Module 13: Reliability Engineering; Safety Engineering

Objectives:

1. Understand how specifications for critical systems differ from specifications for non-critical systems.
2. Understand how risk analysis is used to derive safety requirements for a safety-critical system.
3. Understand several metrics for measuring reliability.
4. Understand how security specifications differ from safety specifications.
5. Be familiar with several ways of engineering dependability into software.
6. Know the difference between redundancy and diversity.
7. Understand how N-version programming works.
8. Be acquainted with several guidelines for dependable programming.

Assigned Reading:

Sommerville, chapters 11 & 12.

Assignment:

No assignment for Module 13.

Quiz:

Take the quiz for Module 13. (This is the final quiz for the course.)

Lecture:

Risk-Driven Specification

Sommerville starts off Chapter 12 by recounting an incident involving a plane crash where the braking system of the aircraft was controlled by software that performed exactly according to its specification, but nevertheless the system as a whole failed because the plane ran off the runway upon landing. This is an example where verification said the system was correct, because it conformed to its specification, but validation failed to uncover the fact that the specification was incorrect (in this case, incomplete). Although the scenario that led to the failure was considered to have a rare frequency of occurrence, the fact that human lives were impacted by the failure underscores the need for even stricter, more thorough requirements elicitation and specification. When creating a specification for a critical software system, it is not only essential to cover all the functionality the software should perform, it is also important to include certain functionality the software should **not** perform. Including functionality that should not be performed is a good idea for all software systems, but it is especially important for critical systems.

In critical systems, there is a primary focus on what the potential risks are should the software fail, and what the impacts of those risks are. It is therefore common for risks to play a governing role in requirements elicitation and specification. In general, such a risk-driven approach to requirements specification has four primary stages:

1. Risk Identification

   In this stage, as many of the potential risks are identified as possible. Risk identification can involve brainstorming, mathematical analyses, analyses of past trends, etc. The risks are tabulated, and given unique identifiers and descriptions. The focus here is simply to figure out what the possible risks are, and nothing else. Once the identification process is complete (which may or may not have uncovered all the risks), the risks can be further analyzed.

2. Risk Analysis

   Here, each risk is analyzed independently, to determine what conditions are necessary in order for each risk to occur; how likely it is for those conditions to appear; and the quantitative impact each risk will have. The risks are given priorities, and ordered in decreasing level of priority. Since it is possible that not all the identified risks can be dealt with, decisions must be made regarding which risks should be factored into the system design, and which risks are left unaddressed.

3. Risk Decomposition

   In risk decomposition, attempts are made to discover the root causes of the risks from the priority list created during risk analysis.

4. Risk Reduction

   Using the information from the previous two stages, solutions for avoiding the risks are proposed. For risks where there are no acceptable avoidance solutions, steps are outlined to mitigate the impacts of the risks. For critical systems, any practical solution will be considered, including: redundant software and/or hardware, which may operate continuously or serve as backup; provisions for manual overrides and fail-safes; continued testing even after the system has been placed in operation; etc.

It is quite possible that not all the risks will be identified on the first attempt. It is also possible that additional risks may be discovered during the later phases of the process. Therefore, it may be necessary to perform multiple iterations of the process, or even halt the current stage and return to a previous stage. The need for identifying and assessing risks iteratively is what led to Barry Boehm's spiral model of development, which is shown on page 49 of the textbook. In that figure, the stage called "Risk Analysis" encompasses the four stages listed above.

Measuring Software Reliability

Reliability can be defined as the probability of a software failure occurring while the software is in use and operating within specified conditions. Below are six reliability metrics you are most likely to see used for measuring critical software reliability:

1. POFOD — Probability Of Failure On Demand
2. ROCOF — Rate of failure occurrence
3. AVAIL — Availability
4. MTTF — Mean Time To Failure
5. MTTR — Mean Time To Repair

6. MTBF — Mean Time Between Failures

The relationship between the latter three metrics is illustrated in figure 13.1. The terms, MTTF and MTBF, are not always defined consistently. The mean time to failure assumes the system has just become operational, and describes the average time it will take for the system to go from that point to failure. The mean time to repair is the average time it takes to repair the system following a failure. Depending on the type of failure, this can be a short time (e.g., the system simply needs to be rebooted) or a long time (the system must remain offline until a patch can be applied or a component must be replaced). The mean time between failures is the average time between two consecutive failures. As can be seen from figure 13.1, MTBF = MTTR + MTTF.

