HAL
archives-ouvertes.fr

Advances in the simplification of Fault Trees automatically generated from AltaRica 3.0 models

Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy

▶ To cite this version:

Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy. Advances in the simplification of Fault Trees automatically generated from AltaRica 3.0 models. European Safety and Reliability Conference (ESREL 2018), Jun 2018, Trondheim, Norway. pp.907-914. hal-01826645

HAL Id: hal-01826645
https://hal.archives-ouvertes.fr/hal-01826645

Submitted on 29 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Advances in the simplification of Fault Trees automatically generated from AltaRica 3.0 models

M. Batteux
IRT SystemX
Palaiseau, France

T. Prosvirnova
LGI
CentraleSupelec, Gif-sur-Yvette, France

A. Rauzy
MTP
Norwegian University of Science and Technology, Trondheim, Norway.

ABSTRACT: Safety and risk analyses rely on models. These models have several important characteristics. They are event-oriented. The system under study changes of state when events, such as failure, hazard, repair and so on, occur. They are probabilistic. The exact moment of the occurrence of a failure is in essence unpredictable. They are discrete. States are represented by means of variables that take their values into finite, usually very small, domains. The most widely used modeling formalisms such as Fault Trees, Block Diagrams and Event Trees rely on Boolean algebra. There are cases however where binary states are not sufficient. For instance, it is sometimes necessary to represent the level of degradation of a component, the quality of a signal, and so on. This kind of models can be easily represented with AltaRica 3.0 - a high level modeling language dedicated to safety analyses. AltaRica 3.0 is at the core of the OpenAltaRica project which aim is to develop a complete set of assessment tools for the language, including among others compilers to Fault Trees and Markov Chains, stochastic and stepwise simulators. In this article we study how the notion of prime implicants can be extended to finite domain calculus. We discuss the efficient implementation of finite domain calculus and show how these results can be applied to simplify Fault Trees, automatically generated from AltaRica 3.0 models. This simplification in its turn significantly improves the efficiency of the assessment of the automatically generated Fault Trees.

1 INTRODUCTION

Risk analysis relies on models. These models have several important characteristics:

- They are event-oriented. The system under study changes of state when events, such as failure, hazard, repair and so on, occur.
- They are probabilistic. The exact moment of the occurrence of a failure is in essence unpredictable.
- They are discrete. States are represented by means of variables that take their values into finite, usually very small, domains.

The last characteristic is pragmatic: given the difficulty to design models and computational complexity of the calculation of indicators, discrete abstractions are a necessary tradeoff. Hence the role of Boolean algebra in Reliability, Availability, Maintainability, Safety engineering. The most widely used modeling formalisms such as Fault Trees, Block Diagrams and Event Trees rely on Boolean algebra. There are cases however where binary states are not sufficient. For instance, it is sometimes necessary to represent the level of degradation of a component, the quality of a signal, and so on. This kind of models can be easily represented with AltaRica 3.0 - a high level modeling language dedicated to safety analyses (Prosvirnova, Batteux, Brameret, Cherfi, Friedlhuber, Roussel, & Rauzy 2013). AltaRica 3.0 is at the core of the OpenAltaRica project[1] which aim is to develop a complete set of assessment tools for

[1] See https://www.openaltarica.fr

the language, including among others compilers to Fault Trees (Prosvirnova & Rauzy 2015) and Markov Chains (Brameret, Rauzy, & Roussel 2015), stochastic and stepwise simulators (Aupetit, Batteux, Rauzy, & Roussel 2015).

In this article we study how the notion of prime implicants can be extended to finite domain calculus and how to encode it efficiently. The contribution of this article is thus multiple. First, we present how the notion of prime implicants can be extended to finite domain calculus. Second, we discuss the efficient implementation of finite domain calculus. Finally we show how these results can be applied to simplify Fault Trees, automatically generated from AltaRica 3.0 models.

The remainder of this article is organized as follows. Section 2 describes a motivating example. Section 3 presents a theoretical work about finite domain calculus and discusses its implementation. Section 4 shows the application of the finite domain calculus to the simplification of Fault Trees automatically generated from AltaRica 3.0 models. Section 5 presents some experimental results using the motivating example. Section 6 concludes this article.

2 MOTIVATING EXAMPLE

Consider a parametric block diagram use case (see Figure 1) with three parameters:

- s the number of blocks in series;
- p the number of parallel blocks;
- q the level of recursivity (depth).

These relatively simple but large safety models can be easily represented in AltaRica 3.0 and handled simply and efficiently by means of the Fault Tree compilation tool chain.

Note that without loosing the efficiency of the assessment, in AltaRica 3.0, it is possible to represent multi-state blocks, e.g. consider the quality of data with the values `ok`, `lost` or `erroneous`, or the level of degradation with the values `ok`, `degraded` or `failed`.

