

Working Title

M. Sanghavi, S. Tadepalli, M. Nakayama

January 2013

1 Model

We work with the stochastic model of [1], which considers the evolution over time of a repairable dependability system operating in a randomly changing environment. The system consists of a collection $\Omega = \{1, 2, \dots, N\}$ of $N < \infty$ component types. Each component type $i \in \Omega$ has a redundancy $1 \leq r_i < \infty$, and the r_i components of type i are assumed to be identical. A component can be either operational (up) or failed (down).

The environment changes randomly within a set $\mathcal{E} = \{0, 1, 2, \dots, L\}$. For example, we can think of the environment as representing the current load on the system, and if there are two possible environments, 0 and 1, then 0 (resp., 1) may represent a low (resp., high) load. Once the environment enters $e \in \mathcal{E}$, it remains there for an exponentially distributed amount of time with rate $\nu_e > 0$, after which the environment changes to e' with probability $\delta_{e,e'} \geq 0$, where $\delta_{e,e} = 0$ and $\sum_{e' \in \mathcal{E}} \delta_{e,e'} = 1$. We assume the matrix $\delta = (\delta_{e,e'} : e, e' \in \mathcal{E})$ is irreducible; i.e., for each $e, e' \in \mathcal{E}$, there exists $k \geq 1$ and a sequence $e_0 = e, e_1, e_2, \dots, e_k = e'$ with each $e_i \in \mathcal{E}$ such that $\prod_{i=0}^{k-1} \delta_{e_i, e_{i+1}} > 0$. In other words, it is possible to eventually move from environment e to environment e' .

The components in the system can randomly fail and then be repaired. When the environment is $e \in \mathcal{E}$, the failure rate and repair rate of each component of type i are $\lambda_{i,e} > 0$ and $\mu_{i,e} > 0$, respectively. If there is only one environment e , i.e., $|\mathcal{E}| = 1$, then the lifetimes and repair times of components of type i are exponentially distributed with rates $\lambda_{i,e}$ and $\mu_{i,e}$, respectively. Exponential distributions are frequently used to model lifetimes of hardware and software components; e.g., see [2]. We assume that all operating components of a type i have the same failure rate $\lambda_{i,e}$ in environment e . Thus, in a system with redundancies for which not all components of a type are needed for operation of the system, the extras are “hot spares” since they fail at the same rate as the main components.

Our model includes probabilistic instantaneous cascading failures, which occur as follows. The ordered set Γ_i specifies the types of components that a failure of a type- i component can cause to simultaneously fail. When a component of type i fails, it causes a single component of type $j \in \Gamma_i$ to fail at the same time with probability $\phi_{i,j}$ (if there are components of type j up) and we call $\phi_{i,j}$ a *component-affected probability*. The events that the individual components types $j \in \Gamma_i$ fail immediately are independent. Thus, when a component of type i fails, there are independent “coin flips” to determine which components in Γ_i fail, where the coin flip for $j \in \Gamma_i$ comes up heads (one component of type j fails) with probability $\phi_{i,j}$ and tails (no components of type j fail) with probability $1 - \phi_{i,j}$.

We allow for a cascading failure to continue as long as there are still components operational in the system. For example, the failure of a component of type i may cause a component of type j to fail (with probability $\phi_{i,j}$), which in turn makes a component of type k fail (with probability $\phi_{j,k}$), and so on. As noted in [1], the SAVE package [3] allows for only one level of cascading, but the unlimited cascading in our model makes it significantly more difficult to analyze.

We can think of a cascading failure as a tree of simultaneously failing components. The root is the component, say of type i , whose failure *triggers* the cascade. The root's children, which are from Γ_i , are those components whose immediate failures were directly caused by the root's failure. At any non-root level of the tree, these components' failures were directly caused by the failure of their parents at the previous level. Although all the failing components in a cascade fail at the same time, we need to specify an order in which they fail for our problem to be well-defined, as we explain later. We assume the components in a tree fail in breadth-first order.

