Reliability Modeling of Structured Systems: Exploring Symmetry in State-Space Generation *

Arun K. Somani
Department of Electrical Engineering, and
Department of Computer Science and Engineering
University of Washington, Box 352500
Seattle, WA 98195-2500
Tele: (206) 685-1602
email: somani@ee.washington.edu

Abstract

A large number of systems are implemented using regular interconnected topologies. Markov analysis of such systems results in large state spaces. We explore symmetry, in particular rotational and permutational, of such systems to achieve a significant reduction in the size of the state space required to analyze them. The resulting much smaller state spaces allow analyses of very large systems. We define equivalent classes of states and develop an algorithm to generate small state spaces and the corresponding Markov chain for systems with permutation symmetries. The state space generation process is also simplified. We demonstrate our technique using several examples. Our technique is very useful in the exact analysis of large systems.

*This research was supported in part by NASA under NAS1-19480 while the author was on sabbatical at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

A large number of systems are implemented using a regular interconnected topology. Regularity is defined in terms of connectivity of a node, i.e., degree of a node and connections to its neighbors. A structure is regular with respect to a function if the nodes of the structure are renumbered using that function and there exists a link renumbering such that the resulting structure is isomorphic to the original topology. In such a case, the relative position of a node is of no importance. Any of the nodes can be numbered as node 1. For example, in a two-dimensional torus interconnected structure, as shown in Figure 1, each node has a degree of four. Each node (i, j) is connected to nodes $i - 1$ and $i + 1$ in ith dimension and nodes $j - 1$ and $j + 1$ in the jth dimension. Any of the nodes can be numbered as node (0,0). The structure can be redrawn and is isomorphic to the original structure. There are several examples of regular structures. In the analysis of such systems one need not be concerned with the individual configuration of the system but all isomorphic configurations can be treated as a single class. Another example of a structured system is an n-dimensional binary cube or in general any k-ary n-cube. In some of the structures, we may have regularity in only certain dimensions. For example, in a two dimensional mesh if one dimension is connected as a ring but the second dimension is only a linear array then regularity is present in the dimension with the ring structure only.

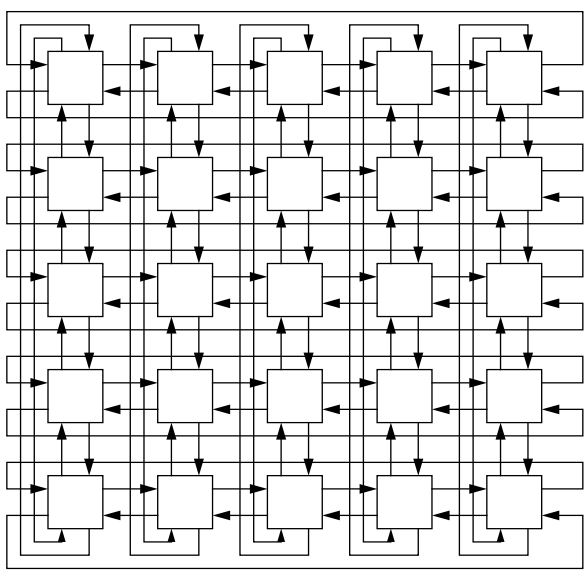

Figure 1: A 5×5 torus network.

Another variation in these structures could be that there is a control node connected to all nodes of the structures belonging to a particular dimension or dimensions. The control node affects the operations of all the nodes it is connected to in an identical fashion. An example of such a class of regular structures is processors/memories or processors/processors connected using a crossbar switch. One processor or one memory is connected to each column or row. We use the terminology processor/memory just to distinguish the column and row connections in the crossbar system. Without the loss of generality, we assume that processors are connected to column connections in the crossbar and memories are connected

to row connections. The model of crossbar we use here is that a cross point connects a column processor to a row memory. If a particular processor fails then all the cross points connecting it to different memories are no more useful for the system operation. Similarly, if a memory fails, the same holds for the cross points connecting that memory to any processor. In the reverse direction, if all cross points in a row fails then the processor cannot perform any meaningful operation. The same relationship holds for a memory and the corresponding cross points in that column.

1.1 Motivation

In the analysis of a system for reliability or performance, our goal is to identify situations or probabilities of occurrence of such situations when a desired configuration is not available due to failures or other reasons. There are performance implication for each system configuration. For example, consider a scenario where we are interested in knowing, when P' out of P processors and Q' out of Q memories are not available if processor, memories, or cross points can fail individually. In the analysis, we need to consider all scenarios when the system fails to deliver the required performance. Since the components can fail in any order and at any location, all combinations of failure must be considered. In a P processors, Q memories, and P*Q cross points system, there are P+Q+P*Q components. A large number of cross point failures can be tolerated before the system really fails. Moreover, with each failure, we may like to model some fine behavior relevant at the time of fault occurrence and system reconfiguration. So a Markov chain may be generated where each state in the Markov chain corresponds to one state of the system. This chain can be analyzed to compute the metrics of interest. Analysis of such systems, combinatorially or otherwise, results in a large number of system states which must be considered. In particular, if we assume a two state model for each component, i.e., processor, memory, or cross point, we may have up to $2^{(P+Q+P*Q)}$ states and a large number of them are operational states. It can be easily seen that the analysis becomes tedious even for a small number of components.

To simplify the analysis, most analysts assume that either cross points failure may not be very critical as there are large numbers of them, and therefore, only consider processor and memory (end point) failure. Another approximation in analysis is made by considering the three subsystems independently. The probabilities of required numbers of processors, memories, and cross points being available are calculated separately and approximate results are synthesized. But, when finer behavior is to be included in the model, such as a fault and error handling model [16] for cross points, then we need to perform the analysis using Markov methods and generate a Markov chain, keeping track of all possible states. A moderate 4×4 systems with eight end points has 2^{24} states. Any larger system than that is almost impossible to analyze. Such difficulties are demonstrated using the example of analysis of a fault tolerant system described in the next section.

Regular structures, however, have symmetry which must be exploited for state space generation. With the symmetry, the number of states can be significantly reduced while fine behavior can be modeled at each transition if so desired. We develop techniques to generate efficient state space exploring of such symmetries and demonstrate the performance and effectiveness of our approach.

To motivate the problem further, we first give an example of a fault tolerant system in Section 2 and show how the number of states can be reduced. Then we develop a

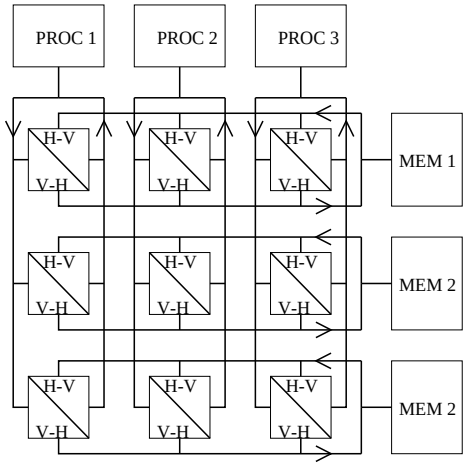

Figure 2: Meshkin Architecture

general technique to generate a smaller state space exploiting the symmetry of the structure. In particular, we look at n-dimensional structures and exploit permutational symmetries. Similar techniques can be applied to rotational or any other well-defined symmetry. Basic classes of permutations of such structures are defined and we develop an algorithm to reach the basic class from a given arbitrary permutation in Section 4.2. We show the possible state space reduction using several examples and list some basic permutations for 3×3, and $2 \times 2 \times 2$ torus-connected systems in Section 4.3. Then we present the results for a larger (4×4) system. Finally our conclusions are presented in Section 5.

2 An Example Fault Tolerant System

Component redundancy is used to achieve high reliability and fault tolerance [7, 8] to tolerate many failures before the system services are not available. Since near coincidence failure may make the system fail, it is necessary to model the fine behavior of the system using a Markov chain. Naive modeling of such systems results in a state space that could be extremely large. For example, Figure 2 shows a fault tolerant system using a mesh-based voter. It has a triple modular redundancy. Three processor modules and three memory modules are interconnected using a mesh (cross bar equivalent) interconnection. These interconnection units are called Bus Interface Units (BIUs). The operation is as follows:

1. For each data request, all processors generate a request synchronously, using their respective buses. One V-H unit on each horizontal bus transfers the request to that horizontal bus. This unit is called a master unit. Other V-H units (called checker units) on that bus compare the request of their respective vertical and horizontal bus and determine if the request being transmitted on the horizontal bus is correct. If any inconsistency is detected, the mesh enters a reconfiguration mode using mesh control signals. Otherwise the memory is allowed to generate the response.

2. When memories generate responses, one H-V unit on each vertical bus transfers that response to that vertical bus and to the processor. This unit is the master unit on the vertical bus. The other V-H units (checker units) on each vertical bus compare their

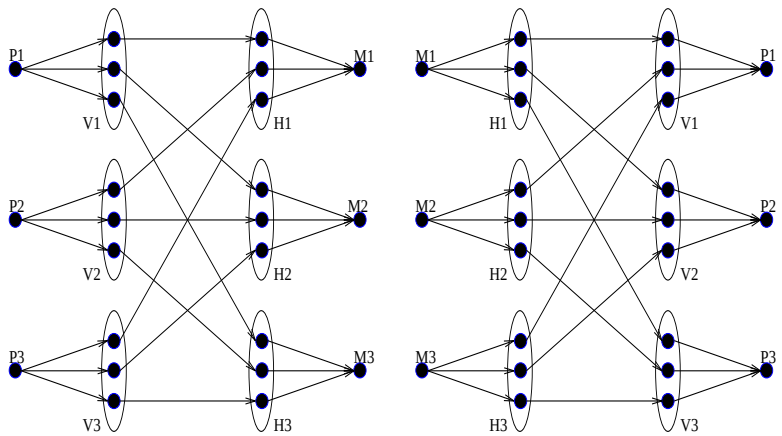

Figure 3: A Graph Model for Meshkin

respective responses with the response being transmitted by the master unit. Again if an inconsistency is found, the mesh enters a reconfiguration mode.

