Testing Reliable Distributed Applications Through Simulated Events

Travis L. Winfrey and Gail E. Kaiser

Columbia University
Department of Computer Science
New York, NY 10027

CUCS-427-89

April 7, 1989

Abstract

There are many distributed applications that incorporate application-specific reliability algorithms which operate on top of general purpose networking, operating system and programming language facilities. We present a framework for application-level reliability testing suitable for a wide range of distributed applications, and describe how we've applied it to one particular application, Mercury, a distributed, multi-user programming environment.

Copyright © 1989

Winfrey is supported in part by the Center for Advanced Technology. Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

Keywords: **distributed applications, test generation, simulated load**

Table of Contents

1. Introduction 1
2. Testing through Simulating Events 3
    2.1. Overview of Testing Methodology 3
    2.2. Simulation of Events 6
    2.3. Scripts 7
        2.3.1. Events 10
        2.3.2. Application-Independent Predicates 11
        2.3.3. Action-independent Actions 14
3. A Specific Application 16
    3.1. Application-Specific Conditions 17
    3.2. Application-Specific Actions 18
    3.3. Pragmatics of Script Execution 20
4. Script Generation 21
    4.1. Associating Properties with Processes 21
    4.2. Specifying Timing of the Generated Scripts 22
    4.3. Specifying Other Parameters to SG 23
5. Conclusions 23
Acknowledgements 26
References 26

1. Introduction

We are interested in devising methodologies for testing reliable distributed applications. The method we describe here is applicable in the testing and debugging of essentially any message-passing application. The test data sets may be written by an implementor or generated from the protocols themselves. Because these test data sets, whether generated or hand-written, are not necessarily tied to any particular implementation, they may be used to aid in the debugging of a program as it gradually evolves towards a working version. They may also be used in comparing separately-designed, competing implementations of the same algorithm. Finally, this method is well-suited for performance testing, both under normal conditions and under abnormal conditions which may be difficult to artificially create during development.

The need for standardized testing is well-known among designers of protocols and distributed systems, but general methods are currently unknown [Sarikaya 88, Haban 87]. Testing reliable, distributed applications is necessary and important — and more difficult than testing ordinary sequential applications. This is true for a variety of reasons:

- The implementation of an algorithm is not the algorithm itself. A proven-correct algorithm of any nature will not necessarily be coded well or accurately [Beizer 83, Myers 79]. Also, the *raison d'être* of reliable systems is coping with failure, which may be difficult or impossible to create in such a way as to exercise a sufficient number of paths through the code. Code which is intended to provide reasonable responses on failure may not be tested at all, simply because the events for which they are written do not frequently happen.
- An application may satisfy its specifications without being usable in a real-world environment. It is therefore necessary to test applications under all possible load conditions. However, it may not be feasible to create a high load with an application through its ordinary use. This difficulty became apparent in our work on distributed programming environments. It was not possible to conduct experiments with large numbers of users, and single users could not create a high load by themselves. Similarly, it may be difficult or impossible to show a program's behavior during system-wide or net-wide degradation.
- Simplifying assumptions are present in many proofs of reliable algorithms [Halpern 86, Wittie 87]. These assumptions may invalidate an implementation's correctness or utility in real world situations. Here is a partial list of common assumptions:
    - insignificant processing time
    - insignificant transmission time
    - unlimited storage, e.g., ingoing or outgoing message queues, number and size of messages
    - no lost messages
    - no corrupted messages

• available services, e.g., synchronous send and receive

These assumptions, implicit or explicit, may provide useful simplifications while solving the problem, but programs will always be run in situations for which they are explicitly not designed; this is in the nature of programming. It is important to find the boundaries of failure during the construction of such applications.

Ultimately, however, the most important design assumption is that the software itself will be reliable while the hardware will be unreliable. While this may have previously been true, the balance has shifted so that hardware is now significantly more reliable than software. Software failures are frequently more evident than hardware failures, with an error rate as high as one bug per hundred statements [Beizer 83].

In summary, an untested implementation of a reliable distributed application is not reliable, no matter what the underlying algorithms are. An implementation which has been tested, even in an *ad hoc* fashion, will be considerably more reliable. In this paper, we describe a framework for the testing of application-level facilities, and a method of generating test sets from abstract descriptions of desired testing situations. We have sought to identify all failure modes common to message-passing distributed applications, and create a system that can simulate them. We emphasize that we are not simulating *systems*, we are simulating events which can occur in the *environment* of such a system. This simulation may be easily achieved in message-passing systems, and it offers a significant validity check of a distributed algorithm. Briefly, the techniques devised here will cause all possible state transitions to occur in a finite-state machine representing a process, thus exercising all code directly related to a distributed algorithm. This testing cannot, in general, find all errors present in a program. However, it is reasonable to claim that a complete test of a distributed program will include data that will also cause all state transitions, in addition to other tests, and that this method can find a large class of these errors. Finally, although some of this work is relevant, we do not address testing layered architecture protocols, such as the ISO-OSI reference model [Zimmerman 80, Rudin 87]. This paper is restricted to a discussion of testing applications without reference to other layers in a network architecture.

This paper is structured as follows. In section 2, we define our testing methodology as the simulation of events presented by an environment to a program. Later, we refine the definition of events, and show how they may be effectively simulated in an application-independent manner. We present *scripts* as a rule-based method of dynamically changing a process' responses to its environment, effectively testing *other* communicating processes. Section 3 discusses the use of application-specific scripts for testing the Mercury distributed programming environment [Kaiser 87a].

By themselves, scripts may be used to validate an application under a specific range of conditions. They may be also be used to create certain conditions while debugging an application. In both cases, the scripts would be written by the programmer. However, it is useful to consider the automatic generation of scripts. Our preliminary work has been with generating scripts based on abstract specifications of testing situations. This is discussed in detail in section 4.

2. Testing through Simulating Events

2.1. Overview of Testing Methodology

Before discussing our testing methodology directly, it is profitable to consider the *external view* of a perfectly-tested distributed application. Given that programs are designed with a number of constraints, a well-tested application will be run successfully over all possible combinations of the constraints. Here are some likely constraints for a reliable distributed program:

- host failure
- network partition
- message-sending rate
- message-missing rate
- data-corruption rate
- data-duplication rate
- message-processing time
- message-transmission time
- data size
- queue size

More precisely, since there will be limits in the real world on each of these parameters, any application will be designed to be correct within a certain range for each parameter. In examining the above list, it is clear that these parameters are effectively input data for a distributed program, in the same manner that a *sine* function takes floating-point variables as its input data. In analogy with functional testing [Howden 87], to test a distributed program properly, it is necessary to vary these parameters.

However, because these parameters reflect external events, many of them are difficult to vary at all, while others prove difficult to vary with any precision. For instance, it may not be possible to crash and reboot a machine within a certain interval, which may in turn cause certain code to go unexecuted during the entire life-cycle of a program. Methods as crude as deliberately crashing an entire machine frequently have other real-world repercussions, such as the sudden termination by others of this line of experiments. Less obtrusive methods, such as deliberately crashing an individual process, are not always effective, because it is difficult to replicate the exact timing of the event during repeated debugging runs.

In addition to experimental difficulties, testing should provide *adequate* coverage of the program with respect to selected criteria [Weyuker 88]. Overall, there are two orthogonal selection criteria for test data sets [Howden 87]: they may be chosen in order to test the functionality of a program, according to its specifications, or they may be chosen to test the implementation of a program, based on the examination of its source code.

