The Dissertation Committee for Vinit Arun Ogale
certifies that this is the approved version of the following dissertation:

Detecting and Tolerating Faults in Distributed Systems

Committee:

Vijay K. Garg, Supervisor

Aristotle Arapostathis

Craig Chase

Mohamed G. Gouda

Sarfraz Khurshid

Alper Sen

Detecting and Tolerating Faults in Distributed Systems

by

Vinit Arun Ogale, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2008

Dedicated to my parents and my little niece, Sakshi.

Acknowledgments

I am greatly indebted to Vijay K. Garg for making my time in graduate school a truly wonderful experience. It is impossible for me to express my gratitude for him in mere prose. If I was a poet, I could have written an ode describing how he encouraged and guided me. If I could paint, I could have perhaps portrayed how he was instrumental in igniting my passion for research. However, I'll just have to be content by thanking him for being such a great teacher, supervisor and friend.

I would also like to thank my committee members Ari Arapostathis, Craig Chase, Mohamed Gouda, Sarfraz Khurshid and Alper Sen for their feedback which help me shape this dissertation. I consider myself lucky that I had to chance to work with, and learn from, Craig Chase. Working as a teaching assistant for him was one of the best jobs that I ever had and I hope I have the chance to work again with him again in the future. I would also like to thank Alper for his guidance and help throughout my PhD.

Working in the lab was never boring thanks to Selma, Bharath, Arindam and Anurang. It is a pleasure to work with motivated people like them and I always had a critical ear available whenever I needed it. I would like to thank Bharath for helping me proofread this dissertation and for being such a great friend.

An equally important part of my life has been my friends in graduate school. I would like to thank Tanmay Patel for almost everything under the sun, from being a project partner in a class to going on long treks with me to remote mountains. I am incredibly lucky to have met Suvid Nadkarni and I have always counted on his unwavering support and friendship whenever times were rough. Research might have become boring if it was not for the uncountable coffee breaks and the humor of Sundar Subramanian. I will always remember fondly all those fun discussions on the weirdest of topics, from Math to Literature to Philosophy. Thanks to Ripple, Sriram, Mihir, Nachiket, Murat, Harshal, Yuklai, Romi, and Jay for being such wonderful friends and the fun times we had together.

A special thanks to Rucha for her encouragement and also for prodding me on whenever I was bored or lazy while writing this thesis.

I would have never completed my PhD or even become an engineer, if it was not for the motivation and support of my brother, Anil Ogale, and my parents. I would like to thank my brother for always believing in me and helping me stand up again whenever I stumbled. I would also like to thank my parents, Arun and Nanda, for their unconditional love and support through all these years.

Detecting and Tolerating Faults in Distributed Systems

Publication No. _____

Vinit Arun Ogale, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Vijay K. Garg

This dissertation presents techniques for detecting and tolerating faults in distributed systems.

Detecting faults in distributed or parallel systems is often very difficult. We look at the problem of determining if a property or assertion was true in the computation. We formally define a logic called BTL that can be used to define such properties. Our logic takes temporal properties in consideration as these are often necessary for expressing conditions like safety violations and deadlocks.

We introduce the idea of a basis of a computation with respect to a property. A basis is a compact and exact representation of the states of the computation where the property was true. We exploit the lattice structure of the computation and the structure of different types of properties and avoid brute force approaches. We have shown that it is possible to efficiently detect all properties that can be expressed by using nested negations, disjunctions,

conjunctions and the temporal operators *possibly* and *always*. Our algorithm is polynomial in the number of processes and events in the system, though it is exponential in the size of the property.

After faults are detected, it is necessary to act on them and, whenever possible, continue operation with minimal impact. This dissertation also deals with designing systems that can recover from faults. We look at techniques for tolerating faults in data and the state of the program. Particularly, we look at the problem where multiple servers have different data and program state and all of these need to be backed up to tolerate failures. Most current approaches to this problem involve some sort of replication. Other approaches based on erasure coding have high computational and communication overheads.

We introduce the idea of *fusible data structures* to back up data. This approach relies on the inherent structure of the data to determine techniques for combining multiple such structures on different servers into a single backup data structure. We show that most commonly used data structures like arrays, lists, stacks, queues, and so on are fusible and present algorithms for this. This approach requires less space than replication without increasing the time complexities for any updates. In case of failures, data from the back up *and* other non-failed servers is required to recover.

To maintain program state in case of failures, we assume that programs can be represented by deterministic finite state machines. Though this approach may not yet be practical for large programs it is very useful for small concurrent programs like sensor networks or finite state machines in hardware

designs. We present the theory of *fusion* of state machines. Given a set of such machines, we present a polynomial time algorithm to compute another set of machines which can tolerate the required number of faults in the system.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

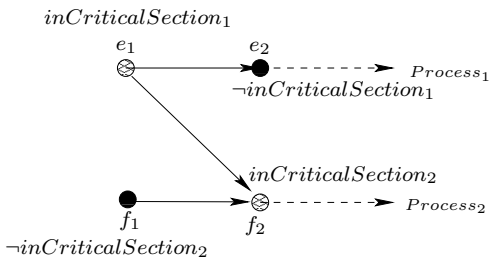
Rapid improvements in hardware and communication infrastructure have propelled distributed and parallel systems from a niche to the mainstream of the computing world. Software tools and programming paradigms have been slower to adapt to this change in the underlying hardware and communication infrastructure. Even today, specialized tools and languages like Erlang for distributed or parallel programs are far from popular. The inherent non-determinism in distributed programs and presence of multiple threads of control make it difficult to write correct distributed software using conventional paradigms. To compound this problem, most currently known techniques for fault detection and fault handling of sequential programs do not scale gracefully in distributed systems. Hence the effects of faults, whether the result of the environment or a human mistake, are amplified in the case of distributed systems.

Our research approaches two problems faced in designing and deploying any distributed or parallel system: detecting faults and tolerating faults.

1.1 Detecting Faults

Fault detection encompasses a myriad of approaches from model checking to manual program testing, each with its own pros and cons. We focus on detecting if a distributed program executed correctly. In many distributed systems, it is often desirable to have a formal guarantee that the program output is correct. One approach is to model check the entire program with respect to the given specification. This is impractical even for most moderately complex programs. For many applications, predicate detection offers a simple and efficient alternative over model checking the entire program. Predicate detection involves verifying the execution trace of a distributed program with respect to a given property (for example, violation of mutual exclusion). For example, in scientific computing, it may be vital to verify that the result of a computation was valid, and if it was invalid due to a rare 'chance' bug, the program can be re-executed. In some cases (especially for *transient bugs*) it maybe possible to automatically add extra synchronization to the program so that the bug does not recur. Predicate detection provides a formal guarantee on the validity of the computation (assuming that the specifications are correct).

A distributed computation, i.e., the execution trace of a distributed program, can either be modeled as a total order, or as a partial order on the set of events in the computation. Representing the computation as a total order can mask some of the bugs in other possible consistent interleavings. A partial order, in contrast, captures all the possible causally consistent interleavings. We use a partial order representation based on Lamport's *happened before*



(i) Partially ordered trace

(ii) Consistent total order without mutex violation

(iii) Consistent total order with mutex violation

Figure 1.1: Partial And Total Orders

relation [33]. For example, consider the partial order trace in figure 1.1(i) and the corresponding totally ordered traces in figures 1.1(ii) and 1.1(iii). If the bug to be detected is represented by the predicate $inCriticalSection_1 \wedge inCriticalSection_2$ then we can see that the total order in figure 1.1(ii) masks the bug that is seen in the total order in figure1.1(iii). Hence, it is better to maintain a partial ordered trace that represents all possible total interleavings rather than maintaining one of the totally ordered traces.

The drawback of using a partial order trace model is that the number of global states of the computation is exponential in the number of processes. This makes predicate detection a hard problem in general [5, 63]. A number

of strategies like symbolic representation of states and partial order reduction have been explored to tackle the state explosion problem [11, 20, 41, 49, 62, 64, 71].

1.1.1 Contribution

We present a technique to efficiently detect all temporal predicates that can be expressed in, what we call, Basic Temporal Logic or BTL. An example of a valid BTL predicate would be a property based on local predicates and arbitrarily placed negations, disjunctions and conjunctions along with the possibly($\Diamond$) and invariant($\Box$) temporal operators (the EF and AG operators defined in [8]).

Our algorithm is based on computing a *basis* which is a compact representation of the subset of the computational lattice containing exactly those global states (or cuts) that satisfy the predicate. In general, it is hard to efficiently compute a basis for an arbitrary predicate. We utilize the fact that the set of global states of a computation forms a distributive lattice and restrict the predicates to BTL formulas. The basis introduced in this paper is a union of smaller sets of cuts called semiregular structures.

Note that, without any restrictions on the predicate formula form, predicate detection is NP-complete with respect to the formula size, and for arbitrary predicates the time complexity could be exponential in the formula size. However, if the input formula is in a 'DNF like' form after pushing in negations, our technique detects it in polynomial time with respect to the formula

size.

Note that other known approaches, like model checking of traces, for detecting a similar class of predicates, are inefficient and require exponential time with respect to the number of processes. *Slicing*, introduced in [43] can be thought of as a special case of our approach.

We validate the practical utility of our technique with experimental studies. The algorithm for computing bases of a computation has been implemented in a prototype tool BTV. BTV is a program agnostic tool, that is it accepts compatible traces generated by a program in any language or platform. The working of the tool is independent of the program generating the traces. The tool accepts traces and the predicate as the input and returns the output of our predicate detection algorithm. To generate traces for testing and to test its utility for real world scenarios, we modified the SystemC kernel (SystemC [30] is a high level hardware design language which is popular for SoC designs). Thus any concurrent hardware model in SystemC can be tested by using this modified kernel along with the BTV tool.

1.2 Tolerating Faults

Once a fault is detected, the program can be halted (and possibly restarted) or the fault could be circumvented. For example, the computation could be rolled back and re-executed if the fault is known to be transient. In other circumstances, when the exact cause of the fault is known, the program execution can be modified (for example, adding extra synchronization)

to prevent the halt from recurring.

Another way of handling faults is to design the system to expect and act on faults. This is typically achieved by adding a certain level of redundancy to the data or the program. Data and program replication are often used to *tolerate* faults and recover from them.

We will focus on fault tolerant data structures and fault tolerance in deterministic finite state machines . Replication is a commonly used technique to achieve fault-tolerance in face of various failures in a distributed system. It is almost considered a self-evident truth that, to tolerate crash of t servers, one must have $t + 1$ copies of identical processes. This approach, for example, is the underlying assumption in the work on replicated state machine approach [12, 34, 48, 53, 61, 65, 70]. In that work, if all $t + 1$ state machines (or servers) start with the identical state, are completely deterministic in execution, and agree on the set and the order of commands they execute, then they will have the identical state at all times. This means that failure of t of them will leave at least one copy available. The optimality of this approach has generally not been questioned.

We initiate study of fusible data structures that allow practical techniques for fault-tolerance with lower space and communication overhead than replication.

1.2.1 Fusible Data Structures

In data storage and communication, coding theory is extensively used to recover from faults. For example, RAID disks use disk striping and parity based schemes (or erasure codes) to recover from the disk faults [7, 47, 50]. As another example, network coding [3, 40] has been used for recovering from packet loss or to reduce communication overhead for multi-cast. In these applications, the data is viewed as a set of data entities such as disk blocks for storage applications and packets for network applications.

By using coding theory techniques [38], one can get much better space utilization than, for example, simple replication. To tolerate crash failures for servers, one can view the memory of the server as a set of pages and apply coding theory to maintain code words. This approach, however, may not be practical because a small change in data may require re-computation of the backup for one or more pages. Since this technique is oblivious to the structure of the data, the details of actual operations on the data are ignored and the coding techniques simply recompute the entire block or page of data.

Hence currently used techniques suffer from one of these two drawbacks:

- (Replication techniques) They require a large number of redundant servers.
- (Coding theory) They are data oblivious and may require higher computational and communication overhead.

1.2.1.1 Contribution

We introduce the concept of *fusible data structures* that enable us to efficiently maintain fault tolerant data in parallel or distributed programs. Our technique revolves around the actual structure of the data and the operations used to change the data. We exploit our knowledge of the data structure and the permitted operations to reduce the space and communications overhead and, at the same time, allowing incremental updates to the data. In a way, our technique is a hybrid of replication and coding theory approaches. The trade-off is that this technique depends on the specific type of data structure used and different algorithms will be required for various data structures. Another part of this research includes discovering efficient algorithms for fusion of commonly used data structures and developing a library for these structures so that distributed system programmers can transparently use fusion-backed up data structures without additional effort or change in the program logic.

As a concrete example, consider a lock server in a distributed system that maintains and coordinates the use of a lock. Figure 1.2 shows such a system with four lock servers, each servicing some clients independently. Each lock server maintains the record of the process that has the lock and maintains the queue of all pending requests. Assume that the size of the pending request queue is n_{max}. Traditionally, if fault-tolerance from a crash is required, we would keep two copies of the queue. If there are k such lock servers in the system, and each one is replicated, we require a space overhead of kn_{max}. Instead fusible data structures, allow us to keep a single back-up data structure

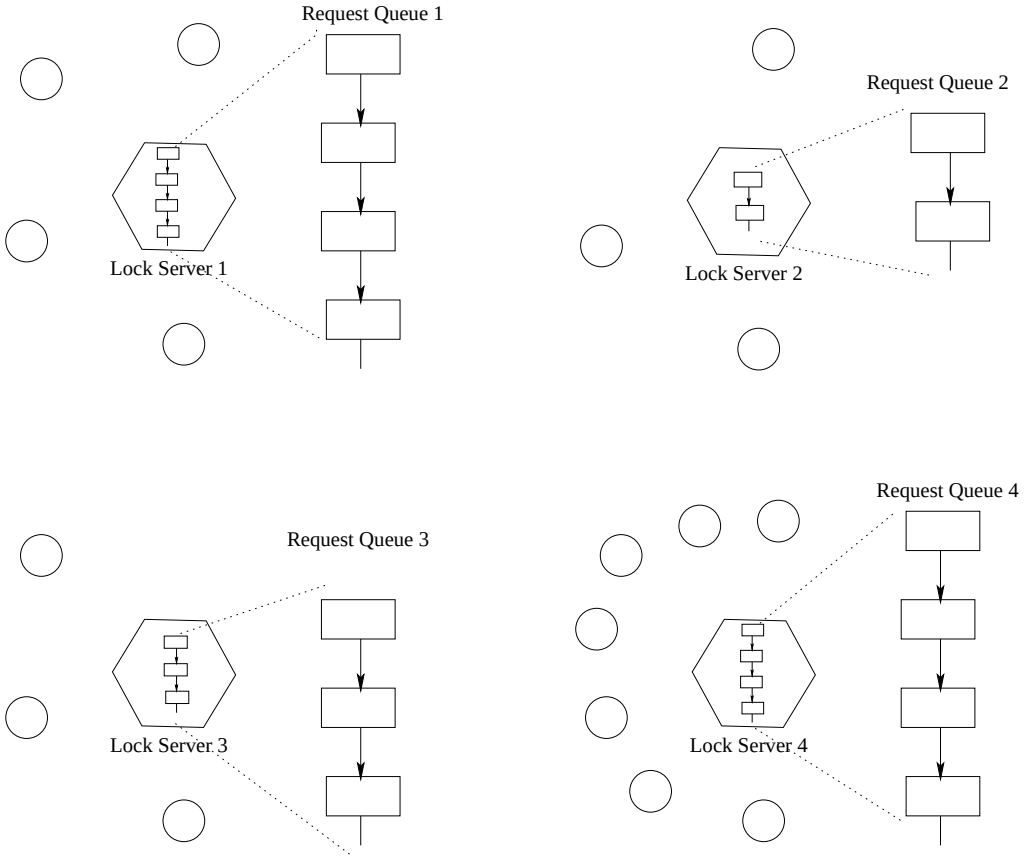


Figure 1.2: System of Four Independent Servers

for all k servers. This back up that is obtained by *fusing* the original queues. In this case, the notion of fusion roughly corresponds to xoring the individual data cells together, while maintaining some additional information like the heads of the queues. As we shall discuss later, the fused queue uses $O(n_{max})$ space, supports recovery and can be updated efficiently when any of the primary queue gets updated. This technique results in k-fold savings.

1.2.2 Fault Tolerance in Finite State Machines

Along with tolerating faults in data, it is also important to recover the program state in case of a failure. A distributed system may be viewed as a set of distinct and independent DFSMs. Hence, we look at the problem of recovering the state of one or more failed DFSMs among the original set of DFSMs. For example, consider a small sensor network with three different sensors measuring the heat, light and humidity in the environment. Assume that these sensors can be modeled as DFSMs and if one of the sensors fail, we need to determine its value (that is, the current state of the DFSM representing the sensor).

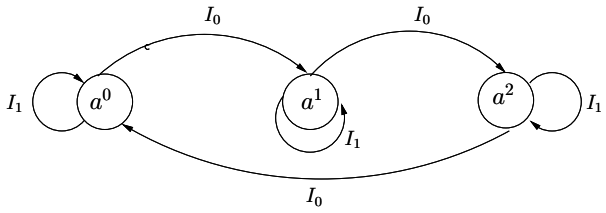
Consider the DFSMs shown in figures 1.3(i) and 1.3(ii). These machines model mod-3 counters operating on different inputs, I_0 and I_1. Assume that one of these machines fail, i.e., the current state of the machine is lost. In case of such a failure, we would like to recover the state of the failed machine. Traditional approaches to this problem require some form of replication. One commonly used technique, which forms the basis of the work done in [12, 34, 48,

53, 61, 70], involves replicating the server DFSMs and sending client requests in the same order to all the servers. Another approach, seen in [2, 65], involves designating one of the servers as the primary and all the others as backups. Client requests are handled by the primary server until it fails, and then one of the backups take over. In both these approaches, to tolerate f faults in n different DFSMs, we need to maintain f extra copies of each DFSM, resulting in a total of $n.f$ backup DFSMs.

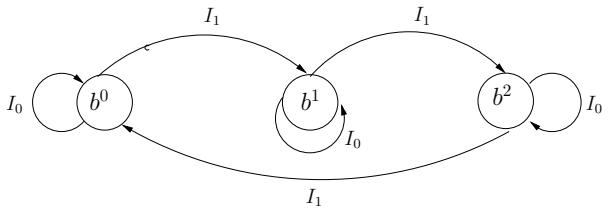
Another way of looking at replication in DFSMs is to construct a machine which contains all states obtained by computing the product set of the states of the original DFSMs. Such a DFSM is called the cross-product of the original DFSMs. We would need one such machine to tolerate a single fault. However, the cross product machine could have a large number of states and would be equivalent to maintaining one copy each of the original DFSMs in terms of complexity.

1.2.2.1 Contribution

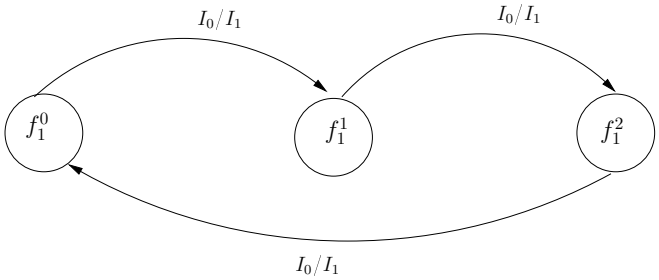
In the example shown in figure 1.3(i) and 1.3(ii), we can intuitively see that a machine which computes $I_0 + I_1 \bmod 3$ (or $I_0 - I_1 \bmod 3$) could be used to tolerate a single fault in the system. If machine A that counts $I_0 \bmod 3$ fails, then by using machine B ($I_1 \bmod 3$) and the machine F_1 ($I_0 + I_1 \bmod 3$) 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 productwith respect to the number of states.



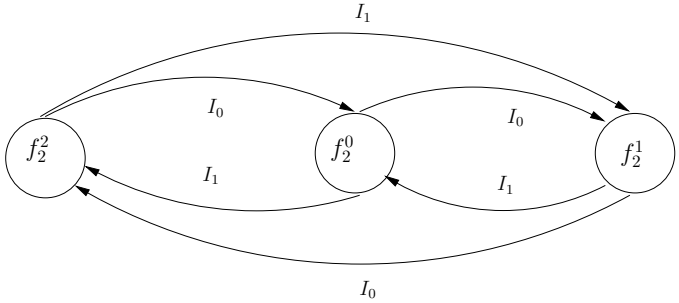
(i) A (mod-3 counter)



(ii) B (mod-3 counter)



(iii) F_1 (mod-3 $I_0 + I_1$ counter)



(iv) F_2 (mod-3 $I_0 - I_1$ counter)

Figure 1.3: 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. Unlike the example in figure 1.3, it is not intuitive whether the machines A and B in figure 1.4 can be efficiently backed up. The main objective of this research 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 original machines, does there exist a more efficient backup machine than the reachable cross product?
- Could we have multiple backup machines enabling design of systems that tolerate multiple faults? (For example, in figure 1.3, DFSMs A and B along with F_1 and F_2 can tolerate two faults. Is it possible to tolerate three faults by adding another machine?).
- What is the minimum number of backup machines required to tolerate f faults?
- Is it possible to compute such backup machines efficiently?

We introduce an approach called (f, m)-*fusion*, that addresses these questions. Given n different DFSMs, we tolerate f faults by having m ($m \leq n.f$) backup DFSMs as opposed to the $n.f$ DFSMs required in the replication based approaches. We call the backup machines, *fusions* corresponding to the given set of machines. Replication is just a special case of our approach with

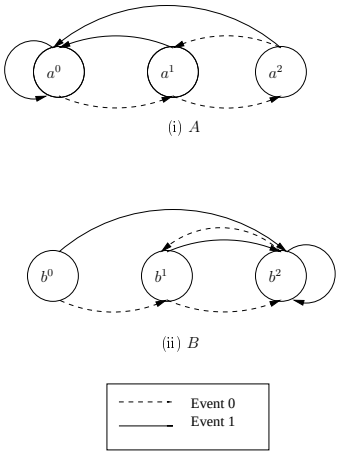


Figure 1.4: Finite State Machines

$m = n.f$. We assume a system model that has fail-stop faults [52]. 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).

