

# Hierarchical Analysis of Fault Trees with Dependencies, using Decomposition

Anju Anand • The Boeing Company • Seattle  
Arun K. Somani • Iowa State University • Ames

Keywords: Fault Trees, Markov Chain, Decomposition, Hierarchical Analysis, Reliability analysis

## SUMMARY & CONCLUSIONS

Analytical methods for fault tree analysis involve the determination of the minimum cut sets, whose number grows extremely quickly with the number of events. This problem can be solved by utilizing the methods of decomposition and aggregation. That is, dividing the system into smaller subsystems, solving the subsystems separately, and then synthesizing these results to produce the larger systems' solution. In this paper, we demonstrate a decomposition scheme where independent subtrees of a fault tree are detected and solved hierarchically, a subtree is replaced by a single event in the parent tree whose probability of occurrence represents the probability of the occurrence of the subtree. The decomposition and hierarchical solution can be more useful in case of fault trees with dependencies. Instead of solving the whole system as a Markov model only the appropriate subsystem needs to be analyzed as a Markov model.

## 1 INTRODUCTION

Engineers need to predict system reliability in the course of designing a system, especially in case of critical applications. The use of mathematical models such as fault trees provides a convenient and effective method for determining such dependability measures.

A fault tree [1] is a mathematical representation of a combination of events that leads to a system failure. The quantitative analysis of a fault tree is used to determine the probability of system failure, given the probability of occurrence of events. Analysis of large fault trees can be computationally expensive, despite recent work on BDD [2]. Modularization is one of the ways to tackle the analysis of large fault trees. A module of a fault tree is a subtree whose events do not occur elsewhere in the fault tree. Various algorithms have been proposed to modularize fault trees eg. [3], [4], [5].

Though relatively efficient algorithms exist for solving fault trees, the main disadvantage is that dependencies of

various kinds that do occur in practice are not easily captured by the fault tree models. In contrast, Markov models are powerful in representing various kinds of dependencies, though the size of a Markov chain grows exponentially with the number of components in a system. To exploit the relative advantages of both fault trees and Markov models, while avoiding many of their short-comings, tools such as HARP [6], E-HARP [7] and HIMAP [8] have introduced several additional gates into the fault tree model to capture various dependencies. This fault tree model (with additional gates) and possible repair specification of components, is converted to its equivalent Markov model for analysis. The resulting Markov model can be solved using standard numerical techniques.

The advantage of allowing a fault tree description of the system is that the modeler need not perform the tedious task of generating the Markov chain representation of a system. However, the Markov chain can be huge in size and a solution can take a lot of time. Both generation and the analysis of a Markov chain of the whole system can be computationally very expensive.

In this paper, we describe a decomposition method implemented in HIMAP (Hierarchical Modeling and analysis Package) [8], where a system, specified as a fault tree model, is decomposed into independent subtrees automatically. The subtrees are then solved either as fault trees or as equivalent Markov chains. A solution for fault trees with repairable components is also provided as repairs introduce dependencies. This hierarchical analysis technique allows the use of Markov models where necessary and retains the efficiency of combinatorial solutions where possible.

Here is how material is organized in this report. Section 2 explains the different modeling types and the solvers as used in HIMAP. In Section 3, we explain the decomposition algorithm with an example. Section 4 gives details of how the decomposition is implemented and the factors which influence the decision. Section 5 covers the details of analysis of systems with repairable components.

## 2 TYPES

Over the years, several model types such as fault trees, Markov chains and Petri nets, have been used to model fault-tolerant systems and to evaluate various dependability measures. These model types differ from each other not only in the ease of use in a particular application, but also in terms of modeling power.

### 2.1 Fault Tree Model

Fault trees are typically constructed by starting with the top event (usually representing some undesirable situation) and determining all possible ways to reach that event. Basic events are at the lowest level of the fault tree, and different combinations of basic events result in the top event. A failure rate (or a probability of occurrence) is assigned to each basic event. Logic gates delineate the causal relations which ultimately result in the top event. HIMAP supports the following logic gates to describe a system:

- **AND** The output occurs when all inputs occur simultaneously.
- **OR** The output event occurs when one or more the input events occur.
- **K/N** The output occurs when K of N input events occur.

