding guarded transition system is given in Figure 7.3.

```
1  domain RepairableUnitState {WORKING, FAILED}
2
3  block RepairableUnit
4      RepairableUnitState _state(init = WORKING);
5      event failure(delay = exponential(1.0e-5));
6      event repair(delay = exponential(1.0e-3));
7      transition
8          failure: _state==WORKING -> _state := FAILED;
9          repair: _state==FAILED -> _state := WORKING;
10 end
```

Figure 7.3: AltaRica code for the state automaton of Figure 7.2

In guarded transition systems, states of components and systems are represented by means of *(state) variables*. Variables take their values into predefined *domains* such as Boolean, integers or real numbers as well as in user defined domains that finite sets of symbolic constants. The AltaRica code starts thus by defining such a domain, named `RepairableUnitState` and containing the two symbolic constants `WORKING` and `FAILED`.

Then comes the model for the repairable unit itself, which described by the block `RepairableUnit`.

This block starts by declaring a state variable `_state`, whose domain is `RepairableUnitState` and initial value is `WORKING`. Pairs (name, value) put in parentheses after a variable or an event declaration are called *attributes*. In this case, the attribute `init`, which introduces state variables and sets their initial value, has the value `WORKING`.

The block `RepairableUnit` then declares two *events* `failure` and `repair` and associates them with exponential delays, i.e. distributions, of respective rates 1.00×10^{-5} and 1.00×10^{-3}, by means of the attribute `delay`.

Finally, the block `RepairableUnit` declares the two *transitions*. Transitions of guarded transition systems are made of three elements:

– The event that labels the transition. The same event may label several transitions.

– A Boolean condition that tells when the transition is *enabled*, called the *guard* of the transition.
– An instruction that modifies the values of variables when the transition is *fired*, called the *action* of the transition.

The `failure` transition is thus enabled when the state variable `_state` has the value `WORKING`. Firing this transition assigns the value `FAILED` to `_state`. Similarly, the `repair` transition is enabled when `_state` has the value `FAILED`. Firing this transition assigns the value `WORKING` to `_state`.

Indeed, nothing prevents domains of variables to contain more than two values. We can consider for instance a component that first degrades, then fails, before being repaired, as shown in Figure 7.4.

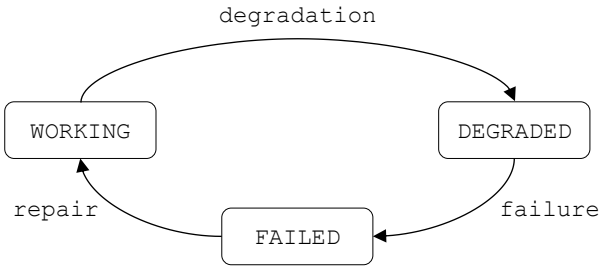


Figure 7.4: State automaton describing a degradable unit

The AltaRica code to encode this state automaton is given in Figure 7.5.

```
1  domain DegradableUnitState {WORKING, DEGRADED, FAILED}
2
3  block DegradableUnit
4      DegradableUnitState _state(init = WORKING);
5      event degradation(delay = Weibull(1.0e+4, 3));
6      event failure(delay = exponential(1.0e-2));
7      event repair(delay = exponential(1.0e-3));
8      transition
9          degradation: _state==WORKING -> _state := DEGRADED;
10         failure: _state==DEGRADED -> _state := FAILED;
11         repair: _state==FAILED -> _state := WORKING;
12 end
```

Figure 7.5: The AltaRica code for the state automaton of Figure 7.4

This code is essentially similar to the one of Figure 7.3. Note however that the `degradation` transition is associated here with a Weibull distribution of scale parameter 1.00×10^4 and shape parameter 3.00. Consequently, the model of Figure 7.5 is no longer a Markov chain.

Note that it is sometimes convenient to use several variables to describe the state of a component (we shall see examples in the next chapter).

7.2.2 Parameters

In our case study, all units are essentially the same, up to the values of their reliability data. It is thus worth to describe a generic model for units, as a class, and then to reuse this model. Parameters are a very convenient way to design reusable on-the-shelf modeling components and to adjust them to the particular needs of a specific model.

As an illustration, consider again the state automaton representing a repairable unit given in Figure 7.2. It is often the case that exponential distributions are assumed for failure and repair of these units. Different units may however have different failure and repair rates and it would be tedious to define a new class for each of these units. An alternative solution consists in declaring these rates as *parameters* that can be modified when instantiating the component. This principle is illustrated in Figure 7.6.

```
domain RepairableUnitState {WORKING, FAILED}

class RepairableUnit
    RepairableUnitState _state(init = WORKING);
    event failure(delay = exponential(failureRate));
    event repair(delay = exponential(repairRate));
    parameter Real failureRate = 1.0e-5;
    parameter Real repairRate = 3.0e-1;
    transition
        failure: _state==WORKING -> _state := FAILED;
        repair: _state==FAILED -> _state := WORKING;
end

block GasProductionFacility
    ...
    RepairableUnit S1(failureRate = 8.50e-5, repairRate = 2.50e-3);
    ...
end
```

Figure 7.6: Parametric AltaRica code for the state automaton of Figure 7.2

A parameter declaration consists of a type, a name and a default value. The value of the parameter is set once for all when the model is instantiated. In our example, the failure and repair rates of the separator `S1` are respectively 8.50×10^{-5} and 2.50×10^{-3}. Note that in the class `RepairableUnit`, parameters `failureRate` and `repairRate` are given default values, which can be kept when instantiating the class.

7.2.3 Flow Variables and Assertions

AltaRica 3.0 is an object-oriented modeling language of the S2ML+X family. Models of systems can be thus assembled from models for components and subsystems, using cloning and instances of classes. The main mechanism to glue together components is provided by flow variables and assertions.

As an illustration, consider again our gas production facility. Assume that for now, we are only interested in whether the facility is able to produce something or not and that we make the simplification that the separator `S3` and the compressor `C3` are in hot redundancy.

We can consider that each unit as an input and an output. More exactly, it outputs something if it receives something in input and it is working.

Figure 7.7 gives a possible AltaRica model for the units under these simplifying assumptions. We use now a class as we plan to instantiate this code several times to represent the system as a whole.

The model is, as previously, a state automaton with two states `WORKING` and `FAILED`, a transition `failure` going from `WORKING` to `FAILED` and a transition `repair` going from `FAILED` to `WORKING`. The novelty stands in the introduction of two ports, technically two Boolean *flow variables* `input` and `output` and an *assertion*.

```
domain RepairableUnitState {WORKING, FAILED}

class RepairableUnit
    RepairableUnitState _state(init = WORKING);
    event failure(delay = exponential(failureRate));
    event repair(delay = exponential(repairRate));
    parameter Real failureRate = 1.0e-5;
    parameter Real repairRate = 3.0e-1;
    Boolean input, output(reset = false);
    transition
        failure: _state==WORKING -> _state := FAILED;
        repair: _state==FAILED -> _state := WORKING;
    assertion
        output := input and _state==WORKING;
end
```

Figure 7.7: An AltaRica model for the units of gas production facility

Flow variables are declared like state variables. The difference stands in the attribute used to set up their default value: `init` for state variables and `reset` for flow variables.

The value of flow variables is recalculated after each transition firing, by means of the assertion. There is there an important rule to remember: the values of state variables are modified by the actions of transitions (and only by them), while the values of flow variables are modified by assertions (and only by them).

This assertion sets the values of output flow variables from the values of input flow variables and state variables. In this case, the Boolean variables `input` and `output` represent respectively whether the gas can flow into the unit, and if it can flow out.

Figure 7.8 shows the AltaRica model for the whole gas production facility.

This model starts by deriving the base class `RepairableUnit` into three classes `Separator`, `Dehydrator` and `Compressor`. These derivation do not do much. They only set the reliability parameters at their right values. They are nevertheless of interest because they make it easier to understand what is what in the description of the whole gas production facility.

The block that describes this facility composes:

- Two Boolean flow variables `W` and `P` representing respectively the well and the pipeline.
- A block `S` representing the separation line. This block composes itself the three separators `S1`, `S2` and `S3`.
- A block `D` representing the dehydration line. This block composes itself the two dehydrators `D1` and `D3`.
- Finally, a block `C` representing the compression line. This block composes itself the three compressors `C1`, `C2` and `C3`.

Assertions are used to connected the units. Note that the dot notation makes it possible to "crosses the walls" of units and subsystems. For instance, we could have plugged inputs of separators directly on the well, by writing the following assertion at system level.

```
S.S1.input := W;
S.S2.input := W;
S.S3.input := W;
```

`S.S1.input` refers to the variable `input` of the subsystem `S1` of the subsystem `S`.

In this simple example, we chose to have only Boolean flow variables. Flow variables can however have enumerated domains, be integers and even reals (floating point numbers).

```
class Separator
    extends RepairableUnit(failureRate = 8.50e-5, repairRate = 2.50e-3);
end

class Dehydrator
    extends RepairableUnit(failureRate = 3.00e-5, repairRate = 3.00e-3);
end

class Compressor
    extends RepairableUnit(failureRate = 3.50e-5, repairRate = 5.00e-3);
end

block GasProductionFacility
    Boolean W, P(reset = false);
    block S
        Boolean input, output(reset = false);
        Separator S1, S2, S3;
        assertion
            S1.input := input;
            S2.input := input;
            S3.input := input;
            output := S1.output or S2.output or S3.output;
    end
    block D
        Boolean input, output(reset = false);
        Dehydrator D1, D2;
        assertion
            D1.input := input;
            D2.input := input;
            output := D1.output or D2.output;
    end
    block C
        Boolean input, output(reset = false);
        Compressor C1, C2, C3;
        assertion
            C1.input := input;
            C2.input := input;
            C3.input := input;
            output := C1.output or C2.output or C3.output;
    end
    assertion
        W := true;
        S.input := W;
        D.input := S.output;
        C.input := D.output;
        output := C.output;
end
```

Figure 7.8: An AltaRica model for the simplified gas production facility

Note finally that guards of transitions can involve both state and flow variables.

7.2.4 Instructions

Instructions are used to describe actions of transitions and assertions. AltaRica 3.0 provides a small set of *instructions*. The experience shows however that they are sufficient.

The most important instructions are the assignment, the conditional instruction and the sequence.

The *assignment*, that we have seen in all AltaRica models given so far, consists in giving to a variable the value of an expression, e.g.

```
B2.input := A1.output or A2.output;
```

The *conditional instruction* consists in a "if-then-else", The else part is optional, e.g.

```
if _state==WORKING then
    _mode := OPERATION;
else
    _mode := REPAIR;
```

Finally, the *sequence* consists in executing several instructions in a row. The instructions of the sequence must then be surrounded by curly braces "`{`" and "`}`", e.g.

```
{
    _state:= WORKING;
    _mode := OPERATION;
}
```

More instructions can be found in the language specification (Batteux, Prosvirnova, and Rauzy 2017b).

7.2.5 Expressions

Conversely to instructions, whose number is limited, AltaRica 3.0 provides a wide range of expressions.

- Constants: Boolean `true` and `false`, integers (e.g. `42`, `-1`), reals (floating point numbers, e.g. `-273.15`, `1.23e-4`), and symbolic constants (e.g. `WORKING`, `FAILED`);
- References to variables and parameters;
- Boolean operations using logical connectors `and`, `or` and `not`;
- Arithmetic operations using usual operators `+`, `-`, `*`, `/`, as well as classical builtin functions, e.g. `mod`, `div`, `exp`, `pow...`;
- Inequalities, e.g. `a==b`, `a!=b`, `a<b`, `a>=b`;
- Conditional operations, e.g. `if _state==WORKING then input else LOST`;
- Builtin and empirical probability distributions, e.g. `exponential`, `Weibull...` (see Section 7.2.6);
- User defined operators (see Section 7.4.1);

The reader is invited to consult the language specification (Batteux, Prosvirnova, and Rauzy 2017b) for a precise description of available expressions, their syntax and their semantics.

7.2.6 Delays

In guarded transition systems, therefore in AltaRica 3.0, deterministic or stochastic *delays* are associated with events labeling the transitions. Stochastic delays are described by means of inverses of cumulative distribution functions. Recall that a cumulative distribution function is a monotone increasing function ϕ from non negative reals to the real interval $[0,1]$, i.e. it must verify that for all non negative real numbers t_1 and t_2, $0 \leq t_1 < t_2$, $\phi(t_1) \leq \phi(t_2)$.

There are two ways to define delays: by means of builtin expressions or by means of empirical distributions. The former method is much more frequent than the latter.

Builtin Distributions

Available builtin delays are the following (see Section 5.2.3 for mathematical definitions).

– Dirac, i.e. deterministic delays, e.g.

```
event startMaintenance(delay = Dirac(timeBetweenMaintenance));
```

– Exponential distributions, e.g.

```
event failure(delay = exponential(failureRate));
```

– Weibull distributions, e.g.

```
event failure(delay = Weibull(scaleParameter, shapeParameter));
```

– Uniform distributions, e.g.

```
event repair(delay = uniform(lowValue, highValue));
```

Empirical Distributions

Empirical distributions are given as a list of pairs $(t_1, p_1), \ldots (t_n, p_n)$, where the t_i's are non decreasing times, i.e. $t_1 \leq \ldots \leq t_n$, and the p_i's are non decreasing probabilities, e.g.

```
event failure (delay = [0.0:0.0, 876.0:0.02, 1752.0:0.4,
    2000.0:0.72, 3504.0:0.72, 5000.0:0.9]);
```

In between two successive t_i's, the value of the probability is linearly interpolated, as explained Section 5.2.3.

7.2.7 Comments

As in all of the languages of the S2ML+X family, there are two forms of comments in AltaRica 3.0.

Single line comments, which starts with a double slash "//" and spreads until the end of the line, e.g.

```
_state := WORKING; // To be checked
```

Multi-line comments that are surrounded with the sequences of characters "/*" and "*/", e.g.

```
/*
 * Repairable Units with exponentially distributed
 * failures and repair times.
 */
```

7.2.8 Coding Conventions

In modeling as in programming, it is of primary importance to adopt coding conventions and to stick to these conventions. It commonly said in software engineering literature that 70% of the cost of a program stands in its maintenance. Consequently, all what can be done to alleviate maintenance tasks is worth to take. Readability of the code is of paramount importance. The person in charge of reading the code should be able to seize at a glance what is what.

Throughout this book, we shall adopt the following coding conventions.

– Identifiers domains, classes, functions and records are capitalized, starting with an upper case letter, e.g. `Pump`, `MotorPump`, `RepairableUnitState`...

– Identifiers of blocks and instances of classes are also capitalized and starting with an upper case letter, but may be short and end with digits, e.g. `P`, `Pump42`...

- Identifiers of symbolic constants are in capital letters `WORKING`, `FAILED`... We try to avoid as much as possible compound words. In case this cannot be avoided, words are separated with underscores, e.g. `FAIL_SAFE`.
- Identifiers of flow variables, events, parameters and observers are capitalized, starting with a lower case letter, `in`, `failureOnDemand`... They may be short and end with digits, e.g. `in2`...
- Identifiers of state variables are similar to those of flow variables, except they start with an underscore, e.g. `_working`, `_mode`...

Identifiers must be significant. Calling a variable `x` does not tell anything about its pragmatics. Adding comments to compensate poor naming tends to make things worse: after a while, comments are dealigned from the code, increasing the mess.

Indentation, i.e. shifting the code to the right, is also key to ensure the readability of the code. It shows the logical structure of the code. Throughout this book, we use tabulation characters as indentation, see for instance Figure 7.6.

These conventions must be obeyed from the very first time the code is written. It is not something that can be postponed until "I will have time to polish the code". The experience shows actually that this time never comes.

7.3 Formal Definition and Semantics

AltaRica 3.0 is a formal language. Its semantics is defined without ambiguity. AltaRica 3.0 models encode well defined mathematical objects and all calculations made on these models have their counterpart in terms of operations performed on these mathematical objects. This section gives a formal definition of guarded transition systems and of their semantics.

7.3.1 Formal Definition

Guarded transition systems are defined as follows (Batteux, Prosvirnova, and Rauzy 2017a; Rauzy 2008b).

Definition 7.3.1 – Guarded Transition System. A *guarded transition system* is a quintuple $\langle V, E, T, A, \iota \rangle$, where:

- V is a finite set of of variables. V is the disjoint union of two subsets S and F ($V = S \uplus F$), where S is the subset of *state variables* and F is the subset of *flow variables*. Each variable v of V comes with its *domain*, denoted $\mathrm{dom}(v)$, i.e. the set of values this variable can take. Domains are either predefined domains (Boolean, integers or floating point numbers), or user defined domain, i.e. finite sets of symbolic constants.
- E is a finite set of events. Each event e of E is associated with a *delay*, denoted δ_e, i.e. the inverse of a cumulative probability distribution.
- T is a finite set of triples $\langle e, g, a \rangle$, called transitions, where
  - e is an event of E, and
  - g is a Boolean expression on the variables of V called the *guard* of the transition,
  - a is an instruction that involves only variables of V and changes values of state variables only. a is called the *action* of the transition.

  A transition $\langle e, g, a \rangle$ is usually denoted by $g \xrightarrow{e} a$.
- A is an instruction called the *assertion* that involves only variables of V and changes values of flow variables only.
- Finally, ι is a variable valuation, i.e. an element of $\mathrm{dom}(V) = \prod_{v \in V} \mathrm{dom}(v)$. If v is a state variable, then $\iota(v)$ is called the *initial value* of v. If v is a flow variable, then $\iota(v)$ is called the *default value* of v.

Let $\langle V, E, T, A, \iota \rangle$ be a guarded transition system.
States of a guarded transition system are thus variable valuations.
A transition $g \xrightarrow{e} a$ is *enabled* in the state σ, if σ satisfies g, i.e. if $\sigma(g) = true$.
Firing the transition $g \xrightarrow{e} a$ in the state σ transforms σ into $A \circ a(\sigma) = A(a(\sigma))$.

7.3.2 Free Product

The composition of two guarded transition systems is itself a guarded transition system. This property makes it possible to create hierarchical models and to use the full power of the S2ML+X paradigm. In the algebraic jargon, the composition of two objects of same type, e.g. two groups, that gives an object of the same type is called a *free product*.

Definition 7.3.2 – Free Product of Guarded Transition Systems. Let $M_1 : \langle V_1, E_1, T_1, A_1, \iota_1 \rangle$ and $M_2 : \langle V_2, E_2, T_2, A_2, \iota_2 \rangle$ be two guarded transition systems such that $V_1 \cap V_2 = \emptyset$ and $E_1 \cap E_2 = \emptyset$. Then, the *free product* of M_1 and M_2, denoted $M_1 * M_2$ is the guarded transition system $M : \langle V, E, T, A, \iota \rangle$ such that:

- $V = V_1 \cup V_2$;
- $E = E_1 \cup E_2$;
- $T = T_1 \cup T_2$;
- $A = A_2 \circ A_1$;
- $\iota = \iota_2 \circ \iota_1$.

Strictly speaking the free product is not commutative, as the composition of assertions and initial valuations are not. However, as we assumed that sets of variables of the two guarded transition systems are disjoint, the order in which assertions and initial valuations are given does not matter. Hence the following property.

Property 7.1 – Free Product of Guarded Transition Systems. The free product of guarded transition systems is commutative and associative, up to an isomorphism.

In AltaRica, prefixing of identifiers of variables and events by the name of the block that composed them during the *instantiation* and the *flattening* of models (see Section 4.10) ensures to make free products of guarded transition systems with disjoint sets of variables and events.

AltaRica models are actually transformed into "pure" guarded transition systems by this process, prior to any further treatment.

As an illustration, let us come back to the gas production system. In the model given in Figure 7.8, the separator `S3` and the compressor `C3` are in hot redundancy with the others separators and compressors, which does not correspond to the description of the system. In reality, these two units are initially in standby. They are turn on when one of the other units they supply fails and turn off when this unit is repaired, unless the other main unit fails in between.

Figure 7.9 gives a possible model for the separation line. Now, the separator `S3` is a spare unit. It works as explained above, using the flow variable `demand` to know when to turn it on and off. The condition is that the separation line as whole can work, i.e. it receives the mix of oil, gas and water as input, and that at least one of the two main separators `S1` and `S2` does not provide the required production.

The AltaRica model for units `S1`, `S2` and `S3` can directly be interpreted as guarded transition systems. The model is thus eventually built by constructing the free product of these 3 guarded transition systems augmented with the assertion given in the block `S`.

The resulting guarded transition system involves 3 state variables (one per unit), 9 flow variables (2 per main unit, 3 for the spare unit, plus 2 declared in the main block), 8 events (2 per main units and 4 for the spare unit) and as many transitions.

```
domain SpareUnitState {STANDBY, WORKING, FAILED}

class SpareUnit
    SpareUnitState _state(init = STANDBY);
    Boolean demand(reset = false);
    event failure(delay = exponential(failureRate));
    event repair(delay = exponential(repairRate));
    event turnOn(delay = Dirac(0));
    event turnOff(delay = Dirac(0));
    parameter Real failureRate = 1.0e-5;
    parameter Real repairRate = 1.0e-3;
    Boolean input, output(reset = false);
    transition
        failure: _state==WORKING -> _state := FAILED;
        repair: _state==FAILED -> _state := STANDBY;
        turnOn: _state==STANDBY and demand -> _state := WORKING;
        turnOff: _state==WORKING and not demand -> _state := STANDBY;
    assertion
        output := input and _state==WORKING;
end

class Separator
    extends RepairableUnit(failureRate = 8.50e-5, repairRate = 2.50e-3);
end

class SpareSeparator
    extends SpareUnit(failureRate = 8.50e-5, repairRate = 2.50e-3);
end

block S
    Separator S1;
    Separator S2;
    SpareSeparator S3;
    Boolean input, output(reset = false);
    assertion
        S1.input := input;
        S2.input := input;
        S3.input := input;
        S3.demand := input and (not S1.output or not S2.output);
        output := S1.output or S2.output or S3.output;
end
```

Figure 7.9: AltaRica code for the separation line, taking into account cold redundancies

Note that as the values of flow variables are fully determined by the values of state variables, the state of the guarded transition system is characterized by the latter.

7.3.3 Semantics

The semantics of guarded transition systems is very similar to the one of continuous time KMF models we shall see in Chapter 9.

Recall that a *schedule* is a mechanism γ that associates a number in $\mathbb{R}^+ \cup \{\infty\}$ with each transition.

Executions of guarded transition systems are sequences of the form $\langle d_0 = 0, \sigma_0 = \iota, \gamma_0 \rangle \xrightarrow{e_1}$

$\langle d_1, \sigma_1, \gamma_1 \rangle \cdots \xrightarrow{e_n} \langle d_n, \sigma_n, \gamma_n \rangle$, $n \geq 0$, where the d_i's are dates (non-negative real numbers), σ_i's are states (variable valuations), the γ_i's are schedules, and the e_i's are events labeling transitions. Formally:

Definition 7.3.3 – Semantics of Guarded Transition Systems. Let $M : \langle V, E, T, A, \iota \rangle$ be a guarded transition system. The set of *executions* of M is the smallest set such that:

- The empty execution $\langle 0, \iota, \gamma_0 \rangle$ is an execution of M if for all transitions $t : g \xrightarrow{e} a \in T$, $\gamma_0(t) = \delta_e(z)$ for some $z \in [0,1]$ if t is enabled in ι and ∞ otherwise.
- If $S : \langle d_0, \sigma_0, \gamma_0 \rangle \xrightarrow{e_1} \langle d_1, \sigma_1, \gamma_1 \rangle \cdots \xrightarrow{e_n} \langle d_n, \sigma_n, \gamma_n \rangle$, $n \geq 0$, is an execution of M and $t : g \xrightarrow{e} a$ is a transition of T such that t is enabled in σ_n and for all $t' \neq t$ of T, $\gamma_n(t) \leq \gamma_n(t')$: then $S \xrightarrow{e} \langle d, \sigma, \gamma \rangle$ is an execution of M if:
  - $\sigma = A \circ a(\sigma_n)$;
  - $\gamma(t) = d + \delta_e(z)$ for some $z \in [0,1]$ if t is enabled in σ and ∞ otherwise;
  - For all other transition $t' : g' \xrightarrow{e'} t'$ of T ($t' \neq t$):
    - If t' is enabled both in σ_n and σ, then $\gamma(t') = \gamma(t)$,
    - If t' is enabled in σ but not in σ_n, then $\gamma(t') = d + \delta_{e'}(z)$ for some $z \in [0,1]$,
    - If t' is not enabled in σ, then $\gamma(t') = \infty$.

As an illustration, consider again the guarded transition system obtained by instantiating and flattening the AltaRica model given in Figure 7.9.

Initially, separators `S1` and `S2` are working while separator `S3` is in standby. Two transitions are thus enabled: `S1.failure` and `s2.failure`. Assume they are respectively scheduled at $t = 1010.4$ and 706.2. As `S2.failure` has the lowest firing date, it is fired (at 706.2).

The transition `S2.repair` is now enabled. It is scheduled for instance at $706.2 + 111.1 = 817.3$ Meanwhile, the variable `S2.output` gets false and `S3.demand` gets true. The transition `S3.turnOn` is now enabled and schedule at $706.2 + 0 = 706.2$. As `S3.turnOn` is the scheduled transition with the lowest firing date, it is fired (at 706.2).

The transition `S3.failure` is now enabled. It is scheduled for instance at $706.2 + 602.2 = 1308.4$.

We have now three transitions scheduled: `S1.failure` (at $t = 1010.4$), `S2.repair` (at $t = 817.3$), and `S3.failure` (at $t = 1308.4$). As `S2.repair` is the scheduled transition with the lowest firing date, it is fired (at 817.3).

And so on.

7.4 Advanced Features

7.4.1 Preprocessed Elements

Preprocessed elements are syntactic constructs that avoid to repeat complex expressions or instructions, or simply that make it possible to separate such expressions and instructions from the core of the description. They are similar to macro-expressions in languages such C.

First, it is possible to name numerical constants, by means of *constant declaration*, e.g.

```
constant Real g = 9.81;
```

`g` can then be used everywhere in the model.

Second, it is possible to define *operators*. Assume for instance we want to work in a trivalued logic, with truth values `TRUE`, `FALSE` and `UNKNOWN`. After defining the corresponding domain, we can declare trivalued logical operators, e.g.

```
domain TruthValue {TRUE, FALSE, UNKNOWN}

operator TruthValue TrivaluedOr(TruthValue arg1, TruthValue arg2)
    if arg1==TRUE or arg2==TRUE then
        TRUE
    else if arg1==FALSE and arg2==FALSE then
        FALSE
    else UNKNOWN
end
```

`TrivaluedOr` can then be applied to any pair of expressions (of type `TruthValue`), e.g.

```
TruthValue in1, in2, out(reset = UNKNOWN);
⋮
out := TrivaluedOr(in1, in2);
```

Operators can take any number of arguments, possibly of different types.

Finally, it is possible to declare *functions* that group together several instructions, e.g.

```
function Swap(Integer v1, Integer v2, Integer q, Integer r)
    q := div(v1, v2);
    r := mod(v1, v2);
end
```

Functions are however seldom used in practice.

Note that, in the current version of the language at least, AltaRica compiler simply substitute the body of operators and functions for their calls. Consequently, it is impossible to define recursive operators and functions.

7.4.2 Synchronizations

When a transition has a coordinated effect on several components of the system under study, it is sometimes convenient to describe its enabling conditions and actions locally then to synchronize these local descriptions to get the full transition. AltaRica 3.0 provides the powerful concept of *synchronization* to do so. This concept has been originally introduced by Arnold and Nivat to describe interactions between concurrent processes (Arnold 1994). It is generalized in AltaRica 3.0.

We shall illustrate the principle of synchronization by means of two important patterns: shared resources and common cause failures.

Shared Resources

As a first illustration, consider again the compressor line of our gas production facility. Repairing compressors is the job of highly specialized technicians.

Assume that, for economic reasons, there is only one maintenance team `M` and that technicians can intervene on only one compressor at a time. In other words, the three compressors `C1`, `C2` and `C3` share a resource, namely the maintenance team. When the compressor `C1` fails and the maintenance team is not already working on another compressor, the repair of `C1` starts. The maintenance team becomes simultaneously unavailable for the two other compressors. Once `C1` is repaired, the resource is released, i.e. the maintenance team becomes available again to repair compressors `C2`, `C3`... and potentially `C1` again. Symmetrically for `C2` and `C3`.

A possible code for the compressor line is given Figure 7.10. For the sake of simplicity, we assume here that the three compressors are in hot redundancy.

In this code, the definitions of classes `Compressor` and `MaintenanceTeam` are no surprise. The block `CompressorLine` declares six events, three to represent the start of repairs

```
domain CompressorState {WORKING, FAILED, REPAIR}

class Compressor
    CompressorState _state(init=WORKING);
    event failure(delay = exponential(lambda));
    event startRepair, completeRepair;
    parameter Real lambda = 1.0e-4;
    transition
        failure: _state==WORKING -> _state:=FAILED;
        startRepair: _state==FAILED -> _state:=REPAIR;
        completeRepair: _state==REPAIR -> _state:=WORKING;
end

domain MaintenanceTeamState {STANDBY, WORKING}

class MaintenanceTeam
    MaintenanceTeamState _state(init=STANDBY);
    event startJob, completeJob;
    transition
        startJob: _state==STANDBY -> _state:=WORKING;
        completeJob: _state==WORKING -> _state:=STANDBY;
end

block CompressorLine
    Compressor C1, C2, C3(startRepair.hidden = true,
        completeRepair.hidden = true);
    MaintenanceTeam M(startJob.hidden = true,
        completeJob.hidden = true);
    event startRepair1, startRepair2,
        startRepair3(delay = Dirac(0));
    event completeRepair1, completeRepair2,
        completeRepair3(delay = exponential(mu));
    parameter Real mu = 0.025;
    transition
        startRepair1: !C1.startRepair & !M.startJob;
        startRepair2: !C2.startRepair & !M.startJob;
        startRepair3: !C3.startRepair & !M.startJob;
        completeRepair1: !C1.completeRepair & !M.completeJob;
        completeRepair2: !C2.completeRepair & !M.completeJob;
        completeRepair3: !C3.completeRepair & !M.completeJob;
end
```

Figure 7.10: AltaRica 3.0 code for units (compressors) sharing a maintenance team

of each component and three to represent their completion. The corresponding transitions are synchronization.

For instance, the transition `startRepair1` results of the simultaneous firing of transitions `startRepair` of compressor `C1` and `startJob` of the maintenance team. The modality `!` prefixing the event names makes the corresponding transition mandatory: the global transition `startRepair1` is enabled only if local transitions `C1.startRepair` and `M.startJob` are. In other words, this transition is equivalent to the following one.

```
startRepair1: C1._state==FAILED and M._state==STANDBY -> {
        C1._state := REPAIR;
        M._state := WORKING;
        }
```

It is however much more convenient to declare local effect locally.

Instances of compressors and maintenance team are declared with the attribute `hidden` of their events `startRepair` and `completeRepair`, respectively `startJob` and `completeJob`,set to `true`. This indicates that these events cannot be fired individually, but only through synchronizations.

Synchronizations can involve more than two events (see exercise 7.14).

Common Cause Failures

Common cause failures are an important contributor to the risk in many technical systems, e.g. in nuclear power plants, see e.g. (Mosleh, Rasmuson, and Marshall 1998). There are many types of common cause failures. We shall restrict our attention the case where an event impacts simultaneously several basic components and fails all impacted components that are not already failed. Fire or flooding are typically such events. Strictly speaking, common cause failures as defined above cannot be described at basic component level since they involve several of them.

As an illustration, consider the compressor line of our gas production facility. Assume that the three compressors are subject to a common cause failure `ccf`, due for instance to problems in lubrication.

The AltaRica 3.0 code that describes a common cause failure acting on three compressors (here assumed to be in hot redundancy) is given in Figure 7.11.

```
1  domain CompressorState {WORKING, FAILED}
2
3  block Compressor
4      RepairableUnitState _state(init = WORKING);
5      event failure(delay = exponential(1.0e-4));
6      event repair(delay = exponential(1.0e-1));
7      transition
8          failure: _state==WORKING -> _state := FAILED;
9          repair: _state==FAILED -> _state := WORKING;
10 end
11
12 block CompressorLine
13     Compressor C1, C2, C3;
14     event ccf (delay = exponential(ccfRate));
15     parameter Real ccfRate = 1.0e-6;
16     transition
17         ccf: ?A.failure & ?B.failure & ?C.failure;
18 end
```

Figure 7.11: AltaRica 3.0 code describing a common cause failure impacting three repairable units

The event and the transition representing the common cause failure are declared in the block `CompressorLine` that composes the compressors `C1`, `C2` and `C3`. The transition `ccf` synchronizes the three events `C1.failure`, `C2.failure` and `C3.failure`, i.e. attempts to fire these three events simultaneously. The modality "`?`" indicates that event it prefixes is fired only if possible (a modality "`!`" would indicate that event it prefixes is mandatory). The transition labeled with `ccf` is enabled if at least one the three local events is, i.e. if at least one of the guards of the

individual transitions is satisfied in the current state. Its firing consists in performing actions of individual transitions that can be fired. Eventually, it is thus equivalent to the following code.

```
ccf: C1._state==WORKING or C2._state==WORKING or C3._state==WORKING -> {
    if C1._state==WORKING then C1._state := FAILED;
    if C2._state==WORKING then C2._state := FAILED;
    if C3._state==WORKING then C3._state := FAILED;
    }
```

7.4.3 More on Assertions

The ability to represent "simply" remote interactions is of paramount importance for probabilistic risk and safety analyses. So far, all models we considered are *data-flow*, i.e. that the information propagates only one way between components. There are systems however in which the information can circulate in either directions, depending on the states of the components, hence introducing loops in the model.

Reliability networks belong to this category of models. Although much less popular than fault trees and reliability block diagrams, they have focused important research efforts, see e.g. (Colbourn 1987; Madre et al. 1994; Rauzy 2003a; Shier 1991). The reason is that power and water distribution networks as well as several other types of infrastructures are typically analyzed as reliability networks.

Handling these looped models requires specific propagation algorithms. Namely, these algorithm must implement, in a way or an other, a fixpoint mechanism. Theoretical developments on this question go beyond the scope of this book. The reader interested by a throughout discussion should refer to articles by the author (Batteux, Prosvirnova, and Rauzy 2017a; Rauzy 2003a; Rauzy 2008b). As of today, AltaRica 3.0 is the only modeling language embedding natively such mechanism. It is used to solve assertions. We shall illustrate here its power by showing how to represent and assess reliability networks in AltaRica 3.0.

Recall that a *reliability network* is a graph with two kinds of nodes: source nodes that produce something (power, information, ...) and target nodes that consume and redistribute this something. Nodes of both types may fail according to some probability distribution. Edges are assumed to be perfectly reliable. This assumptions does not lose any generality by assuming edges are perfect: an imperfect edge between two nodes `A` and `B` can be represented by introducing an imperfect node `N` and two perfect edges form `A` to `N` and from `N` to `B`.

As an illustration, consider the network pictured in Figure 7.12. This network is made of the two source nodes `S1` and `S2` and six target nodes `T1`, ... `T6`.

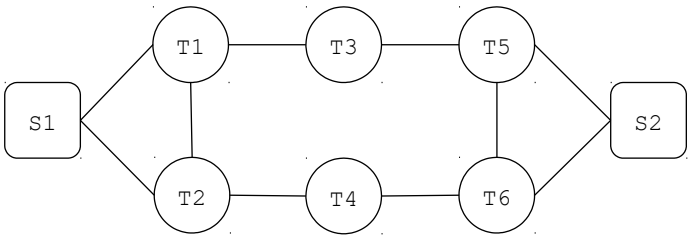


Figure 7.12: A reliability network.

In this network, edges are bidirectional. The information can thus circulate in different directions, depending on the state of components. For instance, the node `T6` can be powered by the node `S1` via the nodes `T2` and `T4`. In case `T2` and `S2` are failed, `T4` can be powered by the node `S1` via the nodes `T1`, `T3`, `T5` and finally `T6`.

A typical question one may ask on such a network is "what is the probability that a specific node is a powered at time t", i.e. what is the probability that there exists a working path from one of the operating source nodes to that particular node. Answering this question raises difficult algorithmic issues. Specialized algorithms have been developed (Madre et al. 1994; Rauzy 2003a).

In AltaRica 3.0, the solution is surprisingly simple and elegant. It relies on the following pattern.

– Each component has a unique Boolean input and a unique Boolean output.
– The input of a component is defined as a Boolean formula of the outputs of the other nodes.

Whether each component is reachable, i.e. has at least one of its inputs true, depends on the global state of the network.

Figure 7.13 gives the AltaRica model that represents the reliability pictured in Figure 7.12.

```
domain RepairableNodeState {WORKING, FAILED}

class RepairableNode
    RepairableNodeState _state(init = WORKING);
    Boolean input, output(reset = false);
    event failure(delay = exponential(1.0e-4));
    event repair(delay = exponential(1.0e-1));
    transition
        failure: _state==WORKING -> _state := FAILED;
        repair: _state==FAILED -> _state := WORKING;
    assertion
        output := input and _state==WORKING;
end

block ReliabilityNetwork
    RepairableNode S1, S2, T1, T2, T3, T4,  T5, T6;
    assertion
        S1.input := true;
        S2.input := true;
        T1.input := S1.output or T2.output or T3.output;
        T2.input := S1.output or T1.output or T4.output;
        T3.input := T1.output or T5.output;
        T4.input := T2.output or T6.output;
        T5.input := S2.output or T3.output or T6.output;
        T6.input := S2.output or T4.output or T5.output;
end
```

Figure 7.13: AltaRica 3.0 code for the reliability network of Figure 7.12

We made here some simplifying assumptions for the sake of conciseness. In particular, all nodes are repairable and their failures and repairs are associated with the same exponential distributions. It is of course possible to release these assumptions.

The key point here is that the assertion of this model is "looped", i.e. that flow variables depends eventually on each other. AltaRica 3.0 fixpoint mechanism to update the value of flow variables after each transition firing is able to deal efficiently with such dependencies (Batteux, Prosvirnova, and Rauzy 2017a).

We can now look at the various tools that can be used to assess AltaRica 3.0 models, starting with step by step simulation.

7.5 Interactive Simulation

7.5.1 Rational

Designing stochastic discrete event systems presents a well-known difficulty: models are hard to debug and to validate because of the existence of infinitely many possible executions, itself due to stochastic delays, which are possibly intertwined with deterministic ones. AltaRica models make no exception to this general observation. This is probably the main limiting factor to their full scale deployment, especially in the context of performance assessment of life-critical systems. Even seasoned analysts experience regularly the frustration of waiting long minutes, if not hours, for the results of a Monte-Carlo simulation to discover eventually that these results are meaningless because of a mistake somewhere in the model.

This is the reason why interactive simulation of models plays in practice a considerable role in the design, the debugging and the validation of models. Interactive simulators allow the analyst to go forth and back, step by step, in sequences of events, enabling in this way to track modeling errors, unexpected behaviors and so on. With that respect, they play a similar role as debuggers like GDB or DDD (Matloff and Salzman 2008) do for C++ programs. Similar tools exist for most of the programming languages.

It is strongly advised to check your AltaRica 3.0 models with the interactive simulator prior any other treatment. Although fairly simple, a model like the one presented in Figure 7.10 may be tricky to design.

7.5.2 Principle

A priori, the principle of interactive simulation is fairly simple: The interactive simulator starts a new simulation, displays the current state of the system and proposes to the analyst a list of transitions that can be fired. The analyst can pick up one of these transitions. The simulator fires it and the process restarts. After a few transitions have been fired, it is possible to backtrack the last one (or ones) so to look at another event sequence. It is possible in this way to explore different parts of the model.

This principle is however hampered by stochastic delays associated with transitions. There are actually infinitely many possible values for these delays. The sequence sketched at the end of Section 7.3.3 of the present chapter illustrates this issue.

Until recently, the developers of interactive simulators faced a quite unpleasant choice: either ignoring delays, which has the major drawback that some non-timed executions may have no timed counterpart, or ask the analyst to enter by hand the delays associated with stochastic transitions, which is tedious and let the analyst pondering which out of delays are the most suitable for his purpose.

The problem has been solved by introducing an abstract semantics for discrete event systems in general and by applying this general framework to AltaRica models (Batteux, Prosvirnova, and Rauzy 2021). In this article, the authors revisit ideas introduced in the framework of model-checking of timed and hybrid systems (Larsen, Pettersson, and W. Yi 1997; W. Yi, Pettersson, and Daniels 1994). They show that it is possible to abstract the time in stochastic discrete event systems. Namely, they define an abstraction of transition schedules by means of systems of linear inequalities. These systems encode the conditions for a transition to be enabled at a given step of an execution: the transition is enabled if and only if the corresponding system has a solution. The key property is that abstract and concrete executions are bisimilar in the following sense (see e.g. (Milner 1989) for a reference textbook on bisimulations): any concrete execution can be simulated by a unique abstract execution and reciprocally any abstract execution corresponds to at least one concrete execution. This property is of a great interest because abstract models can be verified with techniques developed for non-timed discrete event systems, including model-checking techniques (Baier and Katoen 2008; Clarke, Grumberg, and Peled 2000). Even without entering

into the model-checking framework, this property makes it possible to perform abstract interactive simulations, therefore alleviating considerably debugging and validation tasks of AltaRica models.

Concretely, the analyst does not need to look at the systems of linear inequalities generated by the AltaRica interactive simulator. They are run behind the scene in order to let the analyst focus on the validation of his model.

7.6 Stochastic Simulation

Monte-Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The idea is to use randomness to solve problems that might be deterministic in principle. This idea has been introduced in the late 1940s, by Stanislaw Ulam and John von Neumann while Ulam was working on nuclear weapons projects at the Los Alamos National Laboratory.

Stochastic simulation is the application of Monte-Carlo methods to the specific case where the underlying model involves variables that can change stochastically (randomly) according individual probability distributions. Thanks to the progress of computers and random number generators, stochastic simulation is nowadays the Swiss knife of reliability engineering (Zio 2013). It makes it possible to calculate key performance indicators, almost whatever the underlying modeling formalism. It plays also a central role in the AltaRica technology.

7.6.1 Principle

In the context of AltaRica, stochastic simulation consists in drawing at random sufficiently many executions (event sequences) and to make statistics on these executions. Figure 7.14 gives the pseudo-code of the stochastic simulation algorithm.

```
1  StochasticSimilation()
2      for n:=1 to numberOfExecutions:
3          RestartSimulation()
4          simulationOver := false
5          while not simulationOver:
6              transition := SelectFirstScheduledTransition()
7              if transition==none or FiringDate(transition)>missionTime:
8                  simulationOver := true
9              else:
10                 FireTransition(transition)
11                 UpdateSchedule()
12         RecordValuesOfPerformanceIndicators()
13     MakeStatisticsOnPerformanceIndicators()
```

Figure 7.14: Pseudo-code for the stochastic simulation

In the above description, all terms are important and need to be explained:

- Where the randomness comes from?
- How to define indicators on which statistics are made?
- How to make statistics on these indicators, in particular how many executions should be performed?

We shall now review these points in turn.

Where the Randomness Comes from?

First, we need to clarify what we mean by drawing at random executions. If all transitions of the model are deterministic, i.e. their events are associated with Dirac delays, the model has only one

possible evolution. If, on the contrary, at least one transition is stochastic (and there is at least one sequence in which this transition can be fired), then infinitely many possible delays can be associated with this transition. The idea is thus to pick up delays at random each time a stochastic transition is enabled, as illustrated in Figure 7.15.

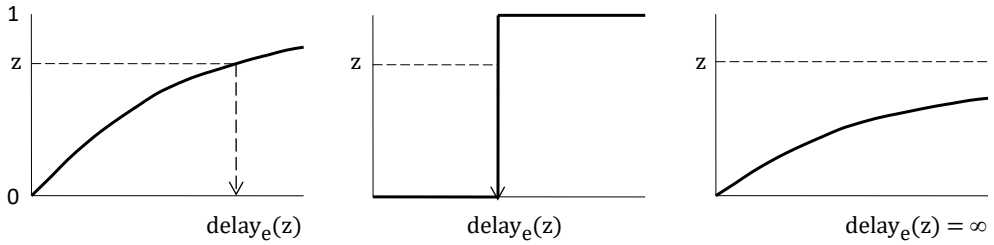


Figure 7.15: Drawing delays at pseudo-random

Conceptually at least, the idea is to draw at random at number z between 0 and 1 according to a uniform distribution and to calculate the corresponding delay $\text{delay}_e(z)$ (recall that delays are inverse of cumulative distribution functions). In practice, the algorithms to generate delays at random may be slightly more complex, but we shall not enter into that here.

Figure 7.15 (left) shows a typically case where the delay obeys an exponential distribution. Figure 7.15 (center) shows a case where the delay obeys a stepwise distribution. Finally, Figure 7.15 (right) shows the case where the number z does not correspond to any value of the cumulative distribution function, in which case $\text{delay}_e(z)$ is set to infinity.

Stochastic simulation requires thus a mechanism to generate numbers uniformly at random between 0 and 1. In practice, algorithmic generators are used. More specifically, the AltaRica stochastic simulator uses the Mersenne-Twister generator (Matsumoto and Nishimura 1998) that is considered as one of the very best available today, see C.4 for a discussion.

Performance Indicators

The AltaRica 3.0 stochastic simulator has access to the current state of the model through observers. The current version of the simulator works with three types of observers: Boolean observers, numerical observers (integer or real), and user-defined domain observers. Boolean and user-defined domain observers are actually similar as what is actually observed is the predicate *observer* = *value*, where *value* is either `true` or `false` in the case of Boolean observers, and a symbolic constant of the domain of the observer in case of a user-defined domain observer. For this reason, Boolean and user-defined domain observers are just referred to as *symbolic observers*.

The value symbolic and numerical observers at a given step of an execution are fully determined by the values of state and flow variables at that step. In this sense, these observers can be seen as a special type of flow variables (with the difference that observers cannot be used in guards and actions of transitions nor in assertions).

In practice, it is useful to make statistics not only on the state of the model at time t, but also on its evolution from time 0 to time t. The AltaRica 3.0 stochastic simulator introduces thus *indicators*. Indicators are values calculated at time t from the successive values of observers from time 0 to time t.

There are three types of indicators: *Boolean indicators* (that take either the value true or the value false), *numerical indicators* (that take integral or real values) and *duration indicators* (whose value is a positive real number). Duration indicators are typically used to measure the first date at which the observer took a given value or the mean time between two successive dates at which the observer took that value. Duration indicators differ from numerical observers in that their value may be undefined in some executions: for instance if the observer never takes the given value in the execution, then the first date at which it takes this value just does not exist. Statistics are thus

made only on executions for which duration indicators are meaningful and the number of these executions is recorded.