We propose a method of functionally testing a program in terms of its underlying Finite-State Machine (FSM) structure. Such methods of testing have been explored in theory and practice by other researchers [Chow 78, Bauer 79], and they show particular promise for testing distributed algorithms, which are frequently implemented at some level in terms of FSMs, or in rule-based structures which map directly to

FSMs [West 78, Gouda 84]. Testing coverage of all paths through an FSM's operations will provide a increased level of confidence in an application's correctness, while not completely testing the application itself. Advantages and disadvantages of this method of testing will be discussed in more detail later in this section.

Message-passing applications are naturally suited for this type of debugging, for reasons that we will explain. In a modular system, all ingoing and outgoing messages will pass through a few — frequently only two, **send** and **receive** — routines which handle their processing. A running process will undergo various changes in state because of information passed through these portals. Other internal changes may occur, such as a state change when a timer expires. Allowing these state changes is not an exception, but part of the proper testing of the FSM model. Because the information handled by a distributed program will pass through these few choke-points, it becomes an easy matter to control what a particular process sends and receives, and hence, what it sees and what other processes in communication with it see.[1] For example, a process which neither sends nor receives messages may appear to be non-functioning to any process attempting to communicate with it. This is not generally true: side-effects are possible, such as writing a file on a shared disk, which may invalidate the appearance of being down. "Pure" programs that communicate only by means of a well-defined protocol will be the easiest to deceive by changing the various **send** and **receive** routines.

According to the previous discussion, we may instrument a program for testing simply by changing the **send** and **receive** primitives into user-level interfaces, and by providing a means of controlling the behavior of these interfaces. In other words, the primary intent of this method of testing is to change the functionality of these two routines so that their *failure* is under the tester's control. Although simply implemented, this technique is powerful enough to allow complete testing of the part of a program's control structure devoted to implementing a protocol. Because transitions in the finite-state machine underlying an algorithm will only occur either when a message is sent or received or when a timeout occurs, all possible transitions in the FSM can be forced by controlling the **send** and **receive** operations and by allowing operations to be performed at arbitrary times. While not conclusive, this permits testing that strongly validates the protocol-related code empirically.

This is a novel approach, one which enables us to break away from application-dependent methods of testing. We're not generating test sets as they are commonly understood. No input data in the ordinary sense will necessarily be given to any

---

[1]We assume for the duration of this discussion that there are the only two routines, **send** and **receive**. We are only concerning ourselves with the sending and receiving functionality, not with any particular form in which they may appear.

process which is being tested.[2] Instead, test sets in the form of a script which indicates which actions will succeed or fail, create a "crippling harness" which the program must wear during its ordinary processing. Like blinders on a horse, the test sets force a process to see only what it has been told to see.

Because message-passing processes are in communication with each other, what one process does or does not do affects other processes' actions. By obtaining control over how one process sends and receives messages, one gains implicit control over the control-flow in all other processes communicating with it. Importantly, these other processes need not be instrumented for testing. Any operational changes which may be accidently introduced by the changes in **send** and **receive** in one process will not be present in another process, which responds to messages normally.

Clearly, this method offers a number of advantages to the tester or debugger. To summarize:

- It uses actual implemented code, but sidesteps problems of modifying functionality. Even when only one process is instrumented for debugging, significant testing can still be performed.
- Changes are made at the application level, not at the kernel or hardware level.
- It applies to any message-passing application.
- Arbitrary combinations of event sequences are repeatable, aiding debugging as well as validation. "Events" are defined later in this paper.
- Test sets may be generated from protocol specifications, providing higher-level validation of an algorithm.
- Test sets may be used with different implementations, or different versions of the same implementation.

A number of disadvantages are inherent in this method as well.

- When a protocol is *not* based on Finite-State Machines, the degree to which this method can perform useful testing is difficult to determine. It should be stressed again that we do not claim that this method will flush out most, or even many, errors associated with a program. We do claim that a useful amount of testing may be performed by associating given protocols with a script that tests their responses under various conditions.
- Scripts which do not create information are significantly less useful than those that do. Consider the case of a server waiting for a client to ask it something. If the server has been instrumented for testing, it will have to wait, possibly indefinitely, for the client to initiate transactions.

---

[2]None of our application-independent operations actually create information (i.e., send messages). We provide several application-specific methods of creating information, of course. Currently, all the application-independent operations represent different ways blocking information transfer. More information on this may be found in Section 3.

- Many of the benefits from using this method derive from an implicit assumption of a reliable network. For example, in Figure 2-3, the script for process B will only cause the transitions in processes A and B when all messages are correctly received. In test T3, if a single I_1 message from process A *isn't* sent to process B because of a real network failure, the entire transaction will fail when it should have succeeded. Benchmarking systems encounter the same types of problems in attempting to assure useful results. However, the testing or benchmarking of non-reliable systems does not need to consider all possible system failures, while any such failure will affect the testing of a reliable system. It appears that this limitation is inherent in testing reliable distributed systems. For well-defined results, tests must either be repeated a sufficient number of times, or these tests must be performed with reliable message-passing.
- Our preliminary work has not uncovered problems with the granularity of event scheduling, detailed in later sections. However, our experience has been with relatively slow rates of message passing (less than 500 messages/sec, much less than 1 Kbyte/message). We are cautious about the utility of this method for applications that require very high rates of data transfer.
- It may not be possible for application developers to change the semantics of the **send** or **receive** routines at the application level for all processes because of system- or language-specific reasons. We implicitly understand **send** and **receive** as routines called by a process, ignoring the possibility of changing system-calls in an operating system's kernel, or of changing message-passing semantics in a language-kernel.

2.2. Simulation of Events

From the viewpoint of any distributed system, *events* occur during the execution of the programs that make up this environment. Events may be *internal* or *external*. An internal event might be an attempt to send a message to another process; an external event would be the unnoticed crash of that same process. Clearly, external events may occur without any notification of interested parties. Figure 2-1 shows a time frame with a series of events occurring while two message-passing processes on hosts A and B attempt to communicate.

This sequence of events is interesting only in that it is a specific instance of a much larger class of events that any distributed system is intended to handle. From a testing and performance evaluation point of view, it is useful to know if a particular distributed system can, in fact, correctly deal with this and similar groupings of events. Accordingly, it is useful to be able to *cause* particular groupings of events. At the initial, debugging stage of development, these events may be artificially created on a coarse scale, e.g., processes can be stopped and started by the programmer. Quickly, however, it becomes necessary to acquire a *finer-grain* of control over the sequence and nature of events.

Fine-grain control means that a testing system that induces events will need closer timing to be able to reasonably exercise all transition paths in the FSM. For example,

Key:
⊗ is a message
⊗→B is a host sending a message to B
A→⊗ is a host receiving a message from A
A– is a host locally marked as down
A+ is a host locally marked as up

A crashes   A reboots   B is up

A   ⊗→*B*   ⊗→*B*   *B*→⊗ *B*+

B   *A*→⊗*A*+⊗→*A* ⊗→*A*⊗→*A*   *A*– *A*→⊗   *A*+⊗→*A*

B retries   A marked down   A is up

t   *t*+1 ...

Figure 2-1: One host crashes and reboots while another tries to send messages

manually starting, stopping, and restarting a particular process will causes processes communicating with it to go through some changes, but this technique by itself cannot hope to test all possible states in the other processes.