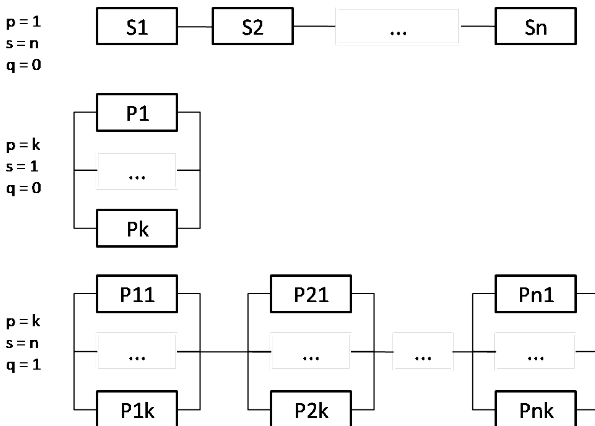

Figure 1: Parametric block diagram use case

This use case is both representative of a class of industrial models and parametric to show the scalability of the approach. We shall use it throughout the article to illustrate the advances in the simplification of Fault Trees.

3 FINITE DOMAIN CALCULUS

3.1 *Definitions*

Let $\Xi = \{X_1, X_2, \ldots, X_n\}$ be a finite set of variables. Each X_i takes its values into a finite domain (a finite set of constants) denoted as $dom(X_i)$. The set of well formed formulas over Ξ is the smallest set such that:

- The two Boolean constants 0 (false) and 1 (true) are formulas.
- If X is a variable and c is a constant then $X = c$ is a formula. Such a formula $X = c$ is called a **literal** and makes only sense if $c \in dom(X)$.
- If f and g are formulas, then so are $f + g$ (disjunction), $f * g$ (conjunction), and $-f$ (negation).

We assume that the negation ($-$) has a higher priority than the conjunction ($*$), which has a higher priority than the disjunction ($+$).

A **product** is a set of literals interpreted as the conjunction of its elements. A product is said **fundamental** if it does not contain two literals built over the same variable. We shall consider only fundamental products. The empty product is denoted **1**.

A **minterm** is a product that contains a literal for each variable of Ξ. As we shall see, minterms play a fundamental role in the finite domain calculus for they are the atoms of the underlying Boolean algebra.

A **sum of products** is a set of products interpreted as the disjunction of its elements. The empty sum of products is denoted **0**.

A **variable assignment** of Ξ is a function $\sigma : \Xi \rightarrow dom(X_1) \times dom(X_2) \times \ldots \times dom(X_n)$, that associates to each variable X_i its value from $dom(X_i)$, $i = 1, ..., n$.

Let f and g be formulas and σ be a variable assignment over Ξ. The value of $\sigma(f)$ is calculated recursively as follows:

- $\sigma(1) = 1, \sigma(0) = 0$;
- $\sigma(X = c) = 1$ if $\sigma(X) = c$ and 0 otherwise;
- $\sigma(f + g) = max(\sigma(f), \sigma(g))$, $\sigma(f * g) = min(\sigma(f), \sigma(g))$, $\sigma(-f) = 1 - \sigma(f)$.

The variable assignment σ **satisfies** the formula f, if $\sigma(f) = 1$, otherwise is **falsifies** it.

There is a one to one correspondence between minterms and variable assignments: the minterm p corresponds to the variable assignment σ if for each variable $X \in \Xi$, $X = c \in p$ if and only if $\sigma(X) = c$.

3.2 *Implication, Equivalence, Properties*

Let $\Xi = \{X_1, \ldots, X_n\}$ be a finite set of finite domain variables. Let f and g be two formulas built over Ξ. f **implies** g, which is denoted as $f \Rightarrow g$, if any variable assignment that satisfies f satisfies g as well. f is **equivalent** to g, which we denote as $f \Leftrightarrow g$, if both $f \Rightarrow g$ and $g \Rightarrow f$.

The usual properties of Boolean algebras hold for the finite domain calculus:

Neutral element: $f + 0 \Leftrightarrow 0 + f \Leftrightarrow f$ and $f * 1 \Leftrightarrow 1 * f \Leftrightarrow f$

Absorbing element: $f + 1 \Leftrightarrow 1 + f \Leftrightarrow 1$ and $f * 0 \Leftrightarrow 0 * f \Leftrightarrow 0$

Idempotence: $f + f \Leftrightarrow f$ and $f * f \Leftrightarrow f$

Commutativity: $f + g \Leftrightarrow g + f$ and $f * g \Leftrightarrow g * f$

Associativity: $f + (g + h) \Leftrightarrow (f + g) + h$ and $f * (g * h) \Leftrightarrow (f * g) * h$

Distributivity: $f + (g * h) \Leftrightarrow f * g + f * h$ and $f * (g + h) \Leftrightarrow (f + g) * (f + h)$

Double negation: $- - f \Leftrightarrow f$

de Morgan's law: $-(f + g) \Leftrightarrow -f * -g$ and $-(f * g) \Leftrightarrow -f * -g$

3.3 *Negation*

The real difference between the propositional and finite domain calculi stands in the negation.