The additional dependency gates that are supported in HIMAP are :

- **Functional Dependency gate** Allows modeling of cases where the occurrence of some event (call it trigger event) causes other dependent components to become inaccessible or unusable.
- **Cold spare gate** Allows the modeling of spare components that are unpowered (and thus do not fail before being used). The output occurs when all the inputs occur but the occurrence of the events is as specified by the modeler.
- **Priority-AND gate** A 2-input AND gate, with the added condition that events must occur in a specific order for output to occur.
- **Sequence enforcing gate** Forces events to occur in a particular order. The output occurs when all the inputs occur but the occurrence of the events is as specified by the modeler.

### 2.2 Markov Model

Fault tree models have limitations in capturing dependencies that may exist in systems. The Markov model allows modeling of any dynamic system and its dependencies. The Markov model is a state-transition diagram in which each state depicts a particular operational configuration of the system. Transitions between states signify

components failing or being repaired. The construction of a Markov model for any system can be tedious and error-prone. A more efficient technique is to describe system behavior using fault trees in conjunction with other descriptors, and then convert that description to a Markov chain for analysis. The expressive power of a fault tree can be expanded by allowing more gates to capture specific types of dependent behaviors. HARP, E-HARP and HIMAP include special purpose gates (as described in the previous section) in the description of fault tree to model dynamic behavior. The modeler can use the parsimony of the fault tree representation of the system to generate the state space of the Markov chain automatically; and then make adjustments to the Markov chain as needed.

Both model types have their advantages and disadvantages. Fault trees are more intuitive in capturing how a component failure propagates into a higher level system or system failure whereas a Markov chain is best suited to model dependencies.

## 3 THE MODELING TYPES

Over the years, several model types such as fault trees, Markov chains and Petri nets, have been used to model fault-tolerant systems and to evaluate various dependability measures. These model types differ from each other not only in the ease of use in a particular application, but also in terms of modeling power.

### 3.1 Fault Tree Model

Fault trees are typically constructed by starting with the top event (usually representing some undesirable situation) and determining all possible ways to reach that event. Basic events are at the lowest level of the fault tree, and different combinations of basic events result in the top event. A failure rate (or a probability of occurrence) is assigned to each basic event. Logic gates delineate the causal relations which ultimately result in the top event. HIMAP supports the following logic gates to describe a system:

- **AND** The output occurs when all inputs occur simultaneously.
- **OR** The output event occurs when one or more the input events occur.
- **K/N** The output occurs when K of N input events occur.

The additional dependency gates that are supported in HIMAP are :

- **Functional Dependency gate** Allows modeling of cases where the occurrence of some event (call it trigger event) causes other dependent components to become inaccessible or unusable.

- **Cold spare gate** Allows the modeling of spare components that are unpowered (and thus do not fail before being used). The output occurs when all the inputs occur but the occurrence of the events is as specified by the modeler.
- **Priority-AND gate** A 2-input AND gate, with the added condition that events must occur in a specific order for output to occur.
- **Sequence enforcing gate** Forces events to occur in a particular order. The output occurs when all the inputs occur but the occurrence of the events is as specified by the modeler.

### 3.2 Markov Model

Fault tree models have limitations in capturing dependencies that may exist in systems. The Markov model allows modeling of any dynamic system and its dependencies. The Markov model is a state-transition diagram in which each state depicts a particular operational configuration of the system. Transitions between states signify components failing or being repaired. The construction of a Markov model for any system can be tedious and error-prone. A more efficient technique is to describe system behavior using fault trees in conjunction with other descriptors, and then convert that description to a Markov chain for analysis. The expressive power of a fault tree can be expanded by allowing more gates to capture specific types of dependent behaviors. HARP, E-HARP and HIMAP include special purpose gates (as described in the previous section) in the description of fault tree to model dynamic behavior. The modeler can use the parsimony of the fault tree representation of the system to generate the state space of the Markov chain automatically; and then make adjustments to the Markov chain as needed.

Both model types have their advantages and disadvantages. Fault trees are more intuitive in capturing how a component failure propagates into a higher level system or system failure whereas a Markov chain is best suited to model dependencies.

## 4 DECOMPOSITION

In [10], it is shown that the problem of computing (or even approximating) the probability of arbitrary fault trees is an NP-hard problem. NP-hard problems never possess algorithms which guarantee to finish in a number of steps bounded by any polynomial in the size of the problem (the number of events in our case). One of the ways to tackle the problem of computing the probability of fault trees is to detect modules and treat these modules as other independent fault trees. When analyzing fault trees, such modules can be treated as equivalent to a basic event in the tree. The effective size of the fault tree is determined by the number of events after all the modules have been

replaced; which will be much less than the original size. This reduces the computation time.