Let us consider a simple example of one process forcing transitions in an another. In figure 2-2, the transitions of a Finite-State Machine are shown as a directed acyclic graph representing part of a two-part transaction, where unacknowledged messages are re-sent up to three times before the transaction fails. Figure 2-3 shows 8 tests which will force all possible transitions of the FSM in A when process B follows the given script. Obviously, other real-world situations could arise — e.g., corrupted, late, or duplicate messages — which are not addressed in this example, although scripts simulating these situations are possible. Also, no script for A is shown, yet A has decided to send messages I_1 and I_2 to B. A may have done so because of a driving script, or because of other factors, e.g., I_1 and I_2 are generated directly by the application.

2.3. Scripts

Scripts are **the means** of specifying when and how a process should alter the behavior of its **send and receive** primitives. As described here, scripts are a low-level command language that a particular tester might use to create the desired behavior in the program. While individual scripts may be written by hand, scripts may also be produced by a **S**cript **G**enerator, **SG**, as we will see in section 4.

Scripts are read by a portion of the application which we will call **SE**, for **S**cript **E**ngine. **SE** is conceptually one entity, but in practice, may be spread throughout all modules that perform sending and receiving. It is useful to consider **SE** as a separate thread of control in a program, whether or not it is actually implemented that way.

Key:
I_1, I_2 are messages
A_1, A_2 are acknowledgements
S is a successful transaction
F is a failed transaction
T is a timeout

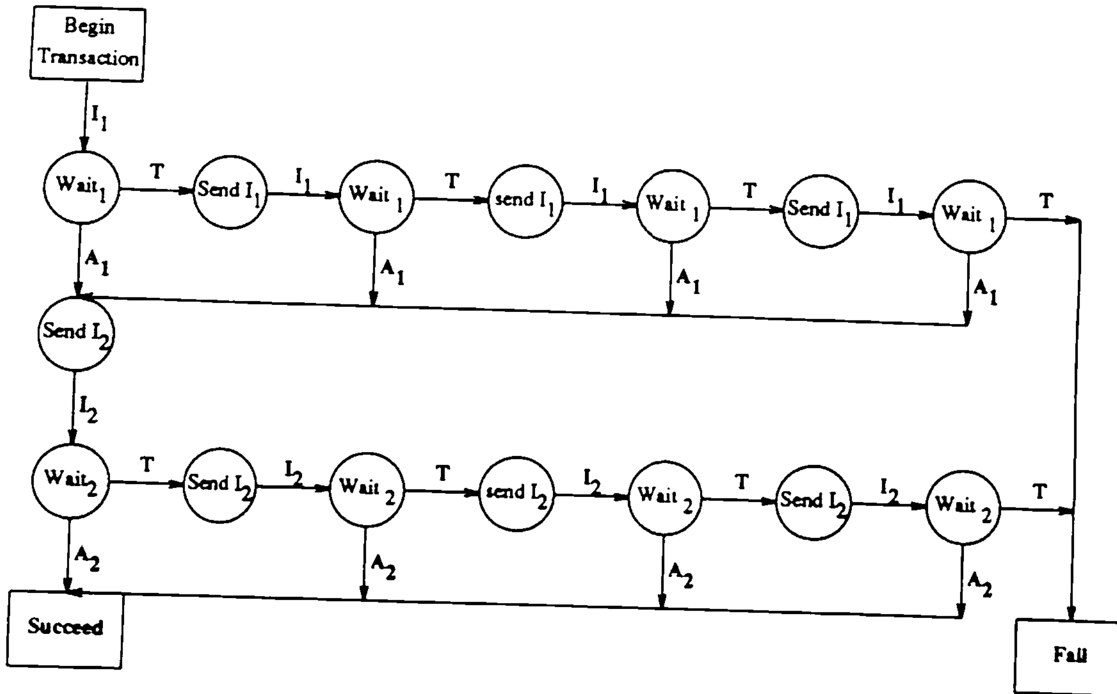

Figure 2-2: A two-part transaction, with up to three retries on failure

Scripts are stored in some internal format, and executed as described below by **SE**. Applications will read scripts at arbitrary times, which may or may not affect the script's actions. Any necessary synchronization can be handled by the script, using the predicates described later. The same script may be read by multiple instantiations of the same process. All commands in a script can apply to all processes, or commands in a script may be limited to a particular sets of processes.

Scripts consist of *events*, *predicates*, and *actions*. Scripts are parsed and stored as *events*. Events (section 2.3.1) are pairs of *predicates* and *actions*.[3] **SE** sweeps through the list of events, testing each one to see if it should run or not. Any number of events may "fire," or be eligible to execute, at the same time. If this happens, ordering on the execution of events is imposed by **SE** according to the given actions, as described in later sections. Once a script is read and stored internally as events, all

[3]The description of events resembles certain types of rule-based protocol specifications [Mackert 87]. No use is currently made of this similarity, but it does suggest later work.

Change in Key:
I_{1R} is a received message
A_{1S} is an acknowledgement sent

Test	*Process A*	*Process B*	*Process B Script*
T0	$I_{1S}, A_{1R}, I_{2S}, A_{2R}$, **S**	$I_{1R}, A_{1S}, I_{2R}, A_{2S}$	*<nothing>*
T1	$I_{1S}, I_{1S}, A_{1R}, I_{2S}, A_{2R}$, **S**	$I_{1R}, A_{1S}, I_{2R}, A_{2S}$	**ignore 1 message**
T2	$I_{1S}, I_{1S}, I_{1S}, A_{1R}, I_{2S}, A_{2R}$, **S**	$I_{1R}, A_{1S}, I_{2R}, A_{2S}$	**ignore 2 messages**
T3	$I_{1S}, I_{1S}, I_{1S}, I_{1S}, A_{1R}, I_{2S}, A_{2R}$, **S**	$I_{1R}, A_{1S}, I_{2R}, A_{2S}$	**ignore 3 messages**
T4	$I_{1S}, I_{1S}, I_{1S}, I_{1S}$, **F**	*<nothing>*	**ignore 4 messages**
T5	$I_{1S}, A_{1R}, I_{2S}, I_{2S}, A_{2R}$, **S**	$I_{1R}, A_{1S}, I_{2R}, A_{2S}$	**when 1 received, drop 1 msg**
T6	$I_{1S}, A_{1R}, I_{2S}, I_{2S}, I_{2S}, A_{2R}$, **S**	$I_{1R}, A_{1S}, I_{2R}, A_{2S}$	**when 1 received, drop 2 msgs**
T7	$I_{1S}, A_{1R}, I_{2S}, I_{2S}, I_{2S}, I_{2S}, A_{2R}$, **S**	$I_{1R}, A_{1S}, I_{2R}, A_{2S}$	**when 1 received, drop 3 msgs**
T8	$I_{1S}, A_{1R}, I_{2S}, I_{2S}, I_{2S}, I_{2S}$, **F**	*<nothing>*	**when 1 received, drop 4 msgs**

Figure 2-3: 8 different tests, the transitions, and the driving script

events will remain in a process, eligible for execution, until they are removed. Events will be removed according to their predicates. *Predicates* (section 2.3.2) are objects consisting of groups of conditions, each of which may test as true or false, according to the current state of the system. For instance, a predicate may test the time of day, or how many messages have been received from a certain host. *Actions* (section 2.3.3) are objects which store pairs of *verbs* and *application-objects*, which are objects in the system being tested, such as a message queue. Actions are performed only when their associated predicate is true.

Figure 2-4 is a sample script that describes the schedule of events for two machines. It contains commands for both machines, each of which will parse and execute its own commands, ignoring the others. Commands outside of a begin/end pair are always executed. In this case, no information is created by the script; it does not actually cause any messages to be sent. These events are overlaid onto the normal processing of the system. While writing a script, it can be assumed that the applications on the two machines will be given this script at more or less the same time, but scripts may be written so that this assumption need not be true, e.g., by waiting for a single message before doing anything, or by waiting for a certain time.