Indicators for symbolic observers

Available indicators for symbolic observers are the following (recall that these indicators are calculated for an observer O and a value v at time t).

`has-value` is a Boolean indicator. Its value is true if the observer O takes the value v at time t and false otherwise.

`had-value` is a Boolean indicator. Its value is true if the observer O took the value v for a non-null period at least once from time 0 to time t and false otherwise.

`sojourn-time` is a numerical indicator. Its value is the time the observer O had the value v from time 0 to time t.

`first-occurrence-date` is a duration indicator. Its value is the first date, if any, at which the observer O took the value v in the time period $[0,t]$.

`number-of-occurrences` is a numerical indicator. Its value is number of times the observer O took the value v for a non-null period from time 0 to time t.

`mean-time-between-occurrences` is a duration indicator. Its value is the mean duration between two successive dates at which the observer O took the value v for a non-null period from time 0 to time t.

Indicators for numerical observers

Available indicators for numerical observers are the following.

`value` is a numerical indicator. Its value is the value of the observer O at time t.

`mean-value` is a numerical indicator. Its value is the mean value of the observer O from time 0 to time t.

Statistics

Executions are developed from time 0 to a given *mission time* T. It is possible however to order the stochastic simulator to make statistics on indicators not only at time T, but also intermediate dates $0 \leq t_1 < t_2 \cdots \leq T$.

The AltaRica 3.0 stochastic simulator makes statistics on indicators. The analyst has to define which statistics are made on which indicator. To make statistics, Boolean indicators are assimilated to 0-1 variables. Similarly, duration indicators are assimilated to numerical indicators, but the statistics are made only on executions (and dates) for which the indicator is meaningful. Available statistics are of two types:

- Moments: mean, standard-deviation and 95% confidence range.
- Distributions: quantiles, bins.

Appendix C.3 provides a comprehensive introduction to these different measures.

Moments

Moments are calculated as follows.

- The (empirical) mean $\mu(X)$ (which is also the expectation $E[X]$) of the indicator X considered as a random variable) is calculated as the sum of the observed values of X divided by the number n of observations.
- The standard-deviation $\sigma(X)$ is by means of the Welford's algorithm (Welford 1962).
- The 95% confidence range via the 95% error factor.

Distributions

Distributions are calculated (in principle) by sorting the n observed values of the indicator X and splitting these n values into k successive bins: the first bin contains the k/n smallest values, the

second one contains the k/n next values and so on. Then, the minimum, maximum and mean values of each bin are calculated. The maximum value of the j-th bin is the j-th k-quantile.

In practice, the number of bins is limited to 100. Moreover, the values are not recorded. Rather, bins are filled in on-the-fly so to avoid too high memory consumption. Therefore, calculated minimum, maximum and mean values are approximated.

Bins are filled on-the-fly by storing results into an intermediate table. When this table is full or at the end of the simulation, it is flushed into the bins. So if, for instance, there is 10 bins, the size of the table is 10000 and the simulation consists of 1000000 executions, the table is flushed 100 times into the bins. The larger the table, the more precise the calculation of bins. The size of the table is calculate from the number of bins by multiplying this number by a *shrink factor* (see in-line help to see how to modify the default value of this parameter).

What should be the size of the sample?

This question goes indeed beyond the specific case of AltaRica 3.0 stochastic simulation and even of discrete event systems.

In theory, stochastic simulation converges. If there are sufficiently many executions, then the law of large numbers states the average value obtained for an indicator must be close to the true value. The central limit theorem states that the average has a Gaussian distribution around the true value.

In practice, the convergence of the method depends on the average value of the indicator and the dispersion of values around the average. When estimating a frequency f, the lower f the higher must be the number n of executions. As a rule of thumb, one can set $n \approx 100/f$. But this does not warranty good results if the dispersion of possible values around f is very important.

In any case, it is strongly advised to make several experiments so to confirm the robustness of the results.

How to make the method sufficiently robust and efficient?

This question is strongly linked to the previous one: the faster an execution, the higher the number of executions that can be performed within a given time (or calculation resources) budget; and the higher this number of executions, the better the results.

This is the reason why the AltaRica 3.0 stochastic simulator works by compiling the model into an executable program. Nevertheless, stochastic simulation is resource consuming.

Workflow

The workflow of the AltaRica 3.0 stochastic simulator is pictured in Figure 7.16.

The steps are the following.

1. The AltaRica model is first instantiated and flattened into a guarded transition system (by the AltaRica 3.0 flattener).
2. The analyst specifies the performance indicators to be calculated. The description of these indicators is stored into an "indicator description file" (concretely an XML file).
3. The AltaRica stochastic simulator compiler compiles the guarded transition system and the indicators into a C++ program, which is in turn compiled to get an executable program.
4. The analyst specifies the mission, i.e. the dates at which statistics must be made as well as the number of executions to be performed. The description of the mission is stored into a "mission description file" (concretely an XML file).
5. Finally, the stochastic simulation strictly speaking is launched. Its results are printed out into a file, usually a CSV file so that it can be read both by text editors and spreadsheet tools.

In the AltaRica Wizard integrated modeling and simulation environment, a series of panels make it possible to perform this successive steps via a graphical user interface.

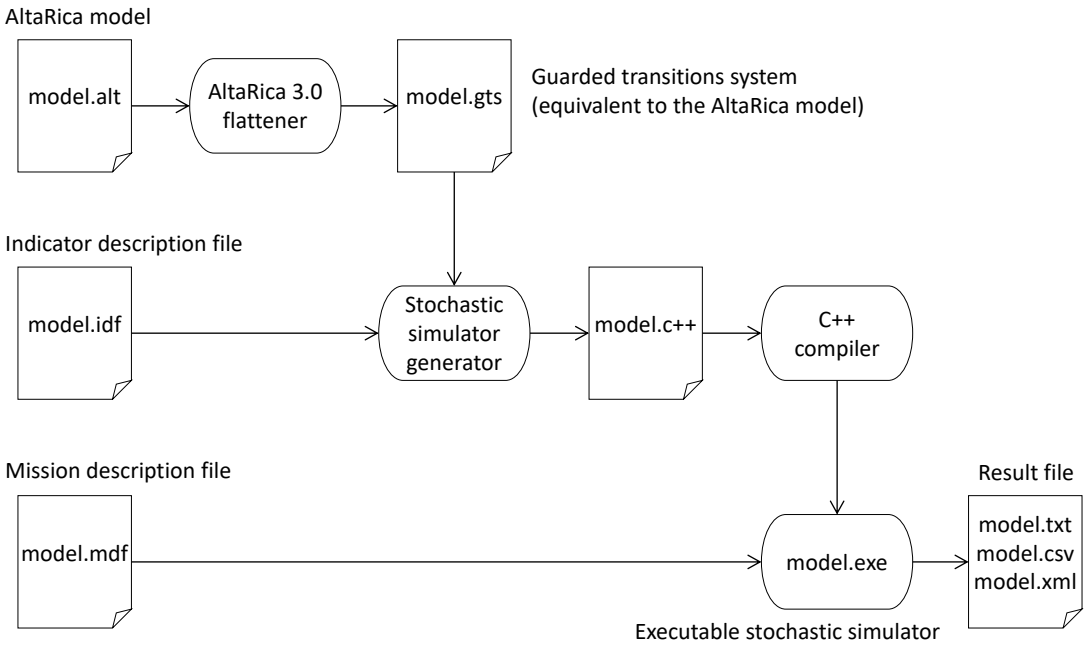


Figure 7.16: Workflow of the AltaRica 3.0 stochastic simulator

7.6.2 Case Study

As an illustration, let us come back to our case study.

Model

This time, we shall represent the behavior of the gas production facility in more details.

Figure 7.17 gives the AltaRica 3.0 model for the different units of the system.

This code is follows the same pattern as previous ones. All units derive from the same model, namely they inherit from the class `RepairableUnit`. Units can be turned on and off depending the demand. They are assumed not to fail when in standby. Moreover, their repair starts as soon as they are failed and they are as good as new after a repair.

The difference with what we have seen so far stands in the flow variables `input` and `output`. These variables are now real (or more exactly floating point numbers). Each unit has a maximum capacity. When working, the output of the unit is the minimum of its input and its capacity.

Note that the variable `output` does not represent the production of each unit. Rather, it represents its maximum production capacity conditioned by what the maximum production it receives as input, its intrinsic capacity and its state.

Figure 7.18 gives the AltaRica 3.0 model for the facility itself.

This code applies at lines' level the same principle as units' level: the variable `output` of the line represents the maximum production capacity of the latter conditioned by what the maximum production it receives as input, the intrinsic capacities and the states of its units.

This code declares two observers:

– A real-valued observer `production` that gives the production level of the facility, i.e. the output of the compressor line.
– A symbolic observer `productionLevel` that takes the value `NULL` when the production is stopped, `LOW` when it is less or equal to 50% (but not null), the value `MEDIUM` when it is in between 50% (excluded) and 80% (included) and finally `HIGH` when it is above 80%.

We added the observer `productionLevel` mostly for the sake of the completeness of the presentation of the stochastic simulator.

```
domain RepairableUnitState {STANDBY, WORKING, FAILED}

class RepairableUnit
    RepairableUnitState _state(init = STANDBY);
    Boolean failed, demand(reset = false);
    event failure(delay = exponential(failureRate));
    event repair(delay = exponential(repairRate));
    event turnOn(delay = Dirac(0));
    event turnOff(delay = Dirac(0));
    parameter Real failureRate = 1.0e-5;
    parameter Real repairRate = 3.0e-1;
    parameter Real capacity = 100;
    Real input, output(reset = 0.0);
    transition
        failure: _state==WORKING -> _state := FAILED;
        repair: _state==FAILED -> _state := STANDBY;
        turnOn: _state==STANDBY and demand -> _state := WORKING;
        turnOff: _state==WORKING and not demand -> _state := STANDBY;
    assertion
        failed := _state==FAILED;
        output := if _state==WORKING then min(input, capacity) else 0.0;
end

class Separator
    extends RepairableUnit(failureRate = 8.50e-5, repairRate = 2.50e-3,
        capacity = 50);
end

class Dehydrator
    extends RepairableUnit(failureRate = 3.00e-5, repairRate = 3.00e-3,
        capacity = 85);
end

class Compressor
    extends RepairableUnit(failureRate = 3.50e-5, repairRate = 5.00e-3,
        capacity = 60);
end
```

Figure 7.17: AltaRica 3.0 code for the units of the gas production facility (numerical version)

Stochastic Simulation

The following performance indicators could typically be of interest, for their safety or economical significance.

- The expected production of the system over the considered time period (say 5 year).
- The expected number of total shutdowns of the system over the considered time period.
- The expected times spent at each production level over the considered time period.

To get the expected production of the system from time 0 to time t, we can use the indicator `mean-value`(`production`, t). It is probably interesting to have a fine grain analysis for this indicator, typically by calculating not only its mean but also a distribution in 10 buckets.

To get the expected number of total shutdowns of the system from time 0 to time t, we can use the indicator `number-of-occurrences`(`productionLevel`, `NULL`, t). Calculating the mean value and the standard-deviation of this indicator is probably sufficient for practical purposes.

```
domain ProductionLevel {NULL, LOW, MEDIUM, HIGH}

block GasProductionFacility
    Real W, P(reset = 0.0);
    block S
        Real input, output(reset = 0.0);
        Separator S1, S2, S3;
        assertion
            S1.demand := true;
            S2.demand := true;
            S3.demand := S1.failed or S2.failed;
            S1.input := input;
            S2.input := input;
            S3.input := input;
            output := min(S1.output + S2.output + S3.output, input);
    end
    block D
        Real input, output(reset = 0.0);
        Dehydrator D1, D2;
        assertion
            D1.demand := true;
            D2.demand := true;
            D1.input := input;
            D2.input := input;
            output := min(D1.output + D2.output, input);
    end
    block C
        Real input, output(reset = 0.0);
        Compressor C1, C2, C3;
        assertion
            C1.demand := true;
            C2.demand := true;
            C3.demand := C1.failed or C2.failed;
            C1.input := input;
            C2.input := input;
            C3.input := input;
            output := min(C1.output + C2.output + C3.output, input);
    end
    assertion
        W := 100.0;
        S.input := W;
        D.input := S.output;
        C.input := D.output;
        P := C.output;
    observer Real production = P;
    observer ProductionLevel productionLevel = switch {
        case P==0.0: NULL
        case P<=50: LOW
        case P<=80: MEDIUM
        default: HIGH
        };
end
```

Figure 7.18: AltaRica 3.0 code for the gas production facility (numerical version)

Finally, to get an idea of the degradation of the system over the time period, we can just use the indicators `sojourn-time`(`productionLevel`, l, t), where l stands for `NULL`, `LOW`, `MEDIUM` and `HIGH` and to calculate them at different intermediate dates.

We may typically want to study the system over a one year period (8760 hours) with an intermediate assessment after six month (4380 hours). Assume finally that we want to run 10 000 executions.

Running the stochastic simulation, we obtain the following results.

The distribution of the production over the one year period is the following.

Bucket	Mean	Lower-bound	Upper-bound
0% – 10%	97.76	89.31	98.69
10% – 20%	99.05	95.76	96.36
20% – 30%	96.55	96.36	96.73
30% – 40%	96.90	96.73	97.05
40% – 50%	97.18	97.05	97.31
50% – 60%	97.44	97.31	97.58
60% – 70%	97.71	97.56	97.85
70% – 80%	98.00	97.85	98.15
80% – 90%	98.34	98.15	98.54
90% – 100%	98.89	98.54	100.00

The mean and the standard deviation of the number of total shutdowns are respectively 4.30 and 1.88 after six months and 8.67 and 2.69 after one year.

Finally, the sojourn-time in a degraded state is 42.04 hours after six months and 71.84 hours after one year.

7.7 Compilation into Systems of Boolean Equations

Originally, AltaRica has been designed considering the compilation of models into fault trees as the main, if not the unique, assessment tool (Rauzy 2002). Still today, it is often used with this perspective in industry.

It may seem strange at a first glance to use a modeling as powerful as AltaRica to eventually end up with fault trees. There are however good reasons to proceed this way. First, AltaRica makes it possible to build complex models and to simulate them interactively, visualizing "what's going on". This alleviates very significantly the task of designing and validating models. Second, the same model can be used to assess several safety goals, i.e. to generate different fault trees. This makes it possible to save time. Third, it is in general much easier, if the specifications of the system under study changed, to report these changes in the AltaRica model than to check by hand multiple fault trees.

It remains that AltaRica 3.0 is much more expressive than systems of Boolean equations. Consequently, not all of AltaRica models can be compiled into systems of Boolean equations, at least without loosing a significant amount of information. It is therefore important to understand the basic principle of the compilation before using this tool. For instance, it is meaningless to compile a model describing an infinite state space into a fault tree.

We shall describe here only this basic principle. The reader interested in a thorough treatment should refer to author's articles (Prosvirnova and Rauzy 2015; Rauzy 2002).

7.7.1 Principle

Roughly speaking, the compilation of an AltaRica 3.0 model into a system of Boolean equations is performed by means of the following steps.

1. The AltaRica model M is flattened into an equivalent guarded transitions system S. This step is actually common to most of the assessment tools.
2. S is decomposed into a set of independent subsystems $B_1, B_2, \ldots B_k$ together with an assertion A, as illustrated in Figure 7.19 by means of static analysis techniques, see e.g. (Rival and K. Yi 2020) for an introduction. A sub-system B_i is independent from a subsystem B_j, $i \neq j$, if the state of B_i has no impact on whether the transitions of B_j are enabled or not, and vice-versa.
3. The reachability graphs of each independent subsystems are built.
4. A Boolean formula $\Phi(s)$ is associated with each state s of these graphs. $\Phi(s)$ is (equivalent to) the disjunction over the paths π that go from the initial state to the state s of the events labeling the transitions of π.
5. A Boolean formula $\kappa(v,c)$ is associated with each variable v and each value c of the domain of v. $\kappa(v,c)$ describes the (global) states in which the variable v takes the value c. $\kappa(v,c)$ is built using the formulas $\Phi(s)$'s calculated at the previous step.

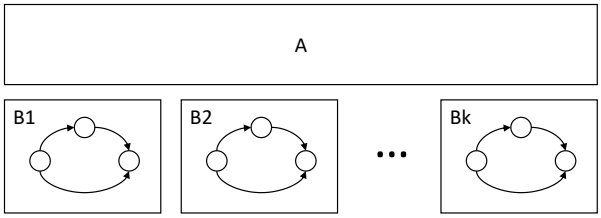


Figure 7.19: Decomposition of a guarded transition system into a set of independent sub-systems

The decomposition of the guarded transition system S into the subsystems $B_1, B_2, \ldots B_k$ and the assertion A is faithful in this sense that composing $B_1, B_2, \ldots B_k$ and A gives back S. The construction of the formulas $\kappa(v,c)$ is also faithful in this sense that it reflects exactly the states s in which the variable v takes the value c. The possible loss of information comes thus from the two intermediate steps:

– The reachability graphs may be only partial in order to fit within reasonable memory size (see (Brameret, Rauzy, and J.-M. Roussel 2015) for a discussion in the context of the compilation of guarded transition systems into Markov chains).
– More importantly, the construction of formulas $\Phi(s)$'s projects sets of executions onto the Boolean algebra, hence losing the order of events in the execution: the execution $\sigma_0 \xrightarrow{e_1} \sigma_2 \cdots \xrightarrow{e_n} \sigma_n$ is compiled into the Boolean formula $e_1 \wedge \ldots \wedge e_n$.

This compilation method is extremely powerful and makes it possible to compile efficiently large and complex models. It has however limitations that must be well understood.

First, it is meaningful only for models in which transitions bring the system from a less degraded state to a more degraded state. In other words, transitions of the models must represent degradations, failures and possibly some reconfigurations, but maintenance operations and repairs cannot be taken into account. The compilation process works even in presence of such "condition-improving" transitions, but these transitions do not influence the final result and decrease significantly its performance.

Second, the compilation process does not assume that transitions are independent one another (although this assumption is made latter on by assessment tools). However, it must be possible to decompose the system into small independent sub-systems. Otherwise, the compilation suffers from the exponential blow-up of reachability graphs (step 3).

Third, in case of dependent events (and transitions), the compilation process may introduce some negations, which means that the generated systems of Boolean equations are not coherent. In many practical cases, the calculated the minimal cutsets are those that are expected by the analyst.

However, he must be aware, that the introduction of dependencies among events goes against one of the basic assumptions of the fault tree and related methods. It is not impossible to do, but this should be done with much care.

7.7.2 Case Study

Model

As an illustration, consider again the model given in Figure 7.9 for the separation line. We can apply the same principle to the compression line. Consequently, separators `S1` and `S2`, dehydrators `D1` and `D2` and compressors `C1` and `C2` are simple non-repairable units, while the separator `S3` and the compressor `C3` are non-repairable, spare units. As discussed above, we could make all these units repairable, but repairs would not be taken into account by the compilation process, but would make it less efficient. Because repair transitions are taken of the model, we can take off turn-off transitions as well.

Figures 7.20 and 7.21 give the corresponding AltaRica model.

Note that we introduced a 2-out-of-3 logic for separator and compressor lines and that we added a Boolean observer `failed` that will be the target of our safety case (the top event of the fault tree).

Now, the transition `S.S3.failure` depends on the state variable `S.S3._state`. The latter depends on the transition `S.S3.turnOn`, which in turn depends on the flow variable `S.S3.demand`. The latter depends on the flow variables `S.S1.output` and `S.S2.output` which themselves depend respectively on the state variables `S.S1._state` and `S.S2._state`. These variables depend respectively on the transitions `S.S1.failure` and `S.S2.failure`. It follows by transitivity that the transition `S.S3.failure` depends on the transition `S.S3.turnOn`, which itself depends on the transitions `S.S1.failure` and `S.S2.failure`.

7.8 Generation of Critical Sequences

7.8.1 Principle

7.8.2 Case Study

7.9 Discussion and Further Readings

7.9.1 Some Perspectives on Discrete Event Systems

The term “discrete event system” has been introduced in the realm of control theory (Cassandras and Lafortune 2008). Computer scientists use to speak rather of “state automata” (Hopcroft, Motwani, and J. D. Ullman 2006). The two concepts are however essentially the same.

It remains that there exist a wide variety of discrete event systems, some very different one another. Any attempt to make a taxonomy of these mathematical frameworks would fail to be exhaustive. Those that can be used to describe the dynamic behavior of complexity technical systems can be classified along three axes:

- Frameworks that assume that the architecture of the system stays the same through its mission versus those that make it possible to represent changes in the architecture.
- Frameworks that describe the behavior of the system as a series of small time steps performed simultaneously by all components versus those that describe the behavior by means of sequences of events involving only part of the components.
- Frameworks that consider that the behavior of the system is deterministic versus those that consider that its behavior is stochastic.

Almost all of the combinations of these three binary choices are possible and have been implemented by some modeling language.

Mathematical frameworks that are the most widely used in systems engineering make the assumption that the architecture of the system, i.e. its number of components and the relations

```
domain MainUnitState {WORKING, FAILED}

class MainUnit
    MainUnitState _state(init = WORKING);
    event failure(delay = exponential(failureRate));
    parameter Real failureRate = 1.0e-5;
    Boolean input, output(reset = false);
    transition
        failure: _state==WORKING -> _state := FAILED;
    assertion
        output := input and _state==WORKING;
end

domain SpareUnitState {STANDBY, WORKING, FAILED}

class SpareUnit
    SpareUnitState _state(init = STANDBY);
    Boolean demand(reset = false);
    event failure(delay = exponential(failureRate));
    event turnOn(delay = Dirac(0));
    parameter Real failureRate = 1.0e-5;
    Boolean input, output(reset = false);
    transition
        failure: _state==WORKING -> _state := FAILED;
        turnOn: _state==STANDBY and demand -> _state := WORKING;
    assertion
        output := input and _state==WORKING;
end

class MainSeparator
    extends MainUnit(failureRate = 8.50e-5);
end

class SpareSeparator
    extends SpareUnit(failureRate = 8.50e-5);
end

class MainDehydrator
    extends MainUnit(failureRate = 3.00e-5);
end

class MainCompressor
    extends MainUnit(failureRate = 3.50e-5);
end

class SpareCompressor
    extends SpareUnit(failureRate = 3.50e-5);
end
```

Figure 7.20: AltaRica 3.0 code for the units of the gas production facility (non-repairable Boolean version)

between these components, stay the same through its mission. As we shall discuss in depth in Chapter 9, there are two main reasons for this *a priori* quite restrictive assumption: first, models

```
block GasProductionFacility
    Boolean W, P(reset = false);
    block S
        Boolean input, output(reset = false);
        MainSeparator S1, S2;
        SpareSeparator S3;
        assertion
            S1.input := input;
            S2.input := input;
            S3.input := input;
            S3.demand := input and (not S1.output or not S2.output);
            output := #(S1.output, S2.output, S3.output) >= 2;
    end
    block D
        Boolean input, output(reset = false);
        MainDehydrator D1, D2;
        assertion
            D1.input := input;
            D2.input := input;
            output := D1.output or D2.output;
    end
    block C
        Boolean input, output(reset = false);
        MainCompressor C1, C2;
        SpareCompressor C3;
        assertion
            C1.input := input;
            C2.input := input;
            C3.input := input;
            C3.demand := input and (not C1.output or not C2.output);
            output := #(C1.output, C2.output, C3.output) >= 2;
    end
    assertion
        W := true;
        S.input := W;
        D.input := S.output;
        C.input := D.output;
        P := C.output;
   observer Boolean failed = not P;
end
```

Figure 7.21: AltaRica 3.0 code for the gas production facility (non-repairable, Boolean version)

involving changes in architecture are much more difficult to debug and to validate; second, the computational complexity of assessments these models is in general much higher. We call systems whose architecture change through their mission *deformable systems*. *Systems of systems* are typical examples of deformable systems (Maier 1998).

Non-deformable systems can be represented by a hierarchy of interacting components. Their states are eventually characterized by means of a fixed set of variables taking their values into basic domains such as Boolean, sets of symbolic constants, integers or reals. Their behavior is described by describing how the values of these variables evolve through the time.

As explained above, this evolution can be described in two ways: either by seeing it as series of small time steps performed simultaneously by all components, or by seeing it as sequences of

events involving only part of the components. Modeling languages implementing the first approach are called *synchronous languages* those implementing the second one are called *asynchronous languages*.

Synchronous languages benefit immediately from what is a the very core of mathematics and physics, namely the *Calculus*, or in other words, *differential equations*. The semantics of these languages can be described directly in terms of systems of differential equations as for Matlab/Simulink (Klee and Allen 2011) or Modelica (Fritzson 2015) and languages supporting *system dynamics* (Sterman 2000). It can also rely on a more abstract view of time, as for instance in Lustre (Halbwachs 1993). These languages are deterministic. Markov chains (Stewart 1994) can also be seen as a basic stochastic synchronous modeling language. Actual modeling languages are built on top of Markov chains, for instance stochastic automaton networks (Plateau and Stewart 1997) or the input language of the PRISM model-checker (Kwiatkowska, Norman, and Parker 2011).

Discrete event systems enter into the category of asynchronous languages. They abstract the time further. They consider that the state of the system under study changes under the occurrence of events, i.e. remains unchanged in between two events. Events are associated with delays that can be either deterministic or stochastic. Petri nets and Harel's state charts (Harel 1987; Harel and Politi 1998), which are embedded into UML (Rumbaugh, Jacobson, and Booch 2005) and SysML (Friedenthal, A. Moore, and Steiner 2011) under the name state machine diagrams, are typical examples of deterministic discrete event systems. Event B (Abrial 2010) can also be seen as a language of this category. Stochastic discrete event systems have been introduced in the realm of control theory (Cassandras and Lafortune 2008; Zimmermann 2010). Among popular modeling formalisms relying on this principle, we can cite stochastic Petri nets (Ajmone-Marsan et al. 1994; Signoret and Leroy 2021) and of course AltaRica.

The choice between synchronous and asynchronous descriptions relies on pragmatic considerations. Synchronous descriptions are closer to the physics of the system. Asynchronous descriptions are more abstract. The author believes that asynchronous models are more suitable in the realm of systems engineering as they correspond better to the required level of abstraction. This said, interesting attempts to embed both types of models into a single framework exist. At a conceptual level, Sifakis's BIP (Behavior, Interaction, Priority) framework (Bliudze and Sifakis 2008) is very seducing. At a tooling level, co-simulation platforms such as Ptolemy (Ptolemaeus 2014) and the so-called Functional MockUp Interface (Modelica Association 2019) could be worth to consider.

7.9.2 Model-Based Risk and Safety Assessment

The promise of the so-called model-based risk and safety assessment approach is to provide analysts with modeling languages that have both a high expressive power and suitable structuring mechanisms. It is possible in this way to design models that reflect the functional and physical architectures of the system under study. Safety models are thus closer to systems specifications, which makes them both easier to share with non-specialists and to maintain. Moreover, a single model can be used to assess several safety goals.

Modeling formalisms that support this approach can be classified into three categories.

The first category consists of specialized profiles of model-based systems engineering formalisms such as SysML, see e.g. (David, Idasiak, and Kratz 2010; Mauborgne et al. 2016; Mhenni et al. 2016; Yakymets, Julho, and Lanusse 2014). The objective here is however more to introduce a safety facet into models of system architecture than to design actual safety models.

The second category consists of extensions of fault trees or reliability block diagrams so to enrich their expressive power. This category includes dynamic fault trees Bouissou and Bon 2003; Dugan, Bavuso, and Boyd 1992, multistate systems Lisnianski and Levitin 2003; Natvig 2010; Papadopoulos et al. 2011, and some other proposals Signoret, Dutuit, et al. 2013. S2ML+SBE, intro-

duced in Chapter 6 enters indeed in this category. S2ML+FDS—FDS stands for finite degradation structures (Rauzy and Yang 2019a; Rauzy and Yang 2019b)—enters also in this category.

The third category, which aims at taking fully advantage of the model-based approach, consists of modeling languages such as SAML Güdemann and Ortmeier 2010, Figaro Bouissou, Bouhadana, et al. 1991 and AltaRica Batteux, Prosvirnova, and Rauzy 2019a. SAML is oriented towards probabilistic model checking (and compiled into PRISM descriptions Kwiatkowska, Norman, and Parker 2011). Figaro has been historically the first modeling language dedicated to probabilistic risk and safety analyses. It is a rule-based systems Bouissou and Houdebine 2002; Bouissou, Humbert, et al. 2002. Some versions of stochastic Petri nets evolve towards the model-based approach by providing structuring constructs (Signoret, Dutuit, et al. 2013; Signoret and Leroy 2021).

Since the very first version(s) of AltaRica (Arnold et al. 2000; Point and Rauzy 1999; Rauzy 2002), the choice has been made to rely on the more natural and mathematically clearer notion of state automata. Guarded transition systems, as defined in author's article (Rauzy 2008b) and later refined in reference (Batteux, Prosvirnova, and Rauzy 2017a), are an important step in this journey. The story is probably not ended yet, as the discovery of finite degradation structures (Rauzy and Yang 2019a; Rauzy and Yang 2019b), is a game changer in the domain. Their seamless integration with guarded transition systems is still to come.

7.10 Exercises and Problems

Exercise 7.1 – Series System. Consider a system consisting of two repairable units `A` and `B` in series. Initially `A` is working and `B` is in standby. If `A` fails, then `B` is put in service while `A` is repaired. `B` stays in operation until fails. `A` is then put in operation (if possible) and so on.

Assume finally that failures and repairs of `A` and `B` are exponentially distributed with failure rate $1.23 \times 10^{-4}\ h^{-1}$ and $2.34 \times 10^{-2}\ h{-}1$.

Question 1. Design an AltaRica model for this system. Check it using the interactive simulator.

Question 2. Propose a possible timed executions of the system.

Question 3. Assess the unavailability and the unreliability of the system each month, over a one year period using the stochastic simulator.

■

Exercise 7.2 – Alternating Units. Consider a system consisting of two repairable units `A` and `B`. Initially `A` is working and `B` is in standby. If `A` fails, then `B` is put in service while `A` is repaired. `B` stays in operation until fails. `A` is then put in operation (if possible) and so on.

Assume finally that failures and repairs of `A` and `B` are exponentially distributed with failure rate $1.23 \times 10^{-4}\ h^{-1}$ and $2.34 \times 10^{-2}\ h{-}1$.

Question 1. Design an AltaRica model for this system. Check it using the interactive simulator.

Question 2. Propose a possible timed executions of the system.

Question 3. Assess the availability and the reliability of the system each month, over a one year period using the stochastic simulator.

■

Exercise 7.3 – Simple Production System. Consider the simple production system pictured in Figure 7.22. It consists of two independent units `A` and `B` in series. `A` and `B` may degrade, then fail.

Once failed they can be repaired. We can assume, without a loss of generality, that degradations, failures and repairs are exponentially distributed with degradation, failure and repair rates given in the following table.

Unit	Degradation rate	Failure rate	Repair rate
A	$1.00 \times 10^{-3}\ h^{-1}$	$1.00 \times 10^{-2}\ h^{-1}$	$1.00 \times 10^{-2}\ h^{-1}$
B	$1.00 \times 10^{-4}\ h^{-1}$	$5.00 \times 10^{-2}\ h^{-1}$	$2.00 \times 10^{-2}\ h^{-1}$

As the two units are in series, if one fails the other one is put in standby. It is assumed that units cannot degrade nor fail when in standby.

Units produce at 100% of their capacities when they are working and at 0% of their capacities when they are failed. `A` produces at 80% of its capacity if degraded. `B` produces at 70% of its capacity if degraded. As `A` and `B` work in series, the production of the plant is as indicated in the following table.

A\B	WORKING	DEGRADED	FAILED
WORKING	100%	70%	0%
DEGRADED	80%	70%	0%
FAILED	0%	0%	0%

Question 1. Design an AltaRica model for this system. Check it using the interactive simulator.

Question 2. Using the stochastic simulator, assess:

(a) The expected number of failures per year.

(b) The average time spent in a degraded state per year.

(c) The expected production of the system each month, over a one year period .

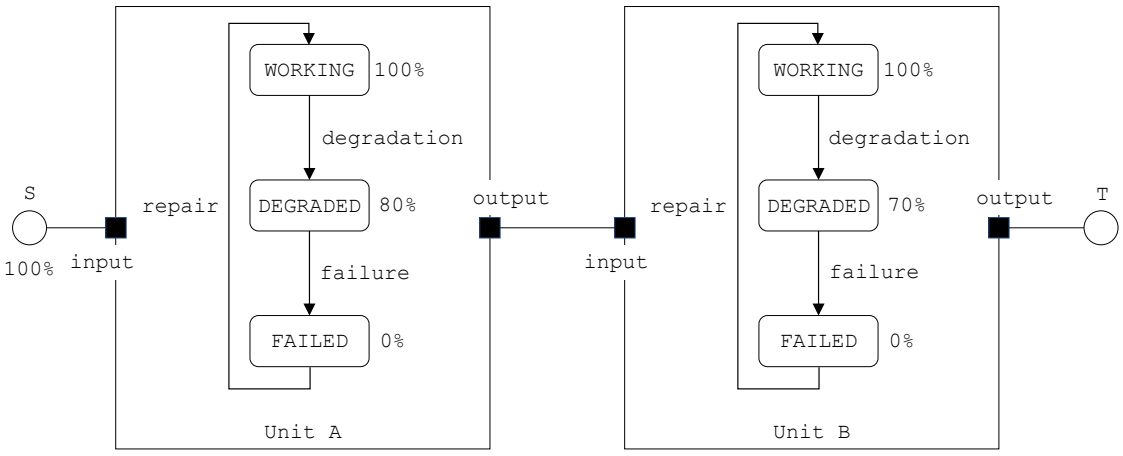


Figure 7.22: A simple production system

Problem 7.4 – Data Treatment System. Consider the hierarchical block diagram pictured in Figure 7.23. This diagram represents an embedded data treatment system as found for instance in airplanes. The systems works as follows. Data are acquired from two different sources by units `A1` and `A2`. They are then processed and cross-checked by processing units `B1` and `B2`. Finally, units `C1` and `C2` elaborate some commands which are then send out via a single output channel.

For physical reasons, the system is decomposed into two subsystems: `AB` containing the units `A1`, `A2` and `B1` and `B2`, and the subsystem `C` containing units `C1` and `C2`. All units may fail and cannot be repaired during the flight.

Assume that they fail according to Weibull distributions with the following parameters:

Unit	Scale parameter	Shape parameter
Acquisition units `A1` and `A2`	$1.00 \times 10^6\ h$	5
Processing units `B1` and `B2`	$5.00 \times 10^7\ h$	2
Command units `B1` and `B2`	$5.00 \times 10^7\ h$	3

Question 1. Design an AltaRica model for the data treatment system. Check that your model is correct by using the interactive simulator.

Question 2. Compile your model into a fault tree at Open-PSA format. Use XFTA to extract its minimal cutsets and calculate its top event probability for a two hours mission time.

Question 3. Use the stochastic simulator to calculate its top event probability for a two hours mission time.

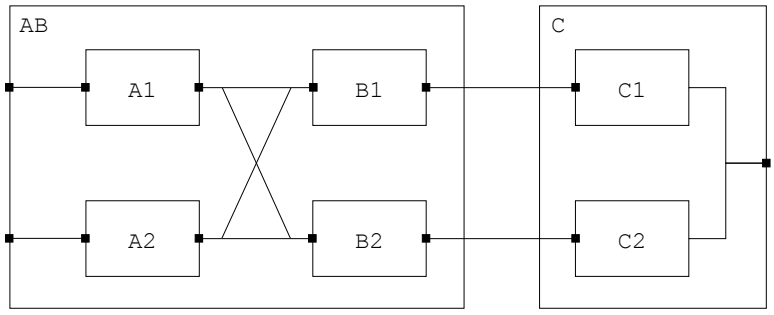


Figure 7.23: An embedded data treatment system

Problem 7.5 – Data Treatment System (revisited). Consider again the data treatment system pictured in Figure 7.23. Assume now that units may degrade or fail and that the signal circulating between units can be either correct, erroneous or lost. The transfer function of units is a follows.

- If the unit is working and its input signal is correct, then its output signal is correct.
- If the unit is failed or its input signal is lost, then its output signal is lost.
- Otherwise, its output signal is erroneous.

Similarly, if two signals are combined in parallel, the result is as follows.

- If one of the input signals is correct, then the output signal is correct.
- If both input signals are lost, then the output signal is lost.
- Otherwise, the output signal is erroneous.

Question 1. Design an AltaRica model for this revisited version of the data treatment system. Check that your model is correct by using the interactive simulator.

Question 2. Compile your model into a fault tree at Open-PSA format. Use XFTA to extract its minimal cutsets and calculate its top event probability for a given mission time (select some realistic degradation and failure rate).

Question 3. Use the stochastic simulator to calculate its top event probability for a given mission time.

■

Exercise 7.6 – 2-out-of-3 System. Consider a system consisting of three non-repairable units `A`, `B` and `C` working according to a 2-out-of-3 logic, i.e. the system works if at least 2 out of the 3 units work.

Assume that failures of units are exponentially distributed with failure rate $1.23 \times 10^{-4}\ h^{-1}$.

Question 1. Design an AltaRica model for this system. Check it using the interactive simulator.

Question 2. Compile your model into a system of Boolean equations. Calculate the minimal cutsets and the top event probability at $t = 8760$ with XFTA.

Question 3. Compile your model into a system of Boolean equations. Assess the top event probability at $t = 8760$ with the stochastic simulator. Compare the result with the one obtained via the compilation into a system of Boolean equations.

■

Exercise 7.7 – Cold Redundancy (fault tree). Consider a system consisting of a main unit `A` and a spare unit `B`. Assume that both units are non repairable and that their failures are exponentially distributed with failure rate $1.23 \times 10^{-4}\ h^{-1}$.

Question 1. Design an AltaRica model for this system. Check it using the interactive simulator.

Question 2. Compile your model into a system of Boolean equations. Calculate the minimal cutsets and the top event probability at $t = 8760$ with XFTA.

Question 3. Compile your model into a system of Boolean equations. Assess the top event probability at $t = 8760$ with the stochastic simulator. Compare the result with the one obtained via the compilation into a system of Boolean equations.

■

Exercise 7.8 – Fault Trees. Consider the fault tree pictured in Figure 7.24. Assume that all components of this fault tree are repairable and that their failures and repairs are exponentially distributed (you shall choose their failure and repair rates).

Question 1. Design an AltaRica model for this fault tree. Check that your model is correct by using the interactive simulator.

Question 2. Compile your model into a fault tree at Open-PSA format. Use XFTA to extract its minimal cutsets and calculate its top event probability for a given mission time.

Question 3. Use the stochastic simulator to calculate its top event probability for a one year mission time.

■

Exercise 7.9 – Exclusive Failures. Consider a system consisting of two units `A` and `B`. The unit `A` is non-repairable. Its failure is exponentially distributed with failure rate $1.23 \times 10^{-4}\ h^{-1}$. The unit `B` is also non repairable. It has however two exclusive failure modes: `failureLeft` and

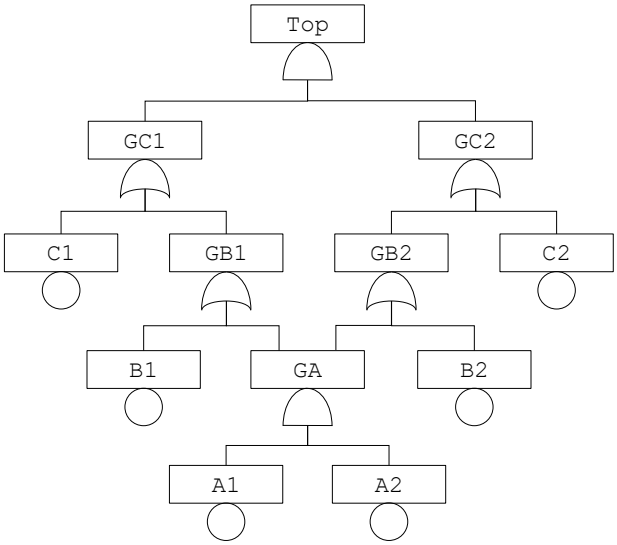


Figure 7.24: A fault tree

`failureRight`. Both failures are exponentially distributed with failure rate $1.23 \times 10^{-4}\ h^{-1}$. `B` has two output channels, left and right. When it fails on left, it ceases to emit on the left output channel. Similarly, it fails on right, it ceases to emit on the right output channel. The system is working if it emits on at least 2 out the 3 output channels `A.output`, `B.outputLeft` and `B.outputRight`

Question 1. Design an AltaRica model for this system. Check it using the interactive simulator.

Question 2. Compile your model into a system of Boolean equations. Calculate the minimal cutsets and the top event probability at $t = 8760$ with XFTA.

Question 3. Compile your model into a system of Boolean equations. Assess the top event probability at $t = 8760$ with the stochastic simulator. Compare the result with the one obtained via the compilation into a system of Boolean equations.

■

Exercise 7.10 – Cross-Checking Units. Consider a system consisting of two embedded data treatment units `A` and `B`. When working, both units emit a correct output signal, assuming the receive a correct input signal. The units may however fail in which case they emit an erroneous signal. Their failures are exponentially distributed with failure rate $1.23 \times 10^{-4}\ h^{-1}$. Units are cross-checking their output signals. If `A` is working normally and it detects that the output signal of `B` is erroneous, it sends a turn off order to `B`. `B` ceases then to emit. And vice-versa. We shall assume that the detection and the emission of the turn off order are instantaneous.

If any of the two units emits an erroneous signal, then so does the system, which is the dangerous situation. Otherwise, if one of the units emits a correct signal, then so does the system.

Question 1. Design an AltaRica model for this system. Check it using the interactive simulator.

Question 2. Compile your model into a system of Boolean equations. Calculate the minimal cutsets and the top event probability at $t = 8760$ with XFTA.

Question 3. Compile your model into a system of Boolean equations. Assess the top event

probability at $t = 8760$ with the stochastic simulator. Compare the result with the one obtained via the compilation into a system of Boolean equations.

■

Exercise 7.11 – Watchdog. Consider a system consisting of two embedded data treatment units `A` and `B` in series. When working, both units re-emit the signal they receive as input:

- If this signal is correct, they emit a correct signal.
- If this signal is erroneous, they emit an erroneous signal.
- Finally, if their input signal is lost, they do not emit any signal.

The units may however fail in which case they emit an erroneous signal if they receive any signal in input. Their failures are exponentially distributed with respective failure rate $1.23 \times 10^{-4}\ h^{-1}$ and $5.67 \times 10^{-6}\ h^{-1}$.

To minimize the risk, a watchdog `W` is installed on the output signal of `A`. If `W` detects that the output signal of `A` is erroneous, it sends immediately a turn of order to `B`. When `B` is turned off, it does not emit any signal. `W` may however fail. Its failure is exponentially distributed with failure rate $3.45 \times 10^{-2}\ h^{-1}$.

The dangerous situation is when the output of the system, i.e. the output of `B` is erroneous.

Question 1. Design an AltaRica model for this system. Check it using the interactive simulator.

Question 2. Compile your model into a system of Boolean equations. Calculate the minimal cutsets and the top event probability at $t = 8760$ with XFTA.

Question 3. Compile your model into a system of Boolean equations. Assess the top event probability at $t = 8760$ with the stochastic simulator. Compare the result with the one obtained via the compilation into a system of Boolean equations.

■

Problem 7.12 – Pumping System. Consider the pumping system pictured in Figure 7.25. This system is made of two main pumps `MP1`, `MP2` and one spare pump `SP` that may fail and be repaired.

In regular operations, both main pumps `MP1` and `MP2` are in action and the spare pump `SP` is in standby. `MP1` and `MP2` supply each 50% of the demand.

If either `MP1` or `MP2` fails, `SP` is put in action. The remaining main pump is then operated at a higher speed and covers 60% of the demand, while `SP` provides 30% of the demand. The pumping capacity is thus degraded as only 90% of the demand is covered.

If either the remaining main pump fails or the spare pump fails, the production must be stopped as the pumping capacity would be too low. The failed pumps are then repaired.

We assume that failures and repairs of all the pumps are exponentially distributed. For main pumps, we have two failure rates: λ_{Mrs} when they are used at regular speed ("rs" stands for regular speed) and λ_{Mhs} when they are used at high speed ("rs" stands for high speed). Of course, λ_{Mhs} is slightly bigger λ_{Mrs}. The failure rate of the spare pump is denoted λ_S.

The repair rate in case the two main pumps are failed is μ_{MM}, while the repair rate in case one of the main pump is failed and the spare pump is failed in μ_{MS}. These rates are not directly given. It is actually easier to obtained them by measuring the mean time to repair a main pump of the spare pump, denoted respectively $MTTR_M$ and $MTTR_S$. The repair rates μ_{MM} and μ_{MS} can be defined

as follows, assuming that pumps are repaired one after the other.

$$\mu_{MM} = \frac{1}{2 \times MTTR_M}$$
$$\mu_{MS} = \frac{1}{MTTR_M + MTTR_S}$$

The following table summarizes reliability data of the system.

Parameter	Value
λ_{Mrs}	$1.00 \times 10^{-4}\ h^{-1}$
λ_{Mhs}	$3.00 \times 10^{-4}\ h^{-1}$
λ_S	$1.00 \times 10^{-2}\ h^{-1}$
$MTTR_M$	$2.40 \times 10^{1}\ h$
$MTTR_S$	$2.00\ h$

Question 1. Draw the Markov chain describing the behavior of the system.

Question 2. Design an AltaRica model for this Markov chain. Check that your model is correct by using the interactive simulator.

Question 3. Use the stochastic simulator to calculate its production of the system every month for a one year mission time.

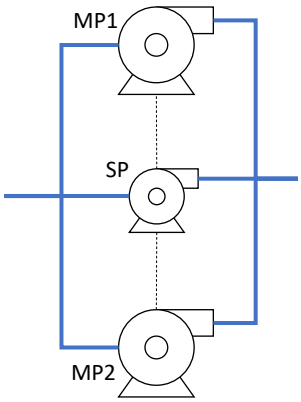


Figure 7.25: A pumping system

Problem 7.13 – Gas Production Facility: Symbolic Model. The objective of this problem is to use operators and functions.

Let us consider again the gas production facility. Assume now that we consider three production levels: none, low and high.

Each individual unit can only produce at low level. Consequently, the input of a unit must be either none or low. It produces at low level if it receives a low level production as input and it is working. Otherwise, it does not produce anything.

A high level production is achieved by a line if it receives a high level production as input and it two of its units are working together.

In each line, the production must be dispatched in priority to first unit, then to the second one,

and eventually to the third one (if any). Units that does not produce anything (that receive no production has input) must be immediately turned off.

Question 1. Design AltaRica operators that make it possible to add up the productions of respectively two and three units.

Question 2. Design AltaRica functions that make it possible to dispatch an input production onto respectively two and three units, taking into account the state of the units.

