Complete Sequence Generation Algorithm for Reliability Analysis of Dynamic Systems with Sequence-Dependent Failures

Monika Shrestha, Liudong Xing, Haiping Xu
University of Massachusetts Dartmouth, 285 Old Westport Road, North Dartmouth, MA 02747, USA
E-mails: {mshrestha, lxing, hxu}@umassd.edu

Key Words: Dynamic fault tree, priority-AND, sequence dependence, sequential binary decision diagram, topological sorting.

Abstract -- Sequence-dependent failures can be found in many real-life fault-tolerant systems where the occurrence order of fault events is important. The priority-AND (pAND) gates have been used to model such dependent behavior in the dynamic fault tree (DFT) reliability analysis. In order to overcome limitations of existing approaches (*e.g.*, state-space-based or simulation-based methodologies), a combinatorial and analytical method has recently been proposed, which offers an exact and efficient solution to the reliability analysis of dynamic systems with sequence-dependent failures. Using this approach, it is necessary to adopt an efficient algorithm to enumerate the list of complete sequences from partial sequences. Thus in this paper, we propose a sorting algorithm for enumerating complete sequences of events from the partial orders/sequences of the events for the reliability analysis of systems subject to sequence-dependent failure behavior. The generation algorithm is based on the topological sorting algorithm which finds the optimal sequence that satisfies the precedence constraints in a directed acyclic graph. Several examples are given to illustrate the basics and application of the proposed approach.

Acronyms

BDD	Binary Decision Diagram
DAG	Direct Acyclic Graph
DFT	Dynamic Fault Tree
IE	Inclusion-Exclusion
pAND	Priority-AND
pdf	probability density function
r.v.	random variable
SBDD	Sequential Binary Decision Diagram

I. Introduction

Traditional static fault trees [1, 2] cannot capture dynamic behavior of system failures related to sequence-dependence in which the order that fault events occur is important. As an example of sequence-dependent failures, consider a standby sparing system shown in Figure 1. The system has one primary unit (*M*) and one standby spare unit (*S*) connected with a switch controller (*Sw*). The system can continue to operate when the switch controller fails after the primary unit fails as the standby is already in use. However, if the switch controller fails before the primary unit fails, then the system fails upon the failure of the primary unit as the standby unit cannot be switched into active operation [1]. Thus, the failure criteria of the system depend not only on the combinations of events, but also on the sequence in which events occur. In order to model such sequence-dependent behavior, a priority-AND (pAND) gate has been proposed in the dynamic fault tree (DFT) reliability analysis [1, 3, 4].

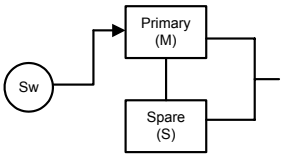


Figure 1. An example of sequence dependent systems

The pAND gate is a dynamic gate that is logically equivalent to an AND gate along with an added condition that events must occur in a specific order (from left to right). As shown in Figure 2, a pAND gate has two inputs *A* and *B* whose output is true if both *A* and *B* have occurred, and *A* occurred before *B*. The gate will not fire if either of the two events has not occurred, or if *B* occurred before *A*.

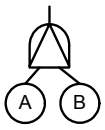


Figure 2. The pAND gate

Figure 3 illustrates the DFT model of the sequence-dependent system of Figure 1 constructed using the traditional AND and OR gates, and the pAND gate. It shows that the system fails when both the primary unit and the standby unit have failed, or when both the primary unit and the switch have failed and the switch fails before the primary unit fails.

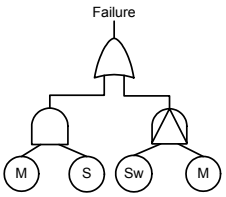


Figure 3. DFT of the example sequence dependent system

There exist various techniques to analyze a DFT with the pAND sequence dependent behavior. Typically, the DFT can be solved by automatic conversion to an equivalent Markov model [2, 5]. However, the Markov-based methods are subject to the well-known state-space explosion problem and typically require exponential time-to-failure distribution for

the system components. Therefore, they are generally applicable for systems with very limited size.

