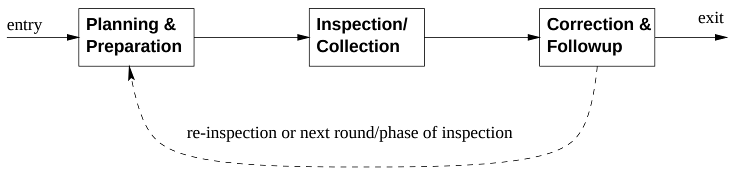

Figure 14.1. Generic inspection process

executable programs before one can start performing inspection. Consequently, the urgent need for QA and defect removal in the early phases of software development can be supported by inspection, but not by testing. In addition, various software artifacts available late in the development can be inspected but not tested, including product release and support plans, user manuals, project schedule and other management decisions, and other project documents. Basically, anything tangible can be inspected.

Because of this wide variety of objects for inspection, inspection techniques also vary considerably. For example, code inspection can use the program implementation details as well as product specifications and design documents to help with the inspection. Inspection of test plans may benefit from expected usage scenarios of the software product by its target customers. Inspection of product support plans must take into account the system configuration of the product in operation and its interaction with other products and the overall operational environment. Consequently, different inspection techniques need to be selected to perform effective inspection on specific objects.

Similarly, there are different degrees of formality, ranging from informal reviews and checks to very formal inspection techniques associated with precisely defined individual activities and exact steps to follow. Even at the informal end, some general process or guidelines need to be followed so that some minimal level of consistency can be assured, and adequate coverage of important areas can be guaranteed. In addition, some organizational and tool support for inspection is also needed.

Generic inspection process

Similar to the generic QA process we described in Chapter 3, all three basic elements of planning, execution, and followup are represented in the generic inspection process in Figure 14.1.. The major elements are explained below:

- *Planning and preparation:* Inspection planning needs to answer the general questions about the inspection, including:
  - What are the objectives or goals of the inspection?
  - What are the software artifacts to be inspected or the objects of the inspection?
  - Who are performing the inspection?
  - Who else need to be involved, in what roles, and with what specific responsibilities?
  - What are the overall process, techniques, and followup activities of the inspection?

  In the inspection literature, the term “inspection” is often used to denote the inspection meeting itself. Preparation for such meetings, as well as all the related activities lead-

Table 15.1. Example symbolic execution traces

part	condition	x	y
`if` $x > 0$	$x > 0$		
$y \leftarrow x$			x

part	condition	x	y
`if` $x > 0$	$x \leq 0$		
$y \leftarrow -x$			$-x$

Symbolic execution plays an important role in this approach. For example, the different traces of "`if`" statement through symbolic execution are used to determine parallel conditional assignments. Similarly, "`while`" involves "`if`" in recursive definition, therefore also involves corresponding symbolic execution. The functional nesting can be traced through symbolic execution as well. For the above example of calculating the absolute values with

$$\texttt{if}\ x \geq 0\ \texttt{then}\ y \leftarrow x\ \texttt{else}\ y \leftarrow -x$$

we have the two traces in the symbolic execution in Table 15.1..

Full details about symbolic execution and its used in this verification approach can be found in (Mills et al., 1987a). In essence, symbolic execution is a forward flow techniques, contrasting with the backward chaining technique for the axiomatic and wp approaches.

15.3.3 General observations

Although we didn't go through detailed examples for the functional and wp approaches, and the proof procedures are somewhat different, several observations are true for all three formal verification approaches, including:

- The difficulty of producing correctness proofs, particularly for loops, where the selection of proper loop invariant plays an important role, but there isn't a uniform formula for doing the selection. Some heuristics based on people's understanding, prior knowledge, or insight, are typically used to select such invariants. Sometimes, a trial-and-error strategy is necessary to consider multiple candidates before a workable solution can be found.

- In general, many steps are involved in the correctness proofs, and the proof can be fairly long and complicated even for relatively small-sized programs. As a rule of thumb, the length of the proof is typically one order of magnitude longer than the program itself.

- The proof process can generally benefit from some hierarchical structures and related abstractions as guide for different parts, in much of the same way as stepwise abstraction used as a code reading techniques described in Chapter 14. These abstractions can also help us in dealing with some of the difficulties noted above, such as deriving loop invariant based on abstraction of the loop body.

The first two of the above observations make it difficult to apply formal verification techniques on large-scale software products. In addition, we also need to deal with various other aspects and complications for larger programs, including: arrays and functions, procedures, modules, and other program components, and sometimes complications from things such as physical limitations, side effects, and aliases. Because of these, various "partial" and/or semi-formal verification techniques have been suggested, as described below.