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 most efficient fusion could be the reachable cross product machine. However, our experiments suggest that there exist efficient 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 fault in such a system, replication based approaches would demand 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.

In this dissertation we addresses all the questions that were posed earlier. To summarize:

- We introduce the concept of (f, m)-fusion, formalize the idea and explore the theory of such machines.
- Using this theory, we present an efficient algorithm for generating the smallest set of backup machines, to tolerate f faults in a given set of machines. We have implemented this algorithm and tested it with real world DFSMs.

1.3 Overview of the Dissertation

The remainder of this dissertation is organized as follows. In chapter 2, we go over some of the background concepts used in our research. This deals with lattice theory concepts and partial ordered representation of computations. In chapter 3, we introduce the predicate detection algorithm using bases. Chapter 4 deals with fusible data structures. In chapter 5, the algorithms to fuse state machines for efficient backups are discussed. We conclude the dissertation and enlist avenues for future research in chapter 6.

Chapter 2

Background

In this chapter we present some of the background of the concepts and define notation that will be used later in this dissertation.

A *relation* R over any set $\mathcal{C}$ is a subset of $\mathcal{C}\mathtt{x}\mathcal{C}$. A *partial order* over a set is any relation that is both irreflexive and transitive. A set, along with a partial order on its elements, is denoted by $\langle \mathcal{C}, \leq \rangle$ and is called a partially ordered set or a *poset*.

We now define the concept of a covering element.

Definition 2.0.1. Given a partially ordered set $\mathcal{C}$ and let x, $y \in \mathcal{C}$. We say that x covers y if $x < y$ and $x \leq z < y$ implies $x = z$.

This leads to the definitions of lower and upper covers.

Definition 2.0.2. (Covers) Given a poset $\mathcal{C}$, a lower cover of $x \in \mathcal{C}$ is the set $L_x = \{y | y \in \mathcal{C} \wedge x \text{ covers } y\}$. Similarly the upper cover is the set $U_x = \{y | y \in \mathcal{C} \wedge y \text{ covers } x\}$.

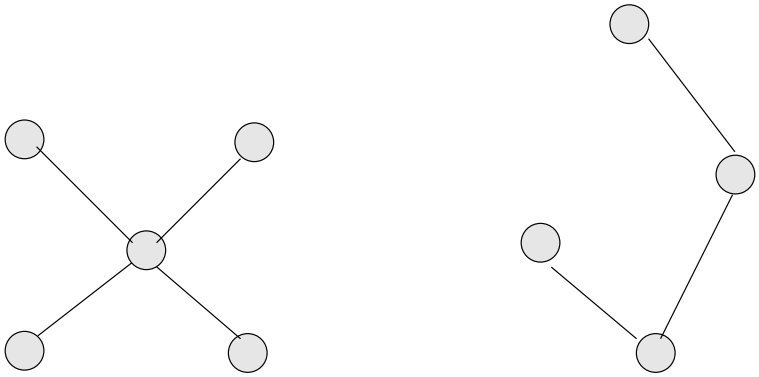


Figure 2.1: Hasse diagrams

2.1 Representing Partial Orders

It is often convenient to represent partial orders graphically. In this dissertation we shall use a representation commonly called the *Hasse diagram*[9]. A Hasse diagram of the set $\mathcal{C}$ is constructed as follows:

1. Each element of $\mathcal{C}$ is represented by a small circle or a dot.
2. If x covers y in $\mathcal{C}$ then x is visually above y in the diagram.
3. There is a line connecting x and y iff x covers y or y covers x.

Some examples of Hasse diagrams are shown in figure 2.1.

In chapter 3, we shall deal with posets representing the execution traces of distributed or parallel *computations*. We use a graphical notation similar in concept to Hasse diagrams for these posets. To construct the diagrams representing a computation $\mathcal{C}$:

1. Each element of $\mathcal{C}$ is represented by a small circle or a dot.

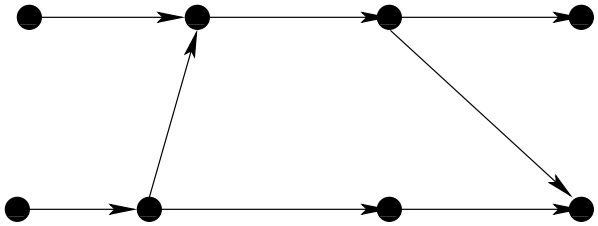


Figure 2.2: Representing computations

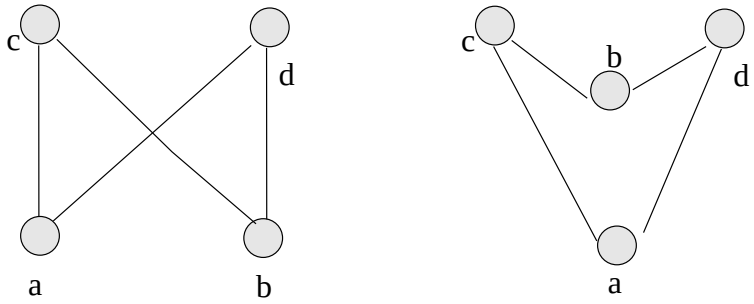


Figure 2.3: Multiple Hasse diagrams of the same poset

2. If x covers y in $\mathcal{C}$ then x is visually to the *right* of y in the diagram.

3. There is a directed arrow from x to y iff y covers x.

Figure 2.1 shows some examples of computational posets. Note that there may be multiple visual representations consistent with the definitions above for both Hasse diagrams and computations.

For example, figures 2.1 and figure 2.1 show different representations of the same poset.

We now define some lattice theoretic concepts.

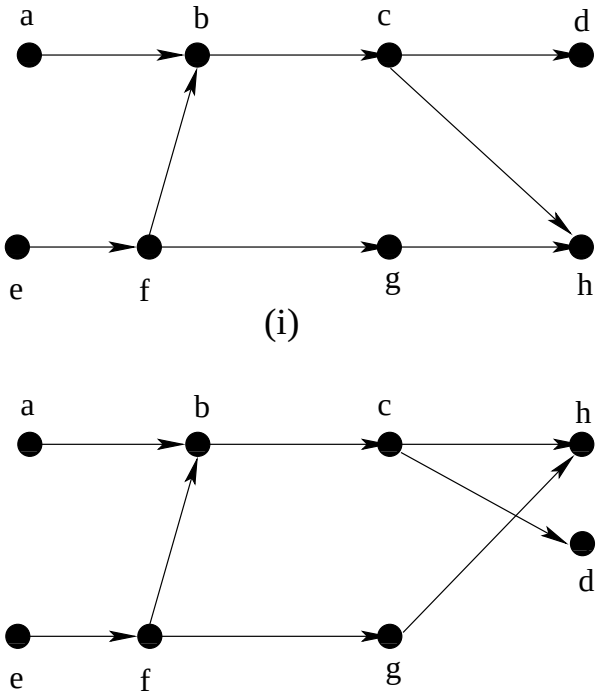


Figure 2.4: Different representations of the same computational poset

2.2 Lattices

First we introduce two operators on the elements of a poset.

Definition 2.2.1. (Join and Meet of two elements) Let $a, b \in \mathcal{C}$ where $\langle \mathcal{C}, \leq \rangle$ is a poset.

For any element $c \in \mathcal{C}$, we say that c is the join of a and b, i.e., $c = a \cup b$ iff

1. $a \leq c$ and $b \leq c$
2. $\forall c' \in \mathcal{C}, (a \leq c' \wedge b \leq c') \Rightarrow c \leq c'$.

The meet of two elements is defined dually. For any $c \in \mathcal{C}$, we say that c is the meet of a and b, i.e., $c = a \cap b$ iff

1. $c \leq a$ and $c \leq b$
2. $\forall c' \in \mathcal{C}, (c' \leq a \wedge c' \leq b) \Rightarrow c' \leq c$.

A lattice is a poset that is closed under meets and joins. Figures 2.2(i), 2.2(ii) and 2.2(iii) are some examples of lattices. In figure 2.2(i) the join of elements c and g is the element i while their meet is b.

Definition 2.2.2. (Lattice) A poset $(\mathcal{C}, \leq)$ is a lattice iff $\forall a, b \in \mathcal{C}$, $a \cup b \in \mathcal{C}$ and $a \cap b \in \mathcal{C}$.

A lattice is distributive if the meet and join operators distribute over each other.

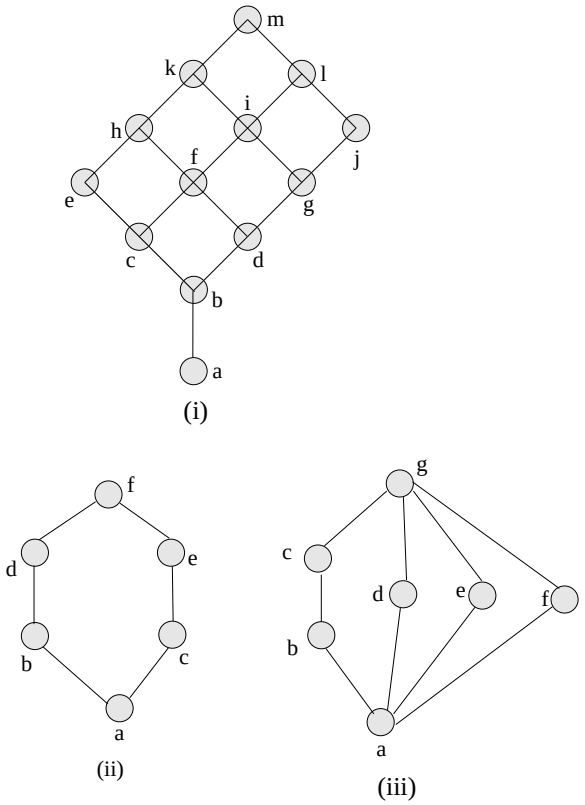


Figure 2.5: Distributive and non-distributive lattices

Definition 2.2.3. (Distributive Lattice) A poset $(\mathcal{C}, \leq)$ is a distributive lattice iff $\forall a, b, c \in \mathcal{C} : a \cup (b \cap c) = (a \cup c) \cap (a \cup c)$

Definition 2.2.4. (Sublattice) Let $\mathcal{C}$ be a lattice and $S \subseteq \mathcal{C}$. S is a sublattice of $\mathcal{C}$ if $a, b \in S$ implies $a \cup b \in S$ and $a \cap b \in S$.

The structure in figure 2.2(i) is an example of a distributive lattice. Figures 2.2(ii) and (iii) are non-distributive lattices.

Elements c, e, i, k, l and m form a sublattice. The elements a, e, g and k on the other hand do not form a sublattice since the meet of e and g (i.e., b) is absent.

Theorem 2.2.1. *[9] A sublattice of a distributive lattice is also distributive.*

Definition 2.2.5. (Ideals and Filters of a Lattice) A sublattice J of a lattice $\mathcal{C}$ is called an ideal if $a \in \mathcal{C}, b \in J$ and $a \leq b \Rightarrow a \in J$.

Dually, a sublattice J of a lattice $\mathcal{C}$ is called a filter if $a \in \mathcal{C}, b \in J$ and $a \geq b \Rightarrow a \in J$.

For example, in figure 2.2(i), the subset $\{a, b, c, d, f\}$ is an ideal of the lattice. Note that the maximal element in the ideal of a lattice, i.e. f in this case, is sufficient to uniquely define and represent the corresponding ideal.

Chapter 3

Predicate Detection

In this chapter we describe our algorithm for predicate detection in polynomial time with respect to the number of processes and events, though it is exponential in the size of the predicate.

3.1 Overview

We examine the problem of detecting nested temporal predicates given the execution trace of a distributed program and present a technique that allows efficient detection of a reasonably large class of predicates which we call the Basic Temporal Logic or BTL predicates. Examples of valid BTL predicates are nested temporal predicates based on local variables with arbitrary negations, disjunctions, conjunctions and the possibly (EF or $\diamondsuit$) and invariant(AG or $\Box$) temporal operators. Our technique is based on the concept of a *basis*, a compact representation of all global cuts which satisfy the predicate. We present an algorithm to compute a basis of a computation given any BTL predicate and prove that its time complexity is polynomial with respect to the number of processes and events in the trace although it is not polynomial in the size of the formula. We do not know of any other technique which detects

a similar class of predicates with a time complexity that is polynomial in the number of processes and events in the system. We have implemented a predicate detection toolkit based on our algorithm that accepts offline traces from any distributed program.

3.2 Related Work

A number of approaches for checking computations using temporal logic are known in the verification and testing community. Temporal Rover [10], MaC [31] and JPaX [24] are some of the available tools. Many of the tools are based on total ordering of events and hence cannot be directly compared to our approach. These tools can miss potential bugs which would be detected by partial order representations. JMPaX [59] is based on a partial order model and supports temporal properties but its time complexity is exponential in the number of processes in the computation.

Another available option to verify computation traces is to use a model checking tool like SPIN [25, 26]. The computation trace needs to be converted to the SPIN input computation and verification takes exponential time in the number of processes.

Computational slicing [43] based approaches can efficiently detect *regular* predicates. POTA [57] is such a partial order based tool which uses computational slicing to detect predicates. POTA guarantees polynomial time complexity only if the predicate can be expressed in a subset of CTL [8] called *Regular CTL* or RCTL [56]. Disjunctions and negations are not allowed in

	SPIN	POTA	BTV
RCTL	exponential in n	polynomial in n	polynomial in n
BTL	exponential in n	exponential in n	polynomial in n

Table 3.1: Time complexities (n = number of processes)

RCTL. If POTA is used with a logic that allows disjunctions or negations (like BTL), it uses a model checking algorithm to explore the reduced state space. Hence the asymptotic time complexity using POTA is exponential in the number of processes when the predicate contains disjunctions. Table 3.1 compares the time complexities of SPIN, POTA and our algorithms implemented in the BTV tool.

3.3 Model and Notation

We assume a loosely coupled, message-passing, asynchronous system model. A distributed program consists of n sequential programs $P_1, P_2, \ldots, P_n$. A computation is a single execution of such a program. A distributed computation ($\langle E, \rightarrow \rangle$) is modeled as a partial order on the set of events E, based on the happened before relation ($\rightarrow$) [33]. The *size of the computation* is the total number of events, $|E|$, in the computation.

Definition 3.3.1. (Consistent Cut) A consistent cut C is a set of events in the computation which satisfies the following property: if an event e is contained in the set C, then all events in the computation that happened before e are contained in C.

$\forall e_1, e_2 \in E : (e_2 \in C) \wedge (e_1 \rightarrow e_2) \Rightarrow e_1 \in C.$

In figure 3.1(i) the set $\{e_1, f_1\}$ is a consistent cut, while $\{e_1, e_2\}$ is not. In the following discussion, we mean 'consistent cut' whenever we simply say 'cut'.

For notational convenience, we simply mention the maximal elements on each process that are elements of the cut to represent that cut. For example, the cut $\{e_1, e_2, f_1, f_2, f_3\}$ is written as $\{e_2, f_3\}$. The set of all consistent cuts in a computation is denoted by $\mathcal{C}$. This set, $\mathcal{C}$, forms a distributive lattice [9] (also called the computational lattice) under the less than equal to relation defined as follows.

Definition 3.3.2. Cut C_1 is less than or equal to cut C_2 if and only if, $C_1 \subseteq C_2$.

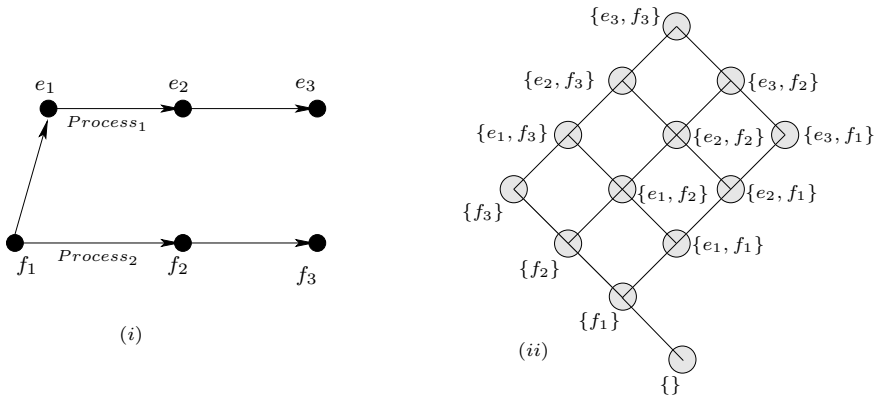


Figure 3.1: A computation and the lattice of its consistent cuts

Figure 3.1(i) depicts a computation. The lattice formed by all consistent cuts of this computation is shown in figure 3.1(ii). Note, the number of

consistent cuts in the computational lattice may be exponential in the number of events and processes in the computation.

A cut C, in a computation E, satisfies a predicate P if the predicate is true in the global state represented by the cut. This is denoted by $(C, E) \models P$ or simply $C \models P$ where the context is clear.

The *join* of two cuts is simply defined as their union, and the *meet* of two cuts corresponds to the intersection of those two cuts.

Birkhoff's representation theorem [9] states that a distributive lattice can be completely characterized by the set of its join irreducible elements. Join irreducibles are elements of the lattice that cannot be expressed as the join of any two elements. Commonly, the bottom element is not considered to be a join irreducible element. However, in this discussion, for notational convenience, we include the bottom element (the initial cut $\{\}$) in the set of join irreducible elements. For example consider figure 3.1 showing a computation and the distributive lattice formed by all the consistent cuts in the computation. In figure 3.1(ii), cuts $\{\}, \{f_1\}, \{f_2\}, \{f_3\}$, $\{e_1, f_1\}$, $\{e_2, f_1\}$, $\{e_3, f_1\}$ are join irreducible. The cut, $\{e_1, f_2\}$ is not join irreducible because it can be expressed as the join of cuts $\{f_2\}$ and $\{e_1, f_1\}$.

The *initial cut* is the least cut, i.e., the empty set $\{\}$ and the *final cut* is the greatest cut, i.e, the set of all events E, in the computational lattice.

3.3.1 Logic Model (BTL)

We now formally define Basic Temporal Logic (BTL), such that any predicate expressible in BTL can be efficiently detected using the algorithm presented later in this chapter. The atomic propositions in BTL are local predicates, i.e., properties that depend on a single process in the computation. Local predicates and their negations are regular predicates. Let AP be the set of all atomic propositions. Given the set of all consistent cuts, $\mathcal{C}$, of a computation, a labeling function $\lambda : \mathcal{C} \rightarrow 2^{AP}$ assigns to each consistent cut, the set of predicates from AP that hold in it. The operators $\wedge$ and $\vee$ represent the boolean conjunction and disjunction operators as usual, $\neg$ represent the negation of a predicate and we define the *possibly* ($\diamondsuit$) temporal operator (called EF in [41]).

Definition 3.3.3. If $\mathcal{C}$ is the set of all consistent cuts of the computation, then $\diamondsuit P$ holds at consistent cut C, if and only if, there exists $C' \in \mathcal{C}$ such that P is true at C' and $C \subseteq C'$.

The formal BTL syntax is given below.

Definition 3.3.4. A predicate in BTL is defined recursively as follows:

1. $\forall l \in AP$, l is a BTL predicate
2. If P and Q are BTL predicates then $P \vee Q$, $P \wedge Q$, $\diamondsuit P$ and $\neg P$ are BTL predicates

We formally define the semantics of BTL.

- $(C, E, \lambda) \models l \Leftrightarrow l \in \lambda(C)$ for an atomic proposition l
- $(C, E, \lambda) \models P \wedge Q \Leftrightarrow C \models P$ and $(C, E, \lambda) \models Q$
- $(C, E, \lambda) \models P \vee Q \Leftrightarrow C \models P$ or $(C, E, \lambda) \models Q$
- $(C, E, \lambda) \models \diamondsuit P \Leftrightarrow \exists C' \in \mathcal{C} : (C \subseteq C'$ and $(C', E, \lambda) \models P)$
- $(C, E, \lambda) \models \neg P \Leftrightarrow \neg((C, E, \lambda) \models P)$

We use $(C, E) \models P$ or simply $C \models P$ in the rest of the discussion when E and λ are obvious from the context. Note that, the AG of a predicate P ($\Box P$) operator in CTL [41] can be written as $\neg\diamondsuit(\neg(P))$ in BTL.

We also define the operator EG recursively as follows:

Definition 3.3.5. $(C, E, \lambda) \models EG(P)$ if $(C, E, \lambda) \models P$ and :

1. C is the top (maximal) element of $\mathcal{C}$ or
2. $\exists C' \in \mathcal{C} : (C'$ covers C and $(C, E, \lambda)' \models EG(P))$

The operator AF on a predicate P is defined as $\neg EG(\neg P)$.

Detecting a predicate in a distributed computation is determining if the initial cut of the computation satisfies the predicate.

3.3.2 Types of Predicates

Definition 3.3.6. (Join-closed, Meet-closed and Regular Predicates) A predicate P is join-closed if all cuts that satisfy the predicate are closed under union.
i.e., $(C_1 \models P \wedge C_2 \models P) \Rightarrow (C_1 \cup C_2) \models P$.

Similarly a predicate P is meet-closed if all the cuts that satisfy the predicate are closed under intersection. A predicate is regular if it is join-closed and meet-closed.

