CHAPTER 8

System Test Categories

As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.
— *Dave Parnas*

8.1 TAXONOMY OF SYSTEM TESTS

The objective of system-level testing, also called system testing, is to establish whether an implementation conforms to the requirements specified by the customers. It takes much effort to guarantee that customer requirements have been met and the system is acceptable. A variety of tests are run to meet a wide range of unspecified expectations as well. As integrated systems, consisting of both hardware and software components, are often used in reality, there is a need to have a much broader view of the behavior of the systems. For example, a telephone switching system not only is required to provide a connection between two users but also is expected to do so even if there are many ongoing connections below a certain upper limit. When the upper limit on the number of simultaneous connections is reached, the system is not expected to behave in an undesired manner. In this chapter, we identify different categories of tests in addition to the core functionality test. Identifying test categories brings us the following advantages:

- Test engineers can accurately focus on different aspects of a system, one at a time, while evaluating its quality.
- Test engineers can prioritize their tasks based on test categories. For example, it is more meaningful and useful to identify the limitations of a system only after ensuring that the system performs all basic functions to the test an engineer's satisfactions. Therefore, stress tests, which thrive to identify the limitations of a system, are executed after functionality tests.
- Planning the system testing phase based on test categorization lets a test engineer obtain a well-balanced test suite. Practical limitations make it difficult to be exhaustive, and economic considerations may restrict the

testing process from continuing any further. However, it is important to design a balanced test suite, rather than an unbalanced one with many test cases in one category and no tests in another.

In the following, first we present the taxonomy of system tests (Figure 8.1). Thereafter, we explain each category in detail.

- *Basic tests* provide an evidence that the system can be installed, configured, and brought to an operational state.
- *Functionality tests* provide comprehensive testing over the full range of the requirements within the capabilities of the system.
- *Robustness tests* determine how well the system recovers from various input errors and other failure situations.
- *Interoperability tests* determine whether the system can interoperate with other third-party products.
- *Performance tests* measure the performance characteristics of the system, for example, throughput and response time, under various conditions.
- *Scalability tests* determine the scaling limits of the system in terms of user scaling, geographic scaling, and resource scaling.
- *Stress tests* put a system under stress in order to determine the limitations of a system and, when it fails, to determine the manner in which the failure occurs.

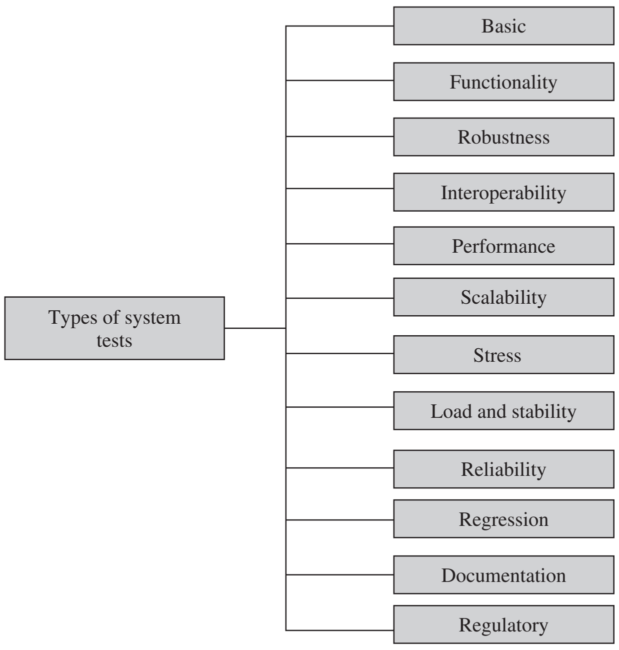

Figure 8.1 Types of system tests.

- *Load and stability tests* provide evidence that the system remains stable for a long period of time under full load.
- *Reliability tests* measure the ability of the system to keep operating for a long time without developing failures.
- *Regression tests* determine that the system remains stable as it cycles through the integration of other subsystems and through maintenance tasks.
- *Documentation tests* ensure that the system's user guides are accurate and usable.
- *Regulatory tests* ensure that the system meets the requirements of government regulatory bodies in the countries where it will be deployed.

8.2 BASIC TESTS

The basic tests (Figure 8.2) give a *prima facie* evidence that the system is ready for more rigorous tests. These tests provide limited testing of the system in relation to the main features in a requirement specification. The objective is to establish that there is sufficient evidence that a system can operate without trying to perform thorough testing. Basic tests are performed to ensure that commonly used functions, not all of which may directly relate to user-level functions, work to our satisfaction. We emphasize the fact that test engineers rely on the proper implementation of these functions to carry out tests for user-level functions. The following are the major categories of subsystems whose adequate testing is called the basic test.

8.2.1 Boot Tests

Boot tests are designed to verify that the system can boot up its software image (or build) from the supported boot options. The boot options include booting from ROM, FLASH card, and PCMCIA (Personal Computer Memory Card International Association) card. The minimum and maximum configurations of the system must be tried while booting. For example, the minimum configuration of a router consists of one line card in its slots, whereas the maximum configuration of a router means that all slots contains line cards.

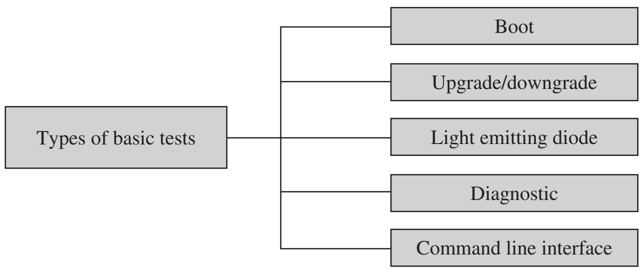

Figure 8.2 Types of basic tests.

8.2.2 Upgrade/Downgrade Tests

Upgrade/downgrade tests are designed to verify that the system software can be upgraded or downgraded (rollback) in a graceful manner from the previous version to the current version or vice versa. Suppose that the system is running the $(n-1)$th version of the software build and the new nth version of the software build is available. The question is how one upgrades the build from the $(n-1)$th version to the nth version. An upgradation process taking a system from the $(n-1)$th version to the nth version may not be successful, in which case the system is brought back to the $(n-1)$th version. Tests are designed in this subgroup to verify that the system successfully reverts back, that is, rolls back, to the $(n-1)$th version. An upgradation process may fail because of a number of different conditions: user-invoked abort (the user interrupts the upgrade process), in-process network disruption (the network environment goes down), in-process system reboot (there is a power glitch), or self-detection of upgrade failure (this is due to such things as insufficient disk space and version incompatibilities).

8.2.3 Light Emitting Diode Tests

The LED (light emitting diode) tests are designed to verify that the system LED status indicators function as desired. The LEDs are located on the front panels of the systems. These provide visual indication of the module operational status. For example, consider the status of a system chassis: Green indicates that the chassis is operational, off indicates that there is no power, and a blinking green may indicate that one or more of its submodules are faulty. The LED tests are designed to ensure that the visual operational status of the system and the submodules are correct. Examples of LED tests at the system and subsystem levels are as follows:

- System LED test: green = OK, blinking green = fault, off = no power.
- Ethernet link LED test: green = OK, blinking green = activity, off = fault.
- Cable link LED test: green = OK, blinking green = activity, off = fault.
- User defined T1 line card LED test: green = OK, blinking green = activity, red = fault, off = no power.