Scripts can be useful in many development situations when simultaneous work by multiple developers could interfere with one another. Figure 2-5 shows an example script. Nine workstations are identified by name and divided among three teams (Hobbit, Orleans, and River). If all processes read this script upon startup, then each group of machines will be partitioned so that they may only pass messages between

```
# basic.scr
# this script is executed at the same time by two
# processes on hosts gollum and frodo

timestamp "executing basic.scr" # executed on
                                # gollum and frodo

begin gollum
    timestamp "gollum pretends to be flaky"

    when message received 2     # after a couple of minutes
        crash                   # stop processing
    when message received 2 and + 0:00:10
        boot                    # wait 10 seconds
                                # before returning
                                # every 100 messages
    whenever (messages-received % 100) == 0
        drop 1 message          # ignore a single message
end gollum

begin frodo
    timestamp "frodo pretends to be slow"

    when messages sent 5        # soon after starting up
        set message-processing-delay 200us
                                # delay 200 millisec/message

    when messages sent 25       # later, get even slower,
        set message-processing-delay 0.5s
                                # always delaying 500us

    when messages sent 50       # after many msgs, randomly
        set message-processing-delay 5s random
                                # take up to 5 seconds

end frodo
```

Figure 2-4: Application-independent script causing two hosts to fail in different ways

themselves. Thus, a runaway process on machine Mojo will only distress members of the Orleans team; it will not be able to contact any process on a machine belonging to the Hobbit or River team.

2.3.1. Events

Events are the entities described in a script. They consist of predicates and actions, and are executed in a manner similar to rule-based systems. Events are kept in a schedule list according to their predicates; they are stored in the order in which their predicates are most likely to test true. If the predicate of the first event will become true at a certain time, then **SE** will run next at that time in order to execute it. However, if there are no predicates which are time-dependent, then **SE** will only wake up when a message is sent or received. If a predicate contains a relative time whose exact meaning cannot be determined, **SE** will simply run at regular intervals.

```
# partition.scr
#
# a debugging script to enable three programming teams to
# work on their communicating programs without adversely
# affecting each other's efforts.

begin gollum, frodo, bilbo          # hosts used by hobbit group
   echo "hobbit group is partitioned from orleans & river group"
   ignore mojo, iko, longhair
   ignore yang-tze, congo, nile
end gollum, frodo, bilbo

begin mojo, iko, longhair           # orleans group
   echo "orleans group is partitioned from river & hobbit group"
   ignore gollum, frodo, bilbo
   ignore yang-tze, congo, nile
end mojo, iko, longhair

begin yang-tze, congo, nile         # river group
   echo "river group is partitioned from orleans & hobbit group"
   ignore mojo, iko, longhair
   ignore gollum, frodo, bilbo
end yang-tze, congo, nile
```

Figure 2-5: Application-independent script for program development

All events which can fire at a given time will do so. They will be sequentially executed, according to the ordering rules given for actions. Allowing all events to fire at one time does not obviate the need for storing the events in the order in which they are most likely to execute, because it is important to know when a predicate is likely to fire, in order to minimize time granularity problems. Details are given in section 3.3.

2.3.2. Application-Independent Predicates

Predicates are logical entities which are tested to be true or false. A predicate is composed of various conditions. All subcomponents of a particular predicate must be true for the entire predicate to be true. A condition might test true after a certain amount of time has passed, or if a certain host is marked as up. A predicate composed of other conditions might test true for all the time *before* the time that a host is marked as up, or test true *while* a specific number of messages have been received. None of the application-independent conditions use any information obtainable from the contents of any message. However, application-*dependent* conditions, section 3.1, may do so.

Conditions for predicates are given in terms of *time contexts* surrounding conditions, which are listed below. Figure 2-6 shows the relation of time contexts to simple predicates on a timeline.

When *condition*

When the condition becomes true, execute the associated action list, then remove the event (the predicate/action pair).

Before *condition*

The associated action list is executed before the condition becomes true. The event is not removed until the condition becomes true. In other words, while the condition is *false*, the associated action will be executed.

After *condition*

This is the opposite of **Before**; it is not equivalent to **When**, its intuitive meaning. The associated action list will be executed, after the time that the condition becomes false. Note that an **After** predicate will not be removed automatically. However, the **clear** action will remove all events.

While *condition*

This has a obvious meaning. Before the condition becomes true, nothing happens. During the period that the condition is true, the associated action list is executed. When the condition becomes false, the event is removed.

Whenever *condition*

Whenever is similar to **while** in that associated actions are executed while the specified condition is true. However, when the condition is false, the event is not removed, but instead becomes *inactive*. This is used for specifying repetitive actions which may be true and false at arbitrary times during execution. **Whenever** will be most useful with expressions that change value many times during the course of execution.

condition **and** *condition*
condition **or** *condition*
not *condition*

All of these boolean operators are interpreted normally. Predicate testing stops when it is clear that an entire predicate will be true or false.

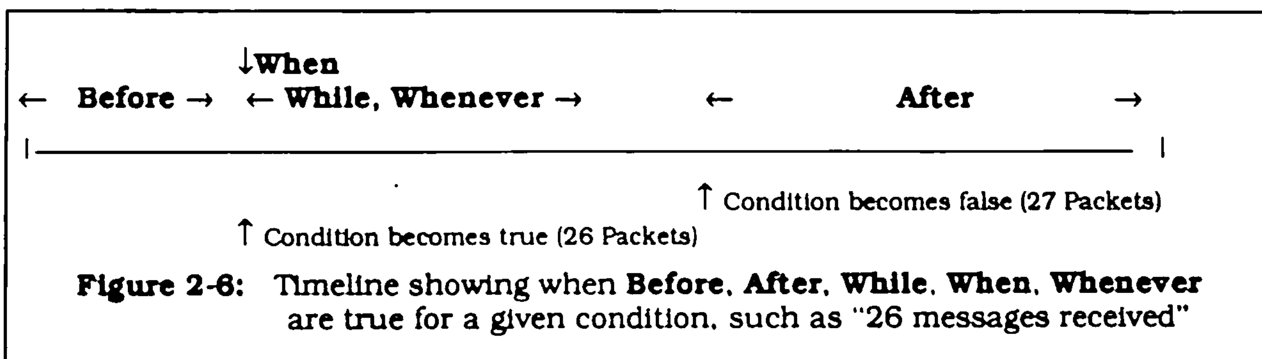

Figure 2-6: Timeline showing when **Before**, **After**, **While**, **When**, **Whenever** are true for a given condition, such as "26 messages received"

Conditions may be thought of as simple predicates. They may be usefully divided into application-specific and application-dependent conditions. The former group describe conditions which may become true for any distributed program which performs message-passing. The latter group describes conditions specific to the APL program, the application that motivated **SE** and **SG**. Conditions may be combined in any order in a single predicate, so that many possible meanings are achievable, e.g., "If host A is up, and we've received 3 messages from Host B, and it's ten minutes after we started — execute the associated action."

Several application-independent predicates are described below. In concept and implementation, they are quite simple, yet powerful enough to describe a very large range of situations because the conditions they test are common to all message-passing systems. Further, a condition such as **Messages Received** *n* permits a better understanding of the current state of another process: while it may be difficult or impossible to precisely determine the real-time of another process' actions, in many situations, testing the number of packets sent to or received from a process can uniquely determine a process' current condition within a subset of possible conditions.