To mitigate the state-space explosion problem of the Markov methods, the modularization technique [2, 5, 6] has been proposed to analyze a large dynamic system via a divide-and-conquer strategy, where the system is divided into independent subtrees. A subtree containing a dynamic gate will be solved using a Markov model; otherwise, it is solved using a combinatorial method called Binary Decision Diagrams (BDD) [7, 8]. Solutions of all the subtrees will be integrated to obtain the solution for the entire fault tree model. However, in practice the modularization technique may not work well for complex systems with lots of repeated or shared events and a high degree of interdependence [9].

Monte Carlo simulation [10] represents another class of methods used to solve DFT. The simulation-based methods can offer great generality in representation and solution to highly complex and dynamic systems. However, they have certain limitations. They can only offer approximate results. They often involve long computational time, especially when results with high degree of accuracy are desired. They also require a completely new simulation to be performed whenever the input failure parameter value changes. Bayesian network approach is another method proposed for the DFT analysis [11]. However, it has the same complexity problem as the Markov-based methods.

Recently, an analytical method based on inclusion-exclusion (IE) formulation [12,13] has been proposed to analyze a DFT with pAND gates, where a set of minimal cut sets/sequences is first generated from the DFT specifications, and is then combined using the IE formula to obtain the system unreliability. The major problem of this method is it requires enumeration or *a priori* knowledge of the minimal cut sets/sequences, which is often a costly process with exponential complexity. Also, the IE-based method in [12] assumes the exponential time-to-failure distribution for the system components.

In order to overcome the limitations of the above described existing methods, a combinatorial and analytical method has been proposed in [9]. The method can offer an exact and efficient solution to the reliability analysis of non-repairable systems with the sequence dependent behavior, without requiring the enumeration or *a prior* knowledge of the minimal cut sets/sequences. Also, this method does not have any limitation on the type of time-to-failure distributions for the system components. In [9], the necessity of an efficient algorithm to generate the complete orders/sequences from the partial orders/sequences for the evaluation of the system unreliability has been pointed out. Hence, in this paper, we address the above need by proposing a complete sequence generation algorithm based on topological sorting. Several examples are given to show the application of the proposed algorithm.

The remainder of the paper is organized as follows. Section II summarizes the combinatorial method proposed in [9] and indicates the necessity of the generation algorithm to be discussed in Section III. Section III presents the proposed complete sequence generation algorithm. Section IV presents the step-by-step analysis of an illustrative example using the proposed algorithm, as well as results for several other examples. Lastly, Section V gives conclusions as well as directions for future work.

II. Background

In order to illustrate the necessity and application of the complete sequence generation algorithm, we brief the analytical approach proposed in [9] for analyzing DFT with pAND gates in this section.

The combinatorial approach [9] integrates an analytical solution for considering pAND dependence at the lower level, and a Sequential BDD (SBDD)-based solution for representing the system structure function at the upper level. This approach can be implemented as a three-step process summarized below and illustrated using the example sequence-dependent system in Figure 1.

Step 1: Transformation of system DFT model: The pAND gates in the system DFT is transformed into a set of sequential events such that the final fault tree does not contain pAND gates. For example, Figure 4 shows the converted fault tree of Figure 3, where "$\prec$" represents the precedence order of component failure. Thus, the sequential event "$Sw \prec M$" means that the switch fails before the primary unit M fails.

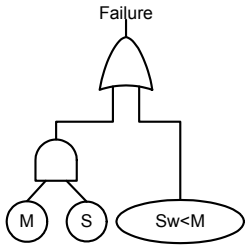


Figure 4. Transformed DFT of the example system

Step 2: Generation of the system SBDD model: The system SBDD model is generated from the transformed fault tree obtained in the first step in the bottom-up manner using manipulation rules of traditional BDD [7]. Figure 5 shows the final SBDD resulting from the application of first ANDing between the basic events M and S and then ORing them with the sequential event ($Sw \prec M$) in the transformed DFT of Figure 4.

