A Markovian Dependability Model with Cascading Failures

Srinivasan M. Iyer, Marvin K. Nakayama, and Alexandros V. Gerbessiotis

Abstract—We develop a continuous-time Markov chain model of a dependability system operating in a randomly changing environment and subject to probabilistic cascading failures. A cascading failure can be thought of as a rooted tree. The root is the component whose failure triggers the cascade, its children are those components that the root's failure immediately caused, the next generation are those components whose failures were immediately caused by the failures of the root's children, and so on. The amount of cascading is unlimited. We consider probabilistic cascading in the sense that the failure of a component of type i causes a component of type j to fail simultaneously with a given probability, with all failures in a cascade being mutually independent. Computing the infinitesimal generator matrix of the Markov chain poses significant challenges because of the exponential growth in the number of trees one needs to consider as the number of components failing in the cascade increases. We provide a recursive algorithm generating all possible trees corresponding to a given transition, along with an experimental study of an implementation of the algorithm on two examples. The numerical results highlight the effects of cascading on the dependability of the models.

Index Terms—Availability, reliability modeling, Markov processes, trees, cascading failures.

1 Introduction

Complex systems often suffer from cascading failures. Specifically, the failure of one component causes others to fail simultaneously; the secondary failures then lead to further instantaneous failures, and the cascading can continue for several iterations. For example, the failure of one computer in a network may cause others to also fail. Alternatively, in an electric power grid, a single failed power line can touch off a string of failures of other power lines and generators, leading to a blackout.

Mathematical modeling of systems suffering from cascading failures is crucial for their design and operation, and some previously developed stochastic models of this phenomenon include [1], [2], [3], and [4]. In the current paper, we consider a continuous-time Markov chain (CTMC) model of a dependability system operating in a randomly changing environment and having probabilistic cascading failures; e.g., see [5, Chapter 8] for details on CTMCs and [6] for CTMC solution techniques. The model includes an environment variable that changes as a CTMC, and this allows us to model randomly changing conditions under which the system operates. For example, the operating environment may randomly move among high-load, medium-load, and low-load conditions. The behavior of the system depends on the environment since we assume that the failure and repair rates of the components depend on the current environment. In addition, we allow for probabilistic cascading failures, where the failure of a component probabilistically causes other components to simultaneously fail with certain probabilities. Goyal and Lavenberg [7] and Goyal et al. [8] also employ a CTMC model to evaluate the dependability of fault-tolerant systems, but they only allow for a single level of cascading failures to occur. In other words, the failure of a single component might cause others to fail simultaneously, but the secondary failures cannot cause other simultaneous failures.

The current paper considers a model that allows for unlimited cascading failures, which significantly complicates the computation of the infinitesimal generator matrix Q of the CTMC. The difficulty arises from the exponential growth in the number of possible ways in which the cascading can occur as the number of failures in the cascade increases. For example, consider a system having three components, labeled 1, 2, and 3. Suppose that when any one of the components fail, it can cause each of the other components to fail simultaneously, with each successive failure having its own probability depending on which component caused the failure of the next component. In the resulting CTMC, consider the state x with all components operational and the state y with all components failed. To determine the transition rate of going from x to y, we need to consider the following nine possible ways of cascading: 1 first fails, then causing both 2 and 3 to fail together; 1 first fails, then 2 fails, which causes 3 to fail; 1 first fails, then 3 fails, which causes 2 to fail; 2 first fails, then causing both 1 and 3 to fail together; 2 first fails, then 1 fails, which causes 3 to fail; 2 first fails, then 3 fails, which causes 1 to fail; 3 first fails, then causing both 1 and 2 to fail together; 3 first fails, then 1 fails, which causes 2 to fail; 3 first fails, then 2 fails, which causes 1 to fail. The components may have unequal failure rates, and the probability of the failure of component i causing

- S.M. Iyer is with Exalead, Inc., 576 Folsom Street, 2nd Floor, San Francisco, CA 94105. E-mail: srinivasan.iyer@exalead.com.
- M.K. Nakayama and A.V. Gerbessiotis are with the Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102. E-mail: marvin@njit.edu, alexg@cs.njit.edu.

Manuscript received 1 Aug. 2007; revised 19 Dec. 2008; accepted 12 Jan. 2009; published online 11 Feb. 2009.
Recommended for acceptance by J. Lach.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-08-0401.
Digital Object Identifier no. 10.1109/TC.2009.31.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

component j to fail may differ from the probability of the failure of component j causing component i to fail. Hence, each of the nine possible ways in which a cascading failure occurs resulting in a transition from x to y has its own rate, and the sum of all of these is the overall rate of moving from state x to y. In general, for a failure transition in which m components fail, there can be up to m^{m-1} trees that need to be considered. This exponential growth is inherent to the problem, and constructing the Q matrix requires examining every tree.

We provide a recursive algorithm that determines all of the possible ways in which cascading can occur for a given set of failing components, which can then be used to compute the overall infinitesimal generator matrix of the CTMC. Once the infinitesimal generator matrix has been computed, we can then compute various dependability measures of the system. We have implemented a computer program that does exactly this. Our program takes as input a high-level description of the system and builds from it the CTMC. We present results from running our program on two examples to analyze the effects of cascading failures. The first is a modification of a fault-tolerant computing system previously considered in [8]. Our version of the model includes significantly more complex cascading failures, which required enumerating and evaluating approximately 200,000 trees to construct the generator matrix of the CTMC. The second model is an example of a search-engine infrastructure. Our numerical results demonstrate the significant degradation in dependability that can result from the presence of cascading failures, thereby highlighting the importance of modeling this phenomenon.

Dependability models have traditionally been classified [6] into two categories: non-state-space models and state-space models. The first do not require enumerating the entire state space of the underlying model, thereby often limiting the scope of possible interactions among components, such as shared repairmen or cascading failures. Solution methods for these models include reliability block diagrams (RBD), fault trees (FT), and reliability graphs (RG). State-space models, such as CTMCs as we consider, require enumeration of all possible combinations of states of components, which grows exponentially in the number of components in the system, but this allows for modeling much greater complexity in component interactions. Other previously developed software packages for computing dependability measures of Markovian models include SHARPE [9], SAVE [7], HARP [10], EHARP [11], SURE [12], TANGRAM [13], SURF-2 [14], and HIMAP [15]. Other packages use alternatives to Markovian modeling, such as stochastic Petri nets, as in SNPN [16]. Also, Galileo [17] incorporates dynamic fault trees (DFT), which can model propagating failures by using functional dependency gates, but since the package does not analyze the complete state-space model of the entire system, it is limited in the types of component interactions it can handle. OpenSESAME [18] builds a stochastic Petri net from a high-level description of the system, in which the user may use failure dependency diagrams (FDD) to model cascading failures, but the models considered do not include the complex types of cascading failures we analyze. Another modeling approach is Boolean logic driven Markov processes (BDMP) [19], which combines concepts from fault trees and Markov processes. Also, the idea of cascading failures is related to other modeling techniques, including common-cause and common-mode failures and coverage [20], [21], [22], which have been incorporated in some of the aforementioned packages. DRBD [23] is another tool allowing for common failure modes by extending traditional RBD to dynamic RBD [24] to allow for certain types of component interactions. Finally, we note that Bayesian networks have also been used in reliability analysis [25].

The remainder of the paper is organized as follows: Section 2 describes the mathematical model, and the details of the CTMC are given in Section 3. In Section 3.1, we discuss the specifics of how our model incorporates cascading failures, and we present an example in Section 3.2. Section 4 gives our tree generation algorithm, and we discuss its performance characteristics in Section 4.1. Section 4.2 shows how to use the tree generation algorithm to compute the infinitesimal generator matrix of the CTMC, and once this has been computed, we describe in Section 5 how it can be used to compute various dependability measures of the CTMC. We present some numerical results from running our program on two examples in Section 6, and Section 7 provides concluding remarks.

2 Model

We now describe our Markovian dependability model. Suppose that there are N types of components, labeled $1, 2, \ldots, N$, and let $\Omega = \{1, 2, \ldots, N\}$ be the set of all component types. Let r_i be the number of components (*redundancy*) of type i, for $i \in \Omega$, where $1 \leq r_i < \infty$. We assume that all components of the same type are identical.

We allow the system to operate in randomly changing *environments*, and the failure and repair rates of components may depend on the current environment. For example, we may have high-load and low-load environments, and failure rates are higher in the high-load environment. We model the environment at time t with an environment variable $Z_0(t)$ that takes on values within the set $\mathcal{E} = \{1, 2, \ldots, L\}$ and whose value changes randomly over time in the following manner. If the system has just entered environment $e \in \mathcal{E}$, it stays in that environment for an exponentially distributed amount of time with rate $\nu_e > 0$ and then moves to environment $e' \in \mathcal{E}$ with probability $\delta_{e,e'}$. We assume that $\delta_{e,e} = 0$ for all $e \in \mathcal{E}$, so the environment cannot stay the same on an environment transition. Moreover, we assume that the matrix $\delta = (\delta_{e,e'} : e, e' \in \mathcal{E})$ is irreducible in the sense that for any $e, e' \in \mathcal{E}$, there exists $k > 0$ and a sequence $e_0 = e, e_1, e_2, \ldots, e_k = e' \in \mathcal{E}$ such that $\prod_{i=0}^{k-1} \delta_{e_i, e_{i+1}} > 0$; i.e., if the environment is currently e, the environment can eventually change to e' later on, and this is true for any pair of environments e and e'. Thus, the environment process $(Z_0(t) : t \geq 0)$ is an irreducible CTMC with infinitesimal generator matrix $Q_0 = (Q_0(e, e') : e, e' \in \mathcal{E})$, where $Q_0(e, e') = \nu_e \delta_{e,e'}$ for $e' \neq e$, and $Q_0(e, e) = -\sum_{e' \neq e} Q_0(e, e')$.