There is a single repairman who fixes failed components using a processor-sharing discipline. Specifically, if the current environment is e and there is only one failed component, which is of type i , then the repairman fixes that component at rate $\mu_{i,e}$. If there are b components currently failed, then the repairman allocates $1/b$ of his effort to each failed component, so a failed component of type i is repaired at rate $\mu_{i,e}/b$.

2 Markov Chain

We want to analyze the behavior of the system as it evolves over time. Because of the processor-sharing repair discipline and the exponential distributions for the event lifetimes, it will suffice to define the state of the system as a vector containing the number of failed components of each type and the current environment. Thus, let $S = \{x = (x_1, x_2, \dots, x_N, x_{N+1}) : 0 \leq x_i \leq r_i \ \forall i \in \Omega, x_{N+1} \in \mathcal{E}\}$ be the state space, and let $Z = [Z(t) : t \geq 0]$ be the continuous-time Markov time (CTMC) living on S keeping track of the current state of the system. (If we had instead considered a first-come-first-served repair discipline, then the state space would need to be augmented to keep track of the order in which current set of down components failed.)

We assume that Z starts in environment $0 \in \mathcal{E}$ with no components failed, i.e., state $(0, 0, \dots, 0)$, and we now describe the CTMC's infinitesimal generator matrix $Q = (Q(x, y) : x, y \in S)$, where $Q(x, y)$ is the rate that the CTMC Z moves from state $x = (x_1, \dots, x_N, x_{N+1})$ to state $y = (y_1, \dots, y_N, y_{N+1})$. If $y_i = x_i$ for each $i \in \Omega$ and $y_{N+1} \neq x_{N+1}$, then (x, y) is an *environment transition* with $Q(x, y) = \nu_{x_{N+1}} \delta_{x_{N+1}, y_{N+1}}$. If $y_i = x_i - 1$ for one $i \in \Omega$, $y_j = x_j$ for each $j \in \Omega - \{i\}$, and $y_{N+1} = x_{N+1}$, then (x, y) is a *repair transition* corresponding to the repair of a component of type i , and $Q(x, y) = x_i \mu_{i, x_{N+1}} / (\sum_{j \in \Omega} x_j)$. If $y_i \geq x_i$ for all $i \in \Omega$ with $y_j > x_j$ for some $j \in \Omega$ and $y_{N+1} = x_{N+1}$, then (x, y) is a *failure transition* in which $y_i - x_i$ components of type i fail, $i \in \Omega$. Any other (x, y) with $x \neq y$ not falling into one of the above three categories is not possible, so $Q(x, y) = 0$. The diagonal entry $Q(x, x) = -\sum_{y \neq x} Q(x, y)$, as required for a CTMC.

We now determine the rate $Q(x, y)$ of a failure transition (x, y) . First consider the case when there is no cascading failures possible, i.e., $\Gamma_i = \emptyset$ for each type i . Then the only

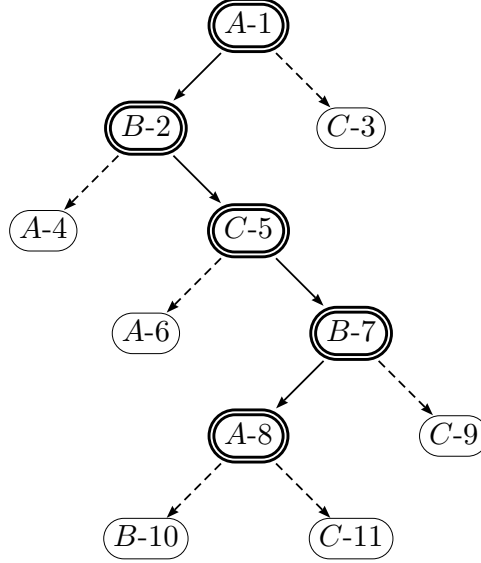


Figure 1: An example of a supertree.

possible failure transitions (x, y) have $y_i = x_i + 1$ for one $i \in \Omega$, $y_j = x_j$ for each $j \in \Omega - \{i\}$, and $y_{N+1} = x_{N+1}$, and this corresponds to a single component of type i failing. Then $Q(x, y) = (r_i - x_i)\lambda_{i, x_{N+1}}$.