Let Ξ be a finite set of variables, let X be a variable from Ξ, and finally let c be a constant of $dom(X)$. Then, $-(X = c) \Leftrightarrow \Sigma_{d \in dom(X), d \neq c}(X = d)$

Theorem 1 (Elimination of negations)**.** *: For any formula of the finite domain calculus, there exists an equivalent formula involving no negation.*

Note that any formula is equivalent to the sum of minterms that satisfies it, which is a first way to demonstrate the theorem. A more syntactic proof consists in pushing negations down to literals, thanks to de Morgan's law, and then to transform negative literals as shown above.

3.4 *Subsumption, Resolution*

Let p and q be two products built over Ξ. We say that p **subsumes** q if $q \Rightarrow p$, i.e. if and only if any literal of p is also a literal of q.If p subsumes q, then $p + q \Leftrightarrow p$.

Let X be a variable of Ξ, let $dom(X) = \{c_1, c_2, \ldots, c_k\}$ and let $p_1, \ldots, p_k$ be k products in which X does not show up. Let r be the product $p_1 * p_2 * \ldots * p_k$. Then the following implication holds: $(X = c_1) * p_1 + \ldots + (X = c_k) * p_k \Rightarrow r$

The product r is called the **resolvent** of the products $(X = c_1) * p_1, \ldots, (X = c_k) * p_k$.

In the case, where there is a product p_j such that $p_j = r$, then the following equivalent holds:

$$\begin{aligned}
&(X = c_1) * p_1 + \ldots + (X = c_k) * p_k \\
\Leftrightarrow\ &(X = c_1) * p_1 + \ldots + (X = c_j) * p_j \\
&\quad + \ldots + (X = c_k) * p_k + r \\
\Leftrightarrow\ &(X = c_1) * p_1 + \ldots + \cancel{(X = c_j) * p_j} \\
&\quad + \ldots + (X = c_k) * p_k + r
\end{aligned}$$

3.5 *Shannon Normal Form*

Let X be a variable of Ξ, let c be a constant of $dom(X)$ and finally let f be a formula built over Ξ. There exist two formulas f_1 and f_0 in which the atom $(X = c)$ does not show up such that:
$f \Leftrightarrow (X = c) * f_1 + f_0$

The above representation is called the **pivotal decomposition** of f with the respect to X and c.

Assume we are given an (arbitrary) order $<$ over the variables of Ξ and over the constants of the domain of the variables of Ξ. The set of formulas in **Shannon Normal Form** is defined inductively as follows:

- The two constants 0 and 1 are in Shannon Normal Form.
- If f and g are two formulas in Shannon Normal Form, X is a variable and c is a constant of $dom(X)$, the formula $(X = c) * f + g$ is in Shannon Normal Form if
  - X does not show up in f, and
  - for all literal $(Y = d)$ showing up in g, either $X < Y$ or $X = Y$ and $c < d$.

3.6 *Representation Theorem*

Let Ξ be a finite set of finite domain variables. Let X be a variable of Ξ, let c be a constant of $dom(X)$ and finally let $f = (X = c) * f_1 + f_0$ be a formula in Shannon Normal Form built over Ξ.

In the above representation we can assume without a loss of generality that:

- $f_1 \neq 0$ as $(X = c) * 0 + f_0 \Leftrightarrow f_0$
- $f_0 \neq 1$ as $(X = c) * f_1 + 1 \Leftrightarrow 1$

From now, we shall assume that these two **simplification rules** are systematically applied.

Theorem 2 (Representation)**.** *: for any formula of the finite domain calculus, there exists at least one equivalent formula in Shannon Normal Form.*

In general, this equivalent formula is not unique. We shall see that two of the formulas that represent a given sum of products are of special interest: the first one can be interpreted as sum of disjoint products, the other one as the set of prime implicants. These two formulas are extremum in a sense we shall explain.

A formula in Shannon Normal Form can be interpreted as a sum of products. Namely,

- SumOfProducts[0] = 0;

- SumOfProducts[1] = 1;
- SumOfProducts[$(X = c) * f + g$] = $\{(X = c) * p; p \in SumOfProducts[f]\}$ $\cup\, SumOfProducts[g]$

Theorem 3 (Sums-of-Products)**.** *: Shannon Normal Formulas one-to-one correspond with Sums-of-Products (for a given order of variables and constants).*

3.7 *Factors and cofactors*

The **factor** and **cofactor** of a formula f with respect to a variable X, denoted respectively as $f|X$ and $f \sim X$, are syntactic operations that select respectively the products of f that contain X and the products of f that do not contain X. The **factor** $f|X$ is defined recursively as follows:

