Testing Distributed Programs with Re-execution

Richard Kilgore   *Craig M. Chase*

rkilgore@ece.utexas.edu   *chase@ece.utexas.edu*

TR-PDS-1997-004   **July 1997**

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

Testing Distributed Programs with Re-execution

Richard Kilgore* Craig M. Chase

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

August 7, 1997

Abstract

Finding errors in non-deterministic programs is complicated by the fact that a software defect-induced anomaly may occur during one program execution, and not the next. Our objective is to provide a practical yet powerful testing environment for distributed systems. By re-executing the program under strictly different message orderings, we can observe different potential behaviors of the distributed program. Contributions of this paper include our funnel and wave theory, and two re-execution algorithms: one that performs an optimal single test run, and another that creates an optimally small test suite of re-executions when the goal is complete coverage.

1 Introduction

Finding bugs in a program is always difficult. Much research has been devoted to testing and debugging of sequential programs [1, 3, 2, 7, 22]. Generally, an error in a program can start a chain reaction of unexpected events that is only noticeable externally much later. Detecting such anomalies usually involves running the program in a debugger, which allows one to view the internal state of the program, and step it forward in a controlled fashion. The process used to locate the bug usually involves running the program over and over, learning more and narrowing the search each time, until the mistake is finally found.

Since a typical execution of the program does not generally use all features of an application or provide all extremes or different types of input values, software testing is often used in an attempt to find more obscure bugs (e.g., divide by zero when the equation for the divisor rarely evaluates to 0). Using a test suite of inputs, the program is run a number of times, checking program trace data and outputs against expected values.

In a distributed computation, the problem is much worse. The inputs in most sequential programs are controllable by the programmer. It is possible to force a sequential program to

*This work supported by the Applied Research Labs

run exactly the same each time. Concurrent programs introduce the difficulty of race conditions. Consider a receive event on a process in a distributed computation at which more than one message is a candidate for delivery. A non-deterministic choice is made by the program when it chooses to receive one of these messages. This choice may affect data in future messages, or it may affect the control flow and consequently which messages are sent by this process in the future.

A bug observed during one execution run, may not have any effect on the computation in a subsequent run. This type of non-deterministic behavior makes software testing more difficult. Checking the results of the test is no longer simply a task of matching outputs with expected results, because the program's trace data varies from one execution to the next.

Most distributed debugging research has focused on either visualization [4, 12, 15, 23], or allowing the programmer to control execution or re-execution [8, 12, 24], or presenting dependence analysis information [6]. Methods for reproducing a distributed computation using trace and replay mechanisms are well known [11, 17, 21]. In [19], given information from the programmer that specifies which message races in a program are intentional, the authors show an approach for reporting the first unintentional non-determinism in a message-passing program.

In Sections 6 and 7, we attempt to automatically search the program execution space, looking for bugs of a type more familiar to a typical programmer. By re-executing the program and imposing a strictly different message ordering, we attempt to force bugs in a distributed program to cause anomalies.

The brute force method of exploring all possible executions of a distributed computation has been effectively argued to be intractable [5, 20]. Thus, we would like to find a method to explore a smaller set of possible executions that somehow captures important characteristics of all (or many) executions in the complete set.

Our current approach is applicable to distributed programs whose semantics dictate a *fixed message set* (See Section 4.1). We believe that certain types of distributed applications meet these requirements (e.g. loosely synchronous, divide-and-conquer-style programs). Additionally, we believe that the theory will be extendible in future work to more general types of distributed programs. We attempt to reverse pairs of messages that raced during the original test run. The algorithm presented in Section 6 produces an optimal single test run. In other words, it creates a single test run that reverses as many pairs of racing messages as possible. The algorithm requires $O(k^2)$ execution time when run on a process that passes k messages in a single execution run. It can be run in parallel on all processes in the system.

In Section 7, we present an algorithm for calculating a minimal set of test runs to reverse all message pairs. The number of test runs required to achieve this 100% coverage can be determined after the first test run.

2 Model

In this section we present our model of a distributed computation, and provide a few definitions and notational conventions.

2.1 Processes and Distributed Systems

In this paper, we model a process as a program that performs Turing-style computations. It can output part of its local state by sending a message to another process in the system, and it can receive a message from another process as input. A distributed system is a set of processes that communicate with one another by sending and receiving messages to and from one another as they perform computations. A process is represented with the symbol P, and we use subscripts when we need to refer to more than one process at a time.

2.2 Test Runs

An execution run on a single process is modeled as a sequence of events. Each event is either a receive event, at which the process delivers a message from some other process in the system; a send event, at which the process sends a message to another process; or an internal event, at which the process's local state changes. We use the symbol T to represent such a sequence, with subscripts if needed. Hereafter, we refer to a sequence T on a single process as a *test run*. A test run must have an initial state (i.e., any prefix is finite).

Given a test run T, if T_{pre} is a prefix of T, then $T_{pre} \leq T$ (or $T_{pre} < T$ if it is a strict prefix). More generally, we call any sequence contained in T a sub-computation of T.

We assume that the program that creates a test run is *piece-wise deterministic*. In other words, the order in which messages are delivered by receive events in the process is its only element of non-determinism. If the program is run twice, and it delivers the same messages in the same sequence, then it will undergo the same state changes, send the same messages, and attempt to receive the same messages.

A distributed computation has an initial state and a final state. The initial state is composed of all processes' initial states, and the final state is reached when all processes have terminated. The algorithms presented in this paper are directly applicable to real-world distributed computations that exhibit this behavior, but they can also be applied to sub-computations of a non-terminating distributed computation.

2.3 Relations

Given two events e_1 and e_2 from test run T, $e_1 \prec_T e_2$ indicates that e_1 was executed before e_2, and $e_1 \preccurlyeq_T e_2$ indicates that they might also be the same event. Since T is a sequential execution on a single process, $\prec_T$ imposes total ordering over the events in T. When T is clear from the context, $\prec_T$, pronounced "precedes", can be represented simply by the symbol $\prec$. As in Lamport [16], there exists a partial order, $\rightarrow$, between events in a distributed computation. For any two events, e and e', $e \ \rightarrow \ e'$ iff:

1. $e \prec_T e'$, where T is a test run on a given process, or
2. e is a send event, at which message m is sent; and e' is a receive event, at which m is delivered, or
3. $\langle \exists e'' :: e \ \rightarrow \ e'' \wedge e'' \ \rightarrow \ e' \rangle$

As discussed in Lamport's paper, this "happens before" relation is also transitive and anti-reflexive. When $e \rightarrow e'$, we also say that e' *causally follows* e. If $e \not\rightarrow e'$ and $e' \not\rightarrow e$, then the two events are *concurrent* and may occur in either order during a given test run.