A module of a fault tree is a subtree composed of events which do not occur elsewhere in the tree i.e. the subtree does not have any inputs from the rest of the tree. Several methods have been proposed to modularize fault trees. General techniques to achieve a linear complexity are of great interest. We have implemented a Linear-Time algorithm proposed in [11] to find modules of a fault tree. The algorithm is derived from the Tarjan algorithm [12] to detect strongly connected components of a graph.

```

DFLM( node) {
    visit(node);
    if(node not marked) {
        for(all children of node)
            DFLM(child);
        visit(node);
        mark the node;
    }
}

```

Figure 1: The DFLM traversal algorithm

A fault tree is a directed acyclic graph whose nodes are either basic events or logical gates. From the definition, the root and the basic events are always modules. A depth-first left-most (DFLM) traversal of the tree is implemented to find the modules of a fault tree. The algorithm for DFLM traversal is implemented as shown in figure 1.

### 4.1 Example of DFLM

A DFLM traversal of the tree in figure 2 would visit the nodes in the following order:

(Top, g1, e1, g4, e3, e4, g4, g1, g2, g4, e2, g2, g3, e2, e5, g3, Top)

Some observations can be made from this traversal:

1. Each interior node is visited at least twice; the first time when descending from its parent, the second time when going back from its right-most child.
2. Leaves are traversed only once.
3. The graph under a vertex is never traversed twice.

1, 2, 3 show that each edge is traversed exactly twice; thus a DFLM traversal of the graph is **linear in the number of edges**.

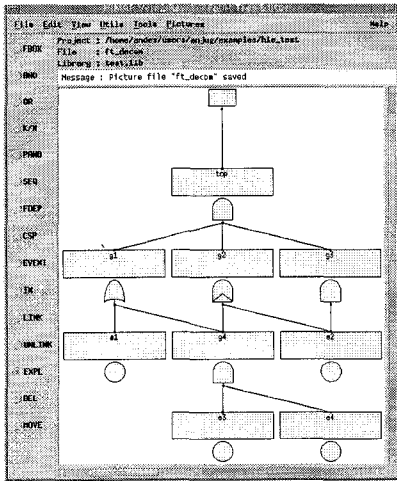


Figure 2: The Fault Tree model

The decomposition algorithm is based on the principle that a node is a module iff none of its descendants are visited before the node's first visit or after the node's second visit.

From the above example,

1. g1 is not a module because g4 is visited after g1's second visit.
2. g2 is not a module because g4 is visited before g2's first visit.
3. g3 is not a module because e2 is visited before g3's first visit.
4. g4 is a module because its descendants e3 and e4 are visited only after g4's first visit and before g4's second visit.

As can be seen from this example, in order to get the modules of a tree it is needed to figure out if any of the descendant of the node has been visited before the node's first visit or after the node's second visit. This requires

- Defining what is a visit.
- Keeping track of the different visits (first visit, second visit and last visit) of every node.
- Somehow get the maximum and the minimum of the visits of all the descendants of a node which then can be compared to see if a node is a module or not.

#### 4.2 Data Structure

In order to implement the algorithm, we needed

1. A variable which keep the visit number, it is stored in *counter*.

2. An array to mark the node visited, stored in *mark*.
3. 3 variables per node - *first\_visit*, *second\_visit* and the *last\_visit*.
4. 2 more variables per node - *max\_visit* and *min\_visit* to store the minimum and the maximum of the visits of a node's descendants.

#### 4.3 Algorithm

The algorithm is described below:

- Initialize all the variables to zero.
- Perform the first DFLM traversal of the graph to set the variables (*first\_visit*, *second\_visit* and *last\_visit*). The variable *counter* is incremented whenever a node is visited. The variables *first\_visit*, *second\_visit* and the *last\_visit* are nothing but the value of the *counter* at the time of the visit.
- Perform the second DFLM traversal of the graph where for each node, collect the minimum (in *min\_visit*) of the *first\_visit* and the maximum (in *max\_visit*) of the *last\_visit* of its children.
- A node M is a module iff
  - a) the collected minimum (*min\_visit* is greater than the *first\_visit* of M AND
  - b) the collected maximum *max\_visit* is less than the *second\_visit* of M.

#### 4.4 Example

Here we demonstrate the algorithm of finding modules for a system shown in figure 2.