If cuts C_1 and C_2 satisfy a regular predicate, then by definition, $C_1 \cup C_2$ and $C_1 \cap C_2$ also satisfy that predicate. For example, the predicate "No process has the token and the token in not in transit" is regular. All conjunctions of local predicates are regular.

Lemma 3.3.1. *[13] Join-closed predicates are closed under conjunction.*

Similarly,

Lemma 3.3.2. *[13] Meet-closed predicates are closed under conjunction.*

From lemmas 3.3.1 and 3.3.2 we can conclude that

Lemma 3.3.3. *[43] Regular predicates are closed under conjunction.*

A predicate is stable if, once it becomes true, it remains true [4]. A stable predicate is always join-closed.

Definition 3.3.7. A predicate P is stable, if $\forall C_1, C_2 \in \mathcal{C} : C_1 \models P \wedge C_1 \leq C_2 \Rightarrow C_2 \models P$.

Some examples of stable predicates are loss of a token, deadlocks, and termination.

From the semantics of the definition of $\Box$, it follows that:

Lemma 3.3.4. *[8] Given a predicate P, $\Box P$ is a stable predicate.*

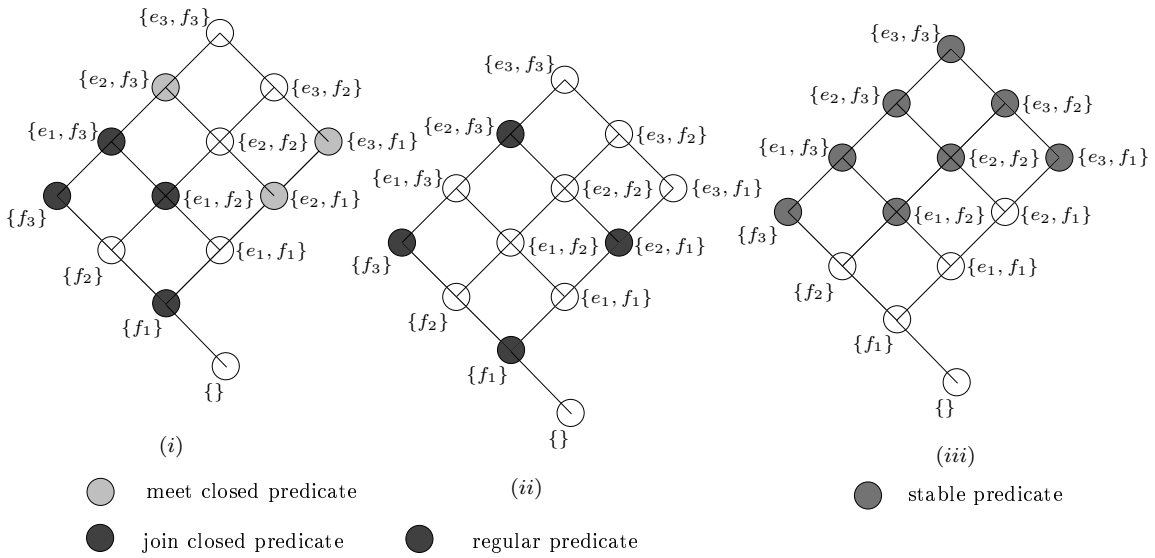


Figure 3.2: Predicates

Figure 3.2 depicts examples of the cuts satisfied by meet-closed, join-closed, regular and stable predicates.

Lemma 3.3.5. *Stable predicates are closed under conjunction, i.e., if P and Q are stable predicates then $P \wedge Q$ is a stable predicate.*

Proof. Given that P and Q are stable, we need to prove that

$\forall C_1, C_2 \in \mathcal{C} : C_1 \models (P \wedge Q) \wedge C_1 \leq C_2 \Rightarrow C_2 \models (P \wedge Q)$

{ from the definition of the $\wedge$ operator }

$\equiv \forall C_1, C_2 \in \mathcal{C} : (C_1 \models P \wedge C_1 \models Q) \wedge C_1 \leq C_2 \Rightarrow (C_2 \models P \wedge C_2 \models Q)$

RHS is true since P and Q are stable predicates.

□

3.4 Basis of a Computation

We now introduce the concept of a *basis* of a computation. Informally, a basis is an exact compact representation of the set of cuts which satisfy the predicate.

Definition 3.4.1. (Basis) Given a computational lattice $\mathcal{C}$, corresponding to a computation E, and a predicate P, a subset $S[P]$ of $\mathcal{C}$ is a basis of P if

1. (Compactness) The size of $S[P]$ is polynomial in the size of computation that generates $\mathcal{C}$.

2. (Efficient Membership) Given any cut (global state) $C \in \mathcal{C}$, there exists a polynomial time algorithm that takes $S[P]$, E and C as inputs and determines if $(C, E) \models P$.

We denote the basis with respect to a predicate P as $S[P]$. Given a

predicate P, a cut C belongs to a basis $S[P]$, if C satisfies that predicate. i.e., $C \in S[P] \Leftrightarrow C \models P$.

Note that direct enumeration of all the states satisfied by a predicate is, in general, not a basis since it is not compact and determining if a cut is a member of that set could take exponential time.

For a simple example of an basis, consider a class of predicates, such that the cuts satisfying a predicate in that class form an ideal in the computational lattice. (An ideal is a sublattice that contains every cut that is less than the maximal cut in the sublattice.) A basis, for such a class of predicates, is just the maximal cut of the ideal. It can be efficiently determined if a cut $C \in \mathcal{C}_p$ by checking if the cut is less than or equal to the maximal cut.

Computational slicing, introduced in [43], is a technique to compute an efficient predicate structure for regular predicates.

Definition 3.4.2. (Slice) The slice $slice[P]$ of a computation with respect to a predicate P is the poset of the join irreducible consistent cuts representing the smallest sublattice that contains all consistent cuts satisfying P.

Though the number of consistent cuts satisfying the predicate may be large, the slice of a predicate can be efficiently represented by the set of the join irreducible cuts in the slice. *Slicing* is the operation of computing the slice for the given predicate.

When the predicate is regular, the computed slice represents exactly those cuts that satisfy the predicate. Given the slice with respect to a predi-

cate, it is possible to efficiently detect if a cut satisfies that predicate. Therefore, a slice is an efficient basis for regular predicates. However, using slicing for predicate detection of non-regular predicates can take exponential time.

In the remainder of this section, we explore a technique to compute a basis for a more general class of predicates, that we call BTL, which can have arbitrary negations, disjunctions, conjunctions and the temporal possibly($\diamondsuit$) operator. Since a BTL predicate can be non-regular, a slice of a BTL predicate is not a valid basis. One naive approach to compute a predicate structure is to maintain a set of slices instead of a single slice. Though this is polynomial in the number of processes n, it results in a large number of slices ($O(n^{2^k})$), where k is the size of the predicate. In this paper, we introduce a *semiregular structure* which can efficiently represent a more general class than regular predicates. A BTL predicate can be represented by using a set of semiregular structures.

We start off by looking at the representation of a stable predicate. Figure 3.3 shows an example of a stable predicate. The set of states satisfying a stable predicate can be considered to be the union of a set of filters of the computational lattice. Thus, a stable predicate can be represented by the set of minimal cuts that satisfy the predicate.

Another representation is to identify a set of ideals, $\mathbb{J} = \{I_1, I_2, \ldots\}$ of the computational lattice such that all the cuts satisfying the stable predicate are contained in the complement of $\bigcup_{I \in \mathbb{J}} I$. The stable predicate in figure 3.3 can be represented by two ideals as seen in the figure. We use the set of ideals

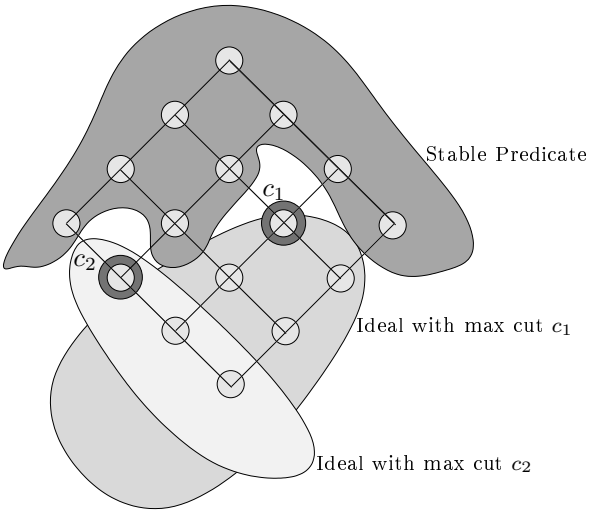


Figure 3.3: Representing stable predicates

representation in this paper for computational efficiency while dealing with BTL predicates.

Definition 3.4.3. (Stable Structure) Given a stable predicate P and the computational lattice $\mathcal{C}$, a stable structure is the set of ideals $\mathcal{I}$ such that a cut satisfies P iff it does not belong to any of the ideals in $\mathcal{I}$. Therefore, $C \models P \Leftrightarrow \neg(C \in \bigcup_{I \in \mathcal{I}} I)$.

A cut C is said to belong to the stable structure if C does *not* belong to $\bigcup_{I \in \mathcal{I}} I$. Note that, any ideal is uniquely and efficiently represented by its maximal cut. In the remainder of this paper we use $\mathcal{I}$ to represent a set of ideals representing the stable predicate and simply $maxCuts$ to denote the set containing the maximal cut from each ideal in $\mathcal{I}$.

Note that, this representation is not a basis since, the set of ideals could

be very large in general. However, we see later, that this leads to an efficient representation when the predicate is expressed in BTL.

3.4.1 Semiregular Predicates and Structures

The conjunction of a stable predicate and a regular predicate is called a semiregular predicate and is more expressive than either of them.

Definition 3.4.4. P is a semiregular predicate if it can be expressed as a conjunction of a regular predicate with a stable predicate.

We now list some properties of semiregular predicates.

1. A regular predicate is semiregular.

   Proof. $true$ is a stable predicate

   $p \wedge true = p$

   Hence any regular predicate p can be expressed as a conjunction of a regular predicate and a stable predicate($true$). □

2. Similarly, any stable predicate is semiregular.

   Proof. $true$ is a regular predicate

   $p \wedge true = p$

   Hence any stable predicate p can be expressed as a conjunction of a regular predicate($true$) and a stable predicate □

3. Semiregular predicates are join-closed.

Proof. Since regular and stable predicates are join-closed, it follows that their conjunction, a semiregular predicate, is also join-closed. □

4. Note that not all join-closed predicates are semiregular. Figure 3.4 shows a join-closed predicate that is not semiregular.

5. Semiregular predicates are closed under conjunction, i.e., if P and Q are semiregular then $P \wedge Q$ is semiregular.

   Proof. Let $P = P_r \wedge P_s$ and $Q = Q_r \wedge Q_s$, where P_r, Q_r are regular and P_s, Q_s are stable.
   $P \wedge Q = (P_r \wedge Q_r) \wedge (P_s \wedge Q_s)$
   From lemma 3.3.3 $(P_r \wedge Q_r)$ is regular and lemma 3.3.5 implies that $(P_s \wedge Q_s)$ is stable. □

6. A semiregular predicate has a unique maximal element.

   Proof. This follows from the property that a semiregular predicate is join-closed. □

7. If P is a semiregular predicate then $\diamondsuit P$ is regular.

   Proof. P is a semiregular predicate then P has a unique maximal element, say C.
   From the definition of $\diamondsuit$, $\diamondsuit P$ is an ideal of the computational lattice. Hence it is regular. □

8. If P is a semiregular predicate then $\Box P$ is semiregular.

Proof. From lemma 3.3.4 we know that $\Box P$ is stable. □

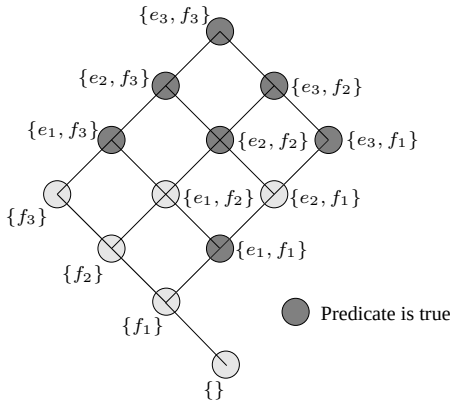


Figure 3.4: A join-closed predicate may not be semiregular

We now present an alternative characterization of a semiregular predicate that offers a different insight into the structure of the cuts satisfying such a predicate.

Lemma 3.4.1. *Predicate P is semiregular iff*

- *P is join-closed, i.e, $C_1 \models P \wedge C_2 \models B \Rightarrow (C_1 \cup C_2) \models P$ and*
- *The meet of two cuts that satisfy P is C, and C does not satisfy P, then any cut smaller than C does not satisfy P. i.e., $(C_1 models P) \wedge (C_2 models P) \Rightarrow (C_1 \cap C_2) \models P \vee (\forall C' \leq (C_1 \cap C_2) : \neg(C' \models P))$.*

Proof. $\Rightarrow$

Let $P = P_r \wedge P_s$ where P_r is a regular predicate and P_s is a stable predicate

- P is join-closed follows from the properties of semiregular predicates.
- $\{(C_1 \models P \wedge C_2 \models P) \Rightarrow (C_1 \cap C_2) \models P_r\}$
  $(C_1 \models P \wedge C_2 \models P \wedge (C_1 \cap C_2) \models \neg P) \Rightarrow (C_1 \cap C_2) \models \neg P_s)$

  {from the definition of stable predicates}
  $\Rightarrow (\forall C' \leq (C_1 \cap C_2) : \neg(C' \models P_s)$

  $\{P = P_s \wedge P_r\} \Rightarrow (\forall C' \leq (C_1 \cap C_2) : \neg(C' \models P))$

$\Leftarrow$ Given P that satisfies the two conditions in the lemma statement, we construct P_r and P_s such that $P = P_r \wedge P_S$.

- Let $\mathcal{C}_{min}$ be the set of minimal cuts that satisfy P. Then P_s is defined as follows: $C \models P_s \Leftrightarrow \exists C' \in \mathcal{C}_{min} : C' \leq C$.
- P_r is the meet closure of P, i.e., $C \models P_r \Leftrightarrow C \models P \vee (\exists C_1, C_2 \models P : C = C_1 \cap C_2)$.

We now show that $C \models P \Leftrightarrow C \models P_r \wedge P_s$.

1. $C \models P$
   $\{P \subseteq P_s \wedge P \subseteq P_r\}$
   $\Rightarrow C \models P_s \wedge P_r$
2. $C \models P_r \wedge P_s \Rightarrow C \models P$:
   We show that $C \models P_s \Rightarrow \exists C' \in \mathcal{C}_{min} : C' \leq C. \equiv C \models P_s \wedge C \models P_r$

{ manipulation and basic predicate calculus }

$\equiv (C \models P) \vee (C \models P_s \wedge C \models P_r \wedge C \models \neg P)$

{ from definition of a stable predicate }

$\equiv (C \models P) \vee (\forall C' \in \mathcal{C}_{min} : C' \leq C \wedge C \models P_r \wedge C \models \neg P)$

{since P_r is constructed by taking the meet-closure of P }

$\equiv (C \models P) \vee (\forall C' \in \mathcal{C}_{min} : C' \leq C \wedge (\exists \mathcal{M} = \{m \models P\} \wedge \cap(\mathcal{M}) = C \wedge C \models \neg P))$

{ the second property in the lemma implies that there can be no element less than C that satisfies P }

$\equiv (C \models P) \vee (\forall C' \in \mathcal{C}_{min} : C' \leq C \wedge \forall C' \in \mathcal{C}_{min} : C' \not\leq C)$

{ second term in the disjunction evaluates to false } $\equiv (C \models P) \vee \texttt{false}$

□

It is also interesting to note that semiregular predicates are closed under the EG operator. More generally, if P is a join closed predicate, $\diamondsuit P, \Box P$ and $EG(P)$ are semiregular predicates.

A few examples of semiregular predicates are listed below.

- All processes are never *red* concurrently at any future state and process 0 has the token. That is $P = \neg\Diamond(\bigwedge red_i) \wedge token_0$.
- At least one process is beyond phase k (stable) and all the processes are red.

We now define a representation for semiregular predicates.

Definition 3.4.5. (Semiregular Structure) A semiregular structure, g, is represented as a tuple $(\langle slice, \mathcal{I} \rangle)$ consisting of a slice and a stable structure, such that the predicate is true in exactly those cuts that belong to the intersection of the slice and the stable structure.

Hence $C \in g \Leftrightarrow (C \in slice) \wedge \neg(C \in \bigcup_{I \in \mathcal{I}} I)$.

Note that, a cut is contained in a semiregular structure if it belongs to the slice *and* the stable structure in the semiregular structure. The maximal cut in a semiregular structure is the maximal cut in the slice if the semiregular structure is nonempty.

We see later that any BTL predicate can be expressed as a basis consisting of a union of semiregular structures. A semiregular structure enables us to easily handle predicates of the form $\neg\Diamond P$. Such a predicate can be represented by n slices or by a single stable structure or a semiregular structure. We use this in our algorithms and prove that it is possible to compute an efficient basis representation for any BTL predicate.

3.4.2 Algorithm

We present an algorithm to compute a basis for any predicate expressed in BTL. The computed basis consists of a set of semiregular structures such that a cut belongs to the basis if it belongs to any semiregular structure in that set.

Definition 3.4.6. Given a BTL predicate P, we define a representation S of the predicate that consists of a set of semiregular structures such that $C \models P \Leftrightarrow (\exists g \in S : C \in g)$.

We assume that the input predicate has negations pushed in to the local predicates or the $\Diamond$ operators. In the following discussion, we often treat $\neg\Diamond$ as single operator. We see later that our algorithm returns an efficient predicate structure which allows polynomial time detection of the predicate.

Each semiregular structure, g, is represented as a tuple $\langle slice, maxCuts \rangle$ where $g.slice$ is the slice in g and $g.maxCuts$ is the set of cuts corresponding to the ideals representing the stable structure. The use of ideals instead of filters is very important and results in the 2^k bound (see theorem 3.6.2) on the size of the stable structure. (The stable structures calculated by the algorithm could require n^k filters to represent it.)

Figure 3.5 outlines the main algorithm to compute a basis of the computation for any BTL predicate. For predicate detection, we simply check if the initial cut of the computation is contained in the computed basis. To determine if a cut is contained within the basis, we need to examine if it belongs

/*The input predicate P_{in} has all negations pushed
- inside to the $\Diamond$ operator or to the atomic propositions */
/* each semiregular structure is represented as a tuple $\langle slice, maxCuts \rangle$
- where $maxCuts$ is the set of maximal cuts
- of the ideals $\mathcal{I}$ representing the stable structure */

function `getBasis`(Predicate P_{in})
output: $S[P_{in}]$, a set of semiregular structures
  Case 1. (Base case: local predicates) : $P_{in} = l$ or $P_{in} = \neg l$
    $S[P_{in}] := \{\langle slice(P), \{\}\rangle\}$
  Case 2. $P_{in} = P \vee Q$
    $S[P] :=$ `getBasis`$(P); S[Q] =$ `getBasis`(Q);
    $S[P_{in}] := \{S[P] \cup S[Q]\}$;
  Case 3. $P_{in} = P \wedge Q$
    $S[P] :=$ `getBasis`$(P); S[Q] =$ `getBasis`(Q);
    $S[P_{in}] := \bigcup_{g_p \in S[P], g_q \in S[Q]} \{(\langle g_p.slice \wedge g_q.slice, g_p.maxCuts \cup g_q.maxCuts \rangle)\}$;
  Case 4. $P_{in} = \Diamond P$
    $S[P] :=$ `getBasis`(P);
    $S[P_{in}] := \bigcup_{g \in S[P]} \{\langle \Diamond(g.slice), \{\}\rangle\}$;
  Case 5. $P_{in} = \neg\Diamond P$
    $S[P] :=$ `getBasis`(P);
    /* $slice_{orig}$ is the original computation */
    $S[P_{in}] := \{\langle slice_{orig}, \cup_{g \in S[P]} \{$`maxCutIn`$(g.slice)\}\rangle\}$;
  Remove all empty semiregular structures from $S[P_{in}]$;
return $S[P_{in}]$

Figure 3.5: Computing a basis

to any semiregular structure in the basis. A basis is nonempty if the predicate is true in any consistent cut of the computation. Note that, in case we need to check whether a predicate P is true at any cut in the computation (and not just the initial cut), we can either apply our algorithm on the predicate $\Diamond P$ or alternatively apply the algorithm on P and check if the returned basis is nonempty.

The algorithm computes the basis by recursively processing the predicate inside out.

- The base case is a local predicate. Note that, the negation of a local predicate is also local. We know that for each atomic proposition l_i, $slice[l_i]$ can be computed in polynomial time. Efficient algorithms to compute $slice[l_i]$ (or $slice[\neg l_i]$) when the atomic propositions are local predicates, can be found in [43]. The basis of a local predicate has a single semiregular structure that consists of a slice and an empty set of ideals. (A local predicate and its negation are regular predicates and hence a slice is an efficient basis for such predicates).

- The second case handles disjunctions. If the input predicate P_{in} is of the form $P \vee Q$ the basis is the structure containing all the cuts in $S[P]$ and $S[Q]$ and is obtained by computing the union of the sets $S[P]$ and $S[Q]$.

- When the input predicate is of the form $P \wedge Q$, the resultant basis is the pairwise intersection of each semiregular structure in $S[P]$ and $S[Q]$. Each semiregular structure consists of a slice and a stable structure.

The intersection of two semiregular structures, say g_p and g_q, is the tuple $\langle g_p.slice \cap g_q.slice, g_p.stable\_structure \cap g_q.stable\_structure \rangle$. The grafting algorithm described in [43] describes a technique to compute the intersection of two slices. Since we use ideals to represent stable structures, the intersection of the stable structures is represented by the *union* of the sets $g_p.maxCuts$ and $g_q.maxCuts$.