Question 3. Using the operators and functions designed in the previous section, design an AltaRica model for the gas production facility hence specified. Check that your model is correct by using the interactive simulator.

Question 4. Use the stochastic simulator to assess the respective expected sojourn times at each production level, over a one year mission time.

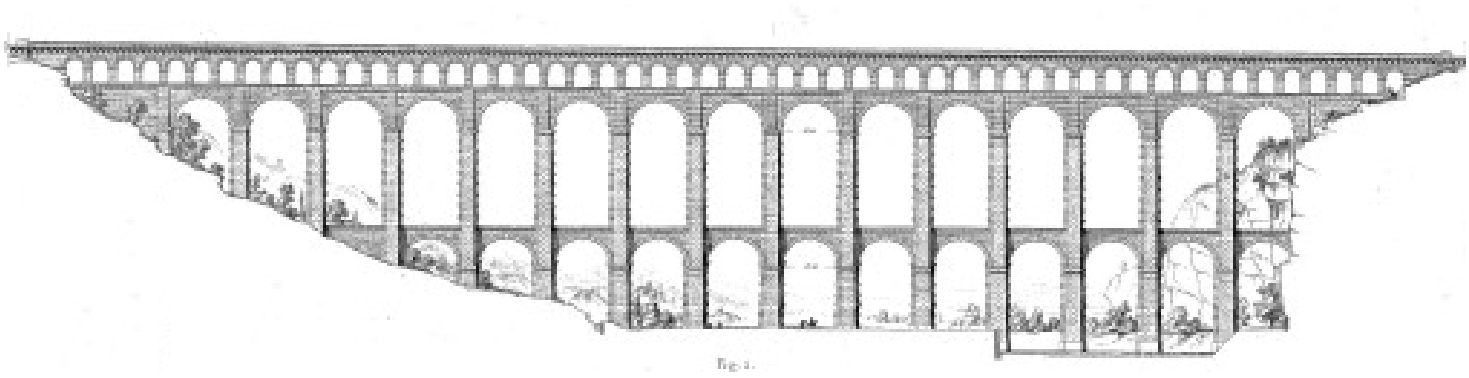
■

Exercise 7.14 – Shared Resource. Consider an offshore subsea system consisting, among other components, of three pumps. The pumps are in hot redundancy but only two of them are required to provide the required pumping capacity. Indeed, these pumps may fail. Their failures are exponentially distributed with a failure rate of $1.23 \times 10^{-4}\ h^{-1}$. When a pump fails, a remotely operated vehicle (ROV) is sent on the seabed to replace it. It takes however 1 month (730 hours) from the time the decision is made to send the ROV from the time the ROV arrives on the spot. Once on the seabed, the ROV replaces the failed pumps and makes the necessary tests before the failed pumps can be put back in operations. No matter the number of failed pumps, it takes 8 hours, once on the spot, to send the ROV on the seabed, replace the failed pumps, test them and put them back in service.

Question 1. Design an AltaRica model for this system. Consider the ROV as a shared resource and use synchronizations to represent actions involving it. Check that your model is correct by using the interactive simulator.

Question 2. Use the stochastic simulator to assess the mean down time of the system over a 5 years period.

■



8. Modeling Patterns

Key Concepts
- Modeling Patterns
- Repairable Components
- Degradable Components
- Exclusive Failures
- Periodically Maintained Components
- Cold and Warm Redundancies
- Shared Resources
- Common Cause Failures
- Broadcast
- Dependent Failures
- Cascading Failures
- Fail-Safe Design
- Block Diagram
- Tree-Like Breakdown
- Monitored Systems

This chapter comes in complement to the previous one. It discusses *modeling patterns*, i.e. prototypical examples of modeling elements that can be used to represent the behavior of complex technical systems. Modeling patterns are a tool to be more efficient in the design, the validation and the maintenance of models. They should be used in combination with a more global approach to study systems, such as the Cube framework presented in the first part of this book.

8.1 Rational

As pointed out in the previous chapter, AltaRica 3.0 combines guarded transition systems with S2ML. Guarded transition system have been designed to increase as much as possible the expressive power of the language without increasing the computational cost of assessment algorithms. This combination results in a powerful, versatile language which exploits in an optimum way assessment

algorithms.

Having a powerful language and efficient assessment algorithms is however not sufficient to make the modeling process efficient. To make it efficient, one needs in addition a methodology to design, document and maintain models. With that respect, it is of primary importance to be able to reuse as much as possible modeling components within models and between models. In languages such as Modelica (Fritzson 2015), this goal is achieved via the design of libraries of on-the-shelf ready-to-use modeling components. Reusing components is also possible in probabilistic risk and safety analyses, but to a much lesser extent. The reason is that these analyses represent systems at a high level of abstraction. Modeling components, except for very basic ones, tend thus to be specific to each system. In AltaRica 3.0, reuse is mostly achieved by the design of modeling patterns, i.e. examples of models representing remarkable features of the system under study. Once identified, patterns can be duplicated and adjusted for specific needs, see e.g. references (Kehren et al. 2004; Kloul and Rauzy 2017) for preliminary studies. Patterns are pervasive in engineering. They have been developed for instance in the field of technical system architecture (Maier 2009), as well as in software engineering (Gamma et al. 1994). Patterns are not only a mean to organize and to document models, but also and more fundamentally a way to reason about systems under study.

It is probably too early to design a taxonomy of modeling patterns encountered in reliability analyses. For the time being, we can classify patterns into two categories: behavioral patterns that aim at describing the behavior of a single component or a small group of components, and architectural patterns that aim at describing the whole model organization.

This chapter presents patterns that are frequently used in the context of AltaRica 3.0.

8.2 Fundamental Behavioral Patterns

This first section presents some key behavioral patterns.

8.2.1 Repairable Components

The most basic behavioral pattern is indeed the one for repairable (and non-repairable) components. Figure 7.2 of the previous chapter shows the state automaton for this pattern. Figure 7.3 gives the corresponding AltaRica 3.0 model.

We discussed the REPAIRABLE COMPONENT pattern at length in the previous chapter, so we shall not extend the discussion further here.

Figure 7.5 shows an extension of this pattern, which we can call the DEGRADABLE COMPONENT pattern. Exercise 8.1 requires to extend this pattern to distinguish safe and dangerous failures, which gives raise to the EXCLUSIVE FAILURE pattern.

8.2.2 Periodically Maintained Components

In many technical systems, some components are periodically inspected and maintained. Failures are detected during these inspections. Figure 8.1 shows the state automaton describing a generic periodically maintained component.

The AltaRica code for this state automaton is given in Figure 8.2.

It is often convenient to use several variables to describe the state of a component. In our example, the state of the component is described by means of a pair consisting of the variable `_state` that represents the intrinsic state of the component (`WORKING` or `FAILED`) and a variable `_mode` that represents its operation mode (`OPERATION` or `MAINTENANCE`). It is assumed that the component can only fail when in operation and can be only repaired by means of maintenance operation. The component is assumed to be as good as new after a repair.

In the state ⟨`WORKING`, `OPERATION`⟩, the two transitions `failure` and `startMaintenance` are in *competition*: in some cases, `failure` is fired first, in some others, it is `startMaintenance`

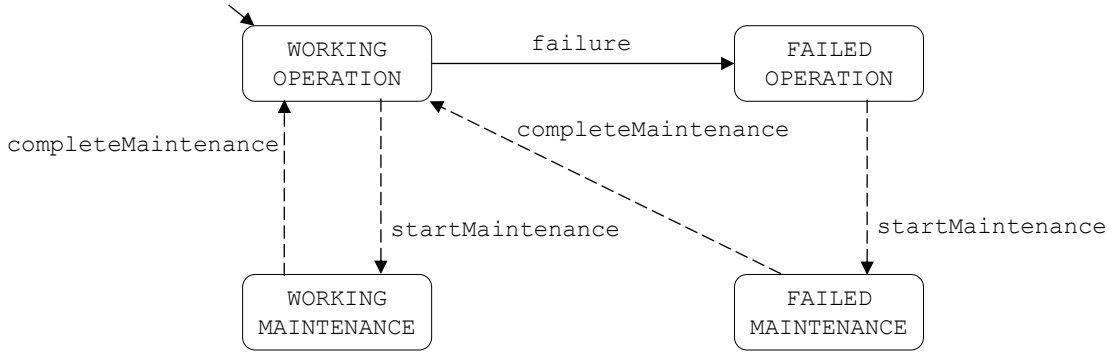


Figure 8.1: State automaton describing a periodically maintained component

```
domain PeriodicallyMaintainedComponentState {WORKING, FAILED}
domain PeriodicallyMaintainedComponentMode {OPERATION, MAINTENANCE}

class PeriodicallyMaintainedComponent
    PeriodicallyMaintainedComponentState _state(init = WORKING);
    PeriodicallyMaintainedComponentMode _mode(init = OPERATION);
    event failure(delay = exponential(1.0e-4));
    event startMaintenance(delay = Dirac(712));
    event completeMaintenance(delay = Dirac(8));
    transition
        failure: _state==WORKING -> _state := FAILED;
        startMaintenance: _mode==OPERATION -> _mode := MAINTENANCE;
        completeMaintenance: _mode==MAINTENANCE -> {
            _state := WORKING; _mode := OPERATION;
            }
end
```

Figure 8.2: The AltaRica code for the PERIODICALLY MAINTAINED COMPONENT pattern

that is fired first. The firing of `failure` does not prevent the firing of `startMaintenance`: if component fails while in operation, it must wait the next maintenance operation to be repaired. On the contrary, the firing of `startMaintenance` cancels the possibility to fire the transition `failure`, as it is assumed that the component cannot fail during maintenance.

Note that in the model of Figure 8.2, we associated an exponential delay with the transition `failure` and deterministic delays with transitions `startMaintenance` and `completeMaintenance`. Of course, other choices can be made, e.g. associate a Weibull distribution with the transition `failure` and an exponential delay with the transition `completeMaintenance`.

Note that the time taken to mobilize the repair crew, i.e. the delay associated with the transition `startMaintenance`, may be dependent of factors that are independent of the component (but dependent on the repair crew). Problem 8.2 discusses an implementation of the PERIODICALLY MAINTAINED COMPONENT pattern presenting such a feature.

8.2.3 Cold and Warm Redundancies

Combinatorial models can only approximate cold and warm redundancies. As an illustration, consider the system pictured in Figure 8.3. It consists of two units `A` and `B`, with `B` in cold/warm redundancy.

A unit `B` is said in *cold redundancy* for another unit `A`, if i) `B` is started, or at least attempted to start, when `A` fails and ii) `B` cannot fail while in standby. *Warm redundancies* are similar to cold

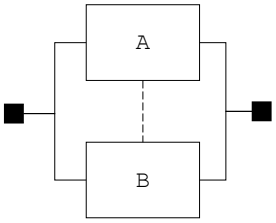


Figure 8.3: Two units in parallel with unit B in cold/warm redundancy

redundancies except that B may fail while in standby, although its probability of failure is in general much lower than if it was operated. Finally, one speaks of *hot redundancy* when the two units are operated in parallel.

Figure 8.4 shows the state automaton representing a unit in warm redundancy.

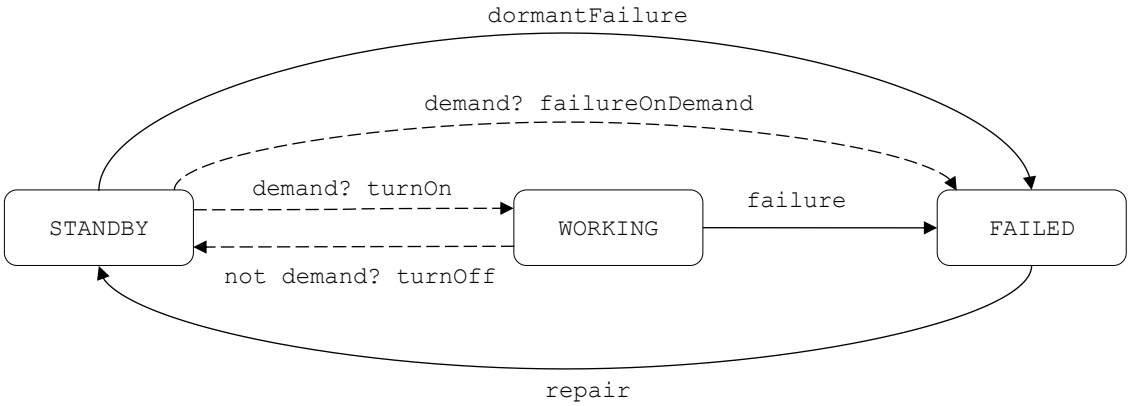


Figure 8.4: The state automaton representing a spare unit in warm redundancy

The unit can be into three states STANDBY, WORKING and FAILED. Initially, it is in standby. It may fail in three different circumstances. First, while in standby. This first type of failure is called a *dormant failure* as it will be probably not detected until the unit is attempted to start or maintained. Second, while working. This second type of failure is in some sense a regular failure. Third, when attempted to start. This third type of failure is called a *failure on demand*.

Dormant and regular failures are associated with stochastic delays. These delays obey typically exponential distributions. In this case, the failure rate of the dormant failure, often denoted by $\lambda^\star$, is in general much lower than the failure rate of the regular failure, traditionally denoted by λ.

On the contrary, the failure on demand is associated with a null delay, but a probability to occur. When attempted to start, i.e. when demanded, the unit takes immediately one of the two transitions turnOn and failureOnDemand. Of course, the transition turnOn has hopefully a much higher probability than the transition failureOnDemand. Transitions turnOn and failureOnDemand are *exclusive*: if one is fired, the other is not enabled anymore.

Figure 8.5 shows an implementation of the WARM REDUNDANCY pattern.

The class RepairableUnit implements a repairable unit. The class SpareUnit implements the spare unit as described in Figure 8.4. The implementation of these two classes is similar to what we have seen so far.

The attribute expectation is used to defined the respective probability of transitions turnOn and failureOnDemand. More exactly, when several transitions $t_1,\ldots t_n$ are enabled and must be fired at the same date, this attribute is used to pick up at random one of these transitions. The probability to fire the transition t_i is then $w(t_i)/\sum_{j=1}^{n} w(t_j)$, where the $w(t_i)$'s are the values of the attribute expectation of the events labeling the transitions. In our example, we could have

```
domain SpareUnitState {STANDBY, WORKING, FAILED}

class SpareUnit
    SpareUnitState _state(init = STANDBY);
    Boolean demand, input, output(reset = false);
    event failure(delay = exponential(1.0e-3));
    event dormantFailure(delay = exponential(1.0e-5));
    event turnOn(delay = Dirac(0), expectation=0.98);
    event failureOnDemand(delay = Dirac(0), expectation=0.02);
    event turnOff(delay = Dirac(0));
    event repair(delay = exponential(1.0e-1));
    transition
        dormantFailure: _state==STANDBY -> _state := FAILED;
        turnOn: _state==STANDBY and demand -> _state := WORKING;
        failureOnDemand: _state==STANDBY and demand -> _state := FAILED;
        turnOff: _state==WORKING and not demand -> _state := STANDBY;
        failure: _state==WORKING -> _state := FAILED;
        repair: _state==FAILED -> _state := STANDBY;
    assertion
        output := _state==WORKING and input;
end

block System
    RepairableUnit A;
    SpareUnit B;
    Boolean output(reset = false);
    assertion
        A.input := true;
        B.input := true;
        B.demand := not A.output;
        output := A.output or B.output;
    observer Boolean failed = not output;
end
```

Figure 8.5: The AltaRica code for the WARM REDUNDANCY pattern

given respectively the values 98 and 2 to the attributes `expectation` of the events `turnOn` and `failureOnDemand` with the same result.

Finally, the block `System` puts the two units `A` and `B` in cold/warm redundancy.

8.3 Behavioral Patterns Involving Synchronizations

This section presents key modeling patterns involving synchronizations.

8.3.1 Shared Resources

It is often the case, in technical systems, that a group of components or sub-systems share some resources, i.e. another group of components or sub-systems. It can be for instance the computers of a network sharing a pool of printers, the units of a plant sharing a pool of repairmen, or the machines of a facility sharing a pool of spare parts.

To represent represent resource sharing, we need the following elements.

– The action by which of the component `C` asks for a resource.
– The action by which the resource `R` is allocated to the component `C`, which means *a priori* that `R` becomes unavailable for the other components.

– The action by which the component C releases the resource R, which means, again *a priori*, that R becomes available again for the other components (and for C itself).

Now, there is a difficulty: how to represent allocation policies?

Consider for instance a group of 4 offshore platforms P1, P2, P3 and P4 sharing two remotely operated vehicles (ROV) ROV1 and ROV2 that are used to inspect installations on the seabed. Now consider the following use case.

1. The platform P1 requests a ROV and obtains it. Say it obtains ROV1.
2. The platform P2 requests a ROV and obtains the only one that is available, i.e. ROV2.
3. The platform P3 requests a ROV and is put on hold, as both ROVs are already allocated.
4. The platform P4 requests a ROV and is put on hold, as both ROVs are already allocated.
5. The platform P1 releases the ROV ROV1.

This use case raises three questions.

First, at step 1, which ROV must be allocated to P1? If ROV1 and ROV2 are perfectly similar, it does not matter. But it may be the case that ROV1 should be allocated preferably, or that ROV1 and ROV2 should be allocated in alternation.

Second, after step 5, to which platform, out of P3 and P4, must be allocated the available ROV, i.e. ROV1? The policy can be to allocate it to the first requester, here P3, or at random, or according to some other criterion.

Now, assume the following continuation of our use case.

6. P3 obtains the ROV ROV1.
7. P1 requests a ROV.
8. P2 releases the ROV R2.

Now, both platforms P1 and P4 are requesting a ROV. Here there is an available ROV: ROV2. If the allocation policy is such that P1 obtains it, we may enter into scenarios in which P4 waits forever or a very long time a ROV. (P1 obtains ROV2, P3 requests a ROV, P1 releases ROV2, P3 gets it, P1 requests a ROV, P3 releases ROV2, P1 gets its and so on).

A *fair execution* is an execution in which a component that requests a resource will eventually get it. Fairness is a important concept in automata theory and practical computer systems, see e.g. (Baier and Katoen 2008).

Representing faithfully allocation policies and ensuring fairness of executions is notoriously difficult. There is no simple solution to do it in AltaRica (nor in any other similar modeling framework). As a rule of thumb, avoid to do it. Approximations are in general good enough.

In our example, we can assume that:

– ROV1 and ROV2 are indistinguishable.
– If two or more platforms are in competition to obtain a ROV, the ROV is allocated at random.

This may be only an approximation of the actual allocation policy, but it is simple and in many cases accurate enough.

We have already seen in Section 7.4.2 an implementation of the SHARED RESOURCE pattern. We can apply rather directly to our problem:

– Platforms are represented like the compressors in the model given in Figure 7.10.
– ROVs are represented as the maintenance team of this model, but the state of the resource (here the ROVs) is a counter of the number of resources available.
– The allocation of a ROV to a platform and the release of this resource by this platform are represented by means of synchronization.

The corresponding model is given in Figure 8.6.

Synchronizations might be replaced by a series of Dirac(0) transitions, like in the WARM REDUNDANCY pattern. However, this proves to be rather tedious to implement.

```
domain PlatformState {WORKING, REQUEST_ROV, USE_ROV}

class Platform
    PlatformState _state(init=WORKING);
    event requestROV(delay = exponential(1.0e-3));
    event getROV, releaseROV(hidden = true);
    transition
        requestROV: _state==WORKING -> _state:=REQUEST_ROV;
        getROV: _state==REQUEST_ROV -> _state:=USE_ROV;
        releaseROV: _state==USE_ROV -> _state:=WORKING;
end

class RemotelyOperatedVehicle
    Integer _count(init=numberOfVehicles);
    parameter Integer numberOfVehicles = 1;
    event allocation, release(hidden = true);
    transition
        allocation: _count>0 -> _count:=_count-1;
        release: true -> _count := _count+1;
end

block System
    Platform P1, P2, P3, P4;
    RemotelyOperatedVehicle ROV(numberOfVehicles = 2);
    event allocation(delay = Dirac(0));
    event release(delay = uniform(12, 24));
    transition
        allocation: !P1.getROV & !ROV.allocation;
        allocation: !P2.getROV & !ROV.allocation;
        allocation: !P3.getROV & !ROV.allocation;
        allocation: !P4.getROV & !ROV.allocation;
        release: !P1.releaseROV & !ROV.release;
        release: !P2.releaseROV & !ROV.release;
        release: !P3.releaseROV & !ROV.release;
        release: !P4.releaseROV & !ROV.release;
end
```

Figure 8.6: AltaRica 3.0 code for shared remotely operated vehicles (SHARED RESOURCE pattern)

8.3.2 Common Cause Failures

We saw in Section 7.4.2 that synchronizations can be used to represent common cause failures. As already pointed out, common cause failures are an important contributor to the risk. Avionic safety standards (*Guidelines for Development of Civil Aircraft and Systems* 2010) and (*Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment* 2004) require to perform a specific activity, so-called common cause analysis to detect and to study them.

The COMMON CAUSE FAILURE pattern is presented in Figure 7.11. Consequently, we shall not come back on it here.

The BROADCAST pattern is a slight variation on the COMMON CAUSE FAILURE pattern. This pattern relies on the idea that one or more sub-systems "emit", their contribution is thus mandatory, while some others "receive", their contribution is thus optional.

As an illustration, consider the electric circuit pictured in Figure 8.7. The electric source `S` powers the devices `D1`, `D2` and `D3`. To prevent damages to these devices in case of a power surge,

a varistor V is installed.

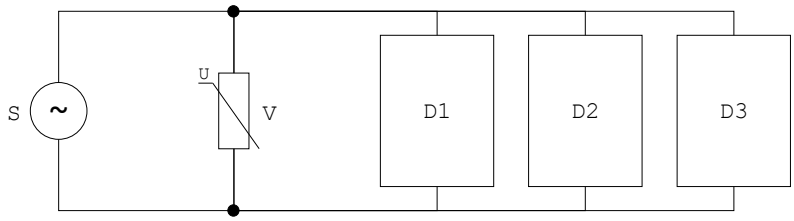


Figure 8.7: An electric circuit

Now, assume that the varistor V itself may be damaged while the system is in operation. In other words, V can experience a dormant failure. It case of an electric surge coming from the S, the protected devices D1, D2 and D3... are not protected anymore. Consequently, they fail, if they are not already failed. In other words, they can experience a common cause failure, the electric surge coming from the source S, under the condition that the varistor V is itself failed.

A way to represent this situation is the use the BROADCAST pattern, as illustrated in Figures 8.8 and 8.9.

```
domain SourceState {REGULAR, SURGE}

class Source
    SourceState _state(init=REGULAR);
    event surge(hidden = true);
    transition
        surge: _state==REGULAR -> _state:=SURGE;
end

domain VaristorState {WORKING, FAILED}

class Varistor
    VaristorState _state(init=WORKING);
    event dormantFailure(delay = exponential(1.0e-3));
    event isWorking(hidden = true);
    event isFailed(hidden = true);
    transition
        dormantFailure: _state==WORKING -> _state:=FAILED;
        isWorking: _state==WORKING -> skip;
        isFailed: _state==FAILED -> skip;
end

domain DeviceState {WORKING, FAILED}

class Device
    DeviceState _state(init=WORKING);
    event failure(delay = exponential(1.0e-3), hidden = false);
    transition
        failure: _state==WORKING -> _state:=FAILED;
end
```

Figure 8.8: AltaRica 3.0 code to represent component of the electric circuit

Figure 8.8 gives the model of each component of the circuit. The attribute hidden of the event surge of the source is set to true as this event is always synchronized. The model of the varistor

involves two fake events `isWorking` and `isFailed` that are just modeling tricks to test, within a synchronization, the state of the component. The attributes `hidden` of both events are set to true for the same reason as previously. On the contrary, the attributes `hidden` of events `failure` of devices are set to false, as the devices may fail independently of the power surge.

```
block System
    Source S;
    Varistor V;
    Device D1, D2, D3;
    event protectedSurge(delay = exponential(1.0e-3));
    event unprotectedSurge(delay = exponential(1.0e-3));
    transition
        protectedSurge: !S.surge & !V.isWorking;
        unprotectedSurge: !S.surge & !V.isFailed
            & ?D1.failure & ?D2.failure & ?D3.failure;
end
```

Figure 8.9: AltaRica 3.0 code for the *Broacast* pattern

Figure 8.9 gives the model of the system as a whole. This model involves two events: `protectedSurge` to represent the occurrence of a power surge when the varistor is functioning, and `unprotectedSurge` to represent the occurrence of a power surge when it is failed. The transition `protectedSurge` is similar to the ones used in the SHARED RESOURCE pattern, except that it involves the fake transition `V.isWorking`. The transition `unprotectedSurge` involves both modalities "`!`" and "`?`": for this transition to be enabled, both transitions `S.surge` and `V.isFailed` must be enabled. Transitions `D1.failure`, `D2.failure` and `D3.failure` are optional. However, if they are enabled, they are fired.

8.4 Behavioral Patterns to Represent Dependencies among Failures

8.4.1 Dependent Failures

Modeling faithfully dependent and cascading failures is one of the difficult problem in probabilistic safety assessment. We have seen in Section 7.4.2 how to represent common cause failures using synchronizations. The model given in Figure 7.11 assumes that if the component subject to a common cause failure can fail, it will fail. There are cases however where the failure of component `A` just increases the probability of failure of the component `B` (and possibly vice-versa). When failures are exponentially distributed, it is possible to represent this situation by making `B` changing of state right after the failure of `A`.

As an illustration, consider a pumping system consisting on three pumps `P1`, `P2` and `P3` operating in parallel. When all three pumps are working, they are operated at a certain rotational speed `RS1`, e.g. `RS1` $= 1500$ *rpm*, so that each pump ensures a third of the required production. At this rotational speed, the failure of the pump is exponentially distributed with a failure rate $\lambda_{RS1} = 1.25 \times 10^{-4}\ h^{-1}$. If only two pumps are working, then they must be operated at the higher rotational speed `RS2`, e.g. `RS1` $= 2000$ *rpm*, so that each of the remaining pumps ensures half of the production. Their failure rate increases to λ_{RS2}, e.g. $\lambda_{RS2} = 2.50 \times 10^{-4}\ h^{-1}$. Finally, if only one pumps remains, it should be run at a third rotational speed `RS3`, e.g. `RS3` $= 3000$ *rpm* to ensure 80% of the production. The failure rate increases to λ_{RS3}, e.g. $\lambda_{RS3} = 5.00 \times 10^{-4}\ h^{-1}$. Assume finally that pumps can be repaired and that their repairs take a constant time, say 24 hours.

Figures 8.10 and 8.11 gives an AltaRica model for this pumping system.

This model implements the DEPENDENT FAILURES pattern.

```
domain PumpState {WORKING, FAILED}
domain RotationalSpeed {NULL, RS1, RS2, RS3}

class Pump
    PumpState _state(init = WORKING);
    RotationalSpeed _rotationalSpeed(init = NULL);
    RotationalSpeed demand(reset = NULL);
    event failureRS1(delay = exponential(1.25e-4));
    event failureRS2(delay = exponential(2.50e-4));
    event failureRS3(delay = exponential(5.00e-4));
    event changeRotationalSpeed(delay = Dirac(0));
    event repair(delay = Dirac(24));
    transition
        failureRS1: _state==WORKING and _rotationalSpeed==RS1 -> {
            _state := FAILED; _rotationalSpeed := NULL; }
        failureRS2: _state==WORKING and _rotationalSpeed==RS2 -> {
            _state := FAILED; _rotationalSpeed := NULL; }
        failureRS3: _state==WORKING and _rotationalSpeed==RS3 -> {
            _state := FAILED; _rotationalSpeed := NULL; }
        changeRotationalSpeed: _state==WORKING and
            _rotationalSpeed!=demand -> _rotationalSpeed := demand;
        repair: _state==FAILED -> _state := WORKING;
end
```

Figure 8.10: The AltaRica code for the pumping system (DEPENDENT FAILURES pattern), part 1

The key idea here is that when the demand changes, due to a failure or a repair, the working pumps change of state. This is only possible, at least without making approximations, because we considered exponential distributions: the probability that a pump fails at time $d+t$, given that it is working at time d (the date where its state changes) is the same as the probability that its fails a t, given that it is working a 0. For non exponentially distributed failures, it would be necessary to take into account the "aging" of the pump at the date it changes of state. The current version of AltaRica does not provide mechanisms (at least easy ones) to do so.

8.4.2 Cascading Failures

Cascading failures are another type of dependent failures. Cascading failures occur typically in systems consisting of interconnected components, in which the failure of one the component can trigger the failure of the components in its neighborhood. There are many such (technical) systems, including power transmission systems, computer networks, transportation systems, chemical plants, oil and gas facilities...

As an illustration consider a chemical plant consisting of 4 units `U1-4`. Each unit may fail for some internal reason. We shall call these failures, intrinsic failures. When a unit fails, this failure may be propagated to units nearby:

- The intrinsic failure of unit `U1` may propagate to units `U2` and `U3`;
- The intrinsic failure of unit `U2` may propagate to units `U1` and `U4`;
- The intrinsic failure of unit `U3` may propagate to units `U1` and `U4`;
- The intrinsic failure of unit `U4` may propagate to units `U2` and `U3`;

We shall moreover assume that:

- Intrinsic failures of units are exponentially distributed with a failure rate $2.34 \times 10^{-5}\ h^{-1}$.
- The failure of the `Ui` propagates to the unit `Uj` with a probability 0.15.
- It takes 3 hours for the failure of the unit `Ui` to propagate to the unit `Uj`.

```
block System
    Pump P1, P2, P3;
    Integer workingPumps(reset = 0);
    assertion
        workingPumps := #(P1._state==WORKING, P2._state==WORKING,
            P3._state==WORKING);
        P1.demand := switch {
            case P1._state==FAILED: NULL
            case workingPumps==3: RS1
            case workingPumps==2: RS2
            default: RS3
            };
        P2.demand := switch {
            case P2._state==FAILED: NULL
            case workingPumps==3: RS1
            case workingPumps==2: RS2
            default: RS3
            };
        P3.demand := switch {
            case P3._state==FAILED: NULL
            case workingPumps==3: RS1
            case workingPumps==2: RS2
            default: RS3
            };
    observer Real production = switch {
            case workingPumps==3: 100.0
            case workingPumps==2: 100.0
            case workingPumps==1: 80.0
            default: 0.0
            };
end
```

Figure 8.11: The AltaRica code for the pumping system (DEPENDENT FAILURES pattern), part 2

We took the same distribution for intrinsic failures of the different units, for the sake of simplicity. Of course, different distributions could be taken as well. Similarly, probabilities and delays of propagating failures could also be different from unit to unit.

The AltaRica model for this system relies onto two classes. A class `Unit` to represent units and a class `Propagation` to represent failure propagation from unit `Ui` to unit `Uj`. It is given in Figures 8.12 and 8.13

This model illustrates the use of the CASCADING FAILURES pattern.

8.5 Discrete-Time Markov Chains

8.5.1 Rational

It is sometimes useful to represent discrete-time Markov chains in AltaRica, either because one wants to study the Markov chain *per se*, or because it is embedded into a larger stochastic model.

As an illustration, assume we want to study a system involving the transfer of oil from a floating production storage and offloading unit located on the oil extraction site into a tanker in charge of bringing back the oil to the shore. Ship to ship transfers can only be performed in some specific weather conditions. Figure 8.14 shows a discrete-time Markov chain representing the evolution by periods of 4 hours of the sea state on a simplified Douglas scale.

```
domain UnitState {WORKING, FAILED}

class Unit
    UnitState _state(init = WORKING);
    Boolean propagating, impacted(reset = false);
    event intrinsicFailure(delay = exponential(failureRate));
    event propagatedFailure(delay = Dirac(0));
    parameter Real failureRate = 1.23e-4;
    transition
        intrinsicFailure: _state==WORKING -> _state := FAILED;
        propagatedFailure: _state==WORKING and impacted ->
            _state := FAILED;
    assertion
        propagating := _state==FAILED;
end

domain PropagatorState {READY, PROPAGATE, OBSTRUCT}

class Propagator
    PropagatorState _state(init = READY);
    Boolean impacted, propagating(reset = false);
    event propagate(delay = Dirac(propagationDelay),
        expectation = propagationProbability);
    event obstruct(delay = Dirac(propagationDelay),
        expectation = 1 - propagationProbability);
    parameter Real propagationDelay = 3.0;
    parameter Real propagationProbability = 0.5;
    transition
        propagate: _state==READY and impacted -> _state := PROPAGATE;
        obstruct: _state==READY and impacted -> _state := OBSTRUCT;
    assertion
        propagating := _state==PROPAGATE;
end
```

Figure 8.12: The AltaRica code for the chemical plant (CASCADING FAILURES pattern), part 1

The question is how to represent such model into AltaRica? *A priori*, the solution looks simple: it suffices to represent the transitions and to associate them with 4 hours Dirac delay, e.g.

```
block Sea
    SeaState _state(init = CALM);
    event calmToCalm(delay = Dirac(4), expectation = 0.5);
    event calmToModerate(delay = Dirac(4), expectation = 0.5);
    ...
    transition
        calmToCalm: _state==CALM -> _state := CALM;
        calmToModerate: _state==CALM -> _state := MODERATE;
        ...
end
```

Initially the state of the sea is `CALM`. The two transitions `calmToCalm` and `calmToModerate` are enabled and scheduled at $t = 4$.

If the transition `calmToModerate` is fired, everything goes fine: the state of the sea becomes `MODERATE`, the transition `calmToCalm` is not enabled anymore, the three transitions

```
block System
    Unit U1, U2, U3, U4;
    Propagator P12, P13, P21, P24, P31, P34, P42, P43;
    assertion
        U1.impacted := P21.propagating or P31.propagating;
        U2.impacted := P12.propagating or P42.propagating;
        U3.impacted := P13.propagating or P43.propagating;
        U4.impacted := P24.propagating or P34.propagating;
        P12.impacted:= U1.propagating;
        P13.impacted:= U1.propagating;
        P21.impacted:= U2.propagating;
        P24.impacted:= U2.propagating;
        P31.impacted:= U3.propagating;
        P34.impacted:= U3.propagating;
        P42.impacted:= U4.propagating;
        P43.impacted:= U4.propagating;
end
```

Figure 8.13: The AltaRica code for the chemical plant (CASCADING FAILURES pattern), part 2

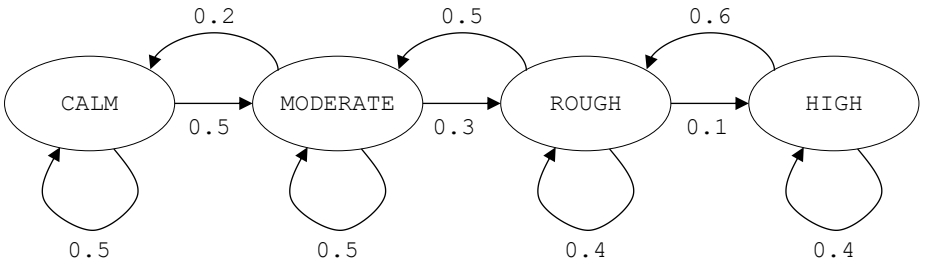


Figure 8.14: A discrete-time Markov chain representing the evolution of the sea state

`moderateToCalm`, `moderateToModerate` and `moderateToRough` become enabled and are schedule at $t = 4+4 = 8$.

However, if the transition `calmToCalm` is fired, the state of the sea stays `CALM`. Consequently, the transition `calmToModerate` stays enabled and scheduled at time $t = 4$. The transition `calmToCalm` is also enabled, but scheduled at time $t = 4+4$. It follows that the only enabled transition is `calmToModerate`, that is fired at $t = 4$.

8.5.2 Pattern

To solve the problem described in the previous section, the idea is to consider two types of transitions, timed deterministic transitions and immediate stochastic ones, and to alternate them. This approach is in some similar to the one of *hybrid automata* (Henzinger 1996). It gives raise to the DISCRETE-TIME MARKOV CHAIN pattern.

To encode the discrete-time Markov chain pictured in Figure 8.14, we must transform it into a hybrid automata. The idea is to introduce timed transitions to make the time elapsed in between two immediate transitions. Here the time elapses by one unit.

Figure 8.15 shows an AltaRica code for the discrete-time Markov chain pictured in Figure 8.14. This code follows the DISCRETE-TIME MARKOV CHAIN pattern.

The timed transition `timeStep` alternates with the other transitions which are immediate. This avoid the problem discussed in the previous section.

Note that there may be several timed transitions (one per state) and that the delays associated

```
domain SeaState {CALM, MODERATE, ROUGH, HIGH}

block Sea
  SeaState _state(init = CALM);
  event calmToCalm(delay = Dirac(0), expectation = 0.5);
  event calmToModerate(delay = Dirac(0), expectation = 0.5);
  event moderateToCalm(delay = Dirac(0), expectation = 0.2);
  event moderateToModerate(delay = Dirac(0), expectation = 0.5);
  event moderateToRough(delay = Dirac(0), expectation = 0.3);
  event roughToModerate(delay = Dirac(0), expectation = 0.5);
  event roughToRough(delay = Dirac(0), expectation = 0.4);
  event roughToHigh(delay = Dirac(0), expectation = 0.1);
  event highToRough(delay = Dirac(0), expectation = 0.6);
  event highToHigh(delay = Dirac(0), expectation = 0.4);
  Boolean _timeStep(init = true);
  event timeStep(delay = Dirac(4));
  transition
    calmToCalm: not _timeStep and _state==CALM -> {
           _state := CALM; _timeStep := true; }
    calmToModerate: not _timeStep and _state==CALM -> {
           _state := MODERATE; _timeStep := true; }
    moderateToCalm: not _timeStep and _state==MODERATE -> {
           _state := CALM; _timeStep := true; }
    moderateToModerate: not _timeStep and _state==MODERATE -> {
           _state := MODERATE; _timeStep := true; }
    moderateToRough: not _timeStep and _state==MODERATE -> {
           _state := ROUGH; _timeStep := true; }
    roughToModerate: not _timeStep and _state==ROUGH -> {
           _state := MODERATE; _timeStep := true; }
    roughToRough: not _timeStep and _state==ROUGH -> {
           _state := ROUGH; _timeStep := true; }
    roughToHigh: not _timeStep and _state==ROUGH -> {
           _state := HIGH; _timeStep := true; }
    highToRough: not _timeStep and _state==HIGH -> {
           _state := ROUGH; _timeStep := true; }
    highToHigh: not _timeStep and _state==HIGH -> {
           _state := HIGH; _timeStep := true; }
    timeStep: _timeStep -> _timeStep := false;
end
```

Figure 8.15: The AltaRica code for the discrete-time Markov chain pictured in Figure 8.14

with these transitions may be stochastic.

Note also that models such as the one given in 8.15 can be assessed by Monte-Carlo simulation. It may look awkward to use Monte-Carlo simulation as very efficient numerical algorithms exist for both discrete-time and continuous-time Markov chains (Stewart 1994). These algorithms require however to build explicitly the vector of all of the states of the chain. For very large models, this is simply impossible. Monte-Carlo simulation is thus a last resort resource to assess these models.

8.6 Structural Patterns

So far, we discussed behavioral patterns, i.e. small models the analyst can take example from to design modeling components suiting to his own needs. We shall discuss now architectural patterns,

i.e. ways to architect models. To start with we shall consider again two structural patterns we encountered many times in this book: tree-like breakdown and block diagrams.

8.6.1 Case Study: a High Integrity Pressure Protection System

To illustrate the presentation, we shall consider the High Integrity Pressure Protection System pictured in Figure 8.16.

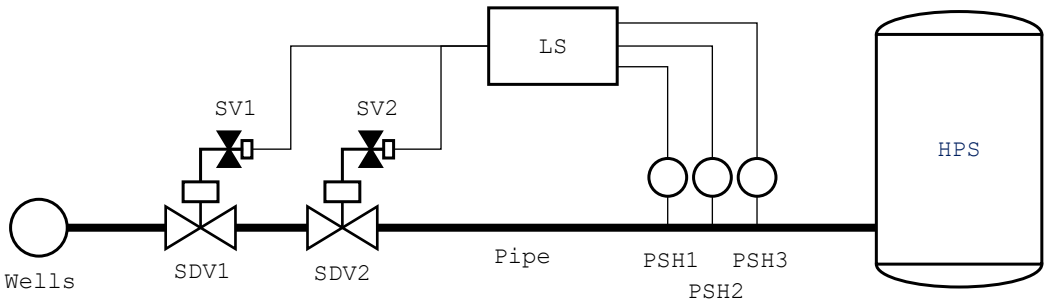


Figure 8.16: A High Integrity Pressure Protection System

The mix of oil, gas and water coming from the wells nearby is sent to the high pressure separator `HPS`. To prevent damages to the separator in case of an overpressure, a high integrity pressure protection system (HIPPS) is installed upstream.

This system consists of three parts:

– A sensor line, made of three pressure sensors `PSH1`, `PSH2` and `PSH3`;
– A 2-out-of-3 logic solver;
– Two actuators, made of a solenoid valve `SVi` and shutdown valve `SDVi`, $\texttt{i} = 1,2$.

As pointed out already several time, a safety study for the system as a whole consists mostly in assessing the reliability of the safety system, here the high integrity pressure protection system, protecting the equipment under control, here the high pressure separator.

8.6.2 Block Diagrams

A first way to architect a model of the system is to start from its physical architecture. This usually leads to a block diagram representation, i.e. to the BLOCK DIAGRAM structural pattern.

Figure 8.17 shows a typical structure of the model organized according to this pattern.

This structure translates easily in a model, as shown in Figure 8.18.

Note that this model involves only Boolean flows. Flows could however belong to any domain, the structure would not change.

8.6.3 Tree-Like Breakdown

An alternative way of structuring the model consists in using a tree-like breakdown, i.e. in using the TREE-LIKE BREAKDOWN structural pattern. In general, this structure is inspired from the functional architecture of the system under study,

The tree-like breakdown for our case study is pictured in Figure 8.19.

As previously, this structure translates easily in a model, as shown in Figure 8.20.

Here again flows could belong to any domain, the structure would not change.

8.6.4 Discussion

A question arises naturally: of the BLOCK DIAGRAM and TREE-LIKE BREAKDOWN patterns, which one is best? As expected, there is no general answer to this question. It depends on the model

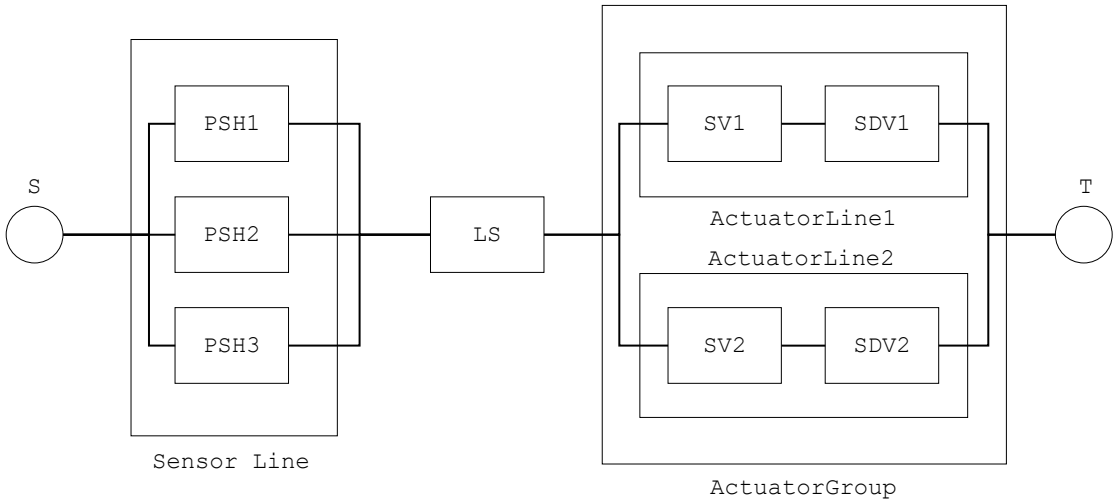


Figure 8.17: A block diagram structure for the model of the HIPPS pictured in Figure 8.16

```
block HIPPS
    Boolean S(reset = false);
    block SensorLine
        Sensor PSH1, PSH2, PSH3;
        Boolean input(reset = false);
        assertion
            PSH1.input := input;
            PSH2.input := input;
            PSH3.input := input;
    end
    LogicSolver LS;
    block ActuatorGroup
        block ActuatorLine1
            SolenoidValve SV;
            ShutdownValve SDV;
            assertion
                SDV.command := SV.output;
        end
        clones ActuatorLine1 as ActuatorLine2;
    end
    Boolean T(reset = false);
    assertion
        S := true;
        SensorLine.input := W.output;
        LS.input1 := SensorLine.PSH1.output;
        LS.input2 := SensorLine.PSH2.output;
        LS.input3 := SensorLine.PSH3.output;
        ActuatorGroup.ActuatorLine1.SV.command := LS.command;
        ActuatorGroup.ActuatorLine2.SV.command := LS.command;
        T := ActuatorGroup.ActuatorLine1.SVD.output
            or ActuatorGroup.ActuatorLine2.SVD.output;
    observer Boolean failed = not T;
end
```

Figure 8.18: AltaRica code that reflects the block diagram structure given in Figure 8.17

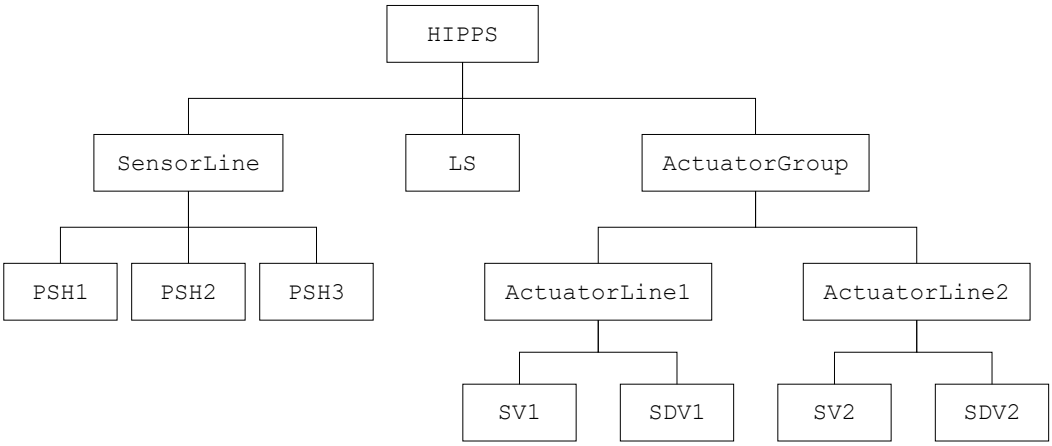


Figure 8.19: Tree-like breakdown structure for the model of the HIPPS pictured in Figure 8.19

```
block HIPPS
    block SensorLine
        Sensor PSH1, PSH2, PSH3;
        Boolean output(reset = false);
        assertion
            output := #(PSH1.output, PSH2.output, PSH3.output) >= 2;
    end
    LogicSolver LS;
    block ActuatorGroup
        Boolean output(reset = false);
        block ActuatorLine1
            SolenoidValve SV;
            ShutdownValve SDV;
            Boolean output(reset = false);
            assertion
                output := SV.command and SDV.output;
        end
        clones ActuatorLine1 as ActuatorLine2;
        assertion
            output := ActuatorLine1.output or ActuatorLine2.output;
    end
    Boolean Top(reset = false);
    assertion
        Top := SensorLine.output and LS.output and ActuatorGroup.output;
    observer Boolean failed = not Top;
end
```

Figure 8.20: AltaRica code that reflects the tree-like breakdown structure given in Figure 8.19

and is to some extent a matter of taste.