8.2.4 Diagnostic Tests

Diagnostic tests are designed to verify that the hardware components (or modules) of the system are functioning as desired. It is also known as the built-in self-test (BIST). Diagnostic tests monitor, isolate, and identify system problems without manual troubleshooting. Some examples of diagnostic tests are as follows:

- **Power-On Self-Test (POST):** This is a set of automatic diagnostic routines that are executed during the boot phase of each submodule in the system. The POSTs are intended to determine whether or not the hardware is in a proper state to execute the software image. It is not intended to be comprehensive in the analysis of the hardware; instead, it provides a high level of confidence that the hardware is operational. The POSTs execute on the following kinds of elements:

Memory

Address and data buses

Peripheral devices

- **Ethernet Loop-Back Test:** This test generates and sends out the desired number, which is a tunable parameter, of packets and expects to receive the same number of Ethernet packets through the loop-back interface—external or internal. If an error occurs (e.g., packet mismatch or timeout), an error message indicating the type of error, its probable cause(s), and recommended action(s) is displayed on the console. The data sent out are generated by a random-number generator and put into a data buffer. Each time a packet is sent, it is selected from a different starting point of the data buffer, so that any two consecutively transmitted packets are unlikely to be identical. These tests are executed to ensure that the Ethernet card is functioning as desired.
- **Bit Error Test (BERT):** The on-board BERT provides standard bit error patterns, which can be transmitted over a channel for diagnostic purpose. BERT involves transmitting a known bit pattern and then testing the transmitted pattern for errors. The ratio of the number of bits with errors to the total number of bits transmitted is called the bit error rate (BER). Tests are designed to configure all BERTs from the command line interface (CLI). These tests are executed to ensure that that the hardware is functioning as desired.

8.2.5 Command Line Interface Tests

The CLI tests are designed to verify that the system can be configured, or provisioned, in specific ways. This is to ensure that the CLI software module processes the user commands correctly as documented. This includes accessing the relevant information from the system using CLI. In addition to the above tests, test scenarios may be developed to verify the error messages displayed.

8.3 FUNCTIONALITY TESTS

Functionality tests (Figure 8.3) verify the system as thoroughly as possible over the full range of requirements specified in the requirements specification document. This category of tests is partitioned into different functionality subgroups as follows.

8.3.1 Communication Systems Tests

Communication systems tests are designed to verify the implementation of the communication systems as specified in the customer requirements specification.

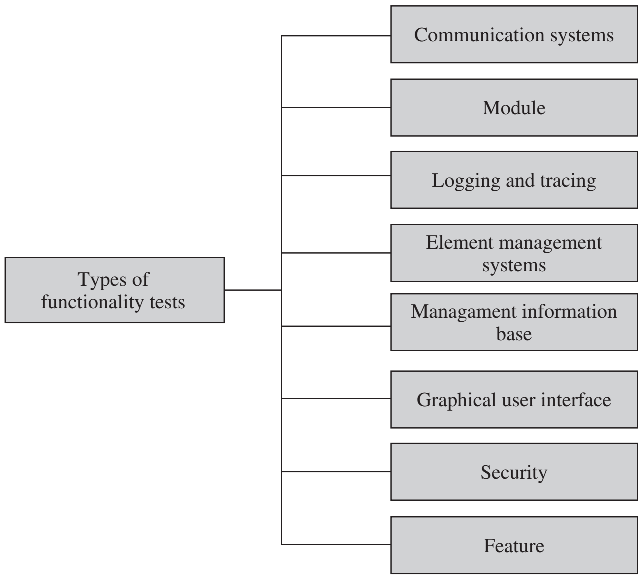

Figure 8.3 Types of functionality tests.

For example, one of the customer requirements can be to support Request for Comment (RFC) 791, which is the Internet Protocol (IP) specification. Tests are designed to ensure that an IP implementation conforms to the RFC791 standard. Four types of communication systems tests are recommended according to the extent to which they provide an indication of conformance [1]:

- *Basic interconnection tests* provide evidence that an implementation can establish a basic connection before thorough testing is performed.
- *Capability tests* check that an implementation provides the observable capabilities based on the static communication systems requirements. The static requirements describes the options, ranges of values for parameters, and timers.
- *Behavior tests* endeavor to verify the dynamic communication systems requirements of an implementation. These are the requirements and options that define the observable behavior of a protocol. A large part of behavior tests, which constitute the major portion of communication systems tests, can be generated from the protocol standards.
- *Systems resolution tests* probe to provide definite “yes” or “no” answers to specific requirements.

8.3.2 Module Tests

Module tests are designed to verify that all the modules function individually as desired within the systems. Mutual interactions among the modules glue these components together into a whole system. The idea here is to ensure that individual modules function correctly within the whole system. One needs to verify that the system, along with the software that controls these modules, operate as specified in the requirement specification. For example, an Internet router contains modules such as line cards, system controller, power supply, and fan tray. Tests are designed to verify each of the functionalities. For Ethernet line cards, tests are designed to verify (i) autosense, (ii) latency, (iii) collisions, (iv) frame types, and (v) frame lengths. Tests are designed to ensure that the fan status is accurately read, reported by the software, and displayed in the supply module LEDs (one green "in service" and one red "out of service"). For T1/E1 line cards, tests are designed to verify:

- **Clocking:** Internal (source timing) and receive clock recovery (loop timing).
- **Alarms:** Detection of loss of signal (LOS), loss of frame (LOF), alarm indication signal (AIS), and insertion of AIS.
- **Line Coding:** Alternate mark inversion (AMI) for both T1 and E1, bipolar 8 zero substitution (B8ZS) for T1 only, and high-density bipolar 3 (HDB3) for E1 only.
- **Framing:** Digital signal 1 (DS1) and E1 framing.
- **Channelization:** Ability to transfer user traffic across channels multiplexed from one or more contiguous or non contiguous time slots on a T1 or E1 link.

8.3.3 Logging and Tracing Tests

Logging and tracing tests are designed to verify the configurations and operations of logging and tracing. This also includes verification of "flight data recorder: non-volatile flash memory" logs when the system crashes. Tests may be designed to calculate the impact on system performance when all the logs are enabled.

8.3.4 Element Management Systems Tests

The EMS tests verify the main functions which are to manage, monitor, and upgrade the communication system network elements (NEs). Table 8.1 summarizes the functionalities of an EMS. An EMS communicates with its NEs by using, for example, the Simple Network Management Protocol (SNMP) [2] and a variety of proprietary protocols and mechanisms.

An EMS is a valuable component of a communication systems network. Not all EMSs will perform all of the tasks listed in Table 8.1. An EMS can support a subset of the tasks. A user working through the EMS graphical user interface (GUI) may accomplish some or all of the tasks. Remote access to an EMS allows the operators to access management and control information from any location. This facilitates the deployment of a distributed workforce that can rapidly respond to failure notifications. This means that thin client workstations can operate over the Internet and service provider intranets. In this subgroup, tests are designed to

TABLE 8.1 EMS Functionalities

Fault Management	Configuration Management	Accounting Management	Performance Management	Security Management
Alarm handling	System turn-up	Track service usage	Data collection	Control NE access
Trouble detection	Network provisioning	Bill for services	Report generation	Enable NE functions
Trouble correction	Autodiscovery		Data analysis	Access logs
Test and acceptance	Back-up and restore			
Network recovery	Database handling			

verify the five functionalities of an EMS (Table 8.1). This includes both the EMS client and the EMS server. Examples of EMS tests are given below.

- **Auto-Discovery:** EMS discovery software can be installed on the server to discover elements attached to the EMS through the IP network.
- **Polling and Synchronization:** An EMS server detects a system unreachable condition within a certain time duration. The EMS server synchronizes alarm status, configuration data, and global positioning system (GPS) time from the NE.
- **Audit Operations:** An audit mechanism is triggered whenever an out-of-service network element comes back. The mechanism synchronizes alarms between out-of-service NEs coming back online and the EMS.
- **Fault Manager:** Generation of critical events, such as reboot and reset, are converted to an alert and stored in the EMS database. The EMS can send an email/page to a configured address when an alarm is generated.
- **Performance Manager:** Data are pushed to the EMS server when the data buffer is full in the NE. Data in the server buffer are stored in the backup files once it is full.
- **Security Manager:** It supports authentication and authorization of EMS clients and NEs. The EMS server does the authorization based on user privileges.
- **Policies:** An EMS server supports schedulable log file transfer from the system. The EMS database is periodically backed up to a disk.
- **Logging:** An EMS server supports different logging levels for every major module to debug. An EMS server always logs errors and exceptions.
- **Administration and Maintenance:** This test configures the maximum number of simultaneous EMS clients. The EMS server backs up and restores database periodically.