If the environment is currently e, each component of type i has a constant failure rate $\lambda_{i,e} > 0$. In the case there is

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

only one environment 1, this means that the lifetime of each component of type i has a lifetime that is exponentially distributed with rate $\lambda_{i,1}$. Exponential distributions are widely used to model lifetimes of hardware and software components; e.g., see [26] for theoretical and practical justifications of this assumption. We assume that all nonfailed components of each component type are operating, and they also fail at the same rate $\lambda_{i,e}$. Thus, our model assumes "hot spares," and if there are currently n_i components operational of type i and the current environment is e, the total failure rate of type i is $n_i \lambda_{i,e}$. (We could easily extend our model to allow for "cold spares," but for simplicity we did not.)

Failed components are repaired according to a processor-sharing discipline (also known as random-order service). Specifically, when there is only one failed component and it is of type i and the current environment is e, the constant rate of repair is $\mu_{i,e} > 0$. When there are b total components currently failed, the repairman devotes $1/b$ of his effort to each failed component, so the repair rate of an individual failed component of type i is $\mu_{i,e}/b$ when the current environment is e. Because of the processor-sharing repair discipline, components do not have to wait in a queue before being fixed. For simplicity, we do not consider other repair disciplines, such as first-in-first-out (FIFO). Assuming a FIFO repair discipline leads to an exponential explosion in the size of the state space, because then the Markov chain would need to also keep track of the order of the types of failed components in the queue. For this reason, the SAVE package [7] also assumes a processor-sharing discipline.

We model cascading failures as follows. For each component type i, let $\Gamma_i \subseteq \Omega$ be the ordered set of components that can fail instantaneously when a component of type i fails. The order in which the elements in Γ_i are listed specify the order in which they can fail, which is needed when considering the probability of a cascading failure. Given a component of type i has failed, each of the component types $j \in \Gamma_i$ fail independently of each other, with $\phi_{i,j}$ being the probability that the failure of a component of type i causes a component of type j to simultaneously fail; with probability $1 - \phi_{i,j}$, a component of type j does not simultaneously fail. A cascading failure can be thought of as a rooted, ordered tree, with the component that initially fails (the triggering component failure) at the root, and the cascading failures are probabilistic. Although all of the components in the tree fail instantaneously, we assume that the order in which they fail is in the order of a breadth-first enumeration of the nodes in the tree [27]. We assume that a component type j can appear at most once in each Γ_i. Section 3.1 provides more details on cascading failures.

3 Markov Chain

We now explicitly describe the resulting CTMC resulting from our model. Define the state space

$$S = \{(x_0, x_1, x_2, \ldots, x_N) : x_0 \in \mathcal{E}, 0 \leq x_i \leq r_i \text{ for all } i \in \Omega\}.$$

For a state $x = (x_0, x_1, \ldots, x_N) \in S$, the environment is x_0, and there are x_i components failed of type i for each $i \in \Omega$. The size of the state space is

$$|S| = L \prod_{i=1}^{N} (r_i + 1) < \infty. \tag{1}$$

Thus, the state space grows linearly in the number of environments in the model. Also, since $r_i + 1 \geq 2$, we have that $|S| \geq L2^N$, so S grows exponentially in the number of component types.

Define $Z(t) = (Z_0(t), Z_1(t), \ldots, Z_N(t))$, where $Z_0(t)$ is the environment at time t, and $Z_i(t)$, $i = 1, \ldots, N$, is the number of failed components of type i at time t. Then $Z = (Z(t) : t \geq 0)$ is the stochastic process keeping track of the evolution of the system over time, and by our assumptions, Z is a CTMC on the state space S. Let $Q = (Q(x, y) : x, y \in S)$ denote the $|S| \times |S|$-dimensional infinitesimal generator matrix of CTMC Z, where $Q(x, y)$ is the total rate of going from state x to state y.

Consider states $x = (x_0, x_1, \ldots, x_N)$ and $y = (y_0, y_1, \ldots, y_N)$ in S. Only certain pairs (x, y) represent valid transitions for our CTMC Z. We specify below the criteria that lead to allowable transitions (x, y):

- $y_0 \neq x_0$ and $y_i = x_i$ for all $i \in \Omega$. Then (x, y) represents a change in the environment from x_0 to y_0, and $Q(x, y) = \nu_{x_0} \delta_{x_0, y_0}$.
- $y_0 = x_0$, $y_i = x_i - 1$ for exactly one $i \in \Omega$, and $y_j = x_j$ for all $j \in \Omega - \{i\}$. Then, (x, y) represents a *repair transition*, in which a single component of type i is repaired, and $Q(x, y) = x_i \mu_{i,x_0} / (\sum_{j \in \Omega} x_j)$.
- $y_0 = x_0$, and $y_i \geq x_i$ for all $i \in \Omega$ with $y_j > x_j$ for some $j \in \Omega$. Then, (x, y) represents a *failure transition*, in which $y_i - x_i$ components of type i fail, for each $i \in \Omega$. Section 3.1 provides details on computing $Q(x, y)$. In the special case when cascading failures are not possible (i.e., $\Gamma_i = \emptyset$ for all $i \in \Omega$), the rate $Q(x, y) = n_i(x) \lambda_{i,x_0}$, where $n_i(x) \equiv r_i - x_i$ is the number of components of type i that are operational in state x.

For any pair (x, y), $x \neq y$, that does not satisfy any of the above conditions, the transition (x, y) is not possible, so $Q(x, y) = 0$. Also, $Q(x, x) = -\sum_{y \neq x} Q(x, y)$.

Proposition 1. *The CTMC Z is irreducible.*

Proof. Let $x' = (1, r_1, r_2, \ldots, r_N)$ denote the state in which the environment is 1 and all components are failed, and recall that we assumed that the matrix δ is irreducible. Because all operational components continue to fail in any state, the state x' is accessible from any other state $x \in S$. Moreover, our assumption of using a processor-sharing repair discipline ensures that any state $y \in S$ is accessible from x'. Hence, the CTMC Z is irreducible. □

3.1 Cascading Failures

We now describe how cascading failures occur. We can think of a cascading failure as a rooted tree of components that fail in a single transition, with the root being the component whose failure triggers the tree of failures in the cascade. Since we will be working with trees, we now

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

define some related concepts. The *depth* of a node in a tree is the length of the path (i.e., number of edges) from that node to the root. The set of all nodes at a given depth is called a *level* of the tree. The root is at level 0, the children of the root are level 1, and so on. In our tree corresponding to a cascading failure, each node directly caused each of its children to fail. So if a component of type i is the component at the root that triggered the cascading failures, some subset G of Γ_i of types of components is going to fail immediately after i fails, where the subset G is determined probabilistically using the probabilities $\phi_{i,j}$ for $j \in \Gamma_i$. Then for each component of type $j \in G$, a subset of the components in Γ_j will fail, where again the subset is determined probabilistically. This continues until the cascading stops. All of the component failures in the cascading occur instantaneously.

Consider a rooted tree $T = (V, E, r)$, where V is the set of nodes, E is the set of edges, and $r \in V$ is the root of T. Each node $v \in V$ has an ID $\alpha(v) \in \{1, 2, \dots, |V|\}$, which is unique, and also a type $\beta(v) \in \Omega$, which is the type of component whose failure corresponds to node v. We assume that the ID of a node is the order in which the node is visited in a breadth-first enumeration of the nodes in the tree; i.e., the root has ID 1, its children have IDs starting from 2 to 1 + however many children the root has, and so on. All nodes at the same level have IDs in increasing order from left to right. Given a node ID i, let $\theta(i) \in \Omega$ be the type of the component that fails at the node having ID i. Let $K_j(T)$ be the number of nodes that have type j in tree T; i.e., $K_j(T) = \sum_{v \in V} 1\{\beta(v) = j\} = \sum_{i=1}^{|V|} 1\{\theta(i) = j\}$, where $1\{A\} = 1$ if A is true, and 0 otherwise. Hence, the collection $\{\beta(v) : v \in V\} = \{\theta(i) : i = 1, \dots, |V|\}$ is a multiset in which the number of copies of type j is $K_j(T)$.

