Reliability Block Diagrams with AltaRica 3.0: part 1

A step by step introduction

The AltaRica Association

Abstract: This document shows how reliability block diagrams can be simply and efficiently assessed with AltaRica 3.0. No prior knowledge on AltaRica 3.0 is required. This document can thus be seen as a step by step introduction to AltaRica 3.0 for those who have some basic knowledge of reliability engineering modeling formalisms.
The presentation starts with the representation in AltaRica 3.0 of basic reliability block diagrams with non-repairable components. Then, it shows how they can be assessed using AltaRica 3.0 assessment tools.

Author(s)	Antoine B. Rauzy
Reviewer(s)	Tatiana Prosvirnova, André Leblond, Anthony Legendre
Reference	1
Date	01/09/2024

Table of Contents

1 Reliability Block Diagrams

1.1 Informal Presentation

It is often convenient to visualize the architecture of a system by means of diagrams made of blocks and connections between these blocks.

The reliability block diagram formalism applies this idea to probabilistic risk assessment. It is one of the fundamental formalisms in reliability engineering, see e.g. (Andrews and Moss 2002) for a reference textbook. A reliability block diagram describes thus a network of connected blocks. A flow circulates in the network starting from a source node, propagating through blocks and ending in a, usually unique, target node. Each block may have one or more failure modes that prevent the flow to go from its input ports to its output ports. The system described by the network is working if there is at least one operating path from one of the source nodes to the target node.

Figure 1 shows a reliability block diagram. This model is made of one source node `S`, one target node `T`, and eight blocks `B1`,..., `B8`.

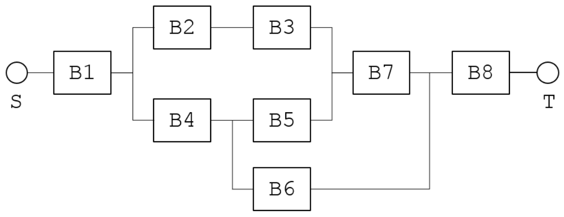

Figure 1: A reliability block diagram

Possible `S-T` paths are `S-B1-B2-B3-B7-B8-T`, `S-B1-B4-B5-B7-B8-T` and `S-B1-B4-B6-B8-T`.

For now, we shall assume that each block `B`i, $1 \leq i \leq 8$, has only one failure mode simply called `failure` and that failures of basic blocks are statistically independent. Moreover we shall assume that the source and target nodes are perfectly reliable.

1.2 Formal Description

In order to model reliability block diagrams as stochastic discrete event systems, we have thus to:

– Describe the possible states of basic blocks and how basic blocks change of state under the occurrence of events (failures).

– Describe the propagation of the flow through the network.

– Associate probabilities or probability distributions to the failures of basic blocks.

According to our assumptions, basic blocks can be only in two states: `WORKING` or `FAILED`. Initially they are in the state `WORKING` and change to state `FAILED` when the event `failure` occurs. For the sake of the simplicity, we can assume moreover that their failures are exponentially distributed. The failure rate may depend on the basic block.

Figure 2 shows a graphical representation of this behavior by means of a so-called finite state automaton.

The automaton is made of two states (represented by ovals) and a transition from the state `WORKING` to the state `FAILED` (represented by an arrow). The transition is labeled with the event name, i.e. `failure`.

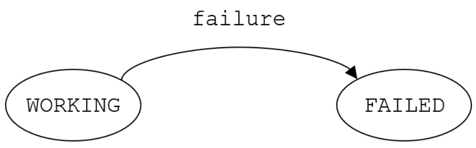

Figure 2: The finite state automaton representing a basic block

Each of the basic blocks can be represented by such a finite state automaton. The global behavior of the system can also be represented by a finite state automaton. This automaton is the "product" of the automata representing basic blocks. The states of the product automaton are vectors of states of basic blocks. In our example, these vectors involve thus eight values in $\{\texttt{WORKING}, \texttt{FAILED}\}$ as there are eight basic blocks. As failures of basic blocks are assumed to be statistically independent, the probability that two of them occur exactly at the same time is null. Therefore, transitions of the product automaton are just failures of basic blocks. They change one component of the vector at a time. For instance, in the initial system state (assuming states of basic blocks are given in lexicographic order):

$$\langle \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING} \rangle$$

The failure of the basic block `B7` changes the system state to:

$$\langle \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{WORKING}, \texttt{FAILED}, \texttt{WORKING} \rangle$$

