A Fusion-based Approach for Tolerating Faults in Finite State Machines

Vinit Ogale
Parallel and Distributed Systems Laboratory,
Dept. of Electrical and Computer Engineering,
The University of Texas at Austin.
ogale@ece.utexas.edu

Bharath Balasubramanian
Parallel and Distributed Systems Laboratory,
Dept. of Electrical and Computer Engineering,
The University of Texas at Austin.
balasubr@ece.utexas.edu

Vijay K. Garg *
IBM India Research Lab (IRL),
Delhi, India.
vijgarg1@in.ibm.com

Abstract

Given a set of n different deterministic finite state machines (DFSMs) modeling a distributed system, we examine the problem of tolerating f crash or Byzantine faults in such a system. The traditional approach to this problem involves replication and requires $n \cdot f$ backup DFSMs for crash faults and $2 \cdot n \cdot f$ backup DFSMs for Byzantine faults. For example, to tolerate two crash faults in three DFSMs, a replication based technique needs two copies of each of the given DFSMs, resulting in a system with six backup DFSMs. In this paper, we question the optimality of such an approach and present an approach called (f,m)-fusion that permits fewer backups than the replication based approaches. Given n different DFSMs, we examine the problem of tolerating f faults using just m additional DFSMs. We introduce the theory of fusion machines and provide an algorithm to generate backup DFSMs for both crash and Byzantine faults. We have implemented our algorithms in Java and have used them to automaticaly generate backup DFSMs for several examples.

*supported in part by the NSF Grants CNS-0509024, CNS-0718990, Texas Education Board Grant 781, and Cullen Trust for Higher Education Endowed Professorship. Part of this work was performed when the author was at the University of Texas at Austin.

1. Introduction

A distributed or parallel system can often be modeled as a collection of distinct and independent deterministic finite state machines or DFSMs (also referred to as *machines*). In this paper, we look at the problem of tolerating faults in these machines. Most commonly occurring faults can be categorized as crash (or fail stop) faults [15] and Byzantine faults [10]. In the case of crash faults, there is a loss of the execution state of the machine. In the case of Byzantine faults, the faulty machines can *lie* or reflect an incorrect execution state. In order to build reliable systems, it is important to detect these faults and recover the correct state of the system. As a motivating example, consider a small sensor network with three different sensors running DFSMs measuring the average heat, light and humidity in the environment over a fixed period of time, say a month. During execution, one of these sensors might fail, resulting either in the loss of its state (crash faults) or an inconsistency in the state (Byzantine faults). At the end of the month we need to determine the correct execution state of the system, i.e., the final states of each of the sensors.

Traditional approaches to tolerating f faults in n different DFSMs require some form of replication. In the case of crash faults, we need to maintain f extra copies of each DFSM, resulting in a total of $n \cdot f$ backup DFSMs [8, 14, 18, 16]. In the case of Byzantine faults, since any f machines can lie about their current state,

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

we need to obtain a majority on the current state of any failed DFSM. Hence, we need to maintain $2 \cdot f$ copies of each DFSM, resulting in a total of $2 \cdot n \cdot f$ backup DFSMs [16].

In this paper, we explore an alternate idea for fault tolerance that requires fewer backup machines than replication based approaches. Consider the DFSMs shown in Fig. 1(i) and 1(ii). These machines model mod-3 counters counting 0s and 1s respectively. We denote the number of 0s seen by the counters as n_0 and the number of 1s as n_1. A crash fault in one these machines will result in the loss of its current state. In case of such a failure, we would like to recover the state of the failed machine.

Another way of looking at replication in DFSMs is by constructing a backup machine that is the *reachable cross product* (formally defined in section 2) of the original machines. As shown in Fig. 1 (iii), each state corresponding to this machine is a tuple, in which the first element corresponds to the state of A, and the second element corresponds to the state of B. We would need one such machine to tolerate a single fault. However, the reachable cross product could have a large number of states and would be equivalent to maintaining one copy each of the original DFSMs in terms of complexity. In the example shown in Fig. 1(i) and 1(ii), we can intuitively see that a machine which computes $\{n_0 + n_1\}$ mod 3 (or $\{n_0 - n_1\}$ mod 3) could be used to tolerate a single fault in the system. If machine A that counts n_0 mod 3 fails, then by using machine B (n_1 mod 3) and the machine F_1 ($\{n_0 + n_1\}$ mod 3) we can compute the current state of the failed machine A. Note that, in this case F_1 is much smaller than the reachable cross product (Fig. 1(iii)) with respect to the number of states.

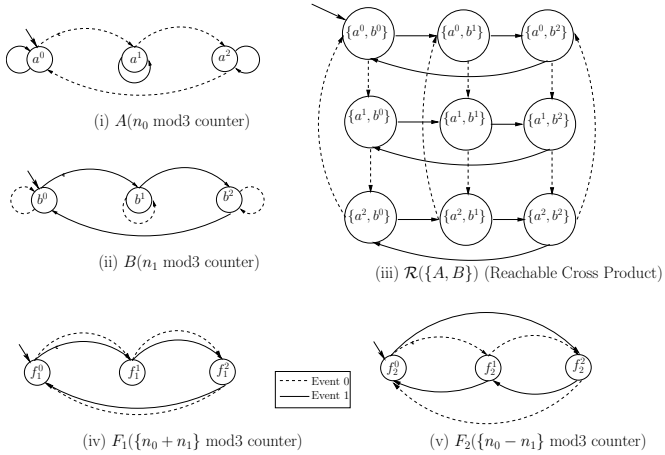

Figure 1. Mod 3 Counters

In the previous example, it was easy to deduce the backup machine purely by observation. For any general set of DFSMs, it is not straightforward to generate such backup machines (for example, consider machines A and B in Fig. 2). The main objective of this paper is to automate the generation of efficient backup machines like F_1 for any given set of machines and formalize the underlying theory. Some of the questions that need to be answered are:

- Given a set of machines, are there backup machines with fewer states than the reachable cross product?
- What is the minimum number of backup machines required to tolerate f crash faults?
- Can these machines tolerate Byzantine faults? (For example, in Fig. 1, DFSMs A and B along with F_1 and F_2 can tolerate one Byzantine fault). What is the number of Byzantine faults that can be tolerated ?
- Is it possible to compute such backup machines efficiently?