- The fourth case in the algorithm handles predicates of the form $P_{in} = \diamondsuit P$. $S[P]$ is the union of a set of semiregular structures. The resultant basis is obtained by computing $\diamondsuit g$ for each g in $S[P]$ and taking the union. Note that $\diamondsuit g$ is equivalent to $\diamondsuit(g.slice)$ and the algorithm for EF of a regular predicate in [56] can be used to determine $\diamondsuit(g.slice)$.

- Since $\neg\diamondsuit P$ is stable, the basis corresponding to $\neg\diamondsuit P$ contains a single semiregular structure g. The slice in this semiregular structure is the original computation while the ideals are represented by the maximal cuts of the slice in each of the semiregular structures that belong to $S[P]$. In this case, it becomes clear that using the 'set of ideals representation' for stable structures is more efficient. The number of ideals is guaranteed to be k if $S[P]$ had k semiregular structures. Using another representation like maintaining a set of filters would have resulted in expensive operations since the number of filters could be n^k in this case.

After each step, the algorithm checks if any of the semiregular structures are empty and discards the empty semiregular structures. A semiregular structure

is empty, if the maximal element of the slice is less than or equal to each cut in $g.maxCuts$.

It can be seen that the structure returned by our algorithm contains exactly those cuts which satisfy the input predicate. We show in section 3.6 that the number of semiregular structures and the number of ideals required to represent the stable structures returned by our algorithm is polynomial in n. This enables us to check whether a cut belongs to the structure in polynomial time and hence the structure is efficient. We now illustrate the basic idea of our algorithm with an example.

3.5 An Example

Figure 3.6(i) shows a poset representing a computation and figure 3.6(ii) shows the corresponding computational lattice. The states where a predicate is true is shown by an area enclosing the states. Figure 3.6(ii) shows the states satisfied by the local predicates p_a, p_b and p_c respectively. The steps involved in detecting the predicate $\neg\Diamond(p_a \vee p_b) \wedge \Diamond p_c$ are:

1. $S[p_a \vee p_b]$: The predicate structure corresponding to $p_a \vee p_b$ is given by $S[p_a] \cup S[p_b]$. Since p_a and p_b are local predicates according to the algorithm the basis for p_a is $\{\langle slice[p_a], \{\}\rangle\}$ and p_b is $\{\langle slice[p_b], \{\}\rangle\}$. Hence as seen in figure 3.6(iii), $S[p_a \vee p_b]$ is
   $\{\langle slice[p_a], \{\}\rangle, \langle slice[p_b], \{\}\rangle\}$

2. $S[\neg\Diamond(p_a \vee p_b)]$: According to step 5 of the algorithm, the basis contains

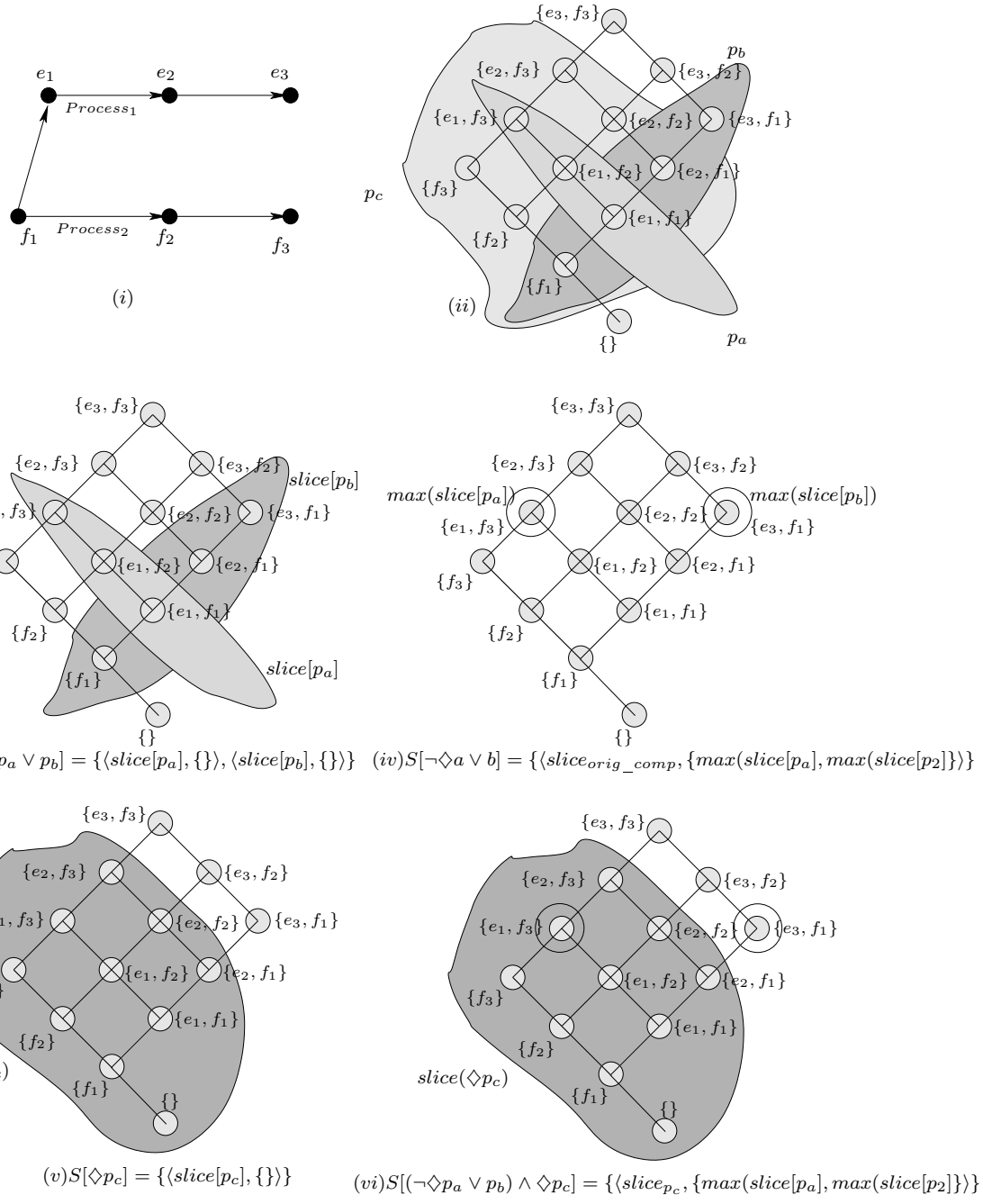


Figure 3.6: Computing a Basis

a single semiregular structure. This semiregular structure is a tuple with a slice representing the entire computation and the set of maximal cuts of each semiregular structure in $p_a \vee p_b$. As seen in the figure, it is given by

$\{\langle slice[orig\_computation], \{max(slice[p_a]), max(slice[p_b])\}\rangle\}$.

3. $S[\Diamond p_c]$: Since p_c is a local predicate, the basis $S[p_c]$ has a single semiregular structure that contains no ideals and just the slice corresponding to p_c. $S[\Diamond p_c]$ is a structure that contains all cuts that are smaller than the maximal cut in $S[p_c]$. The area shaded in figure 3.6(v) shows all the cuts that are contained in $S[\Diamond p_c]$.

4. $S[\neg\Diamond(p_a \vee p_b) \wedge \Diamond p_c]$: The basis corresponding to the conjunction on two predicates is given by the pairwise intersection of the semiregular structures in the bases corresponding to the predicates. In this case $S[\neg\Diamond(p_a \vee p_b)]$ and $\Diamond p_c$ contain exactly one semiregular structure each so the final answer also has one semiregular structure. The intersection of the slices $slice[original\_computation]$ and $slice[\Diamond p_c]$ is simply the $slice[\Diamond p_c]$ as shown in figure 3.6(vi). The set of ideals is the union of the ideals in each semiregular structure and in this case is $\{max(slice[p_a]), max(slice[p_b])\}$.

The final output of the algorithm is the basis:

$\{\langle slice[\Diamond p_c], \{max(slice[p_a]), max(slice[p_b])\}\rangle\}$

The initial cut is not contained within this basis and hence the predicate detection algorithm returns false as expected. Since the basis is not empty, it is easy to conclude that the predicate is true *somewhere* in the computation (albeit not at the initial state).

3.6 Complexity Analysis

The time taken by the algorithm in figure 3.5 depends on the number of ideals representing the stable structure in each semiregular structure and the total number of semiregular structures in the resultant basis (the size of the basis). We first present a result on the bound on the size of computed basis.

Theorem 3.6.1. *The basis $S[P]$ computed by the algorithm in Figure 3.5 for a BTL predicate P with k operators has at most 2^k semiregular structures.*

Proof. Induction on k:

- (Case: $S[P] = s[l]$) $|S|$ is always less than or equal to one in this case. This is the base case ($k = 1$).
- (Case: $S[P] = S_1 \vee S_2$) Let k_1, k_2 be the number of operators corresponding to S_1 and S_2 respectively. $|S| = |S_1| + |S_2| \leq 2^{k_1} + 2^{k_2} \leq 2^k$ (since $k = k_1 + k_2 + 1$)
- (Case: $S[P] = S_1 \wedge S_2$) $|S| = |S_1|.|S_2| \leq 2^{k_1}.2^{k_2} \leq 2^k$ (since $k = k_1 + k_2 + 1$)

- (Case: $S[P] = \Diamond S_1$) $|S| = |S_1| \leq 2^k$ (since $k = k_1 + 1$)
- (Case: $S[P] = \neg\Diamond S_1$) $|S| = 1$

□

This leads to the following theorem.

Theorem 3.6.2. *The total number of ideals $|I|$ in the basis computed by the algorithm in Figure 3.5 for a BTL predicate P is at most 2^k.*

Proof. We prove this by induction on k.

- (Case: $|I| = 0$) This is the base case ($k = 1$).
- (Case: $S[P] = S_1 \vee S_2$) Let k_1, k_2 be the number of operators and $|I_1|$, $|I_2|$ be the number of ideals in S_1 and S_2 respectively. Then, $|I| = |I_1| + |I_2| \leq 2^{k_1} + 2^{k_2} \leq 2^k$ (since $k = k_1 + k_2 + 1$)
- (Case: $S[P] = S_1 \wedge S_2$) Let the bases S_1 and S_2 have $|S_1|$ and $|S_2|$ semiregular structures respectively. Since the output basis is computed by the cross product of the constituent semiregular structures, each ideal in S_1 repeats $|S_2|$ times in the output while each ideal in S_2 appears $|S_1|$ times in the output. Hence the total number of ideals I is $|S_2|.|I_1| + |S_1|.|I_2|$. From Theorem 3.6.1, we know that $|S_1| \leq 2^{k_1}$ and $|S_2| \leq 2^{k_2}$. Also from the induction hypothesis, $|I_1| \leq 2^{k_1}$ and $|I_2| \leq 2^{k_2}$. Therefore, $|I| \leq 2^{k_2+k_1} + 2^{k_1+k_2} = 2^{k_1+k_2+1} = 2^k$ as required (since $k = k_1 + k_2 + 1$).

- (Case: $S[P] = \Diamond S_1$) $|I| = 0 \leq 2^k$
- (Case: $S[P] = \neg\Diamond S_1$) $|I| = |S_1| = 2^{k_1} \leq 2^k$

□

The time required to compute the conjunction of two slices with respect to $\wedge$ is $O(|E|n)$ [43]. It takes $O(|E|n)$ time to compute the slice with respect to the $\Diamond$ operator.

Theorem 3.6.3. *The time complexity of the algorithm in figure 3.5 is polynomial in the number of events ($|E|$) and the number of processes (n) in the computation.*

Proof. The algorithm simplifies the predicate by computing the basis one operator at a time. Hence, if there are k operators in all, it requires k steps to compute the basis for the entire predicate.

Theorem 3.6.1 states that after the l^{th} operator is processed at most 2^l new semiregular structures are generated. The generation of each semiregular structure takes less than or equal to $|E|n$ time. The time required to generate all the semiregular structures is $2^l.|E|n$.

The algorithm compares each ideal to the maximal cut of a slice to check if the semiregular structure is empty. There are at most 2^l semiregular structures (theorem 3.6.1) which implies that there are no more than 2^l slices (since each semiregular structure contains exactly one slice). The total number

of ideals is less than or equal to 2^l (theorem 3.6.2). Since comparing two cuts requires $O(n)$ time, it takes $(2^l + 2^l)n$ time to check which semiregular structures are empty. Hence the time required to process the l^{th} operator is $2^l.(|E|n) + n(2^{l+1})$, i.e, $2^{l+1}.n.(2|E| + 1))$

Therefore the total time required is $\Sigma_{l=1}^{n} 2^{l+1}.n.(2|E| + 1) = O(2^k|E|n)$.

□

If the input predicate is in a 'DNF-like' form then predicate detection is even more efficient (polynomial in k).

Theorem 3.6.4. *If the input predicate has conjunctions only over regular predicates, then the size of the predicate structure and the total number of ideals $|I|$, is at most k.*

Since conjunctions are allowed over regular predicates, the resulting predicate is regular and can be represented by exactly one semiregular predicate with no ideals.

3.7 Implementation

We have implemented a toolkit to verify computation traces generated by distributed programs. This toolkit accepts offline execution traces as its input.

We used a Java implementation of the distributed dining philosophers algorithm from [22] and checked for errors in the system. We injected faults in

the traces and verified the traces using, both, our toolkit and POTA [57]. Note that, for predicates containing disjunctions, POTA reduces the computation size and uses SPIN [26] to check for predicate violations. The POTA-SPIN combination performs well when predicate is regular but it runs out of memory for non-regular predicates when the number of processes is increased, especially when configured to list all predicate violations. BTV, as expected, scales well and we could use it to verify computations with large number of processes.

Note that the toolkit relies on offline traces and hence it is not necessary for the program that is being tested to be implemented in Java. It can be used with any arbitrary distributed program that outputs a compatible trace. The toolkit includes a utility to convert traces from the POTA trace format.

3.8 Remarks

We see that it is possible to efficiently detect nested temporal predicates containing disjunctions and negations (along with conjunctions and $\diamondsuit$). The notion of a semiregular structure allows us to efficiently compute an efficient basis given any BTL predicate. This has many practical applications which require verification of traces. Apart from ensuring the validity of runs, the technique discussed in this paper is also useful in distributed program debuggers. Since the computed basis contains exactly all states where the predicate holds, it can be used to pinpoint the faults in the program.

Chapter 4

Fusible Data Structures for Fault-Tolerance

In this chapter, we introduce the concept of *fusible data structures* to maintain fault-tolerant data in distributed programs.

4.1 Introduction

In distributed systems, it is often necessary for individual servers to recover from faults. One of the important aspects in recovery is to ensure that the dynamic data that was being used by the program, can be restored. We look at the problem of maintaining fault-tolerant data structures in such systems. One commonly used technique is replicating the server data [12, 34, 48, 53, 61, 65, 70]. Hence, to tolerate a single fault, the space requirements are doubled. Although it requires a considerable amount of extra space, this approach is efficient at run time since updating the back up data structures is easy and efficient. Another approach is to use erasure coding approaches [7, 47, 50]. Though such approaches require less space than replication, their main drawback is that they are data-agnostic. This results in high communication overheard and expensive updates of the backup data.

We propose a new idea called fusible data structures with the aim to

have an approach that combines the desirable properties of both these approaches. Given a fusible data structure, it is possible to combine a set of data structures into a single fused structure that is smaller than the combined size of the original structures. When any of the original data structures is updated, the fused structure can be updated incrementally using local information about the update and does not need to be entirely recomputed. In case of a failure, the fused structure, along with the correct original data structures, can be used to efficiently reconstruct the failed structure.

Fusible data structures satisfy three main properties: recovery, space constraint and efficient maintenance. The recovery property ensures that in case of a failure, the fused structure, along with the remaining original data structures, can be used to reconstruct the failed structure. The space constraint ensures that the number of nodes in the fused structures is strictly smaller than the total number of nodes in the original structures. Finally, the efficient maintenance property ensures that when any of the original data structures is updated, the fused structure can be updated incrementally using local information about the update and does not need to be entirely recomputed.

We show that many commonly used data structures like arrays, hash tables, stacks and queues are fusible and present efficient algorithms to fuse such structures. This approach often requires significantly less space than conventional backups by replication and still allows efficient operations on the regular data structures.

As a concrete example, we consider a lock server in a distributed system that maintains and coordinates use of a lock. The lock server maintains the record of the process that has the lock and the queue of all pending requests. Assume that the size of the pending request queue is n_{max}. Traditionally, if fault-tolerance from a crash is required, we would keep two copies of the queue. If there are k such lock servers in the system, and each one is replicated, we get the space overhead of kn_{max}. In our proposal, we keep a single backup queue for all k servers that is obtained by fusing the original queues. Our fused queue uses $O(n_{max})$ space, supports recovery and can be updated efficiently when any of the primary queues gets updated. This technique results in k-fold savings.

We have experimentally evaluated our technique by implementing a library supporting arrays, queues, stacks, sets, priority queues, linked lists and hash tables in a distributed programming framework. A brief summary of our experimental results is given in table 4.1. The space requirements are averaged over multiple runs. For example, the space required by active replication of 50 queues is around 40 times more than the space required by fusion. For systems with k servers, queues, arrays, stacks and sets require around $k/2$ times less space than replication. The improvement is not so drastic for linked lists and priority queues, but the space requirements are still reduced by a factor of two or more when k is large.

We focus primarily on single crash failures. The ideas presented here can be easily generalized to the case when there can be t concurrent failures

Number of processes	10	20	30	40	50
Queues	8.6	16.4	23.9	31.7	38.7
Stacks	9.6	18.3	26.7	34.8	43.7
Priority Queues	1.3	1.6	1.8	2.0	2.2
Sets	6.5	12.3	18.5	24.3	30.5
Linked Lists	1.5	1.8	2.1	2.4	2.6

Table 4.1: Experimental results: (space used by replication)/(space used by fusion)

by using erasure codes (like Reed-Solomon codes [38]) as described in section 4.6.

In summary,we make the following contributions:

- We introduce the notion of fusible data structures that provide fault-tolerance with reduced space overhead as compared to replication based schemes that are widely used now.

- We present algorithms to efficiently fuse standard data structures including arrays, stacks, queues, sets, priority queues and hash tables.

- We have a Java implementation of fusible data structures which can be transparently used in distributed programs. Since the overhead is minimal, this can also be used to easily increase reliability in programs which inherently do not have any fault tolerance. For example, instead of using the standard queue, the programmer can easily use the queue from our implementation library and get fault tolerance without significant programming effort or computational overhead.

- We apply our technique to an extensively studied problem of lock server in distributed systems. Our experimental results show k-fold improvement in space complexity for maintaining k lock servers in a fault-tolerant manner.

The remainder of this chapter is organized as follows: we first define and explain the idea of fusible data structures. We then present algorithms to fuse commonly used data structures including arrays, stacks, queues, lists and tables. Section 4.5 discusses the performance of fusible techniques in practical applications. We then compare existing approaches using erasure codes or replication with the fusion approach.

4.2 Fusible Data Structures

A data structure is a tuple consisting of a set of nodes and some *auxiliary information*. The auxiliary information may be implicit or explicit and delineates the 'structure' of the data nodes. For example, every element of an array is a node and the auxiliary information that is implicit in the definition of the array specifies that the nodes are contiguous and in a specific order. On the other hand, a linked list has explicit auxiliary data consisting of the next and previous node pointers for each node along with the head and tail pointers of the list.

Every data structure has a set of valid operations associated with it. For example, `push` and `pop` are the valid operations on a stack. The data

structures may also have read-only operations, but we ignore them here since they do not affect the backup data structure.

Given k instances of a data structure, our objective is to efficiently maintain a more efficient backup of these structures than full replication based schemes. If a failure occurs, it should be possible to reconstruct any structure on the failed process using the backup and the remaining structures. We assume that failures are restricted to crash failures. The recovered data structure needs to have the exact values of the nodes and *equivalent* auxiliary data. For example, the actual values of the pointers in a linked list will not be preserved and the reconstruction will simply be a list with identical values in the same order.

We define a *fusible* data structure X as follows.

Definition 4.2.1. Let $x_1, \ldots, x_k$ be instances of a data structure X where $k > 1$ and each node in X has size s. Assume that each of $x_1, \ldots, x_k$ contain $n_1, \ldots, n_k$ nodes respectively and let $N = \Sigma_{i=1}^{k} n_i$. Then X is fusible if there exists a data structure Y with an instance y such that:

1. **(Recovery)** Given any $k-1$ of $x_1, \ldots, x_k$ and y, there exists an algorithm to recreate a missing x_i.

2. **(Space Constraint)** The number of nodes in y is strictly less than N. The size of any node in Y is $O(k + s)$ and the space required for any additional data maintained by Y is independent of N.

3. **(Decentralized Maintenance)** For any operation that updates one of the x_i, there exists an operation to update y using information only from y and x_i. The time required for the operation on y is of the same order as the operation of x with respect to the number of nodes in x_i and y.

The data structure Y is called a *fusion* of X. We refer to the process of computing y as *fusing* the structures $x_1, \ldots, x_k$ and to the instance y as the fused structure or fusion.

The *recovery* property is the crucial property required for fault-tolerance. Whenever one of the x_i is unavailable (for example, due to a process failure), it should be possible to recreate x_i with the remaining objects and y. When the server that stores y crashes, then our requirement is slightly weaker. The recovered object may even have different structure; the only requirement is that it be a valid fusion of x_i's.

The *space constraint* property rules out trivial algorithms based on simple replication. Note that, simple replication satisfies recovery property; we can easily recover from one fault, but it does not satisfy the space constraint. On the other hand, erasure coding is data structure oblivious and though it results in space savings, it does not allow efficient update of data.