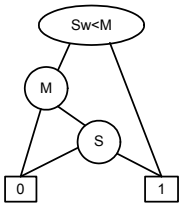
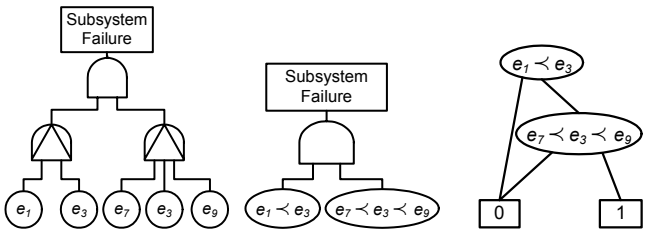


Figure 5. SBDD model of the transformed DFT

Step 3: Evaluation of the system SBDD model: The system unreliability can be calculated as the sum of probabilities for all the disjoint paths from the root to the sink node '1' in the

system SBDD. Specifically, all the paths to the sink node are enumerated and their probabilities are evaluated. Consider the SBDD of Figure 5, there are two paths to the sink node '1': $(Sw \prec M) \rightarrow$ '1' and $(Sw \prec M)' \rightarrow (M) \rightarrow (S) \rightarrow$'1'. Thus, $\Pr\{\text{System Failure}\} = \Pr\{Sw \prec M\} + \Pr\{(Sw \prec M)'.(M.S)\}$. This is a simple case where each path involves at most one sequential event. For some cases, we may obtain a path that involves more than one sequential event. For example, Figure 6(a) shows the DFT model of a subsystem with two pAND gates, each representing a sequential event. Figure 6(b) shows the transformed DFT model. Figure 6(c) is the SBDD model generated from the transformed DFT model.



(a) DFT *(b)Transformed* *(c) SBDD*

Figure 6. An example with more than one sequential event

In Figure 6(c), there is one path to the sink node '1': $(e_1 \prec e_3) \rightarrow (e_7 \prec e_3 \prec e_9) \rightarrow$'1'. And the two sequential events involved in this path are not independent because they share the same event e_3. For such cases, when we calculate the path probability, we must generate the complete sequences from the presented partial sequences for considering the dependence between them. For example, to calculate $\Pr\{(e_1 \prec e_3).(e_7 \prec e_3 \prec e_9)\}$, we must expand the partial sequences $(e_1 \prec e_3)$ and $(e_7 \prec e_3 \prec e_9)$ into complete sequences over all the four basic events as $(e_1 \prec e_7 \prec e_3 \prec e_9)$ and $(e_7 \prec e_1 \prec e_3 \prec e_9)$. The resultant complete sequences must preserve all the ordering constraints imposed by the input partial sequences. Because the two complete sequence events are mutually exclusive, we have: $\Pr\{(e_1 \prec e_3).(e_7 \prec e_3 \prec e_9)\} = \Pr\{e_1 \prec e_7 \prec e_3 \prec e_9\} + \Pr\{e_7 \prec e_1 \prec e_3 \prec e_9\}$. The focus of this paper is to propose an algorithm to implement the generation of the complete orders/sequences from the partial orders/sequences, which is a necessary and significant task for accomplishing the reliability analysis of sequence-dependent systems using the approach of [9].