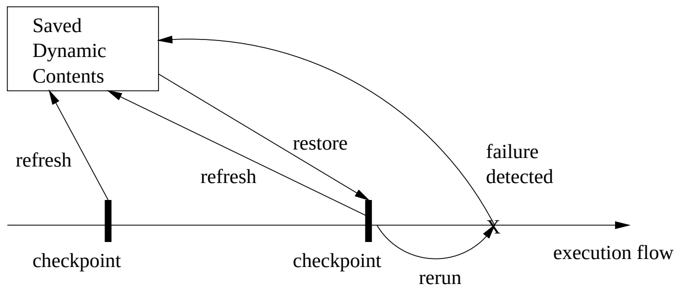

Figure 16.1. Fault tolerance with recovery blocks

are some problems or suspected irregularities associated with this primary version. When the backup is provided by the same program, we have the recovery block or checkpointing-and-recovery technique described in Section 16.2. In the case where backup is provide by a different program, if the program has the same functionality, we can consider it as a combination of backup and duplication ideas. However, this backup program is more likely to have somewhat different functionality than the primary program it backs up, for example, with reduced functionality to allow for speedy recovery, or to have general backup procedures that serves multiple purposes. All these and other fault tolerance techniques and related topics are described in much more detail in (Lyu, 1995b).

Within failure containment techniques, we can focus either on the accident prevention before accidents happen, or focus on damage control or reduction after accidents happen. In the former case, we try to limit the scope and impact of failures so that they do not lead to accidents. In the latter case, we try to reduce the accident damage through various techniques. Both these categories are described in Section 16.4, and related analysis techniques are also covered therein. All these and other failure containment techniques and related topics on software safety and embedded systems are described in much more detail in (Leveson, 1995).

16.2 FAULT TOLERANCE WITH RECOVERY BLOCKS

With the increasingly faster and faster processors, we may have the luxury of repeating some computational tasks within a prescribed time limit without seriously affecting the system performance. Under this circumstance, we can use recovery blocks to repeatedly establish checkpoints, and repeat certain computational steps when dynamic problems are observed or suspected, as described in this section.

Basic operations of systems using recovery blocks

The use of recovery blocks introduces duplication of software executions so that occasional failures only cause loss of partial computational results but not complete execution failures. For example, the ability to dynamically backup and recover from occasional lost or corrupted transactions is built into many critical databases used in financial, insurance, health care, and other industries. Figure 16.1. illustrates this technique, and depicts the major activities involved:

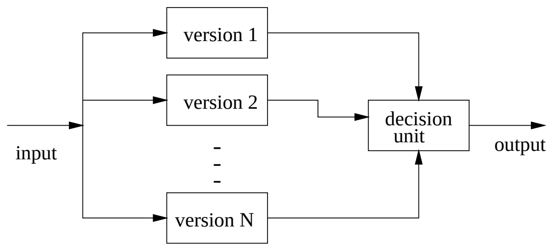

Figure 16.2. Fault tolerance with NVP

while lower frequency leads to longer and more costly recovery. An optimal frequency balances the two and incurs minimal overall cost. Alternative checkpoint strategies might also be used, for example, performing partial checkpointing for only those contents that are more likely to change more frequently than other contents. Therefore, the overall performance could be improved.

Another issue is the maintenance and followup activities to normal operations. As we noted before, repeated failures need to be dealt with by taking the system off-line for defect analysis and fixing. However, for normal operations, some information about the re-runs should be recorded and analyzed at a later time, either parallel to system operations or when the system is off-line. The key determination is whether these re-runs are truly due to rare environmental disturbances, or if software faults are to blame. In the latter case, the related software faults need to be located and fixed at the earliest opportunity.

16.3 FAULT TOLERANCE WITH N-VERSION PROGRAMMING

N-version programming (NVP) is another way to tolerate software faults by directly introducing duplications into the software itself (Avižienis, 1995). NVP is generally more suitable than recovery blocks when timely decisions or performance are critical, such as in many real-time control systems, or when software faults, instead of environmental disturbances, are more likely to be the primary sources of problems.

16.3.1 NVP: Basic technique and implementation

The basic technique is illustrated in Figure 16.2. and briefly described below:

- The basic functional units of the software system consist of N parallel independent versions of programs with identical functionality: version 1, version 2, . . ., version N.
- The system input is distributed to all the N versions.
- The individual output for each version is fed to a decision unit.
- The decision unit determines the system output using a specific decision algorithm. The most commonly used algorithm is a simple majority vote, but other algorithms are also possible.