Cascading failures complicate the computation of $Q(x, y)$ for a failure transition (x, y) . As mentioned before, a cascading failure is modeled as a tree T built from the multiset B of simultaneously failing components, where B has $y_\ell - x_\ell$ failing components of type ℓ , $\ell \in \Omega$. A tree T has a rate $R(T)$ determined by the failure rate $(r_i - x_i)\lambda_{i, x_{N+1}}$ of the root (assumed here to be of type i), as well as the product of $\phi_{j,k}$ for a parent node of type j causing a child of type $k \in \Gamma_j$ to fail, and $1 - \phi_{j,k}$ from a node of type j *not* causing a component of type $k \in \Gamma_j$ to fail when there are components of type k up. A difficulty arises since there can be many such trees corresponding to the multiset B of components failing in (x, y) , and determining $Q(x, y)$ requires summing $R(T)$ over all possible trees T that can be constructed from B . The number of such trees grows exponentially in the number of failing components in the cascade; see [1].

[The following is not complete and will need to be changed for another example.] Computing the rate $R(T)$ of a tree T requires that the components fail in a certain order, even though all of the failures occur simultaneously. To see why, consider the following example. Let $\Omega = \{A, B, C\}$, with $r_A = r_B = r_C = 4$. Also, let $\Gamma_A = \{B, C\}$, $\Gamma_B = \{A, C\}$, and $\Gamma_C = \{A, B\}$. Suppose that $\mathcal{E} = \{0\}$, and consider the failure transition (x, y) with $x = (2, 2, 2, 0)$ and $y = (4, 4, 3, 0)$. Thus, (x, y) corresponds to 2 components each of types A and B failing and a single component of type C failing. One possible tree T corresponding to (x, y) is shown in Figure 1.

The nodes depicted as double circles form the tree of failing components. The single circles correspond to components in some Γ_i set but did not fail. A component type j in some Γ_j could have not failed because there are components of type j up at this point but

its coin flip came up tails (with probability $1 - \phi_{i,j}$), or there were no more components of type j up at this point. Each node has a label of the form $t-i$, where t denotes the type of the component for that node, and i is the ID, which is the position of the node in a breadth-first ordering of all the nodes (single and double circles). We include the IDs just to simplify the discussion here. We call the tree of all nodes the *supertree* corresponding to the tree T of failing nodes. The supertree is used to compute the rate $R(T)$ of T , as follows. Since the root is a component of type A and there are $r_A - x_A = 2$ components of type A at the start of the transition (x, y) , the rate of the trigger of the cascade is $2\lambda_{A,0}$. The root then causes a component of type B to fail at node ID 2, and this occurs with probability $\phi_{A,B}$. The node at ID 3 did not fail, and at this point there are $r_C - x_C = 2 > 0$ components of type C still up, so this non-failure occurs with probability $1 - \phi_{A,C}$.

3 Algorithms

We now provide efficient algorithms for generating all possible trees and constructing the Q -matrix. A tree corresponds to a multiset of particular components failing, and cascading failures starting from different states can have the same multiset of components failing. Hence, a particular tree may correspond to several different transitions. Our algorithm generates each possible tree only once and determines all the transitions to which this tree corresponds. This avoids generating the same tree numerous times for each corresponding transition, as was originally done in [1]. Moreover, rather than building each new tree from scratch, as was done in [1], our current algorithm builds larger trees from smaller ones already considered, leading to additional savings in the overall computational effort.