It should be pointed out that these metrics were originally designed to measure the reliability of hardware, not software. Failures in software tend to be of a different nature than failures in hardware. For example, if a hardware component fails the system can usually be corrected by replacing the failed component. Replacing a software component is not so simple. Software components that fail have a fault that must be corrected, which requires locating the fault, correcting it, and updating all necessary documentation. Quick-fix patches are routinely offered by many software vendors as immediate solutions, sort of like a band-aid until the problem can be fixed properly.

Although all the metrics listed above make use of the term "time" in their definitions, time doesn't necessarily have to refer to calendar time. Calendar time is most often used for systems that are in continuous operation, where the functionality is not based on transactions. For systems that do involve transactions, reliability may be expressed in terms of the frequency of failure per X number of transactions.

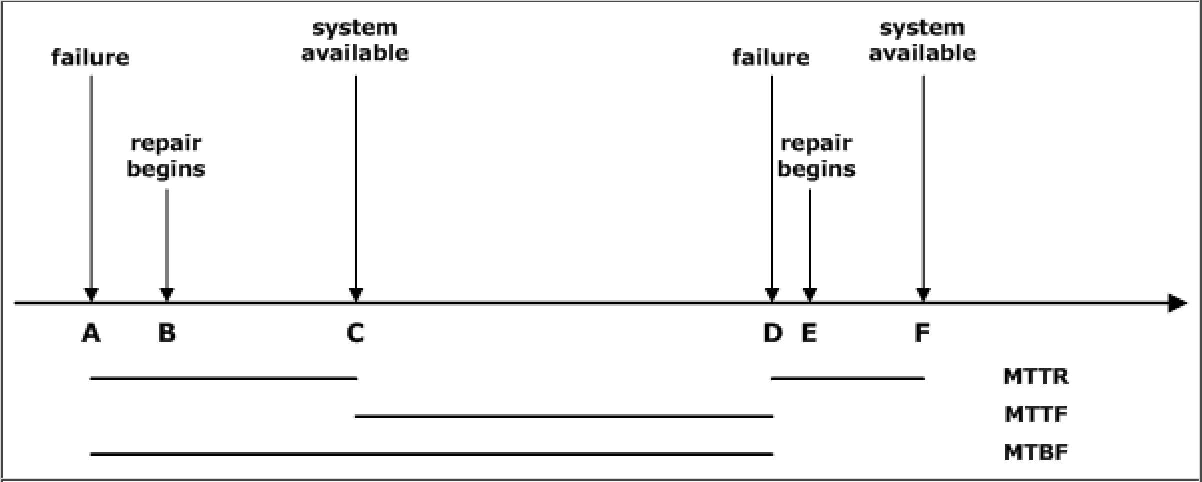

Figure 13.1. Relationship between the metrics, Mean Time To Repair (MTTR), Mean Time To Failure (MTTF), and Mean Time Between Failures (MTBF). The bars below the graph indicate the regions of the graph that would be used in calculating each of the three metrics.

Security Specifications

Sommerville makes an important distinction between system failures that are related to safety and system failures that are related to security, and it is worth emphasizing that difference here. When a failure causes physical harm to people, it is due to a malfunction, software fault, or a design flaw, all of which can result from a faulty specification. In other words, no one is actively trying to cause the failure. A security failure, on the other hand, very often occurs because someone is making an attempt to cause the system to fail. It is also possible for security failures to happen accidentally, but with security it has to be assumed that all failures are deliberate, and most likely malicious in nature. It must also be assumed that whoever initiates a security attack is intelligent, and is capable of employing multiple types of potentially coordinated attacks. The "intruder" may be a person, or it may be a cleverly designed "bot" that is programmed to rapidly launch attacks, and may have the capacity to conceal itself from detection systems.

A security failure can be manifested in one or more ways, including:

- allowing access to unauthorized individuals
- allowing authorized individuals to gain unauthorized privileges
- propagation of a virus, worm, or spam
- corruption of, deletion of, or theft of data
- website hacks (e.g., altering a website's homepage)
- denial of service (DoS) attacks

Establishing security requirements, and designing a system for security, are especially challenging, both because of the wide and varied nature of possible security threats, and because an intelligent attacker can find ways to penetrate a system's defenses even after one avenue of attack has been shut down. There is a continuous measure-countermeasure type of battle occurring as attackers find new ways to exploit old security holes, or discover new security holes.