- **Invoke Clients:** Several clients can be invoked to interact with the EMS server.
- **Live Data View:** A client can provide a live data viewer to monitor performance and show statistics of the system.
- **System Administration Task:** A client can shut down the server, configure logging levels, display server status, and enable auto-discovery.
- **File Transfer:** A client can check on-demand file transfer with a progress bar to indicate the progress and abort an on-demand file transfer operation.

SNMP Example The SNMP is an application layer protocol that facilitates the exchange of management information between network elements. The SNMP is a part of the Internet network management architecture consisting of three components: *network elements*, *agents*, and *network management stations* (NMSs). A NE is a network node that contains an SNMP agent and that resides on a managed network. Network elements collect and store management information and make this information available to the NMS over the SNMP protocol. Network elements can be routers, servers, radio nodes, bridges, hubs, computer hosts, printers, and modems. An agent is a network management software module that (i) resides on a NE, (ii) has the local knowledge of management information, and (iii) translates that information into a form compatible with the SNMP. An NMS, sometimes referred to as a console, executes management applications to monitor and control network elements. One or more NMSs exist on each managed networks. An EMS can act as an NMS.

A management information base (MIB) is an important component of a network management system. The MIB identifies the network elements (or managed objects) that are to be managed. Two types of managed objects exist:

- Scalar objects define a single object instance.
- Tabular objects define multiple related object instances that are grouped in MIB tables.

Essentially, a MIB is a virtual store providing a model of the managed information. For example, a MIB can contain information about the number of packets that have been sent and received across an interface. It contains statistics on the number of connections that exist on a Transmission Control Protocal (TCP) port as well as information that describes each user's ability to access elements of the MIB. The SNMP does not operate on the managed objects directly; instead, the protocol operates on a MIB. In turn, a MIB is the reflection of the managed objects, and its management mechanism is largely proprietary, perhaps through the EMS.

The important aspect of a MIB is that it defines (i) the elements that are managed, (ii) how a user accesses them, and (iii) how they can be reported. A MIB can be depicted as an abstract tree with an unnamed root; individual items are represented as leaves of the tree. An object identifier uniquely identifies a MIB object in the tree. The organization of object identifiers is similar to a telephone number hierarchy; they are organized hierarchically with specific digits assigned by different organizations. The Structure of Management Information (SMI) defines the rules for describing the management information. The SMI specifies that all managed objects have a name, a syntax, and an encoding mechanism. The name is used as the object identifier. The syntax defines the data type of the object. The SMI syntax uses a subset of the Abstract Syntax Notation One (ASN.1) definitions. The encoding mechanism describes how the information associated with the managed object is formatted as a series of data items for transmission on the network.

Network elements are monitored and controlled using four basic SNMP commands: *read*, *write*, *trap*, and traversal operations. The read command is used by an NMS to monitor NEs. An NMS examines different variables that are maintained by NEs. The write command is used by an NMS to control NEs. With the write command, an NMS changes the values of variables stored within NEs. The trap command is used by NEs to asynchronously report events to an NMS. When certain types of events occur, an NE sends a trap to an NMS. The traversal operations are used by the NMS to determine the variables an NE supports and to sequentially gather information in a variable table, such as a routing table.

The SNMP is a simple request/response protocol that can send multiple requests. Six SNMP operations are defined. The Get operation allows an NMS to retrieve an object instant from an agent. The GetNext operation allows an NMS to retrieve the next object instance from a table or list within an agent. The GetBulk operation allows an NMS to acquire large amounts of related information without repeatedly invoking the GetNext operation. The Set operation allows an NMS to set values for object instances within an agent. The Trap operation is used by an agent to asynchronously inform an NMS of some event. The Inform operation allows one NMS to send trap information to another.

The SNMP messages consist of two parts: a header and a protocol data unit (PDU). The message header contains two fields, namely, a version number and a community name. A PDU has the following fields: PDU type, request ID, error status, error index, and variable bindings. The following descriptions summarize the different fields:

- **Version Number:** The version number specifies the version of the SNMP being used.
- **Community Name:** This defines an access environment for a group of NMSs. NMSs within a community are said to exist within the same administrative domain. Community names serve as a weak form of authentication because devices that do not know the proper community name are precluded from SNMP operations.
- **PDU Type:** This specifies the type of PDU being transmitted.
- **Request ID:** A request ID is associated with the corresponding response.
- **Error Status:** This indicates an error and shows an error type. In GetBulk operations, this field becomes a nonrepeater field by defining the number of requested variables listed that should be retrieved no more than once from the beginning of the request. The field is used when some of the variables are scalar objects with one variable.

- **Error Index:** An error index associates the error with a particular object instance. In GetBulk operations, this field becomes a Max-repetitions field. This field defines the maximum number of times that other variables beyond those specified by the nonrepeater field should be retrieved.
- **Variable Bindings (varbinds):** This comprises the data of an SNMP PDU. Variables bindings associate particular object instances with their current values (with the exception of Get and GetNext requests, for which the value is ignored).

8.3.5 Management Information Base Tests

The MIB tests are designed to verify (i) standard MIBs including MIB II and (ii) enterprise MIBs specific to the system. Tests may include verification of the agent within the system that implements the objects it claims to. Every MIB object is tested with the following primitives: Get, GetNext, GetBulk, and Set. It should be verified that all the counters related to the MIBs are incremented correctly and that the agent is capable of generating (i) well known traps (e.g., coldstart, warmstart, linkdown, linkup) and (ii) application-specific traps (X.25 restart and X.25 reset).

8.3.6 Graphical User Interface Tests

In modern-day software applications, users access functionalities via GUIs. Users of the client-server technology find it convenient to use GUI based applications. The GUI tests are designed to verify the interface to the users of an application. These tests verify different components (objects) such as icons, menu bars, dialogue boxes, scroll bars, list boxes, and radio buttons. Ease of use (usability) of the GUI design and output reports from the viewpoint of actual system users should be checked. Usefulness of the online help, error messages, tutorials, and user manuals are verified. The GUI can be utilized to test the functionality behind the interface, such as accurate response to database queries. GUIs need to be compatible, as discussed in Section 8.5, and consistent across different operating systems, environments, and mouse-driven and keyboard-driven inputs.

Similar to GUI testing, another branch of testing, called *usability testing*, has been evolving over the past several years. The usability characteristics which can be tested include the following:

- **Accessibility:** Can users enter, navigate, and exit with relative ease?
- **Responsiveness:** Can users do what they want and when they want in a way that is clear? It includes ergonomic factors such as color, shape, sound, and font size.
- **Efficiency:** Can users do what they want with a minimum number of steps and time?
- **Comprehensibility:** Do users understand the product structure with a minimum amount of effort?

8.3.7 Security Tests

Security tests are designed to verify that the system meets the security requirements: *confidentiality*, *integrity*, and *availability*. Confidentiality is the requirement that data and the processes be protected from unauthorized disclosure. Integrity is the requirement that data and process be protected from unauthorized modification. Availability is the requirement that data and processes be protected from the denial of service to authorized users. The security requirements testing approach alone demonstrates whether the stated security requirements have been satisfied regardless of whether or not those requirements are adequate. Most software specifications do not include negative and constraint requirements. Security testing should include negative scenarios such as misuse and abuse of the software system. The objective of security testing is to demonstrate [3] the following:

- The software behaves securely and consistently under all conditions—both expected and unexpected.
- If the software fails, the failure does not leave the software, its data, or its resources to attack.
- Obscure areas of code and dormant functions cannot be compromised or exploited.
- Interfaces and interactions among components at the application, framework/middleware, and operating system levels are consistently secure.
- Exception and error handling mechanisms resolve all faults and errors in ways that do not leave the software, its resources, its data, or its environment vulnerable to unauthorized modification or denial-of-service attack.