The *decentralized maintenance* property ensures that as any object x_i changes, there is a way to update y without involving objects other than x_i and y. The cost of maintenance is one of the main metrics for comparison with standard erasure coding methods. For example, assume that we need to

maintain a fusion of linked lists. Consider the strategy of maintaining y as simple xor of bit arrays corresponding to the memory pages that contain x_i's. Any change in x_i, say inserting a new node, requires the re-computation of y with time complexity that may depend on the size of x_i and y. With fusible data structures, we exploit properties of the data structure so that the cost of updating y is of the same order as of the cost of updating x_i.

An obvious fusible data structure is a single bit. Assume that the operations defined on the bit are: `get` and `set`, where `get` returns the value of the bit and `set` sets the bit to the provided value. Let $x_1, \ldots, x_k$ be k bits. The fused data structure is also a bit initialized to xor of all x_i. Whenever any operation $set(b)$ is issued on x_i, y is updated as :

$$y := y \otimes x_i \otimes b$$

The fused bit satisfies our recovery property because the missing bit can be obtained by xoring the remaining bits. It satisfies the space reduction property because the sum of sizes of all objects is k bits, whereas the fused bit uses a single bit. Finally, the maintenance of y requires values from x_i and y only.

The single bit example can be easily generalized to any data structure with a fixed number of bits like char, int, float etc.

In the previous example, the `set` operation requires information from x_i to update y. This is not always necessary and we could have data structures with operations that do not require any input from the original structure to update the fused structure.

Definition 4.2.2. (Independent operation) An operation is independent with respect to a fusible data structure X, if its corresponding operation on the fusion Y does not require any information from X

If all the operations on a data structure are independent the structure is said to be independent.

The fused bit data structure described previously is not independent because the `set` operation requires the value of x_i to update its fusion y. Now consider the single bit data structure permitting the operations `get` and `toggle` with usual semantics. In this case, the maintenance operation for y is even simpler. Whenever any x_i is toggled, y is toggled as well. Thus no information from X is required to update Y. Hence this data structure is *independent*.

One more example of a fusible data structure is a counter with n bits that takes values from 0 to $2^n - 1$. The operations on the data structure are `set`, `increment` and `decrement`. Assume that both increment and decrement operations are modulo 2^n. Given k counters one can keep the fused counter as *xor* of all the primary counters. This way the space overhead is n bits (instead of kn bits required by replication). However, none of the operations are independent. When a primary counter is updated, updating the fused counter requires the previous value of the primary counter. A better fusion method for the fused counter is to keep the modulo 2^n sum of the k primary counters . Given any $k - 1$ primary counters and the fused counter that has

sum modulo 2^n, one can easily derive the value of the missing counter. Again, we have the same space overhead of n bits. However, now the `increment` and the `decrement` operations can be performed on the fused data structure *independently*. The `set` operation is not independent. Note that, when the operations in the counter are simply `read` and `set`, then keeping the sum would not have any advantage over xor. The efficiency of a fusible data structure crucially depends upon the operations.

In the next few sections we explore how standard data structures such as arrays, stacks, queues and linked lists can be fused.

4.3 Array Based Data Structures

We first extend our example of a single bit to another simple fusible data structure: a bit array of size n. Assume that the following operations are allowed on an object of this class: *get(j)* that returns j^{th} bit and $set(j, b)$ that sets j^{th} bit to b. We maintain a fusion of k such arrays $x_1, \ldots, x_k$ by keeping another bit array y. The i^{th} bit of y is computed by xoring the i^{th} bits from each of $x_1, \ldots, x_k$. This takes care of the efficient update property. If the sizes of the arrays $x_1, \ldots, x_k$ are different then we take the largest of them as the size of y and pad the smaller arrays with zeros. It follows that,

Lemma 4.3.1. *The bit array data structure is fusible.*

Proof. In a bit array structure each bit constitutes a *node*. We maintain a fusion of k such arrays $x_1, \ldots, x_k$ by keeping another bit array y. The i^{th} bit

of y is computed by xoring the i^{th} bits from each of $x_1, \ldots, x_k$. This takes care of the efficient update property. If the sizes of the arrays $x_1, \ldots, x_k$ are different then we take the largest of them as the size of y and pad the smaller arrays with zeros. To recover any bit array, it is sufficient to compute xor of the remaining bit arrays with y. The space constraint is satisfied since the number of nodes in y is $max(n_1, \ldots, n_k)$ and this is always less than N.

□

When the source arrays are of different sizes, we also need to maintain the sizes of the arrays in the fused structures, so that in case of a failure, the correctly sized array is recovered. It is not necessary to store k distinct sizes in the fused structure and a single value that is the xor of these values is stored.

We now generalize the bit array example.

Theorem 4.3.2. *If a data structure is fusible, then any fixed size array of that data structure is also fusible.*

Proof. We consider each element of the array to be a node. A valid fusion is another identical array, where the i^{th} element is a fusion of all the i^{th} nodes from each of the arrays. This fusion exists since it is given that the original data structure is fusible.

□

Thus, arrays of basic data types like short, int, float, double are fusible as well.

4.3.1 Array Based Stacks and Queues

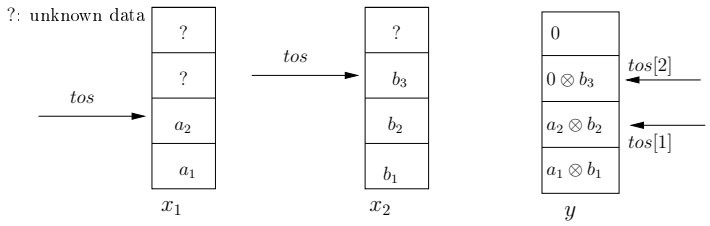


Figure 4.1: Stacks

In the rest of this chapter we will use the $\otimes$ operator to denote fusing nodes in data structures. The actual nodes could be complex fusible objects and in such cases instead of bitwise xoring, we mean computing the fusion of those nodes. We use $\bar{\otimes}$ to denote the recovery operation,i.e., the inverse of the fusion. In case of xor both the fusion and recovery operators are identical. For fusion by modulo n addition, the recovery operator would be modulo n subtraction.

We now consider data structures that encapsulate some additional data besides the array and support different operations. The array based stack data structure maintains an array of data, an index `tos` pointing to the element in the array representing the top of the stack and the usual `push` and `pop` operations.

Lemma 4.3.3. *The array based stack data structure is fusible given $O(k)$ additional storage. The* `push` *operation is independent.*

Proof. For simplicity, the algorithms we discuss in the remainder of this chapter do not include boundary checking for normal data structures unless it adds

to the discussion. We assume that all stacks are initially empty. The fused stack consists of the fusion of the arrays from the source stacks. We keep all the stack pointers at y individually. This additional $O(k)$ additional storage is required for the efficient maintenance property. The following push and pop operations satisfy the efficient maintenance property.

```
function x_i.push(newItem)
   x_i.array[x_i.tos] := newItem;
   x_i.tos++;
   y.push(i,newItem);
end function

function y.push(i, newItem)
   y.array[y.tos[i]] := y.array[y.tos[i]] ⊗ newItem;
   y.tos[i]++;
end function

function x_i.pop()
   x.tos[i] −−;
   y.pop(i, x_i.array[x_i.tos]);
   return x_i.array[x_i.tos];
end function

function y.pop(i, oldItem)
   y.tos[i] −−;
   y.array[y.tos[i]] := y.array[y.tos[i]] ⊗̄ oldItem;
end function
```

Figure 4.2: Fusion of Stacks

When an element is pushed onto one of the source stacks, x_i, the source stack is updated as usual and the request is forwarded to the fused stack. The fused stack does not require any additional information from x_i, i.e., the `push` operation is independent. During a `pop` operation, we xor the corresponding

```
function y.recover(failedProcess)
/*Assume all source stacks have the same size*/
  recoveredArray := new Array[y.array.size];
  for j = 0 to tos[failedProcess] − 1
    recItem := y[j];
    foreach process p ≠ failedProcess
      if (j < tos[p]) recItem := recItem ⊗̄ x_p.array[j];
    recoveredArray[j] := recItem;
  return ⟨ recoveredArray, tos[failedProcess] ⟩ ;
end function
```

Figure 4.3: Recovery from faults

value in y with the value that would be returned by $x_i.pop()$.

The number of elements, n_y, in the array corresponding to the fused stack is the maximum of $n_1, \ldots, n_k$ which is less than N. Therefore, the space constraint is satisfied.

From the algorithm, it is obvious that any stack $x_{failedProc}$ can be recovered by simply xoring the corresponding elements of the other original stacks with the fused stack.

□

Note that, in the usual implementation of stacks, it is not required that the popped entry be zeroed. However, for fusible data structures it is essential to clear the element during a `pop` operation to ensure that the next `push` operates correctly.

It is easy to accommodate different sized stacks in the fused data struc-

ture. In this case, similar to the arrays example in lemma 4.3.3, the fusion y contains an array that is as large as the biggest stack.

Circular array based queues can be implemented similarly by zeroing out deleted elements and keeping the individual head and tail pointers.

4.4 Dynamic Data Structures: Stacks, Queues, Linked Lists

So far, we have discussed data structures based on arrays. We now move on to structures like stacks, queues and sets that are based on linked lists. Instead of using a generic fusion algorithm for all structures based on linked lists, we use specific properties of each structure to make the fusion more efficient wherever possible.

4.4.1 Stacks

We start with the linked list based stacks, i.e., a linked list which supports inserts and deletes at only one end, say the tail. The fused stack is basically another linked list based stack that contains k tail pointers, one for each contributing stack x_i.

When an element $newItem$ is pushed onto stack x_i, then

- if $tail[i]$ is the last element of the fused stack, i.e, $tail[i].next = null$, a new element is inserted at the end of the fused queue and $tail[i]$ is updated.

- otherwise, $newItem$ is xored with $tail[i].next$ and $tail[i]$ is set to $tail[i].next$.

```
function y.push(i, newItem)
  if(tail[i].next = null)
       tail[i].next := new_empty_node();
  tail[i] = tail[i].next;
  tail[i].value := tail[i].value ⊗ newItem;
end function

function y.pop(i, oldItem)
  tail[i].value := tail[i].value ⊗̄ oldItem;
  oldTail := tail[i];
  tail[i] := tail[i].previous;
  if((oldTail.next = null) ∧(∀ j: tail[j] ≠ oldTail))
       delete_from_list(oldTail);
end function
```

Figure 4.4: List based stacks

When a node is popped from a stack x_i, the value of that node is read from x_i and passed on to the fused stack. In the fused stack, the node pointed to by $tail[i]$ is xored with the old value. If $tail[i]$ is the last node in the fused list and no other $tail[j]$ points to $tail[i]$, then the node corresponding to $tail[i]$ can be safely deleted once the value of $tail[i]$ is updated. Note that in this case, a push takes $O(1)$ time but a pop operation may require $O(k)$ time, since we check if any other tail points to the node being deleted. This satisfies the efficient maintenance property of fusible structures since the time required is independent of the size of the total number of nodes in the original data structure. For constant time pop operations, the algorithm for fused queues in section 4.4.2.1 can be applied for stacks.

The fusion of the list based stack requires no more nodes than the maximum number of nodes in any of the source stacks. The size of each node in the fused stack is the same as s, the size of the nodes in the original stack X. The only extra space overhead is the k tail pointers maintained. If all the stacks are approximately of the same size, the space required is k times less than the space required by replication.

4.4.2 Queues

The fusion of list based queues is more involved than stacks since both ends of the list are dynamic. We assume that elements in a queue are inserted at the tail of any queue and only the head of a queue can be deleted.

We begin with examining why an algorithm similar to the list based stacks cannot be used for queues. If we modify the algorithm so that it applies to queues, we get a structure that seems to be a fused queue but does not always satisfy the space constraint since it could have exactly N nodes in the worst case. An example of when this could happen is shown in figure 4.5.

To ensure that it meets the space constraint, we need to merge nodes $head[i]$ and $head[i].previous$ if possible after $head[i]$ is deleted in the fused structure. Determining if the nodes can be merged in the data structure described above can take $O(N)$ time, violating the efficient update property.

We now present an algorithm for fusing queues that satisfies the space constraint and the efficient update property by maintaining an extra $log(k)$ bits at each node.

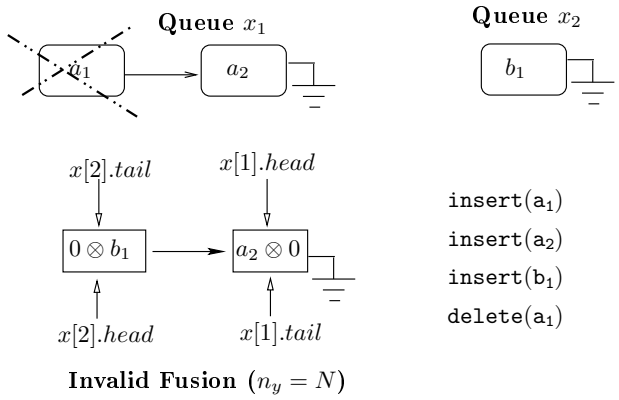


Figure 4.5: The resultant data structure is not a valid fusion

4.4.2.1 Fused Queues

As in the previous algorithm, the fused data structure is implemented as a linked list with each node in the list containing the *xor* of elements from the original queues. Each node in the fused structure also contains an extra variable, the *reference count* which is a count of the number of source queues whose data is maintained by that node. The reference count enables us to decide when a node in the fused structure can be safely deleted (when its reference count is 0) or merged. The fused data structure Y contains a list of head and tail pointers for each component queue, pointing to the corresponding node in fused list. For example, in figure 4.6(ii), $x[1].tail$ has a reference count of 1 since it contains a value from only x_1 while $x[1].head$ contains the fusion of values from x_1 and x_2 and hence has a count of 2.

As in the case of stacks, deleting a value from the fused structure requires access to the old value that is to be deleted. The old value is xored

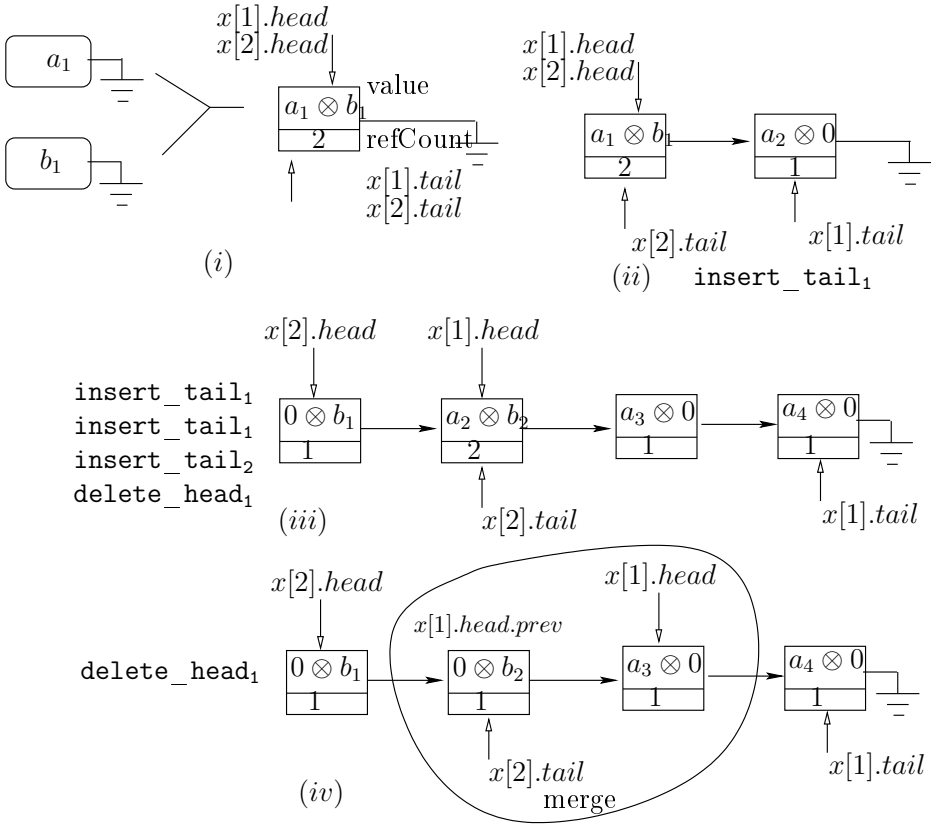


Figure 4.6: Example of a fused queue

```
function y.insertTail(i, newItem)
  if(tail[i].next = null)
      tail[i].next := new_empty_node();
  tail[i] = tail[i].next;
  tail[i].value := tail[i].value ⊗ newItem;
  tail[i].refCount ++ ;
end function

function y.deleteHead(i, oldItem)
  oldHead := head[i];
  oldHead.value := oldHead.value ⊗̄ oldItem;
  oldHead.referenceCount −−;
  head[i] := oldHead.next;
  if(oldHead.refCount = 0) delete_from_list(oldHead);
  else if (head[i].refCount = 1 ∧
      head[i].previous.refCount = 1)
      merge(head[i], head[i].previous);
end function
```

Figure 4.7: Queues with Reference Counts

with the value at $head[i]$, $head[i]$ is set to $head[i].next$ and the reference count is decremented by one. If the new reference count is 0 then the old node at $head[i]$ is deleted. The reference count also simplifies the merging. The new $head[i]$ and $head[i].previous$ are merged when both their reference counts are equal to one.

1. Figure 4.6(i) shows a fusion of two queues, x_1 and x_2 containing one element each. In this case, the fused structure contains exactly one node containing the xor of a_1 and b_1. The pointers $x[1].head$, $x[1].tail$, $x[2].head$ and $x[2].tail$ point to this node and the reference count of this

node is two.

2. The insertion of a new element is similar to pushing an element in the stack. Figure 4.6(ii) shows the fused structure after an element is inserted in queue x_1. A new node is created in the fused linked list and $x[i].tail$ is updated so that it points to this node. The reference count of the new node is one.

3. Figure 4.6(iii) shows the fused queue after a sequence of two inserts and a delete on queue x_1 and an insert in queue x_2.

4. Now if the head of queue x_1 is deleted, the resulting fused queue is shown in figure 4.6(iv). In this case, the reference count of both $x[1].head$ and $x[1].head.previous$ is one. Hence each of these nodes contains data from a distinct source queue and the nodes are merged together forming a single node by xoring the values and setting the reference count to two. Note that $x[2].tail$ and $x[1].head$ both point to the same newly merged node.

Lemma 4.4.1. *The number of nodes in the fused queue is less than N.*

Proof. Since we delete empty nodes the maximum number of nodes in the fused queue is bounded by N. N nodes in the fused queue implies that there are adjacent nodes with a reference count of one containing data from different processes (for example the scenario in figure 4.5). However, this can never happen with the algorithm just described since adjacent nodes are always

merged after a delete whenever each of them contains elements from a different process.

□

Hence the space constraint is satisfied. Note that, unlike the list based stack algorithm discussed earlier, this algorithm requires $O(1)$ time for insertions as well as deletions. It requires an extra $log(k)$ bits at every node of the fused list.

4.4.3 Dequeues

Dequeues are a generalization of list based stacks and queues. The reference count based fusion implementation of linked lists is easily extensible to a dequeue allowing two new operations `insert_head` and `delete_tail`.

4.4.4 Efficient Fused Queues Using an Auxiliary List

It is possible to further modify the algorithm described in the previous section so that it is not necessary to maintain $log(k)$ extra bits at each node (i.e., $Nlog(k)$ bits in all).

The main idea is based on the observation that the reference count changes only at nodes that are pointed to by a head or a tail pointer. Hence, instead of maintaining a reference count at each node in the fused list, we maintain the count only for those nodes pointed to by the heads and the tails. We do this by adding another level of indirection by maintaining an auxiliary

```
function y.insertTail(i, newItem)
  if(tail[i].pointer.next = null)
       tail[i].pointer.next := new_empty_node();
  newNode = tail[i].pointer.next;
  newNode.value := newNode.value ⊗ newItem;
  /* check if we need to
   create new node in the auxiliary list */
  if(tail[i].pointer.next != tail[i].next.pointer)
       newTail := new_node_in_aux_list_after(tail[i]) ;
       newTail.referenceCount = tail[i].referenceCount;
       tail[i].pointer := newNode;
       newTail.auxCount := 1;
  tail[i].auxCount −−;
  if(head[i].auxCount = 0)
       delete_node_in_auxiliary_list(tail[i]);
  tail[i] := newTail;
  tail[i].refCount ++;
end function
```

Figure 4.8: Algorithm for insertTail

linked list that contains the reference count and a pointer to the node in the actual fused list.