- $0|X = 0$ and $1|X = 1$
- $[(X = c) * f + g]|X = (X = c) * f + [g|X]$
- $[(Y = c) * f + g]|X = 0$ if $X < Y$
- $[(Y = c) * f + g]|X = (Y = c) * [f|X] + [g|X]$ if $X > Y$

The **cofactor** $f \sim X$ is defined recursively as follows:

- $0 \sim X = 0$ and $1 \sim X = 1$
- $[(X = c) * f + g] \sim X = g \sim X$
- $[(Y = c) * f + g] \sim X = g$ if $X < Y$
- $[(Y = c) * f + g] \sim X = (Y = c) * [f \sim X] + [g \sim X]$ if $X > Y$

3.8 *Logical operations*

Let Ξ be a finite set of finite domain variables. Let X and Y be two variables of Ξ with $X < Y$. Let c, d and e be three constants such that $c, d \in dom(X)$ with $c < d$, and $e \in dom(Y)$. Finally let $f = (X = c) * f_1 + f_0$, $g = (X = c) * g_1 + g_0$, $h = (X = d) * h_1 + h_0$ and $I = (Y = e) * I_1 + I_0$ be four formulas built over Ξ in Shannon Normal Form. The following equivalences hold and they are used as recursive equations to perform logical operations on formulae in Shannon Normal Form:

- $f + g \Leftrightarrow (X = c) * [f_1 + g_1] + [f_0 + g_0]$
- $f + h \Leftrightarrow (X = c) * f_1 + [f_0 + h]$
- $f + I \Leftrightarrow (X = c) * f_1 + [f_0 + I]$
- $f * g \Leftrightarrow (X = c) * [f_1 * g_1 + f_1 * g_0 \sim X + f_0 \sim X * g_1] + [f_0 * g_0]$
- $f * h \Leftrightarrow (X = c) * [f_1 * h1 + f_1 * g_0 \sim X] + [f_0 * g]$
- $f * I \Leftrightarrow (X = c) * f_1 + [f_0 * I]$
- $-f \Leftrightarrow [\Sigma_{d \in Dom(X), d \neq c}(X = d) * -g] + [-f * -g]$

3.9 *Subsumption*

As we shall see, it is of interest to remove from a formula f all the products that are subsumed by a product of a formula g. This operation, denoted $f \div g$, can be defined by means of the following recursive equations. Let Ξ be a finite set of finite domain variables. Let X and Y be two variables of Ξ ($X < Y$), let c, d and e be three constants such that $c, d \in dom(X)$, $c < d$, and $e \in dom(Y)$. Then:

- $f \div 0 = f$, $f \div 1 = 0$, $0 \div g = 0$ and $1 \div g = 1$
- $[(X = c) * f_1 + f_0] \div [(X = c) * g_1 + g_0]$ $= (X = c) * [(f_1 \div g_1) \div g_0] + f_0 \div g_0$
- $[(X = c) * f_1 + f_0] \div [(X = d) * g_1 + g_0]$ $= (X = c)[f_1 \div g_0] + f_0 \div g_0$
- $[(X = c) * f_1 + f_0] \div [(Y = e) * g_1 + g_0]$ $= (X = c)[f_1 \div [(Y = e) * g_1 + g_0]]$ $+ f_0 \div [(Y = e) * g_1 + g_0]$
- $[(X = d) * f_1 + f_0] \div [(X = c) * g_1 + g_0]$ $= (X = c)[f_1 \div g_0] + f_0 \div g0$
- $[(Y = e) * f_1 + f_0] \div [(X = c) * g_1 + g_0]$ $= [(Y = e) * f_1 + f_0] \div g_0$

3.10 *Prime implicants*

Let Ξ be a finite set of finite domain variables with an order over variables and constants. Let f and p be respectively a formula and a product built over Ξ.

- p is an **implicant** of f if $p \Rightarrow f$.
- p is a **prime implicant** of f if it is an implicant of f and no strict sub-product (subsuming product) of p is.

The set of prime implicants of f is denoted $PI[f]$.

Theorem 4 (Decomposition of Prime Implicants)**.** *: Let f be a formula in Shannon Normal Form. Then $f = (X = c_1) * f_1 + ((X = c_2) * f_2 + \ldots + ((X = c_k) * f_k + f_0)) \ldots)$ for some constants $c_1, \ldots, c_k$ from $dom(X)$ and some formulas $f_1, \ldots, f_k, f_0$ in Shannon Normal Form in which X does not occur.*

Let $h = (f_1 * f_2 * \ldots * f_k) + f_0$. Then, the set of prime implicants of f denoted by $PI[f]$ are calculated as follows:

$$\begin{aligned} PI[f] = \{&(X = c_1) * p; p \in PI[f_i] \div PI[h]\} \\ &\ldots \\ &\cup\{(X = c_k) * p; p \in PI[f_k] \div PI[h]\} \\ &\cup\{PI[h]\} \end{aligned}$$

The decomposition theorem gives an algorithm to calculate for any formula f in Shannon Normal Form an equivalent formula h such that:
$g = SumOfProducts[h] = PI[f]$

Because all possible resolutions and subsumptions have been performed, g can be considered as the **most simplified form** of f.