The popularity of the Internet and wireless data communications technologies have created new types security threats, such as un-authorized access to the wireless data networks, eavesdropping on transmitted data traffic, and denial of service attack [4]. Even within an enterprise, wireless local area network intruders can operate inconspicuously because they do not need a physical connection to the network [5]. Several new techniques are being developed to combat these kinds of security threats. Tests are designed to ensure that these techniques work—and this is a challenging task. Useful types of security tests include the following:

- Verify that only authorized accesses to the system are permitted. This may include authentication of user ID and password and verification of expiry of a password.
- Verify the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.
- Verify that illegal reading of files, to which the perpetrator is not authorized, is not allowed.
- Ensure that virus checkers prevent or curtail entry of viruses into the system.
- Ensure that the system is available to authorized users when a zero-day attack occurs.

- Try to identify any "backdoors" in the system usually left open by the software developers. Buffer overflows are the most commonly found vulnerability in code that can be exploited to compromise the system. Try to break into the system by exploiting the backdoors.
- Verify the different protocols used by authentication servers, such as Remote Authentication Dial-in User Services (RADIUS), Lightweight Directory Access Protocol (LDAP), and NT LAN Manager (NTLM).
- Verify the secure protocols for client–server communications, such as the Secure Sockets Layer (SSL). The SSL provides a secure channel between clients and servers that choose to use the protocol for web sessions. The protocol serves two functions: (i) authenticate the web servers and/or clients and (ii) encrypt the communication channel.
- Verify the IPSec protocol. Unlike the SSL, which provides services at layer 4 and secures the communications between two applications, IPSec works at layer 3 and secures communications happening on the network.
- Verify different wireless security protocols, such as the Extensible Authentication Protocol (EAP), the Transport Layer Security (TLS) Protocol, the Tunneled Transport Layer Security (TTLS) Protocol, and the Protected Extensible Authentication Protocol (PEAP).

8.3.8 Feature Tests

Feature tests are designed to verify any additional functionalities which are defined in the requirement specifications but not covered in the above categories. Examples of those tests are data conversion and cross-functionality tests. Data conversion testing is testing of programs or procedures that are used to convert data from an existing system to a replacement system. An example is testing of a migration tool that converts a Microsoft Access database to MySQL format. Cross-functionality testing provides additional tests of the interdependencies among functions. For example, the verification of the interactions between NEs and an element management system in a 1xEV-DO wireless data network, as illustrated later in Figure 8.5, is considered as cross-functionality testing.

8.4 ROBUSTNESS TESTS

Robustness means how sensitive a system is to erroneous input and changes in its operational environment. Tests in this category are designed to verify how gracefully the system behaves in error situations and in a changed operational environment. The purpose is to deliberately break the system, not as an end in itself, but as a means to find error. It is difficult to test for every combination of different operational states of the system or undesirable behavior of the environment. Hence, a reasonable number of tests are selected from each group illustrated in Figure 8.4 and discussed below.

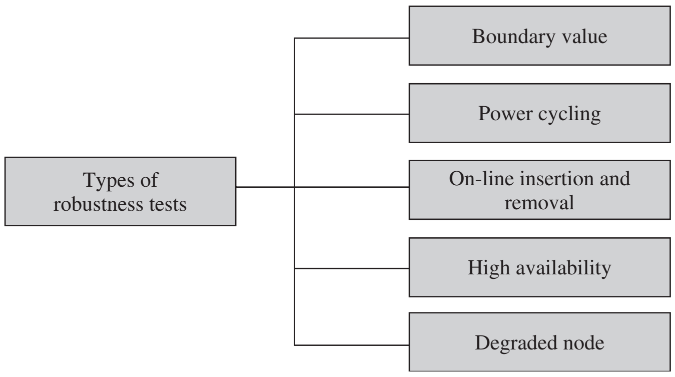

Figure 8.4 Types of robustness tests.

8.4.1 Boundary Value Tests

Boundary value tests are designed to cover boundary conditions, special values, and system defaults. The tests include providing invalid input data to the system and observing how the system reacts to the invalid input. The system should respond with an error message or initiate an error processing routine. It should be verified that the system handles boundary values (below or above the valid values) for a subset of configurable attributes. Examples of such tests for the SNMP protocol are as follows:

- Verify that an error response wrong_type is generated when the Set primitive is used to provision a variable whose type does not match the type defined in the MIB.
- Verify that an error response wrong_value is generated when the Set primitive is used to configure a varbind list with one of the varbinds set to an invalid value. For example, if the varbind can have values from set 0,1,2,3, then the input value can be – 1 to generate a wrong_value response.
- Verify that an error response too_big is generated when the Set primitive is used to configure a list of 33 varbinds. This is because the Set primitive accepts 32 varbinds at a time.
- Verify that an error response not_writable is generated when the Set primitive is used to configure a variable as defined in the MIB.
- Assuming that the SNMP protocol can support up to 1024 communities, verify that it is not possible to create the 1025th community.

Examples of robustness tests for the 1xEV-DO network, as shown in Figure 8.5, are as follows:

- Assuming that an EMS can support up to 300 NEs, verify that the EMS cannot support the 301st NE. Check the error message from the EMS when the user tries to configure the 301st network element.

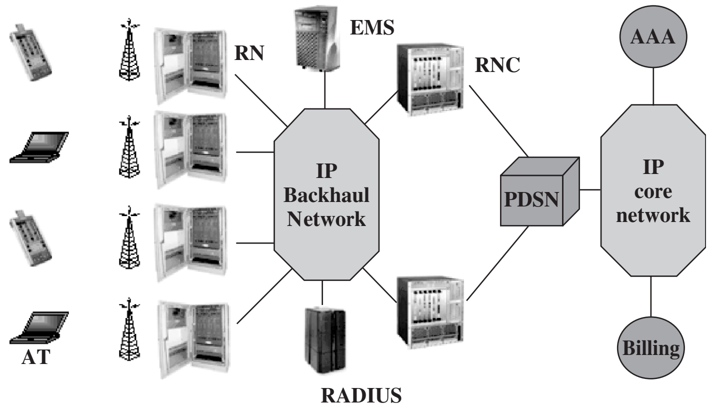

Figure 8.5 Typical 1xEV-DO radio access network. (Courtesy of Airvana, Inc.)

- Assuming that an RNC can support 160,000 simultaneous sessions, verify that the 160,001th session cannot be established on an RNC. Check the error message from the EMS when the user tries to establish the 160,001th session.
- Assuming that a base transceiver station (BTS) can support up to 93 simultaneous users, verify that the 94th user cannot be connected to a BTS.

8.4.2 Power Cycling Tests

Power cycling tests are executed to ensure that, when there is a power glitch in a deployment environment, the system can recover from the glitch to be back in normal operation after power is restored. As an example, verify that the boot test is successful every time it is executed during power cycling.

8.4.3 On-Line Insertion and Removal Tests

On-line insertion and removal (OIR) tests are designed to ensure that on-line insertion and removal of modules, incurred during both idle and heavy load operations, are gracefully handled and recovered. The system then returns to normal operation after the failure condition is removed. The primary objective is to ensure that the system recovers from an OIR event without rebooting or crashing any other components. OIR tests are conducted to ensure the fault-free operation of the system while a faulty module is replaced. As an example, while replacing an Ethernet card, the system should not crash.