Engineering Dependability into Software

In the previous module we discussed four primary, and four auxiliary facets of dependability:

1. Availability
2. Reliability
3. Safety
4. Security
5. Repairability
6. Maintainability
7. Survivability
8. Error Tolerance

We also talked about three primary approaches to ensuring the above characteristics in software: fault avoidance, fault detection and removal, and fault tolerance. A combination of all three approaches will go a long way to ensuring system dependability, but unfortunately the efficacy of using any of these approaches diminishes as more faults are found and removed. As the system becomes more dependable, the cost of finding and correcting further faults quickly becomes prohibitive, as shown in Figure 13.1 in the text. The reason for this is simple enough to understand. Assuming a thorough test suite was used the first time around, the faults that are relatively easy to detect will have been detected and removed. Subsequent rounds of testing must therefore focus on faults that become increasingly difficult to find, and their detection may depend on just the right combination of inputs, or intuitive insights into the project itself. Remember, in most cases exhaustive testing of all possible input combinations is impossible, so we tend to rely on approaches such as partition testing to allow us to test fewer inputs that presumably cover the range of possible input classes. This assumption, along with other assumptions, may result in catastrophe if a fault is allowed to slip through for a critical system. Thus, for critical systems, we're willing to pay for a certain amount of additional testing to try to discover hard-to-find faults, but at some point the testing will either become too costly, or it will prevent the system from ever being released. The upshot of this is that a decision must be made at some point to stop testing with the acknowledgement that there are probably some faults still present in the software. The goal, then, is to try to ensure that the faults that *could* remain will have only minor impacts should they end up causing a system failure, and that appropriate mechanisms are in place for dealing with the consequences of system failures when they occur. Thus, there is a primary focus on engineering fault tolerance into critical systems, which is responsible for much of the high cost associated with critical systems development.

Redundancy and Diversity

We mentioned in Module 12 a little bit about redundancy in fault tolerant systems. Redundancy essentially means that additional resources and/or duplicate functionality is included in the system. Redundant functionality may be held in reserve and used only when needed, or it may be used concurrently with the main system. The idea behind redundancy is that if one component fails, there are backups that can ensure normal system operation until the problem that caused the failure can be corrected.

Diversity means that different components are used. The differences between the components may lie in the characteristics of the components themselves, or in how the components are manufactured. Diversity does not necessarily imply redundancy, but it can be used as one means of providing redundancy, for example when diverse components are used together in the same system. Diversity is easiest to see with hardware. For example, two different hardware components might be installed on a system. One of the components may perform more efficiently, but it may tend to overheat under certain conditions. The other component may function less efficiently, but not be prone to overheating. The system could be configured to rely primarily on the more efficient component, but in the event that component reaches a specified threshold temperature, the system can switch to using the less efficient component until the primary one has cooled down. Diversity can also be present in software, however. For example, two or more different algorithms might be used for accomplishing the same task. The algorithms may differ in the amounts of system resources they use, they may rely on different approaches to performing the task, or both. Again, the system could be designed to monitor resource usage patterns, and if it is anticipated that the primary algorithm will not be able to function given the current level of resources, an alternate algorithm could be used.

Of course, engineering redundancy into a system, especially if system monitoring features are included, is very costly and makes the system much more complex. But for a critical system, especially one where safety is a prime concern, the extra effort and cost are justified.

N-Version Programming

Redundancy and diversity can also be applied to the software development process itself. A common form of this is known as **N-version programming**. In N-version programming, multiple teams are given the same requirements specification and each team is directed to build a system based on that specification—*independently of the other teams*. This approach takes advantage of two important facts:

1. Different people solve problems in different ways.
2. Almost any given problem can have more than one valid solution.

The independence between the teams is crucial for N-version programming to work. The teams should not be allowed to communicate with each other, or the results could be biased towards one particular solution. N-version programming works best when it is done blindly, with the teams having no knowledge of each other, and even separated geographically, if possible.

Usually, all the systems that result from an N-version programming project are run concurrently, and the results for critical functions are compared, and synchronized according to a simple majority voting process. For example, if there are three independent systems running concurrently, and for a particular function, two of the systems produce an output of 12.5, while the third system produces an output of 12.7, the value of 12.5 will be used. If this output value is to be subsequently passed as input to another system function, all three systems will be synchronized to have 12.5 used as the input value. Naturally, a system of this type also requires an additional subsystem that manages the voting process and ensures the systems remain synchronized.

Dependable Programming Guidelines

What constitutes dependable programming can vary to a certain extent depending on which programming language is used, but there are a number of good practices that can be applied almost universally. Sommerville mentions some of these in closing out Chapter 13, and I summarize them below.