Suppose that the system currently is in state $x = (x_0, x_1, \dots, x_N)$, and a cascading failure corresponding to a tree T occurs. Algorithm 1 computes $\texttt{Rate}(x, T)$, which is the corresponding rate, and the calculation works as follows. The variable $\texttt{nfailed}[i]$ keeps track of the number of components of type i that are currently failed, which is initially x_i. The cascading failure begins with the component at the root failing, and this component is of type $\theta(1)$. This happens at rate $n_{\theta(1)}(x)\lambda_{\theta(1),x_0}$ since there are $n_{\theta(1)}(x)$ components of type $\theta(1)$ operational in state x and each has failure rate $\lambda_{\theta(1),x_0}$. Consider the node having ID i, which has type $\theta(i)$. When a component of this type fails, it probabilistically causes components of types $j \in \Gamma_{\theta(i)}$ to also fail simultaneously. When the failure of the component of type $\theta(i)$ at the node having ID i causes a component of type $j \in \Gamma_{\theta(i)}$ to fail, we multiply the rate by $\phi_{\theta(i),j}$. When the failure of the component of type $\theta(i)$ at node having ID i does not cause a component of type $j \in \Gamma_{\theta(i)}$ to fail, we multiply the rate by $1 - \phi_{\theta(i),j}$, but only if there were still components of type j operational at that point (i.e., if $\texttt{nfailed}[j] < r_j$).

For a tree T and state x, define $x(T) = (x_0(T), x_1(T), \dots, x_N(T))$ with $x_0(T) = x_0$ and $x_i(T) = x_i + K_i(T)$ for each $i \in \Omega$. Thus, $x(T)$ defines another potential state that would be reached from x after a cascading failure corresponding to tree T. If $x_i(T) > r_i$ for any $i \in \Omega$, then the number of components of type i that failed in the tree T is more than the number operational of type i in state x, so $x(T)$ is not a valid state and $(x, x(T))$ is not a valid transition. Otherwise (i.e., if $x_i(T) \le r_i$ for all $i \in \Omega$), $(x, x(T))$ is a valid transition.

For a given failure transition (x, y), there can be many trees T, possibly with roots of different types, such that $x(T) = y$, so to compute $Q(x, y)$, we need to sum over all of the rates of trees T with $x(T) = y$ to get $Q(x, y)$. Specifically,

$$Q(x, y) = \sum_{\text{Trees } T:\, x(T) = y} \texttt{Rate}(x, T).$$

The sum over the trees T includes all possible types for the root.

Algorithm 1. Rate(x, T), where x is a state and $T = (V, E, r)$ is a rooted, ordered tree

```
1:  for i = 1 to N do
2:    nfailed[i] = x_i;
3:  end for
4:  Rate = n_θ(1)(x) λ_θ(1),x_0;
5:  nfailed[θ(1)] = nfailed[θ(1)] + 1;
6:  for i = 1 to |V| do
7:    for j ∈ Γ_θ(i) do
8:      if i has a child of type j then
9:        Rate = Rate * φ_θ(i),j;
10:       nfailed[j] = nfailed[j] + 1;
11:     else
12:       if nfailed[j] < r_j then
13:         Rate = Rate * (1 − φ_θ(i),j);
14:       end if
15:     end if
16:   end for
17: end for
18: y = x(T); {Compute state y corresponding to the
      transition from x on tree T}
19: Q(x, y) = Q(x, y) + Rate; {Add in rate of current tree into
      current entry of Q-matrix}
20: return Rate;
```

3.2 An Example

We now consider an example. Suppose that there are three types of components ($N = 3$):

- Type 1 has redundancy $r_1 = 3$. When a component of type 1 fails, it causes components of types 2 and 3 to fail with probabilities $\phi_{1,2}$ and $\phi_{1,3}$, respectively. Thus, $\Gamma_1 = \{2, 3\}$.
- Type 2 has redundancy $r_2 = 3$. When a component of type 2 fails, it causes components of types 1 and 3 to fail with probabilities $\phi_{2,1}$ and $\phi_{2,3}$, respectively. Thus, $\Gamma_2 = \{1, 3\}$.
- Type 3 has redundancy $r_3 = 5$. When a component of type 3 fails, it causes a component of type 2 to fail with probability $\phi_{3,2}$. Thus, $\Gamma_3 = \{2\}$.

Consider state $x = (x_0, x_1, x_2, x_3) = (1, 2, 1, 4)$. In this state, the environment is 1, and

- type 1 has two components failed and $n_1(x) = 1$ component operational,
- type 2 has one component failed and $n_2(x) = 2$ components operational,

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

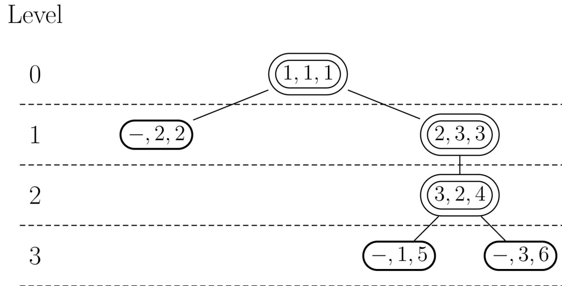

Fig. 1. Tree T_1.

- type 3 has four components failed and $n_3(x) = 1$ component operational.

Also, consider the state $y = (y_0, y_1, y_2, y_3) = (1, 3, 2, 5)$, and we want to determine the total rate for the transition (x, y). Note that $y - x = (0, 1, 1, 1)$, so the environment stays the same and a single component of each type fails in the transition.

Consider the tree in Fig. 1. Let $T_1 = (V_1, E_1, 1)$ be the tree consisting of only the nodes drawn with a double circle, and this denotes a cascading failure corresponding to the transition (x, y). The single-circled nodes correspond to components that might potentially have failed in the cascading (because they belong to a set Γ_i of a component type i that failed) but actually did not fail. Potential failures may not have occurred because either the event probabilistically did not occur or there were no more components of that type still operational at that point and so it could not have failed. Let $T_1^* = (V_1^*, E_1^*, 1)$ be the supertree consisting of all the nodes. Each node in T_1^* has a label (z_1, z_2, z_3). For a node that belongs to T_1, the first label z_1 is the node's ID within the tree T_1; if the node does not belong to T_1, then z_1 is null and is denoted by a dash. We call z_1 the *tree ID* of the node. The second label z_2 is the component type of the node, and z_3 denotes the node's ID within the supertree T_1^*. We call z_3 the *supertree ID* of the node. The tree IDs and supertree IDs give breadth-first enumerations of their respective trees. For example, for the node in level 2, its tree ID is 3, its component type is 2, and its supertree ID is 4. For the left node in level 3, its tree ID is null, its component type is 1, and its supertree ID is 5.

We now use Algorithm 1 to calculate $\texttt{Rate}(x, T_1)$:

- Before the cascading failure begins, $\texttt{nfailed}[1] = 2$, $\texttt{nfailed}[2] = 1$, and $\texttt{nfailed}[3] = 4$.
- The cascade is triggered by a failure of a component of type 1 at the node with supertree ID 1. Hence, we include a factor $n_1(x)\lambda_{1,1} = \lambda_{1,1}$ in the rate, and increment $\texttt{nfailed}[1]$ to 3.
- The failure of the component of type 1 at the node with supertree ID 1 can potentially cause failures of components of types in $\Gamma_1 = \{2, 3\}$. The component of type 2 at the node with supertree ID 2 does not fail; since $\texttt{nfailed}[2] = 1 < 3 = r_2$, there are still components of type 2 operational at this point, so the rate includes a factor of $1 - \phi_{1,2}$. The component of type 3 at the node with supertree ID 3 fails, so the rate includes a factor of $\phi_{1,3}$. Also, we increment $\texttt{nfailed}[3]$ to 5.
- The failure of the component of type 3 at the node with supertree ID 3 can potentially cause failures of components of types in $\Gamma_3 = \{2\}$. The component of type 2 at the node with supertree ID 4 fails, so the rate includes a factor of $\phi_{3,2}$. Also, we increment $\texttt{nfailed}[2]$ to 2.
- The failure of the component of type 2 at the node with supertree ID 4 can potentially cause failures of components of types in $\Gamma_2 = \{1, 3\}$. The component of type 1 at the node with supertree ID 5 does not fail; since $\texttt{nfailed}[1] = 3 \not< 3 = r_1$, there are no more components of type 1 operational at this point, so the rate does not include the factor $1 - \phi_{2,1}$. The component of type 3 at the node with supertree ID 6 does not fail; since $\texttt{nfailed}[3] = 5 \not< 5 = r_3$, there are no more components of type 3 operational at this point, so the rate does not include the factor $1 - \phi_{2,3}$.

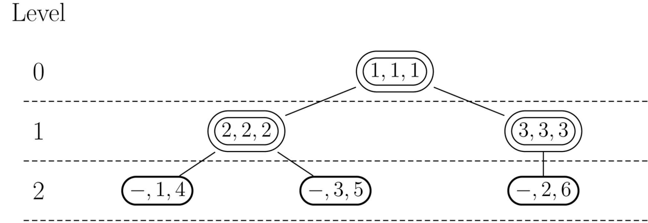

Fig. 2. Tree T_2.