The counter value for this example is shown in Table 1.

Table 1: The counter value in DFLM traversal

counter	1	2	3	4	5	6	7	8
nodes	Top	g1	e1	g4	e3	e4	g4	g1
9	10	11	12	13	14	15	16	17
g2	g4	e2	g2	g3	e2	e5	g3	Top

After the first DFLM traversal of the tree, the variables (*first\_visit*, *second\_visit* and the *last\_visit* for all nodes will be set as shown in Table 2.

Minimum of a node is the minimum of the *first\_visit* of its descendants and maximum is the maximum of the *last\_visit*. After the second DFLM traversal of the tree, the minimum and the maximum will be set as shown in Table 3.

A node is a module if the minimum is greater than its *first\_visit* and maximum is less than the *second\_visit*. From the two tables, it can be seen that Top and g4 are the only modules of this tree.

Table 2: Results of 1st traversal

	Top	g1	g2	g3	g4
<i>first_visit</i>	1	2	9	13	4
<i>second_visit</i>	17	8	12	16	7
<i>last_visit</i>	17	8	12	16	10
	e1	e2	e3	e4	e5
<i>first_visit</i>	3	11	5	6	15
<i>second_visit</i>	3	14	5	6	15
<i>last_visit</i>	3	14	5	6	15

Table 3: Results of 2nd traversal

	Top	g1	g2	g3	g4
<i>min_visit</i>	2	3	4	11	5
<i>max_visit</i>	16	10	14	15	6

## 5 IMPLEMENTATION

In this chapter, the actual decomposition process as implemented in HIMAP is explained. The process of decomposition is inherently recursive, as subtrees may themselves contain independent subtrees. The independent subtrees which are solved separately are reflected as a single basic event in the parent tree. This basic event in the parent tree represents the probability of occurrence of the subtree. For a fault tree model, the probability of the top event of the subtree is directly input to the parent tree as the probability of occurrence of the that basic event replacing the subtree. For a model which needs to be solved as a Markov model, the probability of reaching the failure state in the associated Markov chain is used for the probability of the basic event which replaces the subtree. The decomposition and hierarchical analysis is transparent to the modeler. When a system is modeled as a fault tree in HIMAP, before any analysis is performed, the model is studied to see if any decomposition is needed.

There are 2 parameters that decide whether the model needs to be decomposed.

### 1. The model has a dependency.

A module with dependency needs to be analyzed as a Markov model. A module has a dependency if any of the following criteria is met:

- there is a repairable component
- there is a recovery associated with a component
- there is a dependency gate

### 2. Number of components.

The number of components that mandate the decomposition of a system is different in the case of a system with dependencies as compared to one without dependencies.

## 5.1 Model Analysis

A fault tree model can be categorized into different types depending on the number of components.

### • Small model ( $\leq 10$ components)

For models as small as this, there is no need to decompose the system even if the model has dependencies. A equivalent Markov model of such a system will have a maximum of 1024 states which can easily be analyzed.

### • Medium Model ( $\leq 20$ component)

The analysis of a Markov model of approximately  $2^{20}$  states can be computationally very time consuming. So if the model has any dependency i.e. if the fault tree model needs to be converted to a Markov model for analysis, then the system is decomposed and analyzed hierarchically. On the other hand if the model can be solved through a fault tree solver then there is no need of decomposition.

### • Big Model ( $> 20$ components )

A system with more than 20 components is always decomposed. Even a fault tree model with more than 20 components can become too big to analyze as a single model.

The numbers (10 and 20) which decide the category has been chosen based on the solvers limits. Markov chain solver used for this implementation can easily solve a system of  $2^{10}$  states but anything more than this can be computationally expensive. The fault tree solver used, can analyze a system with less than 20 components easily.

After the fault tree has been decomposed into modules, each module is checked for dependency. Depending on whether there is a dependency or not, appropriate files as needed by a fault tree solver (in case of a fault tree solution) or Markov chain solver (in case of a Markov analysis) are generated. Both the solvers give us the probability of the failure of the module. The algorithm as implemented as shown in figure 3.