2.4 Notational Conventions

When a message has been sent, we say it is *available*. When a message is actually passed to an application process by completion of a **msg_recv**( ) call, we say that it is *delivered*. Our testing algorithms predetermine which message to deliver to each **msg_recv**( ) call by the application.

As mentioned earlier, processes in the system will be represented with the symbol P and a test run by T. A message passed during a distributed computation will be represented by m, and send and receive events by s and r, respectively. In all cases, if we need to refer to more than one at a time, we will use subscripts and/or symbols like m'. In a test run, for a given message m, $m.s_T$ represents the send event in T at which m is sent. $m.r_T$ represents the receive event in T at which m is delivered. When the test run to which we are referring is clear, these events will be abbreviated as $m.s$ and $m.r$.

Another necessary distinction is between the original, non-deterministic test run, and a controlled test run. In the *original test run*, each process's behavior is chosen non-deterministically, as processes are allowed to progress at their natural speeds and deliver available messages in any arbitrary order. If we refer generically to a test run on some process P by the symbol T, then the *original test run* will be denoted T_o. The second type of test run is pre-planned by a testing algorithm, and realized as a debugger/tester forces the underlying computation to deliver messages in the planned order.

3 Race Sets

Both contributions in this paper rely on the ability to determine which messages received during a test run race with one another. In this section we group messages into sets: one for each receive event in T_o. If two messages appear together in any of these sets, they race.

Here we need to introduce the concept of a *race set*. *r.raceset* is the race set for receive event r. It contains all messages that might have been available for delivery immediately preceding the occurrence of event r during a given test run, depending upon the relative speeds of the processes and communication channels. First, we define the set *r.avail* of all messages that can exist before the occurrence of r.

Definition 3.1 *Given T on process P and a receive event r on T, define r.avail as follows:*

$$r.avail \triangleq \{m : (m \textit{ is sent to } P) \ \wedge r \not\rightarrow m.s\}$$

The expression $(r \not\rightarrow m.s)$ can be evaluated efficiently using vector clocks [9, 18].

Lemma 3.2 *r.avail is increasing over the sequence of receive events in a test run:*
Proof:

$$
\begin{array}{ll}
 & m \in r.avail \wedge r \prec r' \\
\Rightarrow & \{ \text{ Definitions of r.avail and } \rightarrow \} \\
 & r \not\rightarrow m.s \wedge r \rightarrow r' \wedge (m \textit{ sent to } P) \\
\Rightarrow & \{ \rightarrow \text{ transitivity } \} \\
 & r' \not\rightarrow m.s \wedge (m \textit{ sent to } P) \\
\equiv & \{ \text{ Definition of r.avail } \} \\
 & m \in r'.avail \quad \blacksquare
\end{array}
$$

Next we define *r.recvd* as the set of messages delivered before event r in test run T.

Definition 3.3 *Given T on process P and a receive event r on T, define r.recvd as follows:*

$$r.recvd \triangleq \{m : m.r \prec_T r\}$$

As represented in the next lemma, any message must be available before it can be received.

Lemma 3.4 $r.recvd \subseteq r.avail$
Proof:

$$
\begin{array}{ll}
 & m \in r.recvd \\
\equiv & \{ \text{ Definition of r.recvd } \} \\
 & m.r \prec r \\
\equiv & \{ \ m.s \rightarrow m.r, \text{ definition of } \rightarrow \} \\
 & m.s \rightarrow r \wedge (m \textit{ sent to } P) \\
\Rightarrow & \{ \rightarrow \text{ anti-reflexive } \} \\
 & r \not\rightarrow m.s \wedge (m \textit{ sent to } P) \\
\equiv & \{ \text{ Definition of r.avail } \} \\
 & m \in r.avail \quad \blacksquare
\end{array}
$$

And finally, *r.raceset* is just *r.avail* minus the messages in *r.recvd* (i.e., any message that can possibly be ready for delivery that has not already been delivered).

Definition 3.5 *Given T on process P and a receive event r on T, define r.raceset as follows:*

$$r.raceset \triangleq r.avail - r.recvd$$

Example: Consider a test run in which process P receives 5 messages: a, b, c, d, and e, in that order (see Figure 1). Note that after each of the send events, new messages are introduced into the *r.avail*, and hence *r.raceset*. This is because, these messages are causally related to earlier intervals of P's execution, but after the send event, P has reached an interval with which these message's send events are concurrent.

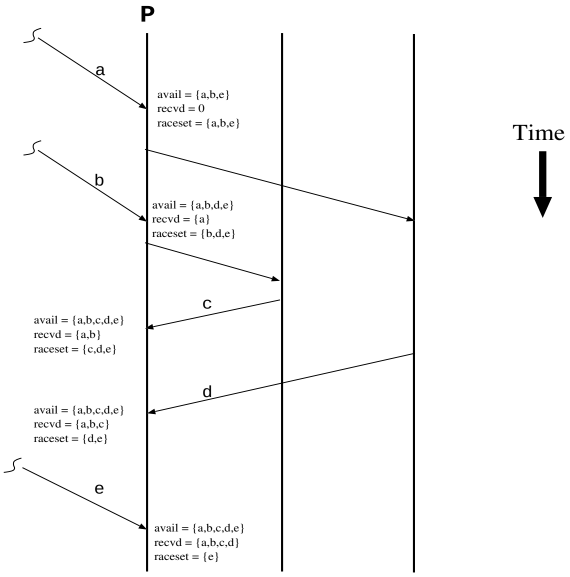

Figure 1: A Test Run and its Race Sets

We need to calculate $r.raceset$ for each receive event in a test run. As demonstrated by the example of Figure 1, each message first becomes available after some send event on P. Thus, the messages received by P during a test run can be partitioned into *groups*: a new (possibly empty) group after each send event. To calculate $r.raceset$ for each receive event r in a test run T, we need only keep a log of these groups, and of the receive event $m.r_T$ for each message m. The groups are totally ordered and should be stored in order. After the test run (or during), we can analyze the receive events in order, and subtract messages from the groups when they are received. In other words, when calculating $r.raceset$ for receive event r, consider all groups for which the associated send event s satisfies $s \prec r$. Set $r.raceset$ to the sum of all such groups. Then, find the message in this set for which $m.r_T = r$, and remove it from its group.

Groups will be discussed in greater detail in Section 5.1. A detailed discussion of detecting race sets *on-line* can be found in [13, 14].

4 Pruning the Execution Space

Instead of involving a programmer with whether or not messages race, another approach is to design a software tester/debugger tool that can explore the effects of possible races automatically. The programmer might be more interested in the fact that a task wait queue has exceeded some threshold, or that "index > array_size".

