Chapter 6
DYNAMIC SAFETY VERIFICATION

After static safety verification and general reliability testing, software engineers must dynamically verify the software’s safety (i.e., safety testing). As previous chapters mentioned, dynamic safety verification tests the software’s safety features by executing the software. Even if the developer uses rigorous formal techniques to show the software’s safety, safety faults may still exist in the system since these techniques do not produce 100% reliable software. For these reasons, safety testing is important and attempts to show that the software conforms to the safety documentation (e.g., fault trees, event trees, failure modes and effects, etc.), which represent the input-output oracles for the safety-testing process. If the safety documentation does not match the software, then the systems safety assessment and risk assessment are wrong.

Just like general reliability testing cannot prove that the software is totally correct, safety testing cannot prove that the software is totally safe since neither approach can, with any assurance, detect faults when the software matches the oracle. However, safety testing can help detect differences between the safety-testing oracle and the software such as incorrect events, imprecise events, incorrect relationships between events, missing code, and incorrect code. An incorrect event is any event such that when the event occurs, the resulting consequence does not occur. An imprecise event is any event such that its description is not clear enough to facilitate test-case generation. An incorrect relationship between events is any event relationship that does not correctly define the association between the event’s causes.

Despite dynamic safety verification’s importance, current research largely ignores techniques for safety testing. Instead, most researchers concentrate on static safety techniques such as VDM, Z, FMEA, FTA, and ETA - refer to Chapter 2 for more information on these techniques. While static techniques are important, dynamic techniques are also important; therefore, this chapter discusses the dynamic-verification process and presents several potential techniques for safety testing.
6.1. SAFETY-RELATED DYNAMIC-VERIFICATION ACTIVITIES

From a safety viewpoint, Figure 6.1 outlines the dynamic-verification activities while showing necessary inputs and outputs for each activity. The basic activities for any testing process are to (1) develop test cases based on some testing oracle, (2) exercise the software over the test cases, and (3) determine whether or not the software produced the correct results according to some testing oracle. Additionally, testers can assess coverage based on some coverage technique. For example, general reliability testing has statement coverage, branch coverage, and multiple-condition coverage to name a few.

6.1.1.DEVELOPING, TEST CASE

Before executing the software in order to help verify that it’s safe, the testing team must first develop test cases that concentrate on uncovering safety-related errors in the software and discrepancies between the software and the safety documentation (e.g., fault trees). As with most testing activities, the testing team follows some oracle to determine test cases. An oracle is an external source of information about the software and describes how the software should function. For this research, the oracles are the safety-related documents, which come about from static verification. If the developmental process followed a life-cycle approach to safety verification, then this information would be present in the safety documentation.

The safety documentation may contain fault trees; event trees; and safety-critical specifications, high-level designs, and detailed designs. The inputs to this activity are the software and its safety documentation. The outputs are test-case sets, which include both input and output information.
**Input data.** In order to determine the inputs that concentrate on uncovering safety-related errors, the tester can reference the safety documentation and software: (1) the list of failure modes and their effects, (2) event trees, (3) fault trees, (4) software-hazard lists, and (5) the fault-tolerant code sections. If this information does not exist, then the tester should develop it according to the guidelines in chapters three through five. The number of test cases to develop depends on the thoroughness that the tester desires, which in turn depends on such factors as system requirements, time, cost, and the software’s safety level. For example, software components that can cause catastrophic consequences (e.g., safety-critical operating systems) will require more thorough testing than other software components.

**Output data.** As with the inputs, the tester can use the safety documentation to determine the proper results (i.e., the outputs). Certain events will result in specific
consequences, and the tester should be able to determine these consequences by examining the safety documentation (e.g., event-tree branches). In addition to determining an event’s consequences, the tester must also determine the other outputs, if any, as well. These outputs might be changes in fuel flow, weight calculations, altitude readings on a display, or some other information.

6.1.2. EXERCISING THE SOFTWARE

Once the software and test cases are ready, the testing team can proceed to exercise the software utilizing the test cases. In order to test the software, developers can use debuggers, real-time emulators, simulators, special testing tools, or a combination of these methods. For all critical systems, final safety testing must take place using the actual hardware as much as possible so as to help insure proper hardware-software interfaces. To obtain adequate coverage for software with high safety levels, the testers must keep track of appropriate events (see Safety Coverage Techniques later in this chapter) that they tested; unfortunately, no tools exist for automatically doing this process.