1. <u>Limit the visibility of information in a program.</u>

   Visibility of information can be controlled using scope modifiers, such as the public, protected, and private scope modifiers found in Java. Developers using components from a library should only have access to the public interface. Anything a developer does not explicitly need to know should be kept below public scope, if for no other reason than to simplify the component's interface. The Principle of Information Hiding is also included here, as it relates to the coupling between components. The inner workings of a component should not be exposed. Any code that is specifically tailored to such knowledge may create incompatibilities should the inner workings be changed at some point.

2. Check all inputs for validity.

   A great number of system failures are associated with invalid input. Simple entry of data that does not conform to the expected format can cause a complete system crash. The best place to validate input data is immediately after the data are entered or received. In the case of human users, data entry typically occurs by filling in on-screen forms. The controls on these forms can be set up to force the user to enter data in certain formats; e.g., dates must be entered in the format MM-DD-YYYY. As long as inputs are validated immediately upon entry, and the system functions are consistently designed to use the same formats, the data should not need to be re-validated during processing.

3. Provide a handler for all exceptions.

   As we have mentioned, even after rigorous testing we have to be prepared to accept that some faults may still remain in a system. We also have to acknowledge that there can be unforeseen events such as hardware hiccups and power failures. An error or unexpected event that occurs while a system is executing is known as an **exception**. Ensuring dependability means ensuring that the system is able to handle exceptions when they occur. Handling an exception can range from displaying a meaningful error message to the user, and possibly prompt for re-entry of data, passing the exception on to a higher-level component, or gracefully terminating execution. Here, the term "gracefully" entails ensuring unfinished transactions are rolled back, ensuring any system resources in use by the system are released, notifying dependent systems, etc. Some programming languages such as Java have built-in constructs for catching and throwing exceptions. For those that don't, exception handling can be accomplished using appropriate conditional statements.

4. Minimize the use of error-prone constructs.

   Certain programming language constructs tend to have a greater propensity for introducing faults into software. Sommerville talks about eleven such constructs:

   a. Unconditional branch statements
   b. Floating-point numbers
   c. Pointers
   d. Dynamic memory allocation
   e. Parallelism
   f. Recursion
   g. Interrupts
   h. Inheritance
   i. Aliasing
   j. Unbounded arrays
   k. Default input processing

   Unconditional branch statements are exemplified by the infamous "GOTO" statement. These statements cause problems with code readability, especially in large programs. They also promote coupling between modules, which can be very problematic.

   Floating-point arithmetic is inherently imprecise, since a fixed number of bytes are used to store an unfixed number of digits for the decimal portion of a number. The loss of precision may only be in the last digit of the decimal, but that is enough to throw off comparisons. When a series of calculations involves floating-point arithmetic, small inaccuracies can be compounded. Two alternatives to floating-point arithmetic are to use fixed-point decimals instead of floating-point decimals; or represent floating-point numbers as strings, which have no inherent limitation in length.

Pointers allow direct access to memory addresses, which can be very useful when used properly, but very dangerous if no checks are implemented to ensure memory accesses are both valid and allowed. They are common in relatively low-level languages like C, and are notorious for their capacity to cause buffer overflows, which can be exploited in an attack on a system.

With dynamic memory allocation, memory is allocated for variables, arrays, and other structures at runtime. The advantage of dynamic memory allocation is that it can optimize memory usage by using only the amount of memory needed at a given time. Unfortunately, memory leaks result if this memory is not deallocated once it is no longer needed, and these leaks can potentially hog all available memory, which can slow down performance and eventually lead to a system failure. Memory leaks can be difficult to plan for during design, as well as detect during testing. Some languages, like C++, provide *destructor* constructs that allow developers to explicitly direct resources used by an object to be freed once the object is no longer needed. Java tries to simplify this task by using an automated "garbage collection" mechanism, which periodically scans memory to deallocate memory used by objects that are no longer referenced by any variables. However, developers have very little control over when the garbage collector performs its scans, and the scans themselves can cause performance decreases.

Concurrent programming (parallelism) is inherently error-prone. Race conditions can cause deadlocks if they are not controlled by semaphores, locks, monitors, or other such constructs. Synchronization of data can also be extremely problematic. Problems caused by concurrency are not always easy to predict, and they can be very difficult to detect during system testing.

Recursion is a very elegant way to solve certain types of problems using relatively little code. However, recursive algorithms can be difficult to read, and even more difficult to trace during execution. Additionally, if the recursion is deep enough, it can cause an overflow of the program stack.

Interrupts, as we mentioned in a previous module, are used to handle aperiodic events. They are most often used when software must take action in response to a signal from a hardware device. The problem with using interrupts is that they may cause the suspension or termination of a critical operation. Even a brief (i.e., several milliseconds) suspension of a time-critical operation can cause serious problems.