As discussed in Section 1, an approach to provide this capability is to prune the execution search space, and re-execute the program using pre-planned test executions. Our approach is to automatically explore alternate orderings on the delivery of pairs of messages that race, thus forcing test runs with different properties. Then, a user-specified predicate, like "$x > 10$" can be checked normally during these runs.

4.1 Fixed Message Set

Once the set of racing messages is determined for each receive event (see Section 3), we need to reorder them in some systematic fashion in an attempt to exploit software defects. The most immediate obstacle is the enormity of the search space. It is easy to imagine an imperfect program that never terminates when racing messages are delivered in a certain order.

If $|r.raceset| > 1$ for some receive event r on process P, then r can introduce non-determinism into the computation. If P received an alternate message at this receive event, it might act differently in the future. We restrict ourselves to distributed programs where the non-determinism introduced by a race does not change the affected process's set of future send and receive events. Regardless of the order in which messages were delivered to process P in its history, its underlying program will still evoke the same sequence of **msg_send**() and **msg_recv**() calls in the future. The data in these messages may be different, but send events will send to the same processes, and receive events will be the same requests. Thus, the set of messages sent by all processes in the distributed computation will also be the same, regardless of the order in which racing messages are delivered. We say that this kind of distributed program has a *fixed message set*. In such a system, messages can be reordered more easily in test runs.

A broad class of programs that typically exhibit this type of behavior are the "loosely synchronous" [10] parallel programs. Fox defined a loosely synchronous program as one that alternates between phases of global computation and communication. Typically, these programs send and receive the same messages during each communication phase, regardless of the results it obtained during the last communication phase. Non-determinism arises in such a program when they are executed on asynchronous systems. Concurrent messages can race during a global communication phase.

This restriction affects our future discussion in a few ways. First, the set of all receive events in T_o is the same set as it is in any test run T. Further, the total order imposed by $\prec_{T_o}$ on the events in this set is identical to the total order imposed by $\prec_T$, for any test run T. The following lemmas are easily derived.

Lemma 4.1 *For a given receive event r in a program with a fixed message set,* $r.avail_{T_o} = r.avail_T$.

Lemma 4.2 *For a given receive event r in a program with a fixed message set,* $|r.recvd_{T_o}| = |r.recvd_T|$, *and* $|r.raceset_{T_o}| = |r.raceset_T|$.

4.2 Reordering Message Deliveries

To force different choices in a test run, we wish to reorder as many pairs of messages as possible. In other words, suppose messages m_1 and m_2 were delivered in T_o such that $m_1.r \prec_{T_o} m_2.r$. We want to construct a test run, T, in which the messages are delivered in the opposite order: $m_2.r \prec_T m_1.r$.

To construct T, we select a single message to deliver at each receive event r from its $r.avail$ set. From Lemma 4.1, $r.avail_T = r.avail_{T_o}$. The only care that must be taken, is that we cannot assign for delivery the same message by more than one receive event in T.

We can simply visit the receive events in the order in which they appear in T_o (and in T), assigning a message for delivery to each receive event in sequence. This way, we know the value of $r.recvd$ in T for each r evaluated, and we can avoid selecting the same message twice by simply selecting a message from only those in $r.avail - r.recvd_T$, (i.e., $r.raceset_T$).

Here, we introduce another type of sequence: a message sequence. The message sequence $M_{T_o} = \langle m_1 m_2 ... \rangle$ represents the scenario that message m_1 is delivered by the first receive event in T_o, m_2 is delivered by the second receive event, and so on. In this sequence, we use the notation $m_1 \prec_{T_o} m_2$ to express the condition that m_1 precedes m_2 in M_{T_o}. This predicate can also be represented using the more cumbersome expression $m_1.r_{T_o} \prec_{T_o} m_2.r_{T_o}$.

5 Groups, Funnels, and Waves

We number the receive events on a given process by their order in the execution: $\langle r_1, r_2, ... \rangle$. Messages are partitioned into **groups** by their first appearance in the $r.avail$ sets.

5.1 Groups

Using the numbering scheme described above for receive events, recall from Lemma 3.2 that

$$r_{x-1}.avail \subseteq r_x.avail$$

is always satisfied. In the following definition, a *group* is formed every time $r_{x-1}.avail \subset r_x.avail$.

Definition 5.1 *Wherever $r_{x-1}.avail \subset r_x.avail$, a* **group** *is formed, containing the set of newly available messages $(r_x.avail - r_{x-1}.avail)$. The messages in $r_1.avail$ also form the first group (G_1) in the execution.* □

An example of groups will be given will be given later in this section (see Figure 3).

For group G, we name the first receive event at which the messages in this group are available $G.r_f$. The last receive event with the same $r.avail$ is denoted $G.r_l$. This event is either the last receive event in the execution, or it immediately precedes r_f of the next group.

Definition 5.2 $G.R \triangleq \{r : G.r_f \preccurlyeq r \preccurlyeq G.r_l\}$

As with receive events, we number groups by their order in the execution: $G_1, G_2, ..., G_g$, and we overload the $\prec$ operator over groups as follows:

$$G_i \prec G_j \triangleq G_i.r_l \prec G_j.r_f.$$

$G_i = G_j$ expresses the condition that these two symbols refer to the same group, and $G_i \preccurlyeq G_j$ is derived as expected.

As a result of the *fixed message set*, the set of receive events and $r.avail$ for each receive event remains constant between test runs. Thus, so does the set of groups. Each receive event r in the subsequence $G.r_f \preccurlyeq r \preccurlyeq G.r_l$, has the same $r.avail$ set. We name this set $G.Avail$.

Definition 5.3 $G.Avail \triangleq \{m : \langle \exists G' \preccurlyeq G :: m \in G' \rangle\}$ □

5.2 Funnels

In a perfect test run, if $M_{T_o} = \langle m_1 m_2 ... m_k \rangle$, we would like to be able to deliver the messages in complete reverse order: $M_T = \langle m_k m_{k-1} ... m_1 \rangle$. Such a test run would successfully reverse every pair of messages. This is not always possible, as shown by the following example.