After generating the list of complete sequences, the probability of the ordered failures can be computed using the following formula based on the basic probability theory on distribution functions:

$$\Pr\{X_1 \prec X_2 \prec \ldots \prec X_n\}$$

$$= \int_0^T \ldots \left[\int_0^{\tau_3} \left[\int_0^{\tau_2} \left[\int_0^{\tau_1} f_1(t)dt \right] f_2(\tau_1)d\tau_1 \right] f_3(\tau_2)d\tau_2 \right] \ldots f_n(\tau_{n-1})d\tau_{n-1}$$

where $X_1 \prec X_2 \prec \ldots \prec X_n$ is an event representing the sequential failures of n components, and $f_i(t)$ is the probability density function (*pdf*) of the random variable (*r.v.*) T_i representing the time-to-failure of component X_i.

Applying the above formula to our example system, $\Pr\{Sw \prec M\}$ can be evaluated as: $\Pr\{Sw \prec M\} = \int_0^T \left[\int_0^{\tau} f_{sw}(t)dt \right] f_m(\tau)d\tau$. When both the switch and the primary unit fail exponentially with constant rates of λ_{sw} and λ_m, respectively, we have

$$\Pr\{Sw \prec M\} = \int_0^T \left[\int_0^{\tau} \lambda_{sw} e^{\lambda_{sw} t} dt \right] \lambda_m e^{\lambda_m \tau} d\tau$$

$$= (1 - e^{-\lambda_m T}) - \frac{\lambda_m}{\lambda_m + \lambda_{sw}} (1 - e^{-(\lambda_m + \lambda_{sw})T})$$

Because $(Sw \prec M)' = Sw'M' + Sw'M + SwM' + (M \prec Sw) = Sw' + SwM' + (M \prec Sw)$ and $\Pr\{(Sw \prec M).(M.S)\} = \Pr\{(Sw \prec M).\Pr(S)\}$, we can calculate the unreliability of the example standby sparing system as:

$$\Pr\{Sw \prec M\} + \Pr\{M.S\} - \Pr\{Sw \prec M\}.\Pr\{S\}$$

$$= (1 - e^{-\lambda_m T}) - \frac{\lambda_m}{\lambda_m + \lambda_{sw}} (1 - e^{-(\lambda_m + \lambda_{sw})T}) + (1 - e^{-\lambda_m T})(1 - e^{-\lambda_s T}) - \left[(1 - e^{-\lambda_m T}) - \frac{\lambda_m}{\lambda_m + \lambda_{sw}} (1 - e^{-(\lambda_m + \lambda_{sw})T}) \right] (1 - e^{-\lambda_s T})$$

$$= (1 - e^{-\lambda_m T}) - \frac{\lambda_m}{(\lambda_m + \lambda_{sw})} (1 - e^{-(\lambda_m + \lambda_{sw})T}) e^{-\lambda_s T}$$

III. Proposed Generation Algorithm

The proposed complete sequence generation algorithm is based on topological sort [14], which is a method of arranging vertices in a directed acyclic graph (DAG) as a sequence such that no vertex appears in the sequence before its predecessor. For a DAG, we define **in-degree** of a vertex as the number of edges/arrows going into the vertex and **out-degree** as the number of edges/arrows coming out of the vertex. In the context of precedence constraints, the in-degree refers to the number of predecessors of a vertex and the out-degree refers to the number of successors of the vertex. Next, we describe the proposed algorithm as a five-step procedure.

Generation Algorithm:

1. Initialization: set up an array R that records the in-degree value of each vertex in a DAG. Initially, the in-degree values are all set to zero.
2. Update the in-degree of each vertex according to partial sequences. Specifically, search each partial sequence. Except the first vertex in the sequence, for each of the remaining vertex appearing in the sequence, increase its in-degree by 1 and update the array R.
3. Let Q be the queue used to keep track of vertices with in-degree of zero. If Q contains more than one element with in-degree of zero, then split Q into $n!$ queues Q_i, where n is the number of elements with in-degree of zero in Q. Each Q_i contains a distinct permutation of those n elements.