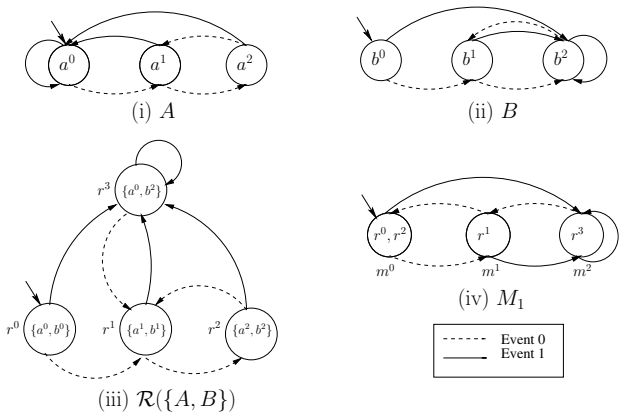

Figure 2. Reachable cross product, Order among DFSMs

In this paper, we explore the idea of a fault graph and use that to define the minimum Hamming distance for a set of machines. Based on this, we introduce an approach called (f, m)-*fusion* that addresses the questions posed. Given n different DFSMs, we examine the problem of tolerating f faults using just m additional machines. Replication is just a special case of (f, m)-fusion where $m = n \cdot f$ for a crash fault tolerant system and $m = 2 \cdot n \cdot f$ for a Byzantine fault tolerant system with n original machines. In general, it is possible to generate backup solutions with fewer machines than replication.

We call the backup machines, *fusions* corresponding to the given set of machines. We assume a system model that has either crash faults or Byzantine faults. Note that, the technique discussed in this paper deals with determining the current *state* of the failed machines and not the entire DFSM (which is usually stored on some form of failure-resistant permanent storage medium).

Our work in [1] introduces the concept of the fusion of finite state machines. That work deals with the special case where the number of backup machines is equal to the number of faults and presents a brute force exponential-time algorithm to generate the equivalent of the (1, 1)-fusion idea in this paper. The system model in that paper only allowed crash faults. In this paper, we generalize that idea to (f, m)-fusion, where we examine the problem of tolerating f faults using m additional machines. Our system model allows both crash and Byzantine faults and we present polynomial time algorithms for generating the backups.

The work presented in [3] introduces the idea of fusible data structures. The authors have shown that commonly used data structures such as arrays, hash tables, stacks and queues can be fused into a single fusible structure, smaller than the combined size of the original structures. Our idea is similar to this approach in the sense that we generate a fused state machine that can enable recovery of any state machine that has crashed. The work presented in this paper effectively presents an algorithm to compute a fusion operation given a set of specific input machines.

In this paper, we exploit the similarity between fault tolerance in DFSMs and fault tolerance in a block of bits using erasure codes [17]. To tolerate bit errors, erasure coding involves adding redundant bits to the data bits. Similarly, to achieve fault tolerance in DFSMs, we add backup machines to the system. We generate the backup machines using the concept of Hamming distances [4]. There is one important difference between erasure codes involving bits and the DFSM problem. In erasure codes, the value of the redundant bits depend on the data bits. In the case of DFSMs, it is not feasible to transmit the state of all the machines after each event transition to calculate the state of the backup machines. Hence, the backup machines have to be designed to act on the same inputs as the original machines and independently transition to suitable states. This implies that instead of designing a erasure code once and reusing it on different inputs, for DFSMs we need to *re-design* the set of backup machines for every distinct set of original machines.

Extensive work has been done [7, 6] on the minimization of completely specified DFSMs. In these approaches, the basic idea is to create equivalence classes of the state space of the DFSM and then combine them based on the transition functions. Even though our approach is also focussed on reducing the reachable cross product corresponding to a given set of machines, it is important to note that the machines we generate need not be equivalent to the combined DFSM. In fact, we implicitly assume that the input machines to our algorithm are reduced a priori using these techniques.

The famous FLP result [2] states that it is impossible to achieve consensus among a given set of machines in an asyncrhonous system with even one faulty machine. Our system model does not assume asynchrony and the state of all the faulty machines are available for recovery. Similarly, there has been a considerable body of research for solving consensus among DFSMs, in synchronous systems. Assume a system of n machines in which upto f machines may be faulty. As far as Byzantine faults are concerned, we cannot achieve consensus unless $n > 3f$ [13]. In the case of crash faults, it has been shown that it will require at least $f + 1$ synchronous rounds to achieve consensus [9]. These results do not apply to this paper because, even though the individual machines may fail, it is assumed that the recovery system, which can detect and correct faults, is not faulty.

In the following sections, we look at the underlying theory behind this approach and also present an efficient algorithm for generating the minimum number of backup machines required to tolerate f faults. Note that, in some cases the smallest fusion could be the reachable cross product machine. However, our experiments suggest that there exist smaller fusions for many of the practical DFSMs in use. This can result in enormous savings in space, especially when a large number of machines need to be backed up. For example, consider a sensor network with 100 sensors, each running a mod-3 counter counting changes to different environmental parameters like temperature, pressure, humidity and so on. To tolerate a crash fault in such a system, replication based approaches would require 100 new sensors for backup. Fusion, on the other hand, could possibly tolerate a fault by using only one new backup sensor with exactly three states. To summarize:

- We introduce the concept of (f, m)-fusion and explore the theory of such machines.
- Using this theory, we present algorithms to generate backup machines and recover from Byzantine or crash failures.
- We have implemented the algorithms in Java and applied them on real world DFSMs.

2. Model and Notation

We now discuss the system model, followed by the notation used in the remainder of this paper. Our system consists of a set of independent servers, which include the original machines and the backups, each running a distinct DFSM with no shared state and no communication during a fault-free run. The events on the DFSMs originate from the environment and are applied to all the machines. For example, one or more client machines (the environment) could send ordered requests (events) which are applied on all the servers. If a received event does not belong to the event set of a server DFSM, then the event is ignored. Note that, synchronous operation is not essential to the underlying theory during normal conditions. The only requirement is that when there are failures, all DFSMs have acted on the same sequence of inputs before the state of the failed DFSM is recovered.

The machines in the system may undergo crash faults or Byzantine faults. We assume that the faults impact only the current state of the faulty machines, and the underlying DFSMs remains intact. When a fault occurs, no requests are sent by the clients till the execution state of the faulty machines have been recovered and the machines resume normal operation. In the case of crash faults, there is a loss of the execution state of the machines. In the case of Byzantine faults, the faulty machines can enter an incorrect state.

