Contents

13

Software Fault Insertion Testing for Fault Tolerance

MING-YEE LAI AND STEVE Y. WANG
Bell Communications Research

ABSTRACT

Fault tolerance of a telecommunications system is critical to provide high-quality services to users during abnormal conditions and failures. Insufficient design, implementation, and testing for fault tolerance could lead to significant outages, impacting millions of users for long hours, as seen in the recent Common Channel Signaling (CCS) and Internet networks outages. This chapter focuses on improving fault tolerance through testing. A clear testing methodology, supporting tools, and a concept of testing adequacy is essential to the completion of testing a large and complex telecommunications software. Toward this end, a methodology for testing of fault tolerance through Software Fault Insertion Testing (SFIT) is discussed. Those logical system components responsible for handling faults and providing services are defined as the *fault manager* and *service manager*, respectively. The goal is to reduce the complexity of testing a system by testing its fault manager. A notion of "testing adequacy" depending on the structure of the system is also proposed in the methodology as the basis to guide the selection of test cases. The proposed methodology is a systematic approach to testing fault tolerance of existing and new telecommunications systems. This methodology has been used in testing fault tolerance of various telecommunications systems with large scale software, such as circuit switches, digital cross connects, signal transfer points, and broadband switches. The experiences of applying the methodology are also discussed.

13.1 INTRODUCTION

Fault tolerance of a telecommunications system is critical to providing high-quality services to users during abnormal conditions and failures. Insufficient design, implementation, and testing for fault tolerance could lead to significant outages, impacting millions of users for

Software Fault Tolerance, Edited by Lyu

long hours, as seen in the Common Channel Signaling (CCS) and Internet network outages [Coa93].

In 1992 and 1993, Federal Communications Commission (FCC) Network Reliability Council (NRC) conducted an extensive study that involved all major key telecommunications system suppliers, network providers, service providers, users, and research institutions in the industry and issued a report entitled *Network Reliability: A Report to the Nation* [NRC93]. In the report, more than 70% of outages in network elements (such as switches, digital cross connects, and signal transfer points) are attributable to software related problems.

There are three lines of defense against software faults: avoidance, elimination, and tolerance. Fault avoidance is achieved during the specification, design, and coding process of the software. Fault elimination is the responsibility of the testing and maintenance processes. Fault tolerance is a property of the deployed software that is designed to survive anticipated possible failures. Fault tolerance, which is complementary to fault avoidance and fault elimination, is indispensable to dependable telecommunications software. It enables graceful service degradation in the face of system failures.

In the NRC report, Software Fault Insertion Testing (SFIT)[1] [Li94, BCR93] was recommended *to be performed as a standard part of a telecommunications system supplier's development process* for improving fault tolerance. However, testing a large, complex system for fault tolerance is a daunting task. A clear testing methodology, supporting tools, and a concept of testing adequacy is essential to the completion of this task. In this chapter, a methodology for testing of fault tolerance through SFIT is discussed.

The main objective of SFIT is to test the fault tolerance capability through injecting faults into the system and analyze if the system can detect and recover from faults as specified by the system or anticipated by the customers of the system. This testing of "rainy-day" scenarios before system deployment can reduce or contain service impacts on the system customers in the presence of failure scenarios. The results from SFIT can lead to either fixing of individual software bugs or discovery of design deficiencies in system fault tolerance. With SFIT, several network-wide telecommunications outages (such as long-distance phone service disruption caused by tandem switch software in January 1990 and local-exchange phone service disruption caused by CCS signal transfer point software in June 1991), could be avoided.

There are several challenges in the pursuit of a methodology for testing fault tolerance.

1. **Complexity of telecommunications software —** A typical telecommunications system may contain millions of lines of source code. Moreover, the software complexity of these systems is growing while the public demands increased simplicity of use, functionality, and dependability at reduced costs.
2. **Dormancy of faults —** Some faults may lie dormant in the system without causing observable failure long after deployment. Those faults may only be triggered when a system is under extreme stress, abnormal use, or severe failures.
3. **Diversity of telecommunications systems —** National and international telecommunications networks are composed of products made by many vendors. This heterogeneity poses a challenge to find a generic methodology for testing fault tolerance.
4. **Constraint of resource availability —** A testing methodology should allow fault tolerance validation of a system by testing organizations that are independent of development. Independent testing may be limited in access to and understanding of proprietary internal information such as source code and in testing time.

[1] Also known as Software Fault Injection Testing (SFIT)

13.2 TESTING FAULT TOLERANCE USING SOFTWARE FAULT INSERTION

13.2.1 Definition of Faults, Errors, and Failures

The International Federation for Information Processing (IFIP) Working Group 10.4 has defined [Lap92]:

- **Fault** — adjudged or hypothesized cause of an error.
- **Error** — part of system state that is liable to lead to failure. Manifestation of a fault in a system.
- **Failure** — deviation of the delivered service from compliance with the specification. Transition from correct service delivery to incorrect service.

Each fault may have the following associated attributes:

- Type: The classification that the fault belongs to. Example types are memory fault, logical fault, etc.
- Locality: The location of a fault in certain component. This is the most important characteristic. A fault in a risky component will inflict more serious damage on the system than the identical fault in a component incapable of affecting system performance.
- Latency: The time interval between insertion and observation of the fault.
- Frequency: The average occurrence of a fault over a given time interval.
- Severity: The magnitude of the fault's effect on system performance. Note that this attribute depends on the combination of the fault type and its locality. For example, the severity may vary for a fault type in two components with different degrees of criticality. A preliminary approach to assigning severity to faults is through error codes, whose severity degree is specified when the system is designed.
  The severity of a fault is classified into *critical*, *major*, or *minor* [BCR90], depending on whether the system can continue to function indefinitely without fixing them.
  - **Critical fault** — a fault that prevents the system or significant parts of it from functioning until it is fixed.
  - **Major fault** — a fault that prevents the system or significant parts of it from functioning, but there is a work-around that can be used for a limited period of time.
  - **Minor fault** — a fault that is neither critical nor major.