Computing the rate of a given tree depends on the state from which the cascading failure began and the multiset of components that fail in the cascade. We also consider the supertree from a given tree by adding in the components that did not fail from the Γ_i that were used, which is necessary to compute the rate of a tree. We do not actually construct the supertree in our algorithm, but instead build a *breadth-first history* to keep track of the information necessary to compute a tree's rate. A breadth-first history accounts for the contribution to the tree's rate of the components that could have failed (i.e., belong to the set Γ_i of a failed node of type i), but did not. We explain the process of starting a tree, building its breadth-first history and computing its rate in Algorithms 1(SeedTrees), 2 (AddTreeLevel) and 3 (ComputeTreeRate), respectively.

SeedTrees starts the tree generation and initializes the necessary data structures for AddTreeLevel. AddTreeLevel adds a new level to an existing tree in a recursive fashion, updates the cumulative failed probability for the tree for the components that actually failed in the cascade; as well as builds the tree's breadth-first history. ComputeTreeRate computes the rate of a completed tree for all the transitions it corresponds to using the cumulative failed probability and the breadth-first history populated in AddTreeLevel. We will now discuss each algorithm in detail. References to line numbers in the algorithms are given within angled brackets $\langle \rangle$.

3.1 SeedTrees

The data structure Γ consists of $(\Gamma_i : i \in \Omega)$. We start by iterating through Ω to choose a root component with type, *rootC.type*, for an initial tree with one node $\langle 1 \rangle$. We then initialize the following data structures $\langle 2 - 4 \rangle$:

- *level*, a dynamic array to hold the failed nodes in the current bottom level of the tree (in breadth-first order);
- *nFailed*, an array that counts the number of failed components of each type in the tree; and
- *BFHist*, a data structure that is the breadth-first history of a tree.

BFHist is implemented as an array of linked lists; where the array is indexed by component type. Each linked list stores the respective parents of nodes exclusive to the supertree (i.e., each node that *did not* fail but could have if its component type's redundancy was not exhausted) and stores the symbol @ for the nodes that *did* fail in the tree. *BFHist* plays the role of a supertree in determining whether to include the complement probabilities of nodes that did not fail. A complement probability is included for a node that did not fail only if there are still components of its type available in the system at that point. To determine the number of components available of a certain type, we first consider a CTMC transition (x, y) corresponding to the tree we are building, where we call x the from-state and y the to-state in the transition. Then the number of available components of a type i is computed as $r_i - n_i(x) - nFailed[i]$, where $n_i(x)$ is the number failed of type i in state x , and $nFailed[i]$ is the number failed of type i in the tree thus far in breadth-first order $\langle 2 - 4 \rangle$.

Since the root must fail, we initialize *level* with *rootC.type*. We add the symbol @ to the list at *BFHist[rootC.type]* to denote that one component of the root's type has failed. We update *nFailed* by setting the counter for *rootC.type* to 1 $\langle 5 - 7 \rangle$.

If the component type, *rootC.type*, at the root cannot cause any other components to fail, only the trivial tree of one node can be made with this type of root. We then evaluate this single-node tree's rate in *ComputeTreeRate* because it cannot be grown further. If $\Gamma_{rootC.type} \neq \emptyset$, i.e., it can cause other types of components to fail, then we call *AddTreeLevel* to proceed with building taller trees by adding another level to the current tree $\langle 8 - 12 \rangle$.