Object-oriented languages make extensive use of inheritance to abstract commonalities of two or more classes into a shared superclass, and to allow derived classes to override inherited functionality. Unfortunately, inheritance usually leads to source code being separated in multiple source files, which negatively impacts readability. When inheritance is extensive, it can be extremely difficult to follow, and it can be very easy to make incorrect assumptions about which data type an object is at a particular point in time.

Aliasing occurs when the same entity is referenced using multiple identifiers, which complicates development since it is more difficult to keep track of several identifiers than it is to keep track of only one.

Unbounded arrays, like pointers, tend to only occur in relatively low-level languages like C. In C, array indexes function like pointers, allowing direct access to memory addresses, but because C performs no implicit bounds checking on arrays, it is possible to cause a buffer overrun at either end of the array.

Default values are often used as a means to handle exceptions that can occur when improperly formatted or incorrect data are used for input. An exception handler catches the problem, and substitutes a default value to prevent a system crash. Using defaults can be potentially dangerous. For one thing, a single default value may not be appropriate for all situations. How the default values are stored presents another problem. Default values may be hardcoded into the source code, they may be retrieved from a data file, or they may be stored in cached memory. The latter two storage methods are particularly troublesome from a security standpoint. If an attacker were to somehow gain access to the default value file, they could alter the default values to gain unauthorized access to some resource protected by the system, or they could impede the abilities of authorized users to use the system. An example of cached default values is the form auto-fill feature found in most web browsers.

Although the auto-fill feature may be convenient when used on a home computer, when used on a public computer it can allow someone unauthorized access to another individual's personal, sensitive data if the cache is not emptied, and the browser's history deleted.

5. Provide restart capabilities.

   Systems that are in continuous operation, or systems that must process lengthy transactions, should have recovery capabilities should the system fail. Most recovery mechanisms involve creating and maintaining periodic copies of the state of the system, or of transaction details. Sommerville uses the term **checkpoint** to describe a saved state, but the term **restore point** is also common. Should a system failure occur, when the system is restarted it can resume operation on the most recent checkpoint. Multiple checkpoints can be stored in chronological order, allowing for prior checkpoints to be used if necessary. The Windows operating system is one example of this. If the system restore capability is enabled, restore points are created each time a system update is installed. If an update proves to be incompatible with an installed application, the system can be rolled back to the first restore point where the application is functional.

6. Check array bounds.

   This is the third time we've mentioned array bounds in this section, which underscores the importance of arrays in internal data storage. If you have a 0-based array of length 10, it is essential to ensure that all operations on that array occur at the memory addresses referenced by array indexes 0 through 9. Allowing access beyond those bounds constitutes an error, which can be exploited as a security hole. Many programming languages have built-in mechanisms that prevent array bounds from being violated. These mechanisms incur a certain amount of processing overhead that impacts performance. In many cases the overhead cost is negligible, but for systems where time is critical, bounds checking may be problematic. In those cases, it is necessary to use a programming language such as C, that does not perform implicit bounds checks, and ensure that the code is written to prevent array bounds from being violated.

7. Include timeouts when calling external components.

   Systems that must communicate with external devices have little control over how quickly those devices respond, if they respond at all. If a server goes down, for example, processes that are trying to communicate with the server could wind up being stuck in an infinite loop of waiting for a response. Depending on the number of processes attempting to communicate with external components, and the amount of resources used by those processes, a system can quickly grind to a halt. To prevent a system from freezing up in this way, all communications with external devices should be governed by a timeout mechanism that allows an attempt at communication to proceed for a specified time limit. If communication is established within the specified time window, operation can continue; if communication fails, however, the timeout mechanism can terminate the connection and inform the user or some other system component of the communication failure. Timeouts are not foolproof, though. It is conceivable to have a situation where each communication with an external device takes nearly the entire specified time window to establish. Since communication is eventually established, the system continues operating, but the result is a drastic slowdown in performance.

8. Name all constants that represent real-world values.

   Most programs, at some point, make use of values that never change; i.e., constants. Due to their static nature, constants are typically hardcoded into a program. Giving constants meaningful names improves the readability of the code, and helps prevent incorrect values from being used. The key word here is *meaningful*. Names should be chosen that concisely impart what the constants mean. They should not be too short, or make use of non-standard abbreviations. Neither should they be too lengthy. Consider the following examples based on the speed of light example Sommerville mentions in the text. Can you think of some pros and cons for each?

- SOL
- SPEED_OF_LIGHT
- SPEED_OF_LIGHT_M-S
- SPEED_OF_LIGHT_METERS_PER_SECOND