13.2.2 Why Use SFIT?

The fault tolerance capability of a typical telecommunications system is critical since it is responsible for rescuing the system from errors in the field. Therefore, it should be tested in a controlled and systematic way before deployment. To this end, SFIT has emerged as an important technique for at least two reasons:

- Failure acceleration: Testing fault tolerance by waiting for the occurrence of failures in the field is not desirable. By intentionally inserting faults to invoke the fault tolerance capability, one can achieve more thorough testing in a controlled environment and within a desirable time frame.
- Systematic testing: Without systematic testing and analysis, it is hard to know which fault

may be activated in the field and which component of the system will respond to the error caused by the fault. By inserting faults designed to invoke a specific fault tolerance capability, one can test that functionality effectively in the testing environment.

13.2.3 Previous Work of SFIT

There has been considerable research [Seg88, Dil91, Ros93, Kao93, Hor92, Arl90, Arl92, Iye93, You93, Kan92, Avr92, Ech92] in the Software Fault Insertion Testing area. The major differences among them are mainly in target systems, fault types and method of injecting faults. The target systems range from real-time distributed dependable systems [Seg88, Avi87] to large-scale operating systems [Chi89]. The fault types injected into the target systems also vary greatly from simple memory bit faults to processor level and system/communication faults.

The insertion method applied to software is inspired by the success of hardware fault insertion testing in the expectation that this kind of testing can be used to evaluate the robustness and expose the deficiency of the tested software. Faults can be inserted through:

- Changing a predetermined portion of the source code to produce a syntactically correct but semantically incorrect mutant [Dem88]; or
- Applying a patch to the object code at a designated location [Lai92].

Because such insertion methods deal with source code or object code directly, they are called *code injection* methods. An example of code injection is to change and re-compile the software to produce a ”pointer too large” or an ”invalid input parameter” fault. Such an injection is possible in both the object and source code. Another example of code injection is through the use of patching tool. The source code or value of some data structure can be corrupted by patching the software or data.

Another injection approach, *state injection* , in contrast, is achieved through altering the state or behavior of a running system. More specifically, instead of injecting faults, a system-level fault ”manifestation” is injected as an erroneous state. This fault state is intended to mimic the error produced by a fault. For example, data inconsistency, that is, a common error between two copies of data in a system, can be simulated by corrupting the data of either one. There are several methods for state injection:

1. Process-based: The injection is to be accomplished by a high-priority process modifying the state of the computation [Ros93]. This approach often needs support from the underlying operating system.
2. Debugger-based: Using the facilities of a debugger (e.g. dbx, gdb), errors can be injected to a running process through the composition of breakpointing, evaluating, and jumping.
3. Message-based: For message-oriented communication between two software components, the erroneous state can be created by disrupting message sequences using message-based tools.
4. Storage-based: Using the storage manipulation tools (for memory, disk, or tape), errors can be injected into the system by changing the value at some location of the storage hierarchy, which represents some system state.
5. Command-based: Using the commands from the craft interface or remote maintenance terminal, errors can be injected by changing the states of the system entities for operations, administration, maintenance, and provision.

Most reported studies using fault insertion were aimed at assessing the efficiency in handling faults of fault tolerance mechanisms. The focus of this chapter is on the testing methodology for eliminating design/implementation problems in the fault tolerance mechanisms and improve these mechanisms.

13.3 FAULT MANAGER

For conceptual simplicity, one can view the collection of software safeguard components (such as the network safeguard, node safeguard, and Operation, Administration and Maintenance (OA&M) software) that are dedicated to handle exceptions in a typical telecommunications system as a logical entity — *fault manager* . The other part of the system that provides services to users (such as feature software) can be viewed as another logical entity — *service manager*.

Under normal operation, the service manager provides services to users while the fault manager keeps constant surveillance on the health of the service manager, until an error situation is detected. The fault manager then takes action to get the service manager back to normal operation.

In this view, testing fault tolerance in a telecommunications system amounts to testing the fault manager. Software faults are inserted in the service manager to trigger the reaction in the fault manager.

In more details, the fault manager provides the following capabilities to ensure continuous service to customers: error detection, isolation, recovery, and reporting (DIRR).

- Error Detection — Detects errors through continuous monitoring, periodic tests, per-call tests, or other automatic processes. Software audits are considered as part of the error detection capability.
- Error Isolation — Isolates the error to its source, preferably to a single or a reasonable subset of components.
- Error Recovery — Recovers errors by automatic or manual actions such as retry, rollback, on-line masking, restart, reload, or re-configuration, to minimize the degradation of service.
- Error Reporting — Sends error messages to a display device, a logging device, or an Operations System (OS), describing the error, the place where the error is observed, and system reactions to the error.

Although the fault tolerance strategies employed in different products vary, one common aspect for complex systems is that the DIRR capabilities are rarely implemented in one centralized location physically. Usually, for both efficiency of error handling and convenience of implementation, DIRR capabilities are physically distributed in separate software components. These DIRR components, through cooperation and communication using a fault tolerance protocol, work as a single fault manager. It is nonetheless an important conceptual simplification to view DIRR functions of the fault manager as a *single logical* entity (even though some error detection capability is built-in the application). We take this logical view throughout this chapter.

Two major capabilities in the fault manager are focused in this chapter: error detection and error recovery. Error detection is the beginning of exception handling. Physically, errors can be detected either by an application or by an error detection task (or process). For example, the application system adopts some self-check logic to detect certain kinds of anomalous

conditions and raises exceptions. Other errors beyond the self-check logic can be detected by some error detection task. For example, data inconsistency between two applications can be detected by an error detection task.