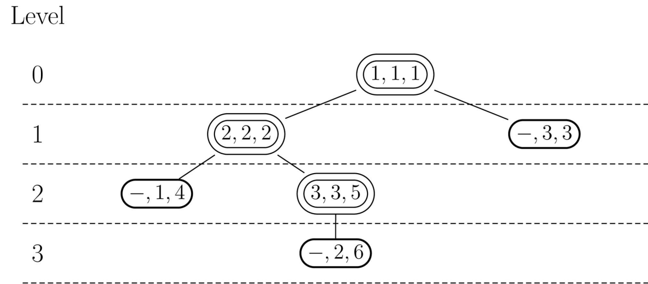

Fig. 3. Tree T_3.

Hence, the overall rate for tree T_1 is

$$\texttt{Rate}(x, T_1) = \lambda_{1,1}(1 - \phi_{1,2})\phi_{1,3}\phi_{3,2}.$$

We then include $\texttt{Rate}(x, T_1)$ as a summand in the sum for $Q(x, y)$.

Figs. 2, 3, 4, 5, 6 show all of the other trees T_2, T_3, T_4, T_5, and T_6 that correspond to a transition from x to y. Trees T_2 and T_3 have roots of type 1, trees T_4 and T_5 have roots of type 2, and tree T_6 has a root of type 3. The rates for these trees are

$$\begin{aligned}
&\texttt{Rate}(x, T_2) = \lambda_{1,1}\phi_{1,2}\phi_{1,3}(1 - \phi_{3,2}),\\
&\texttt{Rate}(x, T_3) = \lambda_{1,1}\phi_{1,2}(1 - \phi_{1,3})\phi_{2,3}(1 - \phi_{3,2}),\\
&\texttt{Rate}(x, T_4) = 2\lambda_{2,1}\phi_{2,1}(1 - \phi_{2,3})(1 - \phi_{1,2})\phi_{1,3}(1 - \phi_{3,2}),\\
&\texttt{Rate}(x, T_5) = 2\lambda_{2,1}\phi_{2,1}\phi_{2,3}(1 - \phi_{1,2})(1 - \phi_{3,2}),\\
&\texttt{Rate}(x, T_6) = \lambda_{3,1}\phi_{3,2}\phi_{2,1}(1 - \phi_{1,2}).
\end{aligned}$$

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

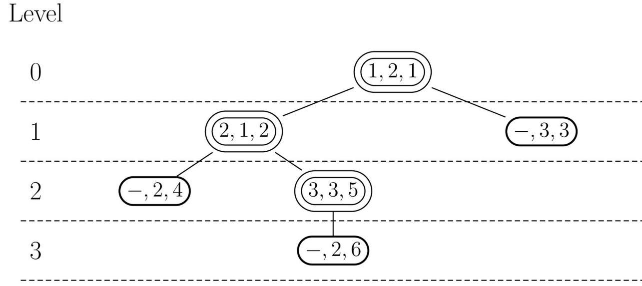

Fig. 4. Tree T_4.

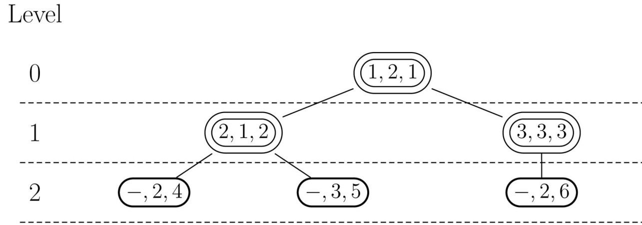

Fig. 5. Tree T_5.

No other trees T with any type of root has $x(T) = y$. Thus, since $y = x(T_i)$ for $i = 1, 2, \ldots, 6$, we have

$$Q(x, y) = \sum_{i=1}^{6} \texttt{Rate}(x, T_i).$$

4 Algorithm to Generate All Trees

We now provide a recursive algorithm to generate all possible trees T for a given collection of component types, possibly with redundancy. Specifically, fix a state $x = (x_0, x_1, \ldots, x_N)$ and consider a multiset M of component types to fail in a cascade starting from state x. Let M_i be the number of copies of type i in the multiset M, and define state $y = (y_0, y_1, \ldots, y_N)$ with $y_0 = x_0$ and $y_i = x_i + M_i$ for each $i \in \Omega$. We now want to compute the transition rate $Q(x, y)$ by generating all possible trees T that correspond to failures of all components in M. There are well-known algorithms (e.g., [28, Section 2.3.4.4]) for enumerating all trees having a given set of labels without any required structure. However, in our problem, the sets Γ_i, $i = 1, \ldots, N$, which specify the possible cascading failures, impose restrictions on a node's possible children and their ordering. Function `Tree` in Algorithm 3 accomplishes this tree-generation task.

In the code in Algorithm 2, `GenTree`(M, Γ) is the top-level calling routine and serves as the wrapper function of `Tree`, which is where all the tree generation work is performed. `GenTree` takes as input M and Γ, which is the collection of all of the Γ_i. In Algorithm 3, function `Tree` grows a tree under the type restrictions and types available using breadth-first search (BFS) [27]. The nodes of the generated tree can be labeled accordingly (e.g., by assigning to them consecutive numerical values starting from an arbitrary base value) and the tree can be represented

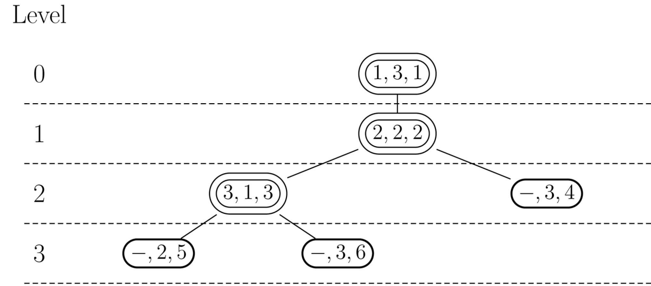

Fig. 6. Tree T_6.

efficiently in sparse form through a parent array p [27], where $p(v) = u$ if v is a child spawned by u (or equivalently, u is the parent of v). We omit details about BFS in `Tree`. For example, a BFS-related function `BFSAssignedLabel` that assigns a numerical label to a node that signifies the order of visit of that node is not explained; incrementing a counter suffices to implement this function. We also do not provide implementation-related details for `Tree` or `GenTree` as this might complicate the exposition of our algorithm. For example, Algorithm 1 requires an alternative representation of a tree (other than the parent array p maintained by BFS), such as in the form of a graph represented with an adjacency matrix or an adjacency list. Implementationwise, the conversion from one representation into another is straightforward. To highlight this, function `EvaluateRate` is used in `Tree` and `GenTree` to distinguish the former from `Rate` defined in Algorithm 1. The distinction is that `EvaluateRate` maintains a tree as a parent array p whereas `Rate` expects (for efficiency purposes) a tree T represented as a graph (as well as the state x that the system is in before the cascading occurs). One can thus consider `EvaluateRate` as a wrapper function for `Rate`: the first step of the former is the conversion of the parent array p into the form acceptable as input by `Rate`.

Algorithm 2. GenTree(M, Γ)

1: $Mtype$ = types(M);
2: $Mtemp = M$;
3: **for** $j = 1$ to $|Mtype|$ **do**
4: $Mtemp = Mtemp - Mtype[j]$; {Remove a node of type $Mtype[j]$ from $Mtemp$}
5: root = CreateNode(); {Create a node for the root of the tree}
6: root.$\alpha = 1$; root.$\beta = Mtype[j]$; {Root has ID 1 and type $Mtype[j]$}
7: **if** NotEmpty($Mtemp$) **then**
8: Tree($Mtemp$,Γ,root,root.β);
{Start a tree with root whose type is one of the possible types of M}
9: **else**
10: EvaluateRate(); {Tree has only one node, its root}
11: **end if**
12: **end for**
13: **return**

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

Algorithm 3. Tree$(M, \Gamma, \mathrm{node}, \mathrm{node}.\beta)$ where node.β is the assigned type of root node