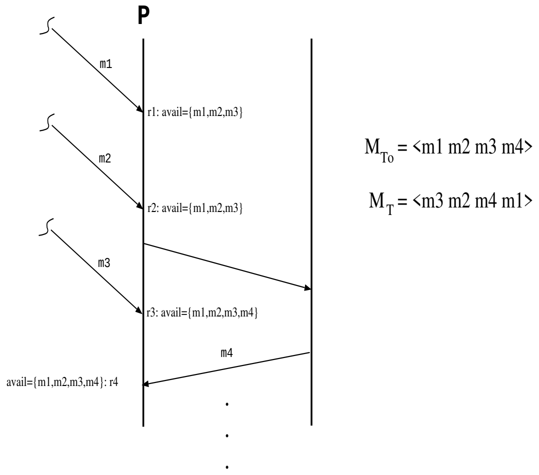

Figure 2: A $\mathbf{T_o}$ that Cannot be Reversed

Example: Consider the execution shown in Figure 2. We would like to deliver the messages in the order $\langle m_4 m_3 m_2 m_1 \rangle$. This sequence would reorder every pair of messages using only a single test run. However, since message m_4 is not included in $r_1.avail$ or $r_2.avail$, only messages from $\{m_1, m_2, m_3\}$ can be delivered by r_1 and r_2 in any test run. This reality is the consequence of a restrictive **funnel**, following the receive event r_2.

Definition 5.4 *For every group G_i, except the last one in the execution, there is a* **funnel**, *F_i, between receive events $G_i.r_l$ and $G_{i+1}.r_f$.*

The funnel has a throughput, given by the following equation:

$$F_i.tp = |G_i.Avail| - |\{r : r \preccurlyeq G_i.r_l\}|. \quad \square$$

We shall see in Section 7 that the key obstacle in reordering all messages in as few test runs as possible will be the need to deliver the messages in $G_i.Avail$ by receive events that follow $G_i.r_l$. We say that such a message is *passed through* the funnel.

Each $r \in \{r : r \preccurlyeq G_i.r_l\}$ must deliver a message from the set $G_i.Avail$ (since other messages are not available at these receive events). Thus, during a single test run, at most $|G_i.Avail| - |\{r : r \preccurlyeq G_i.r_l\}|$ (i.e., $F_i.tp$) messages from $G_i.Avail$ can be delivered after $G_i.r_l$. In other words, $F_i.tp$ messages *"pass through"* funnel F_i in a given test run.

In the discussion that follows, we include funnels in a test run's total order of events. Funnel F_i's position in the order is given by the following expression:

$$G_i.r_l \prec F_i \prec G_{i+1}.r_f.$$

5.3 Waves

We will show in this section that the messages received in a test run T_o, might be divided into *waves*. If two messages were delivered in different waves during the original test run, it is not possible to deliver them in the opposite order during a subsequent test run. Formally, if m_1 and m_2 were in different waves in T_o, and $m_1 \prec_{T_o} m_2$, then $\langle \nexists T :: m_2 \prec_T m_1 \rangle$.

Consider a loosely synchronous distributed computation, like the one described in Section 4.1. Suppose that at the end of every communication phase, the processes in the system perform a barrier synchronization. Process P cannot possibly deliver any messages sent in future communication phases during the current phase, because the other processes in the system will not advance to the point of sending these messages until P delivers all the messages in the current phase. Once P does so, its underlying program has no more receive events left in which a message from a future communication phase could be delivered.

Waves are formed by funnels that do not let *any* messages pass through. Each funnel F, whose $F.tp$ is 0 forms a wave boundary. In general, a program does not have to be structured with explicit synchronization points for the messages it receives to be divided into waves. If a distributed computation simply does not contain very many message races, then most receive events will have a race set of size 1 (i.e., no race). This condition creates a funnel with zero throughput (i.e., a wave boundary).

Lemma 5.5 *Consider two messages m_1 and m_2 received by the same process during T_o. If there exists a funnel F_i, for which $F_i.tp = 0$ and $m_1 \prec_{T_o} F_i \prec_{T_o} m_2$, then m_1 and m_2* can not *be delivered in reverse order during any test run.*

Proof: By contradiction.

Suppose that m_2 is delivered before m_1 in some test run T. Since $m_1 \prec_{T_o} F_i$, $m_1 \in G_i.Avail$. From our discussion of funnels, we know that m_1 can be delivered no later than receive event $G_i.r_l$ in any T. Thus, m_2 must be delivered by some r', $r' \prec_T G_i.r_l$. For this to be possible, we must have $m_2 \in G_i.Avail$. But this contradicts the fact that $F_i \prec_{T_o} m_2$, because no message from $G_i.Avail$ can pass through F_i. ■

In Lemma 5.5, funnel F_i forms a wave boundary. Messages available before F_i are separated from those that are not.

Lemma 5.6 *Consider two messages, m_1 and m_2, where $m_1 \prec_{T_o} m_2$, and*

$$\langle \nexists F_i : m_1 \prec_{T_o} F_i \prec_{T_o} m_2 : F_i.tp = 0 \rangle.$$

m_1 and m_2 can *be delivered in reverse order in some test run T.*

Proof:

Since there is no funnel F_i between $m_1.r_{T_o}$ and $m_2.r_{T_o}$ with $F_i.tp = 0$, m_1 can be passed through all funnels that do exist between them. At each such funnel F_j, $F_j.tp > 0$. Therefore, from the definition of $F_j.tp$,

$$|G_j.Avail| > |\{r : r \preccurlyeq G_j.r_l\}|.$$

Thus, $|\{r : r \preccurlyeq G_j.r_l\}|$ messages other than m_1 can be chosen from $G_j.Avail$ for delivery at the receive events in $\{r : r \preccurlyeq G_j.r_l\}$. Once m_1 has passed through the last funnel between $m_1.r_{T_o}$ and

$m_2.r_{T_o}$, m_2 is in $r.avail$ for the very next receive event r in the test run. Thus, m_2 can be delivered by r and m_1 delivered by a later receive event. ∎

In the lemma above, messages m_1 and m_2 are in the same *wave*, as are the receive events $m_1.r_{T_o}$ and $m_2.r_{T_o}$. We say that any pair of messages or receive events not separated by a funnel with zero throughput are *"in the same wave"*. Thus, from Lemma 5.6, any pair of messages in the same wave can be delivered in reverse order, and from Lemma 5.5, any two messages from different waves cannot.

5.4 Example Wave

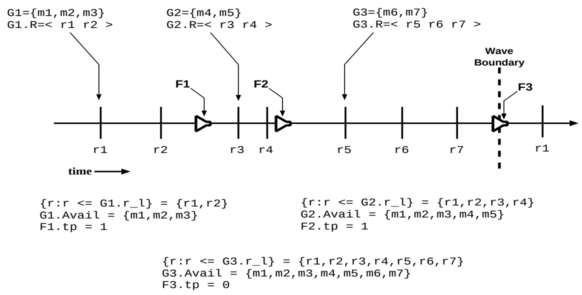

Figure 3: Groups and Funnels

5.5 Funnel and Wave Consequences

For the remainder of this paper, our discussion is limited to the discussion of receive events and messages within a single wave. We define $G_i.Avail_W$ as the messages in $G_i.Avail$ that first become available in the current wave, W. In other words, if r' is the last receive event of the wave before W, $G_i.Avail_W = G_i.Avail - r'.avail$.