8.4.4 High-Availability Tests

High-availability tests are designed to verify the redundancy of individual modules, including the software that controls these modules. The goal is to verify that the system gracefully and quickly recovers from hardware and software failures without adversely impacting the operation of the system. The concept of high availability is also known as fault tolerance. High availability is realized by means of proactive methods to maximize service up-time and to minimize the downtime. One module operates in the active mode while another module is in the standby mode to achieve $1 + 1$ redundancy. For this mode of operation, tests are designed to verify the following:

- A standby module generates an OIR event, that is, hot swapped, without affecting the normal operation of the system.
- The *recovery time* does not exceed a predefined limit while the system is operational. Recovery time is the time it takes for an operational module to become a standby module and the standby module to become operational.
- A server can automatically switch over from an active mode to a standby mode in case a fail-over event occurs. A fail-over is said to occur when a standby server takes over the workload of an active server.

Tests can be designed to verify that a fail-over does not happen without any *observable failure*. A fail-over without an observable failure is called *silent fail-over*. This can only be observed during the load and stability tests described in Section 8.9. Whenever a silent fail-over occurs, a causal analysis must be conducted to determine its cause.

8.4.5 Degraded Node Tests

Degraded node (also known as failure containment) tests verify the operation of a system after a portion of the system becomes nonoperational. It is a useful test for all mission-critical applications. Examples of degraded node tests are as follows:

- Cut one of the four T1 physical connections from one router to another router and verify that load balancing occurs among the rest of the three T1 physical connections. Confirm that packets are equally distributed among the three operational T1 connections.
- Disable the primary port of a router and verify that the message traffic passes through alternative ports with no discernible interruption of service to end users. Next, reactivate the primary port and verify that the router returns to normal operation.

Example of 1xEV-DO Wireless Data Networks The code division multiple-access (CDMA) 2000 1xEV-DO (one-time evolution, data only) is a standardized technology to deliver high data rate at the air interface between an access terminal (AT) and a base transceiver station (BTS), also known as a radio node (RN) [6–8]. The 1xEV-DO Revision 0 delivers a peak data rate of 2.54 Mbits/s on the forward link (from a BTS to an AT) using only 1.25 MHz of spectrum width and a peak data rate of 153.6 kbits/s on the reverse link. We show an architecture for connecting all the BTS with the Internet (IP core network) in Figure 8.5. In this architecture, a base station controller (BSC), also known as a radio network controller (RNC), need not be directly connected by dedicated, physical links with a set of BTSs. Instead, the BTSs are connected to the RNCs via an IP back-haul network. Such an interconnection results in flexible control of the BTSs by the RNCs. The RNCs are connected with the Internet (IP core network) via one or more packet data serving nodes (PDSNs). Finally, the EMS allows the operator to manage the 1xEV-DO network.

The ATs (laptop, PDA, mobile telephone) implement the end-user side of the 1xEV-DO and TCP/IP. The ATs communicate with the RNs over the 1xEV-DO airlink. The RNs are the components that terminate the airlink to/from the ATs. The functions performed by the RNs are (i) control and processing of the physical airlink, (ii) processing of the 1xEV-DO media access control (MAC) layer, and (iii) communication via a back-haul network to the RNC. In addition, RNs in conjunction with the RNCs perform the softer handoff mobility function of the 1xEV-DO protocol, where an AT is in communication with multiple sector antennas of the same RN.

An RNC is an entity that terminates the higher layer components of the 1xEV-DO protocol suite. An RNC has logical interfaces to RNs, the authentication, authorization, and accounting (AAA) servers, other RNCs, and the PDSN. An RNC terminates 1xEV-DO signaling interactions from the ATs and processes the user traffic to pass it on to the PDSN. It manages radio resources across all the RNs in its domain and performs mobility management in the form of softer and soft handoffs. The AAA servers are carrier-class computing devices running RADIUS protocol and having an interface to a database. These servers may be configured for two AAA functions, namely, access network AAA and core network AAA. The access network AAA is connected to the RNCs, which perform the terminal authentication function. The core network AAA is connected to the PDSN, which performs user authentication at the IP level. The PDSN is a specialized router implementing IP and mobile IP. The PDSN may be implemented as a single, highly available device or as multiple devices clustered together to form a high-availability device. The PDSN is the edge of the core IP network with respect to the AT, that is, the point where (i) the Point-to-Point Protocol (PPP) traffic of the AT is terminated, (ii) the user is authenticated, and (iii) the IP service options are determined. The core IP network is essentially a network of routers. The EMS server is a system that directly controls the NEs. It is responsible for fault handling, network configuration, and statistics management of the network interfaces. The EMS server interfaces with the NEs via TCP/IP. The EMS server is accessed via a client—and not directly—by the network management staff of a wireless operator. An EMS client is a workstation from which a network operator manages the radio access network.

8.5 INTEROPERABILITY TESTS

In this category, tests are designed to verify the ability of the system to interoperate with third-party products. An interoperability test typically combines different network elements in one test environment to ensure that they work together. In other words, tests are designed to ensure that the software can be connected with other systems and operated. In many cases, during interoperability tests, users may require the hardware devices to be interchangeable, removable, or reconfigurable. Often, a system will have a set of commands or menus that allow users to make the configuration changes. The reconfiguration activities during interoperability tests are known as *configuration testing* [9]. Another kind of interoperability test is called a (*backward*) *compatibility test*. Compatibility tests verify that the system works the same way across different platforms, operating systems, and database management systems. Backward compatibility tests verify that the current software build flawlessly works with older version of platforms. As an example, let us consider a 1xEV-DO radio access network as shown in Figure 8.5. In this scenario, tests are designed to ensure the interoperability of the RNCs with the following products from different vendors: (i) PDSN, (ii) PDA with 1xEV-DO card, (iii) AAA server, (iv) PC with 1xEV-DO card, (v) laptop with 1xEV-DO card, (vi) routers from different vendors, (vii) BTS or RNC, and (viii) switches.

8.6 PERFORMANCE TESTS

Performance tests are designed to determine the performance of the actual system compared to the expected one. The performance metrics needed to be measured vary from application to application. An example of expected performance is: The response time should be less than 1 millisecond 90% of the time in an application of the "push-to-talk" type. Another example of expected performance is: A transaction in an on-line system requires a response of less than 1 second 90% of the time. One of the goals of router performance testing is to determine the system resource utilization, for maximum aggregation throughput rate considering zero drop packets. In this category, tests are designed to verify response time, execution time, throughput, resource utilization, and traffic rate.

For performance tests, one needs to be clear about the specific data to be captured in order to evaluate performance metrics. For example, if the objective is to evaluate the response time, then one needs to capture (i) end-to-end response time (as seen by external user), (ii) CPU time, (iii) network connection time, (iv) database access time, (v) network connection time, and (vi) waiting time.

Some examples of performance test objectives for an EMS server are as follows:

- Record the CPU and memory usage of the EMS server when 5, 10, 15, 20, and 25 traps per second are generated by the NEs. This test will validate the ability of the EMS server to receive and process those number of traps per second.
- Record the CPU and memory usage of the EMS server when log files of different sizes, say, 100, 150, 200, 250 and 300 kb, are transferred from NEs to the EMS server once every 15 minutes.

Some examples of performance test objectives of SNMP primitives are as follows:

- Calculate the response time of the Get primitive for a single varbind from a standard MIB or an enterprise MIB.

- Calculate the response time of the GetNext primitive for a single varbind from a standard MIB or an enterprise MIB.
- Calculate the response time of the GetBulk primitive for a single varbind from a standard MIB or an enterprise MIB.
- Calculate the response time of the Set primitive for a single varbind from a standard MIB or an enterprise MIB.

Some examples of performance test objectives of a 1xEV-DO Revision 0 are as follows:

- Measure the maximum BTS forward-link throughput.
- Measure the maximum BTS reverse-link throughput.
- Simultaneously generate maximum-rate BTS forward- and reverse-link data capacities.
- Generate the maximum number of permissible session setups per hour.
- Measure the AT-initiated connection setup delay.
- Measure the maximum BTS forward-link throughput per sector carrier for 16 users in the 3-km/h mobility model.
- Measure the maximum BTS forward-link throughput per sector carrier for 16 users in the 30-km/h mobility model.