It can be argued that the block diagram architecture is closer to the view of engineers as it shows the components and the flows of matters, energy and information circulating among the latter. This argument should however be taken with care: the flow in a block diagram architecture do not reflect necessarily the actual flows in the system. Always remember that a model is an abstract, like a map abstracts the territory. The question is not whether a model is faithful, but whether it is useful.

With that respect, a structure that reflects the functional architecture of the system, although *a priori* more abstract, may be more suitable for the problem at stake.

At the end of the day, always remember the KISS principle: keep it simple, stupid.

8.7 Monitored Systems

8.7.1 Preliminary Remarks

This section is the last of this chapter but without any doubt one of the most important, if not the most important.

It is tempting to see the design of an AltaRica model as a Lego® game: one starts with basic bricks then build up the model by assembling these basic bricks. The author must admit that it took him a long time to realize that this metaphor is not only wrong, but actually counterproductive.

The fact of the matter is that systems, sub-systems and components of a complex technical system are all *monitored*, i.e. made essentially of two parts: an equipment under control and a controller. The only question is therefore: where is the controller and what does it do? With that respect, Leveson is right with her STAMP method (Leveson 2012)[1]: designing a model is eventually looking at a series of control problems.

To make thing concrete, consider for instance the WARM REDUNDANCY pattern. In this pattern, some transitions are those of the equipment under control: `dormantFailure`, `failure`, and `repair`. The others, i.e. `turnOn`, `failureOnDemand` and `turnOff` results from an external action, i.e. a control.

Note that the transition `repair`, although involving in reality the action of an external system, the repair crew, is here represented as internal to the component. In other patterns, like the SHARED RESOURCE pattern (used in case the number of repair crews is limited), transitions `startRepair` and `completeRepair` are controlled from outside of the component.

In a system, there may be several controllers. Each of them acts locally on one or more components, one or more subsystems. It is often the case that these controllers are implicitly represented in model as it would overload the model to represent explicitly. Nevertheless, when designing the model, the analyst must bear in mind the fundamental difference between internal actions and controlled ones.

8.7.2 Case Study: a Gas Production Facility

To illustrate the presentation, consider the gas production facility pictured in Figure 8.21.

The facility compresses gas coming from the producer `P` and sends it through the distribution network `D`. It consists of three identical trains `T1`, `T2` and `T3`. Each train consists itself of a motor `M` and a compressor `C`.

The available gas production coming from `P` varies. Figure 8.22 shows a discrete-time Markov chain describing the variations of the available monthly gas production. The production is given in an abstract gas unit. In practice, it is in general given either in multiple of kilowatts per hour or of cubic meters.

The demand of `D` may also vary. Figure 8.23 shows able discrete-time Markov chain describing the variations of the monthly gas demand.

It may be the case that the demand exceeds the available production and vice-versa. The gas production facility must thus adjust accordingly.

Each train can compress up to 40 gas units per month. The operation policy is to operate as few trains as possible to deliver the demand. Production is allocated in priority to train `T1`, then to train `t2` and eventually to train `T3`.

Trains may fail due a either a failure of their motor or their compressor. Their failures are both exponentially distributed with respective failure rates $1.23 \times 10^{-6}\ h^{-1}$ and $3.45 \times 10^{-7}\ h^{-1}$. Once

[1] Although she is wrong about safety analyses in general, as she neglects the technical part of these systems, putting much too much weight on organizational aspects.

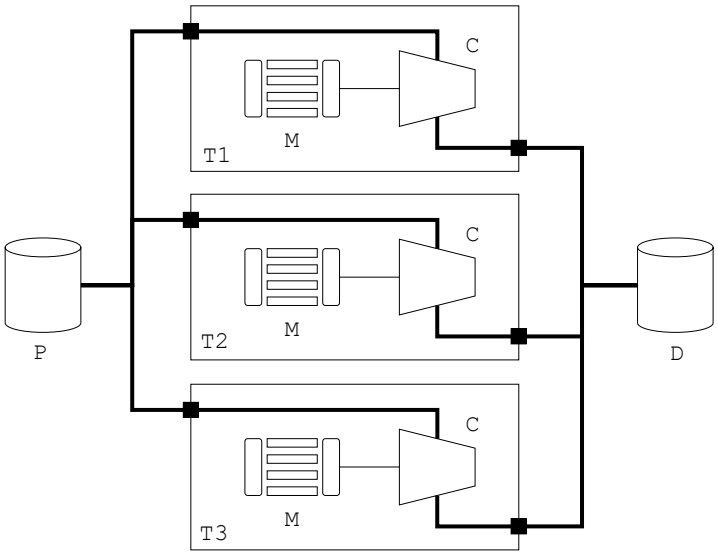


Figure 8.21: A gas production facility

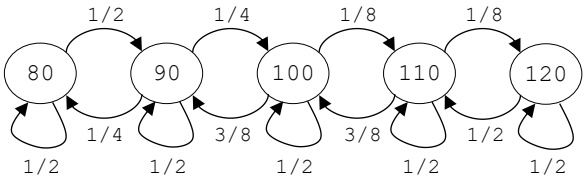


Figure 8.22: Discrete-time Markov chain describing the variations of the available monthly gas production

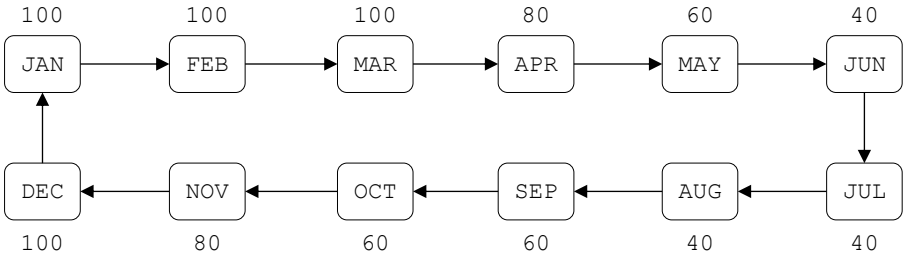


Figure 8.23: Discrete-time Markov chain describing the variations of the monthly gas demand

failed, they can be repaired. Their repair times are also exponentially distributed, with respective mean time to repair 1.20×10^2 *h* and 2.40×10^2 *h*.

The first objective of the study is to estimate the average losses over a 5 years period, i.e. the expected number of gas units that are demanded but not produced.

8.7.3 Model

The model of our gas production facility cannot be obtain by assembling models of components. The production of each train depends on the production capacity received in input, the demand in output and the state of the other trains. In other words, no decision can be made locally. One needs a global controller.

The MONITORED SYSTEM pattern provides a generic architecture for this type of systems. Figure 8.24 illustrates this architecture.

It works as follows.

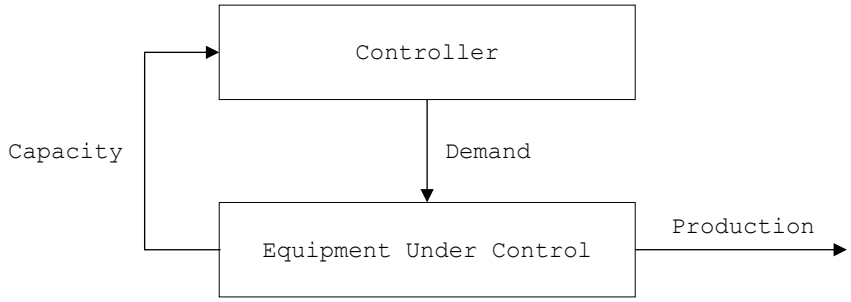


Figure 8.24: Generic architecture of the MONITORED SYSTEM pattern

– Each component of the equipment under control exports its capacity to the controller.
– Based on this diagnostic, the controller sends demands of production to components.
– Eventually, components produce according to these demands.

Technically, exports of capacities are usually implemented by means of flow variables. Demands can be implemented using either flow variables or synchronizations.

In our example, the model consists of the following elements.

– A block `P` to represent the producer.
– Three blocks `T1`, `T2`, and `T3` to represent the trains. Each of these blocks composes two sub-blocks, a sub-block `M` to represent the motor, and a sub-block `C` to represent the controller.
– A block `P` to represent the distribution network.
– Finally, a block `CTRL` to implement the controller.

The producer `P` and the trains `T1`, `T2`, and `T3` export their capacities to the controller `CTRL`. Meanwhile, the distribution network `D` exports the demand to `CTRL`. Based on these capacities, the controller allocates the demand to the trains. Eventually, the trains produce according to the demand.

The block `P` and `P` encode discrete-time Markov chains according to the pattern we have seen Section 8.5. Motors and compressors are designed according to the COLD REDUNDANCY pattern. The block representing trains are just assembling the motors and the compressors as illustrated in Figure 8.25.

```
class Train
    Motor M;
    Compressor C;
    Real capacity, demand, production(reset = 0.0);
    parameter Real intrinsicCapacity = 40;
    assertion
        capacity :=
            if M._state!=FAILED and C._state!=FAILED
            then intrinsicCapacity
            else 0;
        M.demand := demand>0;
        C.demand := demand>0;
        production :=
            if M._state==WORKING and C._state==WORKING
            then demand
            else 0;
end
```

Figure 8.25: AltaRica code for the trains of the gas production facility

The code for the controller is given in Figure 8.26.

```
class Controller
  Real P_production(reset = 0.0);
  Real D_demand(reset = 0.0);
  Real T1_capacity, T2_capacity, T3_capacity(reset = 0.0);
  Real Ts_capacity(reset = 0.0);
  Real demand1, demand2, demand3(reset = 0.0);
  Real T1_demand, T2_demand, T3_demand(reset = 0.0);
  assertion
    Ts_capacity := T1_capacity + T2_capacity + T3_capacity;
    demand1 := min(P_production, D_demand, Ts_capacity);
    T1_demand := min(demand1, T1_capacity);
    demand2 := demand1 - T1_demand;
    T2_demand := min(demand2, T2_capacity);
    demand3 := demand2 - T2_demand;
    T3_demand := min(demand3, T3_capacity);
end
```

Figure 8.26: AltaRica code for the controller of the gas production facility

Finally, the AltaRica code for the gas production facility itself is given in Figure 8.27. This code simply declares the components and connects them.

```
block GasProductionFacility
  Producer P;
  DistributionNetwork D;
  Train T1, T2, T3;
  Controller CTRL;
  Real production(reset = 0.0);
  assertion
    CTRL.P_production := P.production;
    CTRL.D_demand := D.demand;
    CTRL.T1_capacity := T1.capacity;
    CTRL.T2_capacity := T2.capacity;
    CTRL.T3_capacity := T3.capacity;
    T1.demand := CTRL.T1_demand;
    T2.demand := CTRL.T2_demand;
    T3.demand := CTRL.T3_demand;
    production := T1.production + T2.production + T3.production;
  observer Real Production = production;
  observer Real ProductionLoss = D.demand - production;
end
```

Figure 8.27: AltaRica code for the gas production facility

This example concludes this chapter.

8.8 Exercises and Problems

Exercise 8.1 – Truck Braking System. In engineering, a *fail-safe design* for a component is a design so that in case of a specific type of failure, the component responds in a way that will cause minimal or no harm to other equipment, to the environment or to people. Examples of fail-safe design are numerous. For instance, elevators have brakes that are held off brake pads by the tension

of the elevator cable. If the cable breaks, tension is lost and the brakes latch on the rails in the shaft, so that the elevator cabin does not fall.

As an another illustration consider the braking system of a six-wheels truck, three on each side of the truck. There is a brake on each wheel. The brakes are held in the "off" position by air pressure created in the braking system. If the brake line split, the air pressure is lost and the brakes applied, by a spring. A typical fail-safe design. Each brake has thus two failure modes: a fail-safe mode, in which the truck driver is informed that something goes wrong and consequently stops the truck (or is not able to start it), and a dangerous failure mode, e.g. due to a damage to the spring, in which it is not able to provide a braking capacity and the truck driver is not informed of the problem. A catastrophic event would be that the three brakes on one side of the truck are lost, in case of an emergency braking. Note that the hazard "emergency braking" is almost certain to occur in a truck journey.

Assume that both safe- and dangerous failures of brakes are exponentially distributed, with respective failure rates $5.45 \times 10^{-4}\ h^{-1}$ and $1.22 \times 10^{-6}\ h^{-1}$. Assume moreover that after a fail-safe alert, a 2 minutes inspection is sufficient to put things back in place.

Question 1. Modify the repairable component pattern to represent safe and dangerous failures. We can call this new modeling the EXCLUSIVE FAILURE pattern. Use this pattern to design an AltaRica model of the braking system of the truck. Check your model using the interactive simulator.

Question 2. Use the stochastic simulator to assess the following key performance indicators:
- The probability of an accident due to the loss of the braking system in case of an emergency braking, over a 5 years period. Calculate this probability at sufficiently intermediate dates to get a good idea of its evolution through the time.
- The expected number of fail-safe alerts through the same period.
- The expected time lost in inspections after fail-safe events.

Question 3. Based on the results of the above experiment, propose a maintenance/inspection policy for the braking system.

■

Problem 8.2 – Staggered Maintenance Interventions. Consider a system made of three similar actuators `A1`, `A2` and `A3`, two of which are required for the system to work. These actuators are periodically inspected and maintained. Their failures are revealed by these inspections. Assume that these failures are exponentially distributed with a failure rate $3.75 \times 10^{-5}\ h^{-1}$. Assume moreover that an inspection takes normally 2 hours but that, in case the actuator is failed, its repair time is exponentially distributed with a mean time to repair of 8 hours. The actuator are turned off during inspection (and of course repairs).

Question 1. Design a class `RepairCrew` to represent a repair crew that starts the maintenance of the actuators every 9 months.

Question 2. Modify the "periodically maintained component" pattern, described in Figure **??**, to get a class `Actuator` that represents the actuators.

Question 3. Design a model of the system by composing an instance of the class `RepairCrew` with three instances of the class `Actuator`. Synchronize these instances. Validate the model with the interactive simulator.

Question 4. Use the stochastic simulator to calculate:

- The unreliability of the system, over a 5 year period.
- The expected sojourn-time in a state where the system is not working, over the same time period.

One of the problem with the above maintenance policy is that the system is stopped during the maintenance of the components, even if it would have been possible to continue the operations. Hence the idea to stagger maintenance operations, e.g. by maintaining the actuator `A1` at $t = 3$ months and then every 9 months, the actuator `A2` at $t = 6$ months and then every 9 months and finally the actuator `A3` at $t = 9$ months and then every 9 months.

Question 9. Modify the class `RepairCrew` designed in question Question 1. so to represent staggered maintenance interventions. Modify the model of the system designed at question Question 3. to synchronize actions `startMaintenance` of the repair crew with those of the actuators. Validate the model using the interactive simulator.

Question 10. Use the stochastic simulator to re-assess the key performance indicators calculated at question Question 4.. Compare the two series of results.

■

Problem 8.3 – Spare AC/DC Converter Line. Consider the electric system pictured in Figure 8.29. A source of alternative current `PS` is powering a busbar `BB`. The alternative current must however be converted into a continuous current. Two identical converter lines `Line1` and `Line2` are used for this purpose. The line `Line2` is in cold redundancy for the line `Line1`. Each line consists of three units: a converter `CVT` and two circuit breakers, one located upstream (`CBU`) and one located downstream (`CBD`) the converter.

The converter may fail. Its failure is exponentially distributed with a failure rate $4.43 \times 10^{-5}\ h^{-1}$. Converters are assumed not to fail when in standby.

The circuit breakers have two failure modes: "spurious opening", when they are closed, and "stuck closed" when they are open and attempted to close. The spurious opening is exponentially distributed, with a failure rate $7.21 \times 10^{-7}\ h^{-1}$. The failure on demand "stuck closed" has a probability 1.67×10^{-3}. Circuit breakers are assumed not to have dormant failures.

For the purpose of this problem, we shall assume that the power source `PS` and the busbar `BB` are perfectly reliable.

Question 1. Using the WARM REDUNDANCY pattern, design a class `Converter` to represent converters.

Question 2. Using the WARM REDUNDANCY pattern, design a class `CircuitBreaker` to represent circuit breakers.

Question 3. Using classes `Converter` and `CircuitBreaker`, design a model of the system. Validate this model using the interactive simulator.

Question 4. Using the critical sequence generators, extract the sequences of events leading to a loss of power of the busbar.

Question 5. Compile the AltaRica model into a system of Boolean equations. Extract minimal cutsets for the top event representing the loss of power of the busbar. Compare these minimal cutsets with the sequence obtained in the previous question. Using XFTA, calculate the probability that the busbar `BB` is not powered over a 1 year period.

Question 6. Using the stochastic simulator, assess the probability that the busbar BB is not powered over a 1 year period. Compare this probability with the probability obtained at the previous question. What can you conclude?

■

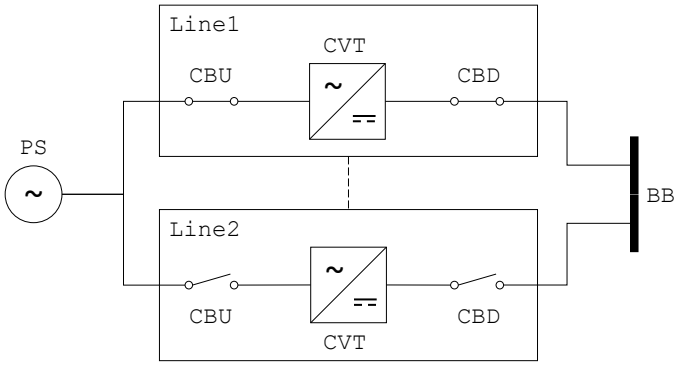


Figure 8.28: A AC/DC converter system

Exercise 8.4 – Backup Batteries. Consider the system pictured in Figure 8.29. The server (or group of servers) S is normally powered by the grid G. The circuit breaker CBG is thus normally closed.

In case the grid is lost, two successive lines of backup batteries are used. The line consisting of the battery CB1 and the circuit breaker CB1 is used first. If this line is lost, the second line consisting of the battery CB2 and the circuit breaker CB2 is used.

We make the following assumptions.

- The loss of the grid is exponentially distributed, with an occurrence rate estimated at $1.50 \times 10^{-4}\ h^{-1}$. The time to put it back in service is also exponentially distributed, with a mean time estimated at 4 hours.
- The batteries are assumed to be regularly tested and maintained so that they are always available when demanded. Once in use their discharge time is uniformly distributed between 6 and 8 hours.
- Circuit breakers have two failure modes:
  - Spurious openings, which are exponentially distributed, with an occurrence rate estimated at $2.50 \times 10^{-5}\ h^{-1}$.
  - The impossibility to close them when needed, which occurs with a probability estimated at 0.005.

Question 1. Design classes to represent the grid, the batteries and the circuit breakers.

Question 2. Using these classes design a model of the system. Validate the model with the interactive simulator.

Question 3. Using the stochastic simulator, assess the following key performance indicators.

- The probability that the server is not powered, over a one year period.
- The expected time during which the server is not powered, still over a one year period.

■

Exercise 8.5 – Alarm Management. Consider a chemical plant where two reactors R1 and R2 are

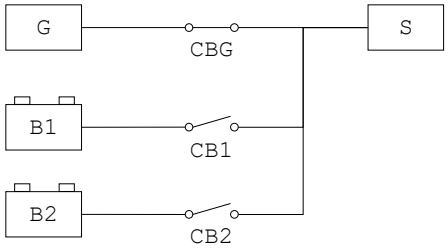


Figure 8.29: Backup batteries

used in turn, every 12 hours. A pump P is in charge of delivering cooling fluid to the reactors. In case the reactor in operation is not delivered cooling fluid, it resists between 1 and 2 hours before exploding, which causes a major accident.

To prevent such an accident to happen, a monitoring system M is installed on the pump P. This monitoring system is able to detect the degradation of the pump. When such a degradation is detected, M raises an alarm that stops in emergency the reactors (the reactor that was not in operation is put in a safe mode as well).

The degradation of the pump P is exponentially distributed, with a degradation rate $1.45 \times 10^{-4}\ h^{-1}$. Once degraded, it takes between 4 and 6 hours for the pump to fail completely.

The monitoring system M can itself fail. Its failure is exponential distributed with a failure rate $6.43 \times 10^{-6}\ h^{-1}$.

Question 1. Design classes to represent the reactors, the pump and the monitoring system.

Question 2. Using these classes design a model of the system. Validate the model with the interactive simulator.

Question 3. Using the stochastic simulator, assess the probability of a major accident over a 10 years period.

■

Exercise 8.6 – Gambler's Ruin. Alice and Bob plays at head and tail. They bet 1 euro on each throw, i.e. a head is tossed, Alice gives 1 euro Bob, while if a tail is tossed, it is Bob who gives 1 euro to Alice. They are using a fair coin, i.e. head and tail have both the probability $1/2$ to be tossed. The game ends when one of the two players is ruined.

Question 1. Propose a discrete time Markov chain to represent the game.

Question 2. Design an AltaRica model for this system.

Question 3. Using the stochastic simulator, assess the probabilities that Alice is ruined and that Bob is ruined. Assess also the mean number of throws before one of the players gets ruined.

■

Exercise 8.7 – Level Evolution. Consider again the discrete-time Markov chain pictured in Figure 5.11, page 116.

Question 1. Design an AltaRica model for this Markov chain.

Question 2. Using the stochastic simulator, assess the steady state probabilities to be at each level.

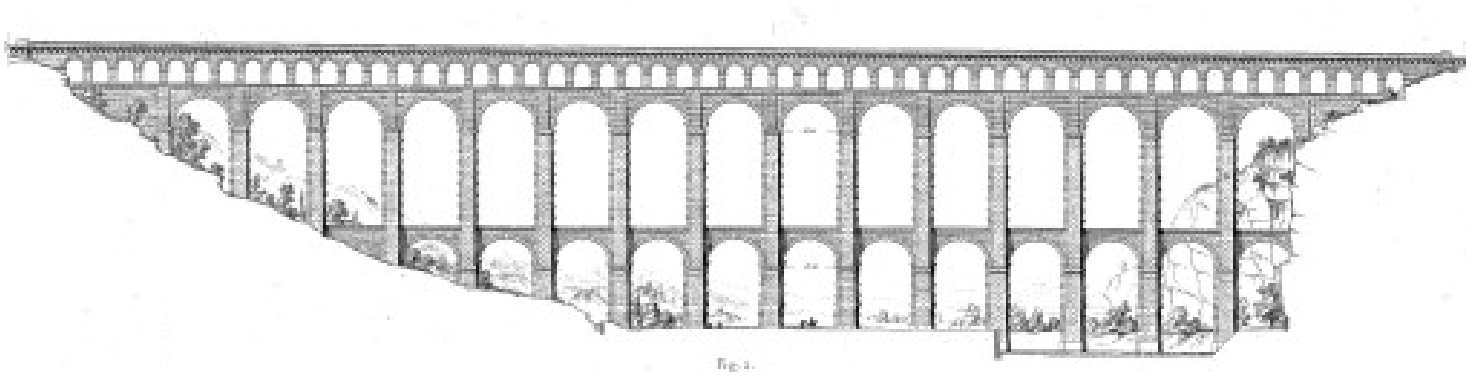
■

Exercise 8.8 – Gas Production Facility (bis). Consider again the gas production facility discussed Section 8.7.

Question 1. Modify the controller so that in case two or more trains are required to satisfy the demand, they all produce the same quantity of gas.

Question 2. Modify the controller so that the priority among the trains changes every three months, e.g. the first three months `T1` is used in priority, then `T2`, then `T3`; the next three months its `T2` which is used in priority, then `T3`, then `T1`; the next three months its `T3` which is used in priority, then `T1`, then `T2`; and so on.

■



9. Computational Complexity Issues

Key Concepts
- Experiments and problems
- Turing machines, universal computers
- Decidability and complexity
- Complexity classes
- Expressive power
- Combinatorial models, state automata and process algebras

Computational complexity theory is a branch of theoretical computer science that aims at classifying problems according to the cost, in terms of computational resources, of solving them. In this chapter, we recall essential notions of this theory and review results that have important consequences for model-based systems and reliability engineering. The reader interested in a broader perspective should look at reference textbooks (Arora and Barak 2009; Garey and Johnson 1979; Papadimitriou 1994).

We propose a taxonomy of modeling languages used in reliability engineering. This taxonomy classifies problems according to their expressive power. It is made of three nested classes: combinatorial models, state automata and process algebras. We provide three mathematical frameworks, one per class, to illustrate our taxonomy. We describe also typical experiments that can be performed on models designed in these frameworks and we study the computational complexity of these experiments.

9.1 Experiments and Problems

Models are of interest thanks to experiments we can perform with them. There are indeed many types of possible experiments, ranging from brainstorming among stakeholders to computationally intensive simulations on supercomputers. We shall focus in this chapter on experiments *in silico*, i.e. experiments that consists in calculating something, using a computer (a machine) and applying an algorithm, i.e. a mechanical, predefined method. These experiments are virtual in this sense that they answer questions about the models and not about the systems. It is then up to the analyst to

interpret their results and to draw conclusions about the system under study. This latter remark has several important theoretical and practical consequences.

First, to study a given feature of the system, one needs to design a model that is rich enough to capture that feature and thus to use a modeling language that is expressive enough to make possible the design of this model. It is therefore of primary importance to characterize the expressive power of modeling languages.

This is however only one face of the medal. The other and equally important face regards the feasibility of experiments. Algorithms have actually a computational complexity, i.e. they require a certain amount of time and computer memory to be executed. As a general rule, the richer the models, the more computationally costly the assessment algorithms.

Too costly experiments would make us jump from the frying pane (of experiments on real systems) into the fire (of virtual experiments on models). Consequently, designing a model results always of a tradeoff: on the one hand, one would like to make the model as precise as possible; on the other hand, the experiments performed on this model must remain tractable, i.e. feasible within a reasonable amount of computation resource. A key issue is therefore to assess the cost of computerized experiments.

This is where computational complexity theory comes into the play. Computational complexity theory considers problems stated in mathematical terms, as the following one.

Definition 9.1.1 – SHORTESTPATH. Consider a network such as the one pictured Figure 9.1. Each edge of the network is labeled with a length (no matter what this length represents, a distance, a travel time, a cost...). Consider moreover two distinguished nodes of this network, say `A` and `I`. What is the shortest path from the first node to the second one?

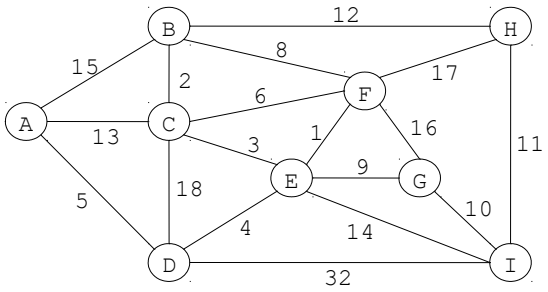


Figure 9.1: A network

Computational complexity theory aims at relating the cost of solving problems to the size of their description. It considers thus families of problems rather than individual problems: we are interested in the computational cost of the best algorithm that given the description of any network such as the one pictured Figure 9.1 and two distinguished nodes s and t of that network, returns the shortest path from s to t.

There are however several major issues here:

– First, before speaking about cost, we need to ensure that there exists an algorithm to solve the problem.
– Second, two problems P and Q of the family of under study may have representations of the same size, but widely different solving costs.
– Third, algorithms are executed by computers, humans or machines. But computers are not all working at the same speed, as we can experience in our daily life. Moreover, there is *a priori* no reason for two different computers to have proportional efficiency on different basic steps: the computer C may execute the instructions (basic steps) I and J in respectively 1 and 1000 time units, while for the computer D the ratio is exactly inverse.

We shall see how computational complexity theory tackles these issues. But for now, let us come back in the framework of model-based systems and reliability engineering.

9.2 Taxonomy of Modeling Languages

9.2.1 Overview

In the context of model-based systems and reliability engineering, a few "natural" problems are associated to each modeling language. These problems correspond to properties one aims to study by designing models with this language. The more expressive the modeling language, the more computationally complex the associated problems.

With that respect, modeling languages involved in reliability engineering can be split into three nested categories, ordered by increasing expressive power: combinatorial models, state automata and process algebras.

Combinatorial models describe essentially the possible states of the system as a combination of the possible states of its components. As one considers a finite and small number of states for each individual component, the number of states of the system as a whole is itself finite, although subject to a combinatorial explosion when the number of components increases. Combinatorial models provide a static view on the system.

State automata describe not only states of components, but also transitions between these states under the occurrence of events. In each state of the system, some transitions are enabled and some are not. Firing a transition changes the state of the system. Transitions are in general described at component level. Starting from an initial state, the system may follow different trajectories, depending on which sequence of transitions is fired. Conversely to combinatorial models, state automata make possible the description of the temporal evolution of the system. Although state automata can potentially have infinite state spaces—e.g. if they involve counters of the number of times the system passes in a given configuration—they are in general restrained to finite ones.

In both combinatorial models and state automata, the number of components and the organization of the components, in a word the architecture of the system, is defined once for all. The study of some systems however require to represent evolving architectures. Process algebras or equivalently agent systems are commonly used to describe these systems. In a process algebra, the system is represented by means of a set of processes evolving in parallel. At each step of the evolution, a process (agent) or a small group of processes (agents) performs an action which modifies both the processes involved in the action and the state of system. New processes can be created and existing ones deleted.

Process algebras are thus strictly more expressive than state automata which are themselves strictly more expressive than combinatorial models. There is indeed a computational price to pay for this additional expressive power: models are harder to design and to validate and problems to be solved are computationally more complex. In each case, restrictions, approximations and various other techniques can be used to lower the computational complexity of the problems at stake. It remains that this complexity frames the whole modeling process: in model-based systems engineering, a model results always of a tradeoff between the accuracy of the description of the system under study and the ability one has to perform experiments *in silico* on this description within a reasonable amount of computational resources.

Table 9.1 summarizes our taxonomy of modeling languages.

In the remainder of this section, we shall discuss further each category of modeling languages, by considering three mathematical frameworks BMF, KMF, and TMF, hence called in honor of respectively Georges Boole, Saul Kripke and Alan Turing (MF stands for mathematical framework). Note that BMF, KMF and TMF can be turned into full fledged modeling languages, thanks to the S2ML+X paradigm.

Table 9.1: Taxonomy of modeling languages used in model-based systems engineering

	Combinatorial models	State automata	Process algebras
Description	States	States and transitions	States and processes
View	Static	Dynamic	Dynamic
Architecture	Fixed	Fixed	Evolving

9.2.2 BMF

In BMF, a system is seen as a finite set of components. Each component can be in a finite number of states. The state of the system as a whole is described by means of a logical formula built over the state of the components with the usual connectives: $\wedge$ (and), $\vee$ (or) and $\neg$ (not). Formally:

Definition 9.2.1 – BMF (Syntax). A BMF model is a pair $\langle V, P \rangle$ where:
- V is a finite set of symbols called *variables*. Each variable v of V takes its value in a finite (and non empty) set of symbolic constants called its domain and denoted $\mathrm{dom}(v)$.
- P is a finite set of *predicates*. Each predicate p of P is a Boolean formula built over the variables of V as explained below.

The set of *Boolean formulas* built over V is the smallest set such that:
- The two Boolean constants 0 (false) and 1 (true) are Boolean formulas.
- If v is a variable of V and c is a constant of $\mathrm{dom}(v)$, then $v = c$ is a Boolean formula.
- f, f_1,... f_n are Boolean formulas, then so are $f_1 \wedge \cdots \wedge f_n$, $f_1 \vee \cdots \vee f_n$, $\neg f$ and (f).

Variables are indeed used to represent states of components and predicates to describe the state or some property of the system as a whole.

Boolean formulas are named after the British mathematician, philosopher, and logician George Boole (1815-1864) who was the first to study the logical reasoning from an algebraic point of view in his famous book "The Laws of Thought" (Boole 1854).

The semantics of BMF models is defined via the notion of variable valuations.

Definition 9.2.2 – BMF (Semantics). Let $\langle V, P \rangle$ be a BMF model. A *variable valuation* of V is a function that associates to each variable v of V a constant of $\mathrm{dom}(v)$.

Variable valuations are lifted-up to predicates of P as follows. Let σ be a variable valuation of V, let v be a variable of V, let c be a constant of $\mathrm{dom}(v)$ and finally let f, f_1,... f_n be Boolean formulas built over V. Then,

$$\begin{aligned}
\sigma(0) &= 0 \\
\sigma(1) &= 1 \\
\sigma(v = c) &= 1 \text{ if } \sigma(v) = c \text{ and } 0 \text{ otherwise} \\
\sigma(f_1 \wedge \cdots \wedge f_n) &= min(\sigma(f_1), \ldots \sigma(f_n)) \\
\sigma(f_1 \vee \cdots \vee f_n) &= max(\sigma(f_1), \ldots \sigma(f_n)) \\
\sigma(\neg f) &= 1 - \sigma(f) \\
\sigma(\,(f)\,) &= \sigma(f)
\end{aligned}$$

A variable valuation σ *satisfies* the predicate p if $\sigma(p) = 1$, it *falsifies* p otherwise, i.e. if $\sigma(p) = 0$.

BMF models enter into the family *combinatorial models*, to which belong the classical fault trees, event trees, reliability block diagrams (see Chapter 5), but also finite degradation models, recently introduced by the author (Rauzy and Yang 2019b).

9.2.3 KMF

The expressive power of BMF is indeed limited: we can describe properties of the system under study according to its state, but not how it reaches these states. The mathematical framework KMF increases dramatically the expressive power of BMF by introducing events and transitions that make it possible to describe the dynamics of systems.

Definition 9.2.3 – KMF (Syntax). A *KMF model* is a quadruple $\langle V, P, T, \iota \rangle$ where:
- V is a finite set of variables, as in BMF models;
- P is a finite set of predicates, as in BMF models;
- T is a finite set of *transitions*, i.e. of triples $\langle e, g, a \rangle$, where:
  - e is a symbol called the *event* labeling the transition. We assume that the event uniquely identifies the transition, i.e. two different transitions are labeled with two different events;
  - g is a Boolean formula built over the variables of V called the *guard* of the transition;
  - a is an *instruction*, i.e. the description of a mechanism that transforms a variable valuation σ into another, *a priori* different, variable valuation $a(\sigma)$.

  For the sake of the clarity, transitions $\langle e, g, a \rangle$ are denoted $g \xrightarrow{e} a$. A transition $g \xrightarrow{e} a$ is *enabled* in the *state* σ, i.e. the variable valuation σ, if $\sigma(g) = 1$. *Firing* the transition $g \xrightarrow{e} a$ in the state σ transforms the state σ into the state $a(\sigma)$.
- ι is the *initial state*, i.e. the initial variable valuation, of the model.

KMF is named so after the American philosopher and logician Saul Kripke (1940-), who introduced a semantics for modal logic, now called Kripke semantics (Kripke 1963).

The semantics of KMF models is defined as their set of executions.

Definition 9.2.4 – KMF (Semantics). Let $M : \langle V, P, T, \iota \rangle$ be a KMF model. The set of *executions* of M is the smallest set such that:
- The empty execution ι is an execution of M.
- If $S : \sigma_0 \xrightarrow{e_1} \sigma_1 \cdots \xrightarrow{e_n} \sigma_n$, $n \geq 0$, is an execution of M and $g \xrightarrow{e} a$ is a transition of T enabled in σ_n, i.e. such that $\sigma_n(g) = 1$, then $S \xrightarrow{e} a(\sigma_n)$ is an execution of M.

A state, i.e. a variable valuation, is *reachable* from the initial state ι if there is an execution of S ending in σ, which we denote $\iota \xrightarrow{*} \sigma$. More generally, we say that the state τ is reachable from the state σ, which we denote $\sigma \xrightarrow{*} \tau$, if there is an execution starting in σ and ending in τ.

According to our assumptions, the domain of each variable v of V is finite. Consequently, the set of possible of variable valuations is also finite, namely it is equal to $\mathrm{dom}(V) = \prod_{v \in V} \mathrm{dom}(v)$, i.e. the Cartesian product of the domains of the variables of V. It may be the case however, that not all these states are reachable from the initial state. This leads to the definition of the reachability graph of a KMF model.

Definition 9.2.5 – KMF (Reachability Graph). Let $M : \langle V, P, T, \iota \rangle$ be a KMF model. The *reachability graph* of M is the smallest graph $\langle \mathscr{V}, \mathscr{E} \rangle$, where $\mathscr{V}$ is a set of states (variable valuations) and $\mathscr{E}$ is a set of labeled edges, such that:
- The initial state ι is a state of $\mathscr{V}$.
- If σ is a state of $\mathscr{V}$ and $g \xrightarrow{e} a$ is a transition of T enabled in σ, then $\tau = a(\sigma)$ is a state of $\mathscr{V}$ and $\sigma \xrightarrow{e} \tau$ is an edge of $\mathscr{E}$.

In each state σ of the reachability graph, some of the predicates of P are satisfied, some other are falsified. The reachability graph of the model M is called a *Kripke structure*.

Note that it would have been possible to define the semantics of KMF models in terms of Kripke structures. However, this raises technical problems when introducing timed and stochastic semantics. This is the reason why we chose to define it in terms of sets of executions.

KMF models enter into the large family of *state automata*, to which belong for instance Petri nets (Murata 1989), Moore and Mealy machines (Hopcroft, Motwani, and J. D. Ullman 2006), input/output automata (Lynch and Tuttle 1989), as well as guarded transition systems (Batteux, Prosvirnova, and Rauzy 2017a; Rauzy 2008b), the underlying mathematical framework of AltaRica 3.0.

9.2.4 TMF

The expressive power of KMF is much higher than the one of BMF. Still, there are features of systems that are impossible, or at least not easy, to capture with KMF. There are actually, two categories of such features.

First, continuous phenomena. KMF descriptions are discrete. Encoding systems of differential equations by means of KMF models would be tedious and fundamentally useless, as there exist excellent modeling environments working directly with systems of differential equations, e.g. Matlab/Simulink (Klee and Allen 2011) and Modelica (Fritzson 2015). This is however not really a problem as—remember our Thesis 5—discrete descriptions are the most suitable at the level of abstraction we consider the systems in (model-based) systems engineering and (model-based) reliability engineering.

Second, the dynamic changes in the architecture of the system. We think here especially to systems of systems (Maier 1998), such as the battlefield or networks of mobile components. In these systems, components enter (or are created) and get out (or are destroyed) of the system dynamically. Moreover, the relationships between components may also change dynamically. It is possible to some extent to twist mathematical frameworks such as KMF to represent such systems, but it turns quickly to be tedious and error prone (Kloul, Prosvirnova, and Rauzy 2013).

If it is possible to give a relatively unified presentation of combinatorial models and state automata, such a unification task is nearly impossible for mathematical frameworks that make it possible to represent the dynamic creation and destruction of components. There is for instance little in common between colored Petri nets (Jensen 2014), Milner's π-calculus (Milner 1999), and agent-based languages such as NetLogo (Wilensky and Rand 2015).

TMF extends KMF by allowing actions of transitions to add and to remove variables, predicates and transitions to the model.

Definition 9.2.6 – TMF (Syntax). A *TMF model* is a quadruple $\langle V, P, T, \sigma \rangle$ where:
- V is a finite set of variables;
- P is a finite set of predicates;
- T is a finite set of *transitions*, i.e. of triples $\langle e, g, a \rangle$, where:
  - e is a symbol called the *event* labeling the transition. We assume that the event uniquely identifies the transition, i.e. two different transitions are labeled with two different events;
  - g is a Boolean formula built over the variables of V called the *guard* of the transition;
  - a is an *instruction*, i.e. the description of a mechanism that transforms the TMF model $\langle V, P, T, \sigma \rangle$ into another TMF model $\langle V', P', T', \sigma' \rangle = a(V, P, T, \sigma)$.
- σ is a valuation of variables of V.

As for KMF, the semantics of TMF is defined in terms of sets of executions.

Definition 9.2.7 – TMF (Semantics). Let $M : \langle V, P, T, \sigma \rangle$ be a TMF model. The set of *executions* of M is the smallest set such that:

- The empty execution $\langle V, P, T, \sigma \rangle$ is an execution of M.
- If $S : \langle V_0, P_0, T_0, \sigma_0 \rangle \xrightarrow{e_1} \langle V_1, P_1, T_1, \sigma_1 \rangle \cdots \xrightarrow{e_n} \langle V_n, P_n, T_n, \sigma_n \rangle$, $n \geq 0$, is an execution of M and $g \xrightarrow{e} a$ is a transition of T_n enabled in σ_n, then $S \xrightarrow{e} a(\langle V_n, P_n, T_n, \sigma_n \rangle)$ is an execution of M.

A state, i.e. TMF model $\langle V, P, T, \sigma \rangle$, is *reachable* from the initial state $\langle V_0, P_0, T_0, \sigma_0 \rangle$ if there is an execution of S ending in $\langle V, P, T, \sigma \rangle$, which we denote $\langle V_0, P_0, T_0, \sigma_0 \rangle \xrightarrow{*} \langle V, P, T, \sigma \rangle$.

In a word, a TMF model has the capacity to transform itself.

9.2.5 Discussion

From the above developments, it is clear that KMF is strictly more expressive than BMF. A BMF model is actually a KMF model with no transition (and no initial state). Moreover, except for specific cases, it is impossible to describe all executions of a KMF model within a BMF model, because the number of executions of a KMF model is infinite, while the number of states of a BMF model is finite.

Similarly, TMF is strictly more expressive than KMF. A KMF model is actually a TMF model in which the sets of variables, predicates and transitions stay the same throughout executions. Moreover, it is impossible to describe all executions of a TMF model within a KMF model, because executions can lead to arbitrary large sets of variables, predicates and transitions, while KMF models are finite.

9.3 Formal Problems of Reliability Engineering

In the previous section, we sketched a taxonomy of mathematical frameworks used in reliability engineering and introduced the three frameworks BMF, KMF and TMF. We shall now review the main problems we can state within these frameworks.

9.3.1 Categories of problems

So far we spoke about problems (stated in mathematical terms) in general. It is worth to be more specific and to distinguish problems according to the type and the size of answers. The following list is not restrictive.

- *Decision problems* are problems whose answers are either *yes* or *no*.
- *Optimization problems* are problems that consists in finding the best, according to some criterion, solution (yes answer) to a decision problem.
- *Counting problems* are problems that consists in counting the number of solutions, i.e. of yes answers, of a decision problem.
- *Reliability problems* are problems that consists in calculating the probability of yes answers of a decision problem, assuming a probability measure defined over solutions.
- More generally, *Function problems* are problems that consists in calculating the value of a certain function of the instance. They may differ from the above ones in that the size of the result may be significantly (exponentially) larger than the size of the instance. Extracting all of the solutions of a decision problem enters into this category.

We shall now illustrate these different categories of problems, using BMF, KMF and TMF.

9.3.2 Satisfiability

The problem SAT, for satisfiability of Boolean formulas in conjunctive normal form, is central in computational complexity theory, as we shall see Section 9.5.

Definition 9.3.1 – Conjunctive Normal Form. Let V be a set of Boolean variables, i.e. of variables whose domain is $\{0,1\}$. Then:
- A *literal* is either a variable v of V or its negation $\neg v$.
- A *clause* is a disjunction (an or) of literals.
- Finally, a Boolean formula is in *conjunctive normal form* if it is conjunction (an and) of clauses.

The problem SAT is stated as follows.

Definition 9.3.2 – SAT. Let f be a Boolean formula in conjunctive normal form built over a set of variables V.

Is f satisfiable, i.e. is there a valuation σ of the variables of V such that $\sigma(f) = 1$?

SAT is obviously a special case of the following problem.

Definition 9.3.3 – BMF-SATISFIABILITY. Let $\langle V, P\rangle$ be a BMF model and a predicate p of P. It there a valuation σ of the variables of V that satisfies p?

We kept the same name for both problems as they are actually equivalent, in terms of computational complexity. It is clear that, as SAT is a special case of BMF-SATISFIABILITY, if we are able to solve efficiently this latter problem, we are certainly able to solve efficiently SAT. It turns out that the reverse is true as well.

In terms of systems engineering, BMF-SATISFIABILITY can be stated as: Is there a state of the system in which a certain property is verified?

9.3.3 Reachability, Deadlocks and Liveness

KMF-REACHABILITY is the KMF counterpart of BMF-SATISFIABILITY. It can be stated as follows.

Definition 9.3.4 – KMF-REACHABILITY. Let $\langle V, P, T, \iota\rangle$ be a KMF model and let p be a predicate of P. Is there an execution $\iota \xrightarrow{*} \sigma$ leading to a state σ in which p is satisfied?

KMF-REACHABILITY is of primary importance in reliability engineering. It can be reformulated as: is there an execution leading to a state having a given property, typically characterizing an abnormal behavior, e.g. failure, conflictual access to as a resource...

TMF-REACHABILITY is defined is the same way, considering TMF models instead of KMF ones.

A slight variation on KMF-REACHABILITY is TMF-DEADLOCK which can be stated as follows.

Definition 9.3.5 – KMF-DEADLOCK. Let $\langle V, P, T, \iota\rangle$ be a KMF model and let p be a predicate of P. Is there an execution $\iota \xrightarrow{*} \sigma$ leading to a state σ in which no transition is enabled.

The existence of deadlocks is in general symptomatic of a problem: normally, a system should always be able to evolve.

KMF-LIVENESS is another important problem. It can be stated as follows.

Definition 9.3.6 – KMF-LIVENESS. Let $\langle V, P, T, \iota\rangle$ be a KMF model and let p be a predicate of P. Then, p has the liveness property if there exist a state σ in which p is satisfied and verifying the two following conditions:
- σ is reachable from the initial state ι ($\iota \xrightarrow{*} \sigma$).
- σ is reachable from itself by means of a non-empty execution ($\sigma \xrightarrow{*} \sigma$).

In other words, σ belongs to a loop, and it possible to come back as many time as one wants to this state.

