Sequence Algebra, Sequence Decision Diagrams and Dynamic Fault Trees

Antoine B. Rauzy

LIX – CNRS

École polytechnique

91128 Palaiseau Cedex France

Antoine.Rauzy@lix.polytechnique.fr

Abstract: Dynamic Fault Trees have focused a large attention in the past few years. By adding new gates to static (regular) Fault Trees, Dynamic Fault Trees aim to take into account the order in which events occur. Lesage & al. proposed recently an algebraic framework to give a formal interpretation to these gates.

In this article, we extend Lesage's work by adopting a slightly different perspective. We introduce Sequence Algebras, which can be seen as Algebras of Basic Events representing failures of non-repairable components. We show how to interpret Dynamic Fault Trees within this framework. Finally, we propose a new data structure to encode sets of sequences of Basic Events: Sequence Decision Diagrams. Sequence Decision Diagrams are very much inspired from Minato's Zero-Suppressed Binary Decision Diagrams. We show that all operations of Sequence Algebras can be performed efficiently on this data structure.

1. Introduction

Dynamic Fault Trees have focused a large attention in the past few years (see e.g. [BB03, BC04, Cod05, BCS07a, BCS07b, DD08, MRLB10]). By adding new gates to static (regular) Fault Trees, Dynamic Fault Trees aim to take into account the order in which events occur. Lesage & al. proposed recently an algebraic framework to give a formal interpretation to these gates [MRLB10]. The underlying idea is twofold. First, one

considers basic events that represent failures of non-repairable components; Second, one introduces a new operator ‘$A \triangleleft B$’ to represent the event ‘the event A occurs before the event B’. Priority And gates, Functional Dependency Gates and Spares gates can be interpreted by means of classical combinatorial operators (and, or) and ‘$\triangleleft$’.

In this article, we adopt a slightly different point of view. Rather than introducing constraints on order of events in the Boolean Algebra, we work directly with sequences. This small perspective shift makes it possible too extend Lesage & al ideas into two directions.

First, we introduce the so-called Sequence Algebra, which can be seen as the algebra of Basic Events that represent failures of non repairable components. This algebra lies between the pure syntactic monoid and the Boolean algebra: in a sequence, Basic Events are ordered (like in the monoid) but can occur at most once (like in Boolean algebra). Operators of this algebra are concatenation, hook, symmetric and asymmetric shuffles, and disjunction. We show how to interpret Dynamic Fault Trees within this framework. Although this interpretation relies on the same principles as those proposed by Lesage & al, it gives a new light on Dynamic Fault Trees semantics.

Second, we propose a new data structure, so-called Sequence Decision Diagrams (SDD), to handle set of sequences. SDD are very much inspired from Minato’s Zero-Suppressed Binary Decision Diagrams [Min93]. We show that all operations of Sequence Algebras can be performed efficiently on SDD. SDD are a good candidate for Dynamic Fault Trees assessment at industrial scale.

The remainder of this article is organized as follows. Section 2 presents sequence algebras. Section 3 shows how to encode Dynamic Fault Trees into this framework. Section 4 describes Sequence Decision Diagrams. Finally, Section 5 discusses related works.

2. Sequence Algebras

In the sequel, we assume the reader familiar with Fault Trees.

2.1. Sequences of basic events

As proposed by Lesage & al in [MRLB10], we consider here non-repairable components, i.e. Basic Events that occur at a given date (taken from 0 to $+\infty$). We denote basic events by capital letters possibly with subscript, e.g. A, B_1… The date of occurrence of a basic event A is simply denoted by date(A).

If we observe a system which involves basic events A_1, A_2…A_n throughout a given period of time t, some of the A_i's occur once, some other do not occur. That is, we observe a sequence of basic events A_{i1} A_{i2}… A_{ik} such that $0 \leq \text{date}(A_{i1}) \leq \text{date}(A_{i2}) \leq \ldots < \text{date}(A_{ik}) \leq t$. In general, these inequalities are strict for probability distributions associated with basic events are continuous functions with infinite supports.