The results of performance are evaluated for their acceptability. If the performance metric is unsatisfactory, then actions are taken to improve it. The performance improvement can be achieved by rewriting the code, allocating more resources, and redesigning the system.

8.7 SCALABILITY TESTS

All man-made artifacts have engineering limits. For example, a car can move at a certain maximum speed in the best of road conditions, a telephone switch can handle a certain maximum number of calls at any given moment, a router has a certain maximum number of interfaces, and so on. In this group, tests are designed to verify that the system can scale up to its engineering limits. A system may work in a limited-use scenario but may not scale up. The run time of a system may grow exponentially with demand and may eventually fail after a certain limit. The idea is to test the limit of the system, that is, the magnitude of demand that can be placed on the system while continuing to meet latency and throughput requirements. A system which works acceptably at one level of demand may not scale up to another level. Scaling tests are conducted to ensure that the system response time remains the same or increases by a small amount as the number of users are increased. Systems may scale until they reach one or more engineering limits. There are three major causes of these limitations:

i. Data storage limitations—limits on counter field size and allocated buffer space

ii. Network bandwidth limitations—Ethernet speed 10 Mbps and T1 card line rate 1.544 Mbps

iii. Speed limit—CPU speed in megahertz

Extrapolation is often used to predict the limit of scalability. The system is tested on an increasingly larger series of platforms or networks or with an increasingly larger series of workloads. Memory and CPU utilizations are measured and plotted against the size of the network or the size of the load. The trend is extrapolated from the measurable and known to the large-scale operation. As an example, for a database transaction system calculate the system performance, that is, CPU utilization and memory utilization for 100, 200, 400, and 800 transactions per second, then draw graphs of number of transactions against CPU and memory utilization. Extrapolate the measured results to 20,0000 transactions. The drawback in this technique is that the trend line may not be accurate. The system behavior may not degrade gradually and gracefully as the parameters are scaled up. Examples of scalability tests for a 1xEV-DO network are as follows:

- Verify that the EMS server can support the maximum number of NEs, say, 300, without any degradation in EMS performance.
- Verify that the maximum number of BTS, say, 200, can be homed onto one BSC.
- Verify that the maximum number of EV-DO sessions, say, 16,000, can be established on one RNC.
- Verify that the maximum number of EV-DO connections, say, 18,400, can be established on one RNC.
- Verify that the maximum BTS capacity for the three-sector configuration is 93 users per BTS.
- Verify the maximum softer handoff rate with acceptable number of call drops per BTS. Repeat the process every hour for 24 hours.
- Verify the maximum soft handoff rate with no call drops per BTS. Repeat the process for every hour for 24 hours.

8.8 STRESS TESTS

The goal of stress testing is to evaluate and determine the behavior of a software component while the offered load is in excess of its designed capacity. The system is deliberately stressed by pushing it to and beyond its specified limits. Stress tests include deliberate contention for scarce resources and testing for incompatibilities. It ensures that the system can perform acceptably under worst-case conditions under an expected peak load. If the limit is exceeded and the system does fail, then the recovery mechanism should be invoked. Stress tests are targeted to bring out the problems associated with one or more of the following:

- Memory leak
- Buffer allocation and memory carving

One way to design a stress test is to impose the maximum limits on all system performance characteristics at the same time, such as the response time, availability, and throughput thresholds. This literally provides the set of worst-case conditions under which the system is still expected to operate acceptably.

The best way to identify system bottlenecks is to perform stress testing from different locations inside and outside the system. For example, individually test each component of the system, starting with the innermost components that go directly to the core of the system, progressively move outward, and finally test from remote locations far outside the system. Testing each link involves pushing it to its full-load capacity to determine the correct operation. After all the individual components are tested beyond their highest capacity, test the full system by simultaneously testing all links to the system at their highest capacity. The load can be deliberately and incrementally increased until the system eventually does fail; when the system fails, observe the causes and locations of failures. This information will be useful in designing later versions of the system; the usefulness lies in improving the robustness of the system or developing procedures for a disaster recovery plan. Some examples of stress tests of a 1xEV-DO network are as follows:

- Verify that repeated establishment and teardown of maximum telnet sessions to the BSC and BTS executed over 24 hours do not result in (i) leak in the number of buffers or amount of memory or (ii) significant increase in the degree of fragmentation of available memory. Tests should be done for both graceful and abrupt teardowns of the telnet session.
- Stress the two Ethernet interfaces of a BTS by sending Internet traffic for 24 hours and verify that no memory leak or crash occurs.
- Stress the four T1/E1 interfaces of a BTS by sending Internet traffic for 24 hours and verify that no memory leak or crash occurs.
- Verify that repeated establishment and teardown of AT connections through a BSC executed over 24 hours do not result in (i) leaks in the number of buffers or amount of memory and (ii) significant increase in the degree of fragmentation of available memory. The sessions remain established for the duration of the test.
- Verify that repeated soft and softer handoffs executed over 24 hours do not result in leaks in the number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory.
- Verify that repeated execution of all CLI commands over 24 hours do not result in leaks in the number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory.

Examples of stress tests of an SNMP agent are as follows:

- Verify that repeated walking of the MIBs via an SNMP executed over 24 hours do not result in leaks in number of buffers or amount of memory and do not significantly increase the degree of fragmentation of available memory

- Verify that an SNMP agent can successfully respond to a GetBulk request that generates a large PDU, preferably of the maximum size, which is 8 kbytes under the following CPU utilization: 0, 50, and 90%.
- Verify that an SNMP agent can simultaneously handle multiple GetNext and GetBulk requests over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.
- Verify that an SNMP agent can handle multiple Get requests containing a large number of varbinds over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.
- Verify that an SNMP agent can handle multiple Set requests containing a large number of varbinds over a 24-hour testing period under the following CPU utilizations: 0, 50, and 90%.

8.9 LOAD AND STABILITY TESTS

Load and stability tests are designed to ensure that the system remains stable for a long period of time under full load. A system might function flawlessly when tested by a few careful testers who exercise it in the intended manner. However, when a large number of users are introduced with incompatible systems and applications that run for months without restarting, a number of problems are likely to occur: (i) the system slows down, (ii) the system encounters functionality problems, (iii) the system silently fails over, and (iv) the system crashes altogether. Load and stability testing typically involves exercising the system with virtual users and measuring the performance to verify whether the system can support the anticipated load. This kind of testing helps one to understand the ways the system will fare in real-life situations. With such an understanding, one can anticipate and even prevent load-related problems. Often, operational profiles are used to guide load and stability testing [10]. The idea is to test the system the way it will be actually used in the field. The concept of *operation profile* is discussed in Chapter 15 on software reliability.

Examples of load and stability test objectives for an EMS server are as follows:

- Verify the EMS server performance during quick polling of the maximum number of nodes, say, 300. Document how long it takes to quick poll the 300 nodes. Monitor the CPU utilization during quick polling and verify that the results are within the acceptable range. The reader is reminded that quick polling is used to check whether or not a node is reachable by doing a ping on the node using the SNMP Get operation.
- Verify the EMS performance during full polling of the maximum number of nodes, say, 300. Document how long it takes to full poll the 300 nodes. Monitor the CPU utilization during full polling and verify that the results are within the acceptable range. Full polling is used to check the status and any configuration changes of the nodes that are managed by the server.

- Verify the EMS server behavior during an SNMP trap storm. Generate four traps per second from each of the 300 nodes. Monitor the CPU utilization during trap handling and verify that the results are within an acceptable range.
- Verify the EMS server's ability to perform software downloads to the maximum number of nodes, say, 300. Monitor CPU utilization during software download and verify that the results are within an acceptable range.
- Verify the EMS server's performance during log file transfers of the maximum number of nodes. Monitor the CPU utilization during log transfer and verify that the results are within an acceptable range.

In load and stability testing, the objective is to ensure that the system can operate on a large scale for several months, whereas, in stress testing, the objective is to break the system by overloading it to observe the locations and causes of failures.