The decision algorithm determines the degree of fault tolerance. For example, when the simple majority rule is used, the system output would be the correct one as long as at least

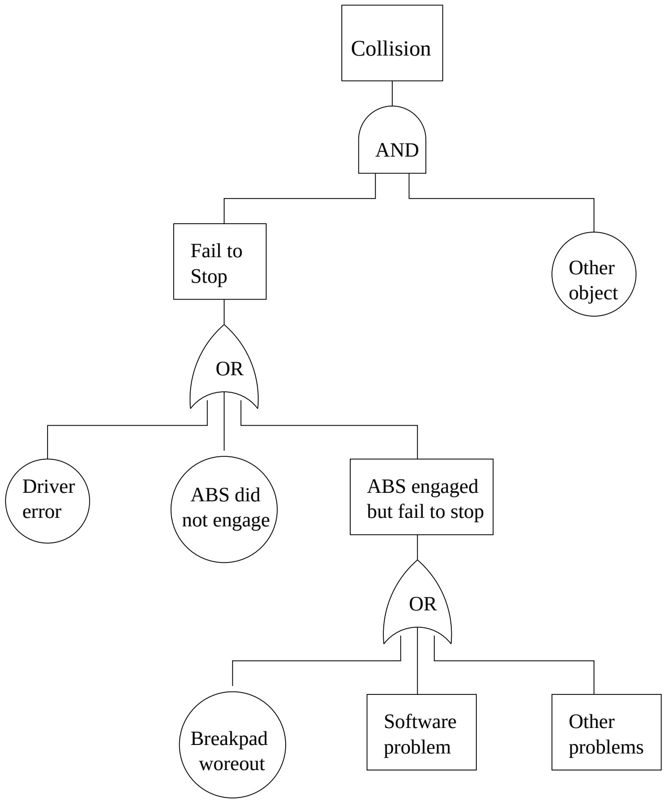

Figure 16.3. Fault-tree analysis (FTA) for an automobile accident

connected through logical connectors, AND, OR, NOT, to represent logical relations among sub-conditions.

- The analysis follows a top-down procedure: starting with the top event and recursively analyzing each event or condition to find out its logical conditions or sub-conditions. The top event is usually associated with an accident and is represented as the root node of the tree.
- We stop this recursive procedure at a terminal node under one of several conditions:
  - The current node is well understood, therefore there is no need to analyzed it further.
  - We can not break a node into its sub-conditions any further (an *atomic* node).
  - We do not have enough information to do so.
- The terminal nodes are associated with the so-called basic or primary events or conditions represented as circles. The non-terminal nodes in-between are associated with intermediate events or conditions represented as rectangles.

As an example of FTA, consider the collision between an object (representing an obstacle) and an automobile that fails to stop, even though it is equipped with an anti-lock break system

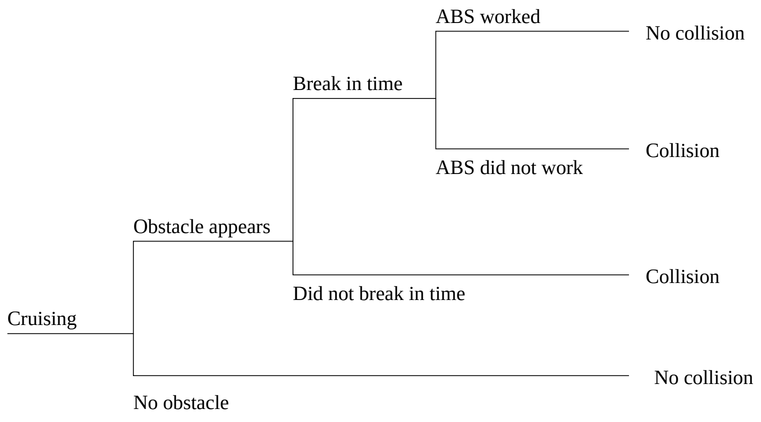

Figure 16.4. Event-tree analysis (FTA) for an automobile accident

techniques, appropriate actions can be applied to negate the conditions, to disrupt the event-chain, or otherwise provide a resolution to these hazards. Generic ways for hazard resolution in accident prevention include the following:

- *Hazard elimination* through substitution, simplification, decoupling, elimination of specific human errors, and reduction of hazardous materials or conditions. These techniques are similar to the defect prevention and defect reduction techniques, but with a focus on those controllable events or conditions (terminal nodes) involved in hazardous situations based on FTA results.