Recovery often takes several levels to restore the system to normal operation. In general, errors can be recovered by an application with self-recovery capability such as the recovery block [Hor74]. It is the objective of a robust system to take the least possible disruptive recovery action in the presence of errors. For example, a local low-level recovery action is preferred when devices or resources are hung. However, a system-wide recovery may be needed when the central processing unit is in an unrecoverable state. Even more drastically, when both the program and data are corrupted, a system reload from a backup device is needed as the last resort for system recovery.

13.4 CATEGORIZATION OF SOFTWARE FAULTS, ERRORS, AND FAILURES

Categorization of software faults, errors, and failures can achieve the following benefits:

- A set of generic faults, errors, and failures can serve as a partial adequacy checklist of common problems against the potential deficiency of fault handling.
- No matter how diversified the systems may be, they share similar user requirements and are susceptible to a set of common and generic faults, errors, or failures.
- Interacting with the fault manager through faults, errors, or failures from the service manager can generate realistic inputs to the fault manager for test execution and observation.

13.4.1 Fault Categorization

Fault categorization is an important research area for fault prevention, elimination, and fault tolerance. Fault categorization has been studied extensively for the purpose of fault prevention during development [Chi92].

This section emphasizes the fault categorization for fault tolerance (as opposed to fault categorization for fault prevention and elimination.) There are many sources for generic fault types. The first source of fault types comes from system architecture analysis [BCR92]. For example, analysis of the system architecture may show that in some situations the execution order of two actions is critical. Hence, the system may have a recovery strategy for handling the event of mis-ordering the execution of these actions. Thus a fault that disturbs the required order will test this recovery mechanism.

The second source of fault types is from the error code of the system. Ordinarily, a list of *error codes* is available for each telecommunications system software. These codes specify which errors may occur and how the errors are to be treated. These errors were anticipated at the time the software was created, and thus reveal the designer's view of errors likely to occur.

The third source of fault types is through root cause analysis of empirical data. Field data on failures and modification requests from the customers indicated that these faults were not anticipated by the software or its designers.

The following is a sample set of common single fault types from various sources. These fault types should be disjoint.

- Intra-module faults

  - Memory faults, such as illegal access, illegal write, pointer too large, array index out of range, memory not allocated, buffer overflow, buffer not allocated, or illegal reference to uninitialized variables.
  - Logic faults, such as undefined state, omitted cases in a case statement, erroneous control statement, wrong algorithm, and incorrect value in data or program (by fault patches, typing errors, or environmental factors).
  - Mutation faults, such as mutating the operators (arithmetic, boolean operators) and operands (counter not incremented/decremented, and incorrect global variables).

- Intermodule faults
  - Incorrect input, such as input value out of valid range and input in mismatched data type.
  - Incorrect output, such as invalid output value.
  - Incorrect sequence/timing, such as wrong message sequences between two modules, messages delayed for too long, and lack of concurrency control.

There is a large number of software faults in a complex system. A library of faults must be systematically refined or enhanced to represent the most important fault types. New fault types must be both realistic and generic and are orthogonal to the existing fault types in the library.

13.4.2 Error and Failure Category

An error or failure is caused by some faults triggered under certain conditions. There are several points that encourage the use of state injection of errors/failures over the code-injection method. In general, the fault manager responds directly to errors (as a result of faults). Thus, injection of errors or failures is a short-cut to fault injection if the error state in the state injection method is reachable by the code injection method with some sequence or combination of input/trigger events. Further, injection of a specific error/failure can provide a better focus on a specific functionality of the fault manager than injection of faults. Lastly, in a distributed environment, some complicated failure modes are caused by the interaction of multiple errors, where error/failure injection is easier to simulate these failure modes.

When we allow error or failure insertion as surrogates for faults, the fault latency may be shortened. For example, to simulate a crash of a processor, it would be easier to send a crash message than to insert faults into processes to actually crash the processor. The following is a sample set of common error/failure types based on field experiences.

- Process errors, such as deadlocks, live locks, process looping, process hung, and using too many system resources.
- Message errors, such as lost message, corrupted message, out-of-order messages, duplicated messages and timeout waiting for message.
- Operating system errors, such as job overflow (overloaded with processes), resources thrashing, wrong signal, and wrong acknowledgement.
- Data errors, such as corrupted data value and structure, data inconsistency, and data integrity violation.
- Hanging resources, such as hanging alarms, terminal, I/O, lines, trunks, and CPU errors.
- CPU or storage devices crash.
- Device or CPU overload.

- Network errors such as network congestion (traffic overflow), link oscillation, and routing error.
- Operational procedure errors, such as wrong data input from the operator, and incorrect maintenance procedures.

13.5 SFIT METHODOLOGY

Testing is time-consuming, especially testing for fault tolerance. To execute a test case, several steps, including test set-up, test execution, and system restoration are required. Some injected faults or errors/failures may change the system into a chaotic state that cannot be automatically recovered. In this case, manual restoration may be required.

SFIT is recommended to be performed at the system testing or acceptance testing during the testing life cycle. In this way, the system's overall reaction to faults/errors/failures can be observed and analyzed. However, in some cases, SFIT can also be performed at the unit testing level where the fault manager functionality resides in a local subsystem level.

