

covered by the methods, and three groups evaluated which of the causal factors the methods could have identified. HAZOP was the best, identifying 36 percent of the contributors. However, only 55 percent of the contributors were expected to be covered by the four methods [323, 322]. This result is particularly poor given that the analysis involved only determining which factors *could potentially* be identified by the methods—the number actually identified in any application would be expected to be lower.

Many evaluations of the predictive accuracy of reliability estimates have been done for individual instruments and components; these studies vary widely in their results. In a reliability benchmark exercise, 10 teams from 17 organizations and from 9 European countries performed parallel reliability analyses on a nuclear power plant primary cooling system. The purpose was to determine the effect of differences in modeling and data. The ratio between the highest and lowest frequencies calculated for the top event of the different fault trees was 36. When a unified fault tree was quantified by different teams using what each considered to be the best data, the corresponding ratio was reduced to 9.

#### 14.15 Conclusions

Many different hazard analysis techniques have been proposed and are used, but all have serious limitations and only a few are useful for software. But whether these techniques or more ad hoc techniques are used, we need to identify the software behaviors that can contribute to system hazards. Information about these hazardous behaviors is the input to the software requirements, design, and verification activities described in the rest of this book.

## Software Hazard and Requirements Analysis

*Computers do not produce new sorts of errors. They merely provide new and easier opportunities for making the old errors.*

—Trevor Kletz  
*Wise After the Event*

The vast majority of accidents in which software was involved can be traced to requirements flaws and, more specifically, to incompleteness in the specified and implemented software behavior—that is, incomplete or wrong assumptions about the operation of the controlled system or required operation of the computer and unhandled controlled-system states and environmental conditions. Although coding errors often get the most attention, they have more of an effect on reliability and other qualities than on safety [80, 200].

This chapter describes completeness and safety criteria for software requirements specifications. The criteria were developed both from experience in building such systems and from theoretical considerations [135, 136] and, in essence, are the equivalent of a requirements safety checklist for software. They can be used to develop informal or formal inspection procedures or tools for automated analysis of specifications. The criteria are general and apply to all systems, unlike the application-specific safety requirements identified in a system hazard analysis. Both application-specific hazards and general criteria need to be checked—in fact, one of the general criteria requires checking the application-specific hazards.

Lutz applied the criteria experimentally in checklist form to 192 safety-critical requirements errors in the Voyager and Galileo spacecraft software. These errors had not been discovered until late integration and system test, and therefore they had escaped the usual requirements verification and software testing process [201]. The criteria identified 149 of the errors.<sup>1</sup> Any after-the-fact experiment of this sort is always suspect, of course; no proof is offered that these errors would have been found if the criteria had been applied to the requirements originally, but the fact that they were related to so many real, safety-critical requirements deficiencies is encouraging. It is not necessarily surprising, however, since most of the criteria were developed using experience with critical errors, incidents, and accidents in real systems.

Jaffe and colleagues have related the original criteria to a general state machine model of process control systems [136] that can be used to derive formal, automated safety analysis procedures for specification languages based on state machines. This chapter describes additional criteria that were not included in earlier papers. The criteria are described only informally here; readers are referred to the research papers for a formal treatment.

## 15.1 Process Considerations

The software hazard analysis process will be influenced by the underlying accident model being used and its assumptions about the contribution of computers to accidents. Computers contribute to system hazards by controlling the actions of other components (including humans) either directly or indirectly. Humans are controlled to some degree by providing the information to operators or designers on which they base their decisions.

In an energy or chain-of-events model of accidents, software contributes to hazards through computer control of the energy sources, the release or flow of energy, the barriers, or the events that lead to accidents. In a systems theory model that assumes accidents arise from the interactions among components, software contributes directly to safety through computer control of these interactions.

The tasks of the software safety process defined in Section 12.1.1 that relate to software hazard analysis include:

1. Trace identified system hazards to the software–hardware interface. Translate the identified software-related hazards into requirements and constraints on software behavior.
2. Show the consistency of the software safety constraints with the software requirements specification. Demonstrate the completeness of the software

<sup>1</sup> Most of the unidentified errors involved design and thus were not the focus of the checklist.

requirements, including the human–computer interface requirements, with respect to system safety properties.

The most direct way to accomplish the first step is with a top-down hazard analysis that traces system hazards down to and into the subsystems. In this type of analysis, the software-related hazards are identified and traced into the software requirements and design. Currently, this goal is often accomplished by a fault tree analysis down to the software interface.

In addition, because software can do more than what is specified in the requirements (the problem of *unintended function*), the code itself must be analyzed to ensure that it cannot exhibit hazardous behavior—that the code satisfies its requirements (even if the required behavior is shown to be safe) is not enough. This chapter looks at requirements analysis, while design and code analysis are described in later chapters.

Software may also be the focus of a bottom-up subsystem hazard analysis. The practicality of this analysis is limited by the large number of ways that computers can contribute to system hazards. For example, a valve that has only two or three relevant discrete states (such as open, closed, or partially open) can be examined for the potential effects of these states on the system state. Computers, however, can assume so many states, exhibit so many visible and potentially important behaviors, and have such a complex effect on the system that complete bottom-up system analyses are, in most cases, impractical.

Bottom-up analyses may have some uses for software, but probably not for identifying software hazards. For example, some specific types of computer failure and incorrect behavior can be analyzed in a bottom-up manner for their effects on the system. In addition, forward analysis can examine (to some degree) a specification of software behavior to make sure that the behavior cannot lead to an identified hazard. To accomplish the latter, the software behavior must be specified completely, and the specification language should have a rigorously and unambiguously defined semantics and be readable by application experts and the user community. If the specification and analysis is not readable and reviewable by system safety and application experts, confidence in the results will be lessened.

Readability and reviewability will be enhanced by using languages that allow building models that are semantically close to the user's mental model of the system. That is, the *semantic distance* between the model in the expert's mind and the specification model should be minimized. In addition, reading the specification or reviewing the results of an analysis should not require training in advanced mathematics. Ideally, the specification language should reflect the way that engineers and application experts think about the system, not the way mathematicians do.

The second step of the process is to document the identified software behavioral requirements and constraints and to show that the software requirement

specification satisfies them. This step also includes demonstrating the completeness of the software requirements specification with respect to general system safety properties.

Most current software hazard and requirements analyses are done in an ad hoc manner. Some more systematic approaches have been proposed in research papers, but they have not been validated in practice on real projects. We do not know at this point which ones, if any, will turn out to be useful and including them here would make this book obsolete almost immediately. Instead, this chapter examines what needs to be accomplished in such an analysis.

## 15.2 Requirements Specification Components

Requirements specifications have three components: (1) a basic function or objective, (2) constraints on operating conditions, and (3) prioritized quality goals to help make tradeoff decisions.

The *constraints* define the range of conditions within which the system may operate while achieving its objectives. They are not part of the objectives; instead, they limit the set of acceptable designs to achieve the objectives. Constraints arise from quality considerations (including safety), physical limitations of the equipment, equipment performance considerations (such as avoiding overload of equipment in order to reduce maintenance), and process characteristics (such as limiting process variables to minimize production of byproducts).

Safety may be and often is involved in both functionality requirements and constraints. In an airborne collision avoidance system, for example, the basic mission—to maintain a minimum physical separation between aircraft—obviously involves safety. There are also safety-related constraints—for example, the surveillance part of the system must not in any way interfere with the radars and message communication used by the ground-based air traffic control (ATC) system; the system must operate with an acceptably low level of unwanted alarms (advisories to the pilot); and the deviation of the aircraft from their ATC-assigned tracks must be minimized. These constraints are not part of the system mission; in fact, they could most easily be accomplished by not building the system at all. Rather, they are limitations on how such a collision avoidance system may be realized.

Goals and constraints often conflict. Early in the development process, tradeoffs among functional goals and constraints must be identified and resolved according to priorities assigned to each. We are most interested in the conflicts and tradeoffs involving safety goals and constraints and in how adequately these goals and constraints are realized in the actual requirements. Goals are just that—they may not be completely achievable. Part of the safety process is to identify not only conflicts, but safety-related goals for the software that cannot be completely achieved. Decisions can then be made about how to protect the system using means other than the software or about the acceptability of the risks if no other

means exist. There is no formal or automated technique for this process; it requires the cooperation and joint efforts of the system and software engineers in applying their own expertise and judgment.

## 15.3 Completeness in Requirements Specifications

The most important property of the requirements specification with respect to safety is completeness or lack of ambiguity. The desired software behavior must have been specified in sufficient detail to distinguish it from any undesired program that might be designed. If a requirements document contains insufficient information for the designers to distinguish between observably distinct behavioral patterns that represent desired and undesired (or safe and unsafe) behavior, then the specification is ambiguous or incomplete [135, 136].