3. As long as sufficient units are available on each bus (to be determined by the actual implementation and fault tolerance requirement) the bus operation can continue.

4. Any unit can be a master unit as long as its source of data is from an operational processor or memory. A failed processor/memory results in effective failure of the those BIUs to which it is connected as source of data. Thus, a processor failure effectively fails the corresponding V-H unit. Similarly, a memory failure effectively fails the corresponding H-V units.

   The V-H and H-V unit at a location in mesh may be two independent units or the same unit (operationally). In the latter case, failure of one implies failure of the other. If a processor has failed it does not matter if the H-V units connected to its bus are operational or not. Similarly, if a memory has failed, it does not matter if the V-H units connected to its bus are operational or not.

   The system has the following implied failures:

   (a) Failure of a processor (memory) implies failures of all V-H (H-V) units connected to it.

   (b) Failures of all V-H (H-V) units connected to a horizontal (vertical) bus imply failure of the corresponding memory (processor).

The communication mechanisms in the Meshkin architecture can be specified using a graph model as shown in Figure 3. A path in the graph is up if every component on that path is up. If a component fails, then all paths which use that component fail. The success and failure criteria are specified using the availability of a set of paths.

Due to the richness of the interconnection, this architecture can tolerate a large number of BIU failures and still may be operational. Suppose the success criteria is that as long as one processor can communicate with one memory and vice-versa, the system remains operational. In one actual implementation of this architecture [8], BIUs have their own fault detection and isolation circuitry to allow a single channel operation. Thus in one specific scenario, out of $3 + 9 + 9 + 3$ component system, as long as some particular four of them

36 States in 3 x 3 Mesh

U U U	D U U	D D U	D U U	D U U	D D D
U U U	U U U	U U U	D U U	U D U	U U U
U U U	U U U	U U U	U U U	U U U	U U U
0	1	2	3	4	5
D D U	D D U	D U U	D U U	D U U	D D D
D U U	U U D	D U U	D U U	U D U	D U U
U U U	U U U	D U U	U D U	U U D	U U U
6	7	8	9	10	11
D D U	D D U	D D U	D D U	D D U	D D U
D D U	D U D	D U U	D U U	D U U	U U D
U U U	U U U	D U U	U D U	U U D	U U D
12	13	14	15	16	17
D D D	D D D	D D D	D D U	D D U	D D U
D D U	D U U	D U U	D D U	D D U	D U D
U U U	D U U	U D U	D U U	U U D	D U U
18	19	20	21	22	23
D D U	D D D	D D D	D D D	D D U	D D U
D U D	D D D	D D U	D D U	D D U	D D U
U D U	U U U	D U U	U U D	D D U	D U D
24	25	26	27	28	29
D D U	D D D	D D D	D D D	D D D	D D D
D U D	D D D	D D U	D D U	D D D	D D D
U D D	D U U	D D U	D U D	D D U	D D D
30	31	32	33	34	35

Figure 4: 36 States of Mesh to Model Meshkin

(one processor, one memory and the two BIU units connecting them) are operational, the system is operational. A naive model generates more than a million states. Even a smart fault tree model of this system will still generate more than 25,000 states for this system.

Since the model is so symmetric, after considering implied failure, we may simply track the status all the BIUs in the mesh. As long as there is one V-H unit along with its corresponding H-V unit operational, the system is operational. To demonstrate this in the paper and to keep things manageable, we make a further simplification to keep the state space even smaller. We assume that a V-H and the corresponding H-V units are tightly coupled for failures and failure of one implies failure of the other. This reduces the number of combinations which we need to track. Even this simplification generated more than 10,000 states using a tool like HARP [16] where input was specified using a fault tree after modifying the fault tree. We encourage readers to try their favorite tool and report to us if they get any significant reduction in number of states using the input language of the tool without making any simplifying assumptions.

Two permutation of mesh states are equivalent if the net effect is the same. For example, in a fully operational system, failures of individual processors are indistinguishable. The same applies for memories as well as for the first BIU failure. However, this cannot be modeled as redundancy in the fault tree as individual BIUs affect the processors and memory differently. For example, failure of two BIUs in a column and the memory element in the row corresponding to the operation BIU essentially makes the processor in that column unusable but another memory element failure does not create the same situation. Thus all situations

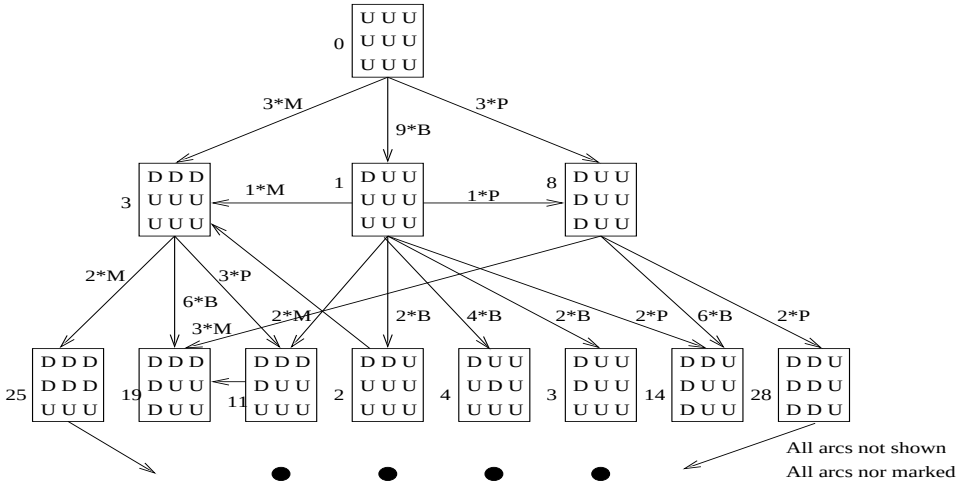

Figure 5: Markov Chain for Meshkin

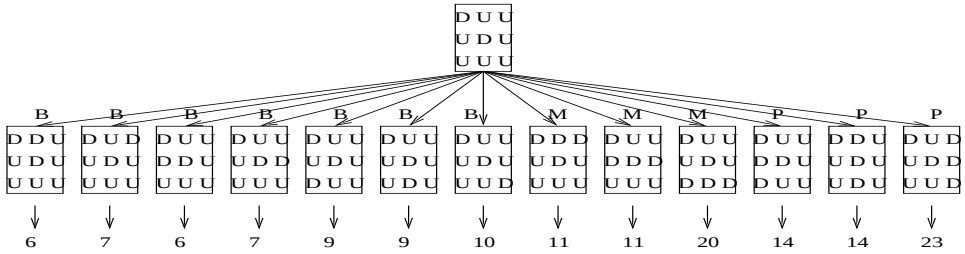

Figure 6: State Expansion in Markov Chain

with failures of two BIUs and one memory element are not identical. The system state is defined by the exact fault locations. There could be situations where the system states are equivalent. If such equivalences are accounted for one can then model the system using 36 states, as shown in Figure 4.

In this figure, a U means the corresponding BIU is up along with its source of data. A D means the BIU or its source is down. Any state is equivalent to one of the permutation of one of the states. A permutation here can be obtained by exchanging rows or columns or both. Thus a state with three row vectors (U U D; U U D; U U U) is equivalent to (D U U; D U U; U U U) and so on. Note that the performance of the system in the two states that are equivalent is identical and, therefore, the reward associated with these states is identical in performability modeling.

A markov chain can then be generated, as shown in Figure 5. We do not show all the arcs or label all of them, but it is easy to see that the model reflects the exact behavior of the system and is sufficient for further analysis. For example, in Figure 6 we expand state 4 by injecting one fault at a time. It can sustain 7 BIU failures, three processor failures, and 3 memory element failures. Thus there are 13 possible next states. However, several of these states are equivalent, as shown in Figure 6. Note that, in the process, we also get a transition rate from one state to another state. For example, in Figure 6, transition rate from state 4 to state 6, 7, and 9 is 2B each and to state 10 it is B. Similarly to state 14, and 23, the transition rates are 2P and P, respectively. In case any of these components are

repairable, the reverse transition can also be generated from any of the next states to state 4 with the appropriate rate. This system may otherwise require up to 2^{15} states. A significant reduction (36 instead of 2^{15}, several orders of magnitude) can be achieved. Sometimes, we may need to keep track of some other states and that depends on what particular metric is of importance in the analysis.

3 How to Get the Reduction

In order to get the reduction there are several helpful hints a system analyst can provide. In the example above, the interaction among processors, memories, busses, and bus interface modules is very regular.

For an n-dimensional structure, to explore permutational and/or rotational symmetry, we need to specify the following parameters for the efficient state space generation:

1. The structure of system interconnection using a multi-dimensional structure format specifying connectivity of the system and permutation and rotational symmetry.
2. End connection (control) nodes affecting the nodes in one or more dimensions specifying depend or affect operators.
3. Group vs individual affects: How a component or a group of components behaves together.
4. Distinguishable and non-distinguishable component failures and reward structure in each case.
5. Possible structure of a state tuple (what components need be specifically tracked or need not be tracked in a state tuple).

The nodes whose individual failures need not be distinguished can be specified as a group as is the case in modeling using fault trees. Language constructs such as *depends on* or *affects* can be used to specify the implied failures. The structure of the system can be specified using a multidimensional structure. The permutation symmetries can be specified for each dimension. The exact nature of this specification is still being researched and is beyond the scope of this paper. Our ultimate goal is to develop a language to be able to specify such behaviors as completely as possible and is part of our future research. We restrict ourselves to simply demonstrate that such a specification can and does lead to the significant reduction in the size of the state space.

The size of the state space also depends on what information needs to be tracked and what can be hidden. For example, in the Meshkin system the state space representation includes only the states of the components in the two dimensional mesh. The end connection nodes (processors and memories) are not represented because we are not interested in tracking their individual states. In some other situations, we may be interested in distinguishing between a row with all BIUs failed vs a row with the corresponding row control element failed. In that case, the state space would be larger (but still not as large as obtained by a naive technique). Each state with one or more rows or columns of D has more than one version, specifying whether the corresponding row/column control points (processors or memories) or their combinations are operational or failed without distinguishing between if there are more than one processor or memories that could have failed. It is easy to see that the number of states is still small (it is exactly 84) which is much lower than 2^{15}.