Definition 1 *(DFSM) A DFSM, denoted by A, is a quadruple,* $(X_A, \Sigma_A, \alpha_A, x_A^0)$*, where,*

- X_A *is the finite set of states corresponding to A.*
- Σ_A *is the finite set of events corresponding to A.*
- $\alpha_A : X_A \times \Sigma_A \rightarrow X_A$*, is the transition function corresponding to A. If the current state of A is s, and an event* $\sigma \in \Sigma_A$ *is applied on it, the next state can be uniquely determined as* $\alpha_A(s, \sigma)$*.*
- x_A^0 *is the initial state corresponding to A.*

The *size* of a machine A, is the number of states in X_A, and is denoted by $|A|$.

A state, $s \in X_A$, is *reachable* iff there exists a sequence of events, which, when applied on the initial state x_A^0, takes the machine to state s. Our model assumes that all the states corresponding to the machines are reachable.

Consider any two machines, $A = (X_A, \Sigma_A, \alpha_A, x_A^0)$ and $B = (X_B, \Sigma_B, \alpha_B, x_B^0)$. Now construct another machine which consists of all the states in the product set of X_A and X_B with the transition function $\alpha'(\{a, b\}, \sigma) = \{\alpha_A(a, \sigma), \alpha_B(b, \sigma)\}$ for all $\{a, b\} \in X_A \times X_B$ and $\sigma \in \Sigma_A \cup \Sigma_B$. This machine $(X_A \times X_B, \Sigma_A \cup \Sigma_B, \alpha', \{x_A^0, x_B^0\})$ may have states that are not reachable from the initial state $\{x_A^0, x_B^0\}$. If all such unreachable states are pruned, we get the *reachable cross product* of A and B, denoted by $\mathcal{R}(\{A, B\})$. In the example shown in Fig. 2, $\mathcal{R}(\{A, B\})$ is the reachable cross product of A and B.

In the remainder of this paper, for notational convenience, we assume that all machines act the on the same set of events. This event set, denoted by Σ, can be generated by taking the union of the event sets of all original machines. If a received event does not belong to the original event set of a machine, then the event is ignored and the state of that machine remains unchanged. In the following subsection, we define a closed partition lattice corresponding to a given set of machines.

2.1. Closed Partition Lattice

A *partition* P, on the state set X_A of a DFSM, $A = (X_A, \Sigma, \alpha_A, x_A^0)$ is the set $\{B_1, \ldots, B_k\}$, of disjoint subsets of the state set X_A, such that $\bigcup_{i=1}^{k} B_i = X_A$ and $B_i \cap B_j = \phi$ for $i \neq j$ [11]. An element B_i of a partition is called a *block*.

A partition, P, is said to be closed if each event, $\sigma \in \Sigma$, maps a block of P into another block. A closed partition P, corresponds to a distinct machine. Each state s of such a machine corresponds to a set of states in machine A. For example in Fig. 2, M_1 corresponds to a closed partition of the set of states of $\mathcal{R}(\{A, B\})$. M_1 has 3 states, $\{r^0, r^2\}$, $\{r^1\}$ and $\{r^3\}$, which we also refer to as the blocks of M_1. The closed partitions described here are also referred to as substitution property partitions or SP partitions in other literature [5].

A partition P_1 is less than or equal to another partition P_2 ($P_1 \leq P_2$) if each block of P_2 is contained in a block of P_1. If DFSMs X_1 and X_2 correspond to partitions P_1 and P_2 respectively, then machine X_1 is less than or equal to machine X_2 ($X_1 \leq X_2$) iff $P_1 \leq P_2$. In Fig 2, each block of $\mathcal{R}(\{A, B\})$ is contained in a block of M_1 and hence, $M_1 \leq \mathcal{R}(\{A, B\})$. Two machines X_1 and X_2 are said to be incomparable iff $X_1 \not\leq X_2$ and $X_2 \not\leq X_1$.

Consider two machines X_1 and X_2 such that $X_1 \leq X_2$. It is clear that given the state of X_2 we can determine the state of X_1. For example, in Fig 2, $M_1 \leq \mathcal{R}(\{A, B\})$. When $\mathcal{R}(\{A, B\})$ is in state r^1, M_1 is in state m^1.

Given a set of n machines, $\mathcal{A} = \{A_1, \ldots, A_n\}$, their reachable cross product is denoted by $\mathcal{R}(\mathcal{A})$. Every machine in $\mathcal{A}$ is less than or equal to $\mathcal{R}(\mathcal{A})$. Hence, given the state of $\mathcal{R}(\mathcal{A})$, we can determine the state of any of the machines in $\mathcal{A}$.

It can be seen that the set of all closed partitions cor-

responding to a machine, form a lattice under the $\leq$ relation [5]. In this paper, we consider the lattice of all closed partitions corresponding to $\mathcal{R}(\mathcal{A})$. Fig. 3 shows the closed partition lattice corresponding to $\mathcal{R}(\{A, B\})$ (denoted by $\top$), shown in Fig. 2(iii). An arrow from one machine to another indicates that the former is less than the latter. Both A (Fig. 2(i)) and B (Fig. 2(ii)) are contained in the lattice. The bottom element (denoted $\perp$) is always a single block partition containing all the states of $\top$. Henceforth, we use $\top = (X_\top, \Sigma, \alpha_\top, t^0)$ or *top* to denote the reachable cross product of the given set of machines in our system. It is important to note that we never have to generate the entire closed partition lattice in any of our algorithms.

We now define the lower cover of a machine, a concept used later in section 5.

Definition 2 *(Lower Cover) The lower cover of any machine $A = (X_A, \Sigma, \alpha_A, x_A^0)$, is the set of machines corresponding to the maximal partitions of X_A that are less than A.*

For example, in Fig. 3, the lower cover of machine A consist of machines M_3 and M_4.

The lower cover of machine A consists of all the incomparable machines obtained by combining two states of X_A into a block and computing the new largest closed partition which is less than this new (possibly not closed) partition.

In our closed partition lattice, the lower cover of $\top$ is called the *basis* of the lattice. In the lattice shown in Fig. 3, the machines A, B, M_1 and M_2 constitute the basis.

3. Fault Tolerance of Machines

In this section, we introduce concepts that enable us to answer fundamental questions about the fault tolerance in a given set of machines. The proofs for all the theorems in the paper are provided in the technical report [12].

We begin with the idea of a *fault graph* of a set of machines $\mathcal{M}$, for a machine T, where all machines in $\mathcal{M}$ are less than or equal to T. This is a weighted graph and is denoted by $G(T, \mathcal{M})$.