The term “completeness” here is not used in the mathematical sense, but rather in the sense of a lack of ambiguity from the application perspective: The specification is incomplete if the system or software behavior is not specified precisely enough because the required behavior for some events or conditions is omitted or is ambiguous (is subject to more than one interpretation).

If the behavioral difference between two programs that satisfy the same requirements is not significant for a subset of the requirements or constraints, such as those related to safety, then the ambiguity or incompleteness may not matter, at least for that subset: The specification is *sufficiently complete*. A set of requirements may be sufficiently complete with respect to safety without being absolutely complete: The requirement specification must simply be complete enough that it specifies *safe* behavior in all circumstances in which the system is to operate. Absolute completeness may be unnecessary and uneconomical for many situations.

Sufficient completeness, as defined here, holds only for a particular system and environment. The same specification that is sufficiently complete for one system may not be sufficiently complete for another. Therefore, software built from a sufficiently complete, but not absolutely complete, requirements specification may not be safe when reused in a different system. If the software is to be reused, either the specification must be absolutely complete (probably impossible in most cases) or a further requirements analysis is necessary.

The rest of this chapter defines criteria for completeness of software requirements specifications. Software requirements for the human–computer interface are no different than other requirements and are included in the completeness criteria described here. The criteria themselves (especially those for the human–computer interface) are not complete themselves and do not constitute the only checks that should be made. But they are useful in detecting incompleteness that is associated with hazards and accidents. In a sense, they represent a starting point for a safety checklist for requirements specification to which additions may be made as we discover the necessity.

Many types of incompleteness are application dependent and must be identified using system hazard analysis or top-down analysis. Jaffe notes that in any application, at any given point in time, there is a set of *kernel* requirements that derive from current knowledge of the needs and environment of the application itself [135]. These kernel requirements are analytically independent of one another—the need for the existence of any one of them cannot be determined from the existence of the others. For example, an autopilot program may or may not control the throttle along with the aerodynamic surfaces.

Without knowledge of the intent of the application, there can be no way to ascertain whether a particular requirements specification has a complete set of kernel requirements. This type of incompleteness must be identified by system engineering techniques that include modeling and analysis of the entire system with respect to various desired properties (such as safety). In other words, any safety implications of such incompleteness must be identified using system hazard analysis (as described, for example, in Section 14.12) rather than the type of subsystem hazard analysis described in this chapter.

On the other hand, subsystem hazard analysis applied to requirements can detect *incompletely specified* kernel requirements. In addition, this type of analysis, involving rigorous examination of the specified software behavior, may also be able to detect some genuine functionality inadvertently omitted during the system engineering process. For example, a specification that includes a requirement to generate an alert condition to tell an air traffic controller that an aircraft is too low is probably incomplete unless it also includes another requirement to inform the controller that an aircraft previously noted as too low is now back at a safe altitude [135]. Safety and robustness considerations can be exploited to develop application-independent criteria for detecting such incompleteness.

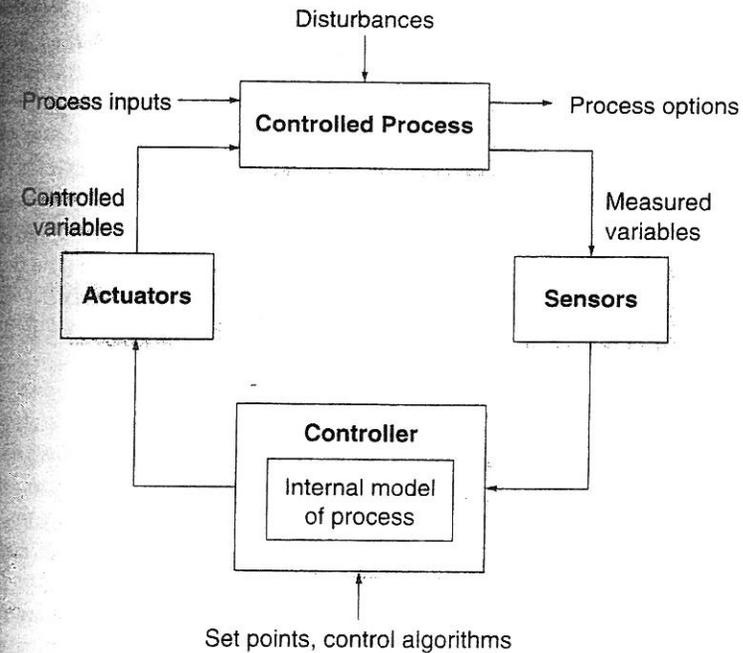


FIGURE 15.1

A black-box requirements specification captures the controller's internal model of the process. Accidents occur when the internal model does not accurately reflect the state of the controlled process.

## 15.4 Completeness Criteria for Requirements Analysis

A requirements specification describes the required black-box behavior of the component. Although design information is sometimes included in software requirements specifications, the safety analysis described here is concerned only with the black-box behavior of the software, which is the only aspect of the specification that can directly affect system hazards. Design analysis is covered in later chapters.

The requirements specification defines the function to be implemented by the computer. A description of any process control function uses as inputs

- The current process state inferred from measurements of the controlled process variables
- Past process states that were measured and inferred
- Past corrective actions (outputs) that were issued by the controller
- Prediction of future states of the controlled process

and produces the corrective actions (or current outputs) needed to achieve the process goals while satisfying the constraints on its behavior.

In this chapter, the control function is described using a state machine model. State machines are convenient models for describing computer behavior, and many specification languages use these models. The criteria are described here in terms of the components of a state machine model, but they could be translated to other models or applied to informal requirements specifications.

A state machine is simply a model that describes a system in terms of states and the transitions between the states. State machines are defined in Section 14.12 and an example is shown in Figure 14.11. The controller outputs to actuators are associated with state changes in the model, which are triggered by measurements of process variables (see Figure 15.1).

Theoretical control laws are defined using the *true* values of the process state. At any time, however, the controller has only *measured* values, which may

be subject to time lags<sup>2</sup> or measurement inaccuracies. The controller must use these measured values to infer the true state of the process and to determine the corrective actions necessary to maintain certain desirable properties in the controlled system. Considering the problems of measurement error and time lags is essential in developing safe control software.

A state machine model is an abstraction. As used here, it models the view of the process maintained by the computer (the internal model of the process), which is necessarily incomplete (Figure 15.1). Hazards and accidents can result from mismatches between the software view of the process and the actual state of the process—that is, the model of the process used by the software gets out of synch with the real process. For example, the software does not think the tank is full and therefore does not stop the flow into the tank, or it does not know that the plane is on the ground and raises the landing gear.

The mismatch occurs because the internal model is incorrect or incomplete or the computer does not have accurate information about the process state. For example, the model may not include a check for the proper process conditions before doing something hazardous—a check for weight on wheels is not included on the state transition associated with the output to raise the landing gear. Alternatively, the check may be included, but the computer may not have correct information about the current state of the plane.

Safety then depends on the completeness and accuracy of the software (internal) model of the process. A state machine specification of requirements explicitly describes this model and the functions performed by the software. The goal of completeness analysis basically is to ensure that the model of the process used by the software is sufficiently complete and accurate that hazardous process states do not occur. Completeness criteria are defined for each of the state machine parts: the states, the transition (triggering) events, the inputs and outputs, and the relationship between the transition events and their related outputs.

Completeness requires that both the characteristics of the outputs and the assumptions about their triggering events be specified:

$$\text{trigger} \implies \text{output}$$

In response to a single occurrence of the given stimulus or trigger, the program must produce only a single output set. A black-box statement of behavior allows statements and observations to be made only in terms of outputs and the externally observable conditions or events that stimulate or trigger them (the *triggers* for short). In terms of the state machine, this restriction means that both the states and the events on the transitions must be externally observable.

Not only must the output be produced given a particular trigger, but it must not be produced without the trigger:

$$\text{trigger} \longleftarrow \text{output}$$

<sup>2</sup> Time lags are delays in the system caused by sensor polling intervals or by the reaction time of the sensors, actuators, and the actual process.

A complete trigger specification must include all conditions that trigger the output, that is, the set of conditions that can be inferred from the existence of output. Such conditions represent assumptions about the environment in which the program or system is to execute.

The next sections informally describe what is required for a complete specification of the triggers and outputs and the other parts of a black-box state machine model of software behavior. Most of this discussion is taken from Jaffe [135].

#### 15.4.1 Human-Computer Interface Criteria

The human-computer interface has many possible completeness criteria. These criteria can be framed in terms of high-level abstractions applicable to this interface. Jaffe suggests that an *alert queue*, for example, is an abstraction with completeness criteria related to alert review and disposal, automatic reprioritization, and deletion [135]. An alert queue is an abstraction external to the computer and thus appropriate for a black-box requirements specification. Some appropriate human-computer abstractions and completeness criteria are presented in this chapter, but the essential requirements needed for other such abstractions can and should be developed.