Figure 13.1 shows the methodology used for SFIT. The methodology provides a systematic approach to validate the fault tolerance capability. The methodology consists of the following steps:

A — Pre-SFIT Steps:

1. Software Architecture Analysis [Eri93]. Before conducting SFIT, a sufficient knowledge of the software (including functions of some key software components, such as error recovery software subsystem) and its architecture is needed. This analysis is proactive and consists of three key parts:
   - service manager analysis: pinpoint potential risky areas based on (a) new and changed software components, (b) functionally critical components, (c) fault-prone software components, (d) low test coverage components, and (e) components with poor software attributes (such as the Non-Commentary Source Lines of code, fault density during the internal testing, design, and code complexity measurement). The output from the analysis includes fault/error type and insertion point in the service manager for the test cases.
   - fault manager analysis: analyze fault handling mechanisms, observation mechanisms, and module interactions within the fault manager (the analysis result can be represented in a graphic form).
   - fault manager/service manager interface analysis: analyze the error codes and their associated attributes (such as error description, severity, and recovery action routines) as well as the possible event sequences (consisting of error code and recovery action) across the interface.
2. Root Cause Analysis. Internal testing results and external field problems often can give a good indication of the product's reliability. Therefore, root cause analysis on the internal trouble reports and customer service reports can help the testing organization to identify common problems that need to be addressed in SFIT. Root cause analysis is more reactive; nevertheless, it can help to identify the area and type of faults/errors to be tested.
3. Test Set Selection. During the test set selection, the following two aspects need to be identified: properties/predicates to be checked for assessing the proper behavior of the Fault

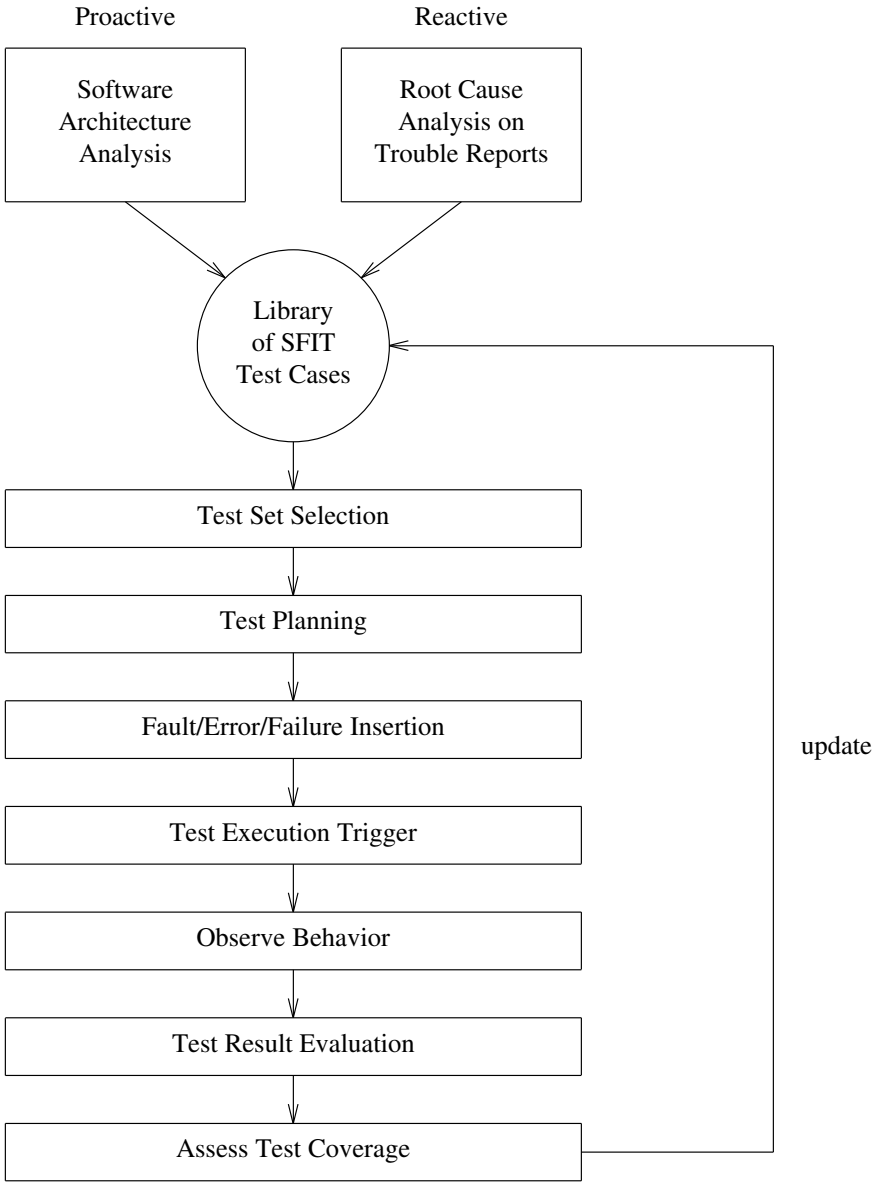

Figure 13.1 SFIT methodology

Manager in the presence of the injected faults (expected system behavior) and observations to be made to verify the assertion of the corresponding predicates (i.e, actual system behavior). Depending on different factors such as test coverage and available resources for testing, the test set needs to be selected before the actual SFIT testing. For coverage, the following criteria should be considered:

- Test cases that enhance the test coverage (see the discussion on the following step "assess test coverage" for more details).
- Test cases that simulate faults/errors that are similar to those that have occurred in the field as seen by customers.
- Test cases for faults/errors that have direct customer service impact.
- Test cases for high-profile faults/errors (for example, faults that catch the attention of the general public).

Another important factor for test set selection, is time available for testing. Since software fault insertion testing tends to be destructive, the time required to restore the system back to normal condition plays an important part. Therefore, if fault latency and the time required to recover from each software fault are taken into consideration, one may see that test set selection is a scheduling problem to maximize the testing in a given amount of time. While this approach is useful from a project management point of view (i.e., the project manager can allocate appropriate resources to conduct SFIT), it is difficult to measure the time required to prepare each test and the time the system will take to react and recover from each inserted fault.