4 Exploiting Multi-Dimensional Symmetry

The example of the previous section shows how a 15 component system can be represented using exactly 36 states. The fine behavior at the time of individual failures and their handling can be exactly modeled with each arc in the Markov chain as each arc represents a failure of a processor or a memory or a BIU. For example, the fault and error handling model as used in the HARP tool [16] can be included without any difficulty. In this section, we develop our methodology to generate state space, exploring permutation symmetry in a k-dimensional structure.

Various researchers have dealt with this in different ways. Arlat and Laprie [10] consider state equivalences assuming only a small number of faults in multistage interconnection networks. Such networks with or without multiple paths are also modeled in [11, 14, 3] in specific manner but symmetry properties are not fully explored. Their techniques are not easy to generalize. Das and Bhuyan have developed combinatorial techniques to model multiprocessor systems, but they do not address the explicit state space generation issue. Chiola et al. [4] use a technique similar to the one proposed here exploiting permutational symmetry to model behavior of different queues in a multiprocessor system using petri nets. Their modeling is equivalent to a single dimension symmetry in our case. However, we are considering multi-dimensional systems in which one component affect the structure (and therefore behavior) in multiple dimensions.

Aupperle and Meyer [2] developed a method to exploit group symmetric properties to reduce the state-space size. Again, our goal is the same as the one in [2] but our approach is different. Both approaches assume that the nodes in the system belong to a symmetric group. They use a group theory method. Using a set of generators for the given system's symmetry group, they generated a branching using the algorithm and state representation from [13]. The time complexity is roughly of an order $O(n^5)$ algorithm and needs $O(n^3)$ space (n = the number of states). The generators need to be identified. The branching is used to compute the order of the symmetry group. Then for each value of number of faults present, a state space representative of those many faults is searched. This step in general is exponential but in specific cases is shown to be $O(n^8)$. This state space is then used to generate necessary transitions to complete the model of the system. That means the set of next states for each state is determined and appropriate transitions rates are assigned. This step also involve searches over the generated state space.

In [2], all the states are generated without regard to how they are going to be used. Additional work is performed to add transitions to the state space. In contrast the above scheme, we generate states one at a time and consider only those states which belong to the minimized state space. Our initial state contains no fault. The next set of states contain exactly one fault and this set is reduced to equivalent classes. Each of these states is then expanded by introducing one more fault as demonstrated in Figure 6. Thus the expansion is slow and can be truncated any time. In our procedure, transition rates are computed at the same time and thus no more search is required afterwards.

4.1 State Space and Markov Chain Generation

We assume that the nodes in the system are connected in a k-dimensional structure. Each vertex represents a node in the system. We also assume that by permuting the indices in

any particular dimension and renumbering the nodes and links, the structural behavior of the system is not affected. The system performs in the same fashion when it is in any one of the two states if representation of one state could be derived by permuting indices in the respective dimensions of the representation of the other state. All such states can be lumped together and represented by a single state in the Markov chain. This state is defined as the basic class state for that set of states. For a given system state, by using the permutation symmetry, we can find the corresponding basic class state, which is equivalent to the given state. Our goal is to keep and track only the basic class states and eliminate all the other states in the state space.

To define an equivalent class of states and the corresponding basic class states, we use the following definitions. We denote the dimensions as $d1$, $d2$, $\cdots$, dk. The indices in dimension di vary from 1 to n_{di}. Each node can be in an operational state (0) or a failed state (1). Thus the state of a node can be represented by a single bit. It is possible to generalize our technique to multiple-state components. The sorting and searching are on different keys like in [4]. This is more complicated but still manageable. However, the rewards will be worth the effort. We use the following state representation.

Definition: A state is represented by a state tuple consisting of a $n_{d1} \times n_{d2} \times \cdots \times n_{dk}$ matrix where each element represents the state of a node.

Definition: A permutation $(p_1 p_2, \cdots p_N)$, where each p_i is unique and $1 \leq p_i \leq N$, of set $(1\ 2\ \cdots\ N)$ implies that 1 goes to position p_1, 2 goes to position p_2 and so on.

Definition: The "corners" of a k-dimensional space are the extreme points or vertices of the space.

Definition: A basic class state SBC_i corresponding to a given state s_i is obtained by permuting the indices in each dimension to obtain a total order on them based on a predefined criterion.

Thus, state SBC_i is isomorphic to state s_i under permutation symmetry in each dimension. The criterion we use to obtain the total order on indices is that all ones are moved towards a *corner of choice*. Without loss of generality, we arbitrarily choose the corner of choice as point $(1, 1, \cdots 1)$. The nearness to a corner may be defined using various criterion. The criteria we use is based on counting ones (number of failures) in hyperplanes defined by indices of a dimension and sorting them in descending order. In case of an equal number of ones, we use a set of rules to break the tie and if none succeeds then we break the tie arbitrarily. The rules are defined in the next subsection.

Let the permutation of indices in dimension di be denoted by P^{di}. Our goal is to find a set of permutations P^{d1}, P^{d2}, $\cdots$, P^{dk}, such that $P^{dk} \cdots P^{d2} P^{d1}(s_i) = SBC_i$. The permutations are applied on s_i one at a time. To find the permutation P^{d1}, P^{d2}, $\cdots$, P^{dk} we proceed in a systematic manner.

Note that we are assuming that we have freedom in permuting indices in each dimension and not the dimensions themselves. However, in some structures like the binary cube, even dimensions themselves can also be permuted. In some other systems, only rotational and not permutation symmetry within a dimension may be permissible. Thus the number of permutations allowed are restricted to those obtainable by a rotation only. In such cases, the algorithm needs to be, and can be, modified suitably. It should be noted that in such cases, the reduction in the state space size may remain small.

In the approach below, we start with one state at a time, generate all the next possible states to that state and reduce the generated states to the basic classes using the permutation

symmetry. We continue the procedure until no more states are added in the generated set. This procedure is similar to the one used in [1, 15]. The state space generation process proceeds in a systematic manner such that the newly added states have one more fault than its predecessor state. This allows the truncation of the state space by terminating addition of states once the state representation satisfies certain specific criteria such as the number of faults exceeding a certain given value. Also, note that the state space is symmetrical for f and $n - f$ faults where n is the total number of components. The state space generation process is outlined below. In the following, t_i and r_i represent a failure and repair rate or a forward and reverse transition rate in general. Moreover, the set T_s contain all possible transitions, each caused by failure of a single node only.

Procedure GenerateSS&MC

begin

1. $S \leftarrow s_0$; s_0 is initial state
2. *While* S contains an unmarked state do {
   (a) Select the next unmarked state s and mark it
   (b) Find Transition Set $T_s = \{(s_i, t_i, r_i)\}$;s_i is a next state of s with transition rate t_i and r_i is the transition rate from s_i to s
   (c) For each transition $tr_i = (s_i, t_i) \in T_s$,
      i. Reduce state s_i to basic class SBC_i
      ii. If $SBC_i \notin S$ then $S = S \cup \{SBC_i\}$
      iii. Add transition from s to SBC_i with rate t_i
      iv. Add [1] the transition from SBC_i to s with rate r_i
      /* If a transition already exists then the rates are added together */

   }*end while*

end Procedure GenerateSS&MC

In the process above, a state s considered in Step 2a is a basic class state and injection of a fault affects the structure of the state in a specific manner and at a specific location. This information can be used to speed up the reduction process in step 2c(i). However, the algorithm in Section 4.2 does not assume that the given state has any structure. Thus it is a general solution for a k-dimensional structure. Also, in Steps 2c(iii) and 2c(iv), there may be more than one transition added from a state s to SBC_i or vice versa. All these transitions are lumped together to one transition with the transition rate equal to the sum of all the rates.

[1] Care should be taken when implied failures are present but state of all system components are not explicitly included in the state representation. For example, in Meshkin suppose a BIU fails and then the corresponding column processor or row memory fails. The second failure will dominate in the state space representation as all BIUs in a column or row are marked failed. In such situations state space representation must include all repairable components. Note that we can still take advantage of the permutation symmetry. For example, in Meshkin it will be a 4×4 system but symmetry is defined for three rows and three columns only.

Number of States. The actual number of states we need to consider is about $|S|m$ where $|S|$ is the number of states in the final set and m is the average number of nodes whose failure is considered in each state. Moreover, since s_i has one more fault than s, the search for $SBC_i \in S$ can be restricted in the relevant range of S.

We start with an initial state s_0 where all the components are operational. This state is also a SBC. State s_0 is considered for failure of individual components, (as shown in Figure 6). All these transitions belong to set T_s. Thus, T_s for $s = s_0$ consists of N states where N is the total number of components in the systems. These will be reduced to SBC with one failure. We denote the number of SBC states with i failures by K_i. Obviously, $K_0 = 1$. For the Meshkin example, T_s for $s = s_0$ is 15 as there are 15 components that can fail. After reducing these states, we are left with K_1 states with one component failure. Note here that due to implied failure, a single component failure may be viewed as multiple node failures in the representation. In the next step, these states are considered for further failures and so on. The total number of states is $\sum_{i=0}^{i=N} K_i$. However, the total number of states that need to be reduced to SBC is considerably larger and is given by

$$\sum_{i=0}^{i=N} K_i * (N - i).$$

This is because each state with i failed components will have $N - i$ operational components whose failures need to be considered. This gives rise to $N - i$ next states.

Pitfall. Our heuristic algorithm does find an appropriate set of permutations in most cases. However, it is possible that states s_1 and s_2 are equivalent to the same basic class state, but our algorithm does not detect that in some very specific situations. We will demonstrate such specific situations in one of the example later on. However, the point to note here is that even if we do not succeed in all cases, we still achieve a significant reduction in the size of the state space. Moreover, the two representations of the same SBC state, when considered to include successive states, may produce the same next states (SBCs). Also notice that it has no implication on the accuracy of the solution and the two states will have state occupation probabilities such that their sum will correspond to the single state occupation probability. Since rewards are identical for the two representations, the total reward will also be computed correctly.