- *Hazard reduction* through design for controllability (e.g., automatic pressure release in boilers), use of locking devices (e.g., hardware/software interlocks), and failure probability minimization using safety margins and redundancy. These techniques are similar to the fault tolerance techniques, where local failures are contained without leading to system failures. However, the actions are guided by FTA and ETA results to focus on the key events, conditions, and sequences that are potentially related to accidents.

- *Hazard control* through reducing exposure, isolation and containment (e.g., barriers between the system and the environment), protection systems (active protection activated in case of hazard), and fail-safe design (passive protection, fail in a safe state without causing further damages). These techniques reduce the severity of failures, therefore weakening the link between failures and accidents.

In the above hazard resolution activities, some specific results from FTA can be used. For example, component replacement could be focus on those parts and areas that are linked through FTA as conditions for accidents. The software components thus identified can be the focus of formal verification activities, in the so called safety verification instead of broad-based formal verification of all the system components. We can also design lock-in, lock-out, and interlock devices, using a mixture of software and hardware technologies, to negate logical relations represented in FTA to prevent related accidents from happening. Similarly, some specific results from ETA can be used in hazard resolution, especially in hazard reduction and hazard control strategies. For example, barriers created between the critical and other paths, as well as other isolation and containment measures, can be applied to break or disrupt the chain of events that can lead to accidents.

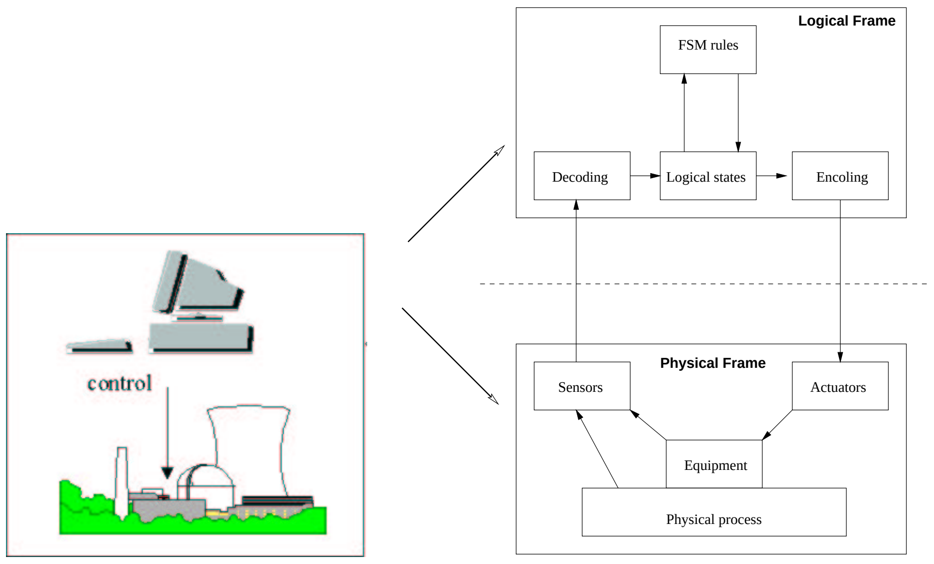

Figure 16.5. Two frame model for a CCSCS

For heterogeneous systems involving computers and related software, the software subsystem and the physical subsystem demonstrate vastly different operational behavior and characteristics. For example, common assumptions for hardware and equipment, such as wear, aging, and decay, are not immediately applicable to software domain. Various models, such as the two-frame model (or TFM) (Yih and Tian, 1998), were developed to analyze such systems.

In TFM, a heterogeneous system, such as a computer-controlled safety-critical system (CCSCS), is divided into a logical subsystem (or logical frame) and a physical subsystem (or physical frame). The logical subsystem corresponds to the computer controller, and the physical subsystem is monitored and controlled by the computer controller through sensors and actuators, as graphically illustrated in Figure 16.5.. TFM is similar to the "four variable model" described in (Parnas and Madey, 1995), but the symmetry between the two frames was highlighted instead of treating the software as the center and the physical subsystem as the environment. This perspective also gives us a better way to analyze the similarities and differences between the two frames to ensure and improve their safety. In such a heterogeneous system, failures may involve many different scenarios, including:

- Software failures due to defects in software design and implementation, which can be addressed to a large degree by QA techniques we have described so far in this book.

- Hardware or equipment failures due to wear, decay, or other physical processes, which is the main subject of traditional reliability and safety engineering, and can be largely addressed by related techniques.

- Communication/interface failures due to erroneous interactions among different subsystems or components.