For human-computer interface queues in general, the requirements specification will include

- Specification of the events to be queued
- Specification of the type and number of queues to be provided (such as alert or routine)
- Ordering scheme within the queue (priority versus time of arrival)
- Operator notification mechanism for items inserted in the queue
- Operator review and disposal commands for queue entries
- Queue entry deletion

A second important abstraction for the human-computer interface is a *transaction*, which may have multiple events associated with it. Multiple-event transactions require additional completeness criteria such as those to deal with preemption in the middle of a transaction.

Often, requirements are needed for the deletion of requested information. An air traffic controller, for example, may request certain graphic information such as the projected path of a trial maneuver for a controlled aircraft. A complete requirements specification needs to state when the trial maneuver graphics should disappear. Some actions by the operator should leave this trial maneuver display untouched (such as retrieving information from the aircraft's flight plan to evaluate the trial maneuver) while other actions should delete the transient information without requiring a separate clearing action (such as operator signoff).

In general, Jaffe identifies three questions that must be answered in the requirements specification for every data item displayable to a human:

1. What events cause this item to be displayed?
2. Can and should the display of this item ever be updated once it is displayed? If so, what events should cause the update? Events that trigger updates may be
  - External observables
  - The passage of time
  - Actions taken by the viewing operator
  - Actions taken by other operators (in multiperson systems)
3. What events should cause this data display to disappear?

In addition to data, the computer may control the labels (such as menus or software-labeled keys or buttons) associated with operator actions. Not only can these labels change, but the software may be responsible for such things as highlighting a recommended action or deleting labels for actions that are unavailable or prohibited under current conditions. Failure to specify all circumstances under which data items or operator-action entry labels should change is a common cause of specification incompleteness for the human-computer interface and a potential source of hazards.

Specific criteria for these human-computer interface requirements are integrated into appropriate sections of this chapter.

### 15.4.2 State Completeness

The operational states will, of course, be specific to the system. But in general, these states can be separated into normal and non-normal processing modes (where modes are just groups of states having a common characteristic), and completeness criteria can be applied to the transitions between these modes.

- *The system and software must start in a safe state. Interlocks should be initialized or checked to be operational at system startup, including startup after temporarily overriding interlocks.*

Transitions from normal operation to non-normal operation are often associated with accidents. In particular, when computers are involved, many accidents and failures stem from incompleteness in the way the software deals with startup and with transitions between normal processing and various types of partial or total shutdown.

- *The internal software model of the process must be updated to reflect the actual process state at initial startup and after temporary shutdown.*

Unlike other types of software, such as data processing software, an important consideration when developing software for process control is that the process continues to change state even when the computer is not executing. The correct behavior of the computer may depend on input that arrived before startup; what to do about this input must be included in the specification. Serious accidents have occurred because software designers did not consider state changes

while the system was in a manual mode and the computer was temporarily off-line. In one such accident in a chemical plant, described in Chapter 1, the computer was controlling the valves on pipes carrying methanol between the plant and a tanker, and a pump was stopped manually without the computer knowing it. A similar accident occurred in a batch chemical reactor when a computer was taken off-line to modify the software [158]. At the time the computer was shut down, it was counting the revolutions on a metering pump that was feeding the reactor. When the computer came back on-line, it continued counting where it had left off, which resulted in the reactor being overcharged.

- *All system and local variables must be properly initialized upon startup, including clocks.*

There are two startup situations: (1) initial startup after complete process shutdown and (2) startup after the software has been temporarily off-line but the process has continued under manual control. In both the initial startup and after temporary computer shutdown, the internal clock as well as other system and local variables will need to be initialized. In addition, the second case (where only the computer has been shut down) requires that the internal model of the process used by the software be updated to reflect the actual process state; the variables and status of the process, including time, will probably have changed since the computer was last operational.

A number of techniques are used for this resynchronization. Message serialization (numbering the inputs), for example, is a commonly used technique that can detect "lost" information and indicate potential discontinuities in software operations. Another technique often used involves checking elapsed time between apparently successive inputs by means of a self-contained timestamp in each input (requiring clock synchronization) or via reference to a time-of-day clock upon the receipt of each input.

- *The behavior of the software with respect to inputs received before startup, after shutdown, or when the computer is temporarily disconnected from the process (off-line) must be specified, or it must be determined that this information can be safely ignored, and this conclusion must be documented.*

If the hardware can retain a signal indicating the existence of an input after computer shutdown and prior to startup, the program has two startup states—the input is present or is not present—and at least two separate requirements must be specified: one for startup when there is indication of a prior input signal and one when there is not.

In the case of inputs that occur before program startup, the time of that input or the number of inputs is not observable by the software, but one or some of the inputs may be available to the computer after startup. Which inputs are retained is hardware dependent: Some hardware may retain the first input that arrived, some the most recent, and so on. To avoid errors, systems where the ordering of incoming data is important must include requirements to handle pre-startup inputs.

- *The maximum time the computer waits before the first input must be specified.*

Any specification for a real-time system should also include requirements to detect a possible disconnect occurring prior to program startup between the computer and the sensors or the process. After program startup, there should be some finite limit on how long the program waits for an input before it tries various alternative strategies—such as alerting an operator or shifting to an open-loop control mechanism that does not use the absent input. This criterion is very similar to a maximum-time-between-events criterion (discussed later), but it applies to the absence of even the first input of a given type. Even if the maximum time between events is checked, the special case of the first such interval after startup is often omitted or handled incorrectly. There may (and in general will) be a series of intervals  $d_1, d_2, \dots$  during which the program is required to attempt various ways of dealing with the lack of input from the environment. Eventually, however, there must be some period after which, in the absence of input, the conclusion must be that a malfunction has occurred.

- *Paths from fail-safe (partial or total shutdown) states must be specified. The time in a safe but reduced-function state should be minimized.*
- *Interlock failures should result in the halting of hazardous functions.*

The software may have additional non-normal processing modes such as partial shutdown or degraded operation. More completeness criteria for some of these mode transitions are described later.

The normal processing states may also be divided into subsets or modes of operation, such as an aircraft taking off, in transit, or landing. For safety analysis, the states may be partitioned into hazardous and nonhazardous modes with different completeness criteria applied to each.

- *There must be a response specified for the arrival of an input in any state, including indeterminate states.*

Completeness considerations require that there be a software response to the arrival of an input in any state, including the arrival of unexpected inputs for that state. For example, if an output is triggered by the receipt of a particular input when a device is in state ON, the specification must also handle the case where that input is received and the device is in state OFF. In addition, not being in state ON is not equivalent to being in state OFF, since the state of the device may be indeterminate (to the computer) if no information is available about its state. Therefore, a requirement is needed also to deal with the case when the input is received and the computer does not know if the device is ON or OFF.

Many software problems arise from incomplete specification of state assumptions. As an example, Melliar-Smith reports a problem detected during an operational simulation of the Space Shuttle software. The astronauts attempted to abort the mission during a particular orbit, changed their minds and canceled the abort attempt, and then decided to abort the mission after all on the next orbit. The software got into an infinite loop that appears to have occurred because the designers had not anticipated that anyone would ever want to abort twice on

the same flight [235]. Another example involves an aircraft weapons management system that attempts to keep the load even and the plane flying level by balanced dispersal of weapons and empty fuel tanks [235]. One of the early problems was that even if the plane was flying upside down, the computer would still drop a bomb or a fuel tank which then dented the wing and rolled off. In yet another incident, an aircraft was damaged when the computer raised the landing gear in response to a test pilot's command while the aircraft was standing on the runway [235].

In some cases, there really is no requirement to respond to a given input except in a subset of the states. But an input arriving unexpectedly is often an indication of a disconnect between the computer and the other components of the system that should not be ignored. For example, a target detection report from a radar that previously was sent a message to shut down is an indication that the radar did not do so, perhaps because its detection logic is malfunctioning. If, in fact, the unexpected input is of no significance, the requirements specification should still document the fact that all cases have been considered and that this case truly can be ignored (perhaps by specifying a “do nothing” response to the input).

### 15.4.3 Input and Output Variable Completeness

The inputs and outputs represent the information the sensors can provide to the software (the controlled variables) and the commands that the software can provide to the actuators (to change the manipulated variables). These input and output variables and commands must be rigorously defined in the documentation.

At the black-box boundary, *only* time and value are observable by the software. Therefore, the triggers and outputs must be defined only as constants or as the value and time of observable events or conditions. Events include program inputs, prior program outputs, program startup (a unique observable event for each execution of a given program), and hardware-dependent events such as power-out-of-tolerance interrupts. Conditions may be expressed in terms of the value of hardware-dependent attributes accessible by the software such as time-of-day clocks or sense switches.

- *All information from the sensors should be used somewhere in the specification.*

If information from the sensors is not used in the requirements, there is very likely to be an important omission from the specification. In other words, if an input can be sent to the computer, there should be some specification of what should be done with it.