4. Test Planning. After the test set has been selected, testing needs to be planned. For example, test scripts need to be prepared based on the selected test set. For code-based injection, software patches need to be prepared in advance. For state-based injection, appropriate tools need to be allocated to change the state of the system. If testing is to be performed at a remote location, proper links and expert support should be established ahead of time. Background traffic needs to be planned for test cases to simulate user's typical operation profile or abnormal traffic conditions.

B — SFIT Steps:

1. Fault/Error/Failure Insertion. With all the test cases available, this step involves the actual insertion of faults, errors or failures. The location of faults or errors should be identified during this step. Depending on the insertion method used (e.g., process-based, debugger-based, or message-based), the corresponding tools can be used to insert the faults/errors/failures.
2. Test Execution Trigger. A fault in the system may not be activated when it is inserted into the system. Therefore, the test trigger needs to be set during this step to activate the inserted faults. This test execution trigger can be used to reduce the natural dormancy of faults. Triggers could be input values from the users, internal and external events, or messages.
3. Observe Behavior. This step observes the system reaction to the inserted faults or errors within a specified time frame.

C — Post-SFIT Steps:

1. Test Result Evaluation. Test results can provide insights on two aspects. First, the test result can reveal the effectiveness of the test cases. Second, the test result can reveal the

weakness of the system's fault tolerance capability. The test result evaluation step can help to eliminate less effective test cases and identify areas for improvements for the system's fault tolerance capability.

2. Assess Test Coverage. Although complete testing coverage with SFIT is not economically possible, a notion of test coverage adequacy is essential to the confidence in the fault tolerance of a system. The criteria for test coverage adequacy consist of:

   - Coverage of the fault, error, and failure types in the library.
   - Functional coverage of the fault manager (test all functionalities in the fault manager).
   - Structural coverage of the fault manager (test all branches in the fault manager).
   - Coverage of the risky components in the service manager.
   - Coverage generated from a fault tree analysis of the software [Lev91].

   Thorough testing for these sorts of faults will constitute adequate testing of the fault manager of a telecommunications system. A test stopping rule may be specified by choosing an appropriate level of adequacy. Of course, setting a stopping rule short of complete adequacy is risky.
3. Update SFIT Test Case Library. A library of common and generic faults, errors, and failures along with their attributes (such as frequency of occurrence or severity) should be collected and stored. This library of faults, errors, and failures can be used to define test input for fault tolerance testing. The library will be updated periodically. In addition, faults designed to test for the rare, unusual, and severe fault tolerance conditions of the system will be added to the repository. These faults can be garnered from an analysis of system architecture and an understanding of the potential weaknesses of the fault manager. Together, the generic fault types and unusual fault types will, as our understanding grows, constitute an *adequacy* criterion for fault tolerance testing.

The details of these steps are described in [BCR92].

13.5.1 SFIT for Subsequent Software Releases

Telecommunications systems grow both in size and complexity. However, the growth is usually in the service manager and less in the fault manager area after the system has been operational in the field for years and its fault manager matures. One may ask why testing the slow changing fault manager for subsequent releases given that the system's fault tolerance capabilities have been tested in the previous release. The reasons for conducting SFIT for new releases are:

- Interaction of the service manager and the fault manager may be different for the new release because of the new or changed parts in the service manager and the fault manager.
- The reaction of the fault manager to the same inserted faults may be different in new releases because of new interactions between inserted faults and latent faults from the new release.
- New services may need some enhancement in the fault manager. For example, new ISDN services may need new audit functions in the fault manager to check the sanity of ISDN-related data structures.
- When a system's outage frequency and downtime do not meet customers' requirements, improvements on the fault manager over releases are essential.

13.6 SAMPLE SFIT TEST PLANS

The purposes of establishing a sample set of SFIT test plans are:

- This set of generic test plans can be used to develop detailed test cases for specific telecommunications products.
- This set of generic test plans can be further refined to improve the test quality as we gain experience with software fault insertion testing.
- This set of generic test plans can be used for repeated testing for future software releases.

In the rest of this section, we concentrate on the sample test plans for recovery actions and audit functions.

13.6.1 Test Plan for Recovery Actions

Service recovery for a typical telecommunications system means the automatic or manual protection actions taken to minimize degradation of service. Actions should be taken in the meantime to identify the cause of the trouble and repair it. Based on Bellcore's Operations Technology Generic Requirements (OTGR) [BCR91], the service recovery strategies for call processing based network elements shall be such that stable calls are protected and then transient calls are protected. The automatic service recovery should be considered before manual service recovery. Section 4 of the OTGR lists complete service recovery requirements. The following list is a sample of SFIT test plans for recovery routines. This list is broken down to generic recovery routines, call-processing specific recovery routines, database management recovery routines, memory management recovery routines, inter-process communication recovery routines, and network management recovery routines.

A — Generic Recovery:

1. Insert a fault and trigger it to invoke the service recovery action to see if it performs automatic recovery as designed.
2. Insert a fault and trigger it to invoke automatic service recovery and check if stable calls are maintained. For transient calls, the automatic service recovery should be satisfactorily completed, without the customer being made aware that a fault exists.
3. For each level or recovery, insert a fault and trigger it to invoke the appropriate recovery action. Observe the system's reaction to see if it performs as designed. For example, faults that have minor impact to the system's operation may result in small and local level of recovery while faults that have serious service impact will result in more drastic reaction such as a large restart or reload of the software generic.
4. Insert a fault to test if the system provides automatic escalation of service recovery. Analyze and observe if the system can do the following:

    - Restore both hardware and software to a known state
    - Allow call/message processing to resume at a safe point
    - Successively encompass more software and hardware than the previous lower level
    - Restrict use of the levels that are service-affecting. In particular, levels affecting stable calls and recent change data shall require manual action. All other levels shall be allowed automatically.