Since it is not always possible to deliver all messages in complete reverse order (see Section 5.2), we will try to reverse as many pairs of messages as possible.

As we discussed earlier, at most $F_i.tp$ messages from $G_i.Avail_W$ can be delivered after $G_i.r_l$. The messages delivered by receive events in $\{r : r \preccurlyeq G_i.r_l\}$ cannot be delivered *after* any message in a future group, such as G_{i+1}. Lemma 5.7 follows trivially from this observation, and the fact that all messages in $G_i.Avail_W$ must be delivered after the messages in future groups to achieve 100% message reordering coverage over wave W.

Lemma 5.7 *100% message reordering coverage over a wave containing funnel F_i will require at least $\left\lceil \frac{|G_i.Avail_W|}{F_i.tp} \right\rceil$ test runs.*

The critical funnel defined below imposes a lower bound on the number of test runs we will need in order to achieve 100% coverage.

Definition 5.8 *The funnel for which the ratio $\left\lceil \frac{|G_i.Avail_W|}{F_i.tp} \right\rceil$ is maximal is called the* **critical funnel**, *F_c.*

6 Single Test Run

In this section, we present an algorithm for reversing as many message pairs as possible in a single test run. On the process P that we are testing, first we allow T_o to run non-deterministically. During T_o's execution, we observe the program's message set and wave boundaries, and then calculate $r.avail$ for all receive events on P.

6.1 Last-First Reordering Algorithm

```
TestRun Last-First(TestRun T_o)
{
  TestRun T := T_o;
  RecvEvent r_o := FirstRecv(T_o);
  RecvEvent r_t := FirstRecv(T);
  SetofMsgs r_t.recvd := ∅;

  /* Note: Messages in sets sorted by
      total order (m.r, ≺_{T_o}).  */
  /* r_o.avail is calculated on T_o from
      Definition 3.1.  */
  /* r_t.recvd is r.recvd_T */

  while(r_o != NULL)
    r_t.raceset := r_o.avail − r_t.recvd;
    msg m := tail(r_t.raceset);
    deliver(r_t, m);
    r_t.recvd := r_t.recvd ∪ {m};
    r_o := NextRecv(T_o);
    r_t := NextRecv(T);
  endwhile

  return T;
}
```

Figure 4: Procedure Last-First( )

A very simple algorithm, shown in Figure 4, can be used to reverse the greatest number of pairs possible in a single test run, T. In the algorithm shown in this figure, r_o and r_t are used as iterators to scroll through the receive events in T_o and T. The set r_t.recvd contains all messages that have been assigned for delivery by some receive event r in T, $r \prec r_t$. Since Last-First( ) subtracts this set from the messages it considers for delivery at each step, it will never deliver the same message twice.

The set $r_o.avail$ is calculated as described in Section 3, and from Lemma 4.1 we know that $r_o.avail = r_t.avail$. This set is sorted by the order in which the messages were received in T_o. Thus, from the set of messages in $r_o.avail$ that have not already been assigned for delivery by an earlier receive event in T, m gets the one that was delivered last in T_o. In other words, for any pair of messages (m_1,m_2) in $r_t.raceset$ where $m_1 \prec_{T_o} m_2$, Last-First( ) will select m_2 before it selects m_1.

An efficient implementation of Last-First( ) would actually keep only the set of messages that become available for the first time at each receive event, and add them to some set S at each iteration of the while loop. Also, instead of keeping r_t.recvd, we can just subtract each message from S as it is received. If we use a balanced tree to store the messages in S, the Last-First( ) algorithm runs in $O(k \cdot log\ k)$, for a sequence T_o with k receive events.

6.2 Last-First Analysis

To see that Last-First( ) is optimal, let's first examine the properties of the messages in the wave, and the effects of delivering one message instead of another at each receive event.

Definition 6.1 *For a given test run T, we define the set $m.lost_T$ for each message m in the wave:*

$$m.lost_T \triangleq \{m' : (m \prec_{T_o} m') \wedge (m \prec_T m')\}$$

Each message m' in $m.lost_T$ represents a pair (m, m') that is *not* reversed in T. The name *lost* refers to the lost opportunity to reverse these pairs. In a wave of size k, there are $\binom{k}{2}$ pairs. An optimal test run reverses the most pairs possible, so it contains the fewest *lost* pairs. For a test run T, the set of all lost pairs is given by:

$$T.lost \triangleq \{(m, m') : m' \in m.lost_T\}$$

Thus, an optimal test run contains a minimal $T.lost$ set.

We can now present an important lemma that is integral to proving our algorithm optimal.

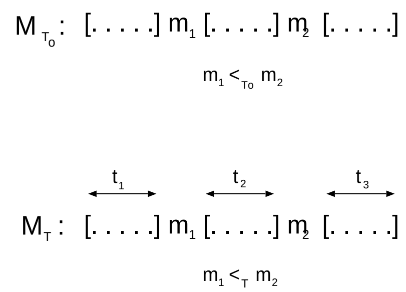

Figure 5: T - A test run that can be improved

Lemma 6.2 *Consider an original test run over a wave, T_o, and a test run T for the same wave. If T contains a pair of messages, (m_1, m_2), such that*

1. *$m_2 \in m_1.r_T.avail$ (i.e., m_1 is delivered by r in T, and $m_2 \in r.avail$)*

2. *$m_2 \in m_1.lost_T$*

then there exists a test run T', such that $|T'.lost| < |T.lost|$.

Proof: We construct T'.
Construct T' as follows: duplicate T, and then swap m_1 and m_2 in $M_{T'}$. Since $m_2 \in m_1.r_T.avail$ (1 above), T' is a valid test run.

The message sequence M_T is shown in Figure 5. The messages in the sequences t_1, t_2, and t_3 are in the same positions in $M_{T'}$. To prove that $|T'.lost| < |T.lost|$, we compare the sets $m'.lost$ for every message m' in both test runs.

Observe that condition 2 in the lemma statement, implies that $m_1 \prec_{T_o} m_2$.