## 5.2 Approximation

The probability of top event occurrence in the module can be directly input as the probability of occurrence of the basic event replacing the module if the parent tree is solved as a fault tree model. This is an approximation if any of the module needs to be analyzed as a Markov model. Markov chain description requires the rate of occurrence of an event. The rate of occurrence of a basic event replacing a module is computed from the probability of occurrence of the top event in the module and mission time, assuming exponential distribution.

```

check for dependency;           1
if(dependency) {                2
  if(!Small model) {           3
    decompose the tree;         4
    for each module {          5
      check for dependency;     6
      if(dependency)           7
        solve as a Markov model; 8
      else                      9
        solve as a fault tree model; 10
    }                           11
  } else /* else of line 3 */   13
    solve as a single Markov model; 14
}
else { /* else of line 2 */     17
  if(Big Model) {              18
    decompose the tree;         19
    solve each module as a fault tree model; 20
  }                              21
  else /* else of line 18 */    22
    solve as a single fault tree model; 23
}                                24

```

Figure 3: The algorithm as implemented in HIMAP

### 5.3 Example

Here we will explain the procedure of hierarchical analysis using decomposition with an example. Consider a system with 12 components as shown in figure 4. The failure rate of each component is  $1D - 5$  failures per hour. According to the algorithm in figure 3, the fault tree is going to be checked for dependencies (line 1). The model has a dependency because of a sequence dependency gate (node g4). The next thing to be checked is the number of components in the model to decide if the model should be decomposed or not (line 3). The model falls under the category of *Medium Model*; a model under this category is decomposed if there is a dependency. The decomposition of this model will result into modules g8, g9, g4, g5, g2, g3, g1. Each module now will be checked for dependency (lines 5, 6). Modules g8, g9 will be solved as fault tree models as there is no dependency. The probability of top event of the fault tree will be taken as the probability of occurrence of the events replacing these modules. The next module in the list, g4, needs to be analyzed as a Markov model because the module has a sequence dependency gate. The events which are replacing the modules g8 and g7 require a rate of occurrence which is computed from their probability of occurrence and the mission time, assuming exponential distribution. The probability of being in the failure states in the equivalent Markov chain of g4 is computed, which is the probability of occurrence of the event replacing g4. The remaining modules in the list g5, g2, g3 and g1 are analyzed as fault tree models.

The same system, when solved without decomposition, i.e. as a single Markov chain produced 4174 states and took 118sec to execute. The probability of the failure was computed to be  $.10022513D-11$  which is very close to the value computed with decomposition ( $.10019842D-11$ ) in 10% of the time (12sec).

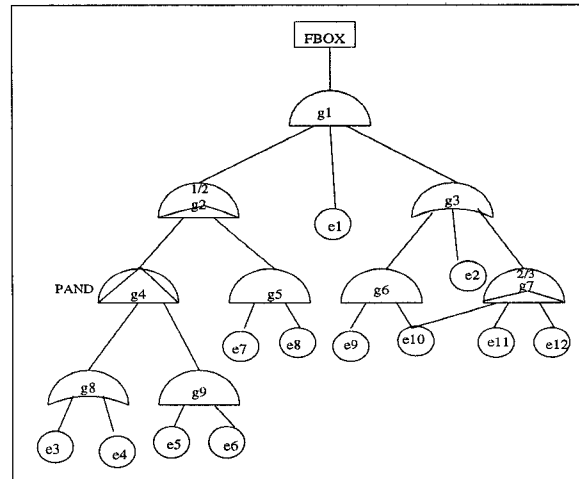


Figure 4: Example to show the decomposition

## 6 FAULT TREE WITH REPAIRABLE COMPONENTS

Since fault trees are inherently acyclic, they can not be used to model systems which have repairs. Due to implicit dependencies introduced by conditional repairs, reliability analysis of systems with repairable components requires the use of a state space model. State space description of a system can be very cumbersome and error prone. HIMAP tackles this problem in the following ways:

- The modeler can define a system as a fault tree. The fault tree is converted to its equivalent Markov chain and can be edited by the modeler to add repair arcs before any analysis is performed.
- When a system is modeled as a fault tree, the modeler can declare any repairable components in the system. The fault tree is converted to its equivalent Markov chain; Markov chain is then automatically edited by HIMAP to add the appropriate repair arcs, with the associated repair rates.

### 6.1 Repair Methodology

After a system has been decomposed into modules, each module is analyzed separately. A module which has a repairable component needs to be analyzed as a Markov model. There can be two approaches to analyze such a subsystem:

1. **Approach I** Assume the failure of this subsystem causes the failure of the whole system i.e. no repair arcs from failure state of the subsystem. This gives us a pessimistic reliability.
2. **Approach II** Assume the failure of this subsystem does not cause the failure of the whole system i.e.

there can be repair arcs from the failure state of the subsystem. This gives us an optimistic reliability.