To explain the parameters chosen in subroutines *ComputeTreeRate* and *AddTreeLevel*, we need to understand how a tree's rate is computed. The total transition rate of a completed tree consists of three parts: the failure rate of the root; cumulative product of the $\phi_{i,j}$'s computed in *AddTreeLevel*, which we denote by ρ ; and the cumulative product of the complement probabilities of components that could have failed but did not, which we denote by η . The failure rate of the root for a transition (x, y) corresponding to a tree is given by the $n_i(x) \lambda_{i, X_{N+1}}$ term, where i is *rootC.type*. Since $\phi_{i,j}$'s do not depend on the transition (x, y) , we can compute ρ as we build a tree in *AddTreeLevel*. Computing ρ entails multiplying $\phi_{i,j}$ for all edges from a parent of type i to a child of type j in the tree. The cumulative product of complement probabilities consists of product terms of the form $1 - \phi_{i,j}$ for all edges from parent of type i to a child of type j in the supertree, where failures did not occur but could have because of components of type j were still up. We multiply the $1 - \phi_{i,j}$ terms and

the $n_i(x) \lambda_{i, x_{N+1}}$ term later in `ComputeTreeRate` as they depend on the transition (x, y) . Since the current tree has only one node, there are no edges from a parent of type i to a child of type j in this tree. Hence, `SeedTrees` passes 1 as the value for ρ to the subroutines `AddTreeLevel` and `ComputeTreeRate`.

Algorithm 1 `SeedTrees(Γ)`

where Γ is an array of ordered sets that describes which components can cause which other components to fail

```

1: for  $rootC.type \in \Omega$  do
2:    $level = []$ ; {Dynamic array of failed components at tree's current bottom level}
3:    $nFailed = [0, 0, \dots, 0]$ ; {Array that counts failed components of each type in the tree}
4:    $BFHist = [(), (), \dots, ()]$ ; {Array of linked lists that keeps a history of parent component types in breadth-first order; BFHist is indexed by component types}
5:   add  $rootC.type$  to  $level$ ;
6:    $nFailed[rootC.type] = 1$ ;
7:   add @ to  $BFHist[rootC.type]$ ; {Signifies one component of type  $rootC.type$  has failed}
8:   if  $\Gamma_{rootC.type} == \emptyset$  then
9:     ComputeTreeRate( $nFailed, BFHist, 1, rootC.type$ );
10:  else
11:    AddTreeLevel( $level, nFailed, BFHist, 1, rootC.type$ );
12:  end if
13: end for

```

3.2 AddTreeLevel

Given a tree, `AddTreeLevel` determines all the possibilities for the next level that can be added to the current tree by taking the Cartesian product of the power sets of the Γ_i for each of the failed components at the current bottom level. For each next level we recursively call `AddTreeLevel`. We use \mathcal{P} to denote the power set-like operation on an ordered set. In our realization of the power-set operation, components in each subset maintain their relative ordering from the original set.

We implement this power set-like operation by generating all possible binary numbers with $\sum_{i=1}^{|level|} \Gamma_{level[i]}$ bits. A total of $\prod_{i=1}^{|level|} 2^{\Gamma_{level[i]}}$ such binary numbers are generated. In the binary number, 1 denotes a failed node and 0 denotes a node that is included in a Γ set but did not fail. If it so happens that we have a tree where none of the leaf nodes at the current bottom level can cause any other components to fail (i.e., have empty Γ_i s), we get no nextLevelPossibilities $\langle 1 \rangle$. If there are no nextLevelPossibilities, the tree cannot grow

any further, so we would proceed to `ComputeTreeRate`.

Otherwise, we choose one possible choice for the failed components in the next level to work with from the `nextLevelPossibilities` $\langle 2 \rangle$. To find out whether any new children will be added in the upcoming next level, we create a Boolean variable *addedChildFlag*, initially with the value of `False` $\langle 3 \rangle$. For each node *parentC* in the current bottom level that acts as a parent, potentially causing other nodes to fail, we iterate through all of its possible children, i.e., through $\Gamma_{parentC}$. If any of these children actually fail, then they will be members of the set *oneNextLevelPossibility* $\langle 4 - 6 \rangle$. Now if the redundancy of the component type we just tried to add as a child has already been exhausted, it cannot fail and hence our tree is invalid and we move on to the next possibility $\langle 7 - 9 \rangle$.