4. For each Q_i, define $R_i = R$. As long as Q_i is not empty, do the following:
   a. Visit each vertex of the queue and move it to an array F_i.
   b. Delete that vertex from the array R_i.
   c. If the vertex has a successor, then decrease the in-degree of its successor and update array R_i.
   d. If the in-degree of that successor becomes zero then keep that successor at the end of queue Q_i. If Q_i contains more than one element with in-degree of zero, then do the following:
      - Split Q_i into n_i! queues Q_{ij}, where j = 1, ..., n_i! and n_i is the number of elements with in-degree of zero in Q_i. Each Q_{ij} contains a distinct permutation of those n_i elements.
      - Set up an array F_{ij} for each Q_{ij}, and initialize it to be the current F_i. Then delete F_i.
      - Also, set up an array R_{ij} for each Q_{ij}, and initialize it to be the current R_i. Then delete R_i.
      - Go back to 4(a).

   Note that when iteration in Step 4 is performed for Q_{ij}, then corresponding arrays F_{ij} and R_{ij} will be used in those four sub-steps. In addition, it is possible that Q_{ij} can be split further into Q_{ijk}. Thus, corresponding arrays F_{ijk} and R_{ijk} will be set up and used in the subsequent operations. Similar split can be done further for Q_{ijk}.
5. Output all the arrays F, each corresponding to a complete sequence.

IV. Illustrative Examples

To illustrate the proposed algorithm presented in Section III, we consider the generation of complete sequences for the following two partial sequences: ($e_1 \prec e_3$) and ($e_7 \prec e_3 \prec e_9$).

Step 1: Set up the array R.

R [ ] = {in(e_1)=0, in(e_3)=0, in(e_7)=0, in(e_9)=0}

Step 2: Update the in-degree of each vertex according to partial sequences. For ($e_1 \prec e_3$), we have: in(e_3)=0+1=1. For ($e_7 \prec e_3 \prec e_9$), we have: in(e_3)=1+1=2, and in(e_9)=0+1=1. The updated array is: R [ ] = {in(e_1)=0, in(e_3) = 2, in(e_7)= 0; in(e_9)=1}

Step 3: Set up Q to contain all vertices with in-degree of zero and do the split. Q = (e_1, e_7) will be split into 2!=2 queues: Q_1 and Q_2, where Q_1 = {$e_1 \prec e_7$} and Q_2 = {$e_7 \prec e_1$}.

Step 4: For each Q_i, as long as Q_i is not empty, do the following:

For Q_1:

R_1 [ ] = R [ ] = {in(e_1)=0, in(e_3) = 2, in(e_7)= 0; in(e_9)=1}

Loop 1: Visit e_1.

a. F_1 [ ] = {e_1} and Q_1 = {e_7}

b. Delete e_1 from the array R_1.
R_1 [ ] = {in(e_3) = 2; in(e_7)= 0; in(e_9)=1}

c. If it has successor then decrease the in-degree of its successor and update R_1.
The successor of e1 is e_3. Hence, in(e_3) =2-1=1.
Therefore, the updated R_1 [ ] = {in(e_3) = 1; in(e_7)= 0; in(e_9)=1}

d. If the in-degree of that successor is zero then keep that successor at the end of queue Q_1.
Here in(e_3)!= 0.

Loop 2: Visit e_7.

a. F_1 [ ] = {e_1, e_7} and Q_1 = {}

b. Delete e_7 from the array R_1. R_1 [ ] = {in(e_3) = 1; in(e_9)=1}

c. If it has successor then decrease the in-degree of its successor and update R_1 [ ]. The successor of e_7 is e_3. Hence, in(e_3) =1-1=0. Therefore, the updated R_1 [ ] = {in(e_3) = 0; in(e_9)=1}

d. If the in-degree of that successor is zero then keep that successor at the end of queue Q_1. Here in(e_3)=0. Hence, Q_1 = {e_3}

Loop 3: Visit e_3.