Formalisms such as Dynamic Fault Trees aim to represent in a convenient way sets of such sequences, namely sets of sequences that produce a failure of the system under study.

We denote sequences by lower case letters, taken from the end of the alphabet, e.g. s, t, u.... We denote sets of sequences also by lower case letters, but taken from the beginning of the alphabet, e.g. f, g, h… Moreover, we denote them as sums (or disjunctions), e.g. ABC+DE+AD. We denote the empty sequence by ε and the empty set of sequences by 0.

By abuse, we say that a basic event E belongs to a sequence s, which we denote $E \in s$, if E occurs in s. Similarly, we use set usual set operations $\cup$ (union), $\cap$ (intersection), $\subseteq$ (inclusion) to manipulate sets of events occurring in sequences (but not the sequences themselves), e.g. $ABC \cap DE = \varnothing$ and $DB \subseteq ABCD$.

Let $s = A_1 \ldots A_m$ and $t = B_1 \ldots B_n$ be two sequences of basic events.

The sequences s and t are said compatible if there is no two events that do not occur in the same order in s and t, i.e. four indices $1 \leq i_1 < i_2 \leq m$ and $1 \leq j_1 < j_2 \leq n$, such that $A_{i1} = B_{j2}$ and $A_{i2} = B_{j1}$. The sequences s and t are said incompatible otherwise. For instance, ABCD and EBFGDH are compatible (the events they share, namely B and D,

occur in the same order in the two sequences), while ABCDE and FEGDH are not (B and D occur in a different order in the two sequences).

The sequence s is a subsequence of the sequence t, which we denote by $s \leq t$, if $s \subseteq t$ and s and t are compatible. For instance, $BD \leq ABCDE$ (in this case, the inequality is strict so we could write $BD < ABCDE$).

Two sequences s and t can be combined in four different "natural" ways (concatenation, hook, symmetric shuffle and asymmetric shuffle), plus two derived ones (left hook and left asymmetric shuffle). Each of these combinations is realized by means of an operator that takes two sequences as input returns a possibly empty set of sequences.

The most restrictive combination is the concatenation. The less restrictive is the symmetric shuffle.

The concatenation of $s = A_1 \ldots A_m$ and $t = B_1 \ldots B_n$, denoted by s.t, returns the singleton $A_1 \ldots A_m B_1 \ldots B_n$ if $s \cap t = \varnothing$ and 0 otherwise.

The symmetric shuffle of sequences s and t, denoted by s*t, is the set of sequences obtained by interleaving s and t in all possible ways, i.e. the sequences u such that:

- $s \cup t = u$.
- s and t are compatible with u.

Here follows examples of symmetric shuffles.

- ABC * DE = ABCDE + ABDCE + ABDEC + ADBCE + ADBEC + ADEBC + DABCE + DABEC + DAEBC + DEABC.
- ABCD * BEFD = ABCEFD + ABECFD + ABEFCD
- ABCDE * DFB = 0

Note that the shuffle of two sequences is empty if (and only if) these sequences are incompatible.

The hook, denoted by s,t, is similar to the concatenation but a bit less restrictive. It is defined as follows: s,t = sv if there exists two sequences u and v such $t = uv$, $u \leq s$, and $v \cap s = \varnothing$. For instance, ABCD , BDE = ABCDE. We shall see section 3 that the hook may be used to approximate the semantics of spare gates.

The asymmetric shuffle, denoted by s;t, is defined as the symmetric shuffle, with the additional condition that the last event comes from t. For instance, ABC ; DE = ABCDE + ABDCE + ADBCE + DABCE. We shall see section 3 that the asymmetric shuffle can be used to model priority and gates.

Aside the hook and the asymmetric shuffle there are their left counterparts. We mention them here for the sake of the completeness, but we won't use them. The left hook, denoted by s<,t, is defined as s <, t = ut, if there exists two sequences u and v such that s = uv, u ∩ t = ∅ and v ≤ t. The left asymmetric shuffle, denoted by s <; t, is defined as the symmetric shuffle, with the additional condition that the first event comes from A.