Hence, in the fused queue, the head and the tail pointers point to an auxiliary node which in turn points to the primary node containing the data. From our algorithm we can see that auxiliary queue does not contain more that $2k$ nodes at any time. Each node in the auxiliary queue has two counter fields, one which maintains a reference count similar to that of the algorithm in figure 4.7 and another which counts the number of head an tail pointers pointing to that node in the auxiliary queue.

```
function deleteHead(i, oldItem)
  oldHead.pointer.value := oldHead.pointer.value ⊗̄ oldItem;
  /* check if we need to
   create new node in the auxiliary list */
  if(oldHead.pointer.next != oldHead.next.pointer)
    newHead = new_node_in_auxiliary_list_after(head[i]) ;
    newHead.pointer := head[i].pointer.next;
    newHead.refCount := head[i].refCount;
    newHead.auxCount = 1;
  head[i].auxCount −−;
  head[i].referenceCount −−;
  if(head[i].refCount = 0)
    delete(head[i].primaryNode);
  if(head[i].auxCount = 0)
    delete_node_in_auxiliary_list(head[i]);
  head[i] = newHead;
  if [(head[i].referenceCount = 1) ∧
    (head[i].previous.referenceCount = 1)]
    merge_aux_and_primary_lists(head[i] , head[i].previous);
end function
```

Figure 4.9: Algorithm for deleteHead

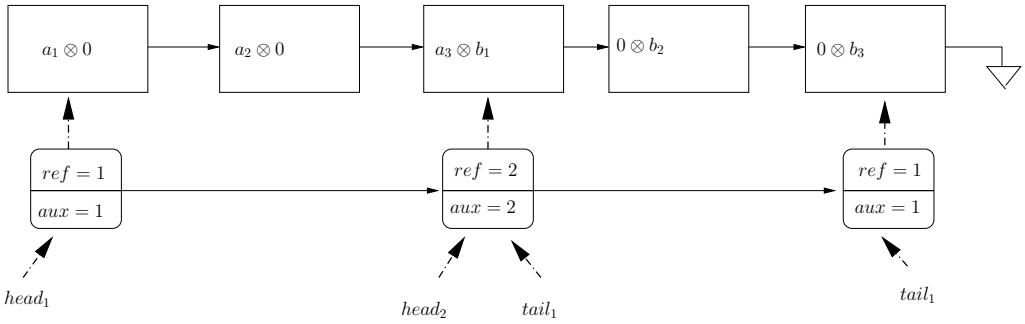


Figure 4.10: Fused queues with auxiliary list

To insert a new element at the tail of a process i, the value of $tail_i$ is used to obtain a node in the auxiliary list. As seen in figure 4.10, this node contains a pointer to the actual node containing the data, along with a reference counter and an auxiliary counter. The reference count is similar to the reference count described earlier, except for that fact that we maintain it in the few nodes of the auxiliary list. The auxiliary counter maintains a count of the number of head or tail pointers pointing to that specific auxiliary node. This enables us to delete unnecessary nodes in the auxiliary list. Using this we can easily show that the maximum size of the auxiliary list is bounded by $2.k$.

While deleting the heads, a similar double dereferencing is carried out. Also if the reference count of an auxiliary node drops to zero, the corresponding data node can be safely deleted. To check if adjacent data nodes can be merged, we check the reference count of the neighbors in the auxiliary list after every delete. Note that, if the adjacent data nodes are merged, we also merge the corresponding nodes in the auxiliary list.

The pseudo code for insertion and deletion of elements in the queue is outlined in figures 4.8 and 4.9 respectively.

4.4.5 Linked lists

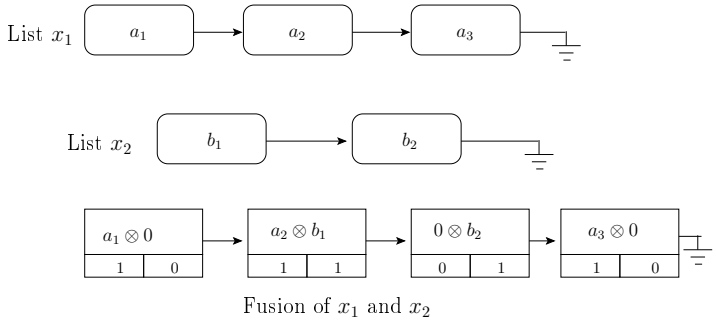


Figure 4.11: Linked Lists

We now examine a generic linked list. The fused structure for a linked list is also valid for priority queues and sets. Linked lists can be fused together in a manner similar to queues, except that we also maintain a bit array of size k, at each node of the fused list, with a bit for every source linked list. Each bit in the array enables us to determine if a node contains data from the list corresponding to that bit. It also enables us to determine if adjacent nodes can be merged.

It is not necessary to maintain a list of head or tail pointers corresponding to each source list in this case. However in priority queues or sets, the head or tail pointers can optionally be maintained to eliminate the search overhead for deletes and inserts.

```
structure nodeInFusedList
  pointers next, prev
  boolean bitField[1 ... numLists]
  value
end structure
```

To insert or delete an element, the source list x_i sends the element along with its index in the source list. The algorithm examines the bit field of each fused node, counting the number of nodes which have data from x_i to determines the location for the insert or delete. To insert a node after say the m^{th} element of the source list, the algorithm advances through the fused list till it encounters m elements with their i^{th} value in the bitfield set to nonempty. If the next node does not have an empty space for source list i, a new node with the input value is created.

4.4.5.1 Performance Considerations

Though the number of nodes in a fused linked list is guaranteed to be less than the number of nodes in the source lists, the fused structure has an overhead of k bits at each node. Hence in the adversarial case, the actual space required by the fusion algorithm could be more than the space required by a replication scheme.

If N_y is the number of nodes in the fused structure the space required by the fused list is $N_y(s+k)$. Fusion is advantageous when $(s+k).N_y < s.N$, i.e., $N/N_y > (s+k)/s$. Note that N_y is always less than N. When $s >> k$, $N/N_y > (s+k)/s$ is always satisfied and the fused structure outperforms replication

```
function y.insert(i, index, newItem)
  node = y.head;
  while(count ≤ index)
       if(node.bitField[i]) count ++;
       node := node.next;
  if (node.next = null ∧ node.next.bitField[i] = notEmpty)
            node := insert_new_node_after(node);
       else node = node.next;
  node.value := node.value ⊗ newItem;
  node.bitField[i] := notEmpty; /* notEmpty = 1 */
end function

function y.delete(i, index, oldItem)
  node = y.head
  while (count < index)
       if(node.bitField[i]) count ++ ;
       node := node.next;
  node.value := node.value ⊗̄ oldItem;
  node.bitField[i] := empty; /*empty = 0*/
  mergeNodes(node);
end function
```

Figure 4.12: Linked Lists

based techniques. If we assume that s and k are approximately equal then the N/N_y needs to be greater than two. Note that the actual value of N/N_y may vary from run to run. In our simulations, N/N_y was always around two or more when k was greater than ten. Thus it is reasonable to assume that the fused linked list outperforms replication unless s is significantly less than k, that is, when we have very small nodes in the linked list and a large number of source lists.

Sets are special cases of linked lists in which the order of nodes does not matter. Hence we always insert new nodes at one end of the list. In our simulations, this drastically improved the performance of the fused structure and the ratio N_y/N always hovered close to the theoretical maximal value k. Therefore in practice, our fusion algorithm can always be applied to sets. Priority queues, which allow insertions at any locations and deletions at one end, perform similar to normal linked lists and the results for the general linked list are applicable to priority queues too.

4.4.6 Hash Tables

A chained hash table can be constructed using a fixed sized array of the sets. Sets are a special case of linked lists and a the fusion of a set can be computed as described in subsection 4.4.5. Such a table is fusible and the fusion consists an array of fused linked lists from the individual hash tables.

Lemma 4.4.2. *Hash tables are fusible.*

Proof. Sets are a special case of linked lists and linked lists are fusible. A hash table is a fixed size array of such lists. Therefore, from theorem 4.3.2, it follows that hash tables are fusible. □

4.5 Experimental Evaluation

For some structures like stacks and simple arrays, it is clear from the algorithms that fusion always performs much better than replication. However

for queues and other structures, in the worst case, only one node may be fused, resulting in space requirements that are almost equivalent to active replication. In the optimal scenario, however, the number of nodes required by the fused structure is k times smaller than active replication.

We have implemented a library of common data structures for regular use based on the distributed programming framework used in [14]. Using this, we examined the performance of the data structures in different scenarios. We also implemented a k-server lock based distributed application.

4.5.0.1 Fault-Tolerant Lock Based Application

We now look at a distributed computing application that uses queues using fusion for backup. It consists of k lock servers that arbitrate access to shared resources. Each lock server maintains a queue of pending requests from clients. We use a single fused queue to backup the queues in all k servers. Every time a server modifies its queue the changes are propagated to the fused queue which is updated as described before.

We modified the lock server program [14] by simply substituting fusible queues instead of the normal queues. The backup space required by k servers was drastically lower, by a factor of more than $k/2$, as compared to active replication.

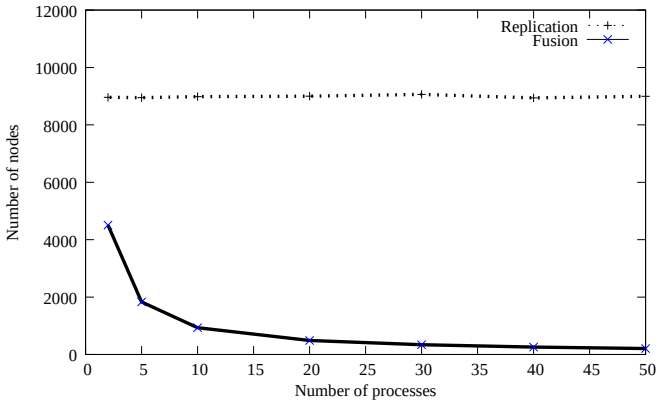


Figure 4.13: Stacks

4.5.0.2 Simulation Results

As seen in the lock server scenario, queues perform well in almost all scenarios resulting in a savings factor of $k/2$ or more. We also tested queues and other data structures by simulating random inserts and deletes and averaging the results over multiple runs. To examine the performance when the data structures are large, we biased the simulations to favor inserts over deletes.

Like queues, sets and stacks showed noticeable and consistent savings of around $k/2$. General linked lists and priority queues, however, do not show consistently significant savings, requiring around half the number of nodes as compared to replication. Stacks, queues and sets show a huge improvement over replication. Table 4.1 lists the space saving due to fusion for these data structures. Figure 4.13 compares the number of nodes in fused structures (the total space required by all the processes in the system) with the number of

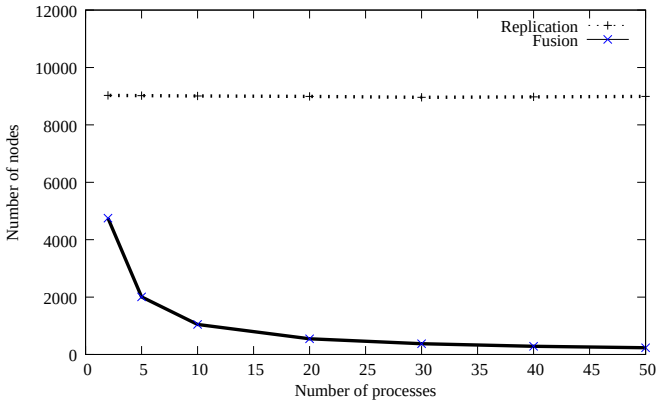


Figure 4.14: Queues

nodes required by replication as the number of processes is varied. Note that the total number of nodes across all the processes was kept constant and the average number of nodes per process decreased, as the number of processes increased.

It may seem that fusible data structures are not useful when the application maintains a single queue, list, or a table. However, this is not true. Assume that we have to maintain a single list L with n nodes in a distributed system with three processors such that failure of any single processor is recoverable. A replication based scheme will maintain the primary copy of the list L on one processor and its mirror on the other processor resulting in space overhead of n nodes. Another possibility based on fusible data structures is to split the list L into two lists L_1 and L_2 maintained by different processors and store the fusion of these two lists on the third processor. This scheme may result in space savings by a factor of two.

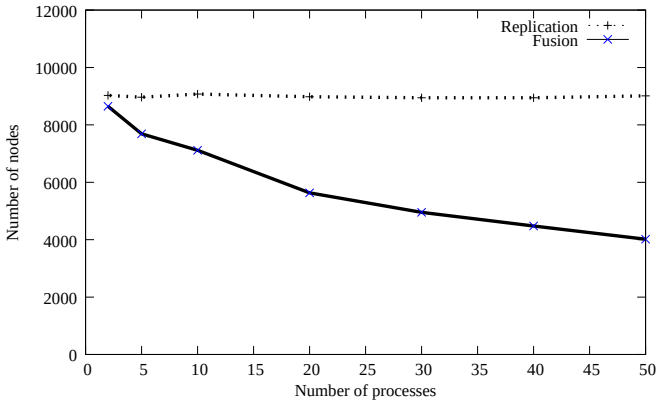


Figure 4.15: Priority Queues

4.6 Fusion Operators and Tolerating Multiple Faults

So far we have used xor as the principal operator for fusing key parts of the data structures. However, some different techniques may be more suitable for different applications. For example, consider an application that maintains counts of various events in the system. The operations on the data structure are $increment(j)$, $reset(j)$ and $decrement(j)$. Given k count tables, one may use xor technique for fusion. However, in that case none of the operations are independent. Another way to fuse data in this case is as follows. Assume that each counter uses n bits, i.e., it stores a natural number between 0 and $2^n - 1$. We keep as the fused data, sum of jth entry of all the count tables modulo 2^n. It is clear that any missing counter table can be recreated from the available tables and the fused data. Moreover, both increment and decrement operations can be performed on the fused data structure efficiently and independently. Hence modulo-addition and modulo-subtraction can be used as fusion and

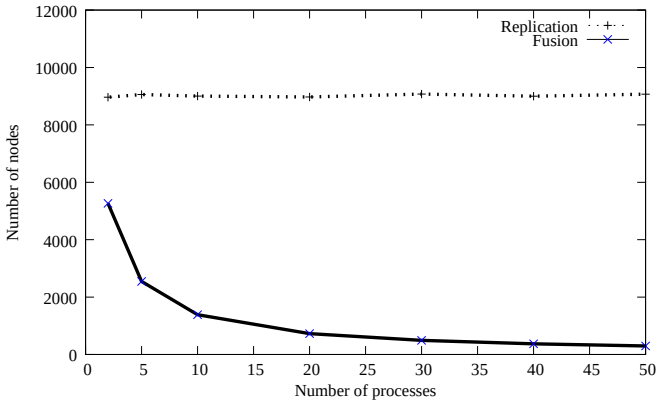


Figure 4.16: Sets

inverse fusion operators instead of xor.

In general it is possible to have more complex fusion operators without affecting most of the previous discussion and algorithms. Mainly the idea of a fusible data structure can be loosely divided into the following parts.

1. Distinguishing between structural information and node data in a data structure.

2. Identifying and maintaining a fused structure that allows us to save storage space by fusing together nodes.

3. Techniques to fuse individual node data (throughout this paper we have used bitwise xoring or modulo addition for this).

4. Updating the backup structure efficiently when operations on the source structures modify some data.

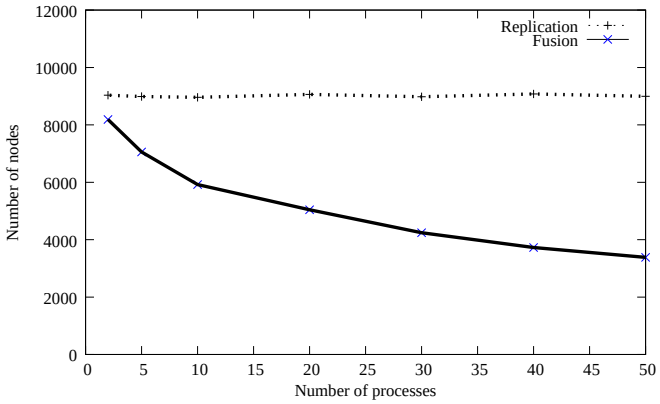


Figure 4.17: Linked Lists

It can be seen that error correcting codes deal exclusively with the third item on the list, .i.e., fusing a set of data blocks into one or more backup blocks allowing recovery in case of failures. We have used xoring which is a parity based erasure coding, assuming only one failure. Our approach is directly extensible to the case of t failures using t backup structures by using other erasure coding techniques (like Reed-Solomon codes [38]) to fuse each node together. The algorithms presented in this paper deal with steps 1, 2 and 4 and remain unchanged. Thus, fusion is orthogonal to error coding and advances in error codes can be transparently included in the fusion techniques. This is particularly interesting as it enables us to extend the fault tolerance capability to more multiple faults without changing any of the algorithms described before. We give an overview of Reed-Solomon codes since these are suited for fusion when we need to tolerate multiple faults.

4.6.1 Reed-Solomon Coding [50]

RS coding employs a combination of an Identity and Singular matrix and also performs all operations over a Galois Field.

Fusing (Encoding) : The basic algorithm comprises of a Vandermonde matrix, that is a combination of an Identity and Singular matrix as shown below. Let S be a Vandermonde matrix. Then we construct a matrix A of the form, $A = \begin{bmatrix} I \\ S \end{bmatrix}$ Let D be the data matrix and E the encoded matrix obtained after multiplying A with D.

$A \times D = E$ i.e.,

$$\begin{bmatrix} I \\ S \end{bmatrix} \times \begin{bmatrix} D \end{bmatrix} = \begin{bmatrix} D \\ C \end{bmatrix}$$

where C is the set of check sums (the fused data) computed for the data set D.

Recovery (Decoding) : In order to recover the data set D, we need to get n of the $(n + k)$ rows of matrix E. Then,

$$A' \times D = E' \Rightarrow (A')^{-1} \times E' = D$$

Thus the data can be recovered in its entirety.

4.7 Remarks

It may seem that fusible data structures are not useful when the application maintains a single queue, list, or a table. However, this is not true. Assume that we have to maintain a single list L with n nodes in a distributed

system with three processors such that failure of any single processor is recoverable. A replication based scheme will maintain the primary copy of the list L on one processor and its mirror on the other processor resulting in space overhead of n nodes. Another possibility based on fusible data structures is to split the list L into two lists L_1 and L_2 maintained by different processors and store the fusion of these two lists on the third processor. This scheme may result in space savings by a factor of two.

We have seen that the algorithms for fusing data structures, result in better space utilization than replication. However, recovery of faulty processes is much more efficient with replication. Hence, when failures are frequent, using replication might be preferable to fusion techniques. Replication is easily extensible for byzantine faults by increasing the number of backups [36, 54]. The fusion techniques discussed in this paper, in contrast, are not directly applicable for byzantine errors. Also, the operations on the some of the fused data structures, require more time than replication. For example, linked list operations are slower by a factor of k.

Chapter 5

Faults in Finite State Machines

We now discuss the problem of tolerating faults in deterministic finite state machines.

5.1 Overview

Given a set of n different deterministic finite state machines (DFSMs) modeling a distributed system, we examine the problem of tolerating f crash faults in such a system. The traditional approach to this problem involves replication and requires $n.f$ backup DFSMs. For example, to tolerate two faults in three machines a replication based technique needs two copies of each of the given machines, resulting in a system with six backup machines. We question the optimality of such an approach and present a generic approach called (f, m)-fusion that allows for more efficient backups. Given n different DFSMs, it is possible to tolerate f faults using m additional machines ($m \leq n.f$). We introduce the theory of fusion machines and provide an efficient algorithm to generate the minimum set of backup machines required to tolerate f faults in a given set of machines.

5.2 Related Work

The work presented in chapter 4 introduces the idea of fusible data structures. It was 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 chapter effectively presents an algorithm to compute a fusion operation given a set of specific input machines.

Extensive work has been done [27, 29] 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.

Another area of research which is conceptually similar to our approach is erasure coding [60]. An erasure code transforms a message of n data blocks into a message with more than n blocks, such that the original message can be recovered from a subset of those blocks. An (f, m)-fusion could be thought of as an input dependent erasure code corresponding to given set of machines. The central idea for generating these machines is analogous to the concept of

the minimum Hamming distance in an erasure code [21]. However, an erasure code based approach would have required sending the state of each machine in the system to the backup. The state space, is in general, very large and transferring state information would result in a huge communication overload, making the idea impractical for all but very small machines. The fundamental difference in the fusion based approach introduced here is that the backup state machines are designed to act on the same sequence of inputs as the original machines without requiring any transfer of state during fault-free operation.

5.3 Model and Notation

We now discuss the system model, followed by the notation used in the remainder of this chapter.

The system under consideration consists of deterministic finite state machines (DFSMs) satisfying the following conditions:

- Every DFSM in the system has a current state associated with it. The current state depends on the sequence of inputs received from the environment.
- The system model assumes fail-stop failures [52]. A failure in any of the DFSMs results in the loss of the current state but the underlying DFSM remains intact. We assume that this failure can be detected and the goal of the fault tolerant system is to determine the current state of the failed machines.

- The DFSMs execute independently with no shared state or communication between them during a fault-free run.
- The DFSMs act concurrently on the same set of events. If some event e is not applicable for a certain DFSM, we assume that e is ignored by that DFSM. 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.

Definition 5.3.1. (DFSM) A DFSM, denoted by A, is a quadruple, (X, Σ, α, a^0), where,

- X is the finite set of states corresponding to A.
- Σ is the finite set of events common to all the DFSMs in the system.
- $\alpha : X \times \Sigma \rightarrow X$, is the transition function corresponding to A. If the current state of A is s, and an event $\sigma \in \Sigma$ is applied on it, the next state can be uniquely determined as $\alpha(s, \sigma)$.
- a^0 is the initial state corresponding to A.

In the remainder of this discussion we interchangeably use the terms DFSM or state machine or machine.

A state, $s \in X$, is *reachable* iff there exists a sequence of events, which, when applied on the initial state a^0, takes the machine to state s. This is denoted by $s = \alpha^k(a^0)$, where α^k denotes a sequence of k operations, $\alpha^1, \ldots, \alpha^k$

applied to the initial state a^0. Our model assumes that all the states corresponding to the machines are reachable.

The *size* of a machine A, is the number of states in X, and is denoted by $|A|$. We now define *homomorphism* [19] that gives us a relation that partially orders the set of all DFSMs.

Definition 5.3.2. (Homomorphism) A *homomorphism* from a machine $A(X_A, \Sigma, \alpha_A, a^0)$ onto a machine B $(X_B, \Sigma, \alpha_B, b^0)$, is the mapping, $\psi : X_A \rightarrow X_B$, satisfying the following relationship:

- $\psi(a^0) = b^0$
- $\forall s \in X_A, \forall \sigma \in \Sigma,\ \psi(\alpha_A(s, \sigma)) = \alpha_B(\psi(s), \sigma)$

If such a homomorphism, ψ, exists from X_A onto X_B, B is said to be homomorphic to A and we denote it as $B \leq A$ and say B is less than or equal to A. Given that machine A is in state a, we can identify the state of machine B as $\psi(a)$.