TMF-DEADLOCK and TMF-LIVENESS are the straightforward extensions of KMF-DEADLOCK and KMF-LIVENESS to TMF models.

Note that KMF/TMF-REACHABILITY, KMF/TMF-DEADLOCK and KMF/TMF-LIVENESS, as stated above, are decision problems, i.e. receive a yes-no answer. In practice, we are in general interested not only in the existence of an execution verifying this or that property, but in an example of such execution.

Note also that is possible to extend slightly the definition of these problems to introduce constraints on the execution verifying the property. These constraints may apply on the states and transitions encountered along the execution. The vast literature on model-checking deals with such constraints, often described by means of temporal logics, see e.g. (Clarke, Grumberg, Kroening, et al. 2018).

BMF-SATISFIABILITY, KMF/TMF-REACHABILITY, KMF/TMF-DEADLOCK, and KMF/TMF-LIVENESS can be used to study properties of the system under study, typically by making the predicate p describing the failed states. However this use is relatively limited in the context of reliability engineering: we know that the system may fail and it is precisely the reason why we assess its reliability.

More interestingly, BMF-SATISFIABILITY, KMF/TMF-REACHABILITY, KMF/TMF-DEADLOCK, and KMF/TMF-LIVENESS can be use to validate the model, by performing various tests on it. As we shall see in the second part of this book, having means to validate models is of primary importance in systems engineering.

9.3.4 BMF-Availability

As pointed out above, in reliability engineering, we are in general not interest in determining whether the system under study can fail: we know upfront that it can. Rather, we are interested in the likelihood of this event. Likelihood is however an informal notion. To formalize it and make it quantifiable, we must dive into the realm of probabilities, which, as we shall see, is technically more demanding that it may seem at a first glance.

Let $\langle V,P\rangle$ be a BMF problem. Intuitively, we want to associate a probability $pr(v=c)$ to each pair made of a variable v of V and a constant c of $\mathrm{dom}(v)$, i.e. a probability for a component to be in a given state. Then, assuming that changes of states of components are statistically independent, we want to assess the probability that the system as a whole fails, or more generally the probability that it is in a state characterized by a given predicate.

Let pr be a function that associates with each variable $v \in V$ and each constant $c \in \mathrm{dom}(v)$ a real number $pr(v=c)$ such that:

(i) For all $v \in V$ and $c \in \mathrm{dom}(v)$, $0 \leq pr(v=c) \leq 1$;

(ii) For all $v \in V$, $\sum_{c\in\mathrm{dom}(v)} pr(v=c) = 1$.

Then it is possible to lift-up pr into a function from $\mathrm{dom}(V)$ into $[0,1]$ by pausing:

(iii) For all $\sigma \in \mathrm{dom}(V)$, $pr(\sigma) = \prod_{v\in V} pr(v=\sigma(v))$.

Informally, pr assumes, by the above condition, that $v_1 = c_1$ and $v_2 = c_2$ are statistically independent for all $v_1, v_2 \in V$, $v_1 \neq v_2$ and all $c_1 \in \mathrm{dom}(v_1)$ and $c_2 \in \mathrm{dom}(v_2)$.

Finally, it is possible to lift-up pr into a function from $2^{\mathrm{dom}(V)}$, i.e. the set of subsets of variable valuations of V, into $[0,1]$ by pausing:

(iv) For all $E \in 2^{\mathrm{dom}(V)}$, $pr(E) = \sum_{\sigma\in E} pr(\sigma)$.

Now, $2^{\mathrm{dom}(V)}$ is clearly a σ-algebra (see Appendix C), as:

– It is non empty.

– It is closed by complementation and (countable) union.

Moreover a function pr defined as above, i.e. verifying conditions (i)-(iv), is a *probability measure* over $2^{\mathrm{dom}(V)}$ as:

– $\forall E \in 2^{\mathrm{dom}(V)}\ 0 \leq pr(E) \leq 1$;
– $pr(2^{\mathrm{dom}(V)}) = 1$;
– $\forall E, F \in 2^{\mathrm{dom}}(V)$ such that $E \cap F = \emptyset$ then $pr(E \cup F) = pr(E) + pr(F)$.

Definition 9.3.7 – Stochastic BMF. A *stochastic BMF* model is a triple $\langle V, P, pr \rangle$, where:
– $\langle V, P \rangle$ is a BMF model
– Let pr be a mechanism (an instruction, a table) defining a probability measure over V defined as above, i.e. verifying the above conditions (i)-(iv).

We can now state BMF-AVAILABILITY, the central problem of reliability engineering.

Definition 9.3.8 – BMF-AVAILABILITY. Let $\langle V, P, pr \rangle$ be a stochastic BMF model and a predicate p of P. Then, what is the probability of p, i.e. $pr(p)$?

As we shall see, the SAT version of BMF-AVAILABILITY was first considered (from a computational complexity point of view) by Valiant (Valiant 1979b). Valiant named this problem RELIABILITY. This is unfortunate because, as we shall see in the next section availability and reliability are different notions in reliability engineering and what Valiant meant was availability rather than reliability. We chose here a naming compatible with reliability engineering use, for the sake of coherence with the remainder of the book.

9.3.5 KMF/TMF-Availability and KMF/TMF-Reliability

As pointed out in the discussion of Section 9.3.3, REACHABILITY and related problems are not of direct interest in the context of reliability engineering, but as a means to check the validity of models. What is however of great interest is to assess the likelihood of scenarios of evolution of the system under study, typically the likelihood to reach a failure state within a given mission time. Ideally, one would define a probability space on top of KMF and TMF models, as we have done for BMF models, and then assess the probability to reach a state verifying a certain predicate p within a given mission time. There are however two technical difficulties here: first, probabilities must be carried out by transitions; second, one must formalize what means "within a given mission time".

There are actually two quite different although indeed related ways of weighting executions of KMF and TMF models, i.e. to express their likelihood: the discrete-time approach and the continuous-time approach. Moreover, within the continuous-time approach, it is worth to distinguish Markovian and non-Markovian models. We shall review them in turn, for KMF models.

Discrete-time models

In the discrete-time model, transitions are associated with a probability. Let $\langle V, P, T, \iota \rangle$ be a KMF model. We consider (mechanisms implementing) functions pr from $\mathrm{dom}(V) \times T$ into real numbers such that the following conditions hold.

(i) For any state σ and any transition t of T, $0 \leq pr_\sigma(t) \leq 1$.
(ii) For any state σ of the model, $\sum_{t \in T,\, t \text{ enabled in } \sigma} pr_\sigma(t) = 1$.

Definition 9.3.9 – Discrete-Time Stochastic KMF. A *discrete-time stochastic KMF model* is a quintuple $\langle V, P, T, \iota, pr \rangle$ where:
– $\langle V, P, T, \iota \rangle$ is a KMF model.
– pr is a mechanism implementing function from $\mathrm{dom}(V) \times T$ into real numbers obeying the conditions (i) and (ii) given above.

The probability of an execution is then defined as follows.
- The probability $pr_0(\iota)$ of the empty execution is 1.
- If $S : \sigma_0 \xrightarrow{e_1} \sigma_1 \cdots \xrightarrow{e_n} \sigma_n$, $n \geq 0$, is an execution of M and $t : g \xrightarrow{e} a$ is a transition of T enabled in σ_n, then $pr_{n+1}\left(S \xrightarrow{e} a(\sigma_n)\right) = pr_n(S) \times pr_{\sigma_n}(t)$.

Note that $pr_\sigma(t)$ does not depend on the step.

It is easy to verify that the function pr_n as defined above is a probability measure over the set of n-steps executions of the KMF model $\langle V, P, T, \iota \rangle$.

We can now define the probabilities of states and predicates in n-steps executions as follows.
- The probability $pr_n(\sigma)$ to be in the state σ after a n-steps execution is the sum of the probabilities of n-steps executions ending in this state.
- The probability $pr_n(p)$ to satisfy a predicate p after a n-steps execution is the sum, over the states that satisfy the predicate, of the probabilities to be in these states by means of a n-steps execution.

At this point we can make the following remarks.
- The number of reachable states at step n is always finite. This is true not only for KMF models, for which the number of states is anyway finite, but also for TMF models.
- Because $pr_\sigma(t)$ does not depend on the step, $pr_n(\sigma)$ depends only the probabilities $pr_{n-1}(\sigma')$ and the $pr_\sigma(t)$ for transitions $\sigma' \xrightarrow{t} \sigma$. Namely, for all $n > 0$ state σ reachable from the initial state in n steps, the following equality holds.

$$pr_n(\sigma) = \sum_{t \in T; \sigma' \xrightarrow{t} \sigma} pr_{n-1}(\sigma') \times pr_{\sigma'}(t) \tag{9.1}$$

In other words, discrete-Time Stochastic KMF and TMF models are implicitly defined discrete-time Markov chains.

We can now state KMF-AVAILABILITY problem for the discrete-time models:

Definition 9.3.10 – DTKMF-AVAILABILITY. Let $\langle V, P, T, \iota, pr \rangle$ be a discrete-time stochastic KMF model, let p be a predicate of P and let n be an integer. Then, what is the probability of p at step n, i.e. $pr_n(p)$?

DTKMF-RELIABILITY is a follows.

Definition 9.3.11 – DTKMF-RELIABILITY (discrete-time models). Let $\langle V, P, T, \iota, pr \rangle$ be a discrete-time stochastic KMF model, let p be a predicate of P and let n be an integer. Then, what is the probability that p is satisfied at any step i, $0 \leq i \leq n$?

In other words, the availability of a system of step n is the probability that it works at step n, while its reliability is the probability that it worked continuously from step 0 to step n.

As previously, it is straightforward to extend DTKMF-AVAILABILITY and DTKMF-RELIABILITY into respectively DTTMF-AVAILABILITY and DTTMF-RELIABILITY applying to discrete-time TMF models.

DTKMF-AVAILABILITY and DTKMF-RELIABILITY have a clear and simple mathematical formulation. Moreover, they can be used to solve some practical reliability engineering problems. The idea is to consider time steps dt, sufficiently small for that only one transition can be fired during the time step. It is then possible to determine the probability for each transition to be fired in this small time interval and to take the number of steps n sufficiently large so that $n \times dt$ corresponds to the mission time. Numerical solutions of Markov chains rely for instance on this principle, see e.g. (Stewart 1994) for a reference book.

This small steps approach is however not very suitable in reliability engineering where events causing transitions (like failures and repairs of components) are scarce. This is the reason why continuous-time models are in general preferred, at least at system level.

Continuous-time models

Continuous-time models consist in associating a delay with each transition. Delays implement inverse functions of cumulative probability distribution. Recall that a *cumulative probability distribution* is a non-decreasing, in our case invertible, function from the set of non-negative real numbers $\mathbb{R}^+$ into the real interval $[0,1]$. see Appendix C for more details and examples of cumulative probability distributions.

Technically, a *delay* δ is a mechanism that given a transition t, a state σ, a date $d \in \mathbb{R}^+$ and a real z in $[0,1]$ such that:

(i) For all $z_1, z_2 \in [0,1]$, $z_1 < z_2 \Rightarrow \delta(t,\sigma,d,z_1) \leq \delta(t,\sigma,d,z_2)$.

Note that we assume here that δ is fully deterministic. The stochasticity of delays is taken into account via the parameter z.

Definition 9.3.12 – Continuous-Time Stochastic KMF (Syntax). A *continuous-time stochastic KMF model* is a quintuple $\langle V,P,T,\iota,\delta\rangle$ where:
- $\langle V,P,T,\iota\rangle$ is a KMF model.
- δ is a mechanism verifying the condition (i) given above.

Conversely to the discrete-time case, the semantics of KMF models is changed when considering the continuous-time case. It relies on the notion of schedule. A *schedule* is a mechanism γ that associates a number in $\mathbb{R}^+ \cup \{\infty\}$ with each transition. The idea here is to associate infinite firing dates to non enabled transitions.

Executions of continuous-time stochastic KMF models are sequences of the form $\langle d_0 = 0, \sigma_0 = \iota, \gamma_0\rangle \xrightarrow{e_1} \langle d_1,\sigma_1,\gamma_1\rangle \cdots \xrightarrow{e_n} \langle d_n,\sigma_n,\gamma_n\rangle$, $n \geq 0$, where the d_i's are dates (non-negative real numbers), σ_i's are states (variable valuations), the γ_i's are schedules, and the e_i's are events labeling transitions. Formally:

Definition 9.3.13 – Continuous-Time Stochastic KMF (Semantics). Let $M : \langle V,P,T,\iota,\delta\rangle$ be a continuous-time stochastic KMF model. The set of *executions* of M is the smallest set such that:
- The empty execution $\langle 0,\iota,\gamma_0\rangle$ is an execution of M if for all transitions $t \in T$, $\gamma_0(t) = \delta(t,\iota,0,z)$ for some $z \in [0,1]$ if t is enabled in ι and ∞ otherwise.
- If $S : \langle d_0,\sigma_0,\gamma_0\rangle \xrightarrow{e_1} \langle d_1,\sigma_1,\gamma_1\rangle \cdots \xrightarrow{e_n} \langle d_n,\sigma_n,\gamma_n\rangle$, $n \geq 0$, is an execution of M and $t : g \xrightarrow{e} a$ is a transition of T such that t is enabled in σ_n and for all $t' \neq t$ of T, $\gamma_n(t) \leq \gamma_n(t')$: then $S \xrightarrow{e} \langle d,\sigma,\gamma\rangle$ is an execution of M if:
  - $\sigma = a(\sigma_n)$;
  - $\gamma(t) = d + \delta(t,\sigma,d,z)$ for some $z \in [0,1]$ if t is enabled in σ and ∞ otherwise;
  - For all $t' \neq t$ of T:
    - If t' is enabled both in σ_n and σ, then $\gamma(t') = \gamma(t)$,
    - If t' is enabled in σ but not in σ_n, then $\gamma(t') = d + \delta(t',\sigma,d,z)$ for some $z \in [0,1]$,
    - If t' is not enabled in σ, then $\gamma(t') = \infty$.

We can restrict our attention to models delays are exponentially distributed. In this case, they have the memoryless propery, which is another way to say that continuous-time stochastic KMF/TMF models with exponentially distributed delays are actually implicitly defined continuous-time Markov chains.

We can now state CTKMF-AVAILABILITY and CTKMF-RELIABILITY for the continuous-time models.

CTKMF-AVAILABILITY is as follows.

Definition 9.3.14 – CTKMF-AVAILABILITY. Let $\langle V, P, T, \iota, \delta \rangle$ be a continuous-time stochastic KMF model, let p be a predicate of P and let d be a date (a mission time). Then, what is the probability that p is satisfied at the date d?

CTKMF-RELIABILITY is a follows.

Definition 9.3.15 – CTKMF-RELIABILITY. Let $\langle V, P, T, \iota, \delta \rangle$ be a continuous-time stochastic KMF model, let p be a predicate of P and let d be a date (a mission time). Then, what is the probability that p is satisfied without interruption from date 0 to date d?

As previously, it is straightforward to extend CTKMF-AVAILABILITY and CTKMF-RELIABILITY into respectively CTTMF-AVAILABILITY and CTTMF-RELIABILITY applying to continuous-time TMF models.

9.3.6 Minimal Solutions

Let $\langle V, P \rangle$ be a BMF model and let p be a predicate of P. One is often interested in solutions of p, i.e. in variable valuations that satisfy P. There can be indeed a huge number of solutions. Consequently, one is more specifically interested in solutions of p that satisfy some minimality criterion. Intuitively, one would like to consider solutions that deviate the less from the normal state of the system, or to put it differently, that involve as few failures as possible, if the predicate p characterizes failed states. The question is indeed to give a mathematical formulation to this idea.

In the fault tree analysis context (that we shall discuss in full extent Chapters 5 and 6), minimal solutions are called *minimal cutsets*. I gave a formal definition of this notion in my 2001 article (Rauzy 2001).

It is only recently that I generalized it to BMF models, via the notion of finite degradation structures (Rauzy and Yang 2019a; Rauzy and Yang 2019b). The key idea is to consider that domains of variables are partially ordered and that they have a least element representing the normal state of the component represented by the variable. Here follows some examples of such domains:

- $\{$working, failed$\}$ with working $<$ failed;
- $\{$working, degraded, failed$\}$ with working $<$ degraded $<$ failed;
- $\{$working, failed-safe, failed-dangerous$\}$ with working $<$ failed-safe and working $<$ failed-dangerous;

Formally:

Definition 9.3.16 – Finite Degradation Structure. A *finite degradation structure* is a finite meet semi-lattice, i.e. a finite, partially ordered set with a unique least element.

Now, we can define the product of two finite degradation structure:

Definition 9.3.17 – Products of Finite Degradation Structure. Let $\langle D_1, <_1 \rangle$ and $\langle D_2, <_2 \rangle$ be two finite degradation structures. Then, their product, denoted $\langle D_1, <_1 \rangle \otimes \langle D_2, <_2 \rangle$, is the finite degradation structure $\langle D, < \rangle$ such as:

- $D = D_1 \times D_2$;
- $\langle x_1, y_1 \rangle < \langle x_2, y_2 \rangle$ if either $x_1 <_1 x_2$ and $y_1 \leq_2 y_2$, or $x_1 \leq_1 x_2$ and $y_1 < y_2$.

It is easy to verify that the product of finite degradation structures is commutative and associative, up to an isomorphism. Consequently, if $\langle V, P \rangle$ is a BMF model such that the domains of the variables of V are finite degradation structures, then $\mathrm{dom}(V)$ is itself a finite degradation structure. As elements of $\mathrm{dom}(V)$ one to one correspond with variable valuations of V, we can define a notion of minimal solutions.

Definition 9.3.18 – Minimal Solutions. Let $\langle V, P\rangle$ is a BMF model such that the domains of the variables of V are finite degradation structures. Let $<$ denote the resulting partial order on elements of $\mathrm{dom}(V)$ (and variable valuations of V). Let, finally, p be a predicate of P.

Then, a minimal solution of p is a variable valuation σ such that:

- σ is a solution of p, i.e. $\sigma(p) = 1$;
- No other solution of p is smaller than σ, i.e. $\forall \tau \in \mathrm{dom}(V),\ \tau < \sigma \Rightarrow \tau(p) = 0$.

BMF-MINIMALSOLUTIONS is the problem of extracting the minimal solutions of such a BMF-model.

9.3.7 Minimal Traces

As previously, we would like to lift-up BMF-MINIMALSOLUTIONS to KMF and TMF models. This requires defining a partial order on executions, i.e. eventually on sequences of events labeling the transitions.

Let $\langle V, P, T, \iota\rangle$ be a KMF (or TMF) model, and let E be the set of events labeling the transitions of T. The sequence $e_1 e_2 \cdots e_n$ of events labeling the transitions of an execution $\iota \xrightarrow{e_1} \cdots \xrightarrow{e_n} \sigma_n$ is *word* on the *alphabet* E. The set of words that can be built over the alphabet E is denoted $E^\star$.

In automata and language theory, there are two concepts of subword: the ones of substring and subsequence.

Definition 9.3.19 – Substrings and Subsequences. Let Σ be a finite alphabet and let u and v be two words built on Σ. Then,

- u is a *substring* of v if there exist two, possibly empty, words p and q built over Σ such that $v = puq$.
- u is a *subsequence* of v if u can be obtained from v by deleting some letters.

Consider for instance the word $w = model$, then ode is both a substring and a subsequence of w, while mdl is only a subsequence of w.

In our context, the notion of subsequence is clearly preferable to the one of substring. Still, it is probably insufficient as it does account for independent events like independent failures of components. For such events, the order in which they occur does not matter. This leads us to the concept of trace (Diekert and Rozenberg 1995).

Definition 9.3.20 – Traces. Let Σ be a finite alphabet.

1. A *independence relation* on Σ is a symmetric, irreflexive relation $\sim$ over the letters of Σ.
2. Two words u and v built over Σ are *equivalent* for the independence relation $\sim$, which is denoted $u \simeq v$, if v can be obtained from u by a series of permutations of independent adjacent letters.
3. It is easy to verify that $\simeq$ is a reflexive, symmetric and transitive relation, i.e. an equivalence relation, over $\Sigma^\star$. The classes of this equivalence relation are called the *traces* of alphabet Σ equipped with the independence relation $\sim$.

In practice, it is in general more convenient to define a dependence relation $\nsim$ over the events of a KMF-model. The corresponding independence relation $\sim$ is simply the complementary of $\nsim$: $u \sim v$ if and only if $\neg u \nsim v$. Note also that $\nsim$ can be automatically extracted from the model by means of static analysis, i.e. without executing the model. The compiler of AltaRica into systems of Boolean equations implements such a decomposition of the model (Prosvirnova and Rauzy 2015; Rauzy 2002).

We can now combine the concepts of subsequences and traces to get the expected partial order relation over executions of a KMF model.

Definition 9.3.21 – Minimal Traces. Let $\langle V, P, T, \iota \rangle$ is a KMF model and E be the set of events labeling transitions of T. Let $\sim$ be an independence relation over the events of E and let $\simeq$ the corresponding equivalence relation over $E^\star$. Let finally u and v be two words built over E. Then, u is a subtrace of v, which is denoted $u \precsim v$ if there exists a word w built over E such that the two following conditions hold.

(i) $w \simeq v$;

(ii) u is a subsequence of w.

As usual, $u \prec v$ if $u \precsim v$ and not $u \simeq v$.

Now let p be a predicate of P, let S be an execution of the model and let u be the sequence of events of E labeling the transitions of S, i.e. $S = \iota \xrightarrow{u} \sigma$ for some variable valuation σ. Then,

- u is a *satisfying trace*, or simply a trace, of p, if $\sigma(p) = \text{true}$;
- u is a *minimal satisfying trace*, or simply a minimal trace, of p if it is a trace of p and there is no other trace v of p such that $v \prec u$.

KMF-MINIMALTRACES (respectively TMF-MINIMALTRACES) is the problem of extracting the minimal traces of a KMF model (respectively a TMF model).

Note that, if we assume in addition that domains of variables of V are finite degradation structures, we could reinforce the minimality condition by saying that u is a minimal trace of p if it is a trace of p and there is no execution $\iota \xrightarrow{v} \tau$ such that v is a trace of p and either $v \prec u$, or $v \simeq p$ and $\tau < \sigma$.

This last remark concludes our tour of key formal problems of reliability engineering.

9.4 Turing Machines and Decidability

We are now in position to introduce computational complexity theory, starting from Turing machines.

9.4.1 Turing Machines

Turing machines are a mathematical model of computation, i.e. both they model simultaneously the computer and the algorithm. They have been introduced by the English mathematician of genius Alan Turing in his 1936 article (Turing 1936). A Turing machine operates mechanically on a tape. On each cell of this tape there is at most one symbol. The machine can read and write these symbols, one at a time, using a tape head that points onto a cell. More formally:

Definition 9.4.1 – Turing machine. A *Turing machine* is made of the following elements.

- A *tape* divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains at least one special blank symbol and one or more other symbols. The tape is assumed to be arbitrarily extendable to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written before are assumed to be filled with the blank symbol.
- A *head* that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time.
- A finite set of *states*. This set contains at least one state, the initial state of the computation.
- A finite table of *instructions* that, given the state the machine is currently in and the symbol it is reading on the tape, tells the machine to do the following in sequence:
  - Write another (or the same) symbol in the cell the head is currently on.
  - Move the head to the left (if possible), to the right or keep it at the same place.
  - Define the new state (which can be the same as the current one).

Initially, the tape may contain a finite number of non-blank symbols, which describe the

input of the algorithm, and the head is positioned on the left-most cell of the tape. Performing the computation consists then in executing instruction after instruction, according to the current combination of state and symbol. The computation halts when there is no entry in the table for the current combination. The sequence of symbols written on the tape starting from the left-most cell of the tape to the right-most cell containing a non-blank symbol describes the output of the algorithm.

Figure 9.2 shows a Turing machine whose table of instructions is represented as a finite state automaton.

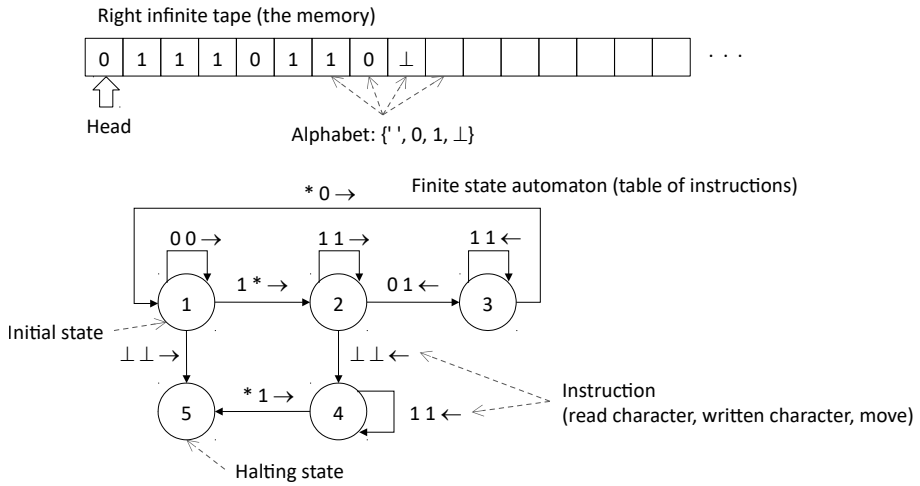


Figure 9.2: A Turing machine

Every part of the machine, i.e. its alphabet, set of states, table of instructions, and used tape at any given time, is finite, discrete and distinguishable. It is only the unlimited amount of tape and runtime that gives it an unbounded amount of storage space. It follows that a Turing machine can be described by means of finite sequence of symbols and thus be given as input another Turing machine (and, by to the way, to itself too).

In his seminal article, Turing showed the following theorem.

Theorem 9.1 – Universal Turing machines. There exists universal Turing machines, i.e. Turing machines U such that for any other Turing machine A and input data d, the execution of U on the input (A,d) gives the same result as the execution of A on d. Moreover, the number of steps of the execution of U on (A,d) is polynomially related to the number of steps of the execution of A on d.

This theorem is a mathematical confirmation of a very intuitive, yet concrete, remark: the computers we are daily using are universal in this sense that we can program them and that everything we can program on computer A can be adjusted to run on computer B. This idea is formalized and enriched by the Church-Turing thesis.

Thesis 8 – Church-Turing thesis. If we call *algorithm* a method each step of which is precisely predetermined and which is certain to produce the answer in a finite number of steps, then, a function on the natural numbers is computable by means of an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.

In other words, anything which is computable is computable by a Turing machine. Note that the above statement is a thesis and not a theorem. The definition of algorithm is actually too large to make possible a formal proof. Nevertheless, it is widely accepted as a valid result by

mathematicians and even taken as a definition: we call computable what is computable by a Turing machine. Non-computable functions have been exhibited, like the "busy beaver function", but they are only mathematical curiosities.

As a consequence of Church-Turing thesis, we can use liberally any reasonable model of computation, including the pseudo-code in which we describe algorithm in this book, to study the feasibility of experiments.

9.4.2 Decidability

The goal of Turing, when introducing what is now called Turing machines, was to design a simplest and more appealing proof of the two incompleteness theorems Kurt Gödel had shown in 1931 (Gödel 1951). Informally, the first incompleteness theorem states that no consistent system of axioms and deduction rules whose theorems can be listed by an algorithm is capable of proving all truths about the arithmetic of the natural numbers. A system of axioms and deduction rules is *consistent* if it is not possible to prove a proposition P and its opposite $\neg P$ within the system. For any such system, there will always be statements about the natural numbers that are true, but that are unprovable within the system. The second incompleteness theorem, an extension of the first, shows that the system cannot demonstrate its own consistency.

Turing's idea was to show that no algorithm can be designed to solve the Halting problem:

Definition 9.4.2 – HALTING PROBLEM. Let A be a Turing machine. Does A halt on any input d?

Turing proved the following theorem.

Theorem 9.2 – Undecidability of the Halting problem. The Halting problem is *undecidable*, i.e. there cannot exist an algorithm, i.e. eventually a Turing machine, that given a Turing machine A answers *yes* if A halts on any input d and *no* otherwise.

Turing's proof goes as follows.

Proof. Assume for a contradiction that such Turing machine exists. Let us call it `HALT`.

It is easy to design another Turing machine, let us call it `DIAGONAL`, that given a Turing machine A:

- loops forever if `HALT` answers *yes* on A, and
- answers *no* otherwise.

The pseudo-code for such machine is just as follows.

```
function DIAGONAL(A)
    if HALT(A):
        loop forever
    else:
        return no
```

Now, let us apply `DIAGONAL` on itself:

- If `DIAGONAL` loops forever, it means that `HALT(DIAGONAL)` returns *yes*, i.e. that `DIAGONAL` applied on itself halts. A contradiction.
- If, on the contrary, `DIAGONAL` returns *no*, it means that `HALT(DIAGONAL)` returns *no*, i.e. that `DIAGONAL` applied on itself loops forever. Another contradiction.

It follows that no such Turing machine `HALT` can exist. QED

Turing's proof, as Gödel's one, relies on a *diagonalization* argument. In its simplest form, the diagonalization argument takes the form of the *liar paradox* stated by the Cretan philosopher of the

ancient time Epimenides: "*All Cretans are liar*". Or, in an even simpler form: "*I lie*". The English logician Bertrand Russel gave another version that goes as follows.

The barber shaves all the men of the city that do not shave themselves.

Diagonalization arguments played a very important role in the discovery of foundational problems in mathematics in the 19th century. Using a diagonal argument Cantor showed that the sets $\mathbb{N}$ of natural numbers and $\mathbb{R}$ of real numbers have different sizes and that there are infinitely many sizes of infinite sets. His work led eventually to modern set theory, see e.g. (Pinter 2013) for an introduction. Cantor stated also the so-called "*continuum hypothesis*", which asserts that there is no set strictly bigger than $\mathbb{N}$ and strictly smaller than $\mathbb{R}$. This hypothesis, which is still an open problem, is considered as one of the most, if not the most, important and difficult questions in mathematics.

9.4.3 Has this practical consequences?

Since the introduction of the concept on undecidability, many problems have been shown undecidable, including very concrete ones arising in model-based systems engineering, as we shall see now.

Theorem 9.3 – TMF-REACHABILITY is undecidable. TMF-REACHABILITY is undecidable, i.e. that there cannot exist an algorithm, that given a given any TMF model $\langle V, P, T, \imath \rangle$ and predicate p of P determines whether there exists an execution $\imath \xrightarrow{*} \sigma$ leading to a state σ in which p is satisfied.

Proof. To prove this theorem proof, the idea is to show that for any Turing machine, it is possible to design a TMF model that simulates the execution of the machine and that reaches a state verifying a given predicate if and only if the machine halts on this state. We shall only sketch this proof here.

The key idea is to introduce the following variables.

- A Boolean variable `halt` that is true if and only if the machine halts in the current state.
- A variable `state` to encode the state of the Turing machine.
- A finite set of variable $\texttt{cell}_1, \ldots \texttt{cell}_n$ to encode all non-white cells of the tape.
- Finally a variable `head` that indicates which cell the head is located to.

Transitions of the model will then simulate transitions of the Turing machine, setting the variable `halt` to true, when the machine is in a halting state.

Once this done, the conclusion follows immediately: there exists an execution of the model leading to a state where `halt` is true if and only if the Turing machine halts. As the halting problem is undecidable, TMF-REACHABILITY is *a fortiori* undecidable. QED

As a corollary, all problems related to TMF models listed in the previous section are also undecidable. Of course, this does not prevent to design algorithms to solve these problems in this or that particular case. But general algorithms cannot exist. This explains why TMF models and related modeling formalisms such as agent-based models, colored Petri-nets, or process algebras are seldom used in practical reliability engineering.

Note also that if we accept variables with denumerable domains, e.g. integers, in BMF-models, then BMF-REACHABILITY is also undecidable. This the direct consequence of an result and similar important obtained on Petri nets, see e.g. (Esperza 1998).

All these undecidability results do not prevent however engineers to design models and to perform experiments on these models. But this forces us to a certain modesty: not all of the problems are solvable and even very concrete ones are not, whatever efforts we could make to design technologies to solve them.

The stroke of genius of Turing was to provide, in addition to the notion of decidability and before any (machine) computer exists, a formal framework to analyze the complexity of algorithms and problems, as we shall see now.

9.5 Complexity of Algorithms and Problems

To assess the feasibility of solving mathematical problems, assuming they are decidable, we need a formal notion of computational cost. The concept of worst-case complexity provides such a tool.

9.5.1 Worst-case complexity

We shall start by defining the worst-case complexity of algorithms, i.e. according to what we have seen in the previous section, of the cost of the execution a Turing machine A on an input data d. Input data are written as words, i.e. finite sequences of symbols, over a given alphabet Σ (usually $\Sigma = \{0,1\}$). We can assume, without of generality, that A is able to deal with any such word, possibly by rejecting the input as invalid. We denote by $|w|$ the *size*, i.e. the number of symbols, of a word w.

Definition 9.5.1 – Worst-case complexity of algorithms. Assume given a model of computation and an algorithm A that halts on each input d described by words of a finite alphabet Σ. Then, the mapping $t_A : \Sigma^* \to \mathbb{N}$ is called the *time complexity* of A if, for every $d \in \Sigma^*$, A halts after exactly $t_A(x)$ steps. The *worst-case time complexity* $T_A : \mathbb{N} \to \mathbb{N}$ of A is then defined as:

$$T_A(n) \stackrel{def}{=} \max_{w\in\Sigma^*,|w|=n} t_A(n)$$

Similarly, the mapping $s_A : \Sigma^* \to \mathbb{N}$ is called the *space complexity* of A if, for every $d \in \Sigma^*$, A halts using exactly $s_A(x)$ memory cells. The *worst-case space complexity* $S_A : \mathbb{N} \to \mathbb{N}$ of A is then defined as:

$$S_A(n) \stackrel{def}{=} \max_{w\in\Sigma^*,|w|=n} s_A(n)$$

The above definition makes sense thanks to the existence universal Turing machine: the worst-case complexities of an algorithm within two reasonable models of computation will be the same, up to some polynomial factor. This is indeed a rather coarse-grain characterization, but it is good enough for our purposes. Note moreover that, still in practice, if some computers are much faster than others, the speed-up they provide is always close to linear: a supercomputer may run a million times faster than my laptop, but this ratio is about the same whatever the size of the input data. Therefore, by knowing the time complexity of an algorithm on my laptop, I can deduce the one the supercomputer, and vice-versa.

Consequently, the functions t_A and s_A of the above definition do not need to be known precisely. We are rather interested in their asymptotic behaviors. To do so, we shall use the big-O notation. The big-O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity:

Definition 9.5.2 – Big-O notation. Let f be a real valued function and g a real valued function, both defined on some unbounded subset of the real positive numbers, such that $g(x)$ is strictly positive for all large enough values of x. Then, we say that f is in $\mathcal{O}(g)$ if and only if there exist two positive real numbers x_0 and c such that $f(x) \leq c.g(x)$ for all $x \geq x_0$.

For instance, sorting the element of a list using the quick-sort algorithm is in $\mathcal{O}(n.\log n)$, where n denotes the number of elements of that list.

We can now lift-up our characterization of the cost of algorithms into a characterization of the cost of solving problems. By *problem*, we mean a formalized question on mathematical objects, as

explained Section 9.1. More precisely, a problem is a countable family of words over a given finite alphabet Σ called *instances*. We assume moreover that each instance I of P has a *solution*, itself encoded as a word over Σ, i.e. P can be interpreted as a function from Σ^ω into itself.

Definition 9.5.3 – Worst-case complexity of problems. Let P a problem. Then, the *worst-case complexity* of P is the worst-case complexity of the best algorithm to calculate $P(I)$, for each instance I of P, i.e. the algorithm with the lowest worst-case complexity.

Note that the above definition is not constructive: it does tell how to obtain the best algorithm in question.

Note also that we are speaking here of worst-case complexity. It is also possible to consider average complexity, but results are then much more difficult to establish because this requires to establish a suitable probability measure on problems, then to average the cost with respect to this measure. So far, results obtained into this direction have not reached the level of generality and interest of results obtained for the worst-case complexity.

In the framework of model-based systems engineering, the above definition can be stated as follows. Let $\mathscr{L}$ be a modeling language. $\mathscr{L}$ is nothing but a countable family of words over a certain finite alphabet Σ, the models that we can write with $\mathscr{L}$. We can assume, without a loss of generality that experiments performed of models of $\mathscr{L}$ are also described by means of words over Σ. Therefore, we consider problems that consists of pairs (M, Q) where M is a model of M and Q is a question on M. The solution to such a problem is the result of the experiment. Consequently, it makes sense to speak about cost of experiments.

9.5.2 First complexity classes

It is agreed by mathematicians and computer scientists that decidable problems fall in one of the three main categories, with respect to their complexity.

- Provably easy problems, i.e. those for which algorithms with polynomial complexity are known, i.e. problems in $\mathscr{O}\left(n^k\right)$, for some constant k. These problems are said *P-easy*. Some of them are also *P-hard*, meaning that no algorithm with a lower complexity than polynomial can be designed to solved them.
- Provably hard problems, i.e. those for which it can be proved that any algorithm has at least an exponential complexity. These problems are said *EXP-hard*.
- Problems that are neither provably easy nor provably hard. There is a wide variety of very practical such problems.

The above classification is rather rough as a problem in $\mathscr{O}\left(n^{100}\right)$ can hardly be considered as easy in any practical sense. But very few, if any, such problems have been exhibited so far, so the classification is widely accepted.

Each problem belongs thus to a class that characterizes its complexity.

Definition 9.5.4 – Complexity class P. The class PTIME, or simply P, is the class of decision problems that can be solved in polynomial time (with a Turing machine).

Definition 9.5.5 – Complexity class EXPTIME. The class EXPTIME is the class of decision problems that can be solved in exponential time (with a Turing machine).

We shall say come back on complexity classes Section 9.5.5

A common point to decision, optimization and counting problems is that their answer can be encoded in a small space compared to the size of the problem. In theory, it is not the case of reliability problems as a probability is a real number and the encoding of a real number may be infinite. However, floating point numbers provide sufficiently good approximations, and their

encoding is of constant size.

Optimization, counting and reliability problems are indeed at least as hard, and in general much harder, than their decision counterpart: If we know the best solution of a problem, we know *a fortiori* whether the problem has a solution. Similarly, if we know how to count the number of solutions to a problem or the probability to get a solution, we know if there is a solution to the problem.

There are problems for which the size of the solution may be exponentially larger than the size of the problem. BMF-MINIMALSOLUTIONS enters into this category: Boolean formulas with about $3^n/n$ prime implicants (which are in essence minimal solutions) have been exhibited (Chandra and Markowsky 1978) and it is often the case, in practice, that the number of minimal solution is extremely large, much larger in any case that the problem.

9.5.3 Reductions

There are two ways of assessing the complexity of a problem P: the direct and the indirect ways.

The direct way consists in designing an algorithm A that solves P and in calculating its complexity $\mathscr{O}(f_A)$. If, in addition, one is able to show that no other algorithm is more efficient than that one A, then one proves that the complexity of P itself is in $\mathscr{O}(f_A)$ (otherwise, one proves that its complexity is at most in $\mathscr{O}(f_A)$).

The indirect way consists in designing a pair (R, R^{-1}) of algorithms such that:

- R transforms any instance J of a problem Q for which we know the complexity, into an instance $I = R(J)$ of P.
- I has a solution if and only if J has one, moreover for any solution σ of I, we can use R^{-1} to retrieve a solution $\tau = R^{-1}(\sigma)$ of J.
- Both R and R' are efficient, i.e. have a low complexity compared to the complexity of Q.

The existence of such transformation (R, R^{-1}) shows that solving P is at least as hard as solving Q. Such a transformation R is called a *reduction*. We say that Q reduces to P

In the other way round, if there exists a reduction (R, R^{-1}) of any instance I of P into an instance J of Q such that I and J agree on their solutions, then solving P is a most as difficult as solving Q.

■ **Example 9.1 – Reduction of least common multiplier into greatest common divisor.** Assume that we want to calculate the least common multiplier $LCM(m,n)$ of two natural numbers m and n. Assume moreover that we know an algorithm to calculate their greatest common divisor $r = GCD(m,n)$. We know that $m = r \times p$ and $n = r \times q$ for some natural numbers p and q. p and q are prime one another. We have:

$$LCM(m,n) = r \times p \times q = \frac{r \times p \times r \times q}{r} = \frac{m \times n}{GCD(m,n)}$$

Therefore, LCM reduces to GCD (and vice-versa). ■

It remains to clarify what "efficient reduction" means. In general, it is considered that *log-space reductions*, i.e. reductions that use only $\log n$ fixed-size data in addition their read-only input and write-only output, are a good candidate (Arora and Barak 2009; Papadimitriou 1994). In practice, these reductions are of polynomial time complexity and make it possible to deal with problems which are themselves of polynomial time complexity.

9.5.4 Non-deterministic Turing machines

So far, we have considered implicitly that, when reading a symbol on the tape, a Turing machine had at most one possible entry in its instruction table for that symbol and the current state. That is, we have considered *deterministic* Turing machines, a more generally models of computation.

It is possible however to consider *non-deterministic* Turing machines (and more generally non-deterministic models of computation), i.e. Turing machines that may have several entries in

their instruction table for each pair of symbol and current state. In other words, Turing machines that make arbitrary choices. The notion of complexity is extended to non-deterministic Turing machine as follows.

Definition 9.5.6 – Complexity of non-deterministic Turing machines. The complexity (in time or in space) of a non-deterministic Turing machine is the complexity of its most efficient possible execution.

This is indeed only a theoretical model: any real computational device has necessarily some mechanism to make choices. Even if it makes them at random, this is still by using a mechanism, namely an algorithm or a physical device that produces numbers according to some predefined probability distributions. On the contrary, non-deterministic Turing machines make totally arbitrary choices, and it is assumed that they make always the "right" guess. To put things differently, it is like if they were helped by an oracle that tells them what to do, for free.

Going through the above paragraph, the reader smelt probably immediately the sting. If I am given by an oracle the solution to a problem for free, then it is easy for me to give you in turn this solution. This is true under one condition however: that the oracle is not a crook. In other words, once given a candidate solution, it remains to check that this is actually a solution.

As an illustration, consider again the satisfiability of Boolean formulas (Problem 9.3.2). On the one hand, given a formula f, finding a valuation σ of the variables of f such that $\sigma(f) = 1$ may require an exponential amount of time, as there are 2^n such valuations if f is built over n variables. On the other hand, if the oracle proposes me a candidate solution, i.e. a valuation σ, I can check in polynomial time whether $\sigma(f) = 1$ or not.

This *a priori* crazy idea idea of non-deterministic Turing machine turned out to be an extraordinary tool to characterize the complexity of problems.

Definition 9.5.7 – Complexity class NP. The class NP is the class of decision problems that can be solved in polynomial time with a non-deterministic Turing machine.

Problems in this class have a *polynomial certificate*: given a candidate solution, one can check in polynomial time whether this is actually a solution.

In 1971, Stephen Cook showed the following theorem (Cook 1971).

Theorem 9.4 – NP-completeness of SAT. SAT is NP-complete, i.e. both as easy and as hard as any other problem in the class NP.

The proof goes beyond the objective of this book, but it is easy to understand. I encourage the interested reader to look at Cook's paper or reference books on computational complexity theory (given at the end of this chapter).

Cook's result made SAT the other central problem (with the Halting problem) of computational complexity theory.

9.5.5 More Complexity classes

So far, we have seen the three complexity classes P, EXPTIME and NP. We shall now review a few more that are of interest for model-based systems engineering, and relate these classes one another.

Time complexity

We can obviously define NEXPTIME, the class of decision problems that can be solved in exponential time with a non-deterministic Turing machine. EXPTIME-hard problems are however already much too computationally costly to be of any practical significance. Adding non-determinism would only worsen the situation.

The following result holds, which a consequence of a more general theorem called the time hierarchy theorem, see e.g. (Papadimitriou 1994).

Theorem 9.5 – P versus NP versus EXPTIME.

$$\mathrm{P} \subseteq \mathrm{NP} \subsetneq \mathrm{EXPTIME}$$

This raises indeed the following question.

$$\mathrm{P} \stackrel{?}{=} \mathrm{NP} \tag{9.2}$$

This question is still open and considered as one of the most difficult questions of mathematics. A 1 million dollars price has even been proposed by the Clay foundation to the first person who will solve it.

Space complexity

The class PSPACE is defined as follows.

Definition 9.5.8 – Complexity class PSPACE. The class PSPACE is the class of decision problems that can be solved in polynomial space with a deterministic Turing machine.

It is indeed possible to define NPSPACE, the class of decision problems that can be solved in polynomial space by a non-deterministic Turing machine. However, the following result holds, which is a consequence of the Savitch's theorem (Savitch 1972).

Theorem 9.6 – PSPACE versus NPSPACE.

$$\mathrm{PSPACE} = \mathrm{NPSPACE}$$

Obviously, the following inclusion holds.

$$\mathrm{P} \subseteq \mathrm{PSPACE} \tag{9.3}$$

It is therefore of interest to compare PSPACE with EXPTIME.

We can assume, without a loss of generality, that we use a binary alphabet. If we use at most $q(n)$ cells to solve a problem, where q is some polynomial and n is the size of the instance, then the maximum number of configurations of the tape that can be encountered during the execution is $2^{q(n)}$. If we multiply this number by the number of states of the Turing machine, which is constant, we get the maximum number of steps the machine can do without looping (which is impossible as it is assumed to terminate). Therefore, the following inclusion holds.

$$\mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \tag{9.4}$$

This inclusion is believed to be strict, although no proof has been given so far. The reason for this belief is that EXPTIME allows to use an exponential space, which leads us to the class EXPSPACE.

Definition 9.5.9 – Complexity class EXPSPACE. The class EXPSPACE (respectively NEXPSPACE) is the class of decision problems that can be solved in exponential space with a deterministic (respectively a non-deterministic) Turing machine.

Of course, the following inclusion holds.

$$\mathrm{EXPTIME} \subseteq \mathrm{EXPSPACE} = \mathrm{NEXPSPACE} \tag{9.5}$$

Both the classes EXPTIME and EXPSPACE are however so large that they contain all problems with practical solutions and many without any practical solutions.

The polynomial hierarchy

From a practical point of view, PSPACE can be seen as the (extreme) limit of what is feasible. On the other hand, many problems encountered in systems engineering are NP-hard. Therefore, it is interesting to compare NP and PSPACE and to study which problems lies possibly in between these two classes.

The polynomial hierarchy (Stockmeyer 1976) provides a valuable insight on this question.

Let f be a Boolean formula in conjunctive normal form and let $\text{var}(f) = \{V_1, \ldots, V_n\}$.

SAT consists in determining whether there exists a valuation of V_i's that satisfies the formula f. The problem consists thus in determining the truth value of the following statement.

$$\exists V_1, \ldots, V_n \; f$$

Now consider the following problem.