- *Legal output values that are never produced should be checked for potential specification incompleteness.*

As with inputs, an important requirement for software behavior may have been forgotten if there is a legal value for an output that is never produced.

For example, if an output can have values *open* and *close* and the requirements specify when to generate an OPEN command but not when to generate CLOSE, the specification is almost certainly incomplete. Checking for this property may help to locate specification omissions.

#### 15.4.4 Trigger Event Completeness

The behavior of the control subsystem (in our case, the computer) is defined with respect to assumptions about the behavior of the other parts of the system—the conditions in the other parts of the control loop or in the environment in which the controller operates. A *robust* system will detect and respond appropriately to violations of these assumptions (such as unexpected inputs). By definition, then, the robustness of the software built from the specification depends upon the completeness of the specification of the environmental assumptions—there should be no observable events that leave the program's behavior indeterminate. These events can be observed by the software only in terms of trigger events, and thus completeness of the environmental assumptions is related to the completeness of the specification of the trigger events and the response of the computer to any potential inputs.

Documenting all environmental assumptions and checking them at runtime may seem expensive and unnecessary. Many assumptions are based on the physical characteristics of input devices and cannot be falsified even by unexpected physical conditions and failures. For example, an input line connected to a 1200-baud modem cannot fail in a way that causes the data rate to exceed 1200 baud. The interrupt signal may stick high (on), but for most modern hardware, that will stop data transfer, not accelerate it. If the environment in which the program executes ever changes, however, the assumption may no longer be valid; the 1200-baud modem may be upgraded to 9600 baud, for example. Similarly, if the software is ever reused, the environment for the new program may differ from that of the earlier use. Examples were provided in Chapter 2 of problems arising from the reuse of software in environments different from that for which it was originally built.

In addition to being documented, critical assumptions—those where the improper performance of the software can have severe consequences—should be checked at runtime. Examples abound of accidents resulting from incomplete requirements and nonrobust software. For example, an accident occurred when a military aircraft flight control system was intentionally limited in the range of control (travel) by the software because it was (incorrectly) assumed that the aircraft could not get into certain attitudes.

Even when real-time response is not required, it is important that the software or hardware log violations of assumptions for off-line analysis. A hole in the ozone layer at the South Pole was not detected for six years because the ozone depletion was so severe that a computer analyzing the data had been suppressing it. Having been programmed to assume that deviations so extreme must be sensor

errors [96]. Detecting errors early, before they lead to accidents, is obviously a desirable goal.

##### 15.4.4.1 Robustness Criteria

- *To be robust, the events that trigger state changes must satisfy the following:*
  1. *Every state must have a behavior (transition) defined for every possible input.*
  2. *The logical OR of the conditions on every transition out of any state must form a tautology.*
  3. *Every state must have a software behavior (transition) defined in case there is no input for a given period of time (a timeout).*

A *tautology* is a logically complete expression. For example, if there is a requirement on a transition that the value of an input be greater than 7, then a *tautologically* complete specification would also include transitions from that state when the input is less than 7 and equal to 7.

These three criteria together guarantee that if there is a trigger condition for a state to handle inputs within a range, there will some transition defined to handle data that is out of range. There will also be a requirement for a timeout that specifies what to do if no input occurs at all.

The use of an OTHERWISE clause (in specification languages that permit this) is not appropriate for safety-critical systems. Jaffe writes:

It was always tempting to guarantee the appropriate level of completeness at any given point by just adding an “otherwise, do nothing” requirement. But the more complex the situation, the more likely it is that there will be some interesting case concealed within the “otherwise.” It is better to explicitly delineate exactly what cases provide the “otherwise” condition and then check for tautological completeness [135].

##### 15.4.4.2 Nondeterminism

Another restriction can be placed on the transition events to require deterministic behavior:

- *The behavior of the state machine should be deterministic (only one possible transition out of a state is applicable at any time).*

Consider the case where the conditions on two transitions are that (1) the value of the input is greater than zero and (2) the value of the input is less than 2. If the input value is 1, then both transitions could be taken, leading to nondeterministic behavior of the software with respect to the requirements. The problem is eliminated by forcing all transitions out of a state to be disjoint (two transition conditions can never be true at the same time).

Although a specification does not have to be deterministic to be safe, nondeterminism greatly complicates safety analysis and may make it impractical to

perform thoroughly. Moreover, software to control the operation of many hazardous systems should be repeatable and predictable. Deterministic behavior aids in guaranteeing hard real-time deadlines; in analyzing and predicting the behavior of software; in testing the software; in debugging and troubleshooting, including reproducing test conditions and replicating operational events; and in allowing the human operator to rely on consistent behavior (an important factor in the design of the human-machine interface).

#### 15.4.4.3 Value and Timing Assumptions

Ensuring that the triggers in the requirements specification satisfy the previous four criteria is necessary, but it is not sufficient for trigger event completeness. The criteria ensure that there is always exactly one transition that can be taken out of every state, but they do not guarantee that all assumptions about the environment have been specified or that there is a defined response for all possible input conditions the environment can produce. Completeness depends upon the *amount* and *type* of information (restrictions and assumptions such as legal range) that is included in the triggers. The more assumptions about the triggers included, the more likely that the four above criteria will ensure that the requirements include responses to unplanned events.

Many assumptions and conditions are application dependent, but some types of assumptions are essential and should always be specified for all inputs to safety-critical systems. In real-time systems, the times of inputs and outputs are as important as the values. Digital flight control commands to ailerons, for example, may be dangerous if they do not arrive at exactly the right time: Flutter and instability (which can and do lead to the loss of the aircraft) result from improperly timed control movements, where the difference between proper and improper timing can be a matter of milliseconds [135]. Therefore, both value and time are required in the characterization of the environmental assumptions (triggers) and in the outputs.

#### Essential Value Assumptions

Value assumptions state the values or range of values of the trigger variables and events. An input may not require a specification of its possible values. A hardwired hardware interrupt, for example, has no value, but it may still trigger an output. When the value of an input is used to determine the value or time of an output, the acceptable characteristics of the input must be specified, such as range of acceptable values, set of acceptable values, or parity of acceptable values.

- *All incoming values should be checked and a response specified in the event of an out-of-range or unexpected value.*

As noted earlier, even where an assumption is not essential, it should be specified and checked whenever possible (whenever it is known) because the receipt of an input with an unexpected value is a sign that something in the en-

vironment is not behaving as the designer anticipated. Checking simple value assumptions on inputs is comparatively inexpensive. Since failure of such assumptions is an indication of various reasonably common hardware malfunctions or of misunderstanding about software requirements, it is difficult to envision an application where the specification should not require robustness in this regard—incoming values should have their values checked, *and* there should be a specified response in the event an unexpected value is received.

Some input values represent information about safety interlocks. These *always* need to be checked for values that may indicate failure and appropriate action taken.

#### Essential Timing Assumptions

The need for and importance of specifying timing assumptions in the software requirements stem from the nature and importance of timing in process control, where timing problems are a common cause of runtime failures. Timing is often inadequately specified for software. Two different timing assumptions are essential in the requirements specification of triggers: timing intervals and capacity or load.<sup>3</sup>

**Timing Intervals.** While the specification of the value of an event is usual but optional, a timing specification is *always* required: The mere existence of an observable event (with no timing specification) in and of itself is never sufficient—at the least, inputs must be required to arrive after program startup (or to be handled as described previously).

- *All inputs must be fully bounded in time, and the proper behavior specified in case the limits are violated or an expected input does not arrive.*

Trigger specifications include either the occurrence of an observable signal (or signals) or the specification of a duration of time without a specific signal. Both cases need to be fully bounded in time or a capacity requirement is necessary.

The arrival of an input at the black-box boundary has to include a lower bound on the time of arrival and will, in general, include an upper bound on the interval in which the input is to be accepted. Requirements dealing with input arriving outside the time interval and the nonexistence of an input during a given interval (a duration of time without the expected signal) also have to be defined. The robustness criteria will ensure that a behavior is specified in case the time limits are violated.

The acceptable interval will always be bounded from below by the time of the event that brought the machine to the current state. Some other lower bound may be desirable, but the limit must always be expressed in terms of previous, observable events.

<sup>3</sup> *Load* here refers to a rate, whereas *capacity* refers to the ability to handle that rate.

Even requirements such as “The event  $I$  shall occur at 11:00 A.M.” are ambiguous. The value of the time of  $I$  is the value of the reference clock observed “simultaneously” with the occurrence of  $I$ . Conceptually, the clock is ticking at the rate of one tick per unit of temporal precision. In general,  $I$  will occur between two ticks of any clock, no matter how frequent the ticks. Therefore, to say that it must occur *exactly* at 11:00 A.M. is meaningless unless the specification also states what clock is to be used. Even then, the time cannot be known more precisely than the granularity of the clock. Concrete discussion of specific clocks should be avoided in a software requirements specification; all that is really necessary to know is the required precision of the clock. Translating this precision into an attribute of the input results in a requirement with bounding inequalities rather than an equality, such as 10:59 A.M.  $< time(I) < 11:01$  A.M. (commonly written as  $time(I) = 11:00$  A.M.  $\pm 1$  min), which specifies an accuracy of plus or minus a minute on the timing.