a. F_1 [ ] = {e_1, e_7, e_3} and Q_1 = {}

b. Delete e_3 from the array R_1. R_1 [ ] = {in(e_9)=1}

c. If it has successor then decrease the in-degree of its successor and update R_1 [ ]. The successor of e_3 is e_9. Hence, in(e_9) =1-1=0. Therefore, the updated R_1 [ ] = {in(e_9)=0}

d. If the in-degree of that successor is zero then keep that successor at the end of queue Q_1. Here in(e_9)=0. Hence, Q_1= {e_9}

Loop 4: Visit e_9.

a. F_1 [ ] = {e_1, e_7, e_3, e_9} and Q_1 = {}

b. Delete e_9 from the array R_1. R_1 [ ] = { }

c. If it has successor then decrease the in-degree of its successor and update R_1 [ ]. No successor.

d. If the in-degree of that successor is zero then keep that successor at the end of queue Q_1.

Loop ends for Q_1 because Q_1 is empty.

For Q_2:

R_2 [ ] = R [ ] = {in(e_1)=0, in(e_3) = 2, in(e_7)= 0; in(e_9)=1}

Loop 1: Visit e_7.

a. F_2 [ ] = {e_7}, and Q_2 = {e_1}

b. Delete e_7 from the array R_2. R_2 [ ] = {in(e_3) = 2; in(e_1)= 0; in(e_9)=1}

c. If it has successor then decrease the in-degree of its successor and update R_2 [ ]. The successor of e_7 is e_3. Hence, in(e_3) =2-1=1. Therefore, the updated R_2 [ ] = {in(e_3) = 1; in(e_1)= 0; in(e_9)=1}

d. If the in-degree of that successor is zero then keep that successor at the end of queue Q_2. Here in(e_3)!=0.

Loop 2: Visit e_1.

a. F_2 [ ] = {e_7, e_1} and Q_2 = {}

b. Delete e_1 from the array R_2. R_2 [ ] = {in(e_3) = 1; in(e_9)=1}

c. If it has successor then decrease the in-degree of its successor and update R_2 [ ]. The successor of e1 is e_3. Hence, in(e_3) =1-1=0. Therefore, the updated R_2 [ ] = {in(e_3) = 0; in(e_9)=1}

d. If the in-degree of that successor is zero then again keep that successor at the end of queue Q_2. Here in(e_3)=0. Hence, Q_2= {e_3}

Loop 3: Visit e_3.

a. F_2 [ ] = {e_7, e_1, e_3} and Q_2 = {}

b. Delete e_3 from the array R_2. R_2 [ ] = {in(e_9)=1}

c. If it has successor then decrease the in-degree of its successor and update R_2 [ ]. The successor of e_3 is e_9. Hence, in(e_9) =1-1=0. Therefore, the updated R_2 [ ] = {in(e_9)=0}

d. If the in-degree of that successor is zero then again keep that successor at the end of queue Q_2. Here in(e_9)=0. Hence, $Q_2 = \{e_9\}$

Loop 4: Visit e_9.

a. $F_2[\ ] = \{e_7, e_1, e_3, e_9\}$ and $Q_2 = \{\}$

b. Delete e_9 from the array R_2. $R_2[\ ] = \{\ \}$

c. If it has successor then decrease the in-degree of its successor and update $R_2[\ ]$. No successor.

d. If the in-degree of that successor is zero then again keep that successor at the end of queue Q_2.

Loop ends for Q_2 because Q_2 is empty.

Step 5: Output F_i.

$F_1[\ ] = \{e_1, e_7, e_3, e_9\}$ $F_2[\ ] = \{e_7, e_1, e_3, e_9\}$

In the following, we also show the generated complete sequences for several more complex examples by applying the proposed generation algorithm.