5. Insert a fault so that manual service recovery is required. Test to see if multiple levels of initialization and reconfiguration capabilities are available to supplement automatic recovery capabilities.

B — Call-Processing Specific Recovery:

1. For systems that offer local recovery other than system-wide recovery actions, insert a fault to test each local recovery area to see if the system reacts as designed.

C — Database Management Recovery:

1. Insert transaction aborted failures to verify that the Database Management System's (DBMS) recovery mechanisms work as designed.
2. Insert locking conflict detection or resolution failures to verify that the DBMS's recovery mechanisms work as designed.
3. Insert time-out failures to verify that the DBMS's recovery mechanisms work as designed.

D — Memory Management Recovery:

1. Insert memory operation errors to verify that the memory management's error recovery is as designed.

E — Inter-Process Communication Recovery:

1. Insert message errors to verify that the integrity of the system is maintained as designed.

F — Network Management Recovery:

1. For network management functions, insert a fault to corrupt the network management function parameters to observe the adequacy of fault handling in the network. Example of parameters are:
   - Network performance parameters such as Answer-Seizure Ratio (ASR), percentage of overflow (%OFL), seizures per circuit per hour (SCHP), bits per circuit per hour (BCH), and route load (occupancy).
   - Exchange parameters such as exchange input load, code sender/receiver route load, and processor load.
   - Trunk route and destination performance parameters such as route overflow (congestion), route load, route blocking, route answer-seizure ratio, and route disturbance-seizure ratio (DSR).
2. Changeover Procedure: Insert a fault in the software to shift traffic from a failed link to a standby good link (e.g., CCS changeover software)
3. Link Restoration Procedure: Insert a fault in the software handling restoration from a failed link (e.g., CCS Signaling Link Restoration)
4. Transfer Prohibited Procedure: Insert a fault in the software handling transfer prohibited procedure (e.g., CCS TFP procedure).
5. Transfer Allowed Procedure : Insert a fault in the software handling transfer allowed procedure (e.g., CCS TFA procedure).
6. Transfer Controlled Procedure: Insert a fault in the software handling transfer controlled procedure (e.g., CCS TFC procedure).
7. Traffic Flow Control Procedure: Insert a fault in the software handling signaling traffic flow control procedure (e.g., CCS signaling traffic flow control procedure).

13.6.2 Test Plan for Audit Functions

There are a variety of audit programs in telecommunications system software that are used to examine the validity of programs and data, provide protection against memory errors, and generate alarms and printouts for trouble resolution. Bellcore's OTGR requirements (Section 4) highlights the general audit guidelines that include the following:

- Audit programs should run continuously to check data and programs for correctness and consistency, and to detect all data errors so errors can be corrected before affecting service.
- Audit programs that cannot run continuously shall run at least periodically.
- Audit programs to check for non-critical errors shall run at least periodically.
- Audit programs shall not detract from the quality of service.
- Audit programs shall correct detected errors or initialize the data to a state that would allow normal call or message processing to continue.
- Audit programs shall provide a record of detected errors and switching network element corrective actions.
- It shall be possible to manually inhibit selected audit programs and to have these inhibits reported periodically.
- The switching network element shall provide a notification when it has inhibited audits.
- The switching network element should provide a periodic automatic reminder of all active trouble detection inhibits.

There are many audit techniques that can be used in audit programs. The following are some examples:

- Redundant Software Checking.
- Overwriting of Temporary Memory.
- Image Checking.
- Consistency Checking.
- Range Check.
- Duration Check.
- Data Definition Check.
- Input/Output Check.
- Checksumming.
- Limbo States Detection.
- Point-to, Point-back Check.
- Initialization.

These general audit techniques are to help ensure the quality of individual customer service and maintain the sanity of the telecommunications software. Below is a sample set of test plans for software audits and system audits.

1. Software Audits

   (a) Physical audits on programs. Insert a fault to create a bit fault in either the program or data area to see if the bit fault will be detected (for example, through checksumming) and corrected by the audit program as designed.
   (b) Physical audits on data. Insert a fault such as data inconsistency or data integrity violation to see if the audit program can detect it.
   (c) Memory faults. Insert a fault to create illegal memory access, illegal write, pointer too large, array index out of range, memory not allocated, buffer overflow, buffer not allocated, or illegal reference to uninitialized variables to see the audit program can detect the fault.
   (d) Logical audits on residual design errors, operator input and output errors, translation errors, etc. Insert a fault to test logical errors that can be detected by techniques such as range check or duration check. Observe and analyze the system behavior to see if it reacts as designed.

2. System Audits

   (a) Call processing. Insert a fault to test the progress of call processing for individual calls.
   (b) Call completion. Insert a fault to test the audit program to see if it monitors the call completion rate.
   (c) Equipment. For hardware equipment such as lines, trunks, code senders and receivers, or disk storage, create a fault to simulate a hardware problem (such as device hanging) to see if the audit program can detect it.
   (d) Operating system. For memory allocation, queues, and buffers, create a fault to corrupt the memory or overflow the buffers to see if the audit program can detect the error and correct it.
   (e) Process errors. Insert a fault/error such as sending to the wrong address, deadlocks, live locks, process looping, process hung, or using too much resources to see if the audit program can detect it.
   (f) Message errors. Insert a fault/error such as lost message, corrupted message, misordered messages, duplicated messages or timeout waiting for message to see if the audit program can detect it.
   (g) Alarms. For each alarm situation, create a fault to activate the alarm to see if proper alarm level is activated and reported. Also observe if the alarm is reset after the problem has been resolved.
   (h) Network related errors. Insert an error to simulate network anomalies, such as network congestion (traffic overload), link oscillation, and routing error to see if the audit program can detect it.