The fault graph is an indicator of the capability of the set of machines in $\mathcal{M}$ to correctly identify the current state of T. As described in the previous section, since all the machines in $\mathcal{M}$ are less than or equal to T, the set of states of any machine in $\mathcal{M}$ corresponds to a closed partition of the set of states of T. Considering the lattice shown in Fig. 3, we construct the fault graph $G(\top, \{A\})$. The machine A has three states, $\{t^0, t^3\}$, $\{t^1\}$ and $\{t^2\}$. Given just the current state of machine A, it is possible to determine if $\top$ is in state t^1(exact) or t^2 (exact) or one of t^0 and t^3(ambiguity). Hence, A distinguishes between all pairs of states of $\top$ except (t^0, t^3). This information is captured by the fault graph.

Every state of T corresponds to a node of the fault graph $G(T, \mathcal{M})$ and the graph is fully connected. The weight of the edge between nodes corresponding to states t^i and t^j of the fault graph is the number of machines in $\mathcal{M}$ that have states t^i and t^j in distinct blocks. Hence, in the fault graph $G(\top, \{A\})$, shown in Fig. 4(i), the edge (t^0, t^3) has weight 0 and all other edges have weight 1.

Definition 3 *(Fault Graph) Given a set of machines $\mathcal{M}$ and a machine $T = (X_T, \Sigma, \alpha_T, t^0)$ such that $\forall M \in \mathcal{M} : M \leq T$, the fault graph $G(T, \mathcal{M})$ is a weighted graph with $|X_T|$ nodes such that*

- *Every node of the graph corresponds to a state in X_T*
- *The graph is fully connected*
- *The weight of the edge between two nodes (corresponding to any two states t^i and t^j in X_T) of the fault graph is the number of machines in $\mathcal{M}$ that have states t^i and t^j in distinct blocks*

Given the states of $|\mathcal{M}| - f$ machines in $|\mathcal{M}|$, it is always possible to determine if T is in state t^i or t^j iff the weight of the edge (t^i, t^j) is greater than f. Consider the graph shown in Fig. 4(ii). Given the state of just any one machine in $\{A, B\}$, we can determine if $\top$ is in state t^0 or t^1, since the weight of that edge is greater than one. We cannot do the same for the edge (t^0, t^3), since the weight of the edge is only one.

To understand the idea of fault graphs and their significance to tolerating faults in state machines, we draw an analogy between fault tolerance in DFSMs and fault tolerance in a block of bits using erasure codes. Consider the fault graph $G(T, \mathcal{M})$, where $T = \mathcal{R}(\mathcal{M})$ is the reachable cross product of $\mathcal{M}$. The state of all the machines in $\mathcal{M}$ can be represented by exactly one of the states in T i.e., the machine T lists all the valid states of the system $\mathcal{M}$. The weights of the edges in the graph $G(T, \mathcal{M})$ are in indicator of the how easy it is to distinguish between those states.

The set of states of $T = \mathcal{R}(\mathcal{M})$ is equivalent to the set of all valid code words in an erasure code. The weight of the edge separating the states is the *Hamming distance* between the valid code words. In this case, instead of conventional bit errors, the states of the machines in $\mathcal{M}$ are either incorrect (Byzantine) or unavailable (crash).