For every message m' in t_1 or t_3, pairs of the form (m_1,m') and (m_2,m') are ordered the same in M_T as they are in $M_{T'}$. Thus

$$\langle \forall m' \in t_1 \cup t_3 :: m'.lost_T = m'.lost_{T'} \rangle \tag{1}$$

Consider the set $m'.lost$ for a message m' in t_2. At first glance, it appears it is possible for this set to be larger in T' than it is in T, because while $m_1 \notin m'.lost_T$, it is possible that $m_1 \in m'.lost_{T'}$. However, when this is the case, we can deduce the following:

$$\begin{array}{cl} & m_1 \in m'.lost_{T'} \\ \Rightarrow & \{ \text{ Definition of m.lost } \} \\ & m' \prec_{T_o} m_1 \\ \Rightarrow & \{ \ (m_1 \prec_{T_o} m_2) \text{ from lemma condition 2 } \} \\ & m' \prec_{T_o} m_2 \\ \equiv & \{ \ (m' \prec_T m_2) \text{ and Definition of m.lost } \} \\ & m_2 \in m'.lost_T \end{array}$$

Thus,

$$\langle \forall m' \in t_2 :: |m'.lost_{T'}| \leq |m'.lost_T| \rangle \tag{2}$$

Now we must consider $m_1.lost$ and $m_2.lost$. $m_2.lost$ will possibly be smaller in T than in T', because for each message m' in t_2, it is possible that $m' \in m_2.lost_{T'}$, but we know $m' \notin m_2.lost_T$, because $m_2 \not\prec_T m'$. However, for every such m', we know the following:

$$\begin{array}{cl} & m' \in m_2.lost_{T'} \\ \equiv & \{ \ (m_2 \prec_{T'} m') \text{ and Definition of m.lost } \} \\ & m_2 \prec_{T_o} m' \\ \Rightarrow & \{ \ (m_1 \prec_{T_o} m_2) \text{ from lemma condition 2 } \} \\ & m_1 \prec_{T_o} m' \\ \equiv & \{ \ (m_1 \prec_T m') \text{ and Definition of m.lost } \} \\ & m' \in m_1.lost_T \end{array}$$

Also, $T.lost$ contains one additional pair, because $m_2 \in m_1.lost_T$, and $m_1 \notin m_2.lost_{T'}$. Thus,

$$|m_1.lost_{T'}| + |m_2.lost_{T'}| < |m_1.lost_T| + |m_2.lost_T| \tag{3}$$

From formulae 1, 2, and 3, $|T'.lost| < |T.lost|$. ■

Theorem 6.3 *The Last-First( ) reordering algorithm is optimal.*
Proof: By contradiction.
We prove the stronger condition that Last-First( ) finds the only optimal test run over a wave. Consider the original test run T_o. For the purpose of contradiction, assume there exists an optimal test run T over this wave that differs from T_{lf}, the one created by Last-First( ). In Figure 5, assume that m_1 is the first message in M_T that differs from $M_{T_{lf}}$. Label the receive event that delivers m_1 here r. m_2 from Figure 5 represents the message that T_{lf} delivers at r. m_1 and m_2 are both in $r.avail$. Further, since the Last-First( ) algorithm delivers m_2 at r, we know that $m_1 \prec_{T_o} m_2$, which implies $m_2 \in m_1.lost_T$.

From Lemma 6.2, there exists a test run T' such that $|T'.lost| < |T.lost|$, so T is not optimal: a contradiction. ∎

6.3 Performance Lower Bound

Following the pattern of logic in Lemma 5.6, it is always possible to reorder the message that was delivered by the first receive event in T_o, and deliver it in the very last receive event in T. Doing so reorders this first message with every other message in the wave, successfully reversing $k-1$ pairs. Since the Last-First( ) algorithm is optimal, $k-1$ represents a lower bound on the number of pairs it will reverse. Below we show this to be a tight bound.

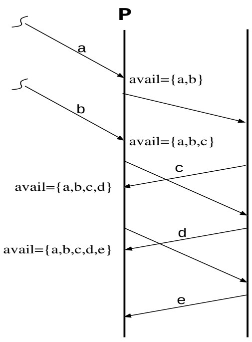

Figure 6: A Worst-Case Wave

Theorem 6.4 *$k-1$ is a tight lower bound on the number of message pairs the Last-First( ) reordering algorithm will be able to reverse in a wave of size k.*
Proof: By example.
Consider the execution shown in Figure 6. For this rather extreme underlying program, our Last-First( ) reordering algorithm will only be able to reverse $k-1$ pairs. It produces the test run $T = bcd...a$, because it chooses any available message before a, but there is always only one other choice in $r.avail - r.recvd$. This test run reverses only $k-1$ pairs. ∎

7 Creating a Test Suite

In this section, we present an algorithm that reverses all message pairs in a wave using the fewest number of test runs possible. Formally, our goal is to construct a *minimal* set of test runs,

$T_1, T_2, ..., T_t$, such that for every pair of messages m_1 and m_2:

1. that are sent to process P,
2. that exist in the same wave, and
3. for which $m_1 \prec_{T_o} m_2$

the following proposition holds:

$$\langle \exists x : 1 \leq x \leq t : m_2 \prec_{T_x} m_1 \rangle.$$

In this section, we present a method for constructing such a minimal set. We call this set the test suite, and denote it T^*.

If no funnel exists, then all messages in the wave are included in $r_1.avail$. Only a single test run is required to reverse all pairs, using the Last-First( ) algorithm from Section 6.1. Assuming the wave contains at least one funnel, the following lower bound exists on the size of T^*.

Lemma 7.1 *100% message reordering coverage over a wave requires* **at least** $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$ *test runs.*

Proof:

From Definition 5.8 (the definition of F_c) and Lemma 5.7. ■

7.1 Multiple Test Run Algorithm

In this section, we present an algorithm for calculating a minimal T^*. Our method constructs one test run at a time: T_1, then T_2, and so forth. Each wave can be evaluated independently, so we discuss reordering the pairs within a given wave. The size of T^* is determined by the wave that requires the most test runs to reverse all its message pairs.

During the construction of T^*, we maintain a set of messages, $m.after$, for each message in the wave. While constructing test run T_x, for message m in wave W, $m.after$ is the set:

$$\{m' : \langle \forall y : 1 \leq y < x : m \prec_{T_y} m' \rangle\}.$$

This set contains the messages that were delivered after m in every previous test run. If $m.after = \emptyset$, then m has been delivered after every other message in the wave during some test run. Thus, our algorithm continues to add test runs to T^* until $m.after = \emptyset$ for every message. The procedure **Create-Test-Run**( ) in Figure 7 is called to calculate each test run, T_x.

To construct T_x, **Create-Test-Run**( ) visits each group in order of the execution sequence. For each group, G_i, it first calls **Pass-Through**( ), which chooses the $F_i.tp$ messages that will be passed through funnel F_i. Then, it calls **Deliver**( ), which delivers the remaining available messages to the receive events in $G_i.R$.

To determine which messages to pass through a funnel, **Pass-Through**( ) uses the pass through relation, $[PT]$.