13.7 APPLICATION AND RESULTS

The SFIT methodology proposed in this chapter has been applied to more than ten telecommunications systems, including end-office circuit switches, common channel signaling systems, signal transfer points, broadband Asynchronous Transfer Mode (ATM) systems, SONET-based Add-Drop Multiplexers (ADMs), and digital cross-connect systems. The total number lines of code for the system tested ranges from 500,000 to more than 5 millions. So far, most

of the tests following SFIT methodology were conducted by Bellcore technical analysis organization and network element software system suppliers. Bellcore has been performing SFIT as an agent to the telephone companies and is independent of the development organization of the network element suppliers. Since individual software systems and their developing environments vary from one to another, the preparation of software faults, actual insertion of software faults, and the results are different.

When supplier's development/testing/quality assurance organizations have good cooperation with the organization conducting SFIT, we have experienced a 2 to 10% of hit ratio (percentage of problem identified with respect to the total number of tests). This 2 to 10% of hit ratio includes both design/implementation deficiencies and actual software bugs in the program. Some examples of common software problems are:

- Incorrect condition check — This includes missing a condition check before processing information, incorrect boundary condition check (e.g., checked value being off by one), and incorrect semantics condition check (e.g., checking predicates contain incorrect value or variable).
- Use of incorrect data value — This is the result of using incorrect data value. For example, instead of using variables that will change values under different situations, the programmer hardcodes the actual number, thus provides inflexibility of the program. Another example is to access a variable with a wrong variable name which is similar to the intended variable.
- Missing or incorrect signal data — This type of fault is due to sending the incorrect signals to other software components or sending a correct signal but with incorrect data value.
- Incorrect set or reset of variables — For example, data is assigned with the incorrect value (or sometimes variable). Uninitialized variables or pointers are another examples.
- Incorrect branching — This type of fault is common in software system with "spaghetti code". In the program with many GOTO statements, it is very easy to branch into an incorrect function or subroutine. Incorrect branching often occurs in a IF-THEN-ELSE statement, where the programmer mistakenly reverse the THEN and ELSE condition.
- Missing or extra code — This is caused by some missing statements or some extraneous code. For example, missing an EXIT statement in a loop, or missing an instruction to reset the memory buffer.
- Incorrect execution order — The order of execution determines the sequence of actions in the software. When the order of execution is reversed, adverse condition can happen.

With software fault insertion testing, passing the test cases only means that the system reacts to the inserted fault as designed. It does not mean that the design is correct or sufficient. Our experience has shown that in many cases, the system reacts to the inserted faults as expected, however, can be improved if more fault tolerance is considered in the design. The following are some examples of design deficiencies found during SFIT:

- Lack of software audits. Software audits are used to detect and correct inconsistencies in the program and data. In many cases, we have found that consistencies checking of program and data are not adequate. Also, software audits should be run periodically (if not run continuously) with priorities assigned so it would not detract the service quality. However, these objectives were often ignored by some design.
- Inadequate system recovery. The current practice for many telecommunications system in reaction to software errors is to take a system-wide restart to clear all variables and counters. During this period, customers suffer loosing connection or no dial-tone. Depending on

the severity and frequency of errors, different level of recovery actions should be provided to minimize service impacts to customers. Also the time for recovery actions needs to be minimized. With many SFIT test cases, we are able to identify areas for service improvement if more levels of system recovery are available.

- Recovery action too drastic. This is related to the previous design deficiency. In many cases during SFIT, the result from a simple mistake (such as one subscriber line interface is hung) caused the system to restart, affecting thousands of subscribers. The system's reaction to this type of fault — taking a system restart — may resolve the problem, but it reveals a design deficiency with respect to minimizing service impacts to customers. Therefore, how to confine the problem to a smaller area and recover from it locally becomes an area for further investigation.
- Lack of system recovery escalation mechanism. When a local-level action to a software problem can not resolve the situation (such as clearing the hanging subscriber line interface in the previous example), a typical telecommunications system is to escalate the problem into a higher level. This is called recovery escalation. We have found that in many test cases there does not exist any guideline as to when to escalate the problem to a higher level. In other cases, we observed there are different recovery escalation mechanisms in a system, but are not followed consistently.

13.8 CONCLUSIONS

The telecommunications network and service providers have been demanding quality and reliability of software delivered to them from the telecommunications systems suppliers. As an agent to the regional telephone companies, Bellcore in conjunction with suppliers has been using SFIT to analyze and test circuit switches, common channel signaling systems, signal transfer points, broadband systems, add-drop multiplexers and digital cross-connect systems since 1992. Through putting SFIT into practice, SFIT has evolved to be an effective method to validate and improve the system's fault handling capabilities for telecommunications systems.

The important benefits resulting from SFIT include:

- Improvement in network/system robustness: This is achieved through more systematic analysis and testing, enhanced system recovery mechanisms, more software audit functions for the critical operations, and enhanced built-in exception handling routines.
- Improvement in network/system operations: This is achieved through operation guidelines or tools that help operators to recover from failures based on the rules, event patterns, and correlation derived from the analysis of observed behaviors in SFIT and field data.

Some SFIT experiences based on practices are summarized below:

- The software architecture analysis is an important first step before conducting SFIT. Through the analysis of the service manager and fault manager of the target system, one can identify the software components that are important for the testing.
- In order for a customer or any independent third party to test the software through SFIT, planning has to occur well ahead of the actual testing. For example, one must collect information with regard to the service manager and fault manager modules, negotiate with the supplier for expert support to create software patches, and schedule lab. time for the testing.