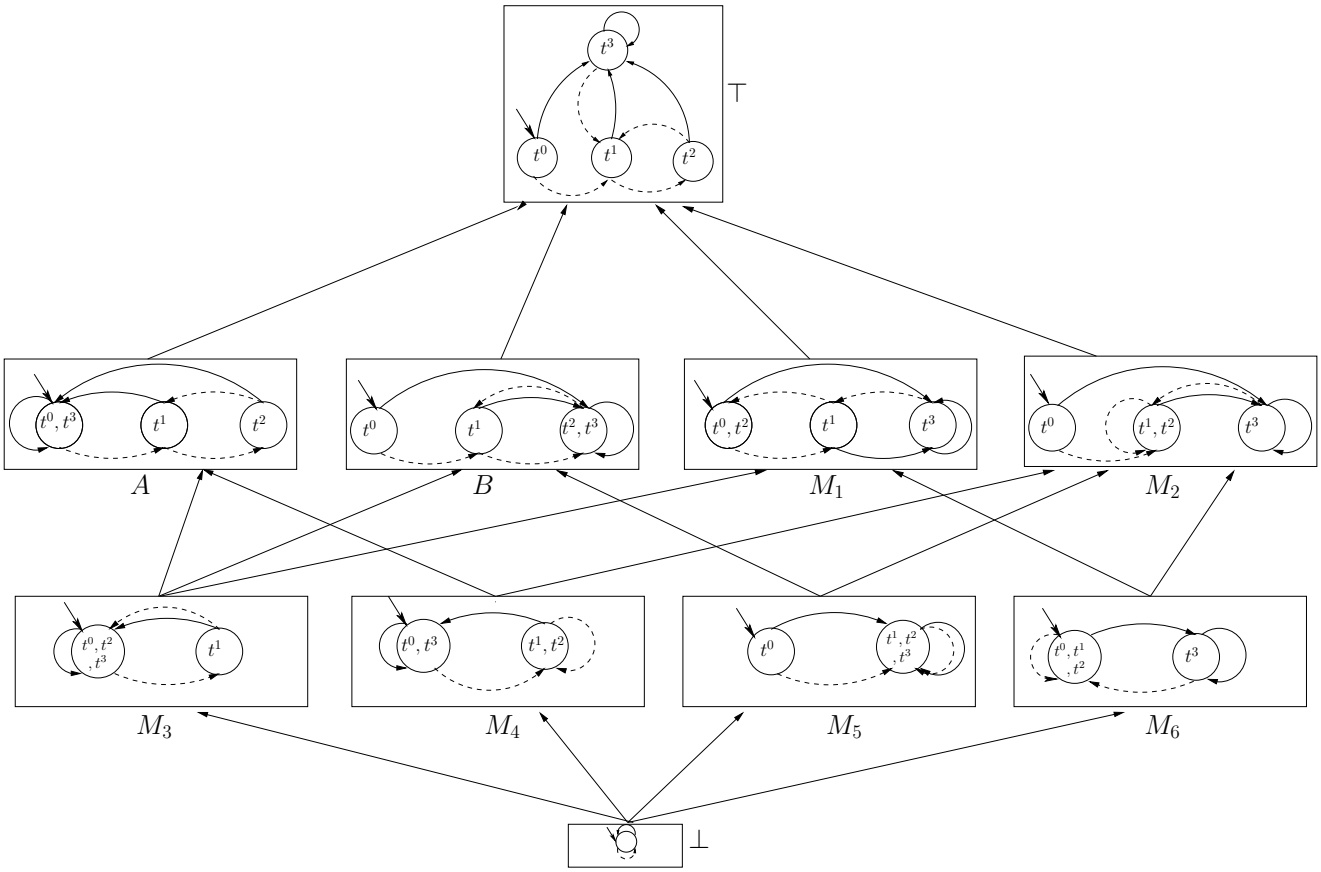

Figure 3. Closed Partition Lattice For Fig. 2

To tolerate bit errors, erasure coding involves adding redundant bits to the data bits. Similarly, in this case we add more machines to the system. These new machines are less than T, the cross product of the original machines. When faults occur, the cross product of the states of the machines in the system may be an incorrect state of T. However, if the system is fault-tolerant, this recovered cross product state will be closest in distance to the correct state in T.

The distance between any two states (or nodes in the fault graph) is the weight of the edge between those nodes in the fault graph.

Definition 4 *(Distance) Given a set of machines $\mathcal{M}$ and their reachable cross product $T = (X_T, \Sigma, \alpha_T, t^0)$, the distance between any two states $t_i, t_j \in X_T$, denoted $d(t_i, t_j)$ is the weight of the edge between the nodes corresponding to t_i and t_j in the fault graph $G(T, \mathcal{M})$.*

Given a fault graph, $G(T, \mathcal{M})$, the smallest distance between the nodes in the fault graph gives us an idea of the fault tolerance capability of the set $\mathcal{M}$. Consider the graph, $G(\top, \{A, B, M_1, M_2\})$, shown in Fig. 4 (iii). Since the smallest distance in the graph is 3, we can remove any two machines from $\{A, B, M_1, M_2\}$ and still regenerate the current state of $\top$. As seen before, given the state of $\top$, we can determine the state of any machine less than $\top$. Therefore, the set of machines $\{A, B, M_1, M_2\}$ can tolerate two crash faults.

The least distance in $G(T, \mathcal{M})$ is denoted $d_{min}(T, \mathcal{M})$.

Theorem 1 *A set of machines $\mathcal{M}$, can tolerate up to f crash faults iff $d_{min}(T, \mathcal{M}) > f$, where T is the reachable cross-product of all machines in $\mathcal{M}$.*

Please refer to the technical report [12] for the proof.

Byzantine faults may include machines which lie about their state. Similar to coding theory, the number of Byzantine faults that can be tolerated by the system of DFSMs is $(d_{min} - 1)/2$. Consider the machines $\{A, B, M_1, M_2\}$ shown in Fig. 3. As shown before, these machines can tolerate two crash faults. Let us consider the case where $\top$ is in state t^3 and two of the machines, say B and M_1, lie about their state. Let the states of the machines A, B, M_1 and M_2 be $\{t^0, t^3\}$, $\{t^0\}$, $\{t^0, t^2\}$ and $\{t^3\}$ respectively. Since, we do not know which machines are lying, we cannot determine the state of $\top$ correctly. If we pick the state which appears the most number of times among these sets, we will determine the state of $\top$ as t^0, which is incorrect. Hence, this set of machines cannot tolerate two Byzantine faults. Assume that only one of the machines, say B, lies about its state. In this case, let the states of the machines A, B, M_1 and M_2 be $\{t^0, t^3\}$, $\{t^0\}$, $\{t^3\}$ and $\{t^3\}$ respectively. Here, we can determine correctly, that the state of $\top$ is t^3, since the majority of machines distinguish between all pairs of states. Hence, since $d_{min}(\top, \{A, B, M_1, M_2\})$ is three, these set of machines can tolerate only one Byzantine fault.

Theorem 2 *A set of machines $\mathcal{M}$, can tolerate up to f Byzantine faults iff $d_{min}(T, \mathcal{M}) > 2f$, where T is the reachable cross-product of all machines in $\mathcal{M}$.*

Please refer to the technical report [12] for the proof.

Just as seen in standard coding theory, the results corresponding to crash and Byzantine faults can be extended to a combination of both these faults.

Observation 1 *A set of machines $\mathcal{M}$, can tolerate up to c crash faults and b Byzantine faults iff $d_{min}(T, \mathcal{M}) > c + 2b$, where T is the reachable cross-product of all machines in $\mathcal{M}$.*

Henceforth, we only consider machines less than or equal to the top element of the closed partition lattice ($\top$) corresponding to the input set of machines $\mathcal{A}$. So, for notational convenience, we use $G(\mathcal{M})$ instead of $G(\top, \mathcal{M})$ and $d_{min}(\mathcal{M})$ instead of $d_{min}(\top, \mathcal{M})$. From theorems 1 and 2, it is clear that we can determine the inherent fault tolerance in a given set of machines $\mathcal{A}$, simply by finding $d_{min}(\mathcal{A})$.

Observation 2 *Given a set of n machines $\mathcal{A}$, the system can tolerate up to $d_{min}(\mathcal{A}) - 1$ crash faults or $(d_{min}(\mathcal{A}) - 1)/2$ Byzantine faults.*

4. Theory of Fusion Machines

To tolerate faults in a given set of machines, we need to add backup machines so that the fault tolerance of the system (original set of machines along with the backups) increases to the desired value. To simplify the discussion, in the remainder of this paper, unless specified otherwise, we mean crash faults when we simply say faults. As seen in theorem 2, the discussion for crash faults also applies to Byzantine faults (where $f/2$ Byzantine faults can be tolerated instead of f crash faults).

Given a set of n machines $\mathcal{A}$, we add m backup machines $\mathcal{F}$, each less than or equal to the top, such that the set of machines in $\mathcal{A} \cup \mathcal{F}$ can tolerate f faults. We call the set of m machines in $\mathcal{F}$, an (f, m)-fusion of $\mathcal{A}$. From theorem 1, we know that, $d_{min}(\mathcal{A} \cup \mathcal{F}) > f$.