In the following discussion, *entity-list* refers to a list of hostnames or process identifiers. The exact format of these names depends on the operating system. In general, however, hosts may be given by their names, addresses, or the keywords **any** (more than zero), **one**, **some** (more than one), **all** or **none**. Processes may be specified with the same keywords, plus **parent**. Obviously, using OS-specific or site-specific names with this command could make a script site-dependent, limiting its usefulness. **SG**-generated scripts avoid this problem because **SG** contains a **Map** command (section 4.1) to allow for the general renaming of hostnames and process identifiers.

Messages Received *n*

This condition is true when exactly this many messages have come from all possible sources since the last reboot. To use this condition in a comparative sense, the **when**, **while**, etc., commands are used. For example, "**when messages received 25**" can be combined with these commands to mean any of the following: "*before we've received 25 messages,*" "*as soon as we've received 25 messages,*" "*after we've received 25 messages but before we've received 26 messages,*" and "*after we've received at least 25 messages.*"

Messages Sent *n*

This checks messages sent from this process as opposed to messages received by it. The exact meaning of this condition may be problematic with broadcast messages, because the predicate may be supposed to determine the number of messages sent to a particular host or process. The problem arises because when a process on machine A broadcasts a message, it cannot know all places where the message is actually received.

Up *entity-list*

True when a specified group of hosts or processes are marked as up by the current program. This condition does *not* actually attempt to check if the host is up or the process is running. When testing this condition, no reference is made to the number of messages sent from any host or process. That is, a process is looked up in a table kept by the application to see if it is still considered up *by the application*. **Up** and **Down** will not necessarily be implementable with these precise semantics for applications which do not keep such a table.

Down *entity-list*

This condition is the reverse of the previous one.

Hosts *hostlist*
: True when one or more hosts has satisfied all associated conditions such as being up or down, or having sent or received a particular number of messages. If this condition is not paired with any other, then it will only check if the remote host appears to be up. For naming conventions with this and the **process** predicate, see the above discussion of entity-lists.

Process *entity-list*
: **Process** is similar to the **Hosts** command. It applies to individual processes which are running on the same host.

Time *time* This condition becomes true when the specified time is greater than the current time. It may be given as a relative time, which will be relative to the time of parsing of the script which contained this command. The time may be given down to the millisecond, although such precise granularity may not be possible to achieve. The time may also include the date.

expression Expressions are any mathematical expressions involving constants, **SE** variables, a number of arithmetic and logical operators, e.g., **+**, **-**, **%**, **==**, **!=**, and parentheses. The following **SE** variables are representative, but do not exhaust all possibilities. Other variables, such as data-corruption-rate, are also conceivable.

messages-received
: The number of messages received since the last process "reboot" (see the **boot** command in the next section).

messages-sent
: The number of messages sent since the last reboot. In some cases, this may represent the number of times the **send** routine has been called, given the previously discussed problems with broadcast messages.

time-up The time, in microseconds, since the last reboot. This variable is zero when the program begins executing.

time-down The time, in microseconds, since the last **crash** action (described in the next section). This variable is zero until the first **crash**.

2.3.3. Action-independent Actions

The *action* component of an event is performed when the associated predicate has tested true. Actions are verbs; they will always change something in the system. Note that iteration is not an action, but is implicitly contained in the predicate system, with **While**, **Whenever**, and so forth. An action is only removed according to the rules for its associated predicate. In the event that several events are read to be executed at the same time, actions will be executed in the order given in the following list, chosen to minimize conflicts between actions which might countermand each other. The ordering is particularly important with the **Clear** command, because it affects when other events execute.

The following application-independent actions are supported. Note that that all of these actions *block* the passage of messages in some manner. None create information or cause messages to be sent. For actions that force messages to be sent, see the application-specific actions, described in section 3.2.

Boot The application appears to have rebooted. All relevant variables are cleared, and any initialization (such as "online" broadcast messages) is performed.

Boot is useful for synchronizing scripts, as well as for simulating the start of a process.

Crash The application will read, but not respond to, all messages from all sources. Other than **SE**'s continued execution, no other processing will take place. If the application would execute any processing on a normal exit, e.g., deleting temporary files, these actions will *not* take place. The decision to read messages and discard them was based on operating system requirements: a certain number of unread messages would crash the UNIX system we were using. Because messages are read without response, this will create the appearance that the machine, not the application, is down. However, the application cannot control all information related to a host. In particular, the status of a particular host may be obtainable from other methods, e.g., a shared file system, or the ICMP protocol in the TCP/IP set of protocols [ICMP 81].

This command simulates total failure of an application. Remotely, the application appears to have stopped running.

Ignore *entity-list*
All messages received from the specified hosts will be read and discarded. The list may not be empty.

This is different from **Crash**, as it simulates network partition rather than an application failure. It is used to implement the **Partition** command of **SG** (Section 4.1), and sample usages of both are shown in the figures 4-1 and 4-2. **Ignore All** has the same effect as **Crash**.

Recall *entity-list*
This is the opposite of **Ignore**, allowing normal processing of messages from the specified hosts. **Recall** is used to implement the **Reliable** command in **SG** (Section 4.1).

Drop *n* Do not process the next *n* incoming messages. This simulates unreliable networks.

Forget *n* Do not process the next *n* outgoing messages. The only distinction between **Forget** and **Drop** will occur in systems with side effects, e.g., updating a visual display, or altering a file on a shared disk.

Set *variable*

message-processing-delay
This variable may be set to a particular time value (in milliseconds) to delay the processing of each message, or it may indicate the modulus for a randomly-determined delay. That is, a random delay will occur each time before the processing of the message. The delay will occur after the message is read.

It is used to simulate slow processing under a high load, such as on an overburdened processor.

message-transmission-delay
As above, but the delay will occur before sending the message. This variable simulates transmission delay in the network. As with **Drop** and **Forget**, any differences between this variable and **message-transmission-delay** will only be apparent on systems with side effects.

Echo *string* The string is printed to the terminal or to an output file. Scripts may use this command to indicate errors or their current stage of execution.

Timestamp *string*
Like **Echo**, the string is printed to the terminal or to an output file. The time and date at the moment of execution is printed along with the string.

Abort Immediately halt the instrumented process. If possible, a core dump of memory is saved.

Clear **Clear** is the only action which affects all other actions. All events are removed from the event queue, effectively cancelling the execution of all scripts running in the process.

3. A Specific Application

Our work on testing was originally motivated by our work in distributed programming environments. Here we describe relevant details of the program environment before discussing the predicates and actions which apply only to this application. The reliability component of the programming environment work is described in detail elsewhere [Kaiser 87b, Hseush 88].

In the Mercury programming environment, users write programs in Mercury-created editors. Detailing all that these editors do would detract from the the purpose of this paper. During the editing process, editors communicate with each other by passing *attributes* about their programs through an *attribute propagation layer*, or APL. For the purposes of this discussion, *attributes* consist of five parts: a sequence number, a timestamp, a system name, a module name, and a data segment. Sequence numbers and timestamps are used to determine the validity of a particular attribute. This is discussed in more detail below. There may be multiple systems, and multiple modules in each system. Attributes store information for each module in the data segment; there is currently one attribute associated with each module. The data segment is used only by the Mercury editor; it is not examined by the APL. Attributes are replicated at each APL in an attribute cache. Hence, the APL behaves as a simple distributed database.