Definition 7.2 *For every pair of messages (m, m') in the wave, where $m \in G$ and $m' \in G'$, define the relation $m[PT]m'$ as the disjunction of the following predicates:*

```
SequenceOfMsgs Create-Test-Run( )
{
  SequenceOfMsgs TestRun := ⟨ ⟩;
  SetOfMsgs ChooseFrom := ∅;
  foreach (Group G_i)
  {
    ChooseFrom := ChooseFrom ⋃ G_i;
    SetOfMsgs PassThrough := Pass-Through(F_i, ChooseFrom);
    SequenceOfMsgs Seq := Deliver(ChooseFrom);
    TestRun := TestRun.Seq; //Concatenation
    //Here, ChooseFrom = ∅
    ChooseFrom := PassThrough;
  }
  return TestRun;
}
```

Figure 7: Procedure Create-Test-Run( )

1. $(m.after \neq \emptyset) \wedge (m'.after = \emptyset)$

2. $((m.after = \emptyset) \equiv (m'.after = \emptyset)) \ \wedge$
   $(G \prec G')$

3. $((m.after = \emptyset) \equiv (m'.after = \emptyset)) \ \wedge$
   $(G = G') \ \wedge$
   $(m' \in m.after)$ $\square$

If $m[PT]m'$, then we want to pass m through a funnel more than m'. **Pass-Through**( ) maps $[PT]$ over the messages in the set $(G_i.Avail - G_i.r_f.recvd)$. Then, it removes $F_i.tp$ messages from the set, each one at the top of a chain formed by $[PT]$ over those messages that remain. These removed messages will be passed through F_i.

```
SetOfMsgs Pass-Through(Funnel F, SetOfMsgs ChooseFrom)
{
  SetOfMsgs PassThrough := ∅;
  while (|PassThrough| < F.tp)
  {
    Msg m := any message from ChooseFrom
    foreach (Msg m' ∈ ChooseFrom)
    {
      if (m'[PT]m)
      {
        m := m';
      }
    }
    PassThrough := PassThrough ⋃ {m};
    ChooseFrom := ChooseFrom − {m};
  }
  return PassThrough;
}
```

Figure 8: Procedure Pass-Through( )

To determine how to order the delivery of messages within some group G_i, **Deliver**( ) uses the simpler relation, $[D]$.

Definition 7.3 *For every pair of messages (m, m') in the wave:*

$$m[D]m' \triangleq m' \in m.after \quad \square$$

Consider two messages, m and m', where $m' \in m.after$. During some test run $T_x \in T^*$, we need to remove m' from $m.after$ by delivering the two messages such that $m' \prec_{T_x} m$. **Deliver**( ) visits the receive events in $G_i.R$ in order. When choosing between m and m' for delivery by a receive event, where $m[D]m'$, it delivers m' and leaves m in the set $ChooseFrom$ for delivery by a later receive event. This strategy is similar to that used by Last-First( ) in Section 6.

```
SequenceOfMsgs Deliver(SetOfMsgs ChooseFrom)
{
  SequenceOfMsgs Order := ⟨ ⟩;
  while (ChooseFrom ≠ ∅)
  {
    Msg m := any message from ChooseFrom
    foreach (Msg m' ∈ ChooseFrom)
    {
      if (m[D]m')
      {
        m := m';
      }
    }
    Order := Order.m; //Concatenated
    ChooseFrom := ChooseFrom − {m};
  }
  return Order;
}
```

Figure 9: Procedure Deliver( )

7.2 Algorithm Analysis

In this section, we prove that our algorithm calculates a minimal T^*.

Definition 7.4 *For a group G_i, consider the set of funnels $\{F_j : G_i \preccurlyeq G_j\}$. Define $G_i.bn$ as the funnel from this set with the smallest throughput.*

The funnel $G_i.bn$ is the **bottleneck** for G_i. Later, we will see that this bottleneck defines an upper bound on the number of test runs required by our algorithm to reduce $m.after$ for every message in group G_i to $\emptyset$. For now, we note that whenever there are at least $G_i.bn.tp$ messages in G_i with $m.after \neq \emptyset$, $G_i.bn.tp$ messages from this set will be passed through all funnels in the wave.

Lemma 7.5 *While constructing test run T_x, consider group G_i.*

Let $S = \{m : m \in G_i.Avail_W \wedge m.after \neq \emptyset\}$ Let $R = \{m : m \in G_i \wedge m.after \neq \emptyset\}$

Either $G_i.bn.tp$ messages from S are passed through all future funnels, $\langle F_i, F_{i+1}, ..., F_{last}\rangle$, or **every** *message in R is passed through all future funnels.*

Proof: By induction over the sequence of funnels.

Base: Funnel F_i. There are two cases:

Case 1: There are at least $G_i.bn.tp$ messages from S still available. This means that at least $G_i.bn.tp - |R|$ messages with $m.after \neq \emptyset$ were passed through funnel F_{i-1} by **Pass-Through( )**. We will show that $G_i.bn.tp$ messages from S are passed through F_i. Suppose that this is not the case. Then some message m' from S was passed to our **Pass-Through( )** procedure and not included in the returned set. Also, since $F_i.tp \geq G_i.bn.tp$, at least one message m'', for which $m''.after = \emptyset$ **was** returned by **Pass-Through( )**. This is a contradiction, because **Pass-Through( )** will include m' in the return set before m'' in accordance with the $[PT]$ relation.

Case 2: Fewer than $G_i.bn.tp$ messages from S are available. In this case, every message in R is passed trough F_i. The proof is similar to **Case 1**.

Step: F_j, for $i < j \leq last$.
By hypothesis, either at least $G_i.bn.tp$ messages from S were passed through F_{j-1}, or all messages in R were.

Case 1: At least $G_i.bn.tp$ massages from S were passed through F_{j-1}. In this case, we show that at least $G_i.bn.tp$ messages from S are passed through F_j. Assume this is not the case. Then some message $m' \in S$ is passed to **Pass-Through( )** and not returned. Also, some message $m'' \notin S$ **is** returned by **Pass-Through( )** for funnel F_j. Since $m'' \notin S$, it is either the case that $m''.after = \emptyset$, or $m'' \in G_k \wedge G_i \prec G_k$. In either case, the $[PT]$ relation causes **Pass-Through( )** to include m' in its return set before m'': a contradiction.

Case 2: All messages in R were passed through F_{j-1}. If $|R| > G_i.bn.tp$, then **Case 1** applies. Otherwise, all messages in R are passed through F_j. The proof is similar to **Case 1**.

■

The following lemma uses the properties of $[D]$ to show that once a message is passed through all funnels in the wave, its $m.after$ set is reduced to $\emptyset$.