At the opposite, we may want to transform f into an equivalent **sum of disjoint products** so to be able to calculate the exact probability of f. Disjoining products encoded by f is performed by the dual operation of calculating resolvents.

Let $f = (X = c_1) * f_1 + ((X = c_2) * f_2 + \ldots + ((X = c_k) * f_k + f_0))\ldots)$. Assume that $dom(X) = \{c_1, \ldots, c_k\}$ (if some constant c_i of $dom(X)$ is missing we can always add the term $(X = c_i) * 0$). Then, f is equivalent to the following formula:
$g = (X = c_1) * [f + f_0] + ((X = c_2) * [f_2 + f_0] + \ldots + ((X = c_k) * [f_k + f_0)] + 0)\ldots)$.

By applying this transformation recursively, we get a sum of disjoint products, which is also **unique**, for a given order on variables and constants.

3.11 *Diagrammatic Representation*

The idea is to represent sums of products in Shannon Normal Form by means of a variant of Bryant's **Binary Decision Diagrams** (Bryant 1992). The idea is therefore to represent formulas in Shannon Normal Form by means of Directed Acyclic Graphs with two types of nodes:

- Leaves, that are labeled with either 0 or 1.
- Internal nodes, that are labeled with a variable X and a constant c of $dom(X)$ and that have two out-edges called the 1-outedge and the 0-outedge. Such a node represents the formula $(X = c) * f + g$, where f and g are the formulas represented respectively by the node pointed by the 1-outedge and the node pointed by the 0-outedge.

The **Shannon Diagram** representing a formula is always built **bottom-up**. Nodes are maintained into a **unique table** (and accessed by means of a **hash-table**). In this way, for any formula f, there is at most one node representing f in the table. Checking the equivalence of two formulas is thus performed in constant time once their Shannon Diagrams are built.

4 APPLICATION

One of the possible applications of the finite domain calculus presented above is the simplification of Fault Trees automatically generated from AltaRica 3.0 models. AltaRica 3.0 is an event-based high level modeling language dedicated to Safety Analyses (Prosvirnova, Batteux, Brameret, Cherfi, Friedlhuber, Roussel, & Rauzy 2013). Its semantics is based on Guarded Transition Systems (Rauzy 2008).

4.1 *Guarded Transition Systems*

A Guarded Transition System (GTS) G is a quintuple $\langle V, E, T, A, \iota \rangle$, where:

- $V = S \uplus F$ is a set of variables, divided into two disjoint sets: a set S of state variables and a set F of flow variables.
- E is a set of events.
- T is a set of transitions. A transition is a triple $t = \langle e, G, P \rangle$, where e is an event from E, G is a Boolean expression built over variables from V and called the guard of the transition, and P is an instruction built over V and called the action or the post-condition of the transition.
- A is an assertion (i.e. an instruction built over V).
- ι is the initial (or default) assignment of variables of V.

A GTS $G = \langle V, E, T, A, \iota \rangle$ is an implicit representation of a labeled Kripke structure, i.e. a graph $\Gamma = (\Sigma, \Theta)$, where

- the set of nodes Σ represent the variable assignments (of V), and
- Θ is the set of edges labeled by the events from E.

Instructions of GTS are defined recursively as follows:

- "$skip$" is an empty instruction.
- If v is a variable and Exp an expression, then "$v := Exp$" is an instruction (called "assignment").
- If C is a Boolean expression, I is an instruction, then "$\text{if } C \text{ then } I$" is an instruction (called "conditional assignment").
- If I_1 and I_2 are two instructions, then so is "$I_1; I_2$" (called "parallel composition").

We shall consider two types of instructions. The "Actions" which are instructions in which left members of assignments are only state variables. The "Assertions" which are instructions in which left members of assignments are only flow variables.

Let denote by $\tau = Propagate(A, \iota, \sigma)$ a variable assignment obtained after applying the assertion A to the variable assignment σ., i.e. the calculation of flow

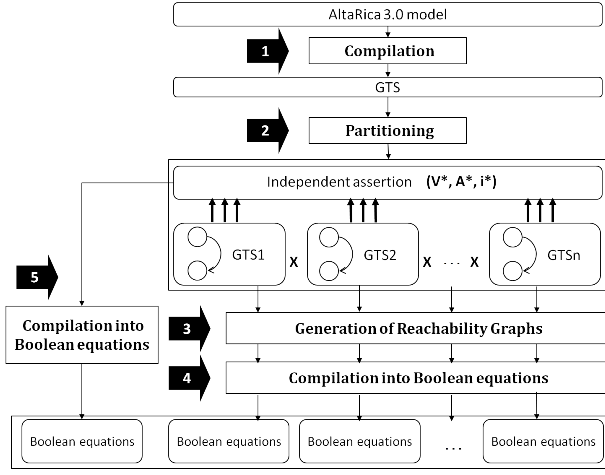

Figure 2: Compilation of AltaRica 3.0 models to Fault Trees

variables value. $Propagate(A, \iota, \sigma)$ computes the values of flow variables using the instructions of the assertion A and the values of state variables in σ. At the end if there are flow variables without any value, they are set to their initial values in ι and the assertion A is applyed to check that all the assignments are satisfied.