Definition 5 *(Fusion) Given a set of n machines $\mathcal{A}$, we call the set of m machines $\mathcal{F}$, an* (f, m)-fusion *of $\mathcal{A}$, if $d_{min}(\mathcal{A} \cup \mathcal{F}) > f$.*

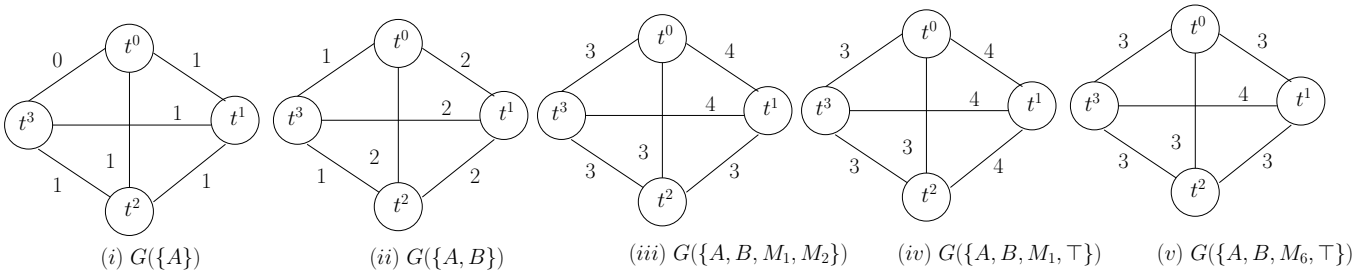

Figure 4. Fault Graphs, $G(\top, \mathcal{M})$, for sets of machines shown in Fig. 3

Any machine belonging to $\mathcal{F}$ is referred to as a *fusion machine* or just a *fusion*. Note that, the top is also a fusion. Consider the set of machines, $\mathcal{A} = \{A, B\}$, shown in Fig. 3. From Fig. 4(ii), $d_{min}(\{A, B\}) = 1$. Hence the set of machines, $\{A, B\}$, cannot tolerate even a single fault.

Let us assume that we want to generate a set of machines $\mathcal{F}$, such that, $\mathcal{A} \cup \mathcal{F}$ can tolerate two faults. It can been seen from Fig. 4(iii) that $d_{min}(\{A, B, M_1, M_2\}) = 3$, and hence the set of machines $\{A, B, M_1, M_2\}$ can tolerate up to two faults. In this case, the set $\{M_1, M_2\}$ forms a $(2, 2)$-fusion of $\{A, B\}$.

Based on the values of f and m, we discuss three cases of (f, m)-fusion:

- $f = m$: In this case, the number of fusion machines equals the number of faults. The set of machines in $\{M_1, M_2\}$, shown in Fig. 3, form a $(2, 2)$-fusion corresponding to $\{A, B\}$.
- $f < m$: The traditional approach of replication is the simplest example for this case. To tolerate two faults in any two machines $\{A, B\}$, replication will require two additional copies each of A and B. Hence, $\{A, A, B, B\}$ is a $(2, 4)$-fusion of $\{A, B\}$.
- $f > m$: From observation 2, if a system is inherently fault tolerant, then no additional machines may be needed to tolerate faults. In the example shown in Fig. 3, let us assume that the original set of machines is $\{A, B, M_1\}$. Since, $d_{min}(\{A, B, M_1\}) = 2$, these machines can tolerate one fault without any additional machine.

We now consider the existence of an (f, m)-fusion for a given set of machines $\mathcal{A}$. The top machine distinguishes between all the states of $X_\top$. If the union of m top machines along with $\mathcal{A}$ cannot tolerate f faults, then there cannot exist an (f, m)-fusion for $\mathcal{A}$. Let us consider the existence of a $(2, 1)$-fusion for the set of machines $\{A, B\}$, shown in Fig. 3. From Fig. 4(ii), $d_{min}(\{A, B\}) = 1$. We need exactly one machine F, such that, $d_{min}(\{A, B, F\}) > 2$. Even if F was the top machine, $d_{min}(\{A, B, \top\}) = 2$. Hence, there cannot exist a $(2, 1)$-fusion for $\{A, B\}$. We formalize this in the following theorem.

Theorem 3 *(Existence of an (f, m)-fusion) Given a set of n machines $\mathcal{A}$, there exists an (f, m)-fusion of $\mathcal{A}$, iff, $m + d_{min}(\mathcal{A}) > f$.*

Please refer to the technical report [12] for the proof. From this theorem, given a set of n machines $\mathcal{A}$ and an (f, m)-fusion $\mathcal{F}$, corresponding to it, $|\mathcal{F}| > f - d_{min}(\mathcal{A})$. Given a set of machines, we now define an order among (f, m)-fusions corresponding to them.

Definition 6 *(Order among (f, m)-fusions) Given a set of n machines $\mathcal{A}$, an (f, m)-fusion $\mathcal{F} = \{F_1, ..F_m\}$, is less than another (f, m)-fusion $\mathcal{G}$ ($\mathcal{F} < \mathcal{G}$) iff the machines in $\mathcal{G}$ can be ordered as $\{G_1, G_2, ..G_m\}$ such that $(\forall 1 \leq i \leq m : F_i \leq G_i) \wedge (\exists j : F_j < G_j)$.*

An (f, m)-fusion $\mathcal{F}$ is *minimal*, if there exists no (f, m)-fusion $\mathcal{F}'$, such that, $\mathcal{F}' < \mathcal{F}$. From Fig. 4(iv), $d_{min}(\{A, B, M_1, \top\}) = 3$, and hence, $\mathcal{F}' = \{M_1, \top\}$ is a $(2, 2)$-fusion of $\{A, B\}$. We have seen that $\mathcal{F} = \{M_1, M_2\}$, is a $(2, 2)$-fusion of $\{A, B\}$. Since $\mathcal{F} < \mathcal{F}'$, $\mathcal{F}'$ is not a minimal $(2, 2)$-fusion. In the lattice shown in Fig. 3, $\{M_3, M_4, M_5, M_6\}$ are a set of minimal machines. It can be seen that $d_{min}(\{A, B, M_3, M_4, M_5, M_6\}) > 2$ and $\{M_3, M_4, M_5, M_6\}$ is a minimal $(2, 4)$-fusion of $\{A, B\}$.

Using the analogy of Hamming distances presented in Section 3, it is easy to see that the idea of fusions and crash fault tolerance can be extended to Byzantine faults. An (f, m)-fusion can tolerate f crash faults or $f/2$ Byzantine faults.

5. Algorithms

In this section, we present algorithms to generate backup machines, detect faults and recover from them.