It would be indeed tedious and error prone to describe the product automaton by hand. In our small example, as there are eight basic blocks, this automaton has actually $2^8 = 256$ states and $256 \times 4 = 1024$ transitions (4 transitions per state on average). In a industrial size model, with dozens if not hundred of blocks, the number of states is so large that enumerating them is just impossible. Fortunately, as we shall see in the next section, AltaRica makes it possible to describe implicitly these states, just by describing the behavior of each basic block and then assembling these individual behaviors.

Regarding flow propagation, it is ruled by two types of equations: equations to describe how the output flow of a basic block depends on its input flow and its internal state on the one hand, equations to describe how basic blocks are connected on the other hand.

There is a flow going out a basic block, if there is a flow coming in and it is in the state `WORKING`. We can thus write for each block $\texttt{B}i$, $1 \leq i \leq 8$, a Boolean equation of the following form.

$$\texttt{B}i\texttt{.out} \;\; = \;\; \texttt{B}i\texttt{.in and } (\texttt{B}i\texttt{.state} = \texttt{WORKING})$$

Where $\texttt{B}i\texttt{.in}$, $\texttt{B}i\texttt{.out}$ and $\texttt{B}i\texttt{.state}$ represent respectively the input flow, the output flow and the state of the component $\texttt{B}i$.

Connections between basic blocks can be described in similar way but involve only input and output flows of the blocks. Table 1 gives these equations for our example.

2 AltaRica 3.0 Model

2.1 Model for Basic Blocks

We need first to encode a finite state automaton representing basic blocks. Figure 3 gives thus the AltaRica 3.0 code for basic blocks.

Table 1: Equations describing connections between basic blocks.

$$\begin{aligned}
\texttt{T} &= \texttt{B8.out} \\
\texttt{B8.in} &= \texttt{B6.out or B7.out} \\
\texttt{B7.in} &= \texttt{B3.out or B5.out} \\
\texttt{B6.in} &= \texttt{B4.out} \\
\texttt{B5.in} &= \texttt{B4.out} \\
\texttt{B4.in} &= \texttt{B1.out} \\
\texttt{B3.in} &= \texttt{B2.out} \\
\texttt{B2.in} &= \texttt{B1.out} \\
\texttt{B1.in} &= \texttt{S} \\
\texttt{S} &= \texttt{true}
\end{aligned}$$

```
1  domain State {WORKING, FAILED}
2
3  class BasicBlock
4      State _state (init = WORKING);
5      Boolean in (reset = false);
6      Boolean out (reset = false);
7      event failure (delay = exponential(lambda));
8      parameter Real lambda = 1.0e-3;
9      transition
10         failure: _state==WORKING -> _state := FAILED;
11     assertion
12         out := in and _state==WORKING;
13 end
```

Figure 3: AltaRica 3.0 model for basic blocks

This listing starts with a declaration of domain (line 1). A *domain* is a set of symbolic constants. Here the domain is named `State` and consists of two symbolic constants: `WORKING` and `FAILED`.

Basic blocks are described by means of the class `BasicBlock` (lines 3–13). AltaRica 3.0 is an object oriented modeling language. A *class* is a reusable modeling component.

The class `BasicBlock` declares first a variable named `_state` (line 3). This variable takes its value into the domain `State` declared above. The initial value of the variable `_state`, in this case `WORKING`, is declared via the attribute `init`. An *attribute* is a quantity with a name associated with an element (variable, event...). Attributes are declared after the name of the element inside parentheses.

Then, the class `BasicBlock` declares two Boolean variables (i.e. variables taking their value into the set {`false`, `true`}) `in` and `out`. The default values of these two variables, in this case `false`, are declared via the attribute `reset`.