1: $\mathrm{Queue} = \{\ \}; \mathrm{Stack} = \{\ \}; \mathrm{Enqueue}(\mathrm{Queue}, \mathrm{node});$
2: **while** (NotEmpty(Queue)) || ((Empty(Queue)) && (NotEmpty(Stack))) **do**
3: **if** (Empty(Queue) && NotEmpty(Stack)) **then** {A tree T has been formed}
4: **if** (Empty(M)) **then** {If M is empty, then, T utilizes all types. Run EvaluateRate}
5: EvaluateRate();
6: **end if**
7: goto line 27; {Proceed to generating/enumerating another T}
8: **end if**
9: $u = \mathrm{Dequeue}(\mathrm{Queue})$; {Pick next node from Queue to expand using BFS}
10: $X = \{\ \}$; {X will hold the types of us children}
11: $Mtype = \mathrm{types}(M)$; $ntypes = |\mathrm{types}(M)|$;
12: **for** $j = 1$ to $ntypes$ **do**
13: **if** BelongsTo$(Mtype[j], \Gamma_{u.\beta})$ **then**
14: $X = X \cup \{Mtype[j]\}$;
15: **end if**
16: **end for**
17: $d = |X|$; {d is the maximum number of different types of children u can have}
18: **if** ($d == 0$) **then** (No type in M can be a child of u}
19: **if** Empty(M) **then** {Tree T is complete}
20: EvaluateRate();
21: $\langle M, \mathrm{Queue}\rangle = \mathrm{Top}(\langle M, \mathrm{Queue}\rangle, \mathrm{Stack})$; {Reset M and Queue for backtracking}
22: continue;
23: **else**
24: continue; {Current u cannot have children}
25: **end if**
26: **end if** {d might be 0 because u cannot be expanded even if other siblings can}
27: $Y = \mathrm{EnumerateNextSubset}(u, d, X)$ for subsets of size $k = 0$ to d with increasing k; {$Y = \{j_1, \dots, j_k\}$ is the next subset of types from X to be u's children, where $j_1, \dots, j_k$ are in the lexicographic ordering specified by $\Gamma_{u.\beta}$}
28: **if** FirstEnumeratedSet(Y) **then** {Is the current Y the first set to be generated?}
29: $\mathrm{Push}(\langle M, \mathrm{Queue}\rangle, \mathrm{Stack})$; {Remember state by pushing M and Queue onto Stack}
30: **else if** LastEnumeratedSet(Y) **then** {Is **Y** the last set, i.e., is its cardinality d?}
31: $\mathrm{Pop}(\langle M, \mathrm{Queue}\rangle, \mathrm{Stack})$; {Erase saved state}
32: continue;
33: **else** {Y is neither the first nor the last enumerated set}
34: $\langle M, \mathrm{Queue}\rangle = \mathrm{Top}(\langle M, \mathrm{Queue}\rangle, \mathrm{Stack})$; {Reset M and Queue from Stack}
35: **end if**
36: **for** $t = 1$ to k **do** {Create nodes for the types in Y to be the children of u}
37: v_t = CreateNode(); {Create node for the tth child of u}
38: $v_t.\alpha = \mathrm{BFSAssignedLabel}()$; $v_t.\beta = j_t$; {Assign ID and type to node v_t}
39: $\mathrm{TypeUpdate}(M, j_t)$; {Remove from M type j_t that we just used}
40: $\mathrm{Enqueue}(\mathrm{Queue}, v_t)$; {Insert node v_t into Queue for later expansion}
41: **end for**
42: **end while**
43: **if** (Empty(M)) **then** {This is similar to line 4 as Stack and Queue are now both empty}
44: EvaluateRate();
45: **end if**
46: **return**

Functions GenTree and Tree utilize a variety of objects and auxiliary functions. Several of them are self-explanatory; we provide some details below for the nonobvious ones. Object Queue is an implementation of the abstract data type of a FIFO (first-in, first-out) queue, and Stack is an implementation of a LIFO (last-in, first-out) stack, with operations Enqueue and Dequeue defined for Queue and Push, Pop and Top for Stack. Operations Enqueue and Push define an appropriate insertion and Dequeue and Pop an appropriate deletion; Top is a probing operation that returns the object at the top of the stack without removing it. Function types return the distinct types of M into an array *Mtype*, and *ntypes* denotes the length of *Mtype*. Function $|\cdot|$ returns the cardinality of a set. In Algorithms 2 and 3, attributes α and β denote the unique ID $\alpha(u)$ and the type $\beta(u)$ of a node u. In line 27 of Algorithm 3, u is the current node to be expanded (i.e., assigned children), X is the maximal set of types for the children of u, and d is the cardinality of X. For a given (u, d, X), each successive call to function EnumerateNextSubset(u, d, X) returns the next subset of X in lexicographic order (using the ordering given in $\Gamma_{u.\beta}$). The subset returned from EnumerateNextSubset contains the types of the children of the node u in the current tree that is being enumerated. Other functions are self-explanatory (e.g., NotEmpty, Empty.)

As pointed out before, function Tree builds a tree using BFS under the type restrictions specified by the Γ_i. We maintain information about this BFS operation through Queue. A particular node u might get expanded in a variety of ways depending on its type, and such an expansion also affects BFS. To account for all possible alternatives, we use Stack. Every time a node u is about to be expanded, the algorithm pushes information onto Stack. Every time a different subset of children is assigned to node u, information stored in Stack is used to restore the values of M and Queue that were changed since the previous expansion of u. After all possible sets of children for u have been generated, we pop from Stack the information used in the expansion of u. In Stack, we maintain information about the current state of Queue and the current M. Other information pertinent to the generation might also be stored in Stack; e.g., in the operations needed in line 27 in Algorithm 3, information about the size k of Y might also be stored in Stack, as well as information that will allow the labeling of lines 38-40.

4.1 Performance Characteristics of Algorithms 2 and 3

We first want to study the number of trees that Algorithm 2 produces. The number depends on the component types in

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

multiset M, the number M_i of copies of component type i in M, and the ordered sets Γ_i, $i \in \Omega$, which restrict the possible children of a node of type i. Recall N is the total number of component types, and let $m = \sum_{i=1}^{N} M_i$ be the total number of components of all types in M. In general, it is difficult to determine the exact number of trees that will be generated for arbitrary sets Γ_i. However, we can do this for the special case when every component type can cause every component type to fail, i.e., when $\Gamma_i = \Omega$ for each $i \in \Omega$.

Proposition 2. *If $\Gamma_i = \Omega$ for each $i \in \Omega$, then the number of trees produced by Algorithm 2 is $m^{m-1}/(\prod_{i\in\Omega} M_i!)$.*

Proof. Suppose initially that each M_i is 0 or 1, i.e., there is at most 1 component of each type in M. The number of trees that Algorithm 2 will then generate is the number of labeled trees on m nodes. Cayley's formula (e.g., [28, Section 2.3.4.4]) gives this number as m^{m-2} for unrooted trees. Since we have rooted trees, the number of trees generated by Algorithm 2 in this case is m^{m-1}. More generally, when each M_i can take on any value, the total number of trees is $m^{m-1}/(\prod_{i\in\Omega} M_i!)$. □

More restrictive sets Γ_i reduce the number of trees possible, and the number given in Proposition 2 then provides an upper bound.

The space requirements for Algorithm 3 are polynomial in the number of nodes of the generated tree, which is no more than m. The `Queue` data structure maintains at most m nodes. Each entry of `Stack` maintains information related to the current state of the `Queue`, which is proportional to m, and also to the current state of the multiset M, which is proportional to N. Given that the total number of nodes expanded determines the amount of information maintained by `Stack`, this information is thus proportional to $m(N+m)$.

4.2 Constructing the Q-Matrix

Algorithm 4 shows how to use function `GenTree` to construct the entire infinitesimal generator matrix Q. The algorithm loops through all possible transitions (x, y). Line 4 handles the case that (x, y) corresponds to a change in the environment. Line 6 handles the case that (x, y) corresponds to a repair of a component. Line 8 handles the case that (x, y) corresponds to a failure transition, possibly a cascading failure. Line 9 determines the multiset M of components that fail in the transition (x, y), and function `GenTree` then generates all of the trees having nodes with labels being the types in M. For each of these trees, `GenTree` calls `EvaluateRate` to compute the rate of the tree and add it to the current value of $Q(x, y)$. Note that `GenTree` needs to be called for each possible failure transition (x, y), and each of these calls requires generating up to m^{m-1} trees, where m is the total number of components that fail in the transition (x, y).

Algorithm 4. ConstructQmatrix(Q)

1: **for** $x = (x_0, x_1, \ldots, x_n) \in S$ **do**
2: **for** $y = (y_0, y_1, \ldots, y_n) \in S$, $y \neq x$ **do**
3: $Q(x, y) = 0$;
4: **if** $y_0 \neq x_0$ and $y_i = x_i$ for all $i \in \Omega$ **then**
5: $Q(x, y) = \nu_{x_0}\delta_{x_0,y_0}$; {$(x, y)$ is an environment-change transition}
6: **else if** $y_0 = x_0$, $y_i = x_i - 1$, and $y_j = x_j$ for all $j \in \Omega - \{i\}$ **then**
7: $Q(x, y) = x_i\mu_{i,x_0}/(\sum_{j\in\Omega} x_j)$; {$(x, y)$ is a repair transition}
8: **else if** $y_0 = x_0$ and $y_i \geq x_i$ for all $i \in \Omega$ with $y_j > x_j$ for some $j \in \Omega - \{i\}$ **then**
9: $M =$ multiset with $y_i - x_i$ copies of i, for each $i \in \Omega$;
10: GenTree(M, Γ); {(x, y) is a failure transition, so generate trees}
11: **end if**
12: **end for**
13: $Q(x, x) = -\sum_{y\neq x} Q(x, y)$;
14: **end for**
15: **return** Q;

Remark. Instead of filling in the entries in the matrix Q corresponding to failure transitions as in Algorithm 4, an alternative approach is as follows. First loop over all states x. Each x corresponds to a certain collection $C(x)$ of components (with possible redundancies) that are operational in x and could fail. For each x, generate every tree that could be constructed having all the components in $C(x)$. Then, as each of these trees is being constructed, evaluate the rate of each partially built tree and update $Q(x, y)$ as $Q(x, y) \leftarrow Q(x, y) + \mathrm{rate}$(current partially built tree), where y corresponds to the state we end in when starting in x and having all components fail in the current partially build tree. Implementing this idea requires modifying Algorithm 3 so that it calls `EvaluateRate` in different places. Specifically, eliminate all of the existing calls to `EvaluateRate` and add a new one right after line 41. We can think of this as being a "tree-based" approach to constructing the Q-matrix, and it fills in each row by jumping around to different entries corresponding to the partially built trees. In contrast, the "state-based" approach given in Algorithm 4 fills in Q in an orderly fashion by generating all of the trees for one failure transition (x, y) before moving to the next entry in Q. The tree-based approach may lead to improvement in the time required to fill in Q, but the total number of trees that need to be evaluated in both approaches is the same.