- A trigger involving the nonexistence of an input must be fully bounded in time.

For requirements that involve the nonexistence of a signal during a given interval, both ends of the interval must be either bound by or calculable from observable events. Informally, there must be an upper bound on the time the program waits before responding to the lack of a signal. There must also be a specific time to start timing the lack of inputs or an infinite number of intervals (and thus outputs) will be specified. For example, a requirement of the type “If there is no input  $I$  for 10 seconds, then produce output  $O$ ” is not bound at the lower end of the interval and is therefore ambiguous. Should the nonexistence interval start at time  $t$ , at  $t + \epsilon$ ,  $t + 2\epsilon$ , . . . ? An example of a complete specification might be “If there is no input  $I_1$  for 10 seconds after the receipt of the previous input  $I_2$ , then produce output  $O$ .” The observable event need not occur at either end of the interval—the ends need only be calculable from that event, such as “There is no input for 5 sec preceding or following event  $E$ .”

**Capacity or Load.** In an interrupt-driven system, the count of unmasked input interrupts received over a given period partitions the computer state space into at least two states: normal and overloaded. The required response to an input will differ in the two states, so both cases must be specified.

Failures of critical systems due to incorrectly handled overload conditions are not unusual. A bank in Australia reportedly lost money from the omission of proper behavior to handle excessive load in an automated teller machine (ATM) [266]. When the central computer was unable to cope with the load, the ATMs dispensed cash whether or not the customer had adequate funds to cover the withdrawal. Failure to handle the actual load, although annoying to customers, would not by itself have caused as much damage as that resulting from the lack of an explicit (and reasonable) overload response behavior. Much more serious consequences resulted from the failure of a London ambulance dispatching system in 1992 under an overload condition [68]. According to reports, neither of these

systems had been tested under a full load, and each, obviously, had inadequate responses to a violation of the load assumptions.

Although inputs from human operators or other slow system components may normally be incapable of overloading a computer, various malfunctions can cause excessive, spurious inputs and so they also need a load limit specified. In one accident, an aircraft went out of control and crashed when a mechanical malfunction in a fly-by-wire flight control system caused an accelerated environment that the flight control computer was not programmed to handle [88]. Robustness requires specifying how to handle excessive inputs and specifying a load limit for such inputs as a means of detecting possible external malfunctions.

- A minimum and maximum load assumption must be specified for every interrupt-signaled event whose arrival rate is not dominated (limited) by another type of event.

In general, inputs to process control systems should have both minimum and maximum load assumptions for all interrupt-signaled events whose arrival rate is not dominated by another type of event. If interrupts cannot be disabled (locked out) on a given port, then there will always be some arrival rate for an interrupt signaling an input that will overload the physical machine. Either the machine will run out of CPU resources as it spends execution cycles responding to the interrupts, or it will run out of memory when it stores the data for future processing. Both hardware selection and software design require an assumption about the maximum number of inputs  $N$  signaled within an interval of time  $d$ , so this information should be in the requirements specification.

Multiple load assumptions are meaningful although not necessarily required in any given case. For example, the load could be 4 per second but not more than 7 in any two seconds nor more than 13 in four seconds, and so on. One load assumption is required; multiple assumptions may derive from application-specific considerations. Multiple loads can also be assumed for a given input based on additional data characteristics, such as not more than 4 inputs per second when the value of input  $I$  is greater than 8, but not more than 3 per second when  $I$  is greater than 20.

- A minimum-arrival-rate check by the software should be required for each physically distinct communication path. Software should have the capacity to query its environment with respect to inactivity over a given communication path.

A load assumption with  $N$  equal to 1 is the same as an assumption on the minimum time between successive inputs. Robustness requires the specification of a minimum arrival rate assumption for most, if not all, possible inputs since indefinite, total inactivity by any real-world process is unlikely. Robust software should be able to query its environment about inactivity over a given communication path. Requirements of this type lead to the use of sanity and health checks in the software, as described in Chapter 16.

Where interrupts can be masked or disabled, the situation is more complicated. If disabling the interrupt can result in a “lost” event (depending on the hardware, the duration of the lockout, and the characteristics of the device at the other end of the channel), the need for a load assumption will depend on how the input is used. If the number of inputs  $I$  is completely dominated by (dependent on) the number of inputs of a different type, then a load assumption for  $I$  is not needed.

Even if a particular statistical distribution of arrivals over time is assumed and specified, a load limit assumption is still required. Assuming that the arrival distribution fits a Poisson distribution, for example, does not preclude the possibility, no matter how improbable, of it exceeding a given capacity. If capacity is exceeded, there must be some specification of the ways that the system can acceptably fail soft or fail safe.

- *The response to excessive inputs (violations of load assumptions) must be specified.*

The requirements for dealing with overload generally fall into one of five classes:

1. Requirements to generate warning messages.
2. Requirements to generate outputs to reduce the load (messages to external systems to “slow down”).
3. Requirements to lock out interrupt signals for the overloaded channels.
4. Requirements to produce outputs (up to some higher load limit) that have reduced accuracy or response time requirements or some other characteristic that will allow the CPU to continue to cope with the higher load.
5. Requirements to reduce the functionality of the software or, in extreme cases, to shut down the computer or the process.

The first three classes are handled in an obvious way. The behavior in the fourth and fifth classes (commonly called performance degradation and function shedding) should be graceful—that is, predictable and not abrupt.

- *If the desired response to an overload condition is performance degradation, the specified degradation should be graceful and operators should be informed.*

Abrupt or random (although bounded) degradation often needs to be avoided. Certainly for operator feedback, predictability is preferable to variability, at least within limits, even if the cost is a slight increase in average response time [84]. For safety considerations, however, as discussed in Chapters 6 and 17, when the program changes to a degraded performance mode or the computer is compensating for extreme or non-normal conditions, the operator should always be informed. Additional action may be required, such as disabling or requesting resets of busy interfaces or recording critical parameters for subsequent analysis.

*If function shedding or reconfiguration is used, a hysteresis delay and other checks must be included in the conditions required to return to normal processing load.*

Once a state of degraded performance has been entered, a specification of the conditions required to return to a normal processing mode, including a *hysteresis delay*, is necessary. After detecting a capacity violation, the system must not attempt to return to the normal state too quickly; the exact same set of circumstances that caused it to leave may still exist. For example, assume that the event that caused the state to change is the receipt of the  $n$ th occurrence of input  $I$  within a period  $d$ , where the load is specified as limited to  $n - 1$ . Then, if the system attempts to return to normal within a period  $x \ll d$ , the very next occurrence of an  $I$  might cause the state to change again to the overload state. The system could thus ping-pong back and forth. A hysteresis factor simply ensures that the transition to normal operation is not too close in time to the inputs that caused the overload.<sup>4</sup>

Besides a hysteresis delay, system robustness requires specification of a series of checks on the temporal history of mode exit and resumption activities to avoid constant ping-ponging.

#### 15.4.5 Output Specification Completeness

As with trigger events, the complete specification of the behavior of an output event requires both its value and its time.

- *Safety-critical outputs should be checked for reasonableness and for hazardous values and timing.*

Checking to make sure that output values are legal or reasonable is straightforward and helpful in detecting software or other errors. In general, this should always be done for safety-critical outputs and may be desirable for other outputs. Hazardous values can be determined by a top-down hazard analysis that traces system hazards to the software, as described previously.

There is no limit to the complexity of timing specifications for outputs, but, at the least, specification of bounds and minimum and maximum time between outputs is required, as it is for inputs. In addition, there are some special requirements for the specification of the outputs: environmental capacity, data age, and latency.

#### Environmental Capacity Considerations

The rate at which the sensors produce data and send it to the computer is the concern in input capacity. Output capacity, on the other hand, defines the rate at

<sup>4</sup>Hysteresis intervals are also useful for specifying conditions other than timing that cause transitions between states, especially transitions to non-normal processing modes.

which the actuators can accept and react to data produced by the computer. If the sensors can generate inputs at a faster rate than the output environments can “absorb” or process outputs, an output overload might occur.

- For the largest interval in which both input and output loads are assumed and specified, the absorption rate of the output environment must equal or exceed the input arrival rate.

Output load limitations may be required because of physical limitations in the actuators (such as a limit on the number of adjustments a valve can make per second), constraints on process behavior (excessive wear on actuators might increase maintenance costs), or safety considerations (such as a restriction on how often a catalyst can be safely added to a chemical process).

Differences in input and output capacity result in the need to handle three cases:

1. The input and output rates are both within limits, and the “normal” response can be generated.
2. The input rate is within limits, but the output rate will be exceeded if a normally timed output is produced, in which case some sort of special action is required.
3. The input rate is excessive, in which case some abnormal response is necessary (graceful degradation).