1. $(e_2 \prec e_4)(e_7 \prec e_2 \prec e_1)(e_5 \prec e_2) = \{e_7 \prec e_5 \prec e_2 \prec e_4 \prec e_1\} + \{e_7 \prec e_5 \prec e_2 \prec e_1 \prec e_4\} + \{e_5 \prec e_7 \prec e_2 \prec e_4 \prec e_1\} + \{e_5 \prec e_7 \prec e_2 \prec e_1 \prec e_4\}$

2. $(e_2 \prec e_4)(e_7 \prec e_4 \prec e_1)(e_7 \prec e_6) = \{e_2 \prec e_7 \prec e_6 \prec e_4 \prec e_1\} + \{e_2 \prec e_7 \prec e_4 \prec e_1 \prec e_6\} + \{e_2 \prec e_7 \prec e_4 \prec e_6 \prec e_1\} + \{e_7 \prec e_6 \prec e_2 \prec e_4 \prec e_1\} + \{e_7 \prec e_2 \prec e_6 \prec e_4 \prec e_1\} + \{e_7 \prec e_2 \prec e_4 \prec e_1 \prec e_6\} + \{e_7 \prec e_2 \prec e_4 \prec e_6 \prec e_1\}$

3. $(e_1 \prec e_3)(e_1 \prec e_8)(e_7 \prec e_3 \prec e_9) = \{e_1 \prec e_8 \prec e_7 \prec e_3 \prec e_9\} + \{e_1 \prec e_7 \prec e_8 \prec e_3 \prec e_9\} + \{e_1 \prec e_7 \prec e_3 \prec e_8 \prec e_9\} + \{e_1 \prec e_7 \prec e_3 \prec e_9 \prec e_8\} + \{e_7 \prec e_1 \prec e_8 \prec e_3 \prec e_9\} + \{e_7 \prec e_1 \prec e_3 \prec e_9 \prec e_8\} + \{e_7 \prec e_1 \prec e_3 \prec e_8 \prec e_9\}$

4. $(e_1 \prec e_5)(e_6 \prec e_5 \prec e_7)(e_5 \prec e_2) = \{e_1 \prec e_6 \prec e_5 \prec e_7 \prec e_2\} + \{e_1 \prec e_6 \prec e_5 \prec e_2 \prec e_7\} + \{e_6 \prec e_1 \prec e_5 \prec e_7 \prec e_2\} + \{e_6 \prec e_1 \prec e_5 \prec e_2 \prec e_7\}$

5. $(e_1 \prec e_3)(e_2 \prec e_1)(e_2 \prec e_4)(e_4 \prec e_3)(e_4 \prec e_5) = \{e_2 \prec e_1 \prec e_4 \prec e_3 \prec e_5\} + \{e_2 \prec e_1 \prec e_4 \prec e_5 \prec e_3\} + \{e_2 \prec e_4 \prec e_5 \prec e_1 \prec e_3\} + \{e_2 \prec e_4 \prec e_1 \prec e_3 \prec e_5\} + \{e_2 \prec e_4 \prec e_1 \prec e_5 \prec e_3\}$

6. $(e_1 \prec e_9)(e_6 \prec e_1 \prec e_8)(e_7 \prec e_6 \prec e_{10})(e_{10} \prec e_2) = \{e_7 \prec e_6 \prec e_1 \prec e_8 \prec e_9 \prec e_{10} \prec e_2\} + \{e_7 \prec e_6 \prec e_1 \prec e_8 \prec e_{10} \prec e_2 \prec e_9\} + \{e_7 \prec e_6 \prec e_1 \prec e_8 \prec e_{10} \prec e_9 \prec e_2\} + \{e_7 \prec e_6 \prec e_1 \prec e_9 \prec e_{10} \prec e_2 \prec e_8\} + \{e_7 \prec e_6 \prec e_1 \prec e_9 \prec e_{10} \prec e_8 \prec e_2\} + \{e_7 \prec e_6 \prec e_1 \prec e_9 \prec e_8 \prec e_{10} \prec e_2\} + \{e_7 \prec e_6 \prec e_1 \prec e_{10} \prec e_2 \prec e_8 \prec e_9\} + \{e_7 \prec e_6 \prec e_1 \prec e_{10} \prec e_8 \prec e_2 \prec e_9\} + \{e_7 \prec e_6 \prec e_1 \prec e_{10} \prec e_9 \prec e_2 \prec e_8\} + \{e_7 \prec e_6 \prec e_1 \prec e_{10} \prec e_2 \prec e_9 \prec e_8\} + \{e_7 \prec e_6 \prec e_1 \prec e_{10} \prec e_8 \prec e_9 \prec e_2\} + \{e_7 \prec e_6 \prec e_1 \prec e_{10} \prec e_9 \prec e_8 \prec e_2\}$