4.2 *Compilation to Fault Trees*

The compilation of AltaRica 3.0 models to Fault Trees works (Prosvirnova & Rauzy 2015) in 5 steps (see Figure 2):

1. The AltaRica 3.0 model is flattened into a GTS.
2. The obtained GTS is partitioned into independent GTSs plus an independent assertion.
3. Reachability graphs of each independent GTS are calculated.
4. Each reachability graph is separately compiled into Boolean equations.
5. The independent assertion is compiled into Boolean equations.

The independent assertion $\langle V^*, A^*, \iota^* \rangle$ (the 5th step of the algorithm) is transformed into a set of Boolean formulas in the following way. For each pair (f, d), where $f \in V^*$ is a flow variable and $d \in dom(f)$ is its value, a formula $\phi_{(f,d)}$ is constructed according to the instructions in the assertion A^* and Boolean formulas $\{\phi_{(u,c)}, u \in U, c \in dom(u)\}$ obtained from the compilation of the independent GTSs.

In order to compile the assertion into Boolean formulas efficiently, one need to separate it into independent parts. The dependency relation between variables in the assertion A^* defines a dependency graph. This graph may contain cycles. The strongly connected components of this graph divide variables of A^* into sets and enable to decompose the assertion A^* into blocks of instructions A_i ($i = 1, ..., m$), where m is the number of strongly connected components:
$A^* = A^*_1; A^*_2; \ldots; A^*_m$

Each block of instructions A^*_i is compiled into Boolean formulas recursively. Let denote by

- V^*_i - a set of variables labeling the vertices of the strongly connected component number i.
- A^*_i - an instruction that calculates the values of variables from V^*_i.
- ι^*_i - an initial assignment of variables from V^*_i.
- W^*_i - a set of variables such that variables from V^*_i depend on them in A^*_i.

For all variable v in $V*_i$ the formula $\phi_{(v,c)}$ (where $c \in dom(v)$) is built as follows:

- Let $\Sigma = \underset{w \in W^*_i}{\times} dom(w)$ be the Cartesian product of the domains of variables from W^*_i.
- Let $\sigma \in \Sigma$ be an assignment of variables from W^*_i.
- Let ϕ_σ be a product built over W^*_i (as defined in Section 3.1) calculated as follows:
  $\phi_\sigma = \Pi_{w \in W^*_i}(w = \sigma(w))$
- Let τ be a partial variable assignment, $\tau : V^*_i \cup W^*_i \rightarrow \mathcal{C}$, such that:
  $\forall w \in W^*_i \; \tau(w) = \sigma(w)$
- The partial variable assignment τ can be completed by propagating the assertion A^*_i:
  $\tau = Propagate(A^*_i, \iota^*_i, \tau)$
- Then for each couple (v, c), with $v \in V^*_i$, such that $\tau(v) = c$, the formula associated with (v, c) is updated as follows:
  $\phi_{(v,c)} \leftarrow \phi_{(v,c)} + \phi_\sigma$

At the end of the algorithm, for all variables $v \in V^*$ and their values, we obtain a formula $\phi_{(v,c)}$ built over a finite set of finite domain variables W^*, such that v depends on them in the assertion A^*. We use the diagrammic representation as defined in Section 3.11 to represent these formulas.

As we have seen in Section 3.10, $\phi_{(v,c)} \Leftrightarrow PI[\phi_{(v,c)}]$ and it is the most simplified form of $\phi_{(v,c)}$.

For each variable $v \in V^*$ and its value $c \in dom(v)$, we compute $PI[\phi_{(v,c)}]$ and use this form, which greatly simplifies the generated Fault Tree.

4.3 *Example*

Consider the parametric block diagram use case presented in Section 2. Figure 3 illustrates how each basic block of these diagrams can be represented in AltaRica 3.0.

The variable $State$ represents the internal state of a basic block and takes its value in the domain $BLOCKSTATE = \{ok, ko\}$. A domain is an enumeration having any finite number of values. The

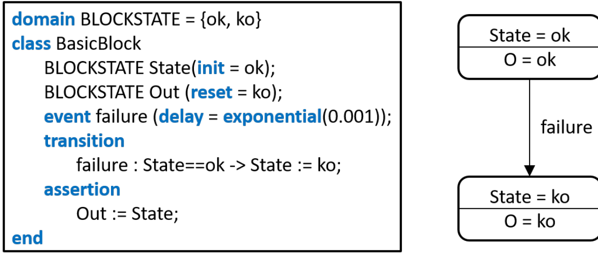

Figure 3: AltaRica 3.0 model of a basic block

event $failure$ represents the internal failure of a basic block. It is possible to associate different probability distributions to the events of basic blocks (e.g. exponential, constant, Weibull). The value of the parameters can also be changed. The behavior of a basic block is represented by a state machine given Figure 3. The variable Out is a flow variable, which represents the output of a basic block. The assertion of a basic block is an instruction, which calculates the value of this variable Out according to the value of the state variable $State$.