4.2 Algorithm for Reduction to a Base Class

Algorithm **GBCP(** $s(d1, d2, \cdots, dk), k$**)** reduces a k-dimensional state into a basic class state representation. We first define a subcube of a cube as follows.

Definition: A subcube of a cube is a sub-structure where a set of indices in each dimension vary only in a sub-interval.

For example, a subcube denoted by $SC(d1 = *, \cdots, di = (\alpha..\beta), \cdots, dk = *)$ is a subcube where dj, $j \neq i$ varies between 1 and n_{dj} and di varies between α and β. This is a smaller structure derived from the given structure. The number of ones in a subcube SC is denoted by $N(SC)$. In the notation, if $di = *$ we drop it in writing. Also, if $\alpha = \beta$, then we only write it once. Thus, $SC(di = \alpha_{ij})$ represents a subcube where $di = \alpha_{ij}$ and the indices in other dimensions can assume any value in their full range.

ALGORITHM: GET BASIC CLASS PERMUTATION GBCP$(s(d1, d2, \cdots, dk), k)$

begin

1. Obtain Partial order and partition sets PT_{di} in each dimension using the following steps.
   (a) Obtain count $N(SC(di = \alpha_{ij}))$ for $i = 1$ to k and $\alpha_{ij} = 1$ to n_{di}.
   (b) Permute indices and obtain partial order in dimension di by sorting $N(SC(di = \alpha_{ij}))$ for $j = 1$ to n_{di}.
   (c) Using sorting, partition in each dimension di to obtain PT_{di}.
2. If $k = 1$ then go to Step 5.
3. Let $lmax = k + k\%2$. $lmax$ is an even number $\geq k$.
4. For $l = 1, 3, \cdots, lmax$ Do
   If $l < k$
   Then call **Partition-two**$(m(n_{dl}, n_{d(l+1)}),\ PT_{dl},\ PT_{d(l+1)})$
   Else call **Partition-two**$(m(n_{d(l-1}, n_{dl}),\ PT_{d(l-1)},\ PT_{dl})$.
5. Return $PT_{d1}, \cdots, PT_{dk}$.

end

Figure 7: Algorithm to obtain basic class permutation.

The algorithm consists of two steps. In the first step, we obtain a partial order on the indices in each dimension separately. The algorithm first counts ones in subcubes defined by $di = \alpha_{ij}$ for $i = 1, \cdots, k$ and $\alpha_{ij} = 1, \cdots, n_{di}$. Then it permutes (renumbers) indices in each dimension di such that $N(di = \alpha_{ij}) \geq N(di = \alpha_{ik})$ for $\alpha_{ij} \leq \alpha_{ik}$ by sorting the counts for indices in each dimension di in descending order. This yields a partial order on the indices in each dimension.

In the next step, we obtain a total order in each dimension to obtain a basic class state representation for the given state. For $k = 1$, no further basis exists to obtain a complete order. Therefore, the algorithm terminates. For $k > 1$, the algorithm continues to work choosing two dimensions at a time. The dimensions are chosen in a specific order (the order itself is chosen arbitrarily). Without loss of generality, we choose $d1$ as the first dimension, $d2$ as the next, and so on. Thus, dimension dk is considered last.

The algorithm divides the indices in each dimension di for $i = 1, \cdots, k$ to form subcubes such that subcubes for all indices in that dimension in a partition have the same number of ones. The sizes of partitions will obviously vary. The partition in dimension di is denoted by PT_{di} and is $\{pdi_1, pdi_2, \cdots, pdi_{n_i}\}$ such that for all values of di in subset pdi_j, $N(SC(di = \alpha_{ij} | \alpha_{ij} \in pdi_j))$ are the same. $|PT_{di}|$ denotes the number of elements in PT_{di}. For example, after sorting in dimension di, if $N(di = \alpha_{ij}) = 5,\ 5,\ 4,\ 3,\ 3,\ 3$, respectively, for $\alpha_{ij} = 1$ to 6, then the indices in dimension di are partitioned to obtain $PT_{di} = \{(1,2), (3), (4,5,6)\}$ with $|PT_{di}| = 3$.

The algorithm in Figure 7 describes the pseudo code for the steps to get a basic class permutation. It calls Algorithm **Partition-two**, described in Figure 8, several times to partition two dimensions at a time.

Algorithm: Partition-two $(m(n_{da}, n_{db}),\ PT_{da},\ PT_{db})$

begin

```
1. IF (|PT_da| = 1) AND (|PT_db| = 1) AND ((n_da > 1) OR (n_db > 1))
     THEN Select (one of) the largest element m_cd of m(n_da, n_db) and
     Use row c and column d to obtain partitions PT_da = {(c)···} and PT_db =
   {(d)···}).
2. i = 1; j = 1.  /* Select the first element of each set */
3. WHILE (|PT_da| < n_da OR |PT_db| < n_db) DO {
    Select i^th element of PT_da, pda_i, and j^th element of PT_db, pdb_i
    IF (|pda_i| = 1) /* Single element forces a partition */
     THEN {
       k = j;
       WHILE (k ≯ |PT_db|) DO {
         IF (|pdb_k| > 1)
           THEN {
             carray(|pdb_k|) = m(i, |pdb_k|); /* Count in subcube */
             PT_c = GBCP(carray(|pdb_k|), 1); /* Partition the subset */
             Replace pdb_k by PT_c in PT_db;
             k = k + |PT_c| − 1; }    /* end THEN */
           ELSE k = k + 1; }    /* end WHILE */
       i = i + 1; }    /* end THEN */
    ELSE IF (|pdb_j| = 1) THEN /* Single element forces a partition */
       k = i;
       WHILE (k ≯ |PT_da|) DO {
         IF (|pda_k| > 1)
           THEN {
             carray(|pda_k|) = m(|pda_k|, j); /* Count in subcube */
             PT_c = GBCP(carray(|pda_k|), 1); /* Partition the subset */
             Replace pda_k by PT_c in PT_da;
             k = k + |PT_c| − 1; }    /* end THEN */
           ELSE k = k + 1; }    /* end WHILE */
       j = j + 1; }    /* end THEN */
     ELSE {
       carray(|pda_i|, |pdb_j|) = m(|pda_i|, |pdb_j|); /* Else partition two-dimension
   subcube */
       PT_c, PT_d = GBCP (carray(|pda_i|, |pdb_j|), 2);
       Replace pda_i by PT_c in PT_da and pdb_j by PT_d in PT_db; }
   }
```

end

Figure 8: Algorithm to find total order in two dimensions.

The goal of the Algorithm **Partition-two** is to obtain a total order in the chosen two dimensions da and db by permuting and partitioning each element in partition set PT_{da} and PT_{db} so that each element consists of only one index value. At the end of this step, $|PT_{da}| = n_{da}$ and $|PT_{db}| = n_{db}$. We start with dimensions $d1$ and $d2$. For this step, we define a two dimensional matrix $m(n_{da}, n_{db})$ where each element (α, β) is the count $N(SC(da = \alpha, db = \beta))$. The algorithm **Partition-two** takes the matrix $m(n_{da}, n_{db})$ and partition sets PT_{da} and PT_{db} as inputs.

Algorithm **Partition-two** finds sub-partitions in dimensions da and db by considering one element from each of the partitioned sets of the two dimensions starting with the first element from each set. It uses them to induce further order in the dimensions until no further progress can be made. Then the algorithm moves to next element in the appropriate set. Each time an element pda_{α} is further partitioned in $pda_{\alpha 1}$, $pda_{\alpha 2}$, $\cdots$, $pda_{\alpha c}$, it is replaced by the sub-partitions while maintaining the sequence in PT_{da}, the elements of the set PT_{da} are renumbered and the count $|PT_{da}|$ is updated appropriately by $c - 1$. The algorithm returns PT_{da} and PT_{db} with total order.

If the whole set in both dimension consists of only one element whose values are greater than one, then the algorithm forces a partition in each dimension by selecting the largest element in matrix m and using the indices of that element as the first index for partition in each dimension. In case of a tie, it arbitrarily picks up one of the elements. This is the first step in the description.

If any of the first elements in both dimensions have only one index, that index is used to induce a further partition in the other dimension. Algorithm **GBCP()** is used to create the partition in that dimension. If any partition is induced, that is included in partition sets PT_{da} or PT_{db}, respectively, and the algorithm is repeated. If the first elements in both sets consist of more than one index, then again algorithm **GBCP()** is called to induce a partition in a smaller structure defined by the two sub-partitioned only. This smaller structure consists of parts of the rows and columns which have the same sums in all rows and in all columns. This 2-dimensional space is further partitioned using the two dimensional subspaces as shown in Figure 9. In the first step, dimension $d1$ and $d2$ has four and three partitions, respectively. In the next step, we choose the first partition of each partition set and recurse through the process. The algorithm terminates when both dimensions achieve the required partitions.