6.1.3. DETERMINING CORRECTNESS

The last safety-testing activity involves determining each test case’s results and whether or not the results were correct (i.e., comparing the expected results against the actual results). Once again, debuggers, real-time emulators, simulators, and other techniques are useful during this activity. In addition to these techniques, having fault-tolerant code sections report activation also aids this activity since the tester needs to know when and if the software detects any faults. With respect to software safety, fault-tolerant code sections exist to detect and recover from faults that could lead to hazards. The verification team can use these code sections not only to detect safety-related failures but also to report the failures as they occur. For some systems, reporting failures may be a requirement (e.g., error logging). Other systems, however, will need to add these features. Furthermore, fault tolerance may uncover new faults in either the software or hardware depending on the specific fault-tolerant technique. For example, N-version programming might uncover errors in the specifications, software, developmental tools, and hardware depending on its implementation.
A key area within software safety is the software’s ability to detect and recover from various faults. However, there are certain faults that software cannot detect such as hard failures in the processor, hard failures in ROM, and hard failures in RAM. For these reasons, developers often create backup systems with separate hardware and software that can take over in the event a hard failure occurs in the primary system.

<table>
<thead>
<tr>
<th>Fault-Tolerant Technique</th>
<th>Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Version Programming</td>
<td>Change the voter to report all non-unanimous votes between the various versions.</td>
</tr>
<tr>
<td>Recovery Blocks</td>
<td>Have the acceptance test report any failures and always call each variant during verification.</td>
</tr>
<tr>
<td>Abbott-Neumann Components</td>
<td>Have all self-protecting and self-checking code sections report any failures or violations.</td>
</tr>
<tr>
<td>Resourcefulness</td>
<td>Change all goal-failure detection routines to report goal failures.</td>
</tr>
<tr>
<td>Other Techniques</td>
<td>Report any failures that occur such as failures for the various self-checks.</td>
</tr>
</tbody>
</table>

Table 6.1 - Modifications for Specific Fault-Tolerant Techniques

However, this research concentrates on those faults that software can detect such as soft failures in ROM, RAM, and other hardware devices; errors in human input; and errors in the software itself that affect safety. The current fault-tolerant techniques address these issues, and this section discusses how to modify specific fault-tolerant techniques to report safety-critical failures when they occur. This process and the testing oracles allow testers to verify that specific fault-tolerant approaches work correctly. Table 6.1 outlines the various fault-tolerant techniques and the modifications necessary to report failure anomalies when they occur.

The inputs for this activity are the software itself and the safety documentation, which identifies safety-critical code sections that provide fault tolerance. The output is the
modified code, which is ready for testing. As Chapter 4 mentioned, implementing the fault reporting is possible by (1) inserting conditional software, (2) developing and using a special procedure, or (3) both these methods. Conditional software provides a means for easily activating and deactivating fault-reporting code sections by changing a flag and recompiling the software - most programming languages provide features for conditional software. When developing the code for fault reporting, the verification team should keep the code very simple, insure that the code is correct, and insure that the code does not affect the system’s safety, timing, or performance. By keeping the code very simple, the developer can improve the process for showing it to be both correct and safe. Also, safety-coverage techniques (later in this chapter) can help show that these modifications do not affect safety.

6.1.3.1. N-VERSION PROGRAMMING.

For N-version programming, the verification team should modify the voter to report any nonunanimous outputs. For some systems, the voter is a hardware device that collects the outputs from different variants running in parallel.

![Diagram of N-version programming showing non-unanimous vote reports.]

If the hardware device is programmable such as a microcontroller, then the tester can modify its software to report nonunanimous outputs. Also, this modification works for software voters that do not run in parallel. The developer should design other hardware voters so that software can read this information from the hardware when necessary. Figure 6.2 shows a modified N-version-programming system that reports non unanimous votes.
6.1.3.2. Recovery blocks.