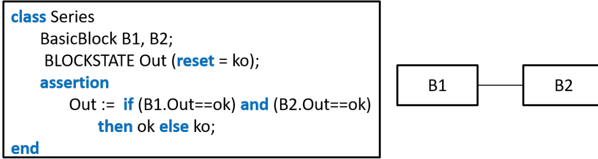

Figure 4: AltaRica 3.0 model of two blocks in series

Figure 4 shows how two blocks in series can be modeled in AltaRica 3.0. The assertion of the whole model is
$B1.Out := B1.State;$
$B2.Out := B2.State;$
Out := if $(B1.Out == ok)$ and $(B2.Out == ok)$ then ok else ko;
The compilation into Fault Trees is performed as follows.
First, local reachability graphs are compiled:
$\phi_{(B1.State,ok)} = true$
$\phi_{(B2.State,ok)} = true$
$\phi_{(B1.State,ko)} = B1.failure$
$\phi_{(B1.State,ko)} = B2.failure$
Second, local assertions are compiled:
$\phi_{(B1.Out,ok)} = (B1.State = ok)$
$\phi_{(B2.Out,ok)} = (B2.State = ok)$
$\phi_{(B1.Out,ko)} = (B1.State = ko)$
$\phi_{(B2.Out,ko)} = (B2.State = ko)$
Third, the global assertion is compiled:
$\phi_{(Out,ok)} = (B1.Out = ok) * (B2.Out = ok)$
$\phi_{(Out,ko)} = ((B1.Out = ok) * (B2.Out = ko)$
$\quad +(B1.Out = ko) * (B2.Out = ok)$
$\quad +(B1.Out = ko) * (B2.Out = ko))$

Figure 5 represents the last formula by means of a variant of Binary Decision Diagram (as presented in Section 3.11). It can be simpified using the algorithm presented in Section 3.10 as follows:
$dom(B1.Out) = dom(B2.Out) = \{ko, ok\}$
$f = (B1.Out = ko)$
$\quad *[(B2.Out = ko) * 1 + [(B2.Out = ok) * 1 + 0]]$
$\quad +[(B1.Out = ok) * [(B2.Out = ko) * 1 + 0] + 0]$
$f_1 = (B2.Out = ko) * 1 + [(B2.Out = ok) * 1 + 0]$
$f_2 = (B2.Out = ko) * 1 + 0$
$f_0 = 0$
$h = f_1 * f_2 + f_0 = (B2.Out = ko) * 1 + 0$
$PI[h] = (B2.Out = ko) * 1 + 0$
$PI[f_1] = 1$
$PI[f_2] = (B2.Out = ko) * 1 + 0$
$PI[f_1] \div PI[h] = 1$
$PI[f_2] \div PI[h] = 0$
$PI[f] = (B1.Out = ko) * 1$
$\quad +[(B2.Out = ko) * 1 + 0]$
which reads as $(B1.Out = ko) + (B2.Out = ko)$. It is the most simplified form of $\phi_{(Out,ko)}$.

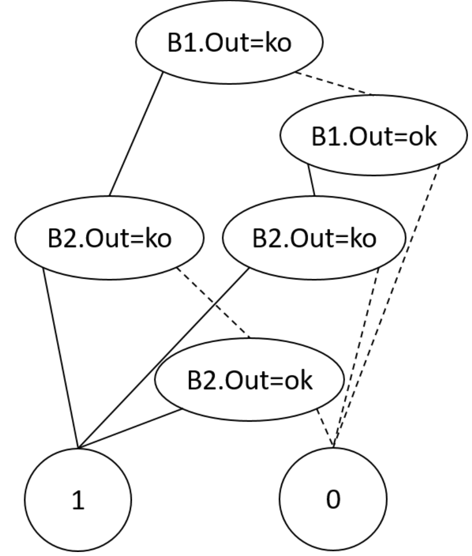

Figure 5: Diagrammatic representation of $((B1.Out = ok) * (B2.Out = ko) + (B1.Out = ko) * (B2.Out = ok) + (B1.Out = ko) * (B2.Out = ko))$

5 EXPERIMENTS

The Fault Tree compiler of the the OpenAltaRica platform produces Fault Trees from AltaRica 3.0 models. More precisely, according to one or several Boolean observers, representing safety cases of the modeled system, the Fault Tree compiler generates Fault Trees in Open-PSA model exchange format (Hibti, Friedlhuber, & Rauzy 2012) with these Boolean observers as top events. The produced Fault Trees can be then assessed by XFTA (Rauzy 2012) to compute Minimal Cut Sets, probabilities of the top events, and so on.