Problem 9.1 – TAUTOLOGY. Let f be a Boolean formula built over a set of variables $\mathscr{V}$. Is f a tautology, i.e. that $\sigma(f) = 1$ for all assignments σ of the variables of $\mathscr{V}$?

TAUTOLOGY consists thus in determining the truth value of the following statement.

$$\forall V_1, \ldots, V_n \; f$$

Although this is rather counter-intuitive at a first glance, as $\forall V_1, \ldots, V_n \; f \equiv \neg\exists V_1, \ldots, V_n \; \neg f$, SAT and TAUTOLOGY are quite different problems. The difference stands in the notion of polynomial certificate: to prove that a given variable valuation σ verifies $\sigma(f) = 1$ is easy. To prove that all variable valuations σ verify $\sigma(f) = 1$ may be substantially harder.

This remarks apply to all decision problems in NP, leading to the definition of the complexity class coNP.

Definition 9.5.10 – coNP. *coNP* is the complexity class of decision problems whose complementary is in NP.

As the reader may expect, this raises another open question:

$$\text{NP} \stackrel{?}{=} \text{coNP}$$

The discovery of a NP-complete problem that belongs to coNP, or vice-versa, would show that NP = coNP. An example of a problem that is known to belong to both NP and coNP (but not known to be in P) is integer factorization: given positive integers p and q determine if p has a factor less than q and greater than 1. This problem plays a central role in cryptography.

NP is defined with by quantifying existentially the variables of a decision problem. coNP is defined by quantifying them universally. We can also alternate existential and universal quantifications. This leads to the definition of the polynomial hierarchy.

Definition 9.5.11 – Polynomial Hierarchy. The polynomial hierarchy is defined as follows.

- The class Σ_k is the class of decision problem that can be written with an alternation of k quantifiers, starting with an existential quantifier.
- The class Π_k is the class of decision problem that can be written with an alternation of k quantifiers, starting with a universal quantifier.
- The class Δ_k is defined as $\Sigma_k \cap \Pi_k$.

– Finally, the class PH is defined as the denumerable union of all these classes, e.g.

$$\mathrm{PH} \stackrel{def}{=} \bigcup_{i=0}^{\infty} \Sigma_i$$

It is not known whether the polynomial hierarchy collapses at some level. It is known that PH $\subseteq$ PSPACE but it is not known whether this inclusion is strict.

The following problem is representative of the polynomial hierarchy.

Problem 9.2 – QBF. QBF stands for quantified Boolean formulas. The problem can be stated as follows. Let f be a Boolean formula built over a set of variables $\mathscr{V} = \{V_1, \ldots, V_n\}$.

Is $\exists V_1 \forall V_2 \ldots f$ true?

QBF is the canonical complete problem for PSPACE.

9.5.6 A class that counts

The class #P

So far, all the classes we have seen are for decision problems. The class #P, introduced in 1979 by Leslie Valiant, is for counting and reliability problems (Valiant 1979a; Valiant 1979b), who got, among other awards and honors, the Turing Award.

Definition 9.5.12 – #P. The complexity class is #P the class of the counting problems associated with the decision problems in the set NP.

#SAT is indeed representative, actually complete, for this class.

Problem 9.3 – #SAT. Let f be a Boolean formula in conjunctive normal form.

How many valuations of the variables of $\mathrm{var}(()\, f)$ satisfy f?

#P can be used to characterize not only counting problems, but also reliability problems, as demonstrated by Valiant.

Problem 9.4 – RELIABILITY. Let f be a Boolean formula over a set of variables $\mathscr{V}$. Assume given a probability $p(V)$ for each variable $\mathscr{V}$. Assume moreover that variables of $\mathscr{V}$ represent statistically independent events, i.e. $p(V \wedge W) = p(V) \times p(W)$, for any $V, W \in \mathscr{V}$, $V \neq W$. It is then easy to verify that p is a probability measure on $\mathscr{V}$.

What is the probability of f?

Note that #SAT reduces easily to RELIABILITY: by assigning the probability 0.5 to each variable, calculating the probability of the formula and then multiplying the result by $2^{|\mathrm{var}(f)|}$, one gets the number of satisfying variable valuations of f.

#P has quite surprising properties. For instance, MONOTONESAT and 2SAT, the restrictions of SAT to respectively monotone formulas (formulas without negation), and clauses of length 2 are in P. This result is trivial for MONOTONESAT (it suffices to check the valuation where all variables are set to 1) and not difficult for 2SAT (see exercise 9.4). However, their counting counterparts are #P-hard (see exercise 9.5).

Toda's theorem

To conclude our introductory tour of computational complexity theory, it is worth to look at Toda's theorem (Toda 1991). This theorem is named after its author, Seinosuke Toda, who got for it the prestigious Gödel Prize. It has, in author's opinion, very important consequences for model-based systems engineering. It can be stated as follows.

Theorem 9.7 – Toda's theorem.

$$\text{PH} \subseteq \text{P}^{\#\text{P}}$$

The above inclusion means that the whole polynomial hierarchy is captured with polynomial time Turing machine with an oracle in #P. In daily English, this means that, if we can count "for free" the number of solutions of problems, then all of the problems of belonging to the polynomial hierarchy become easy, i.e. are in P.

Indeed, we are not able to count for free the number of solutions of problems encountered in systems engineering. Toda's result is thus to be taken the other way round: it shows that counting the number of solutions, or assessing the reliability, are very hard problems, probably among the hardest for which we can still "do something".

9.6 Putting Things Together

We can now review each of the key problems of reliability engineering and look at their complexity.

9.6.1 BMF-SATIFIABILITY

The following result holds.

Property 9.8 – Complexity of BMF-SATIFIABILITY. BMF-SATIFIABILITY is NP-complete.

Proof. The proof goes by reduction to SAT. Such reduction is achieved by noticing that saying that a variable v takes its value into a finite set of constants $c_1,\ldots c_k$ can be expressed using k Boolean variables $v = c_1,\ldots v = c_k$ and a Boolean formula saying that exactly one of these variables is true at a time. QED

BMF-SATIFIABILITY does not play any significant role in practical role in practical reliability engineering: as already pointed out, it is expected that any industrial system can fail. However, the above property asserts that even this very basic problem is already computationally intractable.

9.6.2 KMF-REACHABILITY

For the very same reasons, KMF-REACHABILITY does not play much role neither. Nevertheless, it is worth to study its computational complexity.

Consider a family of parametric models $M_{(n\in\mathbb{N})} : \langle V_n, P_n, T_n, \imath_n\rangle$ representing counters on n bits, i.e.

- V_n contains n $\{0,1\}$ variables $v_1, \ldots v_n$.
- P_n contains only one predicate characterizing the state $\vec{1}$ where all v_i's have the value 1.
- Transitions of T_n encodes increment by 1 of the counter.
- $\imath_n$ represents the state $\vec{0}$ in which all variables are set to 0.

The size of M_n, for a given n, is roughly proportional to n. However, the reachability graph of M_n contains 2^n states. Moreover, the execution leading to the state $\vec{1}$ is of length 2^n. Consequently, an algorithm that exhibits a path from $\vec{0}$ to the state $\vec{1}$ would be of exponential space complexity. Any simulation algorithm must actually at least keep track on the current execution in order to be able to backtrack and to avoid entering in infinite loops. Symbolic encoding of the sets of visited states, e.g. by means of binary decision diagrams as in SMV (McMillan 1993), can reduce the memory consumption, but not prevent the exponential blow-up. This means in turn that a polynomial algorithm to solve KMF-REACHABILITY on this family of models should be based on other principles than exploring in a way or another the reachability graph. Such an algorithm is not likely to be general enough to solve all KMF-REACHABILITY problems in polynomial time.

Although not a fully formal proof, this shows that for practical purposes at least, the following property holds.

Property 9.9 – Complexity of KMF-REACHABILITY. KMF-REACHABILITY is EXPSPACE-hard.

The above result shows the limits of reliability engineering and model-checking: various heuristics can be used to push these limits, but they cannot suppress them. It may sound weird for the reader familiar with the model-checking literature, e.g. (Baier and Katoen 2008; Berard et al. 2010; Clarke, Grumberg, Kroening, et al. 2018), that contains many much more optimistic complexity results. However, these results are obtained either by reducing dramatically the expressive power of the modeling language, or by starting from the reachability graph, or both. Of course, if the reachability graph is given, then KMF-REACHABILITY is in P. It suffices actually to use any graph exploration algorithm to look at reachable states one after the other (using flags to ensure that each transition is fired at most once) and to check for each visited state whether it verifies the property or not. As this check can be performed in polynomial time, the result follows.

As a corollary of the EXPSPACE hardness of KMF-REACHABILITY, KMF-DEADLOCK and KMF-LIVENESS are also EXPSPACE hard.

Recall that by Theorem 9.3, TMF-REACHABILITY is undecidable. Consequently, BMF-DEADLOCK and BMF-LIVENESS are also undecidable.

9.6.3 BMF-AVAILABILITY

Main Results

As said earlier, BMF-AVAILABILITY is the central problem of reliability engineering. We know, by Valiant's results that RELIABILITY is #P-hard. The following property is a direct consequence.

Property 9.10 – Complexity of BMF-AVAILABILITY. BMF-AVAILABILITY is #P-hard.

We could basically stop here and declare that probabilistic risk and safety assessment is hopeless. However, the situation requires a further analysis, for at least two reasons.

First, Valiant's result does not consider how probabilities are encoded into the machine. If finite precision floating point numbers are used, then precision issues and rounding errors come into the play, which makes the assessment of the complexity of the problem much more difficult. If some infinite precision encoding is used, then the result of the algorithm may be arbitrarily large, which makes also the assessment of the complexity of the problem much more difficult.

Second, in practice, it makes no sense to consider extremely low probabilities. For instance, a probability of 10^{-20} per hour is much lower than the one of an event that would have one chance to occur since the Big Bang, i.e ridiculously small. Moreover, probabilities for a variable v to take the value c, that are inputs of models, are estimated by experience feedback, physical modeling and simulation, or expert judgment. Such estimations are by no means precise. It makes no sense to require calculations to be very precise while input data are not.

In a word, approximated values of the availability would be sufficient provided they would be reasonably accurate. Alas, although results on the subject are sparse (Provan and Ball 1983), they are all negative: it is as hard to get warrantied approximations as to get exact values. The reason for that is relatively intuitive. Assume that there exists a polynomial algorithm to calculate an approximation of the availability. The complexity of this algorithm should depend on the accuracy of the approximation, i.e. it should be in $\mathcal{O}\left(n^{f(\varepsilon)}\right)$, where n is the size of the model, ε is the accuracy of the approximation (the maximum distance to the exact value), and f is a function of ε. But then, we could approach as close as we would like to the exact solution in polynomial time, possibly using a dichotomic search, which would contradict property 9.10.

A More Practical View

Looking for accurate approximations seems to let us jumping out of the frying pan into the fire. Actually not quite so, if we accept to drop warranties.

Let $\langle V, P, pr \rangle$ be a stochastic BMF model. Assume that each variable v of V takes its value in some finite degradation structure, i.e. there is a least value $\bot_v$ in $\mathrm{dom}(v)$ that represents the state in which the component represented by v is functioning correctly (is a good as new). We can reasonably assume that components of an industrial system spend most of their life time functioning correctly. In other word, $pr(v = \bot_v)$ must be close to 1 or at least must be much bigger than the probabilities for v to take another value of its domain.

Now consider the variable valuations of V. According to the above assumption, variable valuations that assign the value $\bot_v$ to most of the variables v have a much higher probability than those that assign $\bot_v$ to fewer variables v. The former states concentrate the probability. To put things differently, an industrial system spends most of the time with most of its components functioning correctly.

Consequently, it is sufficient to focus on high probability states, simply ignoring those with a low probability. Technically, one associates a weight $w(v = c)$ to each variable v and each constant $c \in \mathrm{dom}(c)$ such that $w(v = \bot_v) = 0$ and $w(v = c) > 0$ for all $c \neq \bot_v$. One can use the logarithm of $pr(v = c)$ to define $w(v = c)$. Then, one defines the weight of a variable valuation as the sum of the weights of individual valuations. It is then possible to consider only variable valuations whose weight is less than a given threshold τ. One can verify that there can be only a polynomial number of such variable valuations. We call such approximation τ-approximations.

This is the idea I introduced (for Boolean models) in my 2001 article (Rauzy 2001). A full development goes beyond the scope of this book. But we can state the following property that summarizes the approach.

Property 9.11 – Complexity of τ-approximation BMF-AVAILABILITY. τ-approximations of BMF-AVAILABILITY are computable in polynomial time.

This property is what makes probabilistic risk and safety analyses tractable in practice.

9.6.4 KMF-AVAILABILITY and TMF-AVAILABILITY

We face, with KMF-AVAILABILITY, the same problem we faced with KMF-REACHABILITY: the reachability graph of the model may be (and often is) exponentially larger than the model itself. Both in theory and in practice, this makes the calculation of exact solutions intractable in general. But what about specific cases and approximated solutions?

Markov Chains

As we pointed out in Section 9.3.5, discrete-time stochastic KMF/TMF models are actually implicitly defined discrete-time Markov chains. Consequently, if their state space is finite (which is always the case for KMF models), and not too big, they can be assessed by means of numerical algorithms to solve discrete-time Markov chains (Stewart 1994). Some very specific models with countable state spaces can solved as well, e.g. some of those stemmed from queueing theory (Trivedi 2001).

In practice, even for models with very large state space, the probability tends to concentrates into a relatively small number of states. To put it differently, for most of the states σ, the probability to be in the state σ at step n is close to 0. This leads to efficient approximation schemes. The idea is to develop partially the reachability graph in such way that all reached states have a probability bigger than some predefined threshold (Brameret, Rauzy, and J.-M. Roussel 2015). This heuristics gives good practical results.

The same ideas apply to continuous-time stochastic KMF/TMF models involving only exponentially distributed delays, i.e. to implicitly defined continuous-time Markov chains.

Stochastic Simulation

For continuous-time models involving delays that are not exponentially distributed, stochastic simulation is the only assessment tool at hand. It gives only approximated results.

The cost of a stochastic simulation is proportional to the cost of an individual execution times the number of executions performed. The cost of an individual execution depends mostly on the number of events fired during this execution. The cost of firing an event of a TMF model can be much higher than the one of a KMF model, as it involves the dynamic allocation and de-allocation of data structures.

The quality of approximations depends the probability of the property one is looking for. More exactly, the smaller this probability, the higher the number of executions required to get accurate results. As a rule of thumb, the number of executions should be at least 100 times bigger than the inverse of the probability of the property. This means that to assess a property whose probability is about 1.00×10^{-5} one needs to perform at least 1.00×10^{7} executions and assess a property whose probability is about 1.00×10^{-9} one needs to perform at least 1.00×10^{11} executions.

In some specific cases, there are ways to accelerated stochastic simulations, see e.g. (Zio 2013), but these are only heuristics that do not apply to general KMF/TMF models. From a practical point of view, these *ad-hoc* methods are hard and costly to implement.

With the increasing calculation power at hand, stochastic simulation becomes the Swiss knife of reliability engineering. Nevertheless, it is still computationally costly and its results are not necessarily very accurate. We leave here the realm of general theoretical results: whether stochastic simulation is suitable or not depends heavily of the problem at stake and the available computation resources.

9.6.5 BMF-MinimalSolutions

General Result

So far, we looked at problems whose solutions are of constant size. The solutions of BMF/KMF/TMF-Availability are decimal numbers, which potentially requires an arbitrary encoding length. In practice however, an infinite precision is meaningless, or to put it the reverse way, a bounded precision is sufficient.

On the contrary, the size of the solutions of BMF-MinimalSolutions is unbounded. More precisely, the number of minimal cutsets of the top event of a system of Boolean equations can be exponentially larger than this system.

The simplest example of such models is the formula n-out-of-$2n$. The size of encoding of this formula is linear in n, quadratic if we use only connectives $\wedge$ and $\vee$, see exercise 6.8. The number of minimal cutsets is $\binom{2n}{n}$. It can be shown that the following approximation holds.

$$\binom{2n}{n} \approx \frac{4^n}{\sqrt{\pi n}}$$

Hence the result.

Consequently, the following property holds.

Property 9.12 – Complexity of BMF-MinimalSolutions. BMF-MinimalSolutions is EXPSPACE-hard.

This property is again a negative result: *a priori*, the problem is intractable. In practice however, the situation is not as bad as the property indicates, as we shall see now.

Approximation Scheme

Minimal solutions are of interest for two reasons:

– They are used for qualitative analyses, i.e. to look at scenarios of failure. With that respect, the analyst is not interested in obtaining millions of minimal solutions. It would be anyway impossible to inspect them all. Consequently, he or she is interested only in the most relevant, i.e. eventually the most probable, ones.
– They are also used for quantitative analyses, i.e. for the calculation of probabilistic performance indicators. But in that case again, the analyst can focus on the most probable minimal solutions as the others have probably a tiny influence of the value of indicators.

In both cases, the analyst is thus only interested in the most probable minimal solutions, i.e. enough minimal solutions to capture the most part of the probability of the property of interest, but not too many to keep their treatment tractable. The key idea is thus to keep only minimal solutions whose probability is above a certain threshold.

Of course, this works only if a small proportion of minimal solutions concentrates the probability. This is not always the case. For instance, in the above example, n-out-of-$2n$, if all basic events have the same probability, then the probability of each minimal cutsets is close to $\frac{\sqrt{\pi n}}{4^n}$, i.e. tends quickly to 0 as n increases.

Fortunately, in practice, such situations almost never happen. The components of an industrial system spend most of their time in a functioning state. In other words, if the state of a component is represented by a variable v taking its value in a given finite degradation structure $\mathrm{dom}(v)$ with a least element $\bot_v$, then the probability $p_v(\bot_v)$ that the variable v takes the value $\bot_v$ is normally much bigger than the probability $p_v(c)$ of any other constant c of $\mathrm{dom}(v)$. Consequently, the probabilities of variable valuations decrease quickly as the number of variables not assigned to their least value increases. This gives raises to a general polynomial approximation scheme.

Let $\langle V, P\rangle$ be a stochastic BMF model and let w be a function that associates a weight $w_v c$ to each variable v of V and each value c in $\mathrm{dom}(v)$ such that:

i) $w_v \bot_v = 0$, for the least element $\bot_v$ of $\mathrm{dom}(v)$;
ii) $w_v c > 0$, for all elements $c \neq \bot_v$ of $\mathrm{dom}(v)$.

$w_v(c)$ can be defined for instance from the probability $p_v(c)$ as follows:

$$w_v(c) \stackrel{def}{=} \begin{cases} 0 & \text{if } c = \bot_v \\ -log(p_v(c)) & \text{otherwise} \end{cases}$$

The reason to require the weight of the least element to be zero is to stay independent of the order in which variables are considered by algorithms.

The weight $w(\sigma)$ of a valuation σ of the variables of V is defined as the sum of the weights of its individual valuations.

$$w(\sigma) \stackrel{def}{=} \sum_{v \in V} w_v(\sigma(v))$$

Given a cutoff τ, i.e. a maximum weight, we denote by $\mathrm{dom}(V)|\tau$ the set of variable valuation whose weight is lower than τ:

$$\mathrm{dom}(V)|\tau \stackrel{def}{=} \{\sigma \in \mathrm{dom}(V); w(\sigma) \leq \tau\}$$

The idea, which extends what I proposed for the Boolean case in my 2001 article (Rauzy 2001), if to consider $\mathrm{dom}(V)|\tau$ as the care set, i.e. to ignore all variable valuations not in $\mathrm{dom}(V)|\tau$ (Rauzy and Yang 2019a; Rauzy and Yang 2019b).

The following property holds.

Property 9.13 – Complexity of BMF-MinimalSolutions with Cutoff. Let $\langle V, P, w\rangle$ be a weighted BMF model, where the weight w verifies the above conditions i) and ii), let τ a cutoff value. We shall denote:

– κ the minimum value over the variables v of V and the constants $c \neq \bot_v$ in $\mathrm{dom}(v)$ of the

$w_v(c)$'s,
- k the smallest integer such that $\kappa \times k \geq \tau$,
- d the maximum over the variables v of V, of the size of $\mathrm{dom}(v)$,
- finally, n the number of variables of V.

Then, the number of variables valuations in $\mathrm{dom}(V)|\tau$ is in $\mathscr{O}\left(n^k \times d^k\right)$, i.e. polynomial in the size of the model.

In other words, the introduction of cutoffs makes it possible to design polynomial algorithms to extract minimal solutions and to assess probabilistic performance indicators from these minimal solutions. Indeed, these approximations cannot be warranted, as illustrated by our n-out-of-$2n$ example. In practice however, they prove to be good enough is most of the cases.

9.6.6 KMF-MinimalTraces and TMF-MinimalTraces

For the very same reason as BMF-MinimalTraces is EXPSPACE hard, KMF-MinimalTraces and TMF-MinimalTraces are also EXPSPACE hard.

Similar polynomial approximation schemes can also be applied. The idea is to associate a weight with each transition and to define the weight of an execution as the sum of the weights of its individual transitions. Then, one can limit the exploration to executions whose weight is below a certain cutoff. As weights are non negative numbers, they can be seen as distances. Extraction algorithms thus explore the reachability graph until a certain depth (a certain distance from the initial state).

In the case of discrete-time models, the translation of probabilities into weights is straightforward. In the case of continuous-time models, it is by no means obvious. The heuristics proposed for Markov chains in my 2015 article (Brameret, Rauzy, and J.-M. Roussel 2015) is hardly extensible to non Markovian models. At the time I am writing these lines, the subject remains to explore.

9.6.7 Wrap-Up and Concluding Remarks

Throughout this chapter, I tried to present the state of the art knowledge on computational complexity issues raised by the assessment of reliability engineering models. For the system engineer, there are a number of important points to remember:

1. There are three nested categories of modeling frameworks for reliability engineering: combinatorial models, state automata and process algebras.
2. For each of these categories, it is possible to formalize two key problems related to the assessment of models: the calculation of the probability that a certain situation occurs, and the extraction of minimal scenarios leading to this situation.
3. The worst case complexity of these problems is extremely high. In theory, they are intractable, if not undecidable. Fortunately, heuristics can be applied to push the limits of what is feasible to do, by calculating approximated and partial results. Nevertheless, by definition, heuristics do not always work. Limits are there anyway that frame the whole modeling process.

In practice, this means that reliability engineering models result necessarily of a trade-off between the accuracy of the description of the system under study and the ability to perform calculations on this description. In other words, the analyst faces the fundamental epistemic and aleatory uncertainties of risk assessment with a bounded calculation capacity, and this bounded capacity over-determines both the design of models and the decisions that can be made from models. With that respect, he or she is like Simon's economical agent who must make decisions with a bounded rationality (H. Simon 1957).

The problem at stake can be thus formulated as follows: given my limited modeling and calculation capacities, given all the uncertainties of the modeling process, where should I concentrate my efforts so to ensure a reasonably correct and reasonably robust decision process?

9.7 Further Readings

Reference books on computational complexity theory:

- The old, famous and still actual book on NP-completness by Garey and Johnson (Garey and Johnson 1979).
- The very complete book by Papadimitriou (Papadimitriou 1994).
- The more recent book by Arora and Barak (Arora and Barak 2009).

Reference books on model-checking:

- The introductory book by Clarke, Grumberg and Peled (Clarke, Grumberg, and Peled 2000).
- The complete book by Baier and Katoen (Baier and Katoen 2008).

9.8 Exercises and Problems

Problem 9.1 – Computation Times. The objective of this exercise is to make concrete the computational complexities we have seen in this chapter. It is recommended to design a program to solve this exercise. Assume thus that your computer is able to execute 1000000 instructions per second.

Question 1. The following table gives the complexities of 4 algorithms we want to look at.

Algorithm	A1	A2	A3	A4
Complexity	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$

Calculate the execution times of these algorithms on problems of size $n = 100$, 1000, 10000, 100000, 1000000.

Question 2. With the same hypothesis, calculate the execution times of the algorithm A5 that is in $\mathcal{O}(2^n)$, for problems of size $n = 10, 20, \ldots 100$.

Question 3. Let us consider now an algorithm A6 that must explore all possible combinations of k elements out of n. With the same hypothesis, calculate the execution times of the algorithm A5 for $n = 100$, 1000, 10000 and k ranging from 1 to 6.

■

Exercise 9.2 – Conjunctive normal form. The objective of this exercise is to study the transformation of a Boolean formula f into an equivalent Boolean formula g in conjunctive normal form.

Question 1. Applying Boolean algebra identities (see Appendix B.3), transformation the following formula f into an equivalent formula g in conjunctive normal form.

$$f \quad = \quad (\neg(A \vee B) \vee (C \wedge D)) \wedge (\neg A \vee B)$$

Question 2. The problem of the above transformation is that the application of distributivity and de Morgan's law may transform the original formula into an exponentially larger formula. This problem can be avoided by introducing new variables. The idea is that we can replace any sub-formula h by a fresh variable W and to and the resulting formula with the formula $W \Leftrightarrow h$ put in conjunctive normal form.
Formally, let f be a Boolean formula, h be a subformula of f and W be a variable, $W \notin \text{var}(f)$, then the following equivalence holds.

$$f \quad \equiv \quad f[h \leftarrow W] \wedge (W \vee h) \wedge (\neg W \vee \neg h) \tag{9.6}$$

Using equivalence 9.6, transform the formula f of the previous question into an formula g in conjunctive normal form which is satisfiable if and only if f is satisfiable. f and g are said equisatisfiable.

■

Exercise 9.3 – Reduction SAT to 3SAT. The problem 3SAT is similar to the problem SAT, except that all clauses must be of length 3. The objective of this exercise is to show that SAT can be reduced to 3SAT, i.e. for any SAT instance f there is a 3SAT instance g that is satisfiable if and only if f is satisfiable. Moreover, we shall see that this reduction is *faithful*, i.e. it is easy to pass from solutions of f to solutions of g, and vice-versa. Namely, the reduction involves the use of fresh variables. But if we do not look at the values of these variables, the solutions of f and g are exactly the same.

Question 1. Consider first a clause of length 1 or 2. How this clause can be transformed into an equivalent set (conjunction) of clauses of length 3?

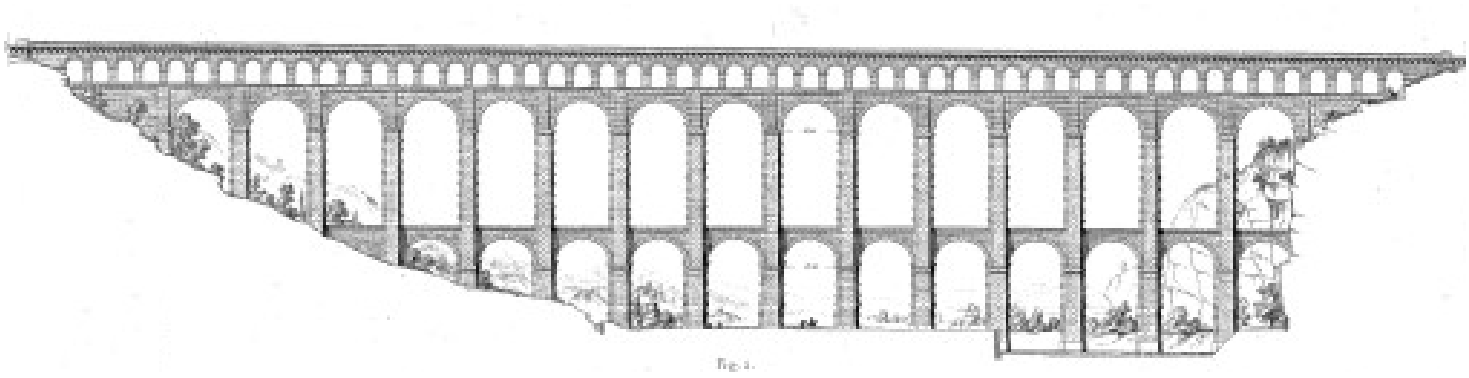
Question 2. Consider now a clause of length $k > 3$. How this clause can be transformed into an equivalent set (conjunction) of clauses of lengths strictly inferior to k?

■

Exercise 9.4 – Complexity of 2SAT. 2SAT is... ■

Exercise 9.5 – Complexity of #MONOTONE-SAT. #MONOTONE-SAT... ■

III Bibliography and Appendices



Bibliography

Abadi, Mauricio and Luca Cardelli (1998). *A Theory of Objects*. New-York, USA: Springer-Verlag. ISBN: 978-0387947754 (cited on pages 15, 18, 82, 83, 91).

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 pages 64, 76).

Abrial, Jean Raymon (Nov. 2010). *Modeling in Event-B: System and Software Engineering*. Cambridge, England: Cambridge University Press. ISBN: 978-0521895569 (cited on page 208).

Aho, Alfred V. et al. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0321486811 (cited on page 63).

Ajmone-Marsan, Marco et al. (1994). *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. New York, NY, USA: John Wiley and Sons. ISBN: 978-0471930594 (cited on pages 123, 124, 208).

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 pages 2, 124, 173).

Apostolakis, George (Dec. 1990). "The concept of probability in safety assessment of technological systems". In: *Science* 250.4986, pages 1359–1364. DOI: 10.1126/science.2255906 (cited on page 96).

Arnold, André (1994). *Finite Transition Systems*. C.A.R Hoare. Upper Saddle River, New Jersey, USA: Prentice Hall. ISBN: 978-0130929907 (cited on page 190).

Arnold, André et al. (2000). "The AltaRica Formalism for Describing Concurrent Systems". In: *Fundamenta Informaticae* 34, pages 109–124 (cited on page 209).

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 pages 243, 263, 274).

Guidelines for Development of Civil Aircraft and Systems (Dec. 2010). Warrendale, Pennsylvania, USA (cited on pages 97, 223).

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 page 223).

Baier, Christel and Joost-Pieter Katoen (June 2008). *Principles of Model-Checking*. Cambridge, MA, USA: MIT Press. ISBN: 978-0262026499 (cited on pages 195, 222, 269, 274).

Batteux, Michel, Tatiana Prosvirnova, and Antoine Rauzy (Sept. 2017a). "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 pages 177, 186, 193, 194, 209, 248).

— (2017b). *AltaRica 3.0: language specification*. AltaRica Association (cited on pages 178, 184).

— (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 4, 15, 71).

— (2019a). "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 4, 13, 177, 209).

— (2019b). "Model Synchronization: A Formal Framework for the Management of Heterogeneous Models". In: *Model-Based Safety and Assessment*. Edited by Yiannis Papadopoulos et al. Volume 11842. LCNS. Thessaloniki, Greece: Springer, pages 157–172. ISBN: 978-3-030-32871-9 (cited on page 2).

— (2021). "Abstract Executions of Stochastic Discrete Event Systems". In: *International Journal of Critical Computer-Based Systems* (cited on page 195).

Berard, Béatrice et al. (Dec. 2010). *Systems and Software Verification: Model-Checking Techniques and Tools*. Berlin, Germany: Springer-Verlag Berlin, Heidelberg GmbH, and Co. K. ISBN: ISBN-10: 3642074782, ISBN-13: 978-3642074783 (cited on page 269).

Berg, Ulf (Apr. 1994). *RISK SPECTRUM, Theory Manual*. RELCON Teknik AB (cited on pages 156, 171, 172).

Berthoz, Alain (2012). *Simplexity: Simplifying Principles for a Complex World*. New Haven, CT, USA: Yale University Press. ISBN: 978-0300169348 (cited on pages 14, 18).

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 170).

Blanchard, Benjamin S. and Wolter J. Fabrycky (2008). *Systems Engineering and Analysis*. Upper Saddle River, NJ 07456, USA: Pearson. ISBN: 978-0137148431 (cited on pages 1, 36).

Bliudze, Simon and Joseph Sifakis (2008). "The Algebra of Connectors - Structuring Interaction in BIP". In: *IEEE Trans. Computers* 57.10, pages 1315–1330 (cited on page 208).

Boole, George (1854). *An Investigation of the Laws of Thought*. Reissued. New York, NJ, USA: Dover. ISBN: 9780486600284 (cited on page 246).

Borges, Jorge Luis (1975). "A Universal History of Infamy". In: translated by Norman Thomas de Giovanni. London, England: Penguin Books, pages 201–236. ISBN: 0-14-003959-7 (cited on page 12).

Bouissou, Marc and Jean-Louis Bon (2003). "A new formalism that combines advantages of Fault-Trees and Markov models: Boolean logic-Driven Markov Processes". In: *Reliability Engineering and System Safety* 82.2, pages 149–163. DOI: 10.1016/S0951-8320(03)00143-1 (cited on page 208).

Bouissou, Marc, Henri Bouhadana, 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 209).

Bouissou, Marc and Jean-Christophe Houdebine (Mar. 2002). “Inconsistency Detection in KB3 Models”. In: *Proceedings of European Safety and Reliability conference, ESREL'2002*. Lyon, France: ISdF, pages 754–759 (cited on page 209).

Bouissou, Marc, Sibylle Humbert, et al. (Mar. 2002). “KB3 tool: feedback on knowledge bases”. In: *Proceedings of European Safety and Reliability conference, ESREL'2002*. Lyon, France: ISdF, pages 114–119 (cited on page 209).

Box, George E. P. and Mervin E. Muller (1958). “A Note on the Generation of Random Normal Deviates”. In: *The Annals of Mathematical Statistics* 29.1, pages 610–611. DOI: 10.1214/aoms/1177706645 (cited on page 322).

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 11).

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 page 156).

Brameret, Pierre-Antoine, Antoine Rauzy, and Jean-Marc Roussel (July 2015). “Automated generation of partial Markov chain from high level descriptions”. In: *Reliability Engineering and System Safety* 139, pages 179–187. DOI: 10.1016/j.ress.2015.02.009 (cited on pages 204, 270, 273).

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 156).

— (Sept. 1992). “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams”. In: *ACM Computing Surveys* 24, pages 293–318 (cited on page 156).

Carnap, Rudolf (2003). *Introduction to Symbolic Logic and Its Applications*. New York, USA: Dover Publications Inc. ISBN: ISBN 978-0486604534 (cited on page 302).

Cassandras, Christos G. and Stéphane Lafortune (2008). *Introduction to Discrete Event Systems*. New-York, NY, USA: Springer. ISBN: 978-0-387-33332-8 (cited on pages 205, 208).

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 263).

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 171).

Chomsky, Noam (1956). “Three models for the description of language”. In: *IRE Transactions on Information Theory* 2, pages 113–124. DOI: 10.1109/TIT.1956.1056813 (cited on page 52).

— (1957). *Syntactic Structures*. The Hague, The Nederland: Mouton. ISBN: 978-9027933850 (cited on pages 13, 63).

Clarke, Edmund M., Orna Grumberg, Daniel Kroening, et al. (Feb. 2018). *Model Checking (second edition)*. Cambridge, MA, USA: MIT Press. ISBN: 978-0262038836 (cited on pages 251, 269).

Clarke, Edmund M., Orna Grumberg, and Doron A. Peled (Feb. 2000). *Model Checking*. Cambridge, MA, USA: MIT Press. ISBN: 978-0262032704 (cited on pages 195, 274).

Cockburn, Alistair (2000). *Writing Effective Use Cases*. Boston, MA 02116, USA: Addison Wesley. ISBN: 978-0201702255 (cited on page 23).

Colbourn, Charlie J. (1987). *The combinatorics of network reliability*. New York: Oxford University Press. ISBN: ISBN 0-19-504920-9 (cited on page 193).

Colmerauer, Alain and Philippe Roussel (1996). “The birth of Prolog”. In: *History of Programming Languages II*. Edited by Thomas J. Bergin and Richard G. Gibson. New York, NY, USA: ACM Press/Addison-Wesley, pages 331–367. ISBN: ISBN:0-201-89502-1 (cited on page 14).

Cook, Stephen (1971). “The complexity of theorem proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, pages 151–158 (cited on page 264).

Cori, René and Daniel Lascar (2003a). *Logique mathématique, tome 1 : Calcul propositionnel, algèbre de Boole, calcul des prédicats*. Malakoff, France: Dunod (cited on page 302).

— (2003b). *Logique mathématique, tome 2 : Fonctions récursives, théorème de Gödel, théorie des ensembles, théorie des modèles*. Malakoff, France: Dunod (cited on page 302).

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 152).

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 156).

Crozier, Michel and Erhard Friedberg (1980). *Actors and Systems: The Politics of Collective Action*. Published first in French in 1977. Chicago, Illinois, USA: University of Chicago Press. ISBN: 978-0226121833 (cited on pages 18, 23).

Cruse, Alan (2011). *Meaning in Language: An Introduction to Semantics and Pragmatics*. Oxford Textbooks in Linguistics. Oxford, U.K.: Oxford University Press. ISBN: 978-0199559466 (cited on pages 13, 48, 63).

David, Pierre, Vincent Idasiak, and Frederic Kratz (2010). “Reliability study of complex physical systems using SysML”. In: *Reliability Engineering & System Safety* 95.4, pages 431–450 (cited on page 208).

Diekert, Volker and Grzegorz Rozenberg (1995). *The Book of Traces*. Inc. River Edge, NJ, USA: World Scientific Publishing. ISBN: 978-9810220587 (cited on page 256).

Dori, Dov (2002). *Object-Process Methodology — A Holistic Systems Paradigm*. Berlin, Heidelberg, New York: Springer Verlag. ISBN: 3-540-65471-2 (cited on page 13).

— (2016). *Model-Based Systems Engineering with OPM and SysML*. Berlin, Heidelberg, New York: Springer Verlag. ISBN: 978-1-4939-3294-8 (cited on page 36).

Dowling, William F. and Jean H. Gallier (1984). “Linear-time algorithms for testing the satisfiability of propositional Horn formulae”. In: *Journal of Logic Programming* 1.3, pages 267–284. DOI: 10.1016/0743-1066(84)90014-1 (cited on page 59).

Dugan, Joanne Bechta, Salvatore J. Bavuso, and Marks A. Boyd (Sept. 1992). “Dynamic fault-tree models for fault-tolerant computer systems”. In: *IEEE Transactions on Reliability* 41.3, pages 363–377. DOI: 10.1109/24.159800 (cited on page 208).

Dutuit, Yves and Antoine Rauzy (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).

Eppinger, Steven D. and Tyson R. Browning (June 2012). *Design Structure Matrix Methods and Applications*. Engineering Systems. Boston, MA 02116, USA: MIT Press. ISBN: 978-0262017527 (cited on page 34).

Esperza, Javier (1998). “Decidability and Complexity of Petri Nets Problems – An introduction”. In: *Lectures on Petri Nets I: Basic Models*. Edited by W. Reisig and G. Rozenberg. Volume 1491. LNCS. Springer, pages 374–428. ISBN: ISBN 3-540-65306-6 (cited on pages 123, 260).

Feiler, Peter H. and David P. Gluch (2015). *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Upper Saddle River, New Jersey, USA: Addison Wesley. ISBN: 978-0134208893 (cited on page 36).

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 pages 13, 17, 18, 25, 36, 71, 75, 208).

Fritzson, Peter (2015). *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Hoboken, NJ 07030-5774, USA: Wiley-IEEE Press. ISBN: 978-1118859124 (cited on pages 13, 208, 218, 248).

Fuhrmann, Hauke A. L. (2011). *On the Pragmatics of Graphical Modeling*. Kiel Computer Science Series. Norderstedt, Germany: Book on Demand. ISBN: 978-3844800845 (cited on pages 17, 63).

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 171).

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 page 156).

Gamma, Erich et al. (Oct. 1994). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Boston, MA 02116, USA: Addison-Wesley. ISBN: 978-0201633610 (cited on pages 15, 18, 30, 84, 218).

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 pages 18, 243, 274).

Girard, Jean-Yves, Yves Lafont, and Paul Taylor (1989). *Proof and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge, United Kingdom: Cambridge University Press. ISBN: 978-0521371810 (cited on page 12).

Gochet, Paul and Pascal Gribomont (1990). *Logique : Volume 1, Méthodes pour l'informatique fondamentale*. Paris, France: Hermès science publications. ISBN: ISBN 978-2-86601-249-6 (cited on page 302).

— (1994). *Logique : Volume 2, Méthodes formelles pour l'étude des programmes*. Paris, France: Hermès science publications. ISBN: ISBN 978-2-86601-395-0 (cited on page 302).

Gödel, Kurt (1951). "Some basic theorems on the foundations of mathematics and their implications". In: edited by Solomon Feferman. Volume III. Kurt Gödel Collected works. Oxford, England, United Kingdom: Oxford University Press, pages 304–323. ISBN: 978-0195147223 (cited on page 259).

Groff, James R., Paul N. Weinberg, and Andrew J. Oppel (2009). *SQL The Complete Reference (3rd Edition)*. New York, New Jersey, USA: McGraw-Hill Osborne. ISBN: 978-0071592550 (cited on page 14).

Güdemann, Matthias and Frank Ortmeier (2010). "A Framework for Qualitative and Quantitative Model-Based Safety Analysis". In: *Proceedings of the IEEE* 12^{th} *High Assurance System Engineering Symposium (HASE 2010)*. San Jose, CA, USA: IEEE, pages 132–141. ISBN: ISBN 978-1-4244-9091-2. DOI: 10.1109/HASE.2010.24 (cited on page 209).

Halbwachs, Nicolas (1993). *Synchronous Programming of Reactive Systems*. MIT Electrical Engineering and Computer Science. Dordrecht, The Netherlands: Kluwer Academic Publisher. ISBN: 978-1441951335 (cited on page 208).

Harel, David (June 1987). "Statecharts: a visual approach to complex systems". In: *Science of Computer Programming* 8.3, pages 231–274. DOI: 10.1016/0167-6423(87)90035-9 (cited on pages 13, 31, 208).

Harel, David and Michal Politi (1998). *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. New York, NY, USA: McGraw-Hill. ISBN: 0070262055 (cited on pages 13, 208).

Hatchuel, Armand and Benoit Weill (2009). “C-K design theory: an advanced formulation”. In: *Research in Engineering Design* 19.4, pages 181–192 (cited on pages 15, 18, 84).

Henzinger, Thomas A. (1996). “The Theory of Hybrid Automata”. In: *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS)* 16, pages 278–292. DOI: 10.1109/LICS.1996.561342 (cited on page 229).

Holt, Jon and Simon Perry (2013). *SysML for Systems Engineering: A Model-Based Approach*. Stevenage Herts, United Kingdom: Institution of Engineering and Technology. ISBN: 978-1849196512 (cited on page 36).

Holt, Jon, Simon Perry, and Mike Brownsword (2016). *Foundations for Model-based Systems Engineering: From patterns to models (Computing and Networks)*. Stevenage Herts, United Kingdom: Institution of Engineering and Technology. ISBN: 978-1785610509 (cited on page 48).

Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman (2006). *Introduction to Automata Theory, Languages, and Computation (third edition)*. New York City, New York, USA: Pearson Education International. ISBN: 978-0321455369 (cited on pages 63, 205, 248).

International electrotechnical vocabulary - Part 192: Dependability (2015). Standard. Geneva, Switzerland: International Electrotechnical Commission (cited on page 101).

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 96, 97, 99, 103).

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 101).

IEEE (2005). *1220-2005 - IEEE Standard for Application and Management of the Systems Engineering Process*. Standard. Piscataway, New Jersey, USA: IEEE (cited on page 35).

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 pages 72, 97).

ISO26262 Functional Safety - Road Vehicle (2012). Standard. Geneva, Switzerland: International Standardization Organization. URL: http://www.iso.org/iso/home.html (cited on page 97).

Jensen, Kurt (2014). *Coloured Petri Nets*. Berlin and Heidelberg, Germany: Springer-Verlag. ISBN: 978-3642425813 (cited on pages 122, 248).

Kaplan, Edward Lynn and Paul Meier (1958). “Nonparametric estimation from incomplete observations”. In: *Journal of American Statistics Association* 53.282, pages 457–481. DOI: 10.2307/2281868 (cited on pages 106, 315).

Kehren, Christophe et al. (2004). “Architecture Patterns for Safe Design”. In: *Electronic proceedings of 1st Complex and Safe Systems Engineering Conference (CS2E 2004)*. Arcachon, France: AAAF (cited on page 218).

Klee, Harold and Randal Allen (Feb. 2011). *Simulation of Dynamic Systems with MATLAB and Simulink*. Boca Raton, FL 33431, USA: CRC Press. ISBN: 978-1439836736 (cited on pages 208, 248).

Kleene, Stephen Cole (1967). *Mathematical Logic*. Dover Books on Mathematics. reprint 2002. St. Mineola, NY 11501, USA: Dover Publications Inc. ISBN: 978-086425337 (cited on page 75).

Kloul, Leïla, Tatiana Prosvirnova, and Antoine Rauzy (Dec. 2013). “Modeling systems with mobile components: a comparison between AltaRica and PEPA nets”. In: *Journal of Risk and Reliability* 227.6, pages 599–613. DOI: 10.1177/1748006X13490497 (cited on page 248).

Kloul, Leïla and Antoine Rauzy (Nov. 2017). “Production trees: A new modeling methodology for production availability analyses”. In: *Reliability Engineering and System Safety* 167, pages 561–571. DOI: 10.1016/j.ress.2017.06.017 (cited on page 218).

Kolmogorov, Andrei N. (1933). *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Berlin, Germany: Springer. ISBN: 978-3540061106 (cited on page 305).

Kreyszig, Erwin (2011). *Advanced Engineering Mathematics*. Hoboken, NJ, USA: John Wiley and Sons. ISBN: 978-3540061106 (cited on page 119).

Kripke, Saul (1963). "Semantical Considerations on Modal Logic". In: *Acta Philosophica Fennica* 16, pages 83–94. DOI: 10.1002/malq.19630090502 (cited on page 247).

Krob, Daniel (Jan. 2017). *CESAM: CESAMES Systems Architecting Method: A Pocket Guide*. CESAMES. http://www.cesames.net (cited on pages 18, 19, 35).

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 124, 168, 173).

Kwiatkowska, Marta, Gethin Norman, and David Parker (2011). "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proceedings 23rd International Conference on Computer Aided Verification (CAV'11)*. Volume 6806. LNCS. New York, NY, USA: Springer, pages 585–591. ISBN: 978-3642221095 (cited on pages 208, 209).

Lakoff, George (Apr. 1990). *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*. Chicago, Illinois, U.S.A: The University of Chicago Press. ISBN: 978-0226468044 (cited on pages 3, 15, 18, 84).

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 170).

Lane, Saunders Mac (1998). *Categories for the Working Mathematician*. Graduate Texts in Mathematics. New York, USA: Springer-Verlag. ISBN: 978-0387984032 (cited on page 302).

Larsen, Kim G., Paul Pettersson, and Wang Yi (Oct. 1997). "UPPAAL in a Nutshell". In: *International Journal on Software Tools for Technology Transfer* 1.1–2, pages 134–152. DOI: 10.1007/s100090050010 (cited on page 195).