Now we examine the total number of trees generated by Algorithm 4. Trees only need to be considered in failure transitions, which account for less than half of the $|S|^2$ entries in the generator matrix Q. On any failure transition, at most $\sum_{i=1}^{N} r_i$ can fail since this is the total number of components in the system, so $m \leq \sum_{i=1}^{N} r_i$ in Proposition 2. Thus, by (1) and since $M_i! \geq 1$, we obtain the following crude upper bound of the total number of trees needed to construct the entire Q-matrix:

$$\frac{1}{2}\left(L\prod_{i=1}^{N}(r_i+1)\right)^2\left(\sum_{i=1}^{N} r_i\right)^{(\sum_{i=1}^{N} r_i)-1}, \tag{2}$$

which is exponential in both the number N of component types and the redundancies r_i. As we will see in Section 6, the bound in (2) can be very loose for models in which the sets Γ_i place restrictions on the possible cascading.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

Algorithm 1 is run on each tree generated. Consider a tree T with $|V| = m$ nodes. Since $\Gamma_i \subseteq \Omega$ and $|\Omega| = N$, we get $|\Gamma_i| \leq N$. Thus, Algorithm 1 requires $O(mN)$ work.

5 Dependability Measures

Once the infinitesimal generator matrix Q has been computed, we can use it to compute various dependability measures of the CTMC Z. Before we do this, we partition the state space S into S_1 and S_2, where S_1 (respectively, S_2) is the set of states in which the system is considered to be operational (respectively, failed). (We discuss one way this can be done in Section 6.) Let $x_* \in S$ be the state with all components operational and the environment is 1, and we assume that $x_* \in S_1$.

One dependability measure we can compute from Q is the system's steady-state unavailability U, which is the long-run fraction of time that the system is in a failed state and can be computed as follows. We first compute the stationary distribution as the nonnegative solution $\pi = (\pi(x) : x \in S) \in \Re^{|S|}$ to the set of equations $\pi^\top Q = 0$ and $\pi^\top e = 1$, where the superscript $\top$ denotes transpose and $e = (1, 1, \dots, 1)^\top \in \Re^{|S|}$ is the vector of all 1s; e.g., see [5, p. 410]. The irreducibility of Z from Proposition 1 and finiteness of S guarantee the existence and uniqueness of π. Then, defining $f = (f(x) : x \in S)$ with $f(x) = 0$ if $x \in S_1$ and $f(x) = 1$ if $x \in S_2$, we compute the steady-state unavailability as $U = \pi^\top f$; e.g., see [5, p. 410].

Another dependability measure that can be computed from Q is the mean time to failure (MTTF), which is the expected amount of time until the system reaches a failed state, given that it starts in state x_*. First, define the transition probability matrix $P = (P(x, y) : x, y \in S)$ of the embedded discrete-time Markov chain, where $P(x, y) = -Q(x, y)/Q(x, x)$ for $x \neq y$, and $P(x, x) = 0$. Next, define the matrix $P_R = (P_R(x, y) : x, y \in S)$ with $P_R(x, y) = P(x, y)$ if $y \in S_1$ and $P_R(x, y) = 0$ if $y \in S_2$. Then, define the vector $h = (h(x) : x \in S)$ with $h(x) = -1/Q(x, x)$, which is the expected holding time in state x for the CTMC Z. Letting I denote the $|S| \times |S|$-dimensional identity matrix, we then define the vector $g = (g(x) : x \in S)$ as $g = (I - P_R)^{-1} h$, and $g(x_*)$ is the MTTF; e.g., see [5, Section 7.9].

Of course, other dependability measures can also be computed from Q. These include transient measures, such as the reliability or interval availability. See [7] for details.

6 Experimental Results

We developed a Java program (about 10,000 lines of code) that implements Algorithms 1-4 to automatically construct the Q-matrix from a system-level specification of a model, which is input to the program. The input the user provides includes the following information:

- the set $\mathcal{E}$ of environments and its associated parameters: the rates ν_e of leaving environment e and transition probabilities $\delta_{e,e'}$ of moving from environment e to e', for all $e, e' \in \mathcal{E}$;
- for each component type i, the redundancy r_i, and failure rate $\lambda_{i,e}$ and repair rate $\mu_{i,e}$ in each environment e;

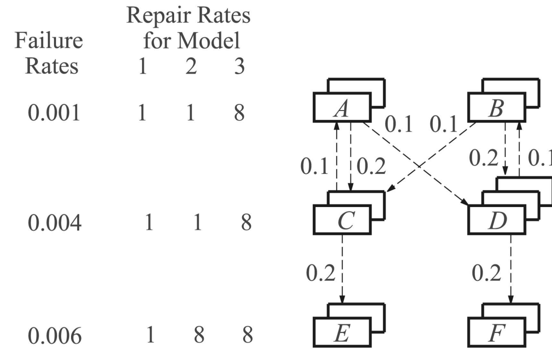

Fig. 7. Diagram of system in Example 1 with probabilities of cascading failures (dashed lines) and failure and repair rates in environment 1. In environment 2, the failure rates are halved, but the repair rates remain the same.

- the cascading failure parameters Γ_i and $\phi_{i,j}$, where Γ_i specifies, when a component of type i fails, the ordered set of components that probabilistically fail simultaneously, and $\phi_{i,j}$ is the probability of i causing j to immediately fail, for each $j \in \Gamma_i$;
- conditions under which the system is considered operational, where the conditions are given as a Boolean expression on the number of components operational of each type, using ANDs, ORs, and comparison operators ($<$, $\leq$, $>$, $\geq$ and $==$). This allows for specifying system operational conditions using series, parallel, and k-out-of-n subsystems. For example, the expression "$((A \geq 1)\text{AND}(B \geq 1))\ \text{OR}(C \geq 3)$" means that the system is operational as long as there is at least one component operational of each type A and B, or there are at least three components operational of type C.

The program uses the last input to automatically partition the state space S into operational states S_1 and failed states S_2. Specifically, for each state $x = (x_0, x_1, \dots, x_N) \in S$, where x_i denotes the number of components failed of type i for $i \in \Omega$, the program evaluates the Boolean expression to determine if x belongs to S_1 or S_2.

Once the Q-matrix has been computed, the program then computes the unavailability and MTTF of the system using the methods described in Section 5. Below we describe the numerical results from running our program on two examples.

6.1 Example 1

We ran our program on different versions of the system depicted in Fig. 7, which is a modification of a fault-tolerant computing system considered in [8]. The system consists of $N = 6$ component types $A, B, \dots, F$. Types A and B are processors, each having redundancy 2. Types C and D are disk controllers, where C has redundancy 2 and D has redundancy 3. Types E and F are disk clusters, each with redundancy 2.

There are two environments, 1 and 2. If the environment is currently 1 (respectively, 2) and there is a change in environment, the environment becomes 2 (respectively, 1); hence, $\delta_{1,2} = \delta_{2,1} = 1 = 1 - \delta_{1,1} = 1 - \delta_{2,2}$. Once there is a change in environment from 1 to 2 (respectively, 2 to 1), the

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

TABLE 1
Numerical Results

Model	Unavailability ($\times 10^{-4}$)		MTTF ($\times 10^{3}$)	
	No Cascading	Cascading	No Cascading	Cascading
1	1.087	5.969	9.51	2.57
2	0.228	3.800	31.18	3.79
3	0.017	0.493	73.40	4.27

environment remains as 2 (respectively, 1) for an exponentially distributed amount of time with rate $\nu_2 = 2$ (respectively, $\nu_1 = 2$). Taking the time units to be days, this means that the environment changes twice a day on average.

When the environment is 1, the failure rates of the components are $\lambda_{A,1} = \lambda_{B,1} = 0.001$, $\lambda_{C,1} = \lambda_{D,1} = 0.004$, and $\lambda_{E,1} = \lambda_{F,1} = 0.006$. When the environment is 2, the failure rates of the components are $\lambda_{i,2} = \lambda_{i,1}/2$ for each component type i. The system is considered operational if and only if at least one component of each type is operational, so the Boolean expression for the system to be operational is $(A \geq 1) \wedge (B \geq 1) \wedge \ (C \geq 1) \wedge (D \geq 1) \wedge (E \geq 1) \wedge (F \geq 1)$, where $\wedge$ denotes AND.