When input and output capacities differ, there must be multiple periods for which discrete load assumptions are specified. For example, the output capacity might be 10 per second but only 40 per minute, while the input sensor might have a peak rate of 12 per second but a sustained rate of only 36 per minute.

- Contingency action must be specified when the output absorption rate limit will be exceeded.

Over the short term, the program can buffer or shield the output environment from excessive outputs. Over the long term, however, the program might never catch up unless, for the largest interval in which both input and output capacities are assumed and specified, the absorption rate of the output environment equals or exceeds the input arrival rate. Contingency action must be specified for cases where these assumptions do not hold.

- Update timing requirements or other solutions to potential overload problems, such as operator event queues, need to be specified.

When the human-machine interface is synchronous—that is, each computer response is matched to a human action—the operator cannot be overloaded, and he or she is never in doubt about which response pertains to which action. Even when the interaction is asynchronous, operator overload may not be a problem. In some displays, such as an air traffic controller’s situation display, much of the data can be added, deleted, or changed in parallel with other human-machine interface activities without interfering with operator performance. In this case, the

operator monitors the display for patterns and relationships and determines what is significant and what constitutes an event requiring operator attention.

In other asynchronous interactions, however, the human-machine interface may need to make operators explicitly aware of events rather than merely highlight potentially interesting data on a parallel display. Examples of such events include alarms and orders or requests from other operators. This type of asynchronous interaction can result in operator overload, but putting load limits on the outputs may not be practical. A general solution to the discrete event overload problem is an *event bucket*—generally, one or more queues of event data waiting for operator review and acknowledgment. The information defining the event may be inserted into the event queue and a standard signal used to signify that an event has been detected and queued. A particular operator position may have several predefined and operator-defined events that can be added to its queues.

- Automatic update and deletion requirements for information in the human-computer interface must be specified.

Events placed in queues may be negated by subsequent events. The requirements specification should include the conditions under which such entries may be automatically updated or deleted from a queue. Some entries should be deleted only upon explicit operator request; however, workload may be such that the entries must be queued until the operator can acknowledge them. For example, when an air traffic control operator asks for the count of aircraft whose velocity exceeds a certain speed, the response may be queued and should not disappear until the operator acknowledges receipt.

Some queued events may become irrelevant to the operator, such as information about a warning to an air traffic controller that an aircraft is too close to the ground or to ground-based hazards such as tall antennas (called a minimum safe altitude warning or MSAW). The warning itself may be shown on the situation display, but additional information that cannot be displayed may be put into a queue. If the portion of the queue that contains the MSAW-related information is not currently visible to the operator, it may be removed from the queue automatically when the MSAW is removed from the situation display. If that portion of the queue is currently visible, the queued information should not be removed. Operators generally find it distressing when information disappears while they are looking at it or while they are temporarily glancing away.

There could be safety implications as well. Suppose that there are MSAWs for two separate aircraft, but the queue display can accommodate only one event at a time. The operator might glance back at the display, not realizing that the first event has been removed and replaced by the second. The operator would then read the recommended course for the second aircraft and transmit it to the first aircraft, not realizing that the event data he or she is reading is not the same data seen a second or two before.

- *The required disposition for obsolete queue events must include specification of what to do when the event is currently being displayed and when it is not.*

In general, obsolete event data currently being displayed cannot be automatically deleted or replaced. It may be modified to show obsolescence and removed when the operator indicates to do so or when the overall display is modified in such a way that the obsolete event display becomes invisible (for example, the queue is advanced and the obsolete information is scrolled off the display).

## Data Age

Another important aspect of the specification of output timing involves data obsolescence. In practical terms, few, if any, input values are valid forever. Even if nothing else happens and the entire program is idle, the mere passage of time renders much data of dubious validity eventually. Although the computer is idle, the real world in which the computer is embedded (the process the computer is controlling) is unlikely to be. Control decisions must be based on data from the current state of the system, not on obsolete information.

- *All inputs used in specifying output events must be properly limited in the time they can be used (data age). Output commands that may not be able to be executed immediately must be limited in the time they are valid.*

Data obsolescence considerations require that all input and output events be properly bounded in time: The input is only valid to trigger an output  $O$  if it occurred within a preceding duration of time  $D$ . As an example of the possible implementation implications of such a requirement, MARS, a distributed fault-tolerant system for real-time applications, includes a validity time for every message in the system after which the message is discarded [165].

Frola and Miller [88] describe an accident related to the omission of a data age factor. A computer issued a CLOSE WEAPONS BAY DOOR command on a B-1A aircraft at a time when a mechanical inhibit had been put on the door. The CLOSE command was generated when someone in the cockpit pushed the close door switch on the control panel during a test. The command was not executed (because of the mechanical inhibit), but remained active. Several hours later, when the maintenance was completed and the inhibit removed, the door unexpectedly closed. The situation had never been considered in the requirements definition phase; it was fixed by putting a time limit on all output commands.

The information used in response to queries from operators may also become obsolete before the operator can receive it. The requirements specification needs to state if a query response sitting in the operator's queue should be automatically updated as the situation changes or flagged as possibly obsolete.

- *Incomplete hazardous action sequences (transactions) should have a finite time specified after which the software should be required to cancel the sequence automatically and inform the operator.*

Data age requirements also apply to human-computer interface action sequences. Some transactions require multiple actions, for example, a FIRE command that is followed by a CONFIRM MISSILE LAUNCH request from the computer and then a CONFIRM action from the operator. Once the FIRE command has been issued, some limit should be imposed on how long it remains active (before it is automatically canceled) without confirmation from the operator. Such a time limit may be important if the incomplete control sequence places the system in a higher risk state: Once such a sequence is started, it may take fewer actions or failures to create a hazard, and thus the exposure should be minimized or at least controlled.

- *Revocation of a partially completed action sequence may require (1) specification of multiple times and conditions under which varying automatic cancellation or postponement actions are taken without operator confirmation and (2) specification of operator warnings to be issued in the event of such revocation.*

In some cases, the partially completed sequence should not be discarded without a warning to the operator. In other cases, a partially completed complex transaction should be set aside for subsequent, manual reactivation that is simpler than complete reinitialization. The "safing" sequence and the time periods allowed may themselves vary with the current state. On combat aircraft, for example, weapon selection or activation actions that are a prerequisite for weapon launch should not be automatically revoked easily. On the one hand, when pilots are busy in combat, they should not be further burdened with alarms notifying them that their preliminary weapon selection will be revoked automatically in  $x$  seconds unless overridden. On the other hand, partial selection and activation states should not be allowed to continue indefinitely. A compromise is to let the times vary as a function of conditions detectable by the computer. If the operator is clearly present and engaged in combat activities, the automatic revocation sequence might be postponed indefinitely until conditions change. A *wheels down and engine idle or off* condition might be the basis for a much shorter delay.

## Latency

Since a computer is not arbitrarily fast, there is a time interval during which the receipt of new information cannot change an output even though it arrives prior to the actual output action. The duration of this latency interval is influenced by both the hardware and the software design. An executive or operating system that permits the use of interrupts to signal data arrival may have a shorter latency interval than one that uses periodic polling, but underlying hardware constraints prevent the latency from being eliminated completely. Thus, the latency interval can be made quite small, but it can never be reduced to zero.

The acceptable length of the latency interval is determined by the process that the software is controlling. In chemical process control, a relatively long latency period might be acceptable, while an aircraft may require a much shorter

one. The choice of operating system, interrupt logic, scheduling priority, and system design parameters will be influenced by the latency requirements. Also, behavioral analysis of the requirements to determine consistency with process functional requirements and constraints may not be correct unless the value of this behavioral parameter is known and specified for the software. Therefore, the requirements specification must include the allowable latency factor.

- *A latency factor must be included when an output is triggered by an interval of time without a specified input and the upper bound on the interval is not a simple, observable event.*

Triggering an output on an interval of time without a specified event occurring always requires the specification of a latency factor between the end of the interval and the occurrence of the output. Where the upper bound on the interval is a simple, observable event, latency is not an issue. However, where the intent is to signal the nonoccurrence of an input after some other event, a latency specification is required.

- *Contingency action may need to be specified to handle events that occur within the latency period.*

Additional requirements may need to be specified to handle the case where an event is observed within the latency period. For example, if an action is taken based on the assumption that some input never arrived and if it is later discovered that the input actually did arrive but too late to affect the output, it may then be necessary to take corrective action.

- *A latency factor must be specified for changeable human-computer interface data displays used for critical decision making. Appropriate contingency action must be specified for data affecting the display that arrives within the latency period.*

Latency considerations also affect specification of the human-computer interface. Whenever a data display changes just prior to an operator basing a critical decision on it, the computer may need to query the operator as to whether the change was noted before action selection. The display might involve, for example, showing a set of operator options, including a recommended option and several indications of poor ones. If the arrival of asynchronous data results in a change to the recommended action, then whether the operator had sufficient opportunity to observe that change will affect the required human-computer interface behavior. As another example, an operator decision to fire a missile at a target that has just had its displayed threat value reduced (but not completely eliminated) may warrant extra interaction between the program and the operator.