Figure 3-1 displays the layering of the information structure. If the Mercury editor 2, which is running on machine A, creates an attribute, it will send it to the local APL process. The APL process running on A will decide whether or not to pass it on to B

after examining the sequence number and timestamp. Upon receiving the message, B will also check the sequence number and timestamp in order to decide whether or not to update its cache with the new information. B always acknowledges the reception of the information, whether or not it uses it. It also updates internal tables indicating that host A seems to be up, and so its APL should receive any new attributes generated on B. There will always be only one APL process per host, and any number of Mercury processes in communication with each APL. Mercury processes automatically communicate with the local APL database; the user is usually unaware of it. Attributes may be sent arbitrarily frequently during the course of an editing session, depending on the nature of the editing.

When a new APL process starts, it asks all other existing APL processes for the contents of their caches, and updates its cache with the correct information. Since all APLs possess their own cache of all attributes in the system, and an arbitrary number of host and message-passing failures may have occured, there may be conflicting information reported from each APL. When this is noted, the newly-created APL process gives all the out-of-date APLs the new information. This process repeats until quiescence, which is guaranteed to occur provided the network ever achieves sufficient stability.

The algorithm underlying the APL process is reliable, in that caches will be correctly updated with the best information available, even when hosts are arbitrarily partitioned away from one another. Any disparity between the global view and the local view (what each APL knows) caused by machine or network failure will eventually be resolved when APLs communicate with one another.

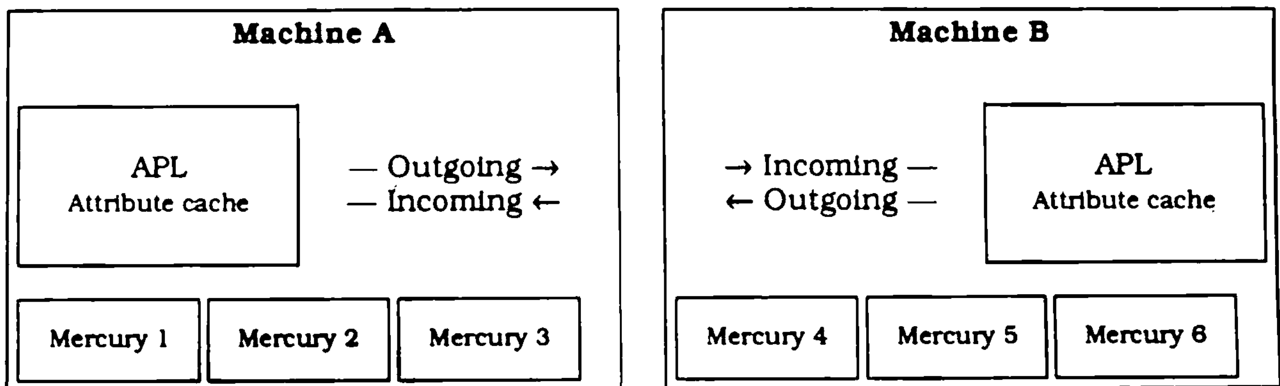

Figure 3-1: Information-passing in Mercury programming environment

3.1. Application-Specific Conditions

The following application-specific conditions are supported. They all work in combination with the conditions previously defined in section 2.3.2. In designing these conditions for the **SE**, we were primarily concerned with determining when an error in the reliability code had occured, and hence we only provide means of comparing attributes against a value stored in a *register*, short-term variables used by **SE**.

Compare *attribute register*
A remotely received attribute is compared with the value contained in a numbered register. It is true only when they are exactly equal. (See the **Load** action, below.) Non-existent attributes or registers will always compare as false: a non-existent register and an empty attribute (i.e., its size is 0) will not compare as equals.

Size *attribute cmp n*
Checks the size of the attribute using the comparator given (e.g., !=, ==, <=, >, etc.) and size given. Non-existent attributes have size -1.

Editors Identical in function to the application-independent **hosts** condition. The APL program communicates with zero or more local text-editors. Editors may not be specified by name, only by the keywords **any** (more than zero), **one**, **some** (more than one), **all**, or **none**.

There are, of course, many other conditions that could be useful for particular applications. Although the following predicates are not currently used in the Mercury application, they are useful enough to merit short descriptions.

Duplicate *n*
Tests if the last *n* messages were identical to each other.

Corrupt *n* Tests if the last *n* messages failed some test for corruption, such as a check-sum on the contents.

Sequence-Number *n*
True if the last message sent or received had sequence number *n*.

Timestamp [+/–] *t*
True if the last message sent or received had timestamp *t*. A plus or minus in front of the timestamp indicates a later or earlier time.

Type *protocol-message-type*
True if the last message had the specified message type. The specification of a particular type might be a number stored in the message, or the symbolic name of that type, e.g., INIT, LOAD, QUIT.

Size [+/–] *n* Compares the size of the last message to the number *n*, returning true if the size is *n*. A plus or minus in front of the number would indicate that the messages size should be greater than or less than *n*.

Contents *contents-specification*
Compares the contents of the last message to the specification given. This predicate could be structured in limitless ways. A specification might be a precise description of what is expected, or perhaps a regular expression would be compared against the contents.

3.2. Application-Specific Actions

As discussed previously, a script should be able to *create* information, so that messages will be sent. The actions described in section 2.3.3 will all hinder the passage of information. We chose to force the APL process to send messages indirectly, by causing their contents to change. This method was chosen because it most closely simu-

lated the behavior of the Mercury editors in communication with the APL process. We could have called the **send** routines directly, at the cost of some conceptual simplicity: all of the following actions are sufficient to cause a message to be sent.

Here are the application-specific actions:

Create *attribute*
An empty attribute is created. The specified system and module are created if necessary. Empty attributes could cause errors by their very existence.

Delete *attribute*
An attribute is deleted.

Update *attribute*
An existing attribute's contents are "slightly" changed, and its timestamp is updated, causing all attendant processing and message-passing to occur.

Grow *attribute n* %
The data segment of an existing attribute is increased in size by the specified percentage. Percentages may be greater than 100%, but they must be positive. Random values are used for the contents of the attribute, which is formatted in a manner which the Mercury editor can read. (These random values represent identifier names from Mercury editors.) Percentages are always rounded up so that any percentage size will cause a growth of at least one byte. If an attribute is empty, then any percentage size will cause it to increase to one byte.

Shrink *attribute n* %
An existing attribute is decreased by the specified percentage. Percentages must be less than or equal to 100%, as well as being positive. An attribute shrunk by 100% will be reduced to size 0, but it will not be deleted.

Load *attribute n*
An existing attribute is loaded into register *n*, in order to be used later by the **Compare** predicate.

We chose not to include the following actions for APL, although they could serve a useful purpose in testing other applications.

Send *contents-specification*
Send a message with the specified contents. A generate-contents function which took parameters appropriate for the application might be useful with this action.

Corrupt *n* Alter the next *n* messages in such a way that they will be detected by any error-detection code present in the application.

Resend *n* Resend the next message *n* times. This is useful in checking any assumptions about duplicated messages.

Figure 4-2 shows an application-specific script generated by the script generator **SG**. This script will partition two hosts away from another, keep a moderately-heavy load

of message-passing on one machine for an hour by growing and shrinking a single attribute, and finally, remove the partition barriers. This script shows the utility — in addition to correctness — of a particular program under a given set of circumstances. No facility is made in this script for the detection of errors during testing. Logging information could be kept by the process, or this test could be performed under the supervision of a programmer.

3.3. Pragmatics of Script Execution

Our implementation of **SE** and **SG** is still in progress. We do have preliminary results to report from our merging of the **SE** thread into the existing APL code. While the technique of modifying **send** and **receive** is a general one, the details of installing these changes will vary considerably from application to application.