V. Conclusions & Future Work

The generation of complete sequences from partial sequences has been identified as one of the essential steps in the combinatorial reliability analysis of dynamic systems with sequence-dependent behavior modeled using pAND gates. In this paper, a topological sort-based algorithm was proposed to generate complete sequences of elements that satisfy all the precedence constraints posed by the input partial sequences. The algorithm involves complex queue split process. In our future work, we will explore more efficient complete sequence generation algorithm and integrate it into the efficient SBDD evaluation procedure for dynamic systems subject to various dependent behaviors. Complex case studies will be performed to verify the performance of the proposed algorithm.

Acknowledgment

This work is partly supported by the US National Science Foundation under Grant No. 0832594.

References

[1] J. B. Dugan and S. A. Doyle, "New results in fault-tree analysis," *Tutorial notes of the Annual Reliability & Maintainability Symposium*, Jan. 1997.

[2] K. B. Misra (Editor), *Handbook of Performability Engineering*, Springer-Verlag, London, ISBN: 978-1-84800-130-5, Oct. 2008.

[3] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems," *IEEE Transactions on Reliability*, 41(3): 363-377, 1992.

[4] J. B. Fussell, E. F. Aber, and R. G. Rahl, "On the quantitative analysis of priority-AND failure logic", *IEEE Transactions on Reliability*, R-25: 324-326, 1976.

[5] R. Gulati and J. B. Dugan, "A modular approach for analyzing static and dynamic fault trees," *Proceedings of the Annual Reliability & Maintainability Symposium*, Philadelphia, PA. pp. 568-573, Jan. 1997.

[6] S. V. Amari, G. Dill, and E. Howald, "A new approach to solve dynamic fault trees," *Proceedings of Annual Reliability & Maintainability Symposium*, pp. 374-379, Jan. 2003.

[7] R. Bryant, "Graph based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, C-35(8): 677-691, Aug. 1986.

[8] L. Xing and J. B. Dugan, "Analysis of generalized phased mission system reliability, performance and sensitivity," *IEEE Transactions on Reliability*, 51(2): 199-211, Jun. 2002

[9] L. Xing, A. Shrestha, and Y. Dai, "Exact combinatorial analysis of dynamic systems with sequence-dependent failures," *IEEE Transactions on Dependable and Secure Computing* (submitted).

[10] W. Long, T. L. Zhang, Y. F. Lu, and M. Oshima, "On the quantitative analysis of sequential failure logic using Monte Carlo method for different distributions," *Proceedings of Probabilistic Safety Assessment and Management*, pp. 391-396, 2002.

[11] H. Boudali and J. B. Dugan, "A discrete-time Bayesian network reliability modeling and analysis framework," *Reliability Engineering & System Safety*, 87(3): 337-349, Mar. 2005.

[12] T. Yuge and S. Yanagi, "Quantitative analysis of a fault tree with priority AND gates," *Reliability Engineering & System Safety*, 93(11): 1577-1583, Nov. 2008.

[13] D. Liu, C. Zhang, W. Xing, R. Li, and H. Li, "Quantification of cut sequence set for fault tree analysis," *HPCC2007, Lecture Notes in Computer Science*, no. 4782, pp. 755-765, Springer-Verlag, 2007.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (2nd Edition). The MIT Press, 2001.