If a child of *childC.type* has failed (i.e., it exists within `nextLevelPossibilities`), the relevant data structures are updated to signify this occurrence. We increment the number of failed components of *childC.type*, add the symbol @ to *BFHist*[*childC.type*] (to mark that a failure has occurred at this location in the tree) and update tree rate with $\phi_{i,j}$ $\langle 11 - 13 \rangle$. If the child does not actually fail but could have (because there are still operational components of *childC.type* at this point), then we add *parentC.type* to *BFHist*[*childC.type*] $\langle 14 - 16 \rangle$.

We do the above updates to data structures for each potential parent node in *level* and each of its children in *oneNextLevelPossibility*. If at least one child has been added, we recursively add another level to the current tree $\langle 19 - 23 \rangle$. Otherwise, this tree has not changed in this pass through `AddTreeLevel` and we call `ComputeTreeRate` to compute the rate of the finalized tree $\langle 24 - 26 \rangle$. We make sure that trees are not double counted because once a tree passes through `AddTreeLevel` unmodified, it is processed and discarded. We avoid making duplicate trees because each *oneNextLevelPossibility* is unique.

Algorithm 2 `AddTreeLevel(level, nFailed, BFHist, ρ , rootC.type)`

where *level* is the current level of failed components,

nFailed counts failed components by type in the tree,

BFHist is breadth-first history,

ρ ρ is a cumulative product of component-affected probabilities (except the root),

rootC.type is the root component's type in the current tree

```

1: nextLevelPossibilities =  $\prod_{i=1}^{|level|} \mathcal{P}(\Gamma_{level[i]});$ 
   {Builds all possibilities for the next level (given the current level) by taking a Cartesian
   product of the power sets of non-empty  $\Gamma$  sets of failed nodes in the current level}
2: for oneNextLevelPossibility  $\in$  nextLevelPossibilities do
3:   addedAChildFlag = False;
4:   for parentC  $\in$  level do
5:     for childC  $\in$   $\Gamma_{parentC}$  do
6:       if childC  $\in$  oneNextLevelPossibility and childC  $\in$  then
7:         if nFailed[childC] == rchildC then
8:           goto line 3; {Invalid Tree, it requires more components than available}
9:         end if
10:      addedAChildFlag = True;
```

```

11:       $nFailed[childC] = nFailed[childC] + 1;$ 
12:      add @ to  $BFHist[childC]$ ; {One component of type  $childC$  has failed}
13:       $\rho = \rho * \phi_{parentC, childC};$ 
      {Update rate with appropriate component-affected probability}
14:    else
15:      add  $parentC$  to  $BFHist[childC]$ ; {One component of type  $childC$  has not failed,
      but was present in  $\Gamma_{parentC}$ }
16:    end if
17:  end for
18: end for
19: if  $addedAChildFlag$  then
20:   AddTreeLevel( $oneNextLevelPossibility, nFailed, BFHist, \rho, rootC.type$ );
   {Tree can be grown further}
21: else
22:   ComputeTreeRate( $nFailed, BFHist, \rho, rootC.type$ );
   {Current tree is complete because it cannot be grown further}
23: end if
24: end for

```

3.3 ComputeTreeRate

In ComputeTreeRate, for a given tree, we determine all of the transitions (x, y) in the Q -matrix that use this tree, and then update the total rates of each of those transitions (x, y) by adding in the rate of the current tree to the current rate for (x, y) . However, even if several transitions use the same tree, the rate of the tree may differ for those transitions, since the failure rate of the root and cumulative product of complement probabilities are transition dependent.