4.3 Examples

To demonstrate the above algorithm, we use several examples. In our first example, a state in a three dimensional structure is shown in Figure 10. Using the first step of counting in Figure 10 counts in dimension are (7, 5, 7, 5) for $d3 = 1$, 2, 3, and 4, respectively. Similarly, the counts in dimension $d2$ are (5, 8, 8, 3) and in dimension $d1$ are (4, 8, 8, 4) for respective indices. Thus example state in row of 1 Figure 10 is permuted in dimension $d3$ using permutation (1 3 2 4). Similarly the state is permuted in dimension $d2$ using permutation (3 2 1 4) and in dimension $d1$ using permutation (3 2 1 4). The resultant state is shown in row 2 of Figure 10 in dimensions $d1$ and $d2$ only with elements as count of number of ones in dimension $d3$. This creates $PT_{d1} = \{(1,2),(3,4)\}$, $PT_{d2} = \{(1,2),(3),(4)\}$ and $PT_{d3} = \{(1,2),(3,4)\}$. After that, we start with dimensions $d1$ and $d2$ and call algorithm **Partition-two**. That algorithm starts with a 2×2 substructure and permute dimensions

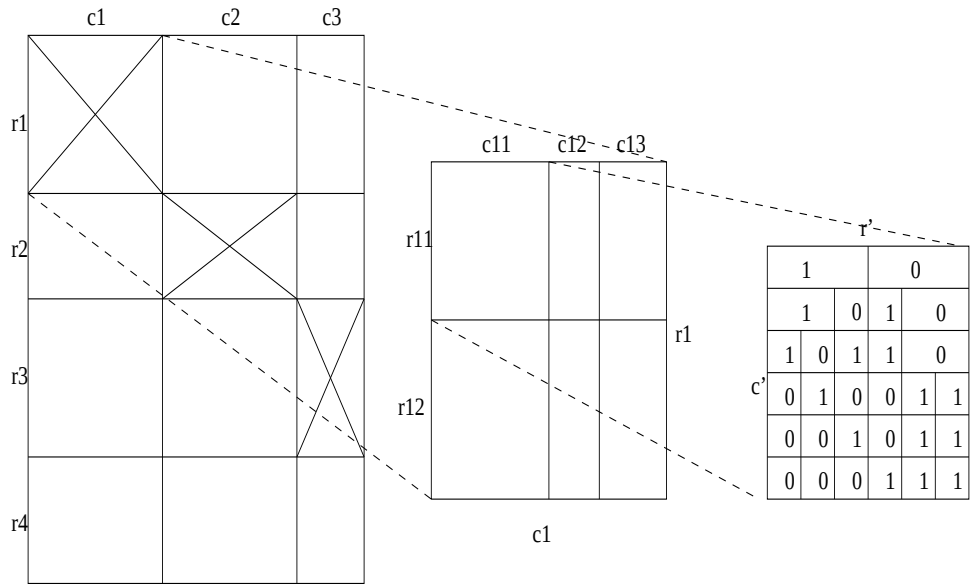

Figure 9: Partition and arrangement of two dimensional space

$d1$ and $d2$ as (2 1) and (2 1) using the algorithm **GBCP()** and also partitions the indices in each dimension. Next, it considers partitions with $d1 = (34)$ and $d2 = (1)$ and further partitions sub-partition $(3, 4)$ into (3) and (4) in dimension $d1$. At this point, the partitions in $d3$ and $d2$ is complete. Then the algorithm works with dimensions $d2$ and $d3$. $d2$ is already partitioned and dimension $d3$ consists of two partitions $(1, 2)$ and $(3, 4)$ (after renumbering). Dimension $d2$ is used to impose a partition on partitions of $d3$. The first element in PT_{d2} induces an order in sub-partitions in PT_{d3}. The resultant partitions are shown in the last row of Figure 10.

In our second example, we have counts in all subcubes in each dimension as 8 (as shown in Figure 11). In this case no obvious partitioning is possible. In dimension $d1$-$d2$, we force a partition by picking up the largest element. We have two choices and we pick one arbitrarily at location (2, 3). Then we sort the elements in row 2 and column 3. That induces a partition and we get a complete order in dimension $d1$ and $d2$. Notice that choosing an alternate value yields a state that is symmetrical with respect to the chosen one along a diagonal (that is why, sometimes, we may not reach the same state.) Using this partition, we still cannot induce any more partitions and force an arbitrary partition in dimension $d3$.

A more complex four dimension example is shown in Figure 12. In this example $n_{d1} = 2$, $n_{nd2} = 2$, $n_{nd3} = 4$, and $n_{nd4} = 4$. For each value of $d3$ and $d4$, $d1$ and $d2$ is a 2×2 matrix as shown in the figure. Working on 2 dimensions at a time, we get a partition as shown in the Figure 12 in a straightforward manner.

4.4 State Spaces Sizes for Systems with Permutation Symmetry

Now, we consider the sizes of the complete state spaces for larger systems. We choose 2-dimensional crossbar and hypercube (binary cube) systems. For a 3×3 crossbar, Table 1 lists all possible states one can generate. There are a total of 36 states which are same as

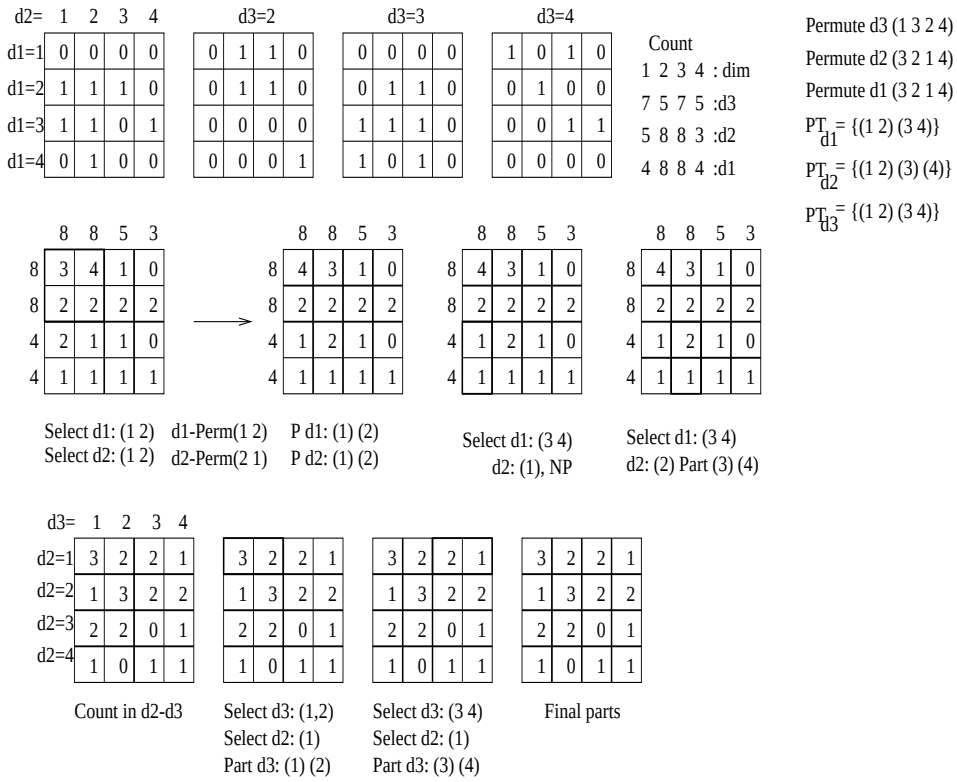

Figure 10: Example 1: A state and its transformation in a $4 \times 4 \times 4$ system

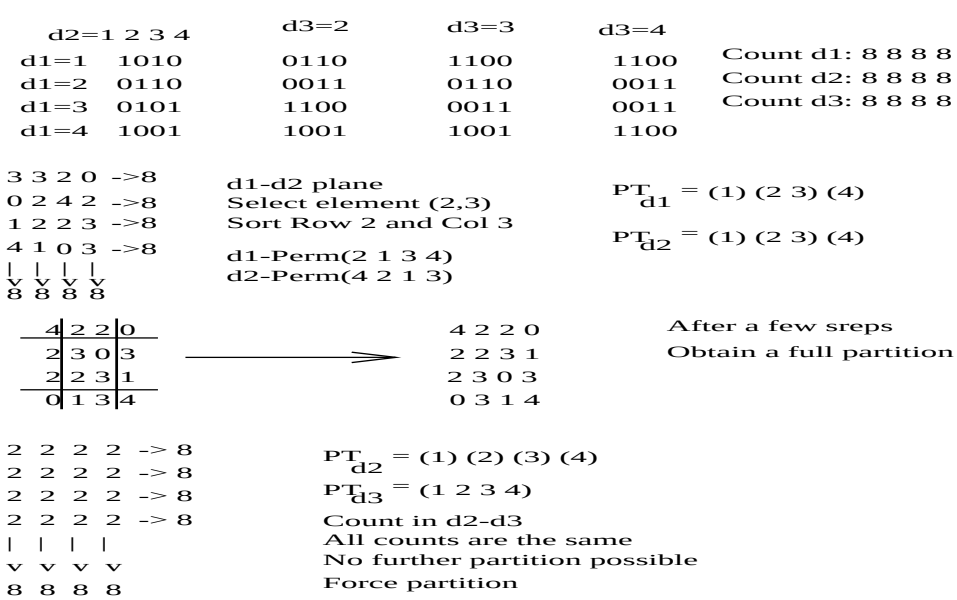

Figure 11: Example 2: A partition with equal counts in all dimension. It needs an element selection for further arrangement.

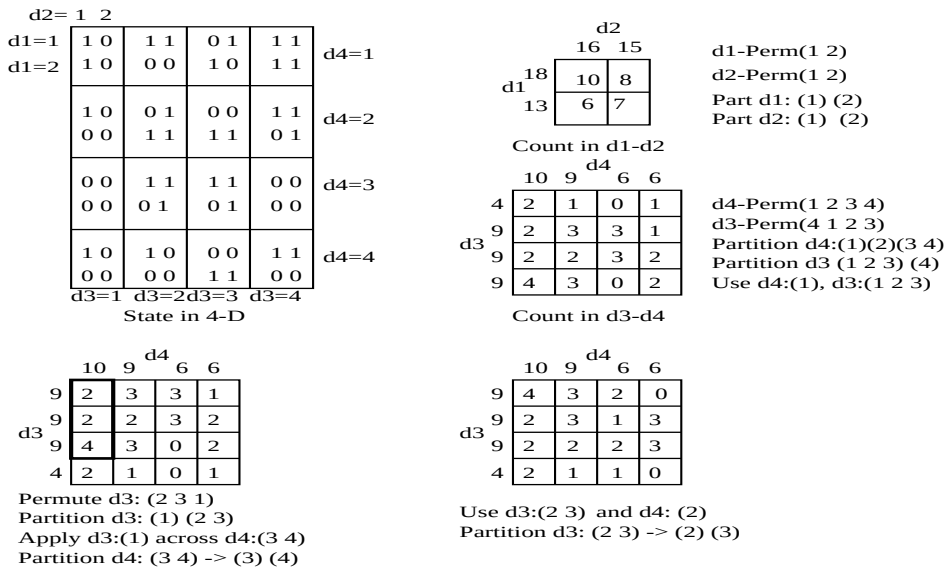

Figure 12: Example 3: A state transformation in 4-dimensions.

shown in Figure 4. For a 4×4 crossbar, we list all possible states in tables 2 to 7 for various node failures in the system. There are exactly 380 states in the system. If we compare them with actual number of possible states, this is a significant reduction. We could possibly have up to 2^{16} states. Thus we achieve a 99.5% reduction in the number of states. We also list the actual arrangements of these states in Tables 8 to 11. Notice that we are not permuting dimensions themselves but within each dimension only.