A recovery-block approach has multiple variants and usually one acceptance test (even though some recovery-block schemes use multiple acceptance tests). Testers should modify the recovery blocks so that they report any acceptance-test failures. Furthermore, the testers should modify the recovery blocks so that they execute and check all variants.

![Diagram of Recovery Blocks]

**Fig 6.3. Recovery-blocks Showing acceptance-test failures**

Figure 6.3 shows a modified recovery-block approach that reports all acceptance-test failures. Remember that recovery blocks normally execute other variants only when necessary (i.e., when an acceptance test detects a failure). By forcing the technique to execute all variants during testing, the testing process may detect errors in variants or the acceptance test. However, some recovery-block approaches may not be able to execute all variants all the time.
6.1.3.3. Abbott-Neumann components.
This technique concentrates on making software components both self-protecting and self-checking (i.e., the component detects any errors it creates as well as errors that subordinate components may cause). The testers should modify the self-checking code sections so that they report any failures during execution. The failures may occur in the component itself or in another component’s return values such as an operating-system request.

6.1.3.4. Resourcefulness.
A resourceful software component is one that is functionally rich (i.e., the component can achieve its goals in multiple ways). During dynamic verification, the testers should modify resourceful components so that they report any goal failures. Furthermore, the testers should insure that the system executes each goal variant. Figure 6.4 shows a modified resourceful component that reports goal failures. The basic diagram is similar to recovery blocks; however, each variant in a recovery-block approach produces the same output whereas each variant in a resourceful approach usually produces a different output since each variant tries to achieve a goal.

6.1.3.5. Other techniques.
Depending on the software system, other fault-tolerant techniques may be present in the software. These techniques check for specific failures such as timing failures or communication failures and implement appropriate recovery procedures. In general, the tester should modify all self-checks to report failures as they occur. Also, the tester should insure that the system exercises all recovery procedures in order to insure that they work correctly.

6.2. SAFETY COVERAGE TECHNIQUES
Software can violate safety requirements by two basic means: (1) functionality in the software that is present but wrong and (2) functionality that is missing from the software. Normal testing techniques, such as branch coverage, may or may not detect safety-critical errors since they may not or cannot exercise safety-critical paths. At the present time, there is no research outlining safety-related testing techniques.
Fig 6.4. Resourcefulness Diagram showing Goal Failures

This section discusses some potential testing techniques and coverages, which help insures that there are no discrepancies between the safety documentation and the software.

<table>
<thead>
<tr>
<th>Testing Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety-Device Testing</td>
<td>Test all interlocks, lockins, and lockouts to insure proper functionality.</td>
</tr>
<tr>
<td>Fault-Tree Testing</td>
<td>Exercise code sections forming various combinations of terminal events.</td>
</tr>
<tr>
<td>Event-Tree Testing</td>
<td>Exercise combinations of code sections corresponding to paths in the event tree.</td>
</tr>
<tr>
<td>Fault-Detection-and-Recovery Testing</td>
<td>Test all fault-detection routines and insure their recovery procedures work.</td>
</tr>
</tbody>
</table>
Table 6.2 - Summary of Safety Coverage Techniques

<table>
<thead>
<tr>
<th>Safety-Device Testing</th>
<th>Failure-Modes-and-Effects Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Force the execution of each failure mode and insure its consequences are correct.</td>
</tr>
</tbody>
</table>

In order to determine test cases for safety testing, the testing team can use various safety coverages just as general reliability testing uses coverage techniques. Each safety coverage technique forces the tester to develop specific test cases in order to satisfy the technique. The safety coverage techniques appropriate for a given project depend on the safety documentation available and the fault-tolerant techniques used. In general, however, testers can use the following techniques: safety-device testing, fault-tree testing, fault-detection-and-recovery testing, event-tree testing, and failure-modes-and-effects testing. Table 6.2 outlines these techniques and provides descriptions for each one. The following sections cover these techniques in more detail.

6.2.1. Safety-Device Testing

Some safety-critical systems have devices, whether hardware or software, that protect the system in some way: Three common safety devices are interlocks, lockins, and lockouts. An interlock device insures event sequences occur in the proper order. For example, an interlock for a simple traffic light would insure that the lights always follow a green-yellow-red sequence. A lockin device maintains a certain state until a specific condition occurs. For example, a lockin device for a traffic light might maintain a flashing-red signal whenever a sequence failure occurs; the light would stay in this state until it is ready to attempt normal operation again. A lockout device prevents a certain event from occurring. For example, a memory-management unit (MMU) can lockout specific software routines from accessing certain memory locations. Also, security levels are a type of lockout device.