Let $x' = (x_1, x_2, \dots, x_N)$ and $S' = (x_1, x_2, \dots, x_N) : 0 \leq x_i \leq r_i, \forall i \in \Omega$. We only loop through x' and not x because the structure of a tree does not depend on the environment; only the tree's rate does $\langle 1 \rangle$. For each component type i in Ω , we calculate the number of components that are available in the system, c_i , as $r_i - n_i(x)$. The number c_i determines until which point, while traversing through the $BFHist[i]$, we can still have components of type i that could have failed but did not fail $\langle 4 \rangle$. We cannot have components of a type that could have failed but did not after a point in the tree where its type has been exhausted. A type is exhausted when the total number of failed components of the type (i.e., the sum of the components already failed in x' and in the tree in breadth-first order until the current node in the $BFHist[i]$, equals the redundancy of the type. Each time we encounter the symbol @ in the $BFHist$, we reduce the number available by one $\langle 5 - 7 \rangle$. If there are components still available, we update η $\langle 8 - 9 \rangle$. As soon as c_i reaches zero, we break because that type has then been exhausted, i.e., there are no more components that could have failed for the type $\langle 10 - 12 \rangle$.

Once we have finished calculating η , we loop through all environments since the same tree can be used for transitions occurring in different environments $\langle 15 \rangle$. We convert x'

into a state x , by appending the environment e $\langle 16 \rangle$. We generate a state y by adding $nFailed$ to x' and appending environment e $\langle 17 \rangle$. Some invalid ys will be generated in some instances, because simply adding $nFailed$ will cause component types' number failed to exceed their redundancies $\langle 18 - 20 \rangle$. The failure rate of the root is calculated as the number of components of type $rootC.type$ up in the system, multiplied by the failure rate of the component in the environment of x $\langle 21 \rangle$. With all parts of the rate calculation done, we add the rate of the current tree to the current cumulative rate for transition (x, y) in the Q -matrix $\langle 22 \rangle$.

Algorithm 3 ComputeTreeRate($nFailed$, $BFHist$, ρ , $rootC.type$)

where $level$ is the current level of failed components,
 $nFailed$ counts failed components of each type in the tree,
 $BFHist$ is breadth-first history,
 ρ is a cumulative product of component-affected probabilities (except the root),
 $rootC.type$ is the root's component type in the tree

```

1: for  $x' \in S'$  do
2:    $\eta = 1$ ; {Cumulative product of complement probabilities of components that could
   have failed but did not}
3:   for  $i \in \Omega$  do
4:      $c_i = r_i - x'[i]$ ;
5:     for  $parentC$  in  $BFHist[i]$  do
6:       if  $parentC == @$  then
7:          $c_i = c_i - 1$ ;
8:       else if  $c_i > 0$  then
9:          $\eta = \eta * (1 - \phi_{parentC, i})$ ;
10:      else
11:        break; { $c_i$  must be greater than 0 or else there cannot be any more failed nodes
        of type  $i$ }
12:      end if
13:    end for
14:  end for
15:  for  $e \in \mathcal{E}$  do
16:     $x = (x', e)$ ;
17:     $y = (x' + nFailed, e)$ ;
18:    if  $y$  is not a valid state then
19:      continue;
20:    end if
21:     $rootFailureRate = (r_{rootC.type} - x[rootC.type]) * \lambda_{rootC.type, x_{N+1}}$ ;
22:     $Q(x, y) = Q(x, y) + rootFailureRate * \rho * \eta$ ;
23:  end for
24: end for

```

4 Experimental

4.1 Nested Experimental

References

- [1] S. M. Iyer, M. K. Nakayama, and A. V. Gerbessiotis, “A Markovian dependability model with cascading failures,” *IEEE Transactions on Computers*, vol. 139, pp. 1238–1249, 2009.
- [2] M. Xie, Y. S. Dai, and K. Poh, *Computing Systems Reliability: Models and Analysis*. New York: Kluwer Academic, 2004.
- [3] A. Blum, P. Heidelberger, S. S. Lavenberg, M. K. Nakayama, and P. Shahabuddin, “Modeling and analysis of system availability using SAVE,” in *Proceedings of the 23rd International Symposium on Fault Tolerant Computing*, pp. 137–141, 1994.