The other system we consider is 3-dimensional $2 \times 2 \times 2$ crossbar structures with permutation symmetry. It can have up to $2^8 = 256$ states that can be reduced to 46 states. The representative states are listed in Table 12. Notice that if it was a 3-binary cube then we also have symmetry in dimensions as well. Then the number of states can be further reduced to 22. For example, two faults states (02;11;02), (11;02;02), and (02;02;11) in the set of 46 states are equivalent under dimensional symmetry. Thus, they can be represented by only one state. Since there are exactly two choices in each dimension, it is relatively simpler to identify equivalent states. Majorization techniques [6] can be used to permute within dimensions after using the technique of the previous section with each dimension.

5 Conclusions

We have presented a techniqueto reduce the size of the state space associated with analysis of a large class of systems which are designed using a regular interconnected topology. We have also presented an algorithm to generate the states as well the Markov chain. In such systems, the relative position of a node is of no importance. We exploit permutation and rotation symmetries present in the system to our advantage in state space generation. Analysis of such systems, combinatorially or otherwise, may result in large number of system states. We demonstrated the reduction in largeness of the state space using several examples. We also presented an algorithm to identify equivalent states. The resulting much smaller state space allows analysis of very large systems. We believe that our technique is going to be

Table 1: Arrangements in 3×3 Crossbar with Permutation Symmetry

S.N.	No. of Failures	Row Conf	Col Conf	Combinations
01	0 or 9	(0, 0, 0)	(0, 0, 0)	1
02	1 or 8	(0, 0, 1)	(0, 0, 1)	1
03	2 or 7	(0, 0, 2)	(0, 1, 1)	1
04	2 or 7	(0, 1, 1)	(0, 0, 2)	1
05	2 or 7	(0, 1, 1)	(0, 1, 1)	1
06	3 or 6	(0, 0, 3)	(1, 1, 1)	1
07	3 or 6	(0, 1, 2)	(0, 1, 2)	1
08	3 or 6	(0, 1, 2)	(1, 1, 1)	1
09	3 or 6	(1, 1, 1)	(0, 0, 3)	1
10	3 or 6	(1, 1, 1)	(0, 1, 2)	1
11	3 or 6	(1, 1, 1)	(1, 1, 1)	1
12	4 or 5	(0, 1, 3)	(1, 1, 2)	1
13	4 or 5	(0, 2, 2)	(0, 2, 2)	1
14	4 or 5	(0, 2, 2)	(1, 1, 2)	1
15	4 or 5	(1, 1, 2)	(0, 1, 3)	1
16	4 or 5	(1, 1, 2)	(0, 2, 2)	1
17	4 or 5	(1, 1, 2)	(1, 1, 2)	2

Table 2: Arrangements in 4×4 Crossbar with Permutation Symmetry(0 to 3 failures)

S.N.	No. of Failures	Row Conf	Col Conf	Combinations
01	0 or 16	(0, 0, 0, 0)	(0, 0, 0, 0)	1
02	1 or 15	(0, 0, 0, 1)	(0, 0, 0, 1)	1
03	2 or 14	(0, 0, 0, 2)	(0, 0, 1, 1)	1
04	2 or 14	(0, 0, 1, 1)	(0, 0, 0, 2)	1
05	2 or 14	(0, 0, 1, 1)	(0, 0, 1, 1)	1
06	3 or 13	(0, 0, 0, 3)	(0, 1, 1, 1)	1
07	3 or 13	(0, 0, 1, 2)	(0, 0, 1, 2)	1
08	3 or 13	(0, 0, 1, 2)	(0, 1, 1, 1)	1
09	3 or 13	(0, 1, 1, 1)	(0, 0, 0, 3)	1
10	3 or 13	(0, 1, 1, 1)	(0, 0, 1, 2)	1
11	3 or 13	(0, 1, 1, 1)	(0, 1, 1, 1)	1

Table 3: Arrangements in 4×4 Crossbar with Permutation Symmetry(4 failures)

S.N.	No. of Failures	Row Conf	Col Conf	Combinations
01	4 or 12	(0, 0, 0, 4)	(1, 1, 1, 1)	1
02	4 or 12	(0, 0, 1, 3)	(0, 1, 1, 2)	1
03	4 or 12	(0, 0, 1, 3)	(1, 1, 1, 1)	1
04	4 or 12	(0, 0, 2, 2)	(0, 0, 2, 2)	1
05	4 or 12	(0, 0, 2, 2)	(0, 1, 1, 2)	1
06	4 or 12	(0, 0, 2, 2)	(1, 1, 1, 1)	1
07	4 or 12	(0, 1, 1, 2)	(0, 0, 1, 3)	1
08	4 or 12	(0, 1, 1, 2)	(0, 0, 2, 2)	1
09	4 or 12	(0, 1, 1, 2)	(0, 1, 1, 2)	2
11	4 or 12	(0, 1, 1, 2)	(1, 1, 1, 1)	1
12	4 or 12	(1, 1, 1, 1)	(0, 0, 0, 4)	1
13	4 or 12	(1, 1, 1, 1)	(0, 0, 1, 3)	1
14	4 or 12	(1, 1, 1, 1)	(0, 0, 2, 2)	1
15	4 or 12	(1, 1, 1, 1)	(0, 1, 1, 2)	1
16	4 or 12	(1, 1, 1, 1)	(1, 1, 1, 1)	1

Table 4: Arrangements in 4×4 Crossbar with Permutation Symmetry(5 failures)

S.N.	No. of Failures	Row Conf	Col Conf	Combinations
01	5 or 11	(0, 0, 1, 4)	(1, 1, 1, 2)	1
02	5 or 11	(0, 0, 2, 3)	(0, 1, 2, 2)	1
03	5 or 11	(0, 0, 2, 3)	(1, 1, 1, 2)	1
04	5 or 11	(0, 1, 1, 3)	(0, 1, 1, 3)	1
05	5 or 11	(0, 1, 1, 3)	(0, 1, 2, 2)	1
06	5 or 11	(0, 1, 1, 3)	(1, 1, 1, 2)	2
07	5 or 11	(0, 1, 2, 2)	(0, 0, 2, 3)	1
08	5 or 11	(0, 1, 2, 2)	(0, 1, 2, 2)	2
09	5 or 11	(0, 1, 2, 2)	(0, 1, 1, 3)	1
10	5 or 11	(0, 1, 2, 2)	(1, 1, 1, 2)	2
11	5 or 11	(1, 1, 1, 2)	(0, 0, 1, 4)	1
12	5 or 11	(1, 1, 1, 2)	(0, 0, 2, 3)	1
13	5 or 11	(1, 1, 1, 2)	(0, 1, 1, 3)	2
14	5 or 11	(1, 1, 1, 2)	(0, 1, 2, 2)	2
15	5 or 11	(1, 1, 1, 2)	(1, 1, 1, 2)	2

Table 5: Arrangements in 4×4 Crossbar with Permutation Symmetry(6 failures)

S.N.	No. of Failures	Row Conf	Col Conf	Combinations
01	6 or 10	(0, 0, 2, 4)	(1, 1, 2, 2)	1
02	6 or 10	(0, 1, 1, 4)	(1, 1, 1, 3)	1
03	6 or 10	(0, 1, 1, 4)	(1, 1, 2, 2)	1
04	6 or 10	(0, 0, 3, 3)	(0, 2, 2, 2)	1
05	6 or 10	(0, 0, 3, 3)	(1, 1, 2, 2)	1
06	6 or 10	(0, 1, 2, 3)	(0, 1, 2, 3)	1
07	6 or 10	(0, 1, 2, 3)	(0, 2, 2, 2)	1
08	6 or 10	(0, 1, 2, 3)	(1, 1, 1, 3)	1
09	6 or 10	(0, 1, 2, 3)	(1, 1, 2, 2)	3
10	6 or 10	(0, 2, 2, 2)	(0, 0, 3, 3)	1
11	6 or 10	(0, 2, 2, 2)	(0, 1, 2, 3)	1
12	6 or 10	(0, 2, 2, 2)	(0, 2, 2, 2)	1
13	6 or 10	(0, 2, 2, 2)	(1, 1, 1, 3)	1
14	6 or 10	(0, 2, 2, 2)	(1, 1, 2, 2)	2
15	6 or 10	(1, 1, 1, 3)	(0, 1, 1, 4)	1
16	6 or 10	(1, 1, 1, 3)	(0, 1, 2, 3)	1
17	6 or 10	(1, 1, 1, 3)	(0, 2, 2, 2)	1
18	6 or 10	(1, 1, 1, 3)	(1, 1, 1, 3)	2
19	6 or 10	(1, 1, 1, 3)	(1, 1, 2, 2)	2
20	6 or 10	(1, 1, 2, 2)	(0, 0, 2, 4)	1
21	6 or 10	(1, 1, 2, 2)	(0, 1, 1, 4)	1
22	6 or 10	(1, 1, 2, 2)	(0, 0, 3, 3)	1
23	6 or 10	(1, 1, 2, 2)	(0, 1, 2, 3)	3
24	6 or 10	(1, 1, 2, 2)	(0, 2, 2, 2)	2
25	6 or 10	(1, 1, 2, 2)	(1, 1, 1, 3)	2
26	6 or 10	(1, 1, 2, 2)	(1, 1, 2, 2)	5