Lazar, Guillaume and Robin Penea (Dec. 2016). *Mastering Qt 5*. Birmingham, United Kingdom: Packt Publishing Limited. ISBN: 978-1786467126 (cited on page 15).

Leveson, Nancy G. (Feb. 2012). *Engineering a Safer World - Systems Thinking Applied to Safety*. Cambridge, MA 02142-1315, USA: MIT Press. ISBN: 978-0262016629 (cited on pages 100, 234).

Lisnianski, Anatoly and Gregory Levitin (2003). *Multi-State System Reliability*. Quality, Reliability and Engineering Statistics. London, England: World Scientific. ISBN: 981-238-306-9 (cited on page 208).

Loeliger, Jon and Matthew Mccullough (Aug. 2012). *Version Control with Git 2e*. Sebastopol, CA 95472, USA: O'Reilly Media, Inc. ISBN: 978-1449316389 (cited on page 22).

Lynch, Nancy A. and Mark R. Tuttle (1989). "An introduction to input/output automata". In: *CWI Quarterly* 2, pages 219–246 (cited on page 248).

Madre, Jean-Christophe et al. (1994). "Application of a New Logically Complete ATMS to Digraph and Network-Connectivity Analysis". In: *Proceedings of the Annual Reliability and Maintainability Symposium, ARMS'94*. Annaheim, California: IEEE, pages 118–123. DOI: 10.1109/RAMS.1994.291093 (cited on pages 193, 194).

Maier, Mark W. (July 1998). "Architecting principles for systems-of-systems". In: *Systems Engineering* 1.4, pages 267–284. DOI: 10.1002/j.2334-5837.1996.tb02054.x (cited on pages 207, 248).

— (2009). *The Art of Systems Architecting*. Boca Raton, FL 33431, USA: CRC Press. ISBN: 978-1420079135 (cited on pages 30, 36, 48, 218).

Matloff, Norman and Peter Jay Salzman (2008). *The Art of Debugging with GDB, DDD, and Eclipse*. San Fransisco, CA, USA: No Starch Press. ISBN: 978-1593271749 (cited on page 195).

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 197, 322).

Mauborgne, Pierre et al. (Aug. 2016). “Operational and System Hazard Analysis in a Safe Systems Requirement Engineering Process – Application to automotive industry”. In: *Safety Science* 87, pages 256–268 (cited on page 208).

McMillan, Kenneth L. (1993). *Symbolic Model Checking*. ISBN 0-7923-9380-5. New York, NY, USA: Kluwer Academic Publisher (cited on page 268).

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 pages 15, 18, 27, 91).

— (1990). *Introduction to the Theory of Programming Languages*. Prentice Hall International Series in Computing Science. Upper Saddle River, Jersey, USA: Prentice Hall. ISBN: 978-0134985107 (cited on page 63).

— (2009). *Touch of Class: Learning to Program Well With Objects and Contracts*. Berlin and Heidelberg, Germany: Springer-Verlag. ISBN: 978-3540921448 (cited on page 27).

Mhenni, Faida et al. (2016). “Flight Control System Modeling with SysML to Support Validation, Qualification and Certification”. In: *IFAC-PapersOnLine* 49.3, pages 453–458 (cited on page 208).

Miller, George Armitage (1970). *The Psychology of Communication*. Pelican books. New York, USA: Penguin. ISBN: 978-0140211412 (cited on page 34).

Milner, Robin (1989). *Communication and Concurrency*. Prentice-Hall international series in computer science. Upper Saddle River, New Jersey, USA: Prentice Hall. ISBN: 9780131150072 (cited on page 195).

— (1999). *Communicating and Mobile Systems: The pi-calculus*. Cambridge, CB2 8BS, United Kingdom: Cambridge University Press. ISBN: 978-0521658690 (cited on page 248).

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 page 156).

Modelica Association (Oct. 2019). *Functional Mock-up Interface for Model Exchange and Co-Simulation*. Technical report 2.0.1. Modelica Association (cited on page 208).

Mosleh, Ali, Dale M. Rasmuson, and F. M. Marshall (1998). *Guidelines on Modeling Common-Cause Failures in PRA*. Technical report NUREG/CR-5485. U.S. Nuclear Regulatory Commission (cited on page 192).

Murata, Tadao (Apr. 1989). “Petri Nets: Properties, Analysis and Applications”. In: *Proceedings of the IEEE* 77.4, pages 541–580 (cited on pages 13, 124, 248).

Natvig, Bent (2010). *Multistate Systems Reliability Theory with Applications*. Hoboken, NJ, USA: Wiley. ISBN: 978-0-470-69750-4 (cited on page 208).

Nielson, Hanne Riis and Flemming Nielson (2007). *Semantics with Applications: an Appetizer*. London, England: Springer-Verlag. ISBN: 978-1-84628-691-9 (cited on pages 55, 63).

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 15, 18, 83, 91).

O’Regan, Gerard (2016). *Guide to Discrete Mathematics: An Accessible Introduction to the History, Theory, Logic and Applications*. Texts in Computer Science. Cham, Switzerland: Springer International Publishing AG. ISBN: ISBN 978-3319445601 (cited on pages 47, 301).

Papadimitriou, Christos H. (1994). *Computational Complexity*. Boston, MA 02116, USA: Addison Wesley. ISBN: 978-0201530827 (cited on pages 16, 18, 243, 263, 265, 274).

Papadopoulos, Yiannis et al. (Mar. 2011). "An approach to optimization of fault tolerant architectures using HiP-HOPS". In: *Journal of Engineering Failure Analysis* 18.2, pages 590–608. ISSN: 1350-6307. DOI: 10.1016/j.engfailanal.2010.09.025 (cited on page 208).

Parr, Terence (2013). *The definitive ANTLR reference: building domain-specific languages*. Raleigh, NC, USA: Pragmatic Bookshelf. ISBN: 978-1-93435-699-9 (cited on page 63).

Petri, Carl Adam (1962). "Kommunikation mit Automaten". PhD thesis. University of Bonn (cited on page 120).

Pierce, Benjamin C. (1991). *Basic Category Theory of Computer Scientists*. Foundations of Computing. Cambridge, MA 02142-1315, USA: MIT Press. ISBN: 978-0262660716 (cited on page 302).

Pinter, Charles C. (2012). *A Book of Abstract Algebra*. New York, USA: Dover Publications Inc. ISBN: 978-0486134796 (cited on pages 47, 302).

— (2013). *A Book of Set Theory*. New York, USA: Dover Publications Inc. ISBN: 978-0486497082 (cited on pages 260, 301).

Plateau, Brigitte and William J. Stewart (1997). "Stochastic Automata Networks". In: *Computational Probability*. Kluwer Academic Press, pages 113–152 (cited on page 208).

Plotkin, Gordon D. (2004). "The Origins of Structural Operational Semantics". In: *Journal of Logic and Algebraic Programming* 60–61, pages 3–15 (cited on page 57).

Pohl, Klaus and Chris Rupp (May 2011). *Requirements Engineering Fundamentals*. Sebastopol, California, USA: O'Reilly. ISBN: 978-1933952819 (cited on page 35).

Point, Gérald and Antoine Rauzy (1999). "AltaRica: Constraint automata as a description language". In: *Journal Européen des Systèmes Automatisés* 33.8–9, pages 1033–1052 (cited on page 209).

Prosvirnova, Tatiana, Michel Batteux, et al. (Sept. 2013). "The AltaRica 3.0 project for Model-Based Safety Assessment". In: *Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013*. York, Great Britain: International Federation of Automatic Control, pages 127–132. ISBN: 978-3-902823-49-6 (cited on page 84).

Prosvirnova, Tatiana and Antoine Rauzy (2015). "Automated generation of Minimal Cutsets from AltaRica 3.0 models". In: *International Journal of Critical Computer-Based Systems* 6.1, pages 50–79. DOI: 10.1504/IJCCBS.2015.068852 (cited on pages 203, 256).

Provan, Gergory M. and Michael O. Ball (Nov. 1983). "The complexity of counting cuts and computing the probability that a graph is connected". In: *SIAM Journal of Computing* 12.4, pages 777–788 (cited on page 269).

Ptolemaeus, Claudius (2014). *System Design, Modeling, and Simulation using Ptolemy II*. http://ptolemy.org/books/Syste Ptolemy.org. ISBN: 978-1304421067 (cited on page 208).

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 97, 166).

Rausand, Marvin (Mar. 2014). *Reliability of Safety-Critical Systems*. Hoboken, New Jersey, USA: Wiley-Blackwell. ISBN: ISBN 978-1-118-11272-4 (cited on page 106).

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 page 156).

— (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 163, 255, 270, 272).

Rauzy, Antoine (Oct. 2002). "Modes Automata and their Compilation into Fault Trees". In: *Reliability Engineering and System Safety* 78.1, pages 1–12. DOI: 10.1016/S0951-8320(02)00042-X (cited on pages 203, 209, 256).

— (2003a). "A New Methodology to Handle Boolean Models with Loops". In: *IEEE Transactions on Reliability* 52.1, pages 96–105. DOI: 10.1109/TR.2003.809272 (cited on pages 193, 194).

— (2003b). "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 156).

— (Oct. 2004). "An Experimental Study on Six Algorithms to Compute Transient Solutions of Large Markov Systems". In: *Reliability Engineering and System Safety* 86.1, pages 105–115. DOI: 10.1016/j.ress.2004.01.007 (cited on page 120).

— (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 page 156).

— (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 pages 177, 186, 193, 209, 248).

— (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 page 156).

— (2018). "Notes on Computational Uncertainties in Probabilistic Risk/Safety Assessment". In: *Entropy* 20.3. DOI: 10.3390/e20030162 (cited on page 4).

— (2020). *Probabilistic Safety Analysis with XFTA*. Les Essarts le Roi, France: AltaRica Association. ISBN: 978-82-692273-0-7 (cited on pages 4, 129, 145, 146, 156, 157, 173).

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 4, 11).

Rauzy, Antoine and Liu Yang (2019a). "Decision Diagram Algorithms to Extract Minimal Cutsets of Finite Degradation Models". In: *Information* 10.368, pages 1–28. DOI: 10.3390/info10120368 (cited on pages 209, 255, 272).

— (2019b). "Finite Degradation Structures". In: *Journal of Applied Logics – IfCoLog Journal of Logics and their Applications* 6.7, pages 1471–1495 (cited on pages 209, 247, 255, 272).

Reliability Prediction of Electronic Equipment: MIL-HDBK-217F. (1995). Technical report. U.S. Department of Defense (cited on page 101).

Rival, Xavier and Kwangkeun Yi (2020). *Introduction to Static Analysis: An Abstract Interpretation Perspective*. Cambridge, MA, USA: MIT Press. ISBN: 978-0262043410 (cited on page 204).

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 pages 15, 17, 18, 25, 36, 91, 208).

Savitch, Walter (1972). "Relationships between nondeterministic and deterministic tape complexities". In: *Journal of Computer and System Sciences* 4.2, pages 177–192. DOI: 10.1016/S0022-0000(70)80006-X (cited on page 265).

Schneier, Bruce (1999). "Attack Trees". In: *Dr Dobb's Journal* 24.12 (cited on page 68).

Shier, Douglas R. (1991). *Network Reliability and Algebraic Structures*. Oxford, England: Oxford Science Publications. ISBN: 978-0198533863 (cited on page 193).

Signoret, Jean-Pierre, Yves Dutuit, et al. (2013). "Make your Petri nets understandable: Reliability block diagrams driven Petri nets". In: *Reliability Engineering and System Safety* 113, pages 61–75. DOI: 10.1016/j.ress.2012.12.008 (cited on pages 208, 209).

Signoret, Jean-Pierre and Alain Leroy (2021). *Reliability Assessment of Safety and Production Systems: Analysis, Modelling, Calculations and Case Studies*. Springer Series in Reliability Engineering. Cham, Switzerland: Springer. ISBN: 978-3030647070 (cited on pages 123, 124, 130, 173, 208, 209).

Simon, Herbert (1957). *Models of Man: Social and Rational. Mathematical Essays on Rational Behavior in a Social Setting*. New York, New Jersey, U.S.A: Wiley (cited on page 273).

— (1996). *The Sciences of the Artificial*. Cambridge, MA 02142-1315, USA: The MIT Press. ISBN: 978-0262691918 (cited on page 36).

Simon, Imré (1978). "Limited subsets of a free monoid". In: *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. Piscataway, N.J., USA: IEEE, pages 143–150. ISBN: 0272-5428. DOI: 10.1109/SFCS.1978.21 (cited on page 302).

SINTEF, prepared by and NTNU, editors (2015). *OREDA Handbook – Offshore Reliability Data, volume 1 and 2, 6th edition* (cited on page 101).

Sterman, John D. (2000). *Business Dynamics: Systems thinking and modeling for a complex world*. New York, NY, USA: McGraw Hill. ISBN: 978-0-07-231135-8 (cited on page 208).

Stewart, William J. (1994). *Introduction to the Numerical Solution of Markov Chains*. Princeton, New Jersey, USA: Princeton University Press. ISBN: 978-0691036991 (cited on pages 119, 124, 208, 230, 253, 270).

Stockmeyer, Larry J. (1976). "The polynomial-time hierarchy". In: *Theoretical Computer Science* 3.1, pages 1–22. DOI: 10.1016/0304-3975(76)90061-X (cited on page 266).

Taleb, Nassim Nicholas (2010). *The Black Swan: the impact of the highly improbable*. London, England: Penguin. ISBN: 978-0-14-103459-1 (cited on page 123).

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 267).

Trivedi, Kishor S. (2001). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Hoboken,New Jersey, USA: Wiley-Blackwell. ISBN: 978-0471333418 (cited on page 270).

Turing, Alan (1936). "On Computable Numbers with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 42.2, pages 230–265. DOI: 10.1112/plms/s2-42.1.230 (cited on page 257).

Ullman, David G. and Brian P. Spiegel (July 2006). "Method and Tools for Constrained System Architecting". In: *Proceedings of the Sixteenth Annual International Symposium of the International Council On Systems Engineering (INCOSE)*. Volume 16. 1. Orlando, FL, USA: Wiley Online Library, pages 1294–1307. DOI: 10.1002/j.2334-5837.2006.tb02813.x (cited on page 32).

Valiant, Leslie G. (1979a). "The Complexity of Computing the Permanent". In: *Theoretical Computer Science* 8.2, pages 189–201. DOI: 10.1016/0304-3975(79)90044-6 (cited on page 267).

— (1979b). "The Complexity of Enumeration and Reliability Problems". In: *SIAM Journal of Computing* 8.3, pages 410–421. DOI: 10.1137/0208032 (cited on pages 252, 267).

van Hentenryck, Pascal (Mar. 1989). *Constraint Satisfaction in Logic Programming*. Cambridge, MA, USA: The MIT Press. ISBN: 978-0262081818 (cited on page 14).

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 pages 30, 86).

von Bertalanffy, Ludwig (2015). *General System Theory: Foundations, Development, Applications (re-publication)*. New York, NY, USA: George Braziller Inc. ISBN: 978-0807600153 (cited on page 36).

Walden, David D. et al. (Aug. 2015). *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, fourth edition*. Hoboken, NJ, USA: Wiley-Blackwell. ISBN: 978-1118999400 (cited on pages 1, 2, 35).

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 171).

Wechler, Wolfgang (Dec. 2013). *Universal Algebra for Computer Scientists*. Monographs in Theoretical Computer Science: An EATCS Series. Berlin and Heidelberg, Germany: Springer-Verlag GmbH & Co. K. ISBN: 978-3642767739 (cited on page 302).

Weilkiens, Tim et al. (2015). *Model-Based System Architecture*. Wiley Series in Systems Engineering and Management. Hoboken, New Jersey, USA: Wiley-Blackwell. ISBN: 978-1118893647 (cited on pages 13, 36).

Welford, B. P. (Aug. 1962). "Note on a method for calculating corrected sums of squares and products". In: *Technometrics* 4.3, pages 419–420. DOI: 10.2307/1266577 (cited on pages 198, 318).

White, Stephen and Derek Miers (2008). *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, FL, USA: Future Strategies Inc. ISBN: 978-0977752720 (cited on pages 13, 25, 36, 84).

Wilensky, Uri and William Rand (2015). *An Introduction to Agent-Based Modeling - Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. Cambridge, MA, USA: The MIT Press. ISBN: 978-0262731898 (cited on page 248).

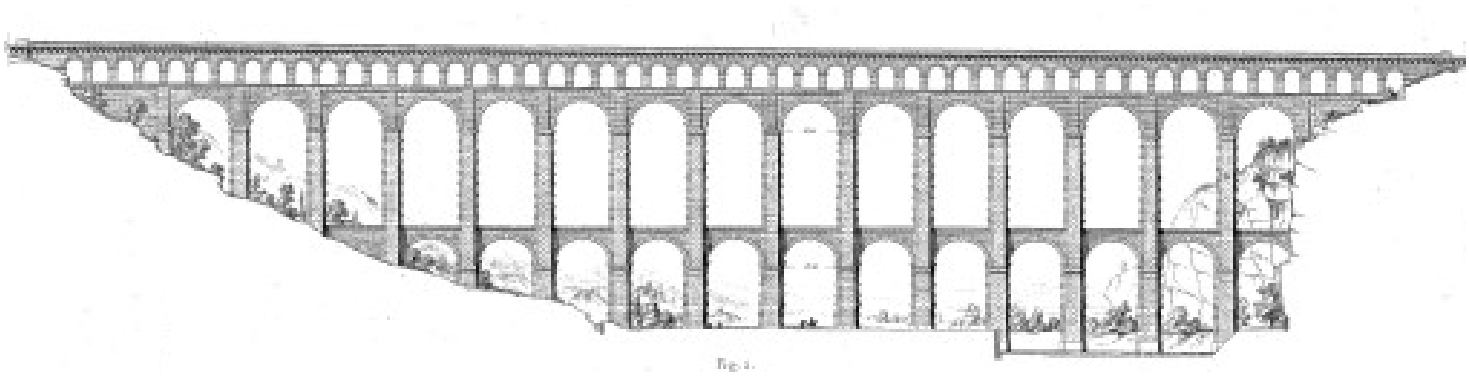
Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Upper Saddle River, New Jersey 07458, USA: Prentice-Hall. ISBN: 978-0130224187 (cited on page 15).

Yakymets, Nataliya, Yupanqui Munoz Julho, and Agnes Lanusse (Oct. 2014). "Sophia framework for model-based safety analysis". In: *Actes du congrès Lambda-Mu 19 (actes électroniques)*. Dijon, France: Institut pour la Maîtrise des Risques. ISBN: 978-2-35147-037-4 (cited on page 208).

Yi, Wang, Paul Pettersson, and Mats Daniels (Oct. 1994). "Automatic Verification of Real-Time Communicating Systems by Constraint Solving". In: *Proceedings of the 7th Conference on Formal Description Techniques, FORTE'1994*. Edited by Dieter Hogrefe and Stefan Leue. Berne, Switzerland: North–Holland, pages 223–238 (cited on page 195).

Zimmermann, Armin (2010). *Stochastic Discrete Event Systems*. Berlin, Heidelberg, Germany: Springer. ISBN: 978-3-642-09350-0 (cited on page 208).

Zio, Enrico (2013). *The Monte Carlo Simulation Method for System Reliability and Risk Analysis*. Springer Series in Reliability Engineering. London, England: Springer London. ISBN: 978-1-4471-4587-5 (cited on pages 196, 271).



A. Sets and Relations

Key Concepts
- Sets
- Relations and functions
- Transitive closures and fixpoints

This appendix recalls a some important notions of discrete mathematics and fixes mathematical notations we use throughout the book. The reader does not need to master these notions, nor even to know them all. But she or he must be aware that no foundational presentation of model-based systems engineering can be made without involving them.

Entire monographs are dedicated to each of the notions introduced here. Consequently, our introduction is necessarily shallow. Section **??** provides some pointers to the litterature for the reader interested in digging this or that topics.

A.1 Sets

Sets are pervasive in mathematics and in systems engineering. The whole edifice of "daily" mathematics is built on the notion of set. We shall mainly recall here the main operations and notations on sets. The reader interested by a more formal presentation (starting with the Zermelo-Fraenkel axiomatization) should refer to mathematical logic textbooks.

A.1.1 Fundamental relations

Set theory begins with the intuitive notion of objects. It assumes a non-empty collection of objects, some of which are sets. Then it introduces a fundamental binary relation "is-member-of" (or "is-element-of") between an object o and a set S, which is denoted as $o \in S$. The complementary relation "is-not-member-of" is denoted $o \notin S$. Of course, for object o and set S, either $o \in S$ or $o \notin S$. Note that, since sets are objects, the membership relation can relate sets as well.

A derived binary relation between two sets is the subset relation, also called set inclusion. If all the members of set S are also members of set T, then S is a subset of T, which is denoted $S \subseteq T$. S

set is a subset of itself: $S \subseteq S$. S is a proper subset set of T if $S \subseteq T$ and there exists at least one object o such that $o \in T$ but $o \notin S$, which is denoted $S \subset T$.

Elements of a set can be defined either extensionally, by enumerating them, e.g. $\{0,2,4,6,8\}$, or intensionally, by means of a well-defined property that is necessary and sufficient to belong to the set, e.g. $\{n \in \mathbb{N}; n \mod 2 = 0 \text{ and } n < 10\}$.

There exists a unique set with no element, which is denoted by $\emptyset$.

Set theory features binary operations on sets:

- The union of the sets S and T, denoted $S \cup T$, is the set of all objects that are a member of S, or T, or both.
- The intersection of the sets S and T, denoted $S \cap T$, is the set of all objects that are members of both S and T.
- Set difference of S and T, denoted $S \setminus T$, is the set of all members of S that are not members of T. When T is a subset of S, the set difference $S \setminus T$ is also called the complement of T in S, which denoted as $T^{\complement}$ when S is clear from the context.

A.1.2 Laws

Set union, intersection and complementary satisfy many identities. Several of these identities or "laws" have well established names.

Identity laws:

$$\begin{aligned} S \cup \emptyset &= S \\ S \cap \emptyset &= \emptyset \end{aligned}$$

Idempotence laws:

$$\begin{aligned} S \cup S &= S \\ S \cap S &= S \end{aligned}$$

Commutativity laws:

$$\begin{aligned} S \cup T &= T \cup S \\ S \cap T &= T \cap S \end{aligned}$$

Associativity laws:

$$\begin{aligned} S \cup (T \cup U) &= (S \cup T) \cup U \\ S \cap (T \cap U) &= (S \cap T) \cap U \end{aligned}$$

Distributivity laws:

$$\begin{aligned} S \cup (T \cap U) &= (S \cup T) \cap (S \cup U) \\ S \cap (T \cup U) &= (S \cap T) \cup (S \cap U) \end{aligned}$$

Absorption laws:

$$\begin{aligned} S \cup (S \cap T) &= S \\ S \cap (S \cup T) &= S \end{aligned}$$

de Morgan's laws:

$$\begin{aligned} (S \cup T)^{\complement} &= S^{\complement} \cap T^{\complement} \\ (S \cap T)^{\complement} &= S^{\complement} \cup T^{\complement} \end{aligned}$$

Involution law:

$$(S^{\complement})^{\complement} = S$$

Set inclusion obeys also the following laws.
Reflexivity law:

$$S \subseteq S$$

Antisymmetry law:

If $S \subseteq T$ and $T \subseteq S$ then $S = T$

Transitivity law:

If $S \subseteq T$ and $T \subseteq U$ then $S \subseteq U$

A.1.3 Other operations on sets

The cardinal of a set is the number of elements of that set. The cardinal of the set S is denoted $|S|$. Model-Based Systems Engineering deals mainly with finite sets, i.e. sets with finitely many elements. Denumerable sets are also commonly used, i.e. sets that can be put in one-to-one correspondence with the set $\mathbb{N}$ of natural numbers. Non-denumerable sets (which include the set $\mathbb{R}$ of real numbers) are seldom used.

The Cartesian product of sets S_1, S_2, $\ldots S_n$ ($n \geq 1$), denoted $S_1 \times \ldots \times S_n$, is the set whose members are all possible ordered tuples $\langle o_1, \ldots, o_n \rangle$ where each object o_i belongs to the set S_i ($1 \leq i \leq n$).

Note that this definition assumes that tuples are objects. This assumption is actually verified as tuples can be seen as convenient notation for a particular type of sets via, for instance, the so-called Kuratowski definition:

$$\langle o_1, \ldots, o_n \rangle = \{\{o_1\}, \{o_1, o_2\}, \ldots, \{o_1, \ldots, o_n\}\}$$

The cartesian product $S \times \ldots \times S$ involving n times, $n > 0$, the same set S is denoted by S^n.

The power set of a set S is the set whose members are all possible subsets of S. The power set of a set S is denoted 2^S. An important result of mathematical logic, obtained by so-called diagonalization arguments, is that for any (finite or infinite) set S, $|S| < |2^S|$.

A cover of a set S is a collection C of sets whose union contains S as a subset. Formally, if

$$C = \{C_i : i \in I\}$$

is an indexed family of sets, then C is a cover of the set S if:

$$S \subseteq \bigcup_{i \in I} C_i$$

C is a partition of S if for any two objects i and j of I, $i \neq j$, $C_i \cap C_j = \emptyset$.

A.2 Relations and functions

A.2.1 Definitions

Relations are at the core of models in systems engineering.

Let $S_1, \ldots, S_n$ be n sets ($n \geq 1$). A relation R over $S_1, \ldots, S_n$ is a subset of the cartesian product $S_1 \times \ldots \times S_n$, i.e. $R \subseteq S_1 \times \ldots \times S_n$.

As for sets, relations can be described extensionally, by enumerating the tuples that belong to the relation, or intensionally, i.e. by a well-defined property that is necessary and sufficient for a tuple to verify to belong to the relation.

Although most of the concepts associated with relations can be generalized to any number of arguments, binary relations focus most of the attention. Let S and T be two sets and R be a relation over $S \times T$. Moreover, let $s \in S$ and $t \in T$. For the sake of the conveniency, $\langle s,t \rangle \in R$ is denoted by sRt (hence the notation $s \in S$ for the membership relation, $S \subseteq T$ for the inclusion relation, and so on).

A function is a particular type of binary relation. Let S and T, a function F from S to T is a relation over $S \times T$ such that for any $s \in S$ there is at most one $t \in T$ such that sFt. The function F is said total if for all $s \in S$ there is a $t \in T$ such that sFt and partial otherwise.

Let $s \in S$. The unique $t \in T$, when it exists, such that sFt is denoted $F(s)$. We say that F maps s on $F(s)$. S is called the domain of the function F and T its codomain (some authors use instead the term range).

We shall denote usually functions with lower case letters f, g, $h\ldots$

Let f be a function from S to T and g be a function from T to U. The composition of f and g, denoted by $g \circ f$ is the function from S to U such that $g \circ f(s) = g(f(s))$. Note that $g(f(s))$ does not always exists if either f or g or both are partial functions. If both f and g are total functions, then so is $g \circ f$.

A.2.2 Properties

Binary relations and functions may satisfy many properties. Several of these properties have well established names.

Let f be a function from the set S to the set T.
- f is injective if for all $t \in T$ there exists at most one $s \in S$ such that $t = f(s)$.
- f is surjective if for all $t \in T$ there exists at least one $s \in S$ such that $t = f(s)$.
- f is bijective if it is both injective and surjective. In this case, f is also said to be a one-to-one correspondence between S and T as for any $s \in S$ there is a unique $t \in T$ such that $f(s) = t$.

Let S be a set and R be a binary relation over $S \times S$.
- R is reflexive if for any $s \in S$, sRs.
- R is irreflexive if for no $s \in S$, sRs.
- R is symmetric if for any $s,t \in S$, sRt implies tRs.
- R is antisymmetric if for any $s,t \in S$, sRt and tRs implies $s = t$.
- R is transitive if for any $s,t,u \in S$, sRt and tRu implies sRu.

R is an order relation if it is reflexive, antisymmetric and transitive. It is a total order if for any $s,t \in S$, either sRt or tRs. It is a partial order otherwise.

R is an equivalence relation if it is reflexive, symmetric and transitive. An equivalence relation R defines a partition of the set S. Namely each subset C of this partition is formed with the elements of S that are in relation with each other. These subsets are called the equivalence classes of the relation.

A.2.3 Transitive closures and fixpoints

The notion of composition of functions can be extended to relations. For the sake of the simplicity, we shall present it here only in the restricted case of binary relations over a set S.

Let S be a set and P and Q be two binary relations over $S \times S$. Then $R = Q \circ P$ is the binary relation over $S \times S$ such that for any $s,t \in S$, sRt if and only if there exists a $u \in S$ such that sPu and uQt.

Let S be a set and R be a binary relation over $S \times S$. The transitive closure of R, denoted by R^+ is the smallest relation over $S \times S$ that is transitive and contains R.

For finite sets, we can construct the transitive closure step by step, starting from R and adding transitive edges. This gives the intuition for a general construction. The transitive closure is then given by the following expression.

$$R^+ = \bigcup_{i \in \mathbb{N}} R^i$$

where R^i is the i-th power of R, defined inductively as follows.

$$\begin{aligned} R^1 &= R \\ R^{i+1} &= R \circ R^i \text{ for } i > 0 \end{aligned}$$

From a practical point of view, rather than to build the R^i's and to accumulate them, we can accumulate them on the fly. This gives raise to two equivalent calculation processes.

$$\begin{aligned} P^1 &= R \\ P^{i+1} &= R^i \cup \left(R \circ P^i\right) \text{ for } i > 0 \end{aligned}$$

$$\begin{aligned} Q^1 &= R \\ Q^{i+1} &= Q^i \cup \left(Q^i \circ Q^i\right) \text{ for } i > 0 \end{aligned}$$

As S is finite, we are sure that the processes of calculating successively $P^1, P^2, \ldots$ and $Q_1, Q_2, \ldots$ reach a fixpoint after a finite number of steps, i.e. there exist integers m and n such that:

$$\begin{aligned} P^{m+1} &= P^m \\ Q^{n+1} &= Q^n \end{aligned}$$

As processes are stabilized, we have $P^m = Q^n = R^+$.

If m and n are the smallest integers verifying the above equaility, it is easy to verify that $2^{n-1} < m \leq 2^n$. The calculation R^+ via the Q_i's seems thus dramatically faster than the calculation via the P^i's. There are some cases however, where it is not possible to use the Q_i's approach because this would involve too large data structures.

■ **Example A.1** Figure A.1 shows a simplified genealogic tree from Noah to Abraham. This tree represents a "is-parent-of" relation: parent(Noah, Shem), parent(Noah, Japeth)...

The transitive closure of the relation "is-parent-of" is the relation "is-ancestor-of".

The first method of calculation (using P-like identities), starts from the parent relation, e.g. parent(Noah, Shem) ⇒ ancestor(Noah, Shem), then adds grand-parents, e.g. parent(Noah, Shem) and ancestor(Shem, Arpachshad) ⇒ ancestor(Noah, Arpachshad), then great-grand-parents, e.g. parent(Noah, Shem) and ancestor(Shem, Shelah) ⇒ ancestor(Noah, Shelah), and so on. It iterates $12 + 1 = 13$ times before reaching the fixpoint.

The second method of calculation (using Q-like identities), starts also from the parent relation, then adds also the grand-parents, but then adds both great-grand-parents and great-great-grand-parents, e.g. ancestor(Noah, Arpachshad) and ancestor(Arpachshad, Eber) ⇒ ancestor(Noah, Eber), and so on. It iterates $4 + 1 = 13$ times before reaching the fixpoint. ■

The notions of transitive closure and fixpoint calculation play a central role in behavioral descriptions of systems. The set S represents the possible states of the system and the relation R is the next-step relation, i.e. sRt if the system can evolve from the state s to the state t under the occurrence of an event such that an operator action, a failure, a reconfiguration, an external event... Starting from a initial state, it is then possible, applying one of the above processes to calculate all of the states the system can reach from this initial state.

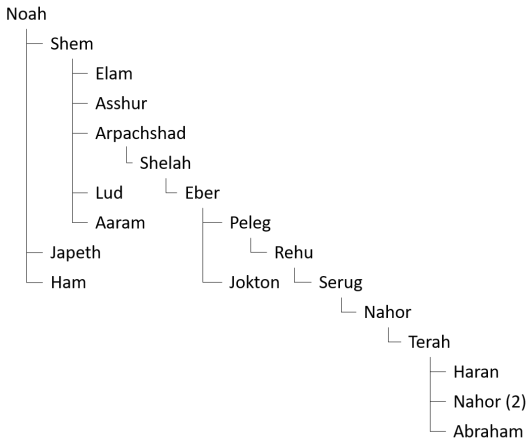
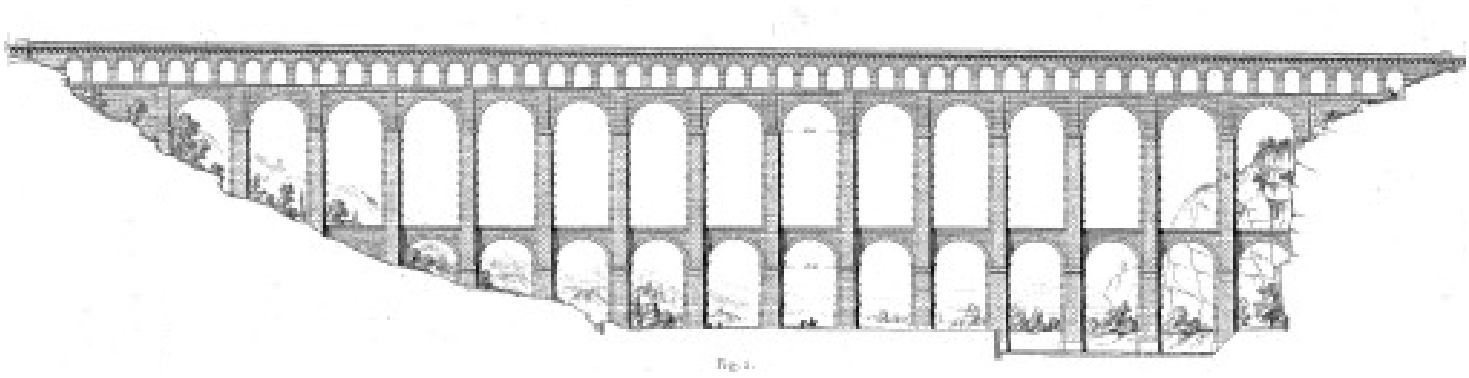


Figure A.1: Simplified genealogic tree from Noah to Abraham



B. Universal Algebra

Key Concepts
- Algebras
- Monoids
- Boolean algebras
- Lattices and morphisms
- Terms and structural induction

Universal algebra (sometimes called general algebra) is the field of mathematics that studies algebraic structures themselves, not examples of algebraic structures. For this reason, it provides an excellent framework to discuss modeling formalisms. We review in this appendix some definitions and concepts that prove to be useful in the framework of model-based systems engineering.

B.1 Definition

An algebra (or algebraic structure) is a pair (U, O), where U is a set (the universe) and O is a collection of operators. Each operator o comes with its arity, a non-negative integer $ar(o)$, and is a, possibly partial, function from $U^{ar(o)}$ to U. A 0-ary (or nullary) operator takes no arguments and returns an element of U, i.e. a constant.

After the operations have been specified, the nature of the algebra can be further limited by axioms, which in universal algebra often take the form of identities, or equational laws.

■ **Example B.1 – Abelian groups.** As an example, consider Abelian groups. They can defined as operating on a set U and three operators: the binary operator $\star$, the nullary operator e (the neutral element) and the unary operator $(.)^{-1}$ (the inverse). Identities defining Abelian groups are then as follows.

$$
\begin{array}{rcll}
x \star y & = & y \star s & \text{Commutativity axiom} \\
x \star (y \star z) & = & (x \star y) \star z & \text{Associativity axiom} \\
x \star e & = & e \star x = x & \text{Neutral element axiom} \\
x \star x^{-1} & = & x^{-1} \star x = e & \text{Inverse axiom}
\end{array}
$$

These axioms are intended to hold for all elements x, y, and z of the universe U. ∎

An algebraic structure that can be defined by identities is called a variety. It is important to note that restricting one's study to varieties rules out quantification, including universal quantification, except before an equation (x, y and z are implicitly universally quantified in the above identities), and existential quantification. It rules out also all relations but equality, in particular inequalities, i.e. both $x \neq y$ and order relations.

Three families of algebraic structures play an important role in Model-Based Systems Engineering: monoids, Boolean algebras and lattices. We shall review them in the next sections, before introducing the concept of morphism which is essential in Model-Based Systems Engineering.

B.2 Monoids

A monoid is a pair made of a set A, called the alphabet, and a set of two operators: a neutral element ε and a binary operator "$\star$" called concatenation (sometimes denoted as "."). Identities verified by these two operators are as follows.

$$\begin{array}{rcccl} x\star(y\star z) & = & (x\star y)\star z & & \text{Associativity axiom} \\ x\star\varepsilon & = & \varepsilon\star x & = & x \quad \text{Neutral element axiom} \end{array}$$

Monoids are also commonly used in computer science, both in its foundational aspects and in practical programming. The set of strings built from a given set of characters is a free monoid. The transition monoid and syntactic monoid are used in describing finite state machines, whereas trace monoids and history monoids provide a foundation for process calculi and concurrent computing. All these topics are indeed strongly connected to behavioral descriptions of systems.

B.3 Boolean Algebras

In mathematics and mathematical logic, Boolean algebra is the branch of algebra in which the values of the variables are the truth values true and false, usually denoted 1 and 0 respectively. 1 and 0 are nullary operators. The main operators of Boolean algebra are the conjunction (and) denoted as $\wedge$, the disjunction (or) denoted as $\vee$, and the negation (not) denoted as $\neg$.

Identities verified by these operators are as follows.

$$
\begin{array}{rcll}
x \wedge y & = & y \wedge x & \text{Commutativity of } \wedge \\
x \vee y & = & y \vee x & \text{Commutativity of } \vee \\
x \wedge (y \wedge z) & = & (x \wedge y) \wedge z & \text{Associativity of } \wedge \\
x \vee (y \vee z) & = & (x \vee y) \vee z & \text{Associativity of } \vee \\
x \wedge (y \vee z) & = & (x \wedge y) \vee (x \wedge z) & \text{Distributivity of } \wedge \text{ over } \vee \\
x \vee (y \wedge z) & = & (x \vee y) \wedge (x \vee z) & \text{Distributivity of } \vee \text{ over } \wedge \\
x \wedge 1 & = & 1 \wedge x = x & \text{Neutral element for } \wedge \\
x \vee 0 & = & 0 \vee x = x & \text{Neutral element for } \vee \\
x \wedge 0 & = & 0 \wedge x = 0 & \text{Annihilator for } \wedge \\
x \vee 1 & = & 1 \vee x = 1 & \text{Annihilator for } \vee \\
x \wedge x & = & x & \text{Indempotence of } \wedge \\
x \vee x & = & x & \text{Indempotence of } \vee \\
x \wedge (x \vee y) & = & x & \text{Absorption via } \wedge \\
x \vee (x \wedge y) & = & x & \text{Absorption via } \vee \\
x \wedge \neg x & = & 0 & \text{Complementation of } \wedge \\
x \vee \neg x & = & 1 & \text{Complementation of } \vee \\
\neg(\neg x) & = & x & \text{Double negation} \\
\neg(x \wedge y) & = & \neg x \vee \neg y & \text{de Morgan's law for } \wedge \\
\neg(x \vee y) & = & \neg x \wedge \neg y & \text{de Morgan's law for } \vee
\end{array}
$$

Aside the above operators, three derived ones are important to know:

$x \Rightarrow y$	which is equivalent to	$\neg x \vee y$	Implication
$x \Leftrightarrow y$	which is equivalent to	$(x \Rightarrow y) \wedge (y \Rightarrow x)$	Equivalence
$x \oplus y$	which is equivalent to	$(x \wedge \neg y) \vee (\neg x \wedge y)$	Exclusive or

Boolean algebra has been fundamental in the development of digital electronics, and is provided for in all modern programming languages. It is also used in set theory, probability theory, statistics, reliability and safety analyses and in branches of systems engineering.

B.4 Lattices

A lattice is an abstract structure consisting of a partially ordered set in which every two elements have a unique least upper bound and a unique greatest lower bound.

Let $(U, \leq)$ be a partially ordered set (poset), and $S \subseteq U$ is an arbitrary subset of U. Then an element $u \in U$ is said to be an upper bound of S if $s \leq u$ for each $s \in S$. A set may have many upper bounds, or none at all. An upper bound u of S is said to be its least upper bound, if $u \leq x$ for each upper bound x of S. A set need not have a least upper bound, but it cannot have more than one. Dually, $l \in L$ is said to be a lower bound of S if $l \leq s$ for each $s \in S$. A lower bound l of S is said to be its greatest lower bound, if $x \leq l$ for each lower bound x of S. A set may have many lower bounds, or none at all, but can have at most one greatest lower bound.

Lattices can be described as algebraic structures over a set U and two binary operators $\sqcap$ and $\sqcup$ obeying the following axioms.

$$
\begin{array}{rcll}
x \sqcap y & = & y \sqcap x & \text{Commutativity of } \sqcap \\
x \sqcup y & = & y \sqcup x & \text{Commutativity of } \sqcup \\
x \sqcap (y \sqcap z) & = & (x \sqcap y) \sqcap z & \text{Associativity of } \sqcap \\
x \sqcup (y \sqcup z) & = & (x \sqcup y) \sqcup z & \text{Associativity of } \sqcup \\
x \sqcap x & = & x & \text{Indempotence of } \sqcap \\
x \sqcup x & = & x & \text{Indempotence of } \sqcup
\end{array}
$$

The partial order relation is then defined by posing $x \leq y$ if and only if $x \sqcap y = x$ or equivalently $x \sqcup y = y$.

Lattices provide thus an algebraic framework to speak about partial order relations, which in turns are used as soon as the notion of preference is involved.

B.5 Morphisms

In many fields of mathematics, morphism refers to a structure-preserving map from one mathematical structure to another.

A morphism f from an algebra $A = (S_A, \{o_1^A, \ldots o_m^A\})$ to an algebra $B = (S_B, \{o_1^B, \ldots o_n^B\})$ is a mapping (a function) from S_A to S_B such that for any operator o^A of arity n, $n \geq 0$, of A there is an operator o^B of arity n of B verifying the following identity (for any $x_1, \ldots, x_n \in S_A$).

$$f(o_A(x_1, \ldots, x_n)) \quad = \quad o_B(f(x_1), \ldots, f(x_n))$$

The source and the target of a morphism (in our case algebras A and B) are often called domain and codomain respectively.

Morphisms are equipped with a partial binary operation, the composition. The composition of two morphisms f and g is defined if and only if the target of f is the source of g, and is denoted $g \circ f$. The source of $g \circ f$ is the source of f, and the target of $g \circ f$ is the target of g. The composition satisfies the following two axioms.

- Identity: For every algebra A, there exists a morphism $I_A : A \to A$ called the identity on A, such that for every other algebra B and every morphism $f : A \to B$ the following identity holds.
$$f \circ I_A \quad = f = \quad I_B \circ f$$
- Associativity: For any three morphisms $f : A \to B$, $g : B \to C$ and $h : C \to D$, the following identity holds.
$$h \circ (g \circ f) \quad = \quad (h \circ g) \circ f$$

A morphism $f : A \to B$ is an isomorphism if there exists a morphism $g : B \to A$ such that $g \circ f = I_A$ (which implies that $f \circ g = I_B$). If it exists, such morphism g is unique, called the inverse of f and denoted by f^{-1}.

The existence of an ismorphism between two algebraic structures shows that these structures are actually the same, up to some notation.

Morphisms (and isomorphisms) play a central role in the definition of the semantics of programming and modeling languages.

B.6 Terms and Structural Induction

So far, we have considered algebraic structures with operators having some specific properties such as commutativity, associativity .... In model-based systems engineering, we have often to describe structures without any particular property but to be represent a hierarchical decomposition. Term algebras capture the mathematical essence of hierarchical decomposition.

Let F be a set of symbols. Let ar be a function of F to non-negative integers. As previously, ar associates to each symbol f of F its arity $ar(f)$. A nullary symbol, i.e. a symbol $f \in F$ such that $ar(f) = 0$ is called a constant. Without a loss of generality, we shall assume that F contains at least one constant.

The set $\mathscr{T}(F)$ of terms over F is the smallest set such that:

- Constants over F are terms.
- If f is a term of F of arity $n > 0$ and $t_1, t_2, \ldots t_n$ are terms, then so is $f(t_1, \ldots, t_n)$.

$\mathscr{T}(F)$ together with F (with the function ar) is called a free algebra. The operator $f \in F$, $ar(f) = n$, takes n terms $t_1,\ldots,t_n$ as arguments, and returns the term $f(t_1,\ldots,t_n)$.

Free algebras play a very important role In mathematical logic through Hebrand's interpretation of predicate calculus. They are also widely used in computer science to represent data structures.

■ **Example B.2 – lists.** Assume for instance that we want to reason about lists of objects. This can be done by introducing a nullary operator "[]" representing the empty list and a binary operator ".". The set of lists is defined as the smallest set such that.

– [] is a list.
– o is an object and L is a list, then $.(o,L)$ is a list.

■

Reasonning about terms means often demonstrating that a given property is true for all terms (or all terms of certain category). Structural induction is a proof method sed to prove that some proposition $P(t)$ holds for all terms t of some sort. It is a generalization of mathematical induction over natural numbers (and can be further generalized to arbitrary Noetherian induction). Structural recursion is a recursion method bearing the same relationship to structural induction as ordinary recursion bears to ordinary mathematical induction.

A proof by structural induction on terms consists in two parts:

– First, a proof that the property holds for all the minimal structures (typically the constants);
– Second, a proof that if it holds for the immediate subterms of a term t, then it holds for t as well.

■ **Example B.3 – Structural induction on binary tree.** Assume for instance we want to prove the property that an ancestor tree extending over g generations shows at most $2^g - 1$ persons.

An ancestor tree is built over a set of persons P, the unary "leaf" operator $\diamond$ and the ternary "node" operator $\triangle$. The set of ancestors trees is defined as the smallest set such that:

– If p is a person of P, then $\diamond(p)$ is an ancestor tree.
– If p is a person of P, and m and f are ancestor trees, then $\triangle(p,m,f)$ is an ancestor tree.

Now the proof goes as follows.