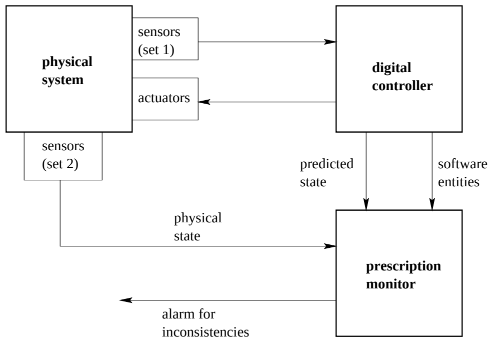

Figure 16.6. Prescription monitor for safety assurance

Most computer-related accidents in CCSCS can be traced back to problems in the interface or interactions among the components of the systems, particularly between the computer controller and the surrounding environment (Mackenzie, 1994). Therefore, hazard analyses focusing on the interaction/interface problems were performed to develop a technique for hazard prevention and safety improvement.

16.5.2 Prescriptive specifications for safety

In TFM, the commonly noted interface/interaction problems are mostly manifested as frame inconsistency problems. The primary causes for these inconsistencies can be identified to be the fundamental differences between the logical and physical frames, as follows:

- Physical states generally demonstrate regular behavior or form *total functions* according to physical laws; while the *discrete* software states usually form *partial functions*.
- There are typically *invariants* or *limits* reflecting physical laws, which, when implemented in software, may be violated or surpassed in failure situations.

However, because of the ultimate flexibility offered by software, if we could derive some *prescriptive specifications* as maintainable formal assertions for the logical frame, we can effectively keep the logical frame consistent with its physical frame, thus preventing various hazardous conditions from occurring. The logical subsystem could be enhanced to include a *prescription monitor*, illustrated in Figure 16.6.. The prescription monitor takes input from both the logical and physical subsystems, automatically checks prescriptive specifications to assure system integrity, and sounds alarms or carries out emergency actions if any of these assertions is violated.

A series of experiments was conducted to evaluate the effectiveness of this technique. based on report of actual scenarios of a nuclear accident. 19 hazard situations were tested in the simulated nuclear reactor control system, covering a wide variety of errors representative of realistic situations. In all the 19 instances, errors have been successfully detected on the spot by checking the prescriptive specifications developed above.

The approach above can be interpreted as using formal methods, in particular model checking, on CCSCS. However, system modeling and hazard analysis play a very important role in identifying the areas to focus, possible prescriptive specification, as well as the checking of these properties. This approach can be considered as a specific adaptation of the comprehensive approach in (Leveson, 1995) where hazard analysis and identification

Table 17.9. General comparison for different QA alternatives

QA alternative	Applicability	Effectiveness	Cost
testing	code	occasional failures	medium
defect prevention	known causes	systematic problems	low
inspection	s/w artifacts	scattered faults	low $\sim$ medium
formal verification	formal spec.	fault absence	high
fault tolerance	duplication	rare-cond. failures	high
failure containment	known hazards	rare-cond. accidents	highest

in Chapter 20 can be performed to identify systematic problems and select specific preventive actions to deal with the identified problems.

- Inspection and testing are applicable to different situations, and effective for different defect types at different defect levels. Therefore, inspection can be performed first to lower defect levels by directly detecting and removing many localized and static faults, then testing can be performed to remove the remaining faults related to dynamic scenarios and interactions. To maximize the benefit-to-cost ratio, various risk identification techniques covered in Chapter 21 can be used to focus inspection and testing effort on identified high-risk product components.

- Software safety assurance (especially hazard and damage control), fault tolerance, and formal verification techniques cost significantly more to implement than traditional QA techniques. However, if consequence of failures is severe and potential damage is high, they can be used to further reduce the failure probability, or to reduce the accident probability or severity.

The comparison of the applicability, effectiveness, and cost of these QA alternatives in this chapter can help software professionals choose appropriate QA alternatives and related techniques, and tailor or integrate them for specific applications. Together with the measurement and analysis activities described in Part IV, they can help us arrive at an optimal strategy for software QA and achieve quantifiable quality improvement.

Problems

17.1 In software engineering literature, there are various studies comparing different QA alternatives and techniques based on empirical data. Scan through some recent publications and read some articles on this topic, and compare their results with the general comparisons described in this chapter.

17.2 Most empirical studies mentioned above typically compare one QA alternative to another (e.g., inspection vs testing), or compare different techniques within a general category (e.g., different inspection processes or techniques). Can you replicate some of these studies in your work?

17.3 Can you use the comparison questions listed in this chapter to compare individual testing techniques?

17.4 How would the applicability of different QA alternatives be different when other software processes (non-waterfall ones) are used?