Lemma 7.6 *If m is passed through every funnel in T_x, then* $\langle \forall m' \in m.after :: m' \prec_{T_x} m \rangle$
Proof: By contradiction.
Suppose that m is passed through every funnel in T_x, $m' \in m.after$, and $m \prec_{T_x} m'$. Then m' was also passed through every funnel, and $m' \in m.r_{T_x}.raceset$. But this observation leads to the contradiction that Deliver( ) will deliver m' instead of m, because $m[D]m'$. ■

By combining the results from Lemmas 7.5 and 7.6, we can obtain an upper bound on the number of test runs our algorithm will require to reduce a given message's $m.after$ set to $\emptyset$.

Lemma 7.7 *Consider message m_x in group G_i. Within $\left\lceil \frac{|G_i.Avail_W|}{G_i.bn.tp} \right\rceil$ test runs, $m_x.after$ will be reduced to $\emptyset$.*
Proof:
From Lemma 7.5, we know that $G_i.bn.tp$ messages from $G_i.Avail_W$ with $m.after \neq \emptyset$ will be passed through all funnels during any test run in which m_x is not. Then, from Lemma 7.6, we know that $m.after$ is reduced to $\emptyset$ for each of these messages. Again from Lemma 7.5, once there are fewer than $G_i.bn.tp$ messages in $G_i.Avail_W$ with $m.after \neq \emptyset$, m_x will have to be passed through all funnels, and $m_x.after$ will consequently be reduced to $\emptyset$. In this worst case scenario, m_x is the last message from $G_i.Avail_W$ to have its $m.after$ set reduced to $\emptyset$. This scenario requires $\left\lceil \frac{|G_i.Avail_W|}{G_i.bn.tp} \right\rceil$ test runs. ■

And finally, if we find the message(s) for which this upper bound is the largest, we know the worst-case largest size of T^*, as calculated by our algorithm. Upon comparing this upper bound with the lower bound from Lemma 7.1, we find that they are the same.

Theorem 7.8 *The reordering algorithm reverses all message pairs in* $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$ *test runs.*
Proof: Lemma 7.7 gives an expression for the worst-case number of test runs our reordering algorithm will require to reverse any given message m with all others in the wave. This expression is maximized by selecting a message from the group corresponding to the funnel F_c. The resulting expression is $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$. From Lemma 7.1, this expression is also a lower bound on the number of test runs required to achieve 100% message reordering coverage. Thus, our reordering algorithm optimally requires exactly $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$ test runs to reorder all message pairs. ■

8 Conclusion

We have presented an algorithm to reorder as many pairs as possible during a single test run, and another to reorder all pairs in as few test runs as possible.

The first area of research has yielded the Last-First( ) algorithm for reordering message deliveries in a test computation. We have proven that it is optimal, and given a tight lower bound of $(k-1)$ reversed pairs in a wave of size k. Questions remain that suggest opportunity for future research in this area. For example, the lower bound of $(k-1)$ is relatively low, but demonstrated by a particularly contrived underlying computation. We suspect that this bound is not indicative of performance for most real-world programs.

In Section 7, we presented an algorithm for calculating a minimal set of test runs to reverse all message pairs. The number of test runs required to achieve this 100% coverage, $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$, can be determined after the first test run. Although our model of a distributed system does not assume FIFO channels, this algorithm can be easily extended to accommodate such a system. The proofs in Section 7.2 are a little more complicated, but the performance of the extended algorithm is the same.

An interesting question is whether the techniques used in Sections 6 and 7 can be modified to allow testing programs that do not enforce a fixed message set. Perhaps a weaker requirement can be found.

Additionally, we intend to investigate alternate methods for reducing a distributed program's execution space to searchable subspaces. To evaluate any alternatives, we will also need a way to evaluate their abilities in exploiting software defects. Such an effort should most likely involve development of a prototype testing system for experimental analysis of their effectiveness. A taxonomy for software defects created and/or hidden by racing messages might also be useful to this end.

References

[1] H. Agrawal, R. DeMillo, and E. Spafford. Dynamic slicing in the presence of pointers. In *ACM Symposium on Testing, Analysis, and Verification*, 1991.

[2] H. Agrawal, R. DeMillo, and E. Spafford. An execution backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[3] H. Agrawal, R. DeMillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–619, June 1993.

[4] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, K. Moore, P. Newton, and V. Sunderam. Hence: A users' guide - version 2.0. Pvm Design and Visualization.

[5] D. Callahan and J. Subhlok. Static analysis for low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[6] J-D Choi, B. P. Miller, and H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[7] S. Feldman and C. Brown. IGOR: a system for program debugging via reversible execution. *SIGPLAN Notices: Workshop on Parallel and Distributed Debugging*, 24(1):112–123, May 1988.

[8] M. G. Fernandez and S. Ghosh. Ddbx-lpp: A dynamic software tool for debugging asynchronous distributed algorithms on loosely coupled parallel processors. *Journal of Systems and Software*, 22(1):27–43, July 1993.

[9] C. J. Fidge. Partial orders for parallel debugging. *SIGPLAN Notices: Workshop on Parallel and Distributed Debugging*, 24(1):183–194, January 1989.

[10] G. C. Fox. *Solving problems on concurrent processors.* Prentice Hall, 1988.

[11] O. Gerstel, M. Hurfin, N. Plouzeau, M. Raynal, and S. Zaks. On-the-fly replay: A practical paradigm and its implementation for distributed debugging. In *Proceedings, Symposium on Parallel and Distributed Processing*, pages 266–272, October 1994.

[12] S. Grabner and J. Volkert. Debugging parallel programs using event graph manipulation. In *International Meeting on Vector and Parallel Processing*, pages 443–450, October 1993.

[13] R. Kilgore and C. Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. In *30th Hawaii International Conference on System Sciences*, 1997.

[14] R. Kilgore and C. Chase. Testing distributed programs containing racing messages. *to appear in The Computer Journal*, November 1997.

[15] J. Kohl. XPvm. http://www.netlib.org/utk/icl/xpvm/ xpvm.html.

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

[17] T. J. LeBlanc and J. M. Mellor-Crummley. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[18] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[19] R. Netzer, T. Brennan, and S. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Symposium on Parallel and Distributed Tools*, pages 31–40, May 1996.

[20] R. Netzer and B. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, August 1990.

[21] R. Netzer and B. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings, Supercomputing '92*, pages 502–511, November 1992.

[22] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. *SIGPLAN Notices: 1988 Workshop on Parallel and Distributed Debugging*, 24(1):124–129, January 1989.

[23] D. A. Reed. An overview of the pablo performance analysis environment. In *Technical Report, University of Illinois, Urbana, Illinois*, November 1992.

[24] R. S. Side and G. C. Shoja. A debugger for distributed programs. *Software - Practice and Experience*, 24(5):507–525, May 1994.