– In the simplest case $\diamond(p)$, the tree shows just one person and hence one generation; the property is true for such a tree, since $1 \leq 2^1 - 1$.
– Now consider a tree $\triangle(p,m,f)$. Let g_m and g_f be respectively the numbers of generations of subtrees m and f and s_m and s_f be respectively the sizes (the numbers of persons) of subtrees m and f. By induction hypothesis, we have $s_m \leq 2^{g_m} - 1$ and $s_f \leq 2^{g_f} - 1$.
  By construction, the number of generations g of the tree $\triangle(p,m,f)$ is $g = max(g_m, g_f) + 1$. The size s of the tree $\triangle(p,m,f)$ is $s = s_m + s_f + 1$. Now, $g_m \leq g - 1$ and $g_f \leq g - 1$. Therefore, $s_m \leq 2^{g-1} - 1$ and $s_f \leq 2^{g-1} - 1$. It follows that $s \leq 2 \times 2^{g-1} - 2 + 1 = 2^g - 1$.

■

Structural induction on terms is correct because subterms of a term are smaller than the term. It is thus possible to define a well-founded partial order over terms, i.e. an order relation with no infinite descending chains. The existence of such well-founded partial is the key of correctness of the structural induction proof.

B.7 Further Readings

This appendix introduces a very large number of topics. Not a single book covers all of them, although the book by O'Regan (O'Regan 2016), can be considered as a rather general introduction. Here follows some suggested readings.

There exist many introductions to set theory, relations and functions and mathematical logic.

– The book by Pinter (Pinter 2013) provides a thorough presentation of set theory.

– The book by Carnap (Carnap 2003) is a rather complete introduction to symbolic logic.

For students who prefer to read French, we can suggest the following.
– The books by Cori and Lascar (Cori and Lascar 2003a) and (Cori and Lascar 2003b).
– The older (but still very good) books by Gochet and Gribomont (Gochet and Gribomont 1990) and (Gochet and Gribomont 1994).

Advanced introductions to universal algebra:
– The book by Wechler (Wechler 2013).
– Another book by Pinter (Pinter 2012).

Category theory is an alternative to set theory as a foundation of mathematics. It has strong connection with programming, especially functional programming. We did not mention it throughout this chapter for this would go beyond the scope of this book. Two useful references about category theory:
– The book by Mac Lane which is rather intended for mathematicians (Lane 1998).
– The book by Pierce which is rather intended for computer scientists (Pierce 1991).

B.8 Exercises

Exercise B.1 The Nim game is a two players games. The initial position is made of three heaps containing respectively 1, 3 and 5 stones. Each player takes in turn either 1, 2 or 3 stones in one of the heap. The player who takes the last stone loses the game.

Question 1. Describe all possible positions of this game as a Cartesian product.

Question 2. Describe all possible moves of this game as a binary relation $move(P,Q)$ over positions.

Question 3. Describe lost positions as a unary relation $lost(P)$ over positions.

Question 4. The winning and losing positions can be described by means of the following recursive identity.

$$winning(P : position) = \exists Q : position\, move(P,Q) \wedge losing(Q)$$
$$losing(P : position) = lost(P) \vee \forall Q : position\, move(P,Q) \Rightarrow winning(Q)$$

Justify this fixpoint definition.

Question 5. Calculate relations *winning* and *losing*. Is there any winning strategy for the first player (or the second one)?

Hint: It may be useful to design a small program to help you in the above task... ■

Exercise B.2 Assume we are given a set $C = \{c_1, \ldots, c_n\}$ of cities together with a partial function d from $C \times C$ to $\mathbb{R}$ that associates with a pair of cities (c,d) the length of the road connecting these two cities, if c and d are directly connected (i.e the road from c to d that does not pass by another city of the set). We would like to calculate the distance between any two pairs of cities. To do so, we shall consider the min-plus algebra, also called tropical algebra, introduced by Simon (I. Simon 1978). This algebra is built over $\mathbb{R}^+ \cup \{+\infty\}$ and the two binary operators $\oplus$ and $\otimes$ defined as follows:

$$\begin{aligned} a \oplus b &= min(a,b) \\ a \otimes b &= a + b \end{aligned}$$

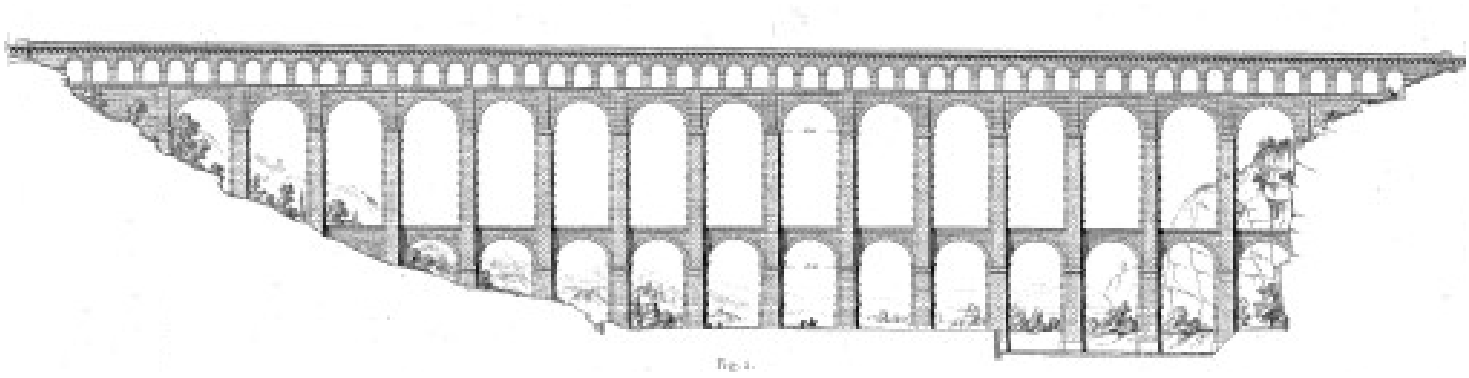
Question 1. Show that $\oplus$ and $\otimes$ are associative and commutative, that there exist a neutral and an absorbing element for $\oplus$ and $\otimes$, and that $\otimes$ is distributive over $\oplus$, i.e. that the following equality holds.

$$(a \oplus b) \otimes c \quad = \quad (a \otimes c) \oplus (b \otimes c)$$

Question 2. Use the min-plus algebra to define the distance between any two pairs of cities as the transitive closure of a certain relation.

■

Hanoi tower?
Quelque chose sur les monoides (trace?)
Quelque chose sur les relations d'ordre
Semantique operationnelle du language while



C. Probability Theory and Statistics

Key Concepts
- Axiomatic of probability theory
- Sample space, event, σ-algebra
- Probability measure
- Sylvester-Poincaré development
- Conditional probabilities, Bayes's theorem
- Random variable, cumulative distribution function
- Expected value
- Mean, variance, standard deviation
- Laws of large numbers, central limit theorem
- Important probability distributions
- Quantiles
- Random number generators

Part of model-based systems engineering relies on probability theory. This appendix recalls the main definitions and results of probability theory, gives some important probability distributions and recalls basic statistics.

C.1 Probability Theory

C.1.1 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 C.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 usualy called Ω.

Definition C.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 fullfil 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 C.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 C.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*).

C.1.2 Additional Definitions and Properties

The above definition induces a number of well-known properties.

Proposition C.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 C.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 C.1.5 – 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 occur-

rence of an event, given that another event occurred

Definition C.1.6 – 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.

$$p(A \mid B) \stackrel{def}{=} \frac{p(A \cap B)}{p(B)}$$

It follows immediately from the definitions that if A and B are independent events, the following equality holds.

$$p(A \mid B) = p(A) \tag{C.1}$$

We shall conclude this section by the very useful Bayes's theorem, also called theorem about the probability of causes.

Theorem C.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.

$$p(A \mid B) = \frac{p(B \mid A) \times p(A)}{p(B)}$$

C.1.3 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 C.1.7 – 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 C.1.8 – 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 $r \in \mathbb{R}$).

$$F_X(r) \stackrel{def}{=} p(\{\omega \in \Omega : X(\omega) \leq r\})$$

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 C.1.9 – Expected value (finite or denumerable case). Let X be a random variable with a countable set of finite outcomes x_1, x_2, ..., occurring with probabilities p_1, p_2, ..., 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.

$$E(X) \stackrel{def}{=} \sum_{i=1}^{\infty} x_i \times p_i$$

In the infinite uncoutable case, the formal definition is as follows.

Definition C.1.10 – 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.

$$E(X) \stackrel{def}{=} \int_{\mathbb{R}} x f(x)\, dx$$

Here follows a basic property of expected value.

Proposition C.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 C.1.11 – 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.

$$\mathrm{Var}(X) \stackrel{def}{=} E\left[(X - E[X])^2\right]$$

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 C.1.12 – 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.

$$\sigma(X) \stackrel{def}{=} \sqrt{\mathrm{Var}(X)}$$

C.1.4 Laws of Large Numbers and Theorem Central Limit

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 C.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.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

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 C.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.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

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.

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 C.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.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$
$$Z_n \stackrel{def}{=} \frac{\overline{X}_n - \mu}{n/\sqrt{n}}$$

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 $z \in \mathbb{R}$,

$$\lim_{n\to\infty} Pr(Z_n \leq z) = \Phi(z)$$

Table C.1: Characteristics of the uniform distribution

Probability density function	$f(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{if } x > b \end{cases}$
Cumulative probability function	$F(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{x-a}{x-b} & \text{if } a \leq x \leq b \\ 1 & \text{if } x > b \end{cases}$
Mean	$\frac{1}{2}(b-a)$
Median	$\frac{1}{2}(b-a)$
Variance	$\frac{1}{12}(b-a)^2$

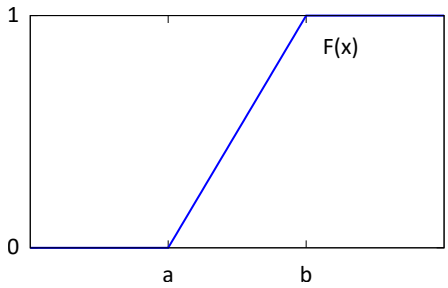


Figure C.1: Cumulative distribution function of the uniform distribution

C.2 Some Important Probability Distributions

Many modeling techniques (beyond systems engineering) require the association of probability distributions with parameters of the model. In practice, there are two ways of defining these probability distributions:

- The first one consists in using parametric distributions.
- The second one consists in using so-called empirical distributions, i.e. distributions defined by a set of points between which the value of the function is interpolated.

We shall review them in turn.

C.2.1 Parametric distributions

The following list of parametric distributions gathers only those that are frequently used in the framework of model-based systems engineering. There exists indeed many others, used in different contexts.

Uniform distribution

The *continuous uniform distribution*, or simply *uniform distribution*, is such that for each member of the family, all intervals of the same length on the distribution's support are equally probable. The support is defined by the two parameters, a and b, $a \leq b$, which are its minimum and maximum values.

Table C.1 gives the characteristics of the uniform distribution.

Figure C.1 shows the shape the cumulative distribution function of the uniform distribution.

Table C.2: Characteristics of the normal distribution

Probability density function	$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
Cumulative probability function	$F(x) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right)$
Mean	μ
Median	μ
Variance	σ^2

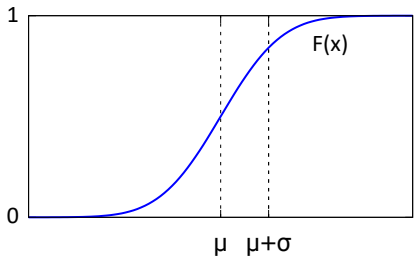


Figure C.2: Cumulative distribution function of the normal distribution

Normal distribution

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 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 $\mathcal{N}(\mu,\sigma)$.

Table C.2 gives the characteristics of the normal distribution. In this table, the function $\text{erf}(x)$ is the error function defined as follows.

$$\text{erf}(x) \stackrel{def}{=} \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{C.2}$$

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

Figure C.2 shows the shape the cumulative distribution function of the normal distribution.

Lognormal distribution

A *lognormal distribution* is a continuous probability distribution of a random variable whose logarithm is normally distributed. Thus, if the random variable X is lognormally distributed, then $Y = \ln X$ has a normal 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 its mean μ and standard-deviation σ.

Table C.3 gives the characteristics of the lognormal distribution.

Figure C.3 shows the shape the cumulative distribution function of the lognormal distribution.

Exponential distribution

The *exponential distribution* represents typically the life-span of a component without memory, aging nor wearing (Markovian hypothesis). The probability that the component is working at least $t+d$ hours knowing that it worked already t hours is the same as the probability that it works d hours after its entry into service. In other words, the fact that the component worked correctly for t hours does not change its expected life duration after this delay.

Table C.3: Characteristics of the lognormal distribution

Probability density function	$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x-\mu)^2}{2\sigma^2}\right)$
Cumulative probability function	$F(x) = \frac{1}{2}\left(1+\text{erf}\left(\frac{\ln x-\mu}{\sigma\sqrt{2}}\right)\right)$
Mean	$\exp\left(\mu+\frac{\sigma^2}{2}\right)$
Median	$\exp(\mu)$
Variance	$\left(\exp\left(\sigma^2\right)-1\right)\exp\left(2\mu+\sigma^2\right)$

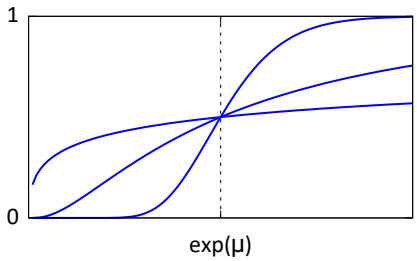


Figure C.3: Cumulative distribution function of the lognormal distribution

The exponential distribution is defined by means of a single parameter, the transition rate, usually denoted by λ. This transition rate is the inverse of the mean life expectation.

Table C.4 gives the characteristics of the exponential distribution.

Figure C.4 shows the shape the cumulative distribution function of the exponential distribution.

Weibull distribution

The exponential distribution assumes a constant failure rate over the time. This is not always realistic because of aging effects: at the beginning of its life the component has a decreasing failure rate, corresponding to debug (or infant mortality), then for a long while, its failure rate remains constant, then the wearout period starts where the failure rate increases. This is the so-called bathtub curve.

This phenomenon is (piece wisely) captured by the *Weibull distribution* which takes two parameters: he shape parameter, usually denoted α, and the scale parameter, usually denoted β.

Table C.5 gives the characteristics of the Weibull distribution. In this table, the Γ function is defined as follows.

$$\Gamma(z) \stackrel{def}{=} \int_0^\infty x^{z-1}e^{-x}dx \tag{C.3}$$

Figure C.4 shows the shape the cumulative distribution function of the Weibull distribution.

Table C.4: Characteristics of the exponential distribution

Probability density function	$f(x) = \lambda e^{-\lambda x}$
Cumulative probability function	$F(x) = 1-e^{-\lambda x}$
Mean	λ^{-1}
Median	$\lambda^{-1}\ln 2$
Variance	λ^{-2}

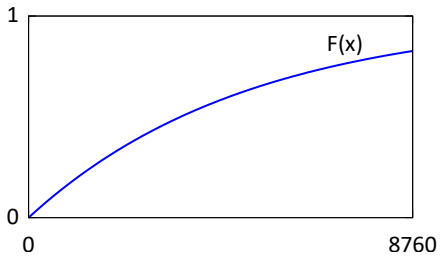


Figure C.4: Cumulative distribution function of the exponential distribution

Table C.5: Characteristics of the Weibull distribution

Probability density function	$f(x) = \frac{\beta}{\alpha}\left(\frac{x}{\alpha}\right)^{\beta-1} e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$
Cumulative probability function	$F(x) = 1 - e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$
Mean	$\alpha\Gamma\left(1+\frac{1}{\beta}\right)$
Median	$\alpha(\ln 2)^{\frac{1}{\beta}}$
Variance	$\alpha^2\left(\Gamma\left(1+\frac{2}{\beta}\right) - \left(\Gamma\left(1+\frac{1}{\beta}\right)\right)^2\right)$

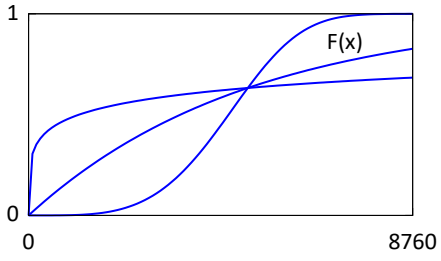


Figure C.5: Cumulative distribution function of the Weibull distribution

Table C.6: Characteristics of the triangular distribution

Probability density function	$f(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{if } a \le x < c \\ \frac{2}{b-a} & \text{if } x = c \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{if } c < x \le b \\ 0 & \text{if } x > b \end{cases}$
Cumulative probability function	$F(x) = \begin{cases} 0 & \text{if } x \le a \\ \frac{(x-a)^2}{(b-a)(c-a)} & \text{if } a < x \le c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & \text{if } c < x < b \\ 1 & \text{if } x \ge b \end{cases}$
Mean	$\frac{a+b+c}{3}$
Median	$\begin{cases} a + \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \ge \frac{a+b}{2} \\ b - \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \le \frac{a+b}{2} \end{cases}$
Variance	$\frac{a^2+b^2+c^2-ab-ac-bc}{18}($

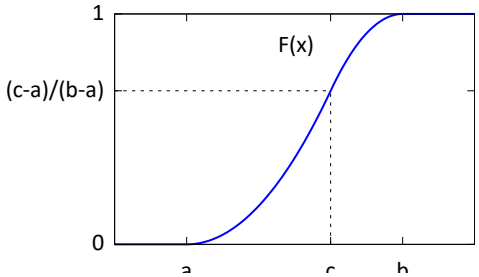


Figure C.6: Cumulative distribution function of the triangular distribution

Triangular distribution

The *triangular distribution* is a continuous probability distribution with lower limit a, upper limit b and mode c, where $a < b$ and $a \le c \le b$.

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).

Table C.6 gives the characteristics of the triangular distribution.

Figure C.6 shows the shape the cumulative distribution function of the triangular distribution.

Binomial distribution

The *binomial distribution* of parameters n and p is a discrete probability distribution. It represents the number of successes in a series of n independent experiments, each asking a yes–no question. Each experiment gives the answer yes with the probability p and no with the probability $q = 1 - p$.

The binomial distribution is frequently used to model the number of successes in a sample of size n drawn with replacement from a population of size N. In case the sample is drawn without replacement,so the resulting distribution is a hypergeometric distribution. However, if N much larger than n, the probability to draw twice the same individual is very low, so the binomial

Table C.7: Characteristics of the binomial distribution

Probability mass function	$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$
Cumulative probability function	$F(k) = \sum_{i=0}^{\lfloor k \rfloor} \binom{n}{i} p^i (1-p)^{n-i}$
Mean	np
Median	$\lfloor np \rfloor$ or $\lceil np \rceil$
Variance	$np(n-p)$

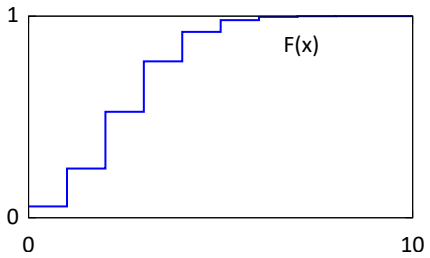


Figure C.7: Cumulative distribution function of the binomial distribution for $n = 10$ and $p = 0.25$

distribution is a good approximation of the hypergeometric distribution.

Table C.7 gives the characteristics of the binomial distribution.

Recall that the expression for the binomial is as follows.

$$\binom{n}{k} \stackrel{def}{=} \frac{n!}{k!(n-k)!} \tag{C.4}$$

Figure C.7 shows the shape the cumulative distribution function of the binomial distribution for $n = 10$ and $p = 0.25$.

The *Bernoulli distribution* is a special case of the binomial distribution where a single trial is conducted, i.e. $n = 1$.

C.2.2 Empirical distributions

Empirical distributions are given by a list of points $(x_1, y_1), \ldots, (x_n, y_n)$, $n \geq 2$, such that $x_1 < \ldots < x_n$ and, if one considers cumulative distribution functions, $y_1 \leq \ldots \leq y_n$. They are typically obtained via series of observations.

In between points, the value of the function is obtained by interpolation. There are basically two ways to do this interpolation:

– To consider the distribution as a *stepwise distribution*, i.e.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases} \tag{C.5}$$

– To consider the distribution as a *piecewise uniform distribution*, i.e.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i + (y_{i+1} - y_i)\frac{x - x_i}{x_{i+1} - x_i} & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases} \tag{C.6}$$

■ **Example C.1 – Piecewise uniform distribution.** Figure C.8 shows a piecewise uniform distribution defined by 4 points: (0,0), (1000, 0.3), (7000, 0.4) and (8760, 0.876). ■

Until recently, empirical distributions were mostly used when it was hard to fit them with any parametric distribution. For instance, the *Kaplan–Meier estimator* (Kaplan and Meier 1958), also

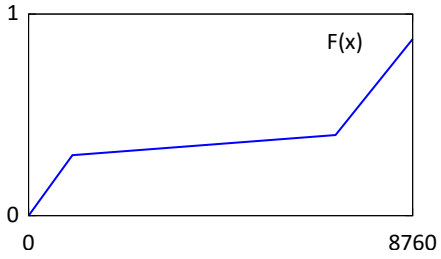


Figure C.8: Cumulative distribution function of the piecewise uniform distribution

known as the product limit estimator, is a non-parametric statistic used to estimate the survival function from lifetime data. In reliability engineering, Kaplan–Meier estimators may be used to measure the time-to-failure of machine parts. In medical research, they are often used to measure the fraction of patients living for a certain amount of time after treatment. This situation may generalize with easy internet communications: there is no more need to store data into a very compact form (the name of the parametric function and its parameters). Therefore, there is less and less reasons to spend time and energy in fitting empirical observations with parametric distributions, not to speak about the arbitrariness of the choice of the parametric distribution.

C.3 Basic Statistics

Essentially two types of indicators can be observed during Monte-Carlo simulations or similar types of experiments:

- Numerical indicators that are assimilated to real-valued variables.
- Discrete indicators such as Boolean variables or variables taking their values into a small set of symbolic constants.

The statistics made on these two types of indicators are different. For numerical indicators, one is mostly interested in how the values of the indicator are distributed. For discrete indicators, one is mostly interested in the frequency of each value of the indicator.

We shall consider in turn both cases. But before doing so, an important practical remark must be made: To get significant results, one often needs to consider very large sample, i.e. number of executions of the model. But in such case, storing all individual values taken by the indicators would lead to a memory overflow. It would be indeed possible to store values into an external memory (e.g. hard disk), but accesses to external memories are very slow. Doing so would therefore slow down dramatically the Monte-Carlo simulation. Consequently, statistics must be made using a reasonable amount of memory. As for modeling in general, there is here a tradeoff to find between the accuracy of the description and the ability to perform calculations.

C.3.1 Moments

In statistics, a *moment* is a specific quantitative measure of the shape of a function. If the function is a probability distribution, then the first moment is the *mean*, the second central moment is the *variance*, the third standardized moment is the *skewness*, and the fourth standardized moment is the *kurtosis*. The mathematical concept is closely related to the concept of moment in physics.

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.

$$\mu_n \stackrel{def}{=} \int_{-\infty}^{\infty} (x-c)^n f(x) dx \tag{C.7}$$

The moment of a function, without further details, refers usually to the above expression with $c = 0$.

Note that the n-moment does not necessarily exist (because the above integral may be infinite).

If f is a probability density function, then the value of the integral above is called the n-th moment of the probability distribution. More generally, if F is a cumulative probability distribution function of any probability distribution, which may not have a density function, then the n-th moment of the probability distribution is given by the following integral.

$$\mu'_n \stackrel{def}{=} E[X^n] = \int_{-\infty}^{\infty} x^n dF(x) \tag{C.8}$$

where X is a random variable that has this cumulative distribution F, and E is the expectation operator or mean.

Mean

The *mean* of a probability distribution is thus the long-run arithmetic average value of a random variable having that distribution. In this context, it is also known as the *expected value* (see Section C.1).

For a data set $S = \{x_1, x_2, \ldots, x_n\}$, typically obtained via a Monte-Carlo simulation, the *arithmetic mean*, also called the *mathematical expectation* or *average*, typically denoted by $\bar{x}$ (pronounced "x bar"), is the sum of the values divided by the number of values in the data set.

$$\bar{x} \stackrel{def}{=} \frac{\sum_{i=1}^{n} x_i}{n} \tag{C.9}$$

$\bar{x}$ must be distinguished from the mean μ of the underlying distribution. 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.

As pointed out above, 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.

■ **Example C.2 – Muffins.** A cafeteria manager wants to analyze the number of muffins sold each day at the cafeteria, in order to better serve clients and avoid losses. To do so, she samples ten days at random over one month and gets the following results.

day	1	2	3	4	5	6	7	8	9	10
x_i	38	11	36	28	10	18	37	12	14	11

To estimate the mean number of muffins sold each day, she can just record, day after day, the sum of the numbers of muffins sold so far.

day	1	2	3	4	5	6	7	8	9	10
$\sum x_i$	38	49	85	113	123	141	178	190	204	215

Then, to get her estimate, she has just to divide the last sum by the number of days: $\bar{x} = \frac{215}{10} = 21.5$. Note that the mean is not an integer, while obviously the cafeteria does not sold fractions of muffins.

■

Variance and standard-deviation

As for the mean, we need to estimate the variance and the standard-deviation from the sample (see Appendix C for mathematical developments on these two indicators).

Recall that the *variance* of a random variable X, denoted $\mathrm{Var}(X)$, is the expectation of the squared deviation of X from its mean μ:

$$\mathrm{Var}(X) \stackrel{def}{=} E\left[(X-\mu)^2\right]$$

Intuitively, $\mathrm{Var}(X)$ measures how far a set of (random) numbers are spread out from their average value.

The *standard-deviation* of a random variable X, denoted $\sigma(X)$, is the square root of its variance.

$$\sigma(X) \stackrel{def}{=} \sqrt{\mathrm{Var}(X)}$$

The expression defining the variance can be expanded:

$$\begin{aligned}\mathrm{Var}(X) &\stackrel{def}{=} E\left[(X-E[X])^2\right]\\ &= E\left[X^2-2XE[X]+E[X]^2\right]\\ &= E[X^2]-2E[X]E[X]+E[X]^2\\ &= E[X^2]-E[X]^2\end{aligned}$$

The naïve algorithm to estimate the empirical variance $\overline{\mathrm{Var}}(X)$ (and the empirical standard deviation $\overline{\sigma}(X)$) from a sample 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 the sample, and in dividing their difference by the number of values, i.e. 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.

$$\overline{\mathrm{Var}}(X) \stackrel{def}{=} \frac{\sum_{i=1}^n x_i^2}{n} - \left(\frac{\sum_{i=1}^n x_i}{n}\right)^2 \tag{C.10}$$

$$\overline{\sigma}(X) \stackrel{def}{=} \sqrt{\overline{\mathrm{Var}}(X)} \tag{C.11}$$

In case the two components of the right hand side of equation C.10 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" (Welford 1962).

In most of practical cases however, the naïve algorithm is good enough.

■ **Example C.3 – Muffins (bis).** Consider again our cafeteria example. To estimate the variance and the standard-deviation of the number of muffins sold, the cafeteria manager can just record, day after day, the sum of the squares of the numbers of muffins sold (in addition to the sum of the numbers of muffins sold):

day	1	2	3	4	5	6	7	8	9	10
x_i	20	16	30	28	21	19	20	21	18	22
$\sum x_i$	20	256	900	784	441	361	400	441	324	215
x_i^2	400	121	1296	784	100	324	1369	144	196	484
$\sum x_i^2$	400	656	1556	2340	2781	3142	3542	3983	4307	4791

Then, she can just apply the naïve algorithm: $\overline{\text{Var}}(X) = \frac{4791}{10} - 21.5^2 = 16.85$ and $\overline{\sigma}(X) = 4.10$. She can notice that, as intuitively expected from the observations, the variance and the standard-deviation are rather large. ∎

Error factors and confidence intervals

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* 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.

Let $\bar{x}$ and $\overline{\sigma}(x)$ be respectively the observed mean and standard-deviation on a sample of size n. Then, the error factor $\text{EF}_\alpha(x)$ corresponding to a confidence level α is defined as follows.

$$\text{EF}_\alpha(x) \stackrel{def}{=} t_\alpha \times \frac{\overline{\sigma}(x)}{\sqrt{n}} \tag{C.12}$$

The confidence interval $\text{CI}_\alpha(x)$ is then defined as follows.

$$\text{CI}_\alpha(x) \stackrel{def}{=} [\bar{x} - \text{EF}_\alpha(x), \bar{x} + \text{EF}_\alpha(x)] \tag{C.13}$$

The 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 $t_{90\%} = 1.64$, $t_{95\%} = 1.96$ and $t_{99\%} = 2.58$.

■ **Example C.4 – Muffins (ter).** Consider again the cafeteria example. Based on her previous calculations, the cafeteria manager can now calculate errors factors and confidence intervals for the number of muffins sold daily (recall that $\bar{x} = 21.5$ and that $\sigma(x) = 11.30$).

Confidence level	Error factor	Confidence interval
90%	2.13	[19.37, 23.63]
95%	2.54	[18.96, 24.04]
99%	3.35	[18.15, 24.85]

The above intervals can be interpreted as follows. There are 90 chances out of 100 that the mean number of muffins sold daily lies somewhere between 19.37 and 23.63, 95 chances out of 100 that it lies between 18.96 and 24.04, and finally 99 chances out of 100 that it lies between 18.15 and 24.85.

Note, and this is very important to understand, that the parameter that is estimated is the mean number of muffins daily sold, not the number itself.

Note also that these figures assume a normal distribution for the number of muffins daily sold. However, a simple look at the data shows two "outliers" at day 3 and 4. It may be the case that, these two days are the two Wednesday's of the sample, which turn out to be the day of the weekly gathering of the Knitting Angels, a gang of grey-haired who use to come at the cafeteria to sip a coffee and eat pastries. The indicators calculated so far would much more informative by treating these two days separately. ∎

C.3.2 Distributions and quantiles

For symbolic values, the calculation of the moments is sufficient. For numerical values, one may be interested in addition to the distribution of values as well as quantiles.

Distributions

In principle, extracting a distribution is rather simple. Let $x_1,\ldots,x_n$ the n numerical values in the sample. The extraction of the distribution 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 same size $I_1,\ldots,I_k$.
3. By scanning again all x_i's, one determines how many values lie in each interval. The cumulative distribution function is obtained by summing the number of values in intervals preceding the considered intervals.

The problem with this approach is indeed that it requires to store all of the values. So in practice, the minimum and maximum values (and therefore the intervals) are chosen *a priori* in such way that all values lie between them and that they are not too far from the actual minimum and maximum. The number of values in each interval is then maintained "on-the-fly".

■ **Example C.5 – Muffins (quater).** Consider again the cafeteria example. The cafeteria manager may decide *a priori* that the number of muffins sold daily ranges from 15 to 30, and to split this interval into four sub-intervals: [15,18], [19,22], [23,26] [27,30] She can then follow on a day by day basis, the evolution of the number of muffins sold daily falling in each of these four sub-intervals.

day	1	2	3	4	5	6	7	8	9	10
x_i	20	16	30	28	21	19	20	21	18	22
[15,18]	0	1	1	1	1	1	1	1	2	2
[19,22]	1	1	1	1	2	3	4	5	5	6
[23,26]	0	0	0	0	0	0	0	0	0	0
[27,30]	0	0	1	2	21	2	2	2	2	2

This distribution is quite informative: it shows that most of the days, the number of muffins sold lies in the second sub-interval, i.e. between 19 and 22. It shows also that there are two outliers, lying in the fourth sub-interval. The fact that they are "exceptional" is confirmed by the emptiness of the third sub-interval. ■

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 splitted 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 book. 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".

■ **Example C.6 – Muffins (cinque).** Consider again the cafeteria example. The cafeteria manager may decide to calculate quartile. To do so she needs first to sort x_i's. Then to split the sorted list into 4 sub-lists of about equal size. E.g.

sorted x_i	16	18	19	20	20	21	21	22	28	30
quartile	1		2			3			4	

The first quartile is thus between 18 and 19, the second one that is the median, between 20 and 21, the third one between 22 and 28, finally the last one is 30. ■

C.4 Random-Number Generators

The Monte-Carlo simulation method relies fully on the generation at random of numbers. This section gives some hints on how this generation is performed in practice.

C.4.1 Algorithmic generators

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 book (see Section **??** for reading advices).

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 model-based systems engineering, 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;

Congruential pseudo-random number generators (attempt to) fulfill the needs. A *congruential pseudo-random number generators* is a function f that takes an integer as input (coded onto a finite number of bits, e.g. 64 bits on modern computers), called the *seed*, and returns an integer. Given the initial seed z_0, it is thus possible to generate an arbitrary long sequence of numbers: $z_1 = f(z_0)$, $z_2 = f(z_1), \ldots$

In the second half of the 20th century, the standard pseudo-random number generators were linear congruential generators, i.e. generators based on functions of the form:

$$f(z) \stackrel{def}{=} (a \times z + c) \mod m$$

Note that, because congruential generators work with only a finite number of integers (those coded by the machine), there necessary exist two indices i and j, $i < j$, such as $z_i = z_j$. Consequently, at some step, the generator loops. The distance between i and j is called the *period* of the generator. For some seeds, the period may be shorter than for some others.

The periodicity of congruential pseudo-random number generators was really an issue for Monte-Carlo simulations when integers were coded on 32 bits. For large number of tries (e.g. 1 million), they were good chances that the same executions were reproduced (as they started with the same seed). The problem was known to be inadequate, but better methods were unavailable.

Fortunately, this problem is 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 librairies of programming languages such as Python, Matlab, R...

C.4.2 Generation according to probability distributions

The algorithmic random generators we presented so far generate numbers ranging from 0 to the biggest integer `maxint` representable in one machine word (i.e. on 64 bits, for the most part of computers we are using today).

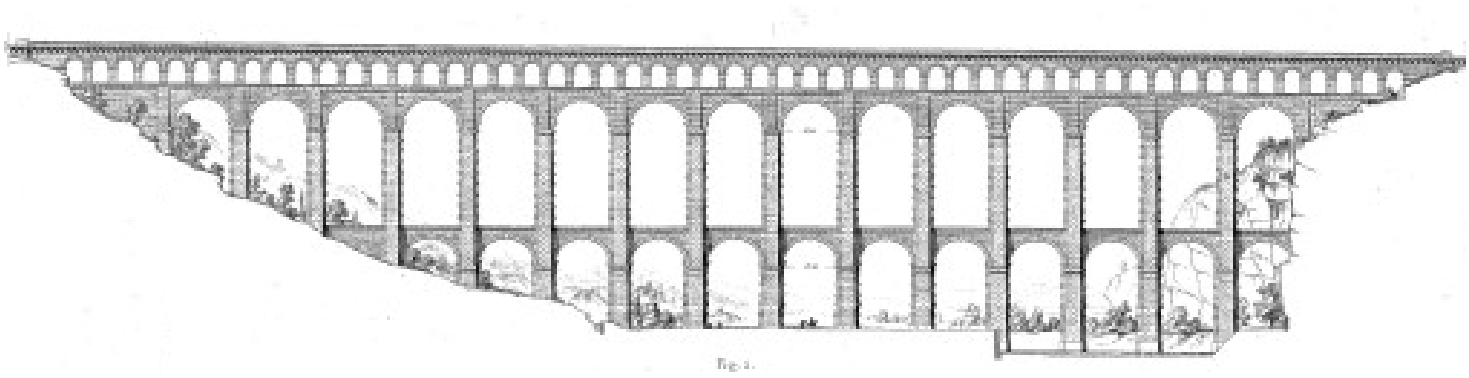
However, what we really would like is to generate real numbers, according to some predefined distribution such as those presented Section C.2.

To get floating point numbers uniformly distributed between 0 and 1, it suffices to divide the generated x_i's by `maxint`.

Having a uniform generator between 0 and 1, it is easy to transform it into a generator according to most of the distributions we have seen so far: for parametric distributions with an invertible cumulative distribution function, it suffices to generate a number z between 0 and 1, and then to take $x = F^{-1}(z)$. This works fine for uniform, exponential, Weibull and triangular distributions. The same idea applies also to empirical distributions.

The situation is more complex when considering normal (and thus lognormal) distributions, as there is no easily calculable inverse function. Fortunately, there exist relatively simple methods to generate a normal distribution from an uniform distribution, e.g. the Box-Muller method (G. E. P. Box and Muller 1958).

Note again that the above mentioned librairies of programming languages such as Python provide built-ins to generate floating point numbers according to wide variety of parametric distributions.



D. Expressions

Here follows a number of operators implemented by the family of languages S2ML+X. Some of these operators are not available in all of the languages. Reciprocally, operators available in some of the languages are not presented here. In any case, the reader should refer to the technical documentation associated with each language (and tool) to know which operators are available and which are not.

D.1 Boolean expressions

Table D.1 presents Boolean operators. These operators come with the two Boolean constants `true` and `false`.
Examples of Boolean expressions:

```
A and B and C
A and B or not A and C
(A or B) and (not A or not B)
x > 1.0 or x <= 2
count(x <= 1.0, y <= 1.0, z <= 1.0)
```

As usual, `not` has priority on `and` which has priority on `or` and parentheses can be used to solve the ambiguities. `A and B or not A and C` reads thus `(A and B) or ((not A) and C)`.

Table D.1: Boolean operators

Operator	Number of arguments	Semantics
`and`	≥ 1	Conjunction
`or`	≥ 1	Disjunction
`not`	1	Negation
`count`	≥ 1	Number of arguments taking the value true

Table D.2: Inequalities

Operator	Number of arguments	Semantics
`==`	2	Equal
`!=`	2	Different
`<`	2	Less than
`>`	2	Greater than
`<=`	2	Less or equal
`>=`	2	Greater or equal

Table D.3: Arithmetic operators

Operator	Number of arguments	Semantics
`+`	≥ 1	Addition
`-`	2	Subtraction
`*`	≥ 1	Multiplication
`/`	2	Division
`-`	1	Unary negation
`min`	≥ 1	Minimum
`max`	≥ 1	Maximum

D.2 Inequalities

Languages of the S2ML+X family implements inequalities. Table D.2 presents inequalities. Examples of inequalities:

```
x == y
p >= 0.5
```

D.3 Arithmetic expressions

Table D.3 presents arithmetic operators. Arithmetic operators apply both with integer arguments and with real (floating point) arguments. If all arguments are integer, so is the result. If at least one of the argument is real, so is the result.

Examples of arithmetic expressions:

```
A + B + C
-A * B * C/2
min(x, y, z + 1.0)
```

As usual, priorities of operators are in order unary minus, division, multiplication, subtraction and addition, and parentheses can be used to solve ambiguities. E.g. `-A * B/2 + C - 2*D` reads `((-A) * (B/2)) + (C - (2*D))`.

In addition, languages of the S2ML+X family implements arithmetic built-ins found in most of the programming languages. These built-ins are presented Table D.4. Trigonometric functions are presented Table D.5.

Table D.4: Arithmetic built-ins

Operator	Number of arguments	Semantics
`exp`	1	Exponential
`log`	1	(Natural) logarithm
`pow`	2	Power
`sqrt`	1	Square root
`floor`	1	Largest integer under
`ceil`	1	Smallest integer above
`abs`	1	Absolute value
`mod`	2	Modulo
`Gamma`	1	Gamma function

Table D.5: Trigonometric functions

Operator	Number of arguments	Semantics
`sin`	1	Sine
`cos`	1	Cosine
`tan`	1	Tangent
`asin`	1	Arc sine
`acos`	1	Arc cosine
`atan`	1	Arc tangent

D.4 Conditional expressions and special built-ins

Languages of the S2ML+X family implements conditional (*if-then-else*) expressions and specific built-ins. Table D.6 presents them.
Examples of conditional expressions:

```
1 if state==OPEN then input else NULL
2 if failureDate<=missionTime() then FAILED else WORKING
```

D.5 Probability Distributions

Some of the languages of the S2ML+X family are dedicated to probabilistic risk assessment, i.e. eventually to the calculation of the probability that something happens at different mission-times. Models involve thus the description of probability distributions of some basic events, and combine these probabilities in various ways. These probability distributions are called failure models. They can be described by means of arithmetic expressions and the built-in `(mission-time)`. Some of them are however frequently used. It is therefore worth to implement dedicated built-ins.

There are two types of probability distributions: parametric distributions and empirical distributions. We shall review them in turn. For a presentation of their theorerical foundations, see

Table D.6: Conditional expressions

Operator	Number of arguments	Semantics
`if`	3	if *condition* then *expression* else *expression*
`missionTime`	0	Current mission time

Table D.7: Parametric probability distributions

Operator	Number of arguments	Semantics
`constantDistribution`	1	Returns the value of its parameter
`exponentialDistribution`	2	Exponential distribution with the given rate and mission time
`WeibullDistribution`	3	Weibull distribution with the given scale and shape parameters and mission time
`DiracDistribution`	2	Dirac distribution with the given delay and mission time

Appendix C.2.

Parametric probability distributions

Table D.7 presents available built-ins to describe parametric probability distributions.

Their definition is as follows.

$$\begin{aligned}
\texttt{constantDistribution}(p) &\stackrel{def}{=} p \\
\texttt{exponentialDistribution}(\lambda,t) &\stackrel{def}{=} 1-e^{-\lambda t} \\
\texttt{WeibullDistribution}(\alpha,\beta,t) &\stackrel{def}{=} 1-\exp\left(-\left(\frac{t}{\alpha}\right)^{\beta}\right) \\
\texttt{DiracDistribution}(d,t) &\stackrel{def}{=} 0 \text{ if } d<t \text{ and } 1.0 \text{ otherwise}
\end{aligned}$$

Examples of expressions with parametric probability distributions:

```
1 WeibullDistribution(alpha, beta, missionTime())
2 DiracDistribution(4000, missionTime())
```

Empirical probability distributions

Empirical distributions take two parameters: a list of points and a mission time. As for parametric distributions, the mission time is in general the built-in function `mission-time`. Lists of points have a special syntax, for the sake of convenience.

Here follows two examples using the two available empirical probability distributions:

```
1 stepwiseDistribution
2     [(0, 0), (1000, 0.2), (2000, 0.6), (2500, 1.0)]
3     (missionTime())
4 piecewiseUniformDistribution
5     [(0, 0), (1000, 0.2), (2000, 0.6), (2500, 1.0)]
6     (missionTime())
```

The list of points $(x_1,y_1),\ldots,(x_n,y_n)$ must verify the following properties.

- $n \geq 2$.
- $0 \leq x_1 < \ldots < x_n$
- $0 \leq y_1 \leq \ldots \leq y_n \leq 1.0$.

They are typically obtained via series of observations.

Table D.8: Parametric random deviates

Operator	Number of arguments	Semantics
`uniformDeviate`	2	Uniform deviate between a lower- and an upper-bound
`normalDeviate`	2	Normal deviate with the given mean and standard-deviation
`lognormalDeviate`	2	Logormal deviate with the given mean and standard-deviation
`triangularDeviate`	3	Triangular deviate with the given lower-bound, upper-bound and mode
`exponentialDeviate`	1	Exponential deviate with the given rate
`WeibullDeviate`	2	Weibull deviate with the given scale and shape parameters

The value of the stepwise distribution is calculated as follows (where x stands for the mission time, i.e. the value of the second parameter).

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases}$$

The value of the piecewise uniform distribution is calculated as follows.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i + (y_{i+1} - y_i)\frac{x - x_i}{x_{i+1} - x_i} & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases}$$

D.6 Random Deviates

Some of the languages of the S2ML+X family implements random deviates. As for probability distributions, there are two types of random deviates: parametric random deviates and empirical random deviates. We shall review them in turn. For a presentation of their theorerical foundations, see Appendix C.2.

Parametric random deviates

Table D.8 presents available parametric random deviates.
Examples of expressions with parametric random deviates:

```
1 uniformDeviate(0.0, 1.0)
2 WeibullDeviate(alpha, beta)
```

Empirical random deviates

Empirical random deviates take only one parameter: the list of points, which have a special syntax, for the sake of convenience.

Here follows two examples using the two available empirical random deviates:

```
1 stepwiseDeviate [(0, 0), (1000, 0.2), (2000, 0.6), (2500, 1.0)]
2 piecewiseUniformDeviate [(0, 0), (1000, 0.2), (2000, 0.6), (2500, 1.0)]
```

The list of points $(x_1, y_1), \ldots, (x_n, y_n)$ must verify the following properties.

– $n \geq 2$.
– $0 \leq x_1 < \ldots < x_n$
– $0 \leq y_1 \leq \ldots \leq y_n \leq 1.0$.

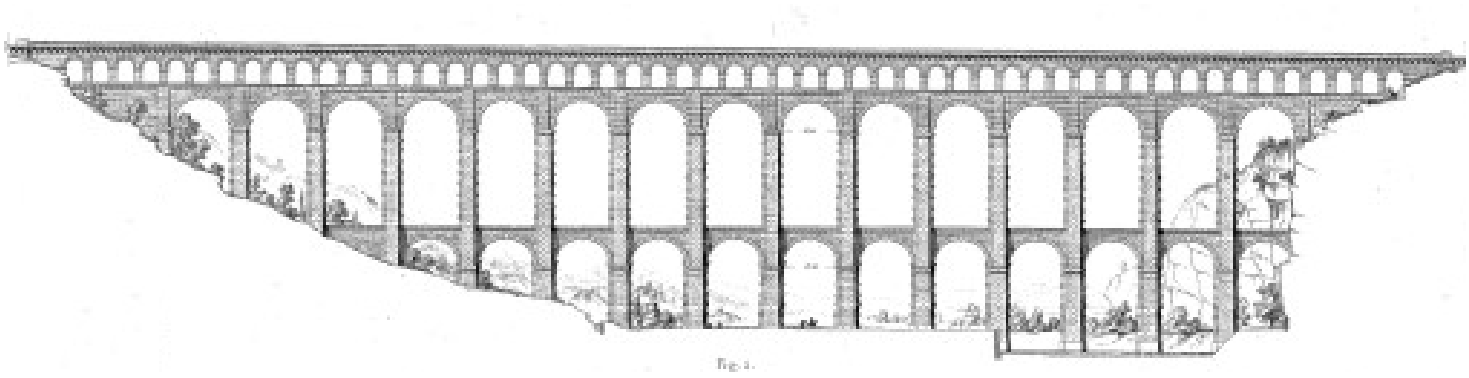
They are typically obtained via series of observations.

The value of the stepwise random deviate is calculated as follows (where y stands for the number generated at random unformly between 0 and 1).

$$F^{-1}(y) = \begin{cases} x_1 & \text{if } y < y_1 \\ x_i & \text{if } y_i \leq y < y_{i+1} \\ y_n & \text{if } y \geq y_n \end{cases}$$

The value of the piecewise uniform random deviate is calculated as follows.

$$F^{-1}(y) = \begin{cases} x_1 & \text{if } y < y_1 \\ x_i + (x_{i+1} - x_i)\frac{y - y_i}{y_{i+1} - y_i} & \text{if } y_i \leq y < y_{i+1} \\ x_n & \text{if } y \geq y_n \end{cases}$$



Index