We first briefly discuss the notation used for representing machines in the algorithms. We are given a set $\mathcal{A}$ of n machines. The following algorithms assume that all state machines are expressed with respect to the reachable cross product of $\mathcal{A}$, denoted $\top$.

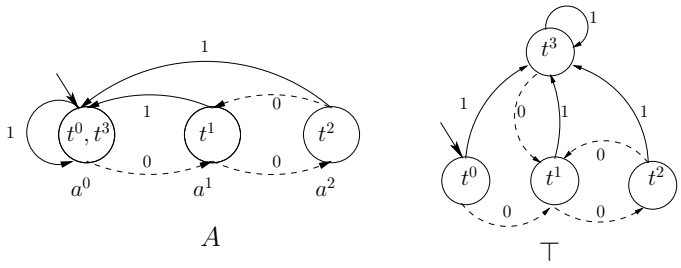

Figure 5. Set Representation of States

All the states in machines less than or equal to $\top$ can be represented as a set consisting of states belonging to $\top$. For example in Fig. 3, the states in all the machines are expressed in this notation. Given machines T and A, such that $A \leq T$, we specify an algorithm in the technical report which outlines the procedure to map the states of T to the states of A.

For example, in Fig. 5, consider the machines $\top$ and A. Since $A \leq \top$, the states of A can be represented as a set of states of $\top$. Firstly, the initial state of $\top$ is mapped to the initial state of A. On the application of event 0 to the initial states of both machines, $\top$ and A transition from $t^0 \rightarrow t^1$ and $a^0 \rightarrow a^1$ respectively. Hence t^1 is mapped to a^1. Similarly, on the application of event 1 to t^0 and a^0 respectively, $\top$ and A transition from $t^0 \rightarrow t^3$ and $a^0 \rightarrow a^0$ respectively. Hence t^3 is mapped to a^0. Continuing this procedure for all the states and events corresponding to the machines, we get the set representation for the states in A. States a^0, a^1 and a^2 can be represented by the sets $\{t^0, t^3\}$, $\{t^1\}$ and $\{t_2\}$ respectively.

5.1. Generation of Backup Machines

In this section, we present an algorithm to generate the minimum set of machines required to tolerate f crash faults among a given set of n machines, with a time complexity polynomial in the size of the top machine. From theorem 2, it is clear that this set of machines can tolerate $f/2$ Byzantine faults.

Given a set of n machines $\mathcal{A}$, algorithm 1 generates the smallest set of machines $\mathcal{F}$, such that, the set of machines in $\mathcal{A} \cup \mathcal{F}$ can tolerate f faults. The outer while loop terminates when the required set of machines is generated, i.e., when the $d_{min}(\mathcal{A} \cup \mathcal{F})$ is equal to (or exceeds) f.

The top ($\top$) is always a valid fusion. Hence, we start with $\top$, which increases d_{min} by one. We then try to find such a machine in the lower cover of $\top$, and continue traversing down the lattice, until we encounter the bottom machine or till the lower cover does not contain such a machine. In the inner loop, the algorithm successively iterates to find out other DFSMs that are less than the previous guess. The inner loop terminates when there is no machine in the lower cover whose inclusion results in the required fault tolerance (line 6 of the algorithm).

Consider the example shown in Fig. 3, with $\mathcal{A} = \{A, B\}$, and $f = 2$. We need to find a set of machines $\mathcal{F}$ such that $d_{min}(\mathcal{A} \cup \mathcal{F}) > 2$. Since $\mathcal{F}$ is empty to begin with, $G(\mathcal{A} \cup \mathcal{F}) = G(\{A, B\})$, shown in Fig. 4(ii). The addition of machine M_1, belonging to the lower cover of $\top$, increases d_{min} of the system by one (i.e. $d_{min}(\mathcal{A}) < d_{min}(\mathcal{A} \bigcup \{M_1\})$). Similarly, machine M_6 in the lower cover of M_1 satisfies this property, while machine M_3 does not. Hence, M_6 is chosen by the algorithm for its next iteration. Since no machine less than M_6 increases d_{min}, M_6 is added to the fusion set.

Algorithm 1 `Generate Fusion`

Input: $\mathcal{A}$: given set of machines, f : number of crash faults to be tolerated
Output: $\mathcal{F}$: set of fusion machines

1: $\mathcal{F} \leftarrow \phi$;
2: **while** $d_{min}(\mathcal{A} \cup \mathcal{F}) \leq f$ **do**
3: $M \leftarrow \top$
4: **while** $M \neq \perp$ **do**
5: $C \leftarrow$ `lower_cover`(M)
6: **if** $\exists F \in C$: $d_{min}(\mathcal{A} \bigcup \mathcal{F} \bigcup \{F\}) > d_{min}(\mathcal{A} \bigcup \mathcal{F})$ **then**
7: $M \leftarrow F$
8: **else**
9: break
$\mathcal{F} \leftarrow \{M\} \bigcup \mathcal{F}$
10: **return** $\mathcal{F}$

Given a set of n machines $\mathcal{A}$, algorithm 1 returns the smallest set of machines $\mathcal{F}$, such that $\mathcal{F}$ is a minimal $(f, |\mathcal{F}|)$-fusion of $\mathcal{A}$. There may be other solutions which have more machines, where each of the machines is less than the machines returned by this algorithm. The proof of correctness of this algorithm is presented in the technical report [12].

Assuming that $|X_\top| = N$, the time complexity of algorithm 1 is $O(N^3 \cdot |\Sigma| \cdot f)$. For a detailed analysis of the time complexity, please refer to the technical report

[12].

5.2. Recovering from Faults

Given a set of n machines $\mathcal{A}$, and a corresponding (f, m)-fusion $\mathcal{F}$, we present an algorithm to recover from f crash faults or $f/2$ Byzantine faults.

As mentioned earlier, we use the set representation for all the states of the machines in our system.

Given the current state of all the machines in our system, algorithm 2 returns the correct state of the top machine from which the state of all the individual machines can be determined. The algorithm iterates through the current states of all the machines in the system and picks the state, $t_c \in X_\top$, which appears the most number of times in these states.

Algorithm 2 `Recover`

Input: S : set of current states of the machines in $\mathcal{A} \cup \mathcal{F}$,
`count` : vector of size N initialized to 0

Output: t_c : the correct state of the top machine.