`_state` is a *state variable*. The values of state variables can only change when a transition is fired. `in` and `out` are *flow variables*. The values of flow variables depend on the values of state variables. They are updated after each transition firing via the assertion (see below).

Identifiers of classes, variables and other named elements obey the same rule as in most of programming and modeling languages: they must be a letter or an underscore character followed by any sequence of letters, digits and underscore characters. Upper and lower case letters are considered as different. As a personal coding style, we use to distinguish syntactically state variables and flow variables by prefixing the former by an underscore letter. This is indeed not required by the grammar of AltaRica 3.0.

After the variables, the class `BasicBlock` declares the *event* `failure` (line 7) and associates it with a negative exponential probability distribution by means of the attribute `delay`. This distribution takes a parameter `lambda` (the failure rate).

Finally, the class `BasicBlock` declares the parameter `lambda` (line 8), which in this case is real-valued, and gives it a default value, in this case (10^{-3}). A *parameter* is an invariable quantity associated with a class. Its value can be set up when the class is instantiated or just kept to its default value.

After declarations of elements like variables, events and parameters, a class usually declares one or more transitions and an assertion.

Declaration of transitions are preceded by the keyword `transition`. The class `BasicBlock` declares only one transition (line 10). A *transition* is made of three elements:

– The event that labels the transition (in our example `failure`).

– The *guard* of the transition which is a Boolean condition over state and flow variables. The transition is enabled in a given state, i.e. in a given valuation of state and flow variables, if its guard is satisfied. In our example the transition is enabled when the variable `_state` has the value `WORKING`.

– The *action* of the transition which is made of one or more *instructions* that modify the values of state variables. In our example, the action is made of only one instruction setting the variable `_state` to the value `FAILED`.

Regular transitions obey the syntax *event*`:`*guard*`->`*action*. Variable assignments have the form *variable*`:=`*expression*`;`.

Finally, the class `BasicBlock` declares the assertion (line 12) which is preceded by the keyword `assertion`. An *assertion* is made of one or more instructions that set up the values of flow variables according to the values of state variables.

In our example, there is only one instruction, setting up the value of the flow variable `out`. This value depends on the value of the state variable `_state` but also on the value of the flow variable `in`. If the class `BasicBlock` is considered in isolation, the flow variable `in` has no value or more exactly its value is constantly equal to its default value `false`. The situation will change when the class is instantiated and basic blocks connected by means of equations similar to those of Table 1.

2.2 Model for the whole Diagram

To get a model for the reliability block diagram as a whole, we need:

– To instantiate the class `BasicBlock` as many times as there are basic blocks;

– To connect instances so to describe the network topology;

– To prepare the calculation of risk indicators.

Figure 4 shows the AltaRica 3.0 code for the network of components. Put together with the code of Figure 3, it gives the complete model for our example.

The code given Figure 4 declares a block. A block is a *prototype* in the sense of object-oriented theory (Abadi and Cardelli 1998), i.e. it is modeling component with a unique occurrence. The

```
 1 block Plant
 2     Boolean S (reset = false);
 3     Boolean T (reset = false);
 4     BasicBlock B1(lambda = 1.0e-6);
 5     BasicBlock B2(lambda = 1.0e-4);
 6     BasicBlock B3(lambda = 1.0e-4);
 7     BasicBlock B4(lambda = 1.0e-4);
 8     BasicBlock B5(lambda = 1.0e-4);
 9     BasicBlock B6(lambda = 1.0e-5);
10     BasicBlock B7(lambda = 1.0e-6);
11     BasicBlock B8(lambda = 1.0e-6);
12     assertion
13         T := B8.out;
14         B8.in := B6.out or B7.out;
15         B7.in := B3.out or B5.out;
16         B6.in := B4.out;
17         B5.in := B4.out;
18         B4.in := B1.out;
19         B3.in := B2.out;
20         B2.in := B1.out;
21         B1.in := S;
22         S := true;
23     observer Boolean failed = not T;
24 end
```

Figure 4: AltaRica 3.0 model for the reliability block diagram

model for the reliability block diagram as a whole could have been declared as a class as well, but it is better to declare it as a prototype since it will not be reused. The structure of a prototype is essentially the same as the one of a class. It starts with variables, events, parameters and sub-components declarations. Then come transitions and the assertion. There are differences between classes and prototypes, but this goes beyond the scope of this study.