- *A hysteresis delay action must be specified for human-computer interface data to allow time for meaningful human interpretation. Requirements may also be needed that state what to do if data should have been changed during the hysteresis period.*

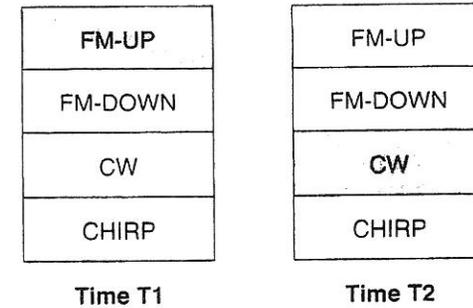


FIGURE 15.2

Two consecutive snapshots of an operation action menu with the recommended action highlighted. The recommendation must be constant long enough for meaningful human interpretation. Requirements are also needed to deal with latency problems when the recommended action changes.

Variable data, such as a computer-recommended operator action, must be constant long enough for meaningful human interpretation, which leads to a requirement for a hysteresis delay (Figure 15.2). An additional requirement will then be needed to cope with situations where the action is selected after the occurrence of an event that should have changed the displayed data but did not because it occurred before the expiration of the hysteresis delay from the previous change.

#### 15.4.6 Output to Trigger Event Relationships

Some criteria for analyzing requirements specifications relate not to input or output specifications alone but to the relationship between them. Although, in general, the relationship depends on the control function being specified, basic process control concepts can be used to generate criteria that apply to all process control systems, such as feedback and stability requirements.

Responsiveness and spontaneity deal with the actual behavior of the controlled process and how it reacts (or does not react) to output produced by the controller. In particular, does a given output cause the process to change, and if so, is that change detectable by means of some input? Basic process control models include feedback to provide information about expected responses to changes in the manipulated variables and information about state changes caused by disturbances in the process.

- *Basic feedback loops, as defined by the process control function, must be included in the software requirements. That is, there should be an input that the software can use to detect the effect of any output on the process. The*

*requirements must include appropriate checks on these inputs in order to detect internal or external failures or errors.*

Feedback is a basic property of almost all process control systems: If feedback information is not used by the software, the requirements specification is probably deficient. Basic feedback loops need to be included in the software requirements, while missing feedback loops provide clues as to incompleteness in the specification.

As an example, an accident occurred when a steel plant furnace was returned to production after being shut down for repair [16]. A power supply had burned out in a digital thermometer during power-up so that the thermometer continually registered a low constant temperature. The controller, knowing it was a cold start, ordered 100 percent power to the gas tubes. The furnace should have reached operating temperature within one hour, but the computer failed to check (and thus detect) that the thermometer inputs were not increasing as they should have been. After four hours, the furnace had burned itself out, and major repairs were required.

This situation could easily have been avoided if information about the characteristics of the process had been used to predict and check for the expected behavior of the system. In this case, the only information needed to avoid the accident was that the temperature should increase if the burners are on.

If the process does not respond to an output as expected and within a given time period, there is presumably something wrong and the software should be required to act accordingly—perhaps by trying a different output, by alerting a human operator, or, at the least, by logging the abnormality for future, off-line analysis.

Ideally, process control systems should be designed such that the effects of every output affecting a manipulated variable in the system can be detected by some input provided by the feedback loop. The situation is not always that simple, however. Disturbances interfering with the process can cause changes that are not initiated by the computer or can inhibit changes that the computer has commanded.

- *Every output to which a detectable input is expected must have associated with it: (1) a requirement to handle the normal response and (2) requirements to handle a response that is missing, too late, too early, or has an unexpected value.*

Every output to which a detectable response is expected within a given time period induces at least two requirements: The “normal” response requirement and a requirement to deal with a failure of the process to produce the expected response. The failure could involve the response having an erroneous or unreasonable value, the response arriving at the wrong time, or the expected response might be missing entirely.

- *Spontaneous receipt of a nonspontaneous input must be detected and responded to as an abnormal condition.*

If the environment responds *too* quickly, coincidence rather than appropriate stimulus-response behavior may be responsible. Most processes do not react *instantaneously*, but only after a delay (time lag). Thus, the specification of a latency factor is required. A value-based handshake protocol can be used to eliminate the need for the latency factor: Some field of the input *I* identifies it as a *unique response* to some specific output *O*.

Some inputs are spontaneous—they may be triggered by environmental factors not necessarily caused by some prior output. However, an input that is supposed to be nonspontaneous (it is only supposed to arrive in response to some prior system output) induces yet another requirement to respond to a presumably *erroneous* (spontaneous) input.

- *Stability requirements must be specified when the process is potentially unstable.*

In addition to feedback requirements, stability requirements, such as a phase margin of at least 45 degrees and a gain margin of at least 3 decibels, may need to be specified for one or more operating states. The stability requirements apply to the process-control function, which is described by a control law or a transfer function relating output to input [23].

### 15.4.7 Specification of Transitions Between States

Requirements analysis may involve examining not only the triggers and outputs associated with each state and the relationship between them, but also the paths between states. These paths are uniquely defined by the sequence of trigger events along the path. Transitions between modes are particularly hazardous and susceptible to incomplete specification, and they should be carefully checked.

#### Reachability

- *All specified states must be reachable from the initial state.*

Informally, a state is said to be *reachable* from another state if there is a path from the first to the second. In most systems, all states must be reachable from the initial state. If a state is unreachable, there are two possibilities: (1) either the state has no function and can be eliminated from the specification, or (2) the state should be reachable and the requirements document is incorrect and must be modified accordingly.

Most state-based models include techniques for reachability analysis. In complex systems, complete reachability analysis is often impractical, but it may be possible in some cases to devise algorithms that reduce the necessary state space search by focusing on a few properties. The backward-reachability hazard analysis techniques for state machine models described in Chapter 14 are examples of algorithms that limit the amount of the reachability graph that must be

generated to get enough information to eliminate hazardous states from the requirements specification.

### Recurrent Behavior

Most process control software is cyclic—it is not designed to terminate under normal operation. Its purpose is to control and monitor a physical environment; the nature of the application usually calls for it to repeat one single task continuously, to alternate between a finite set of distinct tasks, or to repeat a sequence of tasks while in a given mode. Most systems, however, include some states with noncyclic behavior such as temporary or permanent shutdown states or those where the software changes to a different operating mode.

- *Desired recurrent behavior must be part of at least one cycle. Required sequences of events must be implemented in and limited by the specified transitions.*

The specification should be analyzed to verify that desired behavior is repeatable. To be repeatable, the behavior must be part of at least one cycle, but in many cases checking this behavior alone will not be sufficient; more complex sequences of events may need to be identified. An output to turn on a piece of equipment, for example, may be inappropriate unless the last output turned the equipment off. Consider an output to start a piece of equipment. The equipment may need to be started more than once, but it could be damaged if two START commands are issued without an intermediate STOP command. To prevent this hazard, every cycle that includes a START also has to include a STOP.

- *States should not inhibit the production of later required outputs.*

An *inhibiting state* for an output is a state from which the output cannot be generated. If every state from which the output can be generated is unreachable from an inhibiting state, then the output cannot be generated again once the inhibiting state is reached. Whether or not this condition represents an incompleteness depends upon the application.

### Reversibility

In a process control system, a command issued to an actuator often can be canceled or reversed by some other command or combination of commands. This capability is referred to as *reversibility*.

- *Output commands should usually be reversible.*

Outputs will usually require reversing commands. Therefore, outputs should be reviewed and classified as to their reversibility. For an ON command to be reversible, the state in which the canceling OFF command is issued must be reachable from the state in which the ON command was issued. For example, an alert condition to an operator (such as a below-minimum-safe-altitude warning to an

air traffic controller) should be reversible when the condition no longer holds (the aircraft is now at a safe altitude).

- *If  $x$  is to be reversible by  $y$ , there must be a path between the state where  $x$  is issued and a state where  $y$  is issued.*

There will usually be several different classes of the reversing outputs. The appropriate reversing output, for example, may depend on whether the controller has acknowledged the receipt of the original alert, is in the process of reviewing the alert, or has taken positive action to ameliorate the alert condition. The human-computer interface in particular is full of complex classes of reversible phenomena [135].

### Preemption

When the same physical resource, such as a data entry device or display, must be used in distinct multistep actions at the human-computer interface, requirements will be needed to deal with preemption logic. In addition, some actions may have to be prohibited until others are completed. An action to recompute estimated time of arrival might be prohibited until an in-progress, manual navigational update is completed or explicitly canceled.