1: **for all** $s \in S$ **do**
2: **for all** $t_i \in s$ **do**
3: $++$ `count`$[i]$
4: **return** t_c : $1 \leq c \leq N$ and `count`$[c]$ is the maximal element in `count`

Consider the machines A, B, M_1 and M_2 shown in Fig. 3. As shown before, this system of machines can tolerate two crash faults and one Byzantine fault. Let us consider the case of crash faults, where both machines B and M_1 have crashed and the machines A and M_2 are in states $\{t^0, t^3\}$ and $\{t^3\}$ respectively. Algorithm 2 will return t^3 since the count corresponding to t^3, `count`[3] = 2, which is greater than the count corresponding to t^0 (`count`[0] = 1), t^1 (`count`[1] = 0) and t^2 (`count`[2] = 0).

Now let us assume that machines A, B and M_2 are in states in $\{t^0, t^3\}$, $\{t^0\}$, $\{t^0\}$. Assume that M_1 has a Byzantine failure reflecting an incorrect state $\{t^1, t^2, t^3\}$. Algorithm 2 will return t^0 since the count corresponding to t^0, `count`[0] = 3, which is greater than the count corresponding to t^1 (`count`[1] = 1), t^2 (`count`[2] = 1) and t^3 (`count`[3] = 2).

Assuming that $|X_\top| = N$, the time complexity of algorithm 2 is $O((n + m) \cdot N)$. Please refer to the technical report [12] for a detailed analysis of the time complexity and the proof of correctness.

In many situations, it is important to detect the faulty machines. In the case of crash faults, we assume that the recovery system can identify the faulty machines. In the case of Byzantine faults, we will need an explicit algorithm to detect the machines which are lying. A machine can lie either by reflecting an incorrect state or a state which is not even present in it's state set. In the example shown above , M_2 refelects the state $\{t^1, t^2, t^3\}$, which is not part of it's state set (refer to Fig. 3). In order to detect the faulty machines, we first need to obtain the correct state of the top machine, t_c, using algorithm 2. We can then exmamine the state of each of the machines at the point of failure. Every state which does not contain t_c or which is not present in the state set of it's machine, belongs to a faulty machine.

6. Implementation and Results

We have implemented the algorithm specified in section 5 in Java (JDK 6.0) on a machine with an Intel Core Duo processor with 1.83 GHz clock frequency and 1 GB RAM. We tested the algorithms for many practical DFSMs including TCP and the MESI cache coherency protocol along with the examples shown in Fig. 2 (denoted A and B in the results table). In this section we present our results for crash faults.

In the results table, along with the original machines, we have tabulated the number of crash faults tolerated (f), the size of the top ($|\top|$) and the sizes of the backup fusion machines generated by algorithm 1 (|Backup Machines|).

Original Machines	f	$\|\top\|$	|Backup Machines|
MESI, 1-Counter, 0-Counter, Shift Register	2	87	[39 39]
Even Parity , Odd Parity Checker, Toggle Switch, Pattern Generator, MESI	3	64	[32 32 32]
1-Counter, 0-Counter, Divider, A, B	2	82	[18 28]
MESI, TCP, A, B	1	131	[85]
Pattern Generator, TCP, A, B	2	56	[44 56]

The results indicate that there are many practical examples for which our algorithm yields huge savings in state space compared to replication based approaches. Since the largest running time for the execution of our program was only 13.2 minutes, our algorithm can be used to generate backup machines for larger, more complex machines within a feasible time frame.

7. Conclusion and Future Work

In this paper, we have introduced the idea and theory of (f, m)-fusion. We present polynomial time algorithms to generate the minimum set of machines required to tolerate both crash and Byzantine faults among a given set of machines and recover from these faults. We have also implemented and tested this algorithm for real world DFSM models such as TCP and MESI.

The concept of (f, m)-fusion gives us a wide spectrum of choices for fault-tolerance. Replication is just a special case of (f, m)-fusion. Our approach shows that there are many cases for which we can do significantly better than replication. Hence, if we want to tolerate 5 crash faults among 1000 machines, replication will require 5000 extra machines. Using our algorithm, we may achieve this with just 5 extra machines.

In this paper, we only consider machines belonging to the closed partition lattice of $\top$. It would be interesting to explore machines outside the lattice for alternate solutions. Algorithm 1 presented in this paper returns the minimum number of backup machines required to tolerate faults in a given set of machines. We may be able to generate smaller machines if the system under consideration permits a larger number of backup machines.

8. Acknowledgement

We are grateful to Yogish Sabharwal, IBM India Research Lab, for discussions involving Byzantine fault tolerance in DFSMs. We would also like to thank the NSF for supporting this research.

References

[1] B. Balasubramanian, V. Ogale, and V. K. Garg. Fault tolerance in finite state machines using fusion. In *Proceedings of International Conference on Distributed Computing and Networking (ICDCN) 2008, Kolkata*, volume 4904 of *Lecture Notes in Computer Science*, pages 124–134. Springer, 2008.

[2] M. J. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), Apr. 1985.

[3] V. K. Garg and V. Ogale. Fusible data structures for fault tolerance. In *ICDCS 2007: Proceedings of the 27th International Conference on Distributed Computing Systems*, June 2007.

[4] R. Hamming. Error-detecting and error-correcting codes. In *Bell System Technical Journal*, volume 29(2), pages 147–160, 1950.

[5] J. Hartmanis and R. E. Stearns. *Algebraic structure theory of sequential machines (Prentice-Hall international series in applied mathematics)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1966.

[6] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[7] D. A. Huffman. The synthesis of sequential switching circuits. Technical report, Massachusetts, USA, 1954.

[8] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer networks*, 2:95–114, 1978.

[9] L. Lamport and M. Fischer. Byzantine generals and transaction commit protocols. Technical report, 1982.

[10] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[11] D. Lee and M. Yannakakis. Closed partition lattice and machine decomposition. *IEEE Trans. Comput.*, 51(2):216–228, 2002.

[12] V. Ogale, B. Balasubramanian, and V. K. Garg. A report on fusion-based approach for tolerating faults in finite state machines. Technical Report TR-PDS-2008-003, `http://maple.ece.utexas.edu/TechReports/2008/TR-PDS-2008-003.pdf`, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 2008.

[13] M. Pease and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228–234, 1980.

[14] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[15] F. B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984.

[16] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[17] C. E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. J.*, 27:379–423, 623–656, 1948.

[18] F. Tenzakhti, K. Day, and M. Ould-Khaoua. Replication algorithms for the world-wide web. *J. Syst. Archit.*, 50(10):591–605, 2004.