8.10 RELIABILITY TESTS

Reliability tests are designed to measure the ability of the system to remain operational for long periods of time. The reliability of a system is typically expressed in terms of mean time to failure (MTTF). As we test the software and move through the system testing phase, we observe failures and try to remove the defects and continue testing. As this progresses, we record the time durations between successive failures. Let these successive time intervals be denoted by $t_1, t_2, \ldots, t_i$. The average of all the i time intervals is called the MTTF. After a failure is observed, the developers analyze and fix the defects, which consumes some time—let us call this interval the *repair* time. The average of all the repair times is known as the mean time to repair (MTTR). Now we can calculate a value called mean time between failures (MTBF) as $\text{MTBF} = \text{MTTF} + \text{MTTR}$. The random testing technique discussed in Chapter 9 is used for reliability measurement. Software reliability modeling and testing are discussed in Chapter 15 in detail.

8.11 REGRESSION TESTS

In this category, new tests are not designed. Instead, test cases are selected from the existing pool and executed to ensure that nothing is broken in the new version of the software. The main idea in regression testing is to verify that no defect has been introduced into the unchanged portion of a system due to changes made elsewhere in the system. During system testing, many defects are revealed and the code is modified to fix those defects. As a result of modifying the code, one of four different scenarios can occur for each fix [11]:

- The reported defect is fixed.
- The reported defect could not be fixed in spite of making an effort.

- The reported defect has been fixed, but something that used to work before has been failing.
- The reported defect could not be fixed in spite of an effort, and something that used to work before has been failing.

Given the above four possibilities, it appears straightforward to reexecute every test case from version $n - 1$ to version n before testing anything new. Such a full test of a system may be prohibitively expensive. Moreover, new software versions often feature many new functionalities in addition to the defect fixes. Therefore, regression tests would take time away from testing new code. Regression testing is an expensive task; a subset of the test cases is carefully selected from the existing test suite to (i) maximize the likelihood of uncovering new defects and (ii) reduce the cost of testing. Methods for test selection for regression testing are discussed in Chapter 13.

8.12 DOCUMENTATION TESTS

Documentation testing means verifying the technical accuracy and readability of the user manuals, including the tutorials and the on-line help. Documentation testing is performed at three levels as explained in the following:

Read Test: In this test a documentation is reviewed for clarity, organization, flow, and accuracy without executing the documented instructions on the system.

Hands-On Test: The on-line help is exercised and the error messages verified to evaluate their accuracy and usefulness.

Functional Test: The instructions embodied in the documentation are followed to verify that the system works as it has been documented.

The following concrete tests are recommended for documentation testing:

- Read all the documentations to verify (i) correct use of grammar, (ii) consistent use of the terminology, and (iii) appropriate use of graphics where possible.
- Verify that the glossary accompanying the documentation uses a standard, commonly accepted terminology and that the glossary correctly defines the terms.
- Verify that there exists an index for each of the documents and the index block is reasonably rich and complete. Verify that the index section points to the correct pages.
- Verify that there is no internal inconsistency within the documentation.
- Verify that the on-line and printed versions of the documentation are same.
- Verify the installation procedure by executing the steps described in the manual in a real environment.

- Verify the troubleshooting guide by inserting error and then using the guide to troubleshoot the error.
- Verify the software release notes to ensure that these accurately describe (i) the changes in features and functionalities between the current release and the previous ones and (ii) the set of known defects and their impact on the customer.
- Verify the on-line help for its (i) usability, (ii) integrity, (iii) usefulness of the hyperlinks and cross-references to related topics, (iv) effectiveness of table look-up, and (v) accuracy and usefulness of indices.
- Verify the configuration section of the user guide by configuring the system as described in the documentation.
- Finally, use the document while executing the system test cases. Walk through the planned or existing user work activities and procedures using the documentation to ensure that the documentation is consistent with the user work.

8.13 REGULATORY TESTS

In this category, the final system is shipped to the regulatory bodies in those countries where the product is expected to be marketed. The idea is to obtain compliance marks on the product from those bodies. The regulatory approval bodies of various countries have been shown in Table 8.2. Most of these regulatory bodies issue safety and EMC (electromagnetic compatibility)/EMI (electromagnetic interference) compliance certificates (emission and immunity). The regulatory agencies are interested in identifying flaws in software that have potential safety consequences. The safety requirements are primarily based on their own published standards. For example, the CSA (Canadian Standards Association) mark is one of the most recognized, accepted, and trusted symbols in the world. The CSA mark on a product means that the CSA has tested a representative sample of the product and determined that the product meets the CSA's requirements. Safety-conscious and concerned consumers look for the CSA mark on products they buy. Similarly, the CE (Conformité Européenne) mark on a product indicates conformity to the European Union directive with respect to safety, health, environment, and consumer protection. In order for a product to be sold in the United States, the product needs to pass certain regulatory requirements of the Federal Communications Commission (FCC).

Software safety is defined in terms of *hazards*. A hazard is a state of a system or a physical situation which when combined with certain environmental conditions could lead to an accident or mishap. An *accident* or *mishap* is an unintended event or series of events that results in death, injury, illness, damage or loss of property, or harm to the environment [12]. A hazard is a logical precondition to an accident. Whenever a hazard is present, the consequence can be an accident. The existence of a hazard state does not mean that an accident will happen eventually. The concept of safety is concerned with preventing hazards.

TABLE 8.2 Regulatory Approval Bodies of Different Countries

Country	Regulatory Certification Approval Body
Argentina	IRAM is a nonprofit private association and is the national certification body of Argentina for numerous product categories. The IRAM safety mark is rated based on compliance with the safety requirements of a national IRAM standard.
Australia and New Zealand	The Australian Communications Authority (ACA) and the Radio Spectrum Management Group (RSM) of New Zealand have agreed upon a harmonized scheme in producing the C-tick mark that regulates product EMC compliance.
Canada	Canadian Standards Association
Czech Republic	The Czech Republic is the first European country to adopt conformity assessment regulations based on the European Union CE mark without additional approval certification or testing.
European Union	Conformité Européenne
Japan	The VCCI mark (Voluntary Control Council for Interference by Information Technology Equipment) is administered by VCCI for information technology equipment (ITE) sold in Japan.
Korea	All products sold in Korea are required to be compliant and subject to the MIC (Ministry of Information and Communication) mark certification. EMC and safety testing are both requirements.
Mexico	The products must be tested in Mexico for the mandatory NOM (Normality of Mexico) mark.
Peoples Republic of China	The CCC (China Compulsory Certification) mark is required for a wide range of products sold in the Peoples Republic of China.
Poland	The Polish Safety B-mark (B for *bezpieczny*, which means “safe”) must be shown on all hazardous domestic and imported products. Poland does not accept the CE mark of the European Union.
Russia	GOST-R certification. This certification system is administered by the Russian State Committee on Standardization, Metrology, and Certification (Gosstandart). Gosstandart oversees and develops industry mandatory and voluntary certification programs.
Singapore	The PSB mark is issued by the Singapore Productivity and Standards Board. The Safety Authority (PSB) is the statutory body appointed by the Ministry of Trade and Industry to administer the regulations.
South Africa	The safety scheme for electrical goods is operated by the South African Bureau of Standards (SABS) on behalf of the government. Compliance can be provided by the SABS based on the submission of the test report from any recognized laboratory.
Taiwan	Most products sold in Taiwan must be approved in accordance with the regulations as set forth by the BSMI (Bureau of Standards, Metrology and Inspection).
United States	Federal Communications Commission

A software in isolation cannot do physical damage. However, a software in the context of a system and an embedding environment could be vulnerable. For example, a software module in a database application is not hazardous by itself, but when it is embedded in a missile navigation system, it could be hazardous. If a missile takes a U-turn because of a software error in the navigation system and destroys the submarine that launched it, then it is not a safe software. Therefore, the manufacturers and the regulatory agencies strive to ensure that the software is safe the first time it is released.