2.2. Operators

So far, we introduced five operators:

- The concatenation operator '.', which we kept implicit most of the time (ABC stands for A.B.C).
- The symmetric shuffle '*'.
- The hook ','.
- The asymmetric shuffle ';'.
- The disjunction '+'.

The Sequence Algebra is obtained by lifting up the definitions of these operators to sets of sequences. Given an alphabet Σ, i.e. a set of basic events, we define (well formed) formulas over Σ as the smallest set such that:

- The constant 0 (empty set of sequences) and ε (empty sequence) are formulas.
- Basic events of Σ are formulas.
- If f and g are formulas, then so are f.g, f+g, f*g, f,g and f;g.

Formulas are interpreted as disjunctions of sequences of basic events. The interpretation of a formula f is denoted [f]. The interpretation rules are defined recursively as follows.

- Constants are interpreted by themselves: $[0] = 0$, $[\varepsilon] = \varepsilon$.
- A basic event A is interpreted as the set containing only one sequence containing only A. The set and the sequence are both denoted by A: $[A] = A$.
- $[f.g] = \{ w \in u.v \mid u \in [f], v \in [g] \}$
- $[f,g] = \{ w \in f,g \mid u \in [f], v \in [g] \}$
- $[f^*g] = \{ w \in u^*v \mid u \in [f], v \in [g]\}$
- $[f;g] = \{ w \in u;v \mid u \in [f], v \in [g]\}$
- $[f+g] = [f] + [g]$ (assuming that only one copy of each sequence is kept).

We shall use parentheses to make clear the structure of the formulas.

Examples:

- (A.B + C.D) . (A.E + F.D) = A.B.F.D + C.D.A.E (terms A.B.A.E and C.D.F.D are not sequences for they contain repeated events).
- (A.B.C + D.E) * C.E.B = D.E * C.E.B = D.C.E.B + C.D.E.B (terms A.B.C and C.E.B are incompatible).
- A.B.C ; (D.C + E) = A.B.D.C + A.D.B.C + D.A.B.C + A.B.C.E

It is worth noticing that the above interpretation rules are actually an algorithm to reduce a formula into a canonical form, in which hooks and shuffles have been resolved and formulas are written as sums of sequences.

2.3. Static Fault Trees

Static Fault Trees can be seen as encoding sets of sequences, but with the additional assumption that if a given sequence leads to the failure of the system under study, then any permutation of this sequence leads also to the same failure. Under this interpretation, the sequence operators * and + correspond respectively to the Boolean operators $\wedge$ and $\vee$.

Consider for instance a Fault Tree with structure function $(A \vee B \wedge C) \wedge D$. This fault tree has two minimal cutsets AD and BCD and three non minimal cutsets ABD, ACD and ABCD. Its development in the Boolean algebra is as follows.

$(A \vee (B \wedge C)) \wedge D$

$= (A \wedge D) \vee (B \wedge C \wedge D)$

Its interpretation in the sequence algebra is as follows.

(A + B*C) * D

= (A + B.C + C.B) * D

= A*D + (B.C)*D + (C.B)*D

= A.D + D.A + B.C.D + B.D.C + D.B.C + C.B.D + C.D.B + D.C.B

For each cutset of the Boolean interpretation, we retrieve, in the sequence interpretation, all of the sequences that correspond to permutations of that cutset. In that case, all cutsets (and therefore sequences) are minimal.

The above remark can be formalized establishing an isomorphism between the lattice generated by E, $\vee$ and $\wedge$ and a quotient of sub-algebra obtained from E, + and * by a natural equivalence relation (based on minimal sequences and permutations). These developments would bring us too far in the framework of this article.

2.4. Additional Properties

In order to provide the reader with a better intuition we give hereafter a number of properties of operators of the sequence algebra (we give them without proof here but they are easy to check).