It was found necessary to make the following general changes in our code:

- Changes in data structures to install message counters and an ignore-host list for each host-record.
- Changes in the three routines where **send** or **receive** were implemented. Four different variables were used: two flags to allow or disallow sends and receives, and two variables which contained the amount of time to sleep before sending or after reading a message. These changes are simple, yet allow all the functionality described above.
- Changes in the timeout code, which interfered with proper **SE** execution.

The last change resulted from using a system call to perform a blocking read which timed out after a dynamically-computed timeout interval. Event which depended on being executed at a certain time were forced to wait longer than was strictly necessary. Computation of the timeout interval had to considerthe **SE** module. If it had been possible to implement **SE** as a separate thread of control, blocking would not have been a concern.

Other problems resulted from the interval timer granularity problems pre-existent in UNIX, forcing us to to develop heuristics to allow time-dependent events to run within a small interval of accuracy. Obviously, we do not expect that these granularity problems are limited to UNIX. To give an example, two events may be scheduled to start at a certain time, one event within a few milliseconds of another. It may not be possible to set a timer with such accuracy. Faced with a choice between executing one event a few milliseconds earlier than scheduled, or waiting for possibly hundreds of milliseconds to pass, the scheduler should arrange for execution of the first event, then perform the second without waiting for the precise time to arrive. This will execute one event closer to the time at which it would have ideally executed, and it could reduce the overhead of **SE**.

4. Script Generation

The script-generator **SG** takes a simple set of commands as input, and produces scripts tailored for the specified machines. The base idea is that processes are defined as having abstract *properties*, and testing scripts are created which cause those processes to simulate the possession of those qualities. At the moment, our design of **SG** involves a simple understanding of message-passing protocols and various input parameters which would allow a programmer to tailor **SG**-generated scripts for a particular set of tests. For the algorithms similar to the one implemented in our application, this is acceptable, although more general methods based on protocol specifications need to be studied. Despite the limitation of not understanding general protocol specifications, **SG** is a useful tool in that it enables testing to be performed at a higher level of abstraction.

After describing the types of properties that may be associated with a process, section 4.2 shows how the timing of these scripts can be controlled, and section 4.3 describes input parameters for **SG** such as the desired message-sending rate.

4.1. Associating Properties with Processes

The following group of commands describe hosts and processes in terms of *properties*, such as being slow, unreliable, or partitioned. The generated scripts will cause events to occur in the appropriate application which will cause behavior that merits the given description. For example, a script for an unreliable host will lose a certain number of messages every few transactions, stepping through all possible combinations of loss. Properties given for a host will apply to any process on that host.

Hosts { **entity-list** }

The entity-list is a list of names to be used in all generated scripts. The command **Hosts** { **A, B, C, D, E** } means that scripts will be generated for hosts A through E, provided the **Map** command does not substitute names in the output script.

Processes { **entity-list** }

Processes are also specified for a generated script, and **Map** may be used to rename a process in the generated script.

Map { **entity-list** }

Map changes the hostnames or process identities which are output in the generated scripts. It allows scripts to contain general plan of testing partition, high-load, etc., while specific hostnames are given in only one place. Hosts may be given as hostnames or internet addresses. Process identities may be specified in an OS-specific format. The mapping is performed according to the order in the entity-list.

Slow { **entity-list** }

The named processes will be slowed in the generated scripts. If this command is repeated, it doubles the effect.

Generate { **namelist** }

The named hosts will create a "moderate" load on the system, in terms of

sending packets. The load is is determined by the **Sending-Rate** command, described in later sections.

Unreliable { **namelist** }
The named hosts will become unreliable: at pre-set intervals they will crash, reboot, and drop packets. The same series of failure will be repeated for all "unreliable" hosts, but not in the same order. In other words Host A and Host B might both be told to fail after receiving one message, but this failure will not occur at the exact same point in both of their scripts. The intervals between these events will not be random.

Random { **namelist** }
Random is similar to **Unreliable**, except that the named hosts will become crash, reboot, and drop packets at random intervals. Two commands are necessary for this, because debugging programs which are failing in predictable ways will be more fruitful than the reverse. On the other hand, random testing over a period of time constitutes a different type of check for correctness.

Partition { **namelist** } { **namelist** } [...]
Two or more sets of hosts will be partitioned from each other. That is, **Partition** { **A, B** } { **C, D, E** } will separate the first two hosts from the last three hosts. However, A and B will be able to communicate with each other; likewise for C, D, and E.

Reliable { **namelist** }
Reliable reverses any failing properties placed on any specified host by the **Unreliable**, **Random**, or **Partition** commands. Unreliable hosts will become reliable, partitioned hosts will be able to communicate with each other, and so on. It will not affect any loads created by **Generate** (or any actual failures!).

Quiet { **namelist** }
Any named hosts which are generating loads will cease to do so. This will not affect the processing of messages created by ordinary use of the system.

4.2. Specifying Timing of the Generated Scripts

The following commands control the timing in the generated scripts. **SG** generates scripts which are sequentially executed, subject to timings imposed by the following commands. That is, if a host is marked as **Unreliable**, it will stay in that mode until it runs out of commands (in the generated script), or until the effects of a **Reliable** command in the **SG** input appear to reverse it. Commands such as **Sleep** may be used to delay the effects of commands listed in the previous section.

Sleep *n* Any conditions previously set up in scripts generated by **SG** will continue without change until the specified number of seconds passes. Fractional seconds may be specified, which will be executed up to millisecond accuracy. This is a relative time command which may appear more than once in the **SG** input.

Wait *time* As with **Sleep**, **Wait** will not disturb any conditions previously set up until the specified time arrives.

Stop { **namelist** }
All events are removed on the specified hosts, halting the execution of scripts on those machines. If no hostnames are listed, then all hosts will cease executing their list.

4.3. Specifying Other Parameters to SG

The following **SG** commands are parameters indicating the types of testing performed in the generated scripts.

Retry *n* This tells **SG** how many retries on failure are *likely* to occur in the course of executing the algorithm. That is, if the application running on Host A receives no reply from Host B after sending a message, it will resend its message *n* times.

Limit *n* Control the number of combinations of failure for the **Unreliable** and **Random** properties. An unreliable host will fail in up to n^2 different ways, by dropping 1, 2, . . . , or *n* packets after the first, second, . . . , and *n*th packets. Each iteration through these combinations of types of failure will constitute a separate, but not necessarily unique test.

Delay *n* This is a time in milliseconds, which says how long each application is *likely* to delay before resending an unacknowledged message.

Message-sending-rate *n*
This is the rate at which the **Generate** command will cause messages to be set. It is given in units of messages/second. *N* may be given as a fractional rate.

Data-missing-rate *n*
This is the rate at which the **Unreliable** command will cause messages to be forgotten on each individual host. It is given as a percentage, which represents the *maximum* number of messages dropped over a given period of time.

Down-time-delay *n*
This is the time, in seconds, of how long a host which has "crashed" will stay down. It affects the output of the **Unreliable** command. *N* may be given as a fractional rate.

Figure 4-1 shows input for **SG** which will generate scripts for three different machines, testing partition of three hosts into two groups while one host is continuously generating information. The output script from **SG** is shown in Figure 4-2.