The mapping, ψ, is called an *isomorphsim* if it is both one-one and onto. In this case, B is said to be isomorphic to A and vice-versa. We denote it as $B = A$.

Consider the two machines, $\mathcal{R}(\{A, B\})(X_r, \Sigma, \alpha_r, r^0)$ and $M_1(X_1, \Sigma, \alpha_1, m_1^0)$ shown in figure 5.1(iii) and figure 5.1(iv) respectively. Let us define a mapping, $\psi : X_r \rightarrow X_1$, such that, $\psi(r^0) = \psi(r^2) = m_1^0$, $\psi(r^1) = m_1^1$ and $\psi(r^3) = m_1^2$. For $s = r^0$, $\sigma = 0$,

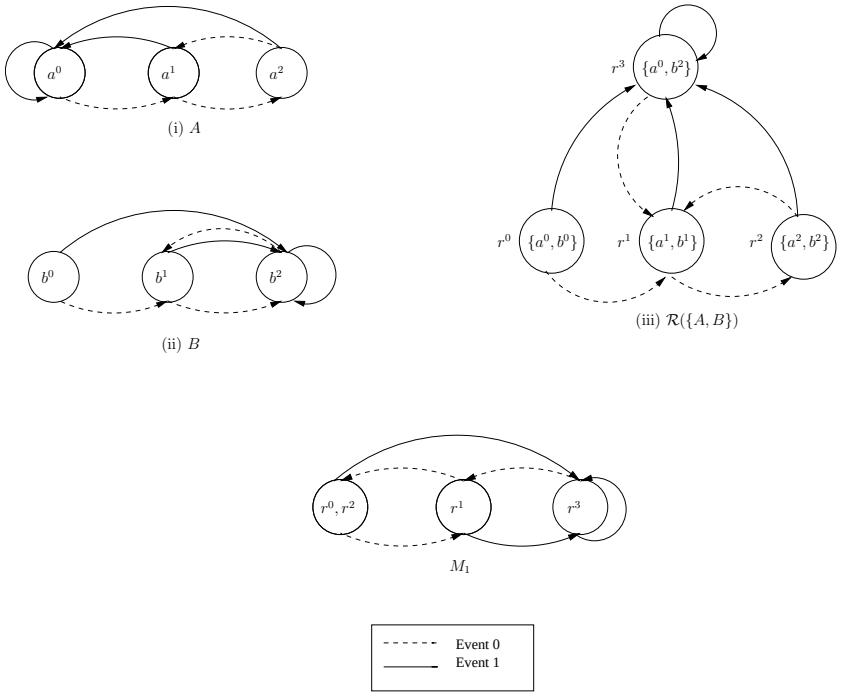


Figure 5.1: DFSMs, Homomorphism and Reachable cross product

$$\psi(\alpha_r(r^0, 0)) = \psi(r^1) = m_1^1 \text{ and } \alpha_1(\psi(r^0), 0) = \alpha_1(m_1^0, 0) = m_1^1$$

It can be verified that,

$$\forall s \in X_r, \forall \sigma \in \Sigma,\ \psi(\alpha_r(s, \sigma)) = \alpha_1(\psi(s), \sigma)$$

Hence, $M_1 \leq \mathcal{R}(\{A, B\})$.

Consider any two machines, $A\ (X_a, \Sigma, \alpha_a, a^0)$ and $B\ (X_b, \Sigma, \alpha_b, b^0)$. 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 $\sigma \in \Sigma$. This machine $(X_a \times X_b,\ \Sigma,\ \alpha',\ \{a^0, b^0\})$ may have states that are not reachable from the initial state $\{a^0, b^0\}$. If all such unreachable states are pruned, we get the *reachable cross product* of A and B, denoted $\mathcal{R}(\{A, B\})$.

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 $\mathcal{A}$.

Based on the partial order imposed by homomorphism, we define a closed partition lattice corresponding to a given set of machines.

5.3.1 Closed Partition Lattice

A *partition* P, on the state set X_a of a DFSM, $A\ (X_a, \Sigma, \alpha_a, a^0)$ 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 $i \neq j$ [37]. The elements B_i of a partition are called *blocks* and 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, on the state set of machine A, corresponds to a machine homomorphic to A. Hence each state s of such a machine corresponds to a set of states in machine A. For example in figure 5.1, 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 [23].

A partition P_1 is greater than or equal to another partition P_2 ($P_2 \leq P_1$) if each block of P_1 is contained in a block of P_2. If DFSMs X_1 and X_2 correspond to partitions P_1 and P_2 respectively, then $X_1 \leq X_2$ is equivalent to $P_1 \leq P_2$. Hence the $\leq$ relation between partitions corresponds to homomorphism in the resulting DFSMs. In Fig 5.1, each block of $\mathcal{R}(\{A, B\})$ is contained in a block of M_1 and hence, $M_1 \leq \mathcal{R}(\{A, B\})$.

Note that, the $\leq$ defines a partial order on the set of all closed partitions. In fact, it can be seen that the set of all closed partitions corresponding to a machine, form a lattice under the $\leq$ relation[23].

Given a set of n machines $\mathcal{A}$, we consider the lattice of all closed partitions corresponding to $\mathcal{R}(\mathcal{A})$. Figure 5.2 shows the closed partition lattice corresponding to $\mathcal{R}(\{A, B\})$ (denoted $\top$), shown in figure 5.1(iii). An arrow

from one machine to another indicates that the former is less than the latter. Both A (figure 5.1(i)) and B (figure 5.1(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.

We now define the lower cover of a partition, a concept used later in section 5.6.

Definition 5.3.3. (Lower Cover) The lower cover of a partition P of the set of states of any machine A , is the set of maximal partitions of A that are less than P.

The lower cover of P can be computed by combining any two blocks of P at a time and computing the new largest closed partition which is smaller 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 figure 5.2, the machines A, B, M_1 and M_2 constitute the basis.

5.4 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. We begin with the idea of a *fault graph* of a set of machines $\mathcal{M}$, for a machine T,

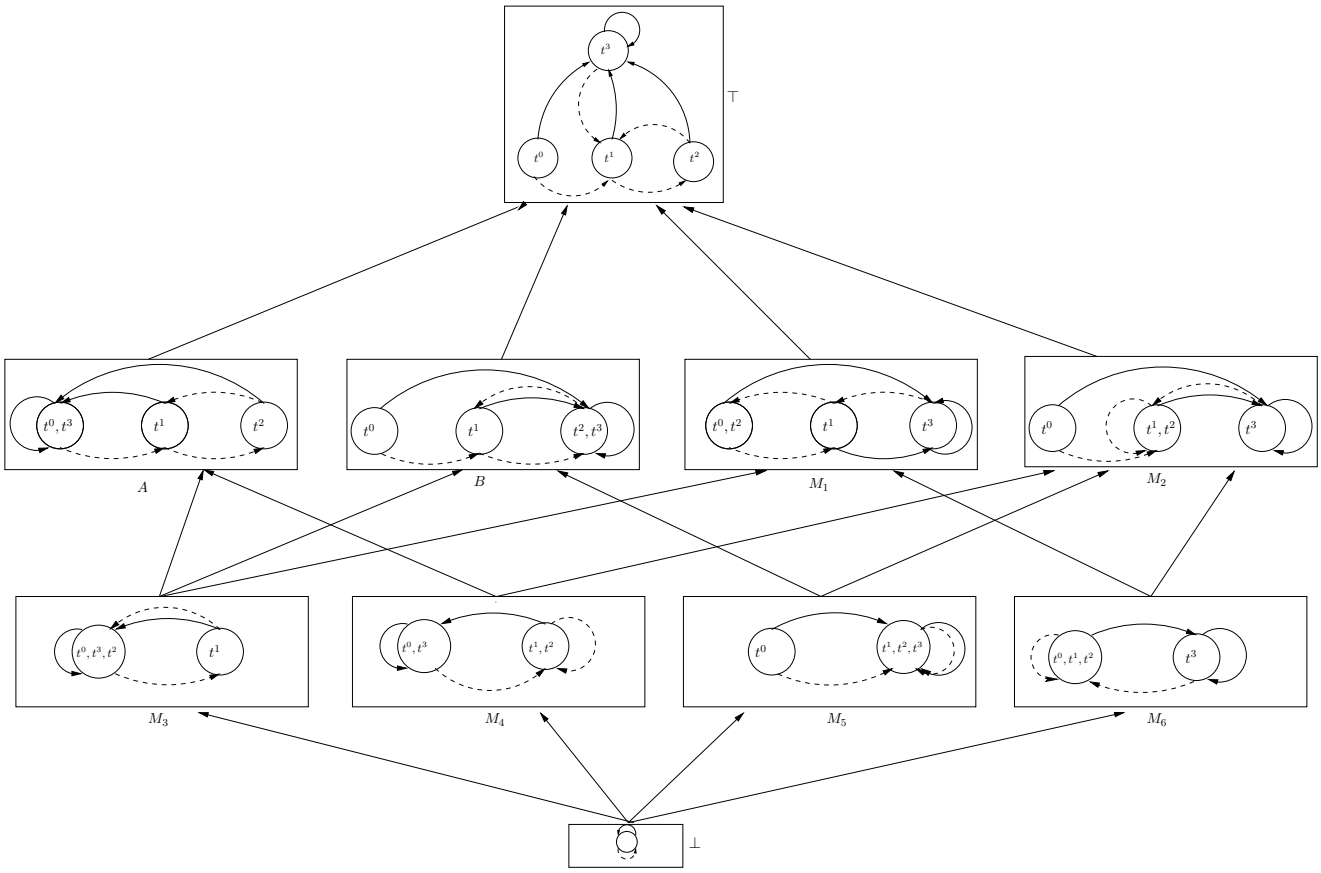


Figure 5.2: Closed Partition Lattice For Figure 5.1

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 figure 5.2, we construct the fault graph $G(\top, \{A\})$. 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 T 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 completely 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 figure 5.3, the edge (t^0, t^3) has weight 0 and all other edges have weight 1.

Definition 5.4.1. (Fault Graph) Given a set of machines $\mathcal{M}$ and a machine T $(X_T, \Sigma, \alpha, t^0)$ 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 completely 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

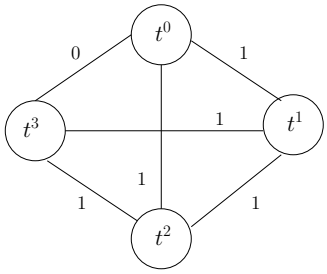
If a machine M has the states t^i and t^j in distinct blocks, it is said to *cover* the edge (t^i, t^j).

Assume there exists an edge (t^i, t^j), with weight 0, in $G(T, \mathcal{M})$. Given the states of all the machines in $\mathcal{M}$, it is impossible to determine if T is in t^i or t^j. Unless the weight of every edge in $G(T, \mathcal{M})$ is 1 or more, it is not possible to always determine the current state of T using $\mathcal{M}$.

Given a fault graph, $G(T, \mathcal{M})$, the lowest weight 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 figure 5.4 (ii). Since the lowest weight 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 faults.

The lowest weight in $G(T, \mathcal{M})$ is denoted $w(T, \mathcal{M})$ and the edges with this value are called the *weakest* edges of $G(T, \mathcal{M})$.

Theorem 5.4.1. *A set of machines $\mathcal{M}$, can tolerate up to f faults iff $w(T, \mathcal{M}) > f$, where T is the reachable cross-product of all machines in $\mathcal{M}$.*



$G(\top, \{A\})$

Figure 5.3: Fault Graph, $G(\top, \{A\})$, for machines shown in figure 5.2

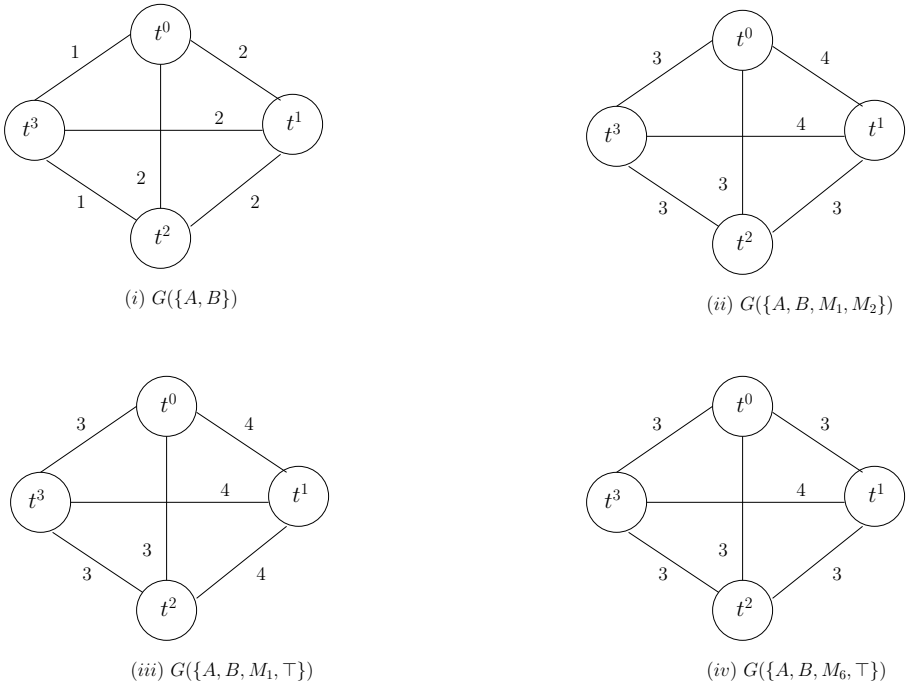


Figure 5.4: Fault Graphs, $G(\top, \mathcal{M})$, for sets of machines shown in figure 5.2

Proof. $\Rightarrow$ Given that $w(T, \mathcal{M}) > f$, we show that any $\mathcal{M} - f$ machines from $\mathcal{M}$ can accurately determine the current state of T. It is obvious that the current state of any DFSM in $\mathcal{M}$ can be determined if the state of T is known. The weight of any edge in fault graph $G(T, \mathcal{M})$ is greater than f since $w(T, \mathcal{M}) > f$. i.e., $f + 1$ or more machines separate any two states in the fault graph. Hence, for any pair of states t_i, t_j in T, after f failures in $\mathcal{M}$, there will always be at least one machine remaining that can distinguish between t_i and t_j. This implies that it is possible to accurately determine the current state of T by using any $\mathcal{M} - f$ machines from $\mathcal{M}$.

$\Leftarrow$ We now show that the system can not tolerate f faults when $w(T, \mathcal{M}) \leq f$. $w(T, \mathcal{M}) \leq f$ implies that there exists an edge, say (t_i, t_j) in $G(T, \mathcal{M})$ with weight, k where $k \leq f$. Hence there exist exactly k machines (say the set $\mathcal{M}' \subset$ of $\mathcal{M}$ in $\mathcal{M}$) that can distinguish between states (t_i, t_j) in T. Assume that all these k machines fail (since $k \leq f$) when T is in either t_i or t_j. Using the states of the remaining machines in $\mathcal{M}$, it is not possible to determine whether T was in state t_i or t_j. Therefore, it is not possible to exactly regenerate the state of any machine in $\mathcal{M}$ using the remaining machines.

□

The set of machines in $\mathcal{M}$ are said to have a fault tolerance of f or are f-fault tolerant.

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 $w(\mathcal{M})$ instead of $w(\top, \mathcal{M})$. From theorem 5.4.1, it is clear that we can determine the inherent fault tolerance in a given set of machines $\mathcal{A}$, simply by finding $w(\mathcal{A})$.

Remark 5.4.1. Given a set of n machines $\mathcal{A}$, the system can tolerate up to $w(\mathcal{A}) - 1$ faults.

Given the state of all the machines in $\mathcal{A}$, we can identify the state of $\top$. Hence, $w(\mathcal{A})$ is always greater than or equal to 1.

5.5 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. In this section, we characterize such backup machines based on the fault graph described in the previous section.

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 5.4.1, we know that, $w(\mathcal{A} \cup \mathcal{F}) > f$.

Definition 5.5.1. (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 $w(\mathcal{A} \cup \mathcal{F}) > f$.

Any machine belonging to m 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 figure 5.2. From figure 5.4(i), $w(\{A, B\}) = 1$. 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 2 faults. It can been seen from figure 5.4(ii) that $w(\{A, B, M_1, M_2\}) = 3$, and hence the set of machines $\{A, B, M_1, M_2\}$ can tolerate up to 2 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. As shown, $\{M_1, M_2\}$ 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 2 faults in any 2 machines $\{A, B\}$, replication will require 2 additional copies each of A and B. Hence, $\{A, A, B, B\}$ is a $(2, 4)$-fusion of $\{A, B\}$.

- $f > m$: From observation 5.4.1, if a system is inherently fault tolerant, then no additional machines may be needed to tolerate faults. In the example shown in figure 5.2, let us assume that the original set of machines are $\{A, B, M_1\}$. Since, $w(\{A, B, M_1\}) = 2$, these machines can tolerate one fault without any additional machine.

As we have seen before, $w(\{A, B, M_1, M_2\}) > 2$. Any machine in the set $\{A, B, M_1, M_2\}$ can at most contribute 1 to the weight of any edge in the graph $G(\{A, B, M_1, M_2\})$. Hence, even if we remove one of the machines, say M_2, from this set, $w(\{A, B, M_1\})$ will still be greater than 1. This implies that $\{M_1\}$ is a $(1, 1)$-fusion of $\{A, B\}$. Similarly, $\{M_2\}$ is also a $(1, 1)$-fusion of $\{A, B\}$. This property is generalized in the following theorem.

Theorem 5.5.1. *(Subset of a Fusion) Given a set of n machines $\mathcal{A}$, and an (f, m)-fusion $\mathcal{F}$, corresponding to it, any subset $\mathcal{F}' \subseteq \mathcal{F}$ such that $|\mathcal{F}'| = m - t$ is a $(f - t, m - t)$-fusion when $t \leq min(f, m)$.*

Proof. Since, $\mathcal{F}$ is an (f, m)-fusion of $\mathcal{A}$, according to the definition of (f, m)-fusion, $w(\mathcal{A} \cup \mathcal{F}) > f$.

Any machine, $F \in \mathcal{F}$, can at most contribute a value of 1 to the weight of any edge of the graph, $G(\mathcal{A} \cup \mathcal{F})$. Similarly, t machines in the set $\mathcal{F}$ can contribute a value of at most t to the weight of any edge of the graph, $G(\mathcal{A} \cup \mathcal{F})$. Therefore, even if we remove t machines from the set of machines in $\mathcal{F}$, $w(\mathcal{A} \cup \mathcal{F}) > f - t$.

Hence, for any subset $\mathcal{F}' \subseteq \mathcal{F}$, of size $m - t$, $w(\mathcal{A} \cup \mathcal{F}') > f - t$. This implies that $\mathcal{F}'$ is an $(f - t, m - t)$-fusion of $\mathcal{A}$.

□

It is important to note that the converse of this theorem is not true. For example, consider the machines M_1 and M_6 shown in figure 5.2. Even though

both $\{M_1\}$ and $\{M_6\}$ are $(1,1)$-fusions of $\{A,B\}$, since $w(\{A,B,M_1,M_6\}) = 2$, $\{M_1,M_6\}$ is not a $(2,2)$-fusion of $\{A,B\}$.

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 $X_\top$. So the basic intuition is that, 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 figure 5.2. From figure 5.4(i), $w(\{A,B\}) = 1$. We need exactly one machine F, such that, $w(\{A,B,F\}) > 2$. Even if F was the top machine, $w(\{A,B,\top\}) = 2$. Hence, there cannot exist a $(2,1)$-fusion for $\{A,B\}$. We formalize this in the following theorem.

Theorem 5.5.2. *(Existence of an (f,m)-fusion) Given a set of n machines $\mathcal{A}$, there exists an (f,m)-fusion of $\mathcal{A}$, iff, $m + w(\mathcal{A}) > f$.*

Proof. $\Rightarrow$

Assume that there exists an (f,m)-fusion $\mathcal{F}$ for the given set of machines $\mathcal{A}$. We will show that $m + w(\mathcal{A}) > f$.

Since, $\mathcal{F}$ is an (f,m)-fusion fusion of $\mathcal{A}$, $w(\mathcal{A} \cup \mathcal{F}) > f$. The m machines in $\mathcal{F}$, can at most contribute a value of m to the weight of each edge in $G(\mathcal{A} \cup \mathcal{F})$. Hence, $m + w(\mathcal{A})$ has to be greater than f.

$\Leftarrow$

Assume that $m + w(\mathcal{A}) > f$. We will show that there exists an (f, m)-fusion for the set of machines $\mathcal{A}$.

Consider a set of m machines $\mathcal{F}$, containing m replicas of the top. These m top machines, will contribute exactly m to the weight of each edge in $G(\mathcal{A} \cup \mathcal{F})$. Since, $m + w(\mathcal{A}) > f$, $w(\mathcal{A} \cup \mathcal{F}) > f$. Hence, $\mathcal{F}$ is an (f, m)-fusion of $\mathcal{A}$.

□

From this theorem, given a set of n machines $\mathcal{A}$ and an (f, m)-fusion $\mathcal{F}$, corresponding to it, $|\mathcal{F}| \geq f - w(\mathcal{A})$.

Given a set of machines, we now define an order among (f, m)-fusions corresponding to them.

Definition 5.5.2. (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 figure 5.4(iii), $w(\{A, B, M_1, \top\}) = 3$, 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 figure 5.2, $\{M_3, M_4, M_5, M_6\}$ are a set of minimal machines. It can be seen that $w(\{A, B, M_3, M_4, M_5, M_6\}) > 2$ and $\{M_3, M_4, M_5, M_6\}$ is a minimal $(2, 4)$-fusion of $\{A, B\}$.