Concatenation:

- $f.0 = 0.f = 0$ $\quad f.\varepsilon = \varepsilon.f = f$
- $f.(g.h) = (f.g).h$

Disjunction:

- $f + 0 = 0 + f = f$
- $f + g = g + f$ $\quad f + (g + h) = (f + g) + h$ $\quad f + f = f$

Shuffle:

- $f * 0 = 0 * f = 0$ $f * \varepsilon = \varepsilon * f = f$
- $f * g = g * f$ $f * (g * h) = (f * g) * h$ $(f + g) * h = (f * h) + (g * h)$
- $(A + B) * (A + B) \neq A + B$
- $(A * B) + C \neq (A + C) * (B + C)$

The two above equality are true if we keep only minimal sequences.

Hook:

- $f , 0 = 0 , f = 0$ $f , \varepsilon = \varepsilon, f = f$
- $(AB , BC) , ACD \neq AB , (BC , ACD)$

Asymmetric shuffle:

- $f ; 0 = 0 ; f = 0$ $f ; \varepsilon = \varepsilon ; f = f$
- $(A ; B) ; C \neq A ; (B ; C)$

3. Dynamic Fault Trees

Several types of dynamic gates have been proposed. We shall consider here Priority And Gates, Functional Dependencies, Sequence Enforcers and different Spare Gates, according to the presentation by Dugan & al. in Ref. [CSD00]. The analysis of the semantics of the different gates is borrowed to Lesage & al. work (especially to G. Merle PhD Thesis [Mer10]).

3.1. Priority And Gates

A Priority And Gate (PAND) over events $E_1 \ldots E_k$ is realized if $E_1 \ldots E_k$ occur in this order. Priority And Gates are interpreted as asymmetric shuffles:

- $PAND(E_1,\ldots,E_k) = (\ldots(E_1;E_2);E_3)\ldots;E_k)$

Note that this interpretation works even if the E_i's share some (basic) events.

3.2. Functional Dependencies

A Functional Dependency (FDEP) asserts a functional dependency. The failure of the trigger event causes the immediate and simultaneous failure of the dependent (basic) events. This is not a gate in the sense that it has no output event. Consider the Functional Dependency Gate FDEP(T; E_1…,E_k), where T is the trigger event and E_1…E_k are the dependent events. As noted by Stamatelatos and Vesely in [SV02], the E_i's can occur either by themselves or because they have been triggered by T (no matter the order in which these actions occur). Each E_i can therefore be seen as an OR gate between T and the event E_i-fails-by-itself.

3.3. Sequence Enforcer

A Sequence Enforcer (SEQ) asserts that events can occur only in a given order. This is not a gate in the sense that it has no output event. Consider the Sequence Enforcer SEQ(E_1…E_k). It means that E_2 must occur after E_1, E_3 after E_2 … E_k after E_{k-1}. So, we can see each E_i ($1< i \leq k$) as a PAND gate (itself interpreted as an asymmetric shuffle), between E_{i-1} and the actual event E_i (E_i-fail-by-itself if one will).

3.4. Spare Gates

Cold, Warm, Hot Spare Gates (CSP, WSP, HSP): When the primary input fails, available spare inputs are used in order until none is left, at which time the gate fails. Spares can be shared among spare gates, in which case the first spare gate to utilize the spare makes the spare inaccessible to the other spare gates. The "temperature" of a spare gate indicates whether unused spares cannot fail (cold), fail at a rate attenuated by the dormancy factor of the spare (warm), or fail at their full rates (hot).

Consider first the simplest case, i.e. a hot spare Gate HSP(P,S) where P is the primary input event and S represents the failure of the spare component. Clearly HSP(P,S) is realized if both P and S are realized. The order does not matter because S is supposed to have the same behavior whether it is active or not. So, a priori, HSP(P,S) could be interpreted as an AND gate.