- The most effective way to improve system robustness is that telecommunications system suppliers make SFIT a standard part of their software development process.
- Participation from the supplier's software experts is extremely important for SFIT. Software patches need to be prepared before the test. Expert support should also be available to cooperatively develop detailed test cases and interpret the actual system behavior.
- Tools need to be developed or refined to make SFIT more efficient and automatic. Software development environment to support design for testability and observability can make SFIT more cost effective.
- Many design deficiencies in fault tolerance in various products have been found through SFIT, such as lack of software audits, inadequate recovery, and lack of recovery escalation mechanisms. These design deficiencies should be used as input to further enhance the fault tolerance requirements in the telecommunications community.
- SFIT helps identify the role of the various fault tolerance mechanisms systematically with respect to fault, error, and failure types. This information can be used to simplify and improve the design and implementation of the fault tolerance mechanisms.

ACKNOWLEDGEMENTS

We would like to acknowledge Dr. Chi-Ming Chen of Bellcore for his joint work related to this chapter. The summer job performed by Mr. Tsanchi Li of Purdue University also provided valuable input to this chapter.

REFERENCES

[Arl90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell. Fault injection for dependability validation — a methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.

[Arl92] J. Arlat. Fault injection for the experimental validation of fault-tolerant systems. In *Proc. Workshop Fault-Tolerant Systems*, pages 33–40, Kyoto, 1992.

[Avi87] A. Avižienis and D. Ball. On the achievement of a highly dependable and fault-tolerant air traffic control system. *IEEE Computer*, 20(20):84–90, February 1988.

[Avr92] D. Avresky, J. Arlat, J.-C. Laprie and Y. Crouzet. Fault injection for the formal testing of fault tolerance. In *Proc. 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 345–354, Boston, 1992.

[BCR90] TR-TSY-000929. *Reliability and Quality Measurements for Telecommunications Systems (RQMS)*, Issue 1, June 1990.

[BCR91] TR-TSY-000474. *OTGR Section4: Network Maintenance: Network Element (A Module of OTGR, FR-NWT-000439*, Issue 3, November 1898, plus Revision 2, July 1991, and Bulletin 1, November 1991.

[BCR92] SR-NWT-002419. *Software Architecture Review Checklists*, December 1992.

[BCR93] SR-NWT-002795, Bellcore Special Report. *Software Fault Insertion Testing Methodology*, December 1993.

[Coa93] Brian A. Coan and Daniel Heyman. Reliable software and communication III: congestion control and network reliability. *IEEE Journal on Selected Areas in Communication*, December 1993.

[Chi89] R. Chillarege and N. Bowen. Understanding large system failures — a fault injection experiment. In *Proc. 19th International Symposium on Fault-Tolerant Computing*, pages 356–363, 1989.

[Chi92] R. Chillarege et. al. Orthogonal defect classification — a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18:830–838, November 1986.

[Dem88] R. A. DeMillo, D. Guindi, W. McCracken, A. Offutt and K. King. Extended overview of the mothra software testing environment. *Second Workshop on Software Testing, Verification and Analysis*, pages 142–151, 1988.

[Dil91] T. Dilenno, D. Yaskin and J. Barton. Fault tolerance testing in the advanced automation system. In *Proc. 21th International Symposium on Fault-Tolerant Computing*, pages 18–25, 1991.

[Ech92] K. Echtle and M. Leu. The EFA fault injector for fault-tolerant distributed system testing. In *Proc. Workshop on Fault Tolerant Parallel and Distributed Systems*, pages 28–35, Amherst, 1992.

[Eri93] R. L. Erickson, N. D. Griffeth, M. Y. Lai and S. Y. Wang. Software architecture review for telecommunications software improvement. *IEEE International Conference on Communications (ICC'93)*, pages 616–620, May 1993.

[Hor92] Marc W. Hornbeek and Robert M. Caza. An integrated design and test strategy for large switching systems. *International Switching Symposium*, pages 444–448, 1992.

[Hor74] J. J. Horning, et. al. *A program structure for error detection and recovery*, pages 171–187 in *Lecture Notes in Computer Science 16*. Springer-Verlat, Berlin, 1974.

[Iye93] R. K. Iyer and D. Tang. Experimental Analysis of Computer System Dependability Technical Report CRHC-93-15, University of Illinois at Urbana-Champaign, 1993.

[Kan92] G. A. Kanawati, N. A. Kanawati and J. A. Abraham. FERRARI: a tool for the validation of system dependability properties. In *Proc. 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 336–244, Boston, 1992.

[Kao93] Wei-Lun Kao, Ravishankar K. Iyer and Dong Tang. FINE: a fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11), November 1993.

[Lai92] M. Lai, C. Chen, C. Hood and D. Saxena. Using software fault insertion to improve CCS network operation. *GLOBECOM'92*, pages 1723–1728, December 1992.

[Lev91] N. G. Leveson and S. S. Cha. Safety verification of ADA programs using software fault tree. *IEEE Software*, 8(4):48–59, July 1991.

[Li94] T. Li, C. M. Chen, R. Horgan, M. Lai, and S. Y. Wang. A software fault insertion testing methodology for improving the robustness of telecommunication systems. In *Proc. ICC 94*, May 1994.

[Lap92] J.-C. Laprie, editor, *Dependability: Basic Concepts and Terminology, Dependable Computing and Fault Tolerance*, Springer-Verlag, Vienna, Austria, 1992.

[NRC93] Network Reliability Council. *Network Reliability: A Report to the Nation*, June 1993.

[Ros93] H. Rosenberg and K. Shin. Software fault injection and its application in distributed systems. In *Proc. 23rd International Symposium on Fault-Tolerant Computing*, pages 208–217, 1993.

[Seg88] Z. Segal, et. al. FIAT — fault injection based automated testing environment. In *Proc. 18th International Symposium on Fault-Tolerant Computing*, pages 102–107, 1988.

[You93] C. R. Yount. *The Automatic Generation of Instruction-Level Error Manifestation of Hardware Faults: A New Fault Injection Model*. PhD dissertation, Carnegie-Mellon University, 1993.