Table 6: Arrangements in 4×4 Crossbar with Permutation Symmetry(7 failures)

S.N.	No. of Failures	Row Conf	Col Conf	Arrangments
01	7 or 09	(0, 0, 3, 4)	(1, 2, 2, 2)	1
02	7 or 09	(0, 1, 2, 4)	(1, 1, 2, 3)	1
03	7 or 09	(0, 1, 2, 4)	(1, 2, 2, 2)	1
04	7 or 09	(0, 1, 1, 4)	(1, 1, 1, 4)	1
05	7 or 09	(0, 1, 1, 4)	(1, 1, 2, 3)	1
06	7 or 09	(0, 1, 1, 4)	(0, 2, 2, 2)	1
07	7 or 09	(0, 1, 3, 3)	(0, 2, 2, 3)	1
08	7 or 09	(0, 1, 3, 3)	(1, 1, 2, 3)	1
09	7 or 09	(0, 1, 3, 3)	(1, 2, 2, 2)	2
10	7 or 09	(0, 2, 2, 3)	(0, 1, 3, 3)	1
11	7 or 09	(0, 2, 2, 3)	(0, 2, 2, 3)	1
12	7 or 09	(0, 2, 2, 3)	(1, 1, 2, 3)	2
13	7 or 09	(0, 2, 2, 3)	(1, 2, 2, 2)	2
14	7 or 09	(1, 1, 2, 3)	(0, 1, 2, 4)	1
15	7 or 09	(1, 1, 2, 3)	(1, 1, 1, 4)	1
16	7 or 09	(1, 1, 2, 3)	(0, 1, 3, 3)	1
17	7 or 09	(1, 1, 2, 3)	(0, 2, 2, 3)	2
18	7 or 09	(1, 1, 2, 3)	(1, 1, 2, 3)	5
19	7 or 09	(1, 1, 2, 3)	(1, 2, 2, 2)	5
20	7 or 09	(1, 2, 2, 2)	(0, 0, 3, 4)	1
21	7 or 09	(1, 2, 2, 2)	(0, 1, 2, 4)	1
22	7 or 09	(1, 2, 2, 2)	(1, 1, 1, 4)	1
23	7 or 09	(1, 2, 2, 2)	(0, 1, 3, 3)	2
24	7 or 09	(1, 2, 2, 2)	(0, 2, 2, 3)	2
25	7 or 09	(1, 2, 2, 2)	(1, 1, 2, 3)	5
26	7 or 09	(1, 2, 2, 2)	(1, 2, 2, 2)	3

Table 7: Arrangements in 4×4 Crossbar with Permutation Symmetry(8 failures)

S.N.	No. of Failures	Row Conf	Col Conf	Arrangments
01	8	(0, 0, 4, 4)	(2, 2, 2, 2)	1
02	8	(0, 1, 3, 4)	(1, 2, 2, 3)	1
03	8	(0, 1, 3, 4)	(2, 2, 2, 2)	1
04	8	(0, 2, 2, 4)	(1, 1, 3, 3)	1
05	8	(0, 2, 2, 4)	(1, 2, 2, 3)	1
06	8	(0, 2, 2, 4)	(2, 2, 2, 2)	1
07	8	(1, 1, 2, 4)	(1, 1, 2, 4)	1
08	8	(1, 1, 2, 4)	(1, 1, 3, 3)	1
09	8	(1, 1, 2, 4)	(1, 2, 2, 3)	2
10	8	(1, 1, 2, 4)	(2, 2, 2, 2)	1
11	8	(0, 2, 2, 3)	(0, 2, 2, 3)	1
12	8	(0, 2, 2, 3)	(1, 1, 3, 3)	1
13	8	(0, 2, 2, 3)	(1, 2, 2, 3)	2
14	8	(0, 2, 2, 3)	(2, 2, 2, 2)	1
15	8	(1, 1, 3, 3)	(0, 2, 2, 4)	1
16	8	(1, 1, 3, 3)	(1, 1, 2, 4)	1
17	8	(1, 1, 3, 3)	(0, 2, 3, 3)	1
18	8	(1, 1, 3, 3)	(1, 1, 3, 3)	1
19	8	(1, 1, 3, 3)	(1, 2, 2, 3)	3
20	8	(1, 1, 3, 3)	(2, 2, 2, 2)	2
21	8	(1, 2, 2, 3)	(0, 1, 3, 4)	1
22	8	(1, 2, 2, 3)	(0, 2, 2, 4)	1
23	8	(1, 2, 2, 3)	(1, 1, 2, 4)	2
24	8	(1, 2, 2, 3)	(0, 2, 3, 3)	3
25	8	(1, 2, 2, 3)	(1, 1, 3, 3)	3
26	8	(1, 2, 2, 3)	(1, 2, 2, 3)	8
27	8	(1, 2, 2, 3)	(2, 2, 2, 2)	2
28	8	(2, 2, 2, 2)	(0, 0, 4, 4)	1
29	8	(2, 2, 2, 2)	(0, 1, 3, 4)	1
30	8	(2, 2, 2, 2)	(0, 2, 2, 4)	1
31	8	(2, 2, 2, 2)	(1, 1, 2, 4)	1
32	8	(2, 2, 2, 2)	(0, 2, 3, 3)	1
33	8	(2, 2, 2, 2)	(1, 1, 3, 3)	2
34	8	(2, 2, 2, 2)	(1, 2, 2, 3)	2
35	8	(2, 2, 2, 2)	(2, 2, 2, 2)	3

Table 8: Actual Arrangements in 4×4 Crossbar with Permutation Symmetry(8 failures)

Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col
0044		0134	0134		0224	0224	0224
2222		1223	2222		1133	1223	2222
1111		1111	1111		1111	1111	1111
1111		0111	0111		0011	0011	0011
0000		0001	1000		0011	0101	1100
0000		0000	0000		0000	0000	0000
1124	1124	1124	1124	1124			
1124	1133	1223	1223	2222			
1111	1111	1111	1111	1111			
0011	0011	0011	0110	0011			
0001	0001	0001	0001	0100			
0001	0010	0100	0001	1000			
0233	0233	0233	0233	0233			1133
0233	1133	1223	1223	2222			0224
0111	0111	0111	0111	0111			0111
0111	1011	0111	1011	1011			0111
0011	0011	1001	0101	1100			0001
0000	0000	0000	0000	0000			0001
1133	1133	1133	1133	1133	1133	1133	1133
1124	0233	1133	1223	1223	1223	2222	2222
0111	0111	0111	0111	0111	0111	0111	0111
1011	0111	1011	0111	1011	1110	0111	1011
0001	0001	0001	0001	0001	0001	1000	0100
0001	0010	0010	1000	0100	0001	1000	1000
1223	1223	1223	1223	1223	1223	1223	1223
0134	0224	1124	1124	0233	0233	0233	1133
0111	0111	0111	1101	0111	0111	0111	0111
0011	0011	0011	0011	0011	0011	0101	0011
0011	0101	1001	0011	0011	0011	0011	0011
0001	0001	0001	0001	1000	0100	0010	1000
1223	1223	1223	1223	1223	1223	1223	1223
1133	1133	1223	1223	1223	1223	1223	1223
0111	1101	0111	0111	1011	0111	0111	1011
0011	0011	0011	0011	0011	0011	1001	0101
1001	0011	0101	1001	0101	1100	0110	0101
0010	0010	1000	0100	0100	0001	0001	0010
1223	1223	1223	1223	2222	2222	2222	2222
1223	1223	2222	2222	0044	0134	0224	1124
1011	1110	0111	0111	0011	0011	0011	0011
0101	0011	1001	1001	0011	0011	0011	0011
0110	0101	0110	1010	0011	0011	0101	0101
0001	0001	1000	0100	0011	0101	0101	1001
2222	2222	2222	2222	2222	2222	2222	2222
0233	1133	1133	1223	1223	2222	2222	2222
0011	0011	0011	0011	0011	0011	0011	0011
0011	0011	0011	0011	0101	0011	1001	0101
0101	0011	0101	0101	1001	1100	0110	1010
0110	1100	1010	1100	0110	1100	1100	1100

Table 9: Actual Arrangements in 4×4 Crossbar with Permutation Symmetry(7 failures)

Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col
0034		0124	0124		1114	1114	1114
1222		1123	1222		1114	1123	1222
1111		1111	1111		1111	1111	1111
0111		0011	0011		0001	0001	0001
0000		0001	0100		0001	0001	0010
0000		0000	0000		0001	0010	0100
0133	0133	0133	0133				
0223	1123	1222	1222				
0111	0111	0111	0111				
0111	1011	0111	1011				
0001	0001	1000	0100				
0000	0000	0000	0000				
0223	0223	0223	0223	0223	0223		
0133	0223	1123	1123	1222	1222		
0111	0111	0111	0111	0111	0111		
0011	0011	0011	1001	0011	1001		
0011	0101	1001	1001	1100	1010		
0000	0000	0000	0000	0000	0000		
1123	1123	1123	1123	1123	1123	1123	1123
0124	1114	0133	0223	0223	1123	1123	1123
0111	0111	0111	0111	0111	0111	0111	1101
0011	1001	0011	0011	0110	0011	1001	0011
0001	0001	0001	0001	0001	0001	0001	0001
0001	0001	0010	0100	0001	1000	0010	0010
1123	1123	1123	1123	1123	1123	1123	
1123	1123	1222	1222	1222	1222	1222	
0111	1110	0111	0111	0111	0111	0111	
1010	0011	0011	0011	1001	1001	1001	
0001	0001	0100	1000	0010	0010	1000	
0001	0001	1000	1000	0100	1000	0010	
1222	1222	1222	1222	1222	1222	1222	
0034	0124	1114	0133	0133	0223	0223	
0011	0011	0011	0011	0011	0011	0011	
0011	0011	0101	0011	0011	0011	0101	
0011	0101	1001	0011	0101	0101	0110	
0001	0001	0001	0100	0010	0010	0001	
1222	1222	1222	1222	1222	1222	1222	1222
1123	1123	1123	1123	1123	1222	1222	1222
0011	0011	0011	0011	0011	0011	0011	0011
0011	0011	0101	0101	0101	0011	0101	0101
0101	1100	1001	1001	1100	1100	0110	1100
1000	0001	0100	1000	0001	0100	1000	0010