However, this interpretation does not work if the spare component is shared. Assume now that we have two spare gates H_1 = HSP(P,S) and H_2 = HSP(Q,S) and a gate H = H_1 or H_2. If P occurs first, then the spare component is activated. Now if Q occurs, then H_2 will occur simultaneously, i.e. without waiting the occurrence of S, because the spare component is already in use. Therefore H will occur. As a consequence, H = H_1 or H_2 ≠ (P and S) or (Q and S). This is a very tricky situation because H_2 depends actually on P. A solution consists in defining H_1 as (P and (S or Q)) and H_2 as (Q and (S or P)). However, the generalization of this idea is tedious.

The above example shows that even in the simplest case, i.e. hot redundancy, the semantics of a spare gate does not depend only of the inputs of the gate. The interpretation algorithm has to browse the Fault Tree, at least if spare components can be used in several places (but in only one place at once).

Warm spare gates are the most general case: a hot spare gate is just a warm spare gate with the same probability distribution of failure for the spare component whether it is active or dormant; a cold spare gate is just a warm spare gate in which the spare component cannot fail when it is dormant. So, let us consider a warm spare gate W = WSP(P,S). Let us denote by S_d and S_a the two events representing the failure of the spare component when it is respectively dormant and active. The event W occurs if P occurs, then S_a occurs or if S_d occurs then P occurs. A candidate interpretation for W is therefore W = $(P;S_a) + (S_d;P)$. Consider however the case where $S_a = U_a.V_a$ (the other events being basic ones). The development of W would be as follows.

- $W = (P;S_a) + (S_d;P) = (P; U_a.V_a) + (S_d;P) = P.U_a.V_a + U_a.P.V_a + S_d.P$

The sequence $U_a.P.V_a$ should not belong to the solution because it would mean that a part of the spare component fails in an active more before the failure of the main component, i.e. before the spare component is activated.

The correct interpretation for W is therefore W = $(P,S_a) + (S_d;P)$. By using a hook rather than an asymmetric shuffle, we warranty that no part of the spare component fails before the failure of the primary component. Strictly speaking, this interpretation is still an approximation because it assumes that S_a “starts” at 0.

Note that if the spare component is used somewhere else, we can use the same method as explained above, i.e. replace the case where the spare component fails in a dormant mode by a disjunction of this failure and the failure of the primary input component(s) of the other spare gate(s). For instance, if W_1 = WSP(P,S) and W_2 = WSP(Q,S), then we can interpret W_1 as follows. W_1 = (P,S_a) + ((Q+S_d);P).

4. Sequence Decision Diagrams

4.1. Data Structure

Sets of sequences over a set of basic events Σ can be seen finite languages, finite sets of words, over the alphabet Σ. They can therefore be represented by finite state automata. For instance, the set of sequences {ABC, ACD, ADC, BD} can be represented by the minimal automaton pictured Figure 1.

Sequence Decision Diagrams (SDD) are a variation on Minato's Zero-Suppressed Binary Decision Diagrams (ZBDD) [Min93] that makes it possible both to encode automata and to perform Sequence Algebra operations in an efficient way. ZBDD are derived from Bryant's Binary Decision Diagrams (BDD) [Bry86, BRB90].

In a Finite State Automaton, there are two kinds of objects: nodes and edges. Edges are labelled with events. Each node has typically a list of out-edges. The idea of ZBDD (and therefore of SDD) is to use nodes both to represent nodes, i.e. lists of out-edges, and edges. The SDD encoding the set of sequences {ABC, ACD, ADC, BD} is pictured Figure 2.

A SDD is a directed acyclic graph with two distinguished nodes: one to represent the empty set $\varnothing$ and one to represent the singleton {$\varnothing$}. The other nodes are labeled with elements of the alphabet Σ and have two out-edges (left and right). Each internal node can therefore be denoted as a triple $\langle A,l,r\rangle$ where $A \in \Sigma$, and l and r are the nodes pointed by respectively the left and right out-edges. On Figure 2, left out-edges are represented by plain lines, while right out-edges are represented by dashed lines. Each node n is associated with a set [n] of sequences by means of the following rules.