In our example, the prototype is named `Plant`.

It starts by declaring two Boolean flow variables `S` and `T` to represent source and target nodes of the network (lines 2 and 3). Then, the class `BasicBlock` is instantiated eight times, one per basic block (lines 4–11). Each instance declares its own value of the failure rate `lambda`.

The prototype declares no event (and thus no transition) as failures of basic blocks are statistically independent.

The assertion just implements the equations of Table 1 that connect together basic blocks.

Finally, the prototype declares the Boolean observer `failed` (line 23). An *observer* is very much like a flow variable except that its value cannot be used in transitions or assertions. Observers are used to declare quantities to be measured by assessment tools. In our example, the only quantity we are interested in is the absence of flow in the target node `T`. But of course, additional observers could be defined.

Appendix A gives some hints on how to author this model using AltaRica Wizard.

3 Assessment

The model designed in the previous section can be assessed by compiling it into Boolean equations (fault trees).

3.1 Compilation into Boolean equations

The compiler to Boolean equations (fault trees) produces a model at Open-PSA format (Epstein and Rauzy 2008) so that it can be assessed with the XFTA calculation engine (Rauzy 2020). XFTA produces the minimal cutsets given in Table 2.

Table 2: Minimal cutsets calculated from the generated Boolean equations (fault tree).

B3.failure	B4.failure	
B2.failure	B4.failure	
B3.failure	B5.failure	B6.failure
B2.failure	B5.failure	B6.failure
B8.failure		
B1.failure		
B4.failure	B7.failure	
B6.failure	B7.failure	

Figure 5 shows the evolution of the unreliability of the system, as calculated by XFTA from the generated Boolean equations (fault tree).

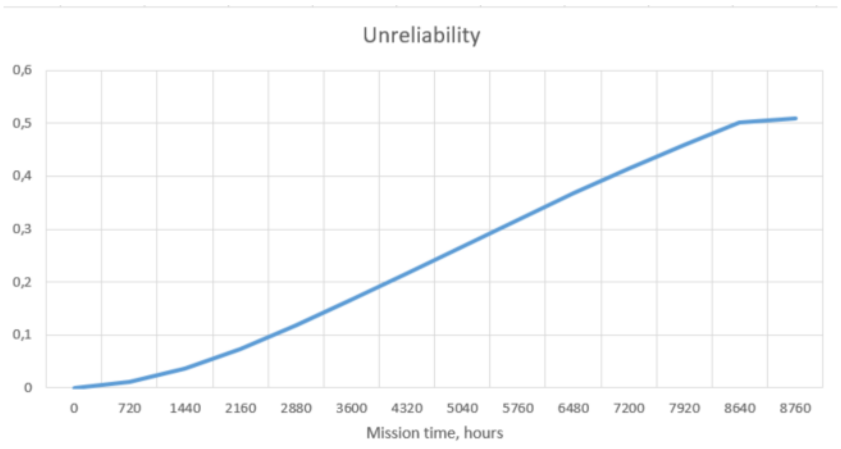

Figure 5: Unreliability calculated from the generated Boolean equations (fault tree).

4 References

Abadi, Mauricio and Luca Cardelli (1998). *A Theory of Objects*. New-York, USA: Springer-Verlag. isbn: 978-0387947754 (cited on page 7).

Andrews, John D. and Robert T. Moss (2002). *Reliability and Risk Assessment (second edition)*. Materials Park, Ohio 44073-0002, USA: ASM International. isbn: 978-0791801833 (cited on page 4).

Epstein, Steven and Antoine Rauzy (2008). *Open-PSA Model Exchange format, version 2.0d*. available at www.altarica-association.org (cited on page 9).