5. Conclusions

The primary contribution of this paper is the development of a testing methodology for reliable distributed applications. The testing methodology described here is suitable for a wide range of applications, both because it is contained in a general method of

```
# load-partition.sg

Hosts { A, B, C }                   # symbolic names
Map { Gollum, Mojo, Frodo }         # the real host names
Generate { A }                      # make A create info
Partition { A, B } { C }            # divide into 2 groups
Sleep 3600                          # wait one hour
Reliable { A, B, C }                # let them talk again
Sleep 900                           # wait 15 minutes
Stop                                # stop the simulation
```

Figure 4-1: Input file which performs some simple load and partition tests

instrumenting applications in terms of their **send** and **receive** operations, and because the idea of testing protocols in terms of an explicit or underlying Finite-State Machine structure is a useful one. We also describe the beginnings of a general method for automatically generating testing scripts in terms of abstract properties on processes. Design testing from a high level of abstraction has been shown to be useful in a number of testing disciplines, including automatic compiler testing [Bazzichi 82], [Demillo 87].

We note the following open problems in our work. Other limitations were discussed in the introduction.

- It is currently difficult to specify the failure or success of a script in terms of the script itself. External methods, such as logs or human supervision, must be used. Our methods are well-suited for burn-out testing; that is, running a program hard until it breaks.
- We provide no means to translate from arbitrary protocols to a Finite-State Machine representation, and hence no means of using this FSM representation in **SG** to generate correct scripts automatically.

There are alternative methods for testing distributed applications similar to the one described, which we briefly discuss:

- Central pass-through points have been suggested as a means of controlling the testing. Messages sent from process A to process B would always pass through a central point, which would log the message and somehow decide whether or not to pass the message on to B. There may be no need to change individual processes, provided kernel or hardware modifications can be installed to create the necessary message diversion. The primary benefit of this method is that centralized control moves the testing closer to a "global knowledge" of what is happening in each process. Also, the semantics of individual processes remain unchanged. However, this method is not necessarily better, even apart from the scope of making the necessary kernel changes. Changing the message-passing would require per-process semantics of the **send** and **receive** system calls in order to provide the behavior possible from our testing methodology. The timing of the **send** and **receive** system calls are likely to be no less perturbed by this scheme than they are by changing the **send** and **receive** calls at the application level. Also, practically speaking, it is better to manage this information at the process level.

```
# script generated by SG
# version 0.5, on Thu Mar 23 03:33:25 1989
# each machine will execute its own part of the script

begin gollum
    timestamp "processing load-partition.sg on gollum"
    boot                                # pretend to begin
    create ("sys", "mod", 0)            # generate load
    when size ("sys", "mod", 0) = 0
        while size ("sys", "mod", 0) < 2000
            grow ("sys", "mod", 0) 10
    while true
        when size ("sys", "mod", 0) > 1900
            shrink ("sys", "mod", 0) 80
    ignore frodo                        # partition
    when +1:00                          # after an hour,
        recall frodo                    # back to normal
    when +1:15                          # fifteen minutes
        timestamp "finished processing load-partition.sg"
    when +1:15                          # finish all processing
        clear
end gollum

begin mojo
    timestamp "processing load-partition.sg on mojo"
    boot
    ignore frodo                        # partition
    when +1:00
        recall frodo                    # back to normal
    when +1:15
        timestamp "finished processing load-partition.sg"
    when +1:15
        clear
end mojo

begin frodo
    timestamp "processing load-partition.sg on frodo"
    boot
    ignore gollum, mojo                 # partition
    when +1:00
        recall gollum, mojo             # back to normal
    when +1:15
        timestamp "finished processing load-partition.sg"
    when +1:15
        clear
end frodo
```

Figure 4-2: Output from **SG**

- All of our predicates passively check the state of the system. That is, they execute their tests *without* generating any new messages. Clearly, much useful information could be passed between a testing module and its duplicate on a remote host; but just as clearly, this would add unacceptably to the complexity of the testing problem.

Acknowledgements

We would like to thank Dan Duchamp, Josephine Micallef, and Rob Lehman for their discussions of this paper, as well as the Mercury programmers who implemented APL, thus motivating this work.

References

[Bauer 79] Jonathan A. Bauer and Alan B. Finger.
Test Plan Generation Using Formal Grammars.
In *Proceedings of the 4th International Conference on Software Engineering.* IEEE Computer Society, Long Beach, CA, Sept., 1979.

[Bazzichi 82] Franco Bazzichi and Ippolito Spadafora.
An Automatic Generator for Compiler Testing.
IEEE Transactions on Software Engineering SE-8(4):343-353, July, 1982.

[Beizer 83] Boris Beizer.
Software Testing Techniques.
Van Nostrand Reinhold, New York, 1983.

[Chow 78] Tsun S. Chow.
Testing Software Design Modeled by Finite-State Machines.
IEEE Transactions on Software Engineering SE-4(3):178-187, May, 1978.

[Demillo 87] Richard A. Demillo, W. Michael McCracken, et al.
Software Testing and Evaluation.
Benjamin/Cummings, Menlo Park, CA, 1987.

[Gouda 84] M. Gouda and Y. Yu.
Synthesis of Communicating Finite-State Machines with Guaranteed Progress.
IEEE Transactions on Communications COM-32(7):779-788, July, 1984.

[Haban 87] Dieter Haban.
DTM — A method for testing distributed systems.
In *Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 45-55. ACM Press, New York, March, 1987.

[Halpern 86] Joseph Halpern (editor).
Fifth Symposium on Reliability in Distributed Software and Database Systems.
ACM Press, New York, 1986.

[Howden 87] William E. Howden.
Software Engineering and Technology: Functional Program Testing & Analysis.
McGraw-Hill Book Co., New York, 1987.

[Hseush 88] Wenwey Hseush and Gail E. Kaiser.
A Network Architecture for Reliable Distributed Computing.
IEEE Network 2(4):28-44, July, 1988.

[ICMP 81] J. Postel (editor).
Internet Control Message Protocol.
Technical Report RFC 792, USC/Information Sciences Institute, September, 1981.

[Kaiser 87a] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiuser, Distributed Language-Based Environments.
IEEE Software :58-67, November, 1987.

[Kaiser 87b] Gail E. Kaiser and Simon M. Kaplan.
Reliability in Distributed Programming Environments.
In *Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 45-55. ACM Press, New York, March, 1987.

[Mackert 87] L.F. Mackert and I.B. Neumeier-Mackert.
Communicating Rule Systems.
In *Proceedings of the IFIP WG 6.1 Seventh International Conference on Protocol Specification, Testing, and Verification*, pages 77-88. May, 1987.

[Myers 79] G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

[Rudin 87] Harry Rudin and Colin H. West, (editors).
Proceedings of the IFIP WG 6.1 Seventh International Conference on Protocol Specification, Testing, and Verification.
Elsevier Science Publishers, Amsterdam, The Netherlands, 1987.

[Sarikaya 88] Behçet Sarikaya.
Protocol Test Generation, Trace Analysis, and Verification Techniques.
In *Second Workshop on Software Testing, Verification, and Analysis*, pages 123-130. July, 1988.

[West 78] C. West.
An Automated Technique of Communications Protocols Validation.
IEEE Transactions on Communications COM-26(8):1271-1275, August, 1978.

[Weyuker 88] Elaine J. Weyuker.
The Evaluation of Program-Based Software Test Data Adequacy Criteria.
Communications of the ACM 31(6):668-675, June, 1988.

[Wittie 87] Larry Wittie (editor).
Sixth Symposium on Reliability in Distributed Software and Database Systems.
ACM Press, Kingsmill — Williamsburg VA, 1987.

[Zimmerman 80] H. Zimmerman.
The ISO Model of Architecture for Open Systems Interconnection.
IEEE Transactions on Communications COM-28(4), April, 1980.