In our implementation, we follow the first approach; the pessimistic one. In case of critical applications, it is always better to underestimate the system, than to overestimate it.

## 7 CONCLUSIONS

We have presented a new approach for solving fault trees models with dependencies. The approach is to decompose the fault tree into modules and solve these modules hierarchically. A module can be solved as a Markov model also. We have presented in this paper examples using our approach and compared with the results with full fault tree or full Markov analysis as the case may be. Our results indicate that the approach of hierarchical analysis using decomposition gives results very close to the exact values (exact values itself in most of the cases) in 1-10% of the time. An approach to solve fault trees with repairable components is also presented.

## References

- [1] W.S. Lee, D.L. Grosh, F.A. Tillman, C.H. Lie, "Fault Tree analysis, methods and applications: A review", IEEE Trans. Reliability, vol. R-34, 1985 Aug, pp. 194-302.
- [2] K. Brace, R. Rudell, R. Bryant, "Efficient implementation of a BDD package", Proc. 27th ACM/IEEE Design Automation Conf., 1990.
- [3] P. Chatterjee, "Modularization of fault trees: A method to reduce the cost of analysis", SIAM Reliability and Fault Tree Analysis, 1975, pp. 101-137.
- [4] A. Rosenthal, "Decomposition methods for fault tree analysis", IEEE Trans. Reliability, vol. R-29, 1980 Jun, pp. 136-138.
- [5] T. Khoda, E.J. Henley, K. Inoue, "Finding modules in fault trees", IEEE Trans. Reliability, vol. 38, 1989 Jun, pp. 165-176.
- [6] J. B. Dugan, K. S. Trivedi, M. K. Smotherman, and R. M. Geist, "The Hybrid Automated Reliability Predictor", AIAA Journal of Guidance, Control and Dynamics, vol. 9, no. 3, May-June 1986, pp. 319-331.
- [7] A. K. Somani, U. R. Sandadi, A. Gupta, P. C. Leung, "EHARP: Enhanced Hybrid Automated Reliability Predictor", Tech Report, DPCNL, University of Washington, Seattle, Dec 1993.
- [8] G. Krishnamurthi, A. Gupta, A. K. Somani, "The HIMAP Modeling Environment", Proc. of Parallel and Distributed Computing Systems, France, Sept 1996 pp. 254-260.
- [9] R.W. Butler, A.L. Martensen, "The Fault tree compiler (FTC)", NASA Tech. Paper 2915, 1989 July.
- [10] A. Rosenthal, "A computer scientist looks at reliability computations", SIAM Reliability and Fault Tree Analysis, 1975, pp. 133-152.

- [11] Y. Dutuit, A. Rauzy, "A linear-time algorithm to find modules of fault tree", IEEE Trans. Reliability, vol. 45, no. 3, 1996 Sept, pp. 422-425.
- [12] R.E. Tarjan, "Depth first search and linear graph algorithms", SIAM J. Computing, vol. 1, 1972, pp. 146-160.

## BIOGRAPHIES

Anand, Anju  
Reliability and Maintainability Engineering  
Boeing Commercial Airplane Group  
P.O. Box 3707, M/S 05-AM  
Seattle, WA 98124 USA  
*Internet (email):* anju.anand@pss.boeing.com

Anju Anand received B.E in Electrical and Electronics Engineering from BITS, India and M.S. in Electrical Engineering from University of Washington, Seattle in 1991 and 1997 respectively. Since August 1997 she has been working for Boeing Commercial Airplane Group as a reliability engineer involved in tools development for modeling and performance evaluation of fault tolerant systems.

Somani, Arun K.  
Professor, 223 Coover  
Department of Electrical and Computer Engineering  
Iowa State University  
Ames, IA 50011 USA  
*Internet (email):* arun@iastate.edu

Arun Somani earned his MSEE and Ph.D degrees in electrical engineering from the McGill University, Montreal, Canada, in 1983 and 1985, respectively. Prior to that, he worked as Scientific Officer for Govt. of India, New Delhi from 1974 to 1982. Arun Somani is a Professor in the Department of Electrical and Computer Engineering, Iowa State University, Ames. His research interests include fault tolerant computing, computer architecture, reliability analysis and parallel computer systems.