Rauzy, Antoine (2020). *Probabilistic Safety Analysis with XFTA*. Les Essarts le Roi, France: AltaRica Association. isbn: 978-82-692273-0-7 (cited on page 9).

A Authoring Models with AltaRica 3.0 Workshop

AltaRica 3.0 Workshop is an integrated modeling environment dedicated to probabilistic risk and safety analyses of complex technical systems. It supports two modeling languages: AltaRica 3.0 and S2ML+SBE. Both languages belong to the S2ML+X family.

AltaRica 3.0 Workshop contains:

– AltaRica Wizard, a graphical interface for authoring AltaRica 3.0 and S2ML + SBE models.

– A complete set of assessment tools:

  – An interactive simulator;

  – A generator of stochastic Boolean equations;

  – A stochastic simulator;

  – A generator of critical sequences;

  – A calculation engine for stochastic Boolean equations XFTA.

This section explains how to author models with AltaRica Wizard.

A.1 Projects

AltaRica Wizard manages models through projects. A *project* contains one or more files organized into folders. Files of a project can be AltaRica source files (having usually the extension "`.alt`"), as well as other types of files generated by assessment tools.

Projects can be created, populated with folders and files, saved and (re)loaded from AltaRica Wizard. They are described into XML files with the extension "`.ar3w`". Project description files can thus be edited manually, although this should be done with care.

Files and folders of a project are those of the underlying operating system. In particular, the project itself is located into a folder of the underlying operating system, the *project folder*. All paths to folders and files are considered relatively to the project folder and the file describing the project is saved into this folder. This is the reason why it is highly recommended to organize all of the files of a project under the project folder. In this way, the project can be moved from one place to another one just by moving the project folder. The same principle applies indeed to project copy: to duplicate a project it suffices to copy the folder that contains this project.

Folders are automatically added to and removed from projects when adding and removing files. This means that a project contains a folder if and only if this folder contains a file of the project (possibly located in a sub-folder, a sub-sub-folder, . . . of that folder). *A contrario*, sub-folders of the project folder do not belong automatically to the project. Only those containing a file of the project do.

Creating a new project, opening an existing one and saving the current project is done via the menu "File". Creating a new project implies selecting a folder for that project. By default, AltaRica Wizard creates this folder for you. Nevertheless, it is possible to create a project without creating a new folder.

A.2 Source Files

Once the project created AltaRica source files can be added to this project. AltaRica source files must have the extension ".alt".

It is possible to create AltaRica source files directly via the menu “Project”. Another solution consists in creating AltaRica source files by means of an external text editor and then to add them to the project, also via the menu “Project”.

When tools are launched, all AltaRica source files of the project are gathered. The model is thus made of all domains, blocks and classes declared in AltaRica source files.

The AltaRica step by step series

AltaRica 3.0 is an object-oriented modeling language dedicated to performance analyses of complex technical systems. It makes it possible to design highly reusable stochastic discrete event models. The AltaRica step by step series aims at showing how different types of systems can be represented in AltaRica 3.0 and how their performance—to be taken in a broad sense—can be assessed by means of AltaRica 3.0 assessment tools.

Each document of the series is a step by step introduction to AltaRica 3.0. We tried to minimize the number of pre-requisites so that all engineers or students in engineering disciplines can take benefit of these presentations.

Beyond being a simple collection of smooth introductions to AltaRica 3.0, the AltaRica step by step series aims at gathering a versatile set of reusable modeling patterns. Using these patterns makes the learning curve steeper and the modeling process more efficient.

AltaRica 3.0 results from two decades of active academic research and industrial experience. It comes with AltaRica Wizard, an integrated modeling and simulation environment. AltaRica Wizard is coupled with a complete set of assessment tools enabling a wide variety of analyses. The specification of the language and the integrated modeling and simulation environment are developed by the not for profit AltaRica Association. They can be used free of charge for research and education purposes. The AltaRica project is supported by several academic institutions and world leading industrial partners.

Already published in AltaRica step by step series

TRAS-2024-001: Reliability Block Diagrams with AltaRica 3.0: part 1