The organizations developing safety-critical software systems should have a safety assurance (SA) program to eliminate hazards or reduce their associated risk to an acceptable level [13]. Two basic tasks are performed by an SA engineering team as follows:

- Provide methods for identifying, tracking, evaluating, and eliminating hazards associated with a system.
- Ensure that safety is embedded into the design and implementation in a timely and cost-effective manner such that the risk created by the user/operator error is minimized. As a consequence, the potential damage in the event of a mishap is minimized.

8.14 SUMMARY

In this chapter we presented a taxonomy of system tests with examples from various domains. We explained the following categories of system tests:

- *Basic tests* provide an evidence that the system can be installed, configured, and brought to an operational state. We described five types of basic tests: boot, upgrade/downgrade, light emitting diode, diagnostic, and command line interface tests.
- *Functionality tests* provide comprehensive testing over the full range of the requirements within the capabilities of the system. In this category, we described eight types of tests: communication systems, module, logging and tracing, element management systems, management information base, graphical user interface, security, and feature tests.
- *Robustness tests* determine the system recovery process from various error conditions or failure situations. In this category, we described five types of robustness tests: boundary value, power cycling, on-line insertion and removal, high availability, degraded node tests.
- *Interoperability tests* determine if the system can interoperate with other third-party products.
- *Performance tests* measure the performance characteristics of the system, for example, throughput and response time, under various conditions.
- *Scalability tests* determine the scaling limits of the system.

- *Stress tests* stress the system in order to determine the limitations of the system and determine the manner in which failures occur if the system fails.
- *Load and stability tests* provide evidence that, when the system is loaded with a large number of users to its maximum capacity, the system is stable for a long period of time under heavy traffic.
- *Reliability tests* measure the ability of the system to keep operating over long periods of time.
- *Regression tests* determine that the system remains stable as it cycles through the integration with other projects and through maintenance tasks.
- *Documentation tests* ensure that the system's user guides are accurate and usable.
- *Regulatory tests* ensure that the system meets the requirements of government regulatory bodies. Most of these regulatory bodies issue safety, emissions, and immunity compliance certificates.

LITERATURE REVIEW

A good discussion of software safety concepts, such as *mishap, hazard, hazard analysis, fault tree analysis, event tree analysis, failure modes and effects analysis, and firewall*, can be found in Chapter 5 of the book by Freidman and Voas [13]. The book presents useful examples and opinions concerning the relationship of these concepts to software reliability.

For those readers who are actively involved in usability testing or are interested in a more detailed treatment of the topic, Jeffrey Rubin's book (*Handbook of Usability Testing—How to Plan, Design, and Conduct Effective Tests*, Wiley, New York, 1994) provides an excellent guide. Rubin describes four types of usability tests in great detail: (i) exploratory, (ii) assessment, (iii) validation, and (iv) comparison. The book lists several excellent references on the subject in its bibliography section. Memon et al. have conducted innovative research on test adequacy criteria and automated test data generation algorithms that are specifically tailored for programs with graphical user interfaces. The interested readers are recommended to study the following articles:

A. M. Memon, "GUI Testing: Pitfalls and Process," *IEEE Computer*, Vol. 35, No. 8, 2002, pp. 90–91.

A. M. Memon, M. E. Pollock, and M. L. Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Transactions on Software Engineering*, Vol. 27, No. 2, 2001, pp. 144–155.

A. M. Memon, M. L. Soffa, and M. E. Pollock, "Coverage Criteria for GUI Testing," in *Proceedings of the 9th ACM SIGSOFT International Symposium on Foundation of Software Engineering*, ACM Press, New York 2001, pp. 256–267.

In the above-mentioned work, a GUI is represented as a series of operators that have preconditions and postconditions related to the state of the GUI. This representation classifies the GUI events into four categories: menu-open events, unrestricted-focused events, restricted-focus events, and system interaction events. Menu-open events are normally associated with the usage of the pull-down menus in a GUI. The unrestricted-focus events simply expand the interaction options available to a GUI user, whereas the restricted-focus events require the attention of the user before additional interactions can occur. Finally, system interaction events require the GUI to interact with the actual application.

An excellent collection of essays on the usability and security aspects of a system appears in the book edited by L. F. Cranor and S. Garfinkel (*Security and Usability*, O'Reilly, Sebastopol, CA, 2005). This book contains 34 groundbreaking articles that discuss case studies of usable secure system design. This book is useful for researchers, students, and practitioners in the fields of security and usability.

Mathematically rigorous treatments of performance analysis and concepts related to software performance engineering may be found in the following books:

R. Jain, *The Art of Computer Systems Performance Analysis*, John, New York, 1991.

C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, Reading, MA, 1990.

Each of these books provides a necessary theoretical foundation for our understanding of performance engineering.

REFERENCES

1. D. Rayner. OSI Conformance Testing. *Computer Networks and ISDN Systems*, Vol. **14**, 1987, pp. 79–98.
2. D. K. Udupa. *TMN: Telecommunications Management Network*. McGraw-Hill, New York, 1999.
3. J. Jarzombek and K. M. Goertzel. Security in the Software Cycle. *Crosstalk, Journal of Defense Software Engineering*, September 2006, pp. 4–9.
4. S. Northcutt, L. Zeltser, S. Winters, K. Kent, and R. W. Ritchey. *Inside Network Perimeter Security*, 2nd ed. Sams Publishing, Indianapolis, IN, 2005.
5. K. Sankar, S. Sundaralingam, A. Balinsky, and D. Miller. *Cisco Wireless LAN Security*. Cisco Press, Indianapolis, IN, 2004.
6. IOS for 1 × EV, IS-878, 3Gpp2, http://www.3gpp2.org, June 2001.
7. CDMA2000 IOS Standard, IS-2001, 3Gpp2, http://www.3gpp2.org, Nov. 2001.
8. CDMA2000 High Rate Packet Data Air Interface, IS-856–1, 3Gpp2, http://www.3gpp2.org, Dec. 2001.
9. B. Beizer. *Software Testing and Quality Assurance*. Von Nostrand Reinhold, New York, 1984.
10. A. Avritzer and E. J. Weyuker. The Automatic Generation of Load Test Suites and Assessment of the Resulting Software. *IEEE Transactions on Software Engineering*, September 1995, pp. 705–716.
11. J. A. Whittaker. What Is Software Testing? And Why Is It So Hard? *IEEE Software*, January/February 2000, pp. 70–79.
12. N. G. Leveson. Software Safety: Why, What, and How. *ACM Computing Surveys*, June 1986, pp. 125–163.
13. M. A. Friedman and J. M. Voas. *Software Assessment: Reliability, Safety, Testability*. Wiley, New York, 1995.

Exercises

1. What is an element management system (EMS)? How is it different from a network management station (NMS)?
2. What are the differences between configuration, compatibility, and interoperability testing?
3. What are the differences between performance, stress, and scalability testing? What are the differences between load testing and stress testing?
4. What is the difference between performance and speed?
5. Buffer overflow is the most commonly found vulnerability in network-aware code that can be exploited to compromise a system. Explain the reason.
6. What are zero-day attacks? Discuss its significance with respect to security testing.
7. Discuss the importance of regression testing when developing a new software release. What test cases from the test suite would be more useful in performing a regression test?
8. What are the differences between safety and reliability? What are the differences between safety testing and security testing?
9. What is the similarity between software safety and fault tolerance?
10. For each of the following situations, explain whether it is a hazard or a mishap:
    (a) Water in a swimming pool becomes electrified.
    (b) A room fills with carbon dioxide.
    (c) A car stops abruptly.
    (d) A long-distance telephone company suffers an outage.
    (e) A nuclear weapon is destroyed in an unplanned manner.
11. What are the similarities and differences between quality assurance (QA) and safety assurance (SA)?
12. For your current test project, develop a taxonomy of system tests that you plan to execute against the implementation.