When a component of type A (respectively, B) fails, it first causes a component of type C (respectively, D) to fail with probability 0.2 and then a component of type D (respectively, C) to fail simultaneously with probability 0.1; hence, $\Gamma_A = \{C, D\}, \Gamma_B = \{D, C\}$, $\phi_{A,C} = \phi_{B,D} = 0.2$, and $\phi_{A,D} = \phi_{B,C} = 0.1$. When a component of type C (respectively, D) fails, it first causes a component of type E (respectively, F) to fail simultaneously with probability 0.2 and then a component of type A (respectively, B) to fail simultaneously with probability 0.1; hence, $\Gamma_C = \{E, A\}$, $\Gamma_D = \{F, B\}$, $\phi_{C,E} = \phi_{D,F} = 0.2$, and $\phi_{C,A} = \phi_{D,B} = 0.1$. When a component of type E or F fails, it does not cause any other components to fail simultaneously, so $\Gamma_E = \Gamma_F = \emptyset$.

We considered three different versions of our example by varying the repair rates. (For simplicity, we specified component repair rates that do not depend on the current environment, so $\mu_{i,e} = \mu_i$ for all component types i and environments e.) In Model 1, the repair rate is 1 for all component types i. In Model 2, component types A-D have repair rate 1, and component types E and F have repair rate 8. In Model 3, all component types have repair rate 8.

For each of the three models, the resulting CTMC has $2 \times 3 \times 3 \times 3 \times 4 \times 3 \times 3 = 1{,}944$ states by (1). Also, the total number of trees evaluated to construct the Q-matrix is 194,404, which is significantly less than the upper bound given by (2). The average number of trees evaluated per failure transition is 4.18, with 504 as the maximum number of trees evaluated for any single failure transition.

Table 1 contains the results for the three models in the columns labeled "Cascading." As expected, the steady-state unavailability U decreases and the MTTF increases as the repair rates increase. However, the relative differences in unavailability are larger than the relative differences in the MTTF across the models, which we now explain. In all three models, when one component fails, the chances that other components simultaneously fail (and bring the system down) are the same since we did not alter the cascading failure probabilities in the different models.

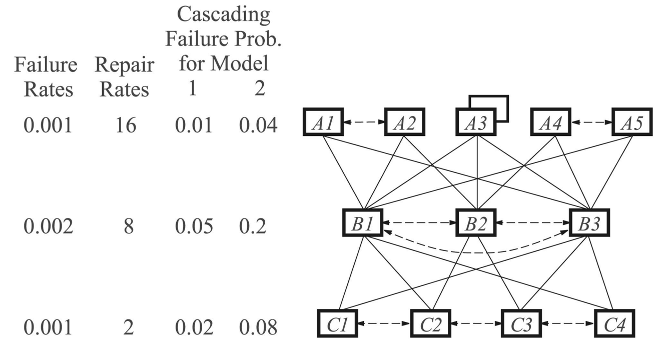

Fig. 8. Diagram of system in Example 2 with probabilities of cascading failures (dashed lines) and failure and repair rates.

Thus, there is not much change in the MTTF results for the three models. But once the system is down, larger repair rates lead to the system being down for shorter times, so the unavailability is more affected by changes in the repair rates than the MTTF.

For comparison, we also computed the unavailability and the MTTF for the same models without cascading failures, i.e., $\Gamma_i = \emptyset$ for all component types i. Table 1 gives these results in the columns labeled "No Cascading." For Model 1, the unavailability and MTTF differ by factors of 5.5 and 3.7, respectively, from the case when there are cascading failures. Thus, we see that cascading failures can have a significant impact on a system's dependability measures, and the differences are even more pronounced for Models 2 and 3. For Model 2, the unavailability and MTTF differ by factors of 16.7 and 8.2, respectively. For Model 3, the unavailability and MTTF differ by factors of 28.8 and 17.2, respectively. The reasons for the larger differences between the results with and without cascading failures for Models 2 and 3 than for Model 1 appear to be as follows. Since Models 2 and 3 use larger repair rates, failed components are repaired more quickly, so other components have less chance of independently failing during the shorter repair times. Because all components of a type need to be failed to bring the system down, it is less likely without cascading failures for both components of one type to be simultaneously failed to bring the system down. However, with cascading failures, it is possible for a single failure to cause other components to fail simultaneously to bring down the system. Hence, changing repair rates has a greater impact on the dependability measures when there are no cascading failures.

6.2 Example 2

We also ran our program on the system depicted in Fig. 8, which can be thought of as an example of a search-engine infrastructure [29]. In this case, $A1, \ldots, A5$ represent query servers, and $B1, B2, B3$ are network switches connecting the servers to computational clusters $C1, \ldots, C4$, which resolve the queries. Each of $A1, \ldots, A5, B1, B2, B3, C1, \ldots, C4$ is a separate component type. Component type $A3$ has redundancy 2 and all other types have redundancy 1.

In this model, there is only a single environment. The failure rates of component types Ai, Bi, and Ci are 0.001, 0.002, and 0.001, respectively. The repair rates of component types Ai, Bi, and Ci are 16, 8, and 2, respectively.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

TABLE 2
Numerical Results for Example 2

Measure	No Cascading	Cascading 1	Cascading 2
Unavailability ($\times 10^{-5}$)	0.315	9.652	43.536
MTTF ($\times 10^{3}$)	135.668	2.559	0.565

The possible cascading failures are denoted by dashed lines in Fig. 8. When either $A1$ or $A2$ fails, it can cause the other to simultaneously fail. A failure of a B-type of component can cause each of the other B-types to simultaneously fail, and a failure of a C-type of component can cause an adjacent C-type component to simultaneously fail. Thus, $\Gamma_{A1} = \{A2\}$, $\Gamma_{A2} = \{A1\}$, $\Gamma_{A4} = \{A5\}$, $\Gamma_{A5} = \{A4\}$, $\Gamma_{B1} = \{B2, B3\}$, $\Gamma_{B2} = \{B1, B3\}$, $\Gamma_{B3} = \{B1, B2\}$, $\Gamma_{C1} = \{C2\}$, $\Gamma_{C2} = \{C1, C3\}$, $\Gamma_{C3} = \{C2, C4\}$, and $\Gamma_{C4} = \{C3\}$.

We considered two different versions of the system by varying the cascading failure probabilities. In the first model, we let $\phi_{A1,A2} = \phi_{A2,A1} = \phi_{A4,A5} = \phi_{A5,A4} = 0.01, \phi_{Bi,Bj} = 0.05$ for all $i \neq j$, and $\phi_{Ci,Cj} = 0.02$ for $j = i - 1$ or $j = i + 1$ with $1 \leq j \leq 4$. In the second model, we increased each of these values by a factor of 4.

The solid lines in Fig. 8 show the interconnections among the components. The query servers $A1, \ldots, A5$ are in three different regions, with $A1$ and $A2$ in the first region, $A3$ in the second region, and $A4$ and $A5$ in the third region. The system is considered operational as long as each region has an operational query server, and there are at least three operational computational clusters that are accessible (through operational switches) to the operational query servers. We now want to give the corresponding Boolean expression. To simplify notation, for $T = A$, B, or C and $i_1, i_2, \ldots, i_n$, let $T_{i_1 i_2 \cdots i_n}$ denote the Boolean expression $\bigwedge_{i=1}^{n} (Ti_n \geq 1)$. For example, A_{135} means $(A1 \geq 1) \wedge (A3 \geq 1) \wedge (A5 \geq 1)$, and B_3 means $(B3 \geq 1)$. Also, let $T_{k:n}$ denote that at least k out of the n T-type components are operational. For example, $B_{2:3}$ means $((B1 == 1) \wedge (B2 == 1)) \vee ((B1 == 1) \wedge (B3 == 1)) \vee ((B2 == 1) \wedge (B3 == 1))$, where $\vee$ denotes OR. Then a Boolean expression for the system to be operational is

$$\begin{aligned} &((A_{134} \vee A_{135} \vee A_{234} \vee A_{235}) \wedge B_{2:3} \wedge C_{3:4}) \\ &\quad \vee (A_{134} \wedge B_3 \wedge C_{134}) \ \vee \ (A_{135} \wedge B_3 \wedge C_{134}) \\ &\quad \vee (A_{135} \wedge B_1 \wedge C_{124}) \ \vee \ (A_{235} \wedge B_1 \wedge C_{124}). \end{aligned}$$

Thus, the system is operational if there is at least one operational query server available in each region, at least two out of the three switches are operational, and at least three out of the four computational clusters are operational. Also, the system is operational if there is only one switch ($B1$ or $B3$) operational but only for certain configurations of at least three operational query servers and at least three operational computational clusters.