- $[\varnothing] = 0$

- $[\{\varnothing\}] = \varepsilon$
- $[\langle A,l,r\rangle] = A.[l] + [r]$

Some of SDD nodes correspond to state of the automaton. On Figure 2, we labelled these nodes with the corresponding numbers of the state of the automaton pictured Figure 1.

SDD nodes are right-ordered, i.e. that a total order over E is defined (typically by means of heuristics) and by construction if the right out-edge of a node labelled with $A \in \Sigma$ points to a node labelled with $B \in \Sigma$, then $A < B$. Here stands the main difference with ZBDD which are both left and right ordered.

Nodes are maintained into a table (as suggested first in [BRB90]) so that for each triple $\langle A,l,r\rangle$ there is (at most) an unique node in the table encoding the triple. This technique ensures that diagrams are minimal and makes it possible to check equality of two sets in constant time.

4.2. Operations

Most of operations on BDD (and ZBDD) are described by means of recursive equations. The same principle applies to SDD.

Disjunction: The simplest operation is the disjunction of two sets of equations encoded by two SDD nodes.

- $n + 0 = n + 0 = n$
- $n + \varepsilon = \varepsilon + n \equiv \varepsilon$
- $(A.l_1 + r_1) + (A.l_2 + r_2) = A.(l_1 + l_2) + (r_1 + r_2)$
- $(A.l_1 + r_1) + (B.l_2 + r_2) = A.l_1 + (r_1 + (B.l_2 + r_2))$ assuming $A < B$

Several remarks can be made about the above equations.

First, strictly speaking '$A + \varepsilon$' is different from ε. However, we are interested in coherent (monotone) systems only. If a given sequence leads to the failure of the system, then

any longer sequence maintains the system in a failure mode. So it is safe to consider 'A + ε' as ε.

Second, it is easy to turn these equations into an actual algorithm. However, experience shows that to be efficient such an algorithm should better use a caching mechanism. In our case, a hashcache contains quadruple (+,n,m,r) where n and m are the SDD nodes given as arguments and r is the result SDD node. Before actually performing the operation n+m, the hashcache is looked-up. If it contains an entry for this operation then the result is returned, otherwise the operation is actually performed and its result is stored into the hashcache.

Concatenation: Equations that define concatenation are straightforward: it suffices to traverse the left argument and to replace ε leaves by the right argument. However, doing so may produce sequences with repeated events. Therefore, the algorithm has to maintain a third argument, which is the list of events already visited. This list can be encoded by means of a SDD (so that the caching mechanism works in a uniform way).

- $m \,.\, n = m \,.\, n \mid 0$ 0 encodes here the empty set of events
- $0 \,.\, n \mid x = 0$
- $\varepsilon \,.\, 0 \mid x = 0$
- $\varepsilon \,.\, \varepsilon \mid x = \varepsilon$
- $\varepsilon \,.\, (A.l + r) \mid x = 0$ if $A \in x$

  $= A.(\varepsilon.l \mid x) + (\varepsilon.r \mid x)$ otherwise
- $(A.l + r) \,.\, n \mid x = A.(l.n \mid x + A) + (r \,.\, n \mid x + A)$

Equations for the shuffle, asymmetric shuffle and hook work in a similar way.

4.3. Minimal Sequences

A sequence is a minimal failure sequence for a given system if it leads to the failure of the system and no smaller sequence does. The SDD encoding with a given set of sequences may contain non minimal sequences. The algorithm to extract minimal sequences is almost the same as the one proposed by the author in [Rau93] to extract

minimal cutsets from a BDD (except that both left and right branches have to be pruned). It is based on the following recursive equations.

- $minseq(0) = 0$
- $minseq(\varepsilon) = \varepsilon$
- $minseq(A.l + r) = A.(minseq(l) \div minseq(r)) + (minseq(r) \div minseq(l))$

where '÷' is the "without" operator defined by the following recursive equations.