We have implemented the algorithm presented in Section 4 and integrated it in the original version of the Fault Tree compiler. This algorithm greatly simplifies the generated Fault Trees, compared to those generated by the original version.

We have performed experiments with different values for the three parameters s, p and q of the motivating example pictured Figure 1.

In Table 1 we present the results obtained with the original version of the Fault Tree compiler and with the new one.

Table 1: Parametric block diagram use case - fault tree compilation.

Case	n_1	n_2	n_3	n_4	n_5	Number of blocks	Number of gates (new)	Number of gates (original)
1st	3	3	3	0	0	27	243	525
2nd	3	3	3	3	0	81	720	1158
3rd	4	4	4	0	0	64	537	1276
4th	4	4	4	4	0	256	2133	5114
5th	0	2	4	4	4	128	1135	3236
6th	0	3	3	3	2	54	558	1264
7th	0	3	3	3	0	27	243	596
8th	0	3	3	3	3	81	768	1914
9th	0	4	4	4	0	64	549	1562

The first column is the number of the considered cases. The five next columns, from the second column to the sixth column, present the number of components contained in each sub-parts. We start with n_1 parallel blocks, in each block there are n_2 sub-blocks in series, into each sub-block there are n_3 parallel sub-blocks, and so on. For example, the first case means 3 parallel blocks, with 3 sub-blocks in series, each one containing 3 parallel sub-blocks; whereas the fifth case means 2 blocks in series, with 4 parallel sub-blocks, each block containing 4 sub-blocks in series, with 4 parallel sub-blocks into each one. The seventh column represents the total number of basic blocks in the AltaRica 3.0 model. Finally, the eighth and ninth columns represent the number of intermediate events in the Fault Trees generated by the original version of the Fault Tree compiler (ninth column) and the new one (eighth column).

The main observation is about the benefit of the number of generated gates with the new version of the Fault Tree compiler in comparison to the original one. In average, this benefit is of 56.9%. It means that in average with the new algorithm, the number of generated gates is less than 56.9% compared to the number of generated gates with the original one. The minimal value is 37.8% in the second case; and the maximum value is 64.9% in the fifth case.

The benefit obtained with the new version of the algorithm implemented in the Fault Tree compiler is important.

6 CONCLUSION

Boolean models are widely used for probabilistic safety analysis. There are cases however where binary states are not sufficient. For instance, it is sometimes of interest to represent the level of degradation of a component, the quality of signal, and so on. This kind of models can be easily represented with AltaRica 3.0, a high level modeling language dedicated to safety analyses. AltaRica 3.0 comes with several efficient assessment tools, amongst them a Fault Tree compiler.

In this article we presented how the notion of prime implicants can be extended to finite domain calculus. We discussed how the finite domain calculus can be efficiently encoded using a variant of Binary Decision Diagrams. We shown, using a parametric block diagram use case, how these results can be applied to simplify Fault Trees automatically generated from AltaRica 3.0 models. The number of generated intermediate events is on average divided by two, which greatly improves Fault Trees readability and the efficiency of their assessment.

REFERENCES

Aupetit, B., M. Batteux, A. Rauzy, & J.-M. Roussel (2015, September). Improving performances of the altarica 3.0 stochastic simulator. In L. Podofillini, B. Sudret, B. Stojadinovic, E. Zio, and W. Kröger (Eds.), *Safety and Reliability of Complex Engineered Systems: ESREL 2015*, Zürich, Switzerland, pp. 1815–1824. CRC Press.

Brameret, P.-A., A. Rauzy, & J.-M. Roussel (2015). Automated generation of partial markov chain from high level descriptions. *Reliability Engineering & System Safety 139*, 179–187.

Bryant, R. E. (1992, sep). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv. 24*(3), 293–318.

Hibti, M., T. Friedlhuber, & A. Rauzy (2012, June). Overview of the open psa platform. In R. Virolainen (Ed.), *Proceedings of International Joint Conference PSAM'11/ESREL'12*.

Prosvirnova, T., M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, & A. Rauzy (2013, September). 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, pp. 127–132. International Federation of Automatic Control.

Prosvirnova, T. & A. Rauzy (2015). Automated generation of minimal cut sets from altarica 3.0 models. *International Journal of Critical Computer-Based Systems 6*(1), 50–80.

Rauzy, A. (2008). Guarded transition systems: a new states/events formalism for reliability studies. *Journal of Risk and Reliability 222*(4), 495–505.

Rauzy, A. (2012, June). Anatomy of an efficient fault tree assessment engine. In R. Virolainen (Ed.), *Proceedings of International Joint Conference PSAM'11/ESREL'12*.