Normal testing techniques do not guarantee adequate safety-device coverage since such testing usually involves paths that statement, branch, and multiple-condition coverage do not touch. For example, testing software to insure it cannot break out of a lockin device. For testing interlocks, the testers should try various sequence violations and insure the interlock detects the failures: state-machine testing may
apply to this area. For lockins, testers should insure that the lockin device does not exit prematurely due to other internal or external failures. For lockout devices, the testing team can create events that attempt to violate the lockout. For example, to insure soft isolation between tasks in an embedded system, the testers can violate stack limits, write to protected memory, or attempt to use privileged instructions.

6.2.2 Fault-Tree Testing
Probably the most common static-analysis technique that developers use for analyzing safety-critical software is fault-tree analysis. Assumptions about the fault tree are often the basis for determining the systems risk and safety levels. However, the fault tree itself may have errors or ambiguities. This research presents several testing coverages to improve the developer’s confidence in the fault tree’s accuracy.

Fig.6.5. Sample Fault Tree showing terminal and Non-Terminal Events

Every fault tree or fault-tree subset contains a set of terminal and non-terminal events. The fault tree’s leaves represent the terminal events while the inner nodes represent
the non-terminal events. The fault tree associates events through logical operators such as "and" and "or" gates. By the fault tree’s nature, testers can produce a logical equation for any non-terminal event. For example, Figure 6.5 shows a sample fault tree where the event $P_4 = E_{41} (E_5 \& E_6 \& E_7) \lor E_8$.

In order to be able to test the accuracy of this fault-tree subset (i.e., event $P_4$), the terminal events must be very specific, which is a criteria for such nodes. Given the event $P_4$ and specific criteria for its terminal events, testers can cover the event at various levels of detail.

The simplest coverage is event coverage, which exercises the software components for each event. Event coverage is analogous to statement coverage in general reliability testing. In fact, statement coverage satisfies event coverage from a purely software standpoint. The result should show that the event tree is correct for the given test case. For example, if only the event $E_6$ is true, then the parent event $P_4$ should not occur (i.e., its false).

The next coverage type is event-condition coverage, which requires test cases that force the parent event to take on its two possible values: true or false. Event-condition coverage is analogous to branch coverage in reliability testing. Event-condition coverage does not imply event coverage since the tester does not necessarily have to exercise all terminal events in order to cause the parent event to be true and false. The previous example already showed how to make $P_4$ false, so to satisfy event-condition coverage, a tester can force $P_4$ to become true by forcing either $E_4$, $P_5$, or $E_8$ to be true.

The most thorough coverage type is multiple event-condition coverage, which requires test cases that try every terminal-event combination for a particular parent event. In the previous example, event-condition coverage tried only two of the 32 possible combinations $P_4$ could be true or false. Multiple event-condition coverage implies event coverage and event-condition coverage. Also multiple event-condition coverage is analogous to multiple condition coverage in reliability testing.

Another way to view fault-tree coverage is by looking at levels of failures. A level-0 coverage would force no failures in the fault tree. A level-1 coverage would
force each terminal event to fail one at a time. For example, a level-1 coverage on P4 would require five test cases: one for each terminal event E4 through E8. A level-2 coverage would force failures for all possible combinations of two terminal events. For example, P4 would require ten test cases (i.e., $10 = 5 \times (5 - 1)/2$), and P1 would require 36 test cases for level-2 coverage. Similarly, level-n coverage would force failures for all combinations of n terminal events. Once again, the goal is to show that actual results from executing the system match the expected results in the fault tree. If the fault tree and software do not agree, then the system’s entire safety analysis is faulty.

The last coverage, safe-event coverage, would involve trying all terminal-event combinations that should result in the parent event being in a safe state. For P4, this coverage would require seven test cases (i.e., all the combinations of E5 through E7 except the one combination where they all occur). This coverage, of course, is not as thorough as multiple event-condition coverage.