5.6 Algorithms

In this section, we present a polynomial time algorithm to generate the minimum set of machines required to tolerate f faults among a given set of machines.

Algorithm 1 `minMachines`

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

```
1: F ← φ ;
2: while w(A ∪ F) ≤ f do
3:     E ← weakest edges in G(A ∪ F)
4:     F ← F∪ {genFusion(E)}
5: return F
```

Algorithm 2 `genFusion`

Input: E: set of edges
Output: F: minimal machine

```
1: R ← ⊤
2: while R ≠ ⊥ do
3:     C ← lower_cover(R)
4:     if ∃F ∈ C : F covers all edges in E then
5:         R ← F
6:     else
7:         break
8: return R
```

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. In each iteration of the while loop, we identify the set of weakest edges in the graph $G(\mathcal{A} \cup \mathcal{F})$ and denote it E. Algorithm 2, returns a minimal machine in the closed partition lattice of $\top$, which covers all the edges in E. The

addition of such a machine to $\mathcal{F}$, increases $w(\mathcal{A} \cup \mathcal{F})$ by 1. We continue till $w(\mathcal{A} \cup \mathcal{F}) > f$, and return the set of machines in $\mathcal{F}$.

In algorithm 2, we start with the $\top$, which clearly covers all the edges in E. We then try to find such a machine in the lower cover of the $\top$, and continue traversing down the lattice, until we encounter the bottom machine or till the lower cover does not contain such a machine.

When the set of machines in $\mathcal{A}$ execute along with the backup machines in $\mathcal{F}$, any f machines among them may crash. To recover the state of the f machines which crash, we first generate the state of the top using the remaining $n + m - f$ machines. Given the state of the top, we can identify the state of all the machines in $\mathcal{A} \cup \mathcal{F}$. Hence, we can recover the state of all the crashed machines. Note that, up to $n + m - f$ machines may be required for recovery. In contrast, recovery while using replication is simpler and requires examining the execution state of no more than f machines.

Consider the example shown in figure 5.2, with $\mathcal{A} = \{A, B\}$, and $f = 2$. We need to find a set of machines $\mathcal{F}$ such that $w(\mathcal{A} \cup \mathcal{F}) > 2$. Since $\mathcal{F}$ is empty to begin with, $G(\mathcal{A} \cup \mathcal{F}) = G(\{A, B\})$, shown in figure 5.4(i). The weakest edges are (t^0, t^3) and (t^2, t^3). Machine M_1, belonging to the basis, covers these edges, and so does machine M_6, in the lower cover of M_1. Since M_6, is a minimal machine of the lattice, algorithm 2 , returns M_6. Since, $w(\mathcal{A} \cup \{M_6\}) = 2$, the next iteration of the loop proceeds and finally, algorithm 1, returns $\mathcal{F} = \{M_6, \top\}$. From figure 5.4(iv), $w(\{A, B, M_6, \top) > 2$. Hence, $\mathcal{F}$ is a $(2, 2)$-fusion of $\{A, B\}$.

It is important to note that, algorithm 1 may generate a trivial solution, i.e., one including the top machine, even though there may exist other solutions with a larger number of non-trivial machines.

We now proceed to prove the correctness of the algorithm. It can be seen from the algorithm that every machine added, covers a set of edges. This set of edges (the input to algorithm 2) is called the edge set of that machine. The edge set of F_j is denoted by E_j.

Lemma 5.6.1. *Given a set of n machines $\mathcal{A}$, and the set $\mathcal{F}$ returned by algorithm 1, let $F_i \in \mathcal{F}$ be the machine returned in the i^{th} iteration. Then, $\forall F_i, F_j \in \mathcal{F} : i < j \Rightarrow E_i \subseteq E_j$.*

Proof. If $\mathcal{F}' \subseteq \mathcal{F}$ is the current fusion set during the execution of algorithm 1, then the edge set for the next iteration consists of the minimal edges of the fault graph $G(\mathcal{A} \bigcup \mathcal{F}')$. Every time a machine is added to $\mathcal{F}'$, the weights of the edges in $G(\mathcal{A} \bigcup \mathcal{F}')$ can increase by at most one and the weight of every minimal edge is incremented by exactly one. Hence, after every iteration the edge set for the next iteration can not decrease in size. This implies $\forall F_i, F_j \in \mathcal{F} : i < j \Rightarrow E_i \subseteq E_j$. □

Theorem 5.6.2. *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 $\mathcal{A}$.*

Proof. 1. $\mathcal{F}$ is a fusion with the minimum number of elements.

We show that $\mathcal{F}$ is an $(f, |\mathcal{F}|)$-fusion of $\mathcal{A}$ where, $|\mathcal{F}| = f - w(\mathcal{A})$. As seen in the previous section, this is the minimum number of machines in any (f, m)-fusion of $\mathcal{A}$.

The addition of any machine, $F \leq \top$, to the set of machines in $\mathcal{A} \cup \mathcal{F}$, can increase $w(\mathcal{A} \cup \mathcal{F})$ by at most 1. In each iteration of the loop in algorithm 1, we find a machine covering the weakest edges in $G(\mathcal{A} \cup \mathcal{F})$ and add it to $\mathcal{F}$. Hence, in each iteration of the while loop we increase $w(\mathcal{A} \cup \mathcal{F})$ exactly by 1 adding exactly one extra machine.

Initially, since $\mathcal{F} = \phi$, $w(\mathcal{A} \cup \mathcal{F}) = w(\mathcal{A})$, and finally, $w(\mathcal{A} \cup \mathcal{F}) = f$. Therefore, the number of machines added to $\mathcal{F}$ is $f - w(\mathcal{A})$. Since, $w(\mathcal{A} \cup \mathcal{F}) > f$, $\mathcal{F}$ is a $(f, |\mathcal{F}|)$-fusion of $\mathcal{A}$.

2. $\mathcal{F}$ is a minimal fusion.

   Lemma 5.6.1 implies that if an edge e occurs in the edge set of any machine in $\mathcal{F}$ and there are k machines in $\mathcal{F}$ that cover e, then in any valid $(f, |\mathcal{F}|)$-fusion there are at least k machines that cover edge e.

   Let there be an (f, m)-fusion $\mathcal{G} = \{G_1, ..G_m\}$, such that $\mathcal{G}$ is less than (f, m)-fusion $\mathcal{F}$ ($\mathcal{F} = \{F_1, F_2, ..., F_m\}$). Hence $\forall j : G_j \leq F_j$.

   Let $G_i < F_i$ and let E_i be the set of edges that needed to be covered by F_i. It follows from algorithm 1, that G_i does not cover at least one edge say e in E_i (otherwise algorithm 1 would have returned G_i instead of F_i). If e is covered by k DFSMs in $\mathcal{F}$, then e has to be covered by k machines in G.

We know that there is a pair of machines F_i, G_i such that F_i covers e and G_i does not cover e. For all other pairs F_j, G_j if G_j covers e then F_j covers e (since $G_j \leq F_j$). Hence e can be covered by no more than $k-1$ in $\mathcal{G}$. This implies that $\mathcal{G}$ is not a valid fusion.

□

We now consider the time complexity for algorithm 2. The time complexity to generate the lower cover of any machine R, less than the top, is $O(N^2.|\Sigma|)$ [37]. While generating a machine in the lower cover, we can determine if the machines cover all the edges in E, without any additional time. Since the number of machines in the lower cover is $O(N^2)$, each iteration of the while loop in algorithm 2, is of time complexity $O(N^2.|\Sigma|) + O(N^2) = O(N^2.|\Sigma|)$. As we traverse down the lattice, we combine at least two blocks of F. Thus the while loop in algorithm 2 is executed at most N times. Hence, the time complexity of algorithm 2 is $O(N^2.|\Sigma|) * O(N) = O(N^3.|\Sigma|)$.

In algorithm 1, in each iteration of the loop, we identify the weakest set of edges in $G(\mathcal{A} \cup \mathcal{F})$ and generate a minimal machine separating these edges, using algorithm 2. The time complexity for finding the weakest edges in $G(\mathcal{A} \cup \mathcal{F})$ is $O(N^2)$. Hence, the time complexity for each iteration in algorithm 1 is $O(N^3.|\Sigma|) + O(N^2) = O(N^3.|\Sigma|)$. Since the are f iterations of the loop, the time complexity of algorithm 1 is $O(N^3.|\Sigma|.f)$. Therefore, algorithm 1 generates the set of fusion machines with a time complexity polynomial in the size of the top machine.

5.7 Implementation and Results

We have implemented the algorithm specified in section 5.6 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 figure 5.1 (denoted A and B in the results table).

In the results table, along with the original machines, we have tabulated the number of faults tolerated (f), the size of the top ($|\top|$), sizes of the backup fusion machines generated by algorithm 1 (|Backup Machines|),and the state space required for our fusion based solution (|Fusion|).

Given a set of n machines, $\mathcal{A} = \{A_1, A_2, \ldots A_n\}$, in order to tolerate f faults among them, replication will require f copies of each machine. Hence the state space for replication is calculated as $(\prod_{i=1}^{i=n} |A_i|)^f$. If the set of backup machines generated by algorithm 1 is, $\mathcal{F} = \{F_1, F_2, \ldots F_m\}$, the state space for fusion is simply calculated as $\prod_{i=1}^{i=m} |F_i|$.

Original Machines	f	$\lvert\top\rvert$	|Backup Machines|	|Fusion|
MESI, One Counter, Zero Counter, Shift Register	2	87	[39 39]	1521
Even Parity Checker, Odd Parity Checker, Toggle Switch, Pattern Generator, MESI	3	64	[32 32 32]	32768
One Counter, Zero Counter, Divider, A, B	2	82	[18 28]	504
MESI, TCP, A, B	1	131	[85]	85
Pattern Generator, TCP, A, B	2	56	[44 56]	2464

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.

5.8 Remarks

We develop the theory for (f, m)-fusion and present a polynomial time algorithm to generate the minimum set of machines required to tolerate faults among a given set of machines. 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 fusion. Our approach shows that there are many cases for which we can do better. Hence, if we want to tolerate 5 faults among 1000 machines, replication will require

5000 extra machines. Using our algorithm we may achieve this with just 5 extra machines.

Chapter 6

Conclusion

6.1 Summary

This dissertation presents theory and algorithms to solve some of the currently faced problems in detecting and tolerating faults in distributed and parallel systems.

6.1.1 Predicate Detection

Predicate detection, i.e., determining formally if a distributed computation was faulty, is a hard problem in general. The state explosion forces the use of heuristics or restricting the classes of faults that can be detected. In this dissertation we introduce the idea of a basis and show that this can be used to efficiently detect predicates containing arbitrarily nested temporal operators $\diamondsuit$ and $\Box$ along with conjunctions, disjunctions and negations. Previously known techniques[57] could not handle disjunctions and negations in polynomial time, thus restricting the class of bugs that could be detected.

A basis of a computation with respect to a predicate is an exact and compact representation of all the consistent cuts in the computation that satisfy the predicate. Given a basis and any consistent cut in the computation,

it is possible to determine in polynomial time if that cut satisfies the predicate. Slicing introduced by Mittal and Garg [43] constitutes a special case of computing a basis when the predicate to be detected is regular.

Like computation slicing, we exploit the structural properties of the computational lattice and the predicate to avoid the state explosion problem. We introduce a new class of predicates called semiregular predicates. A semiregular predicate is more representative than a regular predicate. We show that any predicate expressible in BTL can be represented as a disjunction of semiregular predicates. Further more the number of semiregular predicates required to represent a BTL predicate is always polynomial in the number of processed and events in the system. We provide an algorithm based on this to compute the basis of a BTL predicate. This predicate detection technique has been implemented as Java toolkit that can accept compatible traces from programs in any language or platform.

This research has exciting applications, especially with the rapid growth of parallel and distributed computing. The algorithms presented here can be incorporated into an program development tool or a widely used IDE like Eclipse[51]. Programmers will be able to specify assertions and test cases and the tool can quickly pinpoint any concurrency or synchronization related violations in the program. We see a numerous potential research avenues in this direction. Another interesting avenue for research is controlling the computation [69] so that, whenever possible, previously detected bugs can be automatically in all future runs by adding extra synchronization.

6.2 Tolerating Faults

Reliability and the ability to recover from crashes without data loss is essential for many distributed applications. Replication is the most commonly used technique for tolerating faults. In this dissertation, we question the optimality of replication approaches used for server backups. For systems with fixed resources fusion based techniques offer much higher reliability than replication. For example, if 20 servers are available and say the program needs to maintain 10 name resolution tables each of which uses almost all the memory in the server. Using replication, we would be able to tolerate just one 1 fault in such a setup while fusion would allow up to 10 faults without any data loss.

For data back up, we take into consideration the structure and operations on data to develop fused data structures. Our experiments with the lock server application indicate that as expected, fused data structures work much better than replication. We have implemented a library with commonly used data structures in Java. In distributed applications, it is very easy to substitute standard data structures with the data structures from our library, thus allowing programs to be fault tolerant without any significant overhead.

We have presented algorithms for the fusion of common data structures. An interesting avenue for research would be to develop fusion techniques for other application specific data structures. Also, the fused structure for linked lists described in this paper does not perform as well as queues or sets. It would be interesting to explore new fusion algorithms for linked lists. Another important area of research is the effect of concurrency on the performance of

the fused data structures.

Backing up of program state is slightly more involved. We consider a system where each program can be modeled as a deterministic finite state machine. We introduce the theory of fusion machines that enables us to automatically generate back up machines. The number of states required is often much less than simply replicating the state machines. This makes it possible to have reliable and low cost systems, like sensor networks.

In this dissertation, we have considered machines belonging to the closed partition lattice of $\top$. It is possible that machines outside the lattice may provide more efficient solutions. Also, our algorithm 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. Also, it would be interesting to extend our approach to Byzantine faults.

Bibliography

[1] Bharath Balasubramanian, Vinit A. Ogale, and Vijay K. Garg. Fault tolerance in finite state machines using fusion. In *ICDCN*, volume 4904 of *Lecture Notes in Computer Science*, pages 124–134. Springer, 2008.

[2] N. Budhiraja, K. Marzullo, F. Scneider, and S. Toueg. *The Primary-Backup Approach*, chapter 8. ACM Press, Frontier Series. (S.J. Mullender Ed.), 1993.

[3] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. *SIGCOMM Comput. Commun. Rev.*, 28(4):56–67, 1998.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[5] C. Chase and V. K. Garg. On techniques and their limitations for the global predicate detection problem. In *Proc. of the Workshop on Distributed Algorithms*, pages 303 – 317, Le Mont-Saint-Michel, France, September 1995.

[6] C. Chase and V. K. Garg. Efficient detection of global predicates in a distributed system. *Distributed Computing*, 11(4), 1998.

[7] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[8] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[9] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[10] D. Drusinsky. The temporal rover and the ATG rover. In *Spin Model Checking and Verification*, volume 1885 of LNCS, pages 323–330, 2000.

[11] J. Esparza. Model checking using net unfoldings. In *Science Of Computer Programming*, volume 23(2), pages 151–195, 1994.

[12] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[13] V. K. Garg. *Elements of Distributed Computing*. Wiley & Sons, 2002.

[14] V. K. Garg. *Concurrent and Distributed Computing in Java*. Wiley & Sons, 2004.

[15] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the IEEE International Conference on*

Distributed Computing Systems, pages 423–430, Vancouver, BC, Canada, June 1995.

[16] V. K. Garg and N. Mittal. On slicing a distributed computation. In *21st International Conference on Distributed Computing Systems (ICDCS' 01)*, pages 322–329, Washington - Brussels - Tokyo, April 2001. IEEE.

[17] V. K. Garg, N. Mittal, and A. Sen. Applications of lattice theory to distributed computing. *ACM SIGACT Notes*, 34(3):40–61, September 2003.

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

[19] V M Glushkov. The abstract theory of automata. *RUSS MATH SURV*, 16(5):1–53, 1961.

[20] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

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

[22] S. Hartley. *Concurrent Programming: The Java Programming Language*. Oxford University Press, 1998.

[23] 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.

[24] K. Havelund and G. Rosu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification 2001*, volume 55 of ENTCS, 2001.

[25] Gerald Holzmann. The model checker SPIN. In *IEEE transactions on software engineering*, volume 23.5, pages 279–295, 1997.

[26] Gerald Holzmann. *The Spin Model Checker.* Addison-Wesley Professional, 2003.

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

[28] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Co., Reading, Mass., 1978.

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

[30] Open SystemC Initiative. http://www.systemc.org.

[31] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: A run-time assurance tool for Java programs. In *Runtime Verification 2001*, volume 55 of ENTCS, 2001.

[32] Donald K. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[35] Leslie Lamport. *LATEX: A document preparation system*. Addison-Wesley, 2nd edition, 1994.

[36] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

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

[38] J. H. Van Lint. *Introduction to Coding Theory*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[39] Witold Litwin, Rim Moussa, and Thomas J. E. Schwarz. Lh^*_{RS} - a highly-available scalable distributed data structure. *ACM Trans. Database Syst.*, 30(3):769–811, 2005.

[40] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. In *STOC*

'97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, pages 150–159, New York, NY, USA, 1997. ACM Press.

[41] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[42] N. Mittal and V. K. Garg. Software fault tolerance of distributed programs using computation slicing. In *IEEE International Conference on Distributed Computing Systems*, pages 105 – 113, Baltimore, Maryland, USA, 2003.

[43] Neeraj Mittal and Vijay K. Garg. Slicing a distributed computation: Techniques and theory. In *5th International Symposium on DIStributed Computing (DISC'01)*, pages 78–92, October 2001.

[44] Vinit Ogale and Vijay K. Garg. Brief announcement: Many slices are better than one. In *Proceedings of 20th International Symposium on Distributed Computing:DISC*, Stockholm, Sweden, 2006.

[45] Vinit Ogale and Vijay K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of 21st International Symposium on Distributed Computing:DISC*, pages 420–434, Lemasos, Cyprus, 2007.

[46] Vinit A. Ogale, Bharath Balasubramanian, and Vijay K. Garg. A fusion-based approach for tolerating faults in finite state machines. In

23rd IEEE International Parallel and Distributed Processing Symposium: IPDPS (submitted), 2008.

[47] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM Press.

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

[49] D. Peled. All from one, one for all: On model checking using representatives. In *5th International Conference on Computer Aided Verification (CAV)*, pages 409–423, 1993.

[50] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[51] Eclipse: An Open Development Platfrom. http://www.eclipse.org.

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

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

[54] Fred B. Schneider and Lidong Zhou. Implementing trustworthy services using replicated state machines. *IEEE Security and Privacy*, 3(5):34–43, 2005.

[55] A. Sen and V. K. Garg. Detecting temporal logic predicates in the happened before model. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2002.

[56] A. Sen and V. K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *7th International Conference on Principles of Distributed Systems*, December 2003.

[57] Alper Sen and V. K. Garg. Partial order trace analyzer (POTA) for distributed programs. In *Proceedings of the Third International Workshop on Runtime Verification (RV)*, July 2003.

[58] Alper Sen, Vinit Ogale, and Magdy S. Abadir. Predictive runtime verification of multi-processor socs in systemc. In *Design Automation Conference (DAC)*, June 2008.

[59] Koushik Sen, Grigore Rosu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, 2005.

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

[61] Swaminathan Sivasubramanian, Michal Szymaniak, Guillaume Pierre, and Maarten van Steen. Replication for web hosting systems. *ACM Comput. Surv.*, 36(3):291–334, 2004.

[62] S. D. Stoller and Y. Liu. Efficient symbolic detection of global properties in distributed systems. In *10th International Conference on Computer Aided Verification (CAV)*, volume 1855 of LNCS, pages 264–279, 2000.

[63] S. D. Stoller and F. B. Schneider. Faster possibility detection by combining two approaches. In *Proc. of the 9th International Workshop on Distributed Algorithms*, pages 318–332, Le Mont-Saint-Michel, France, September 1995. Springer-Verlag.

[64] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12thInternational Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279. Springer-Verlag, July 2000.

[65] Jeremy B. Sussman and Keith Marzullo. Comparing primary-backup and state machines for crash failures. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, page 90, New York, NY, USA, 1996. ACM Press.

[66] A. Tarafdar and V. K. Garg. Predicate control in distributed systems. Technical Report TR-PDS-97-006, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, 1997.

[67] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In *Proc. of the 9th Symposium on Parallel and Distributed Processing*, pages 763 – 769, Orlando, FL, April 1998. IEEE.

[68] A. Tarafdar and V. K. Garg. Software fault-tolerance of concurrent programs using controlled reexecution. In *Proceedings of the 13th International Symposium on DIStributed Computing (DISC)*, pages 210 – 224, Bratislava, Slovakia, September 1999.

[69] A. Tarafdar and V. K. Garg. Predicate control: synchronization in distributed computations with look-ahead. *Journal of Parallel and Distributed Computing*, 64(2):219–237, 2004.

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

[71] A. Valmari. A stubborn attack on state explosion. In *International Conference on Computer Aided Verification (CAV)*, volume 531 of LNCS, pages 156–165, 1990.

[72] Lin Yuan, Pushkin R. Pari, and Gang Qu. Finding redundant constraints in fsm minimization. July 2004.

Index

Vita

Vinit Ogale was born in Mumbai on the 3rd of August 1979, to Nanda and Arun Ogale. He graduated in with a degree in Bachelor of Engineering, majoring in Electronics from Datta Meghe College of Engineering, University of Mumbai in 2001. He joined the graduate program in Electrical and Computer Engineering department of the University of Texas in 2002. He was awarded the Master of Science degree in 2004.

Permanent address: 4/29, Moreshwar Krupa CHS,
Datar Colony, Bhandup(East),
Mumbai 400042, India

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.