- $0 \div n = 0$
- $m \div \varepsilon = 0$
- $\varepsilon \div n = \varepsilon$
- $(A.l_1 + r_1) \div (A.l_2 + r_2) = A.((l_1 \div l_2) \div r_2) + (r_1 \div (A.l_2 + r_2))$
- $(A.l_1 + r_1) \div (B.l_2 + r_2) = A.(l_1 \div (B.l_2 + r_2)) + (r_1 \div (B.l_2 + r_2))$ if $A \neq B$

The justification of the above algorithm is essentially the same as in [Rau93].

It is worth noticing that the construction of the SDD using the algorithms given in the previous section can be interleaved with the minimization process proposed in this section. In the same vein, it is possible to define cutoffs (on the length or the probability of sequences) and to use them to keep SDD of reasonable size. These approximations rely on the same principles as those defined by the author in [Rau01].

5. Related Works

In the past few years, several authors proposed to translate Dynamic Fault Trees to Markov chains (e.g. [BCS07a, BCS07b]), Stochastic Petri Nets [BC04, Cod05] and Bayesian Networks [BD05]. Once again, our work is mainly inspired by Lesage & al algebraic framework [MRLB10, Mer10].

In sequence algebras, all basic events are assumed to be associated with probability distributions that are continuous functions with supports in $[0,+\infty)$. This algebraic framework has many good properties. It would be interesting however to extend it by considering supports not starting a 0, but at dynamically determined dates. In other

words events would have a start date and an end date. Such an extension would lead us very close to Allen Algebras [All83]. The comparison of sequence and Allen algebras is certainly worth to do both in terms of expressive power and algorithms. In particular, it would be interesting to investigate the relationships of Allen Algebras with Bon and Bouissou's BDMP [BB03].

SDD can be seen an efficient algorithmic framework to translate Dynamic Fault Trees into Markov chains, although this translation works even if Markovian hypotheses are not verified. Moreover, SDD make it possible to work on minimal sequences only (as shown in the previous section). SDD are very close to Minato's ZBDD. The idea of using BDD like data structures to encode finite state automata appeared in many places in the literature. Zampunieris and Le Charlier Shared Trees [ZlC95] are certainly worth to mention here.

6. Conclusion

In this article, we introduced sequence algebras. We discussed how to use this framework to deal with Dynamic Fault Trees. Finally, we introduced Sequence Decision Diagrams, a ZBDD like data structure to handle set of sequences of basic events.

The translation of Dynamic Fault Trees into sequence formulas is certainly useful to get a better understanding of their semantics. In the reverse way, it may be the case that sequence algebra operators will suggest new types of gates for Dynamic Fault Trees.

Sequence Decision Diagrams provide an efficient algorithmic framework to deal with set of sequences. In this article, we didn't consider quantitative assessments, i.e. determination of top event probability, importance factors… Probabilistic computations can be performed on SDD by considering the latter's as Markov chains or using stochastic simulation. We believe however that specific, therefore more efficient, algorithms can be designed.

Another interesting perspective is to use sequence algebras as the target of the compilation of high level formal modeling languages such as AltaRica [Rau02].

7. References

[All83] J. F. Allen. Maintaining knowledge about temporal intervals. In: *Communications of the ACM*. 26/11/1983. ACM Press. pp 832-843, ISSN 0001-0782.

[BC04] A. Bobbio and D. Codetta Raiteri. Parametric Fault Trees with Dynamic Gates and Repair Boxes. *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS 2004)*, Los Angeles, CA, USA 2004, pp 459-465.

[BCS07a] H. Boudali, P. Crouzen and M. Stoelinga. Dynamic Fault Tree analysis through input/output interactive Markov chains. In *Proceedings of International Conference on Dependable Systems and Networks (DSN 2007)*, IEEE Computer Society, 2007, pp 25-38.

[BCS07a] H. Boudali, P. Crouzen and M. Stoelinga. A Compositional Semantics for Dynamic Fault Trees in terms of interactive Markov chains. In *Proceedings of International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, Springer Verlag – LNCS vol° 4762, 2007, pp 441-456.