- *Preemption requirements must be specified for any multistep transactions in conjunction with all other possible control activations.*

In general, there are three possible system responses to an operator action from a parallel-entry source prior to completion of a transaction initiated by some previous control activation: (1) normal processing in parallel with the uncompleted transaction, (2) refusal to accept the new action, and (3) preemption of the partially completed transaction.

If preemption is possible, then the attempted activation of a multistep sequence requiring the use of a resource already involved in another incomplete transaction provides the following three choices:

1. The new request could completely cancel the previous, incomplete transaction, clearing or replacing any displays associated with it.
2. The new request could preempt the shared resources, but the displayed state could be preserved and restored upon completion of the new transaction.
3. The operator could be prompted and required to indicate the disposition of the incomplete transaction, in which case there will in general be four alternatives:
  - a. Cancel the incomplete transaction and start the newly requested one.
  - b. Complete the old transaction and then proceed automatically with the new request.
  - c. Cancel the new request and continue with the old, incomplete transaction.
  - d. Defer but do not cancel the old, incomplete transaction.

If any transactions are deferred and restored, obsolete information must be identified, as discussed previously.

### Path Robustness

For most safety-critical, process-control software, there are concerns beyond pure reachability: Even if a state fulfills all reachability requirements, there is still the question of the *robustness* of the path, or paths, affecting a particular state.

Consider an output that has the possible values of UP and DOWN. Suppose that every possible path from a state where an UP command is issued to any state where a DOWN command is issued includes the arrival of input *I*. Then if the computer's ability to receive *I* is ever lost (perhaps because of sensor failure), there are circumstances under which it will not be able to issue a DOWN command. Thus, the loss of the ability to receive *I* can be said to be a *soft failure mode*, since it *could* inhibit the software from providing an output with the value DOWN.

If the receipt of input *I* occurs in every path expression from *all* states that produce UP commands to *all* states that produce DOWN commands, the loss of the ability to receive *I* is now said to be a *hard failure mode*, since it *will* inhibit the software from producing a DOWN command.

- *Soft and hard failure modes should be eliminated for all hazard-reducing outputs. Hazard-increasing outputs should have both soft and hard failure modes.*

The more failure modes the requirements state machine specification has, whether soft or hard, the less robust with respect to external disturbances will be the software that is correctly built to that specification. Robustness, in this case, may conflict with safety. A fail-safe system should have no soft failure modes, much less hard ones, on paths between dangerous states and safe states. At the same time, hard failure modes are desirable on the paths from safe to hazardous (but unavoidable) states. An unsafe state, where a hazardous output such as a command to launch a weapon can be produced, should have at least one, and possibly several, hard failure modes for the production of the output command: No input received from the proper authority, no weapons launch.

- *Multiple paths should be provided for state changes that maintain or enhance safety. Multiple inputs or triggers should be required for paths from safe to hazardous states.*

In general, operators should be provided with multiple logical ways to issue the commands needed to maintain the safety of the system so that a single hardware failure cannot prevent the operator from taking action to avoid a hazard. On the other hand, multiple interlocks and checks should be associated with potentially hazardous human actions—such as a requirement for two independent inputs or triggers before a potentially hazardous command is executed by the computer.

## 15.5 Constraint Analysis

In addition to satisfying general completeness criteria, the requirements must also be shown to include the identified, system-specific safety requirements and to be consistent with the identified software system safety constraints.

- *Transitions must satisfy software system safety requirements and constraints.*

In a system hazard analysis, hazards are traced to the software–system interface. Such hazards involve specific software behavior expressed in terms of the value and timing of outputs (or lack of outputs). In general, software-related hazards involve

- *Failing to perform a required function: The function is never executed or no answer is produced.*
- *Performing an unintended (unrequired) function, getting the wrong answer, issuing the wrong control instruction, or doing the right thing but under inappropriate conditions (such as activating an actuator inadvertently, too early or too late, or failing to cease an operation at a prescribed time).*
- *Performing functions at the wrong time or in the wrong order (such as failing to ensure that two things happen at the same time, at different times, or in a particular order).*
- *Failing to recognize a hazardous condition requiring corrective action.*
- *Producing the wrong response to a hazardous condition.*

Constraint analysis on the software requirements specification includes a reachability analysis to determine whether the software, as specified, could reach the identified hazardous states.

- *Reachable hazardous states should be eliminated or, if that is not possible (they are needed to achieve the goals of the system), their frequency and duration reduced.*

It is not always possible to enforce a requirement that the software cannot reach hazardous states—sometimes a hazardous state is unavoidable. But this possibility should be known so that steps can be taken to minimize the risk associated with the hazard, such as minimizing the exposure or adding system safeguards to protect the system against such states.

The type of analysis required to guarantee consistency between the software requirements specification and the safety constraints depends upon the type of constraints involved. The presence of constraints can potentially affect most of the criteria that have been described in this chapter. Some types of constraints can be ensured by the criteria already described; others require additional analysis. For example, basic reachability analysis can verify that only safe states are reachable.

Basic reachability analysis may need to be extended, however, to consider additional constraints on the sequence of events. To illustrate, consider a simple

control system to move the control rods in a reactor up and down. The output actions to move the rods may be properly reachable and the paths robust. In addition, a constraint may require that a rod not be allowed to move within 30 seconds of its previous movement. To guarantee this constraint, all transitions where a MOVE ROD1 command can be issued must first be identified. Path analysis can then be used to find the sequences of events that will make the software issue two consecutive MOVE ROD1 commands. By showing that all possible paths described by these sequences will take at least 30 seconds to traverse, the constraint is guaranteed to be satisfied. If all the criteria described in this chapter for complete specification of timing requirements are satisfied, this analysis should be theoretically possible for a state-machine specification.

More generally, the specification may be checked for a general *safety policy* that is defined for the particular system. This process is very similar to checking that a specification satisfies a particular security policy [180]. The following is an example of a general safety policy for which the specification could be checked:

1. *There must be no paths to unplanned hazardous states.*

The computer never initiates a control action (output) that will move the process from a safe to an unplanned hazardous state.

2. *Every hazardous state must have a path to a safe state. All paths from the hazardous state must lead to safe states. Time in the hazardous state must be minimized, and contingency action may be necessary to reduce risk while in the hazardous state.*

If the system gets into a hazardous state (by an unplanned transition that is not initiated by the computer such as component failures, human error, or environmental stress), then the computer controller will transform the hazardous state into a safe state (every path from a hazardous state leads to a safe state). The time in the hazardous state will be minimized to reduce the likelihood of an accident.

There may be several possible safe states, depending on the type of hazard or on conditions in the environment. For example, the action to be taken if there is a failure in a flight-control system may depend on whether the aircraft is in level flight or is landing.

3. *If a safe state cannot be reached from a hazardous state, all paths from that state must lead to a minimum risk state. At least one such path must exist.*

If a system gets into a hazardous state and there is no possible path to a safe state, then the computer will transform the state into one with the minimum risk possible given the hazard and the environmental conditions, and it will do so such that the system is in a hazardous state for the minimum amount of time possible.

It may not be possible to build a completely safe system—that is, to avoid all hazardous states or to get from every hazardous state to a safe state. In that event, the system must be redesigned or abandoned, or some risk must be accepted. This risk can be reduced by providing procedures to minimize the probability of the

hazardous state leading to an accident or to minimize the effects of an accident. For example, activation of a carbon dioxide firefighting system in what may be an occupied space may kill any occupants, but it may be necessary to prevent the loss of an entire ship. Such difficult decisions obviously must be considered and specified carefully.

## 15.6 Checking the Specification Against the Criteria

The actual procedures that can be used to analyze a particular requirements specification will depend on the form of that specification. The criteria for completeness of states, inputs and outputs, and the relationship between inputs and outputs are easily checked for any type of specification. Criteria for the transitions between states will be checkable to a greater or lesser extent depending on the formality of the specification, the size of the specification, and the availability of software tools to help with the checking.

Heimdahl has automated the checking of the robustness and nondeterminism criteria (Sections 15.4.4.1 and 15.4.4.2) for specifications written in RSML, and validated his tools on an avionics collision avoidance system [116, 117]. Additional tools are being created for safety analysis of RSML requirements specifications.

Some criteria can be enforced simply by using a specification language that incorporates enforcement in its syntax. For example, a language that requires specifying value and time intervals for all inputs and data age limits on all outputs will not require additional analysis. Even if the language syntax does not require specifying a particular characteristic of the inputs or outputs, a syntax that makes omissions immediately apparent will be helpful in locating them.

On many projects, requirements are not complete before software development begins. In addition, changes are often made as the design of the other parts of the system becomes more detailed and problems are found that necessitate changes in the desired software behavior. It is therefore unlikely that the analysis will be completed before software design begins. To avoid costly redesign and recoding, the requirements specification and analysis should be as complete as possible as early as possible. Realistically, however, some of the analysis may need to be put off or redone as the software and system development proceeds.