This section presented five different coverages based on fault-tree testing. Some coverages are more thorough than others; the most thorough coverage is multiple event-condition coverage, and it implies all the other coverages. However, for some systems, multiple event-condition coverage may not be practical since the number of test cases may be too large. At a minimum, the testing team should ensure event coverage and safe-event coverage in order to give reasonable confidence that the software and hardware satisfy the fault tree. The test-coverage thoroughness should also change depending on the parent event’s severity. For example, catastrophic events should require multiple event-condition coverage while minor events may need only safe-event coverage. Further research is necessary here in order to determine coverage for different severities.

6.2.3. Event-Tree Testing

Event trees represent a forward analysis through the safety-critical system by diagramming a given event and various reactions by the system to the event. From a safety standpoint, testers should ensure that the event tree is accurate by generating test cases to cover all paths in the tree that lead to safe outcomes. Such a coverage helps insure that specified safety measures work according to the event tree’s
specification. The prerequisite for using this technique is a complete event-tree analysis since this testing technique uses the information from ETA. Lastly, in order to increase confidence that the entire event tree is accurate, testers should exercise all paths in the event tree (i.e., even the paths that lead to unsafe states). The total number of test cases to cover completely an event tree are equal to the number of leaves in the tree. For example, the tree in Figure 6.6 would require six test cases.

6.2.4 Fault-Detection-and-Recovery Testing

This testing technique involves exercising all fault-detection and fault-recovery paths, which the safety documentation specifies for the software. In general, this technique covers different paths than statement and branch coverage since fault detection normally requires multiple-condition tests in the software and statement and branch coverage do not adequately cover multiple-condition tests.

![Event Tree for a Hypothetical Home Microwave System](image-url)
Furthermore, fault-detection-and-recovery testing covers different paths than multiple-condition coverage since some fault paths have nothing to do with software conditions (e.g., soft RAM failures, asynchronous interrupts, stack overflows, etc.). In fact, the number of paths through the software when including such conditions is large, thus making testing all paths infeasible. For this reason, fault-tree analysis and event-tree analysis attempt to isolate critical faults and their recovery procedures; fault-tree testing and event-tree testing then can help show that the software handles these faults properly.

6.2.5 Failure-Modes-and-Effects Testing

This technique attempts to increase confidence in the information gathered during failure-modes-and-effects analysis such as failures, their causes, and their consequences. The testing team should generate test cases using failure-mode causes and execute these test cases to show that the resulting failure does occur and that the failure’s consequences also occur. For certain failure modes, testing may not be necessary or appropriate. For example, testing the causes for a wing falling off a plane. However, for software failure modes and certain hardware failure modes, testing is necessary and appropriate in order to show the FMEA’s accuracy (as it is for fault-tree analysis and testing).

This chapter presented several testing and coverage techniques that are specific to safety-critical software systems: fault-tree testing (FTT), event-tree testing (ETT), safety-device testing (SDT), fault-detection-and-recovery testing (FDRT), and failure-modes-and-effects testing (FMET). For each technique, there are appropriate oracles for determining test cases and whether or not the software system’s results are correct. The oracles come from the respective safety documents: FTA supplies the oracle for FIT, ETA supplies the oracle for ETT, FMEA supplies the oracle for FMET, and the design specifications supply the oracles for SDT and FDRT. As with most testing techniques, there are separate categories that the techniques naturally fall into. These categories group techniques together that overlap in their testing capabilities.

For the safety-testing techniques, which this chapter presented, two categories arise: (1) safety-device testing; and (2) fault-tree testing, event-tree testing, failure-modes-and-effects testing, and fault-detection-and-recovery testing. Furthermore, as
mentioned earlier, there are various coverage levels for each testing technique. An area that needs further research is selecting techniques from each category depending on the software system’s criticality level. For example, one approach would be to require one testing technique - possibly a specific technique - from each category for Catastrophic-level systems and require certain coverage percentages as well.

**Summary**

Chapter 6 discussed safety-related dynamic verification in detail. The chapter outlines the safety-testing activities and how to use fault tolerance to help the testing process. Chapter 6 also introduced several safety-testing techniques and discusses potential coverage that may be beneficial during dynamic verification.