Table 10: Actual Arrangements in 4×4 Crossbar with Permutation Symmetry(6 failures)

Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col
0024		0114	0114		0033	0033	
1122		1113	1122		0222	1122	
1111		1111	1111		0111	0111	
0011		0001	0001		0111	1011	
0000		0001	0010		0000	0000	
0000		0000	0000		0000	0000	
0123	0123	0123	0123	0123	0123		
0123	0222	1113	1122	1122	1122		
0111	0111	0111	0111	0111	1101		
0011	0011	1001	0011	1001	0011		
0001	0100	0001	1000	0010	0010		
0000	0000	0000	0000	0000	0000		
0222	0222	0222	0222	0222	0222		
0033	0123	0222	1113	1122	1122		
0011	0011	0011	0011	0011	0011		
0011	0011	0101	0101	0011	0101		
0011	0101	0110	1001	1100	1010		
0000	0000	0000	0000	0000	0000		
1113	1113	1113	1113	1113	1113	1113	
0114	0123	0222	1113	1113	1122	1122	
0111	0111	0111	0111	1110	0111	1101	
0001	0001	0001	0001	0001	0001	0001	
0001	0001	0010	0001	0001	0010	0010	
0001	0010	0100	1000	0001	1000	0010	
1122	1122	1122	1122	1122	1122	1122	1122
0024	0114	0033	0123	0123	0123	0222	0222
0011	0011	0011	0011	0011	0011	0011	0011
0011	0101	0011	0011	0101	0110	0011	0101
0001	0001	0001	0001	0001	0001	0100	0010
0001	0001	0010	0100	0010	0001	0100	0100
1122	1122	1122	1122	1122	1122	1122	
1113	1113	1122	1122	1122	1122	1122	
0011	0011	0011	0101	0011	0011	0101	
0101	1100	0011	1010	1100	0101	1001	
0001	0001	0100	0001	0001	0100	0010	
1000	0001	1000	0010	0010	1000	0010	

Table 11: Actual Arrangements in 4×4 Crossbar with Permutation Symmetry(< 6 failures)

Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col	Row/Col
0014	0023	0023		0113	0113	0113	0113
1112	0122	1112		0113	0122	1112	1112
1111	0111	0111		0111	0111	0111	1110
0001	0011	1001		0001	0001	0001	0001
0000	0000	0000		0001	0010	1000	0001
0000	0000	0000		0000	0000	0000	0000
0122	0122	0122	0122	0122	0122		
0023	0122	0122	0113	1112	1112		
0011	0011	0011	0011	0011	0011		
0011	0011	0101	0101	0101	1100		
0001	0100	0010	0001	1000	0001		
0000	0000	0000	0000	0000	0000		
1112	1112	1112	1112	1112	1112	1112	1112
0014	0023	0113	0113	0122	0122	1112	1112
0011	0011	0011	0110	0011	0101	0011	0110
0001	0001	0001	0001	0001	0001	0001	0001
0001	0001	0001	0001	0010	0010	0100	0001
0001	0010	0100	0001	0100	0010	1000	1000
0004		0013	0013		0022	0022	0022
1111		0112	1111		0022	0112	1111
1111		0111	0111		0011	0011	0011
0000		0001	1000		0011	0101	1100
0000		0000	0000		0000	0000	0000
0000		0000	0000		0000	0000	0000
0112	0112	0112	0112	0112	1111	1111	1111
0013	0022	0112	0112	1111	0004	0013	0022
0011	0011	0011	0110	0011	0001	0001	0001
0001	0001	0001	0001	0100	0001	0001	0001
0001	0010	0100	0001	1000	0001	0001	0010
0000	0000	0000	0000	0000	0001	0010	0010
1111	1111	0003	0012	0012	0111	0111	0111
0112	1111	0111	0012	0111	0003	0012	0111
0001	0001	0111	0011	0011	0001	0001	0001
0101	0010	0000	0001	0100	0001	0001	0010
0010	0100	0000	0000	0000	0001	0010	0100
0100	1000	0000	0000	0000	0000	0000	0000
0002	0011	0011	0001	0000			
0011	0002	0011	0001	0000			
0011	0001	0001	0001	0000			
0000	0001	0010	0000	0000			
0000	0000	0000	0000	0000			
0000	0000	0000	0000	0000			

Table 12: Actual Arrangements in $2 \times 2 \times 2$ Crossbar with Permutation Symmetry

No.	pn1/pn2	pn1/pn2	pn1/pn2	pn1/pn2	pn1/pn2	pn1/pn2	pn1/pn2
	00;00;00		01;01;01		34;34;34		44;44;44
0/1	00 00		00 00		01 11		11 11
7/8	00 00		00 01		11 11		11 11
	02;11;02	11;02;02	11;11;02	02;02;11	11;02;11	02;11;11	11;11;11
2	00 00	00 01	00 01	00 00	01 00	00 00	10 00
	00 11	00 01	00 10	01 01	00 01	10 01	00 01
	12;13;12	12;12;12	03;12;12	12;12;12	12;12;12	12;12;12	12;12;03
3	00 01	00 01	00 00	01 00	00 01	00 10	00 01
	01 01	10 01	01 11	00 11	01 10	01 01	00 11
	22;22;22	13;13;22	13;22;22	04;22;22	22;13;13	22;22;13	13;13;13
4	11 00	01 00	10 00	00 00	01 01	10 01	00 01
	00 11	01 11	01 11	11 11	00 11	00 11	01 11
	13;22;13	22;22;22	22;13;22	22;04;22	22;13;22	22;22;22	22;22;22
4	00 10	00 11	01 01	01 01	10 01	01 10	10 10
	01 11	00 11	10 01	01 01	01 01	10 01	01 01
	23;23;23	23;14;23	23;23;23	23;23;23	23;23;23	14;23;23	23;23;14
5	00 11	01 01	01 01	01 10	10 01	00 01	00 11
	11 01	01 11	10 11	01 11	01 11	11 11	01 11
	24;33;24	33;24;24	33;33;24	24;24;33	24;33;33	33;24;33	33;33;33
6	00 11	01 11	10 11	01 01	10 01	11 01	11 01
	11 11	01 11	01 11	11 11	11 11	01 11	10 11

very useful in analyzing large systems exactly.

References

[1] A. Valmari, "Compositional State Space Generation," Advances in Petri Nets 1993. Gjern, Denmark, June 1991, pp. 427-57.

[2] B. E. Aupperle and J. F. Meyer, "State space generation for degradable multiprocessor Systems," in Proc. FTCS-1990, Montreal, June 1991, pp. 308-15.

[3] W. H. Sanders and J. F. Meyer, "Reduced Base'Model Construction Methods for Stochastic Activity Networks," IEEE Trans. on Selected Areas in Comm., Vol. 9, 1991, pp. 25-36.

[4] G. Chiola, C. Dutheillet, G. Franceschini, and S. Haddad, "Stochastic Well-Formed Nets (SWNs) and Multiprocesor Modeling Applications," IEEE Trans. on Comp., Vol. 42, no. 11, Nov, 1993, pp. 1343-1360.

[5] K. Begain, G, Farkas, L. Jereb, and M. Telek, "Step-By-Step State Space Generation For the Reliability of Complex Markovian Systems," in Proc. RELECTRONIC '88, 7th Symp. on Rel. in Electronics, Budapest, Hungary, Vol. 1, Sept. 1988, pp. 314-322.

[6] G. Marshall and J. Olkin, "Inequalities: Theory of Majorization and its Applications," Academic Press, 1979.

[7] A. K. Somani and M. Bagha, "Meshkin: A Fault Tolerant Computer Architecture with Distributed Fault Detection and Reconfiguration," in Proc. of 4th Intl. Conf. on Fault Tolerant Computing Systems held at Baden Baden, September 1989, pp. 197-208.

[8] A. Nordseick, "Issues in Chip-Level Redundancy Architectures," Workshop on Wafer-Scale Integration Systems, 1990, pp. 137-141.

[9] A. K. Somani and T. Sarnaik, "Reliability Analysis Techniques for Complex Multiple Fault Tolerant Computer Architectures," IEEE Trans. on Rel., Vol 39, No. 5, Dec. 1990, pp. 547-556.

[10] J. Arlat and J. C. Laprie, "Performance-Related Dependability Evaluation of Supercomputer Systems," in Proc. of FTCS-13, Milano, Italy, June 26-30, 1983, pp. 276-283.

[11] J. T. Blake and K. S. Trivedi, "Reliability Analysis of Interconnection Networks Using Hierarchical Composition," IEEE Trans. on Rel., vol. 38, no. 1, April 1989, pp. 111-120.

[12] C. R. Das, J. T. Kreulen, M. J. Thazhuthaveetil, and L. N. Bhuyan, "Dependability Modeling for Multiprocessors," IEEE Computer, vol. 23, no. 10, Oct. 1990, pp. 7-19.

[13] M. Jerrum, "A Compact Representation for Permutation Group," Journal of Algorithms, vol. 7. no. 1, pp. 60-78, Mar. 1986.

[14] A. Varma and C. S. Raghavendra, "Reliability Analysis of Redundant-Path Interconnection Networks," IEEE Trans. on Rel., vol. 38, no. 1, April 1989, pp. 130-137.

[15] G. Ciardo, J. Gluckman, and D. Nicol, "Distributed State Space Generation of Discrete-State Stochastic Models," ICASE Report 95-75, 1995.

[16] J. B. Dugan, K. S. Trivedi, M. K. Smotherman, and R. Geist, "The Hybrid Automated Reliability Predictor," Journal of Guidance, Control, and Dynamics, Vol. 9, no. 3, May 1986, pp. 319-331.