The resulting CTMC has $2^{11} \times 3 = 6{,}144$ states. Table 2 gives the values of the steady-state unavailability and MTTF. The column label "Cascading 1" (respectively, "Cascading 2") gives the results with cascading failures having probabilities in Model 1 (respectively, Model 2). We also ran the same model without cascading failures, and the column labeled "No Cascading" contains the values of the dependability measures in this case. Going from no cascading failures to Model 1, the steady-state unavailability (respectively, MTTF) degrades by a factor of 30.6 (respectively, 53.0). The large differences in dependability can be explained as follows. Without cascading failures, the failure of the system requires at least two independent component failures. However, with cascading failures, there are some scenarios having not insignificant probabilities in which a single component failure triggers a cascade that brings the system down (e.g., a C-type failing causing another C-type to fail simultaneously with probability 0.02). Quadrupling the probabilities of cascading failures (i.e., going from Model 1 to Model 2) leads to both dependability measures degrading by a factor of 4.5. Thus, we see that increasing the cascading failure probabilities can have a large impact on the dependability measures. However, the degradation from Model 1 to Model 2 is not as dramatic as going from no cascading to Model 1 because (for our particular values for the cascading probabilities) the introduction of cascading failures into the system has a larger impact on the dependability than increasing the probabilities.

7 Summary and Conclusions

We presented a Markovian dependability model that incorporates cascading failures, which we model as trees. Computation of the infinitesimal generator matrix of the CTMC poses significant challenges because of the exponential growth in the number of trees that need to be considered as the number of failing components in a cascade grows. Specifically, for a failure transition with m components failing simultaneously, there are up to m^{m-1} trees. The algorithms we provide are highly parallelizable with low communication or synchronization overhead. Therefore, the computation of the infinitesimal generator matrix can be sped up, and problems with large state spaces can be solved by parallelizing Algorithm 4 or even Algorithm 2.

Numerical results indicate that cascading failures can significantly alter dependability measures of a system compared to the same system without them. Hence, incorporating cascading failures into models when they can occur is extremely important for accurate evaluation of system designs.

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) under grant NSF/ITR IIS-0324816. The authors thank the anonymous referees, who provided comments that improved the paper. Also, the remark in Section 4.2 was suggested by one of the referees.

References

[1] D.J. Watts, "A Simple Model of Global Cascades on Random Networks," *Proc. Nat'l Academy of Sciences USA,* vol. 99, pp. 5766-5771, 2002.

[2] E. Coffman, Z. Ge, V. Misra, and D. Towsley, "Network Resilience: Exploring Cascading Failures within BGP," *Proc. Allerton Conf. Comm., Control and Computing,* Oct. 2002.

[3] I. Dobson, B.A. Carreras, V.E. Lynch, and D.E. Newman, "Complex Systems Analysis of Series of Blackouts: Cascading Failure, Critical Points, and Self-Organization," *Chaos,* vol. 17, p. 026103, 2007.

[4] J. Chen, J. Thorp, and I. Dobson, "Cascading Dynamics and Mitigation Assessment in Power System Disturbances via a Hidden Failure Model," *Int'l J. Electrical Power and Energy Systems,* vol. 27, pp. 318-326, 2005.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.

[5] K.S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications,* second ed. Wiley, 2001.
[6] J.K. Muppala, R.M. Fricks, and K.S. Trivedi, "Techniques for System Dependability Evaluation," *Computational Probability,* W.K. Grassmann, ed., Kluwer, pp. 445-480, 2000.
[7] A. Goyal and S.S. Lavenberg, "Modeling and Analysis of Computer System Availability," *IBM J. Research and Development,* vol. 31, pp. 651-664, 1987.
[8] A. Goyal, P. Shahabuddin, P. Heidelberger, V. Nicola, and P.W. Glynn, "A Unified Framework for Simulating Markovian Models of Highly Dependable Systems," *IEEE Trans. Computers,* vol. 41, no. 1, pp. 36-51, Jan. 1992.
[9] R.A. Sahner, K.S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems.* Kluwer, 1996.
[10] M.K. Smotherman, J.B. Dugan, K.S. Trivedi, and R.M. Geist, "The Hybrid Automated Reliability Predictor," *AIAA J. Guidance, Control and Dynamics,* vol. 9, pp. 319-331, 1986.
[11] A.K. Somani, J.A. Ritcey, and S.H. Au, "Computationally Efficient Phased Mission Reliability Analysis for Systems with Variable Configuration," *IEEE Trans. Reliability,* vol. 41, no. 4, pp. 504-511, Dec. 1992.
[12] R.W. Butler, "The SURE Reliability Analysis Program," *Proc. AIAA Guidance, Navigation, and Control Conf.,* pp. 198-204, 1986.
[13] S. Bernson, E. de Souza e Silva, and R. Muntz, "A Methodology for the Specification of Markov Models," *Numerical Solution to Markov Chains,* W. Stewart, ed., pp. 11-37, 1991.
[14] C. Beounes, M. Aguera, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J.M. de Souza, D. Powell, and P. Spiesser, "SURF-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing (FTCS-23) Digest of Papers,* pp. 668-673, 1993.
[15] G. Krishnamurthi, A. Gupta, and A.K. Somani, "The HIMAP Modeling Environment," *Proc. Ninth Int'l Conf. Parallel and Distributed Computing Systems,* pp. 254-259, 1996.
[16] C. Hirel, B. Tuffin, and K.S. Trivedi, "Spnp Version 6.0," *Lecture Notes in Computer Science,* vol. 1786, pp. 354-357, 2000.
[17] K.J. Sullivan, J.B. Dugan, and D. Coppit, "The Galileo Fault Tree Analysis Tool," *Proc. 29th Ann. Int'l Symp. Fault-Tolerant Computing,* pp. 232-235, 1999.
[18] M. Walter, M. Siegle, and A. Bode, "OpenSESAME—the Simple but Extensive, Structured Availability Modeling Environment," *Reliability Eng. and System Safety,* vol. 93, pp. 857-873, 2008.
[19] M. Bouissou and J.L. Bon, "A New Formalism that Combines Advantages of Fault-Trees and Markov Models: Boolean Logic Driven Markov Processes," *Reliability Eng. and System Safety,* vol. 82, pp. 149-163, 2003.
[20] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004.
[21] W.G. Bouricius, W.C. Carter, and P.R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems," *Proc. 24th ACM Nat'l Conf.,* pp. 295-309, 1969.
[22] J.B. Dugan and K.S. Trivedi, "Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems," *IEEE Trans. Computers,* vol. 28, no. 6, pp. 775-787, June 1989.
[23] H. Xu, L. Xing, and R. Robidoux, "DRBD: Dynamic Reliability Block Diagrams for System Reliability Modeling," to be published in *Int'l J. Computers and Applications,* 2009.
[24] S. Distefano and L. Xing, "A New Approach to Modelling the System Reliability: Dynamic Reliability Block Diagrams," *Proc. 52nd Ann. Reliability and Maintainability Symp. (RAMSS '06),* pp. 189-195, 2006.
[25] H. Langseth and L. Portinale, "Bayesian Networks in Reliability," *Reliability Eng. and System Safety,* vol. 92, pp. 92-108, 2007.
[26] M. Xie, Y.S. Dai, and K. Poh, *Computing Systems Reliability: Models and Analysis.* Kluwer Academic, 2004.
[27] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. McGraw-Hill, 2001.
[28] D.E. Knuth, *The Art of Computer Programming: Fundamental Algorithms,* third ed. Addison-Wesley, 1997.
[29] L.A. Barroso, J. Dean, and U. Hölzle, "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro,* vol. 23, no. 2, pp. 22-28, Mar./Apr. 2003.

Srinivasan M. Iyer received the MS degree in computer science from the New Jersey Institute of Technology in 2006. He is currently a senior software engineer at Exalead, Inc., where he develops software solutions based on Exalead's search technology. His current research interest is the study and application of search technology in the area of Business Intelligence.

Marvin K. Nakayama received the BA degree in mathematics/computer science from the University of California, San Diego, in 1986, and the MS and PhD degrees in operations research from Stanford University in 1988 and 1991, respectively. He is an associate professor in the Department of Computer Science at the New Jersey Institute of Technology. He has previously held positions at Rutgers University's Graduate School of Management, Columbia Business School in New York, and at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. His research interests include applied probability, statistics, simulation, and dependability modeling. He was the receipient of a second prize in the 1992 George E. Nicholson Student Paper Competition sponsored by INFORMS and a CAREER Award from the US National Science Foundation. He is the stochastic models area editor for *ACM Transactions on Modeling and Computer Simulation* and the simulation area editor for *INFORMS Journal on Computing*.

Alexandros V. Gerbessiotis received the SM and PhD degrees, both in computer science, from Harvard University and has a diploma in electrical engineering from the National Technical University of Athens, Greece. He completed his postdoctoral studies at Oxford University, UK. He is an associate professor in the Computer Science Department, College of Computing Sciences at the New Jersey Institute of Technology. His research interests include high-performance and grid computing, parallel and sequential algorithms, and experimental algorithmics. His research has been supported by various funding agencies including the US National Science Foundation and the US Department of Energy.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: New Jersey Institute of Technology. Downloaded on September 24, 2009 at 14:07 from IEEE Xplore. Restrictions apply.