[BD05] H. Boudali and J.B. Dugan. A discrete-time Bayesian network reliability modeling and analysis framework. *Reliability Engineering and System Safety*, vol 87, pp 337-349, 2005

[BB03] M. Bouissou & J.L. Bon, A new formalism that combines advantages of fault-trees and Markov models: Boolean logic-driven Markov processes. Reliability Engineering & System Safety, vol. 82, 2003, pp. 149-163.

[Bry86] R. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45. IEEE 0738, 1990.

[Cod05] D. Codetta Raiteri. The Conversion of Dynamic Fault Trees to Stochastic Petri Nets, as a case of Graph Transformation. *Electronic Notes on Theoretical Computer Science*, vol. 127, n°2, pp 45-60. 2005.

[CSD00] D. Coppit, K. J. Sullivan, J. B. Dugan, Formal Semantics for Computational Engineering: A Case Study on Dynamic Fault Trees. In proceedings of p. 270, 11th International Symposium on Software Reliability Engineering (ISSRE'00), 2000.

[DD08] J. Dehlinger, J. B. Dugan: Dynamic Event/Fault Tree Analysis of Multi-agent Systems Using Galileo. In *Proceedings Workshop on Integration of Software Engineering and Agent Technologies, 2008: pp 429-434*

[Mer10] G. Merle. Algebraic modeling of Dynamic Fault Trees, contribution to qualitative and quantitative analysis, PhD thesis, LURPA, ENS de Cachan, 218 p., July 2010

[MRLB10] G. Merle, J.-M. Roussel, J.-J. Lesage and A. Bobbio. Probabilistic Algebraic Analysis of Fault Trees with Priority Gates and Repeated Events. *IEEE Transactions on Reliability*, vol. 59, n°1, pp 250-261, 2010.

[Min93] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference, DAC'93*, pages 272–277, 1993.

[Rau93] A. Rauzy. New Algorithms for Fault Trees Analysis. *Reliability Engineering & System Safety*, Volume 59, Issue 2, pp 203–211, 1993.

[Rau01] A. Rauzy. Mathematical Foundations of Minimal Cutsets. *IEEE Transactions on Reliability*, volume 50, number 4, pages 389-396, 2001.

[Rau02] A. Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, Elsevier, Volume 78, Issue 1, pp 1-12, 2002.

[SV02] M. Stamatelatos and W. Vesely. *Fault Trees Handbook with Aerospace Applications*. NASA Office of Safety and Mission Assurance. Vol. 1.1, pp 1-205, 2002.

[ZlC95] D. Zampunieris and B. Le Charlier. Efficient Handling of Large Sets of Tuples with Sharing Trees. In *Proceedings of Data Compression Conference, DCC'95*, October 1995.

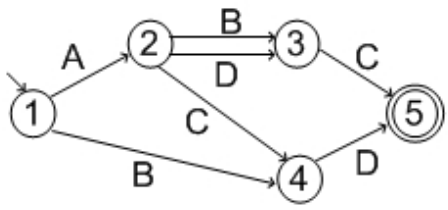

Figure 1. The (minimal) automaton encoding ABC + ADC + ACD + BD.

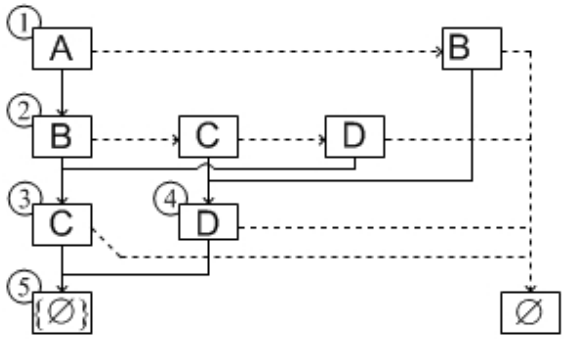

Figure 2. The Sequence Decision Diagram encoding ABC + ADC + ACD + BD