1- Recovery Blocks (RcB)

The basic RcB scheme is one of the two original

. In addition to being a design diverse technique, the RcB is further categorized as a dynamic technique. In dynamic software fault tolerance techniques, the selection of a variant result to forward as the adjudicated output is made during program execution based on the result of the acceptance test (AT). This will be clarified when we examine the operation of the technique below. The hardware fault-tolerant architecture related to the RcB scheme is stand-by sparing or passive dynamic redundancy.

RcB uses an AT and backward recovery to accomplish fault tolerance. We know that most program functions can be performed in more than one way, using different algorithms and designs. These differently implemented function variants have varying degrees of efficiency in terms of memory management and utilization, execution time, reliability, and other criteria. RcB incorporates these variants such that the most efficient module is located first in the series, and is termed the primary alternate or primary try block. The less efficient variant(s) are placed serially after the primary try block and are referred to as (secondary) alternates or alternate try blocks. Thus, the resulting rank of the variants reflects the graceful degradation in the performance of the variants.

2- Recovery Block Operation

The basic RcB scheme consists of an executive, an acceptance test, and primary and alternate try blocks (variants). Many implementations of RcB, especially for real-time applications, include a watchdog timer (WDT). The executive orchestrates the operation of the RcB technique, which has the general syntax:

```
ensure              Acceptance Test
by                  Primary Alternate
else by             Alternate 2
else by             Alternate 3
...
else by             Alternate n
else failure exception
```

The RcB syntax above states that the technique will first attempt to ensure the AT (e.g., pass a test on the acceptability of a result of an alternate) by using the primary alternate (or try block). If the primary algorithms result does not pass the AT, then n-1 alternates will be attempted until an alternates results pass the AT. If no alternates are successful, an error occurs. Figure 4.1 illustrates the structure and operation of the basic RcB technique with a WDT. We examine several scenarios to describe RcB operation:

• Failure-free operation;

• Exception or time-out in primary alternate execution;

• Primary's results are on time, but fail AT; successful alternate execution;

• All alternates exhausted without success.

In examining these scenarios, we use the following abbreviations: A Alternate (when there is a single alternate other than P); Ai Alternate i (when there are multiple alternates other than P); Ai OK Global flag indicating whether or not Ai has failed; AOK Global flag indicating whether or not A has failed; AT Acceptance test; P Primary algorithm/alternate/try block; POK Global flag

indicating whether or not P has failed; RcB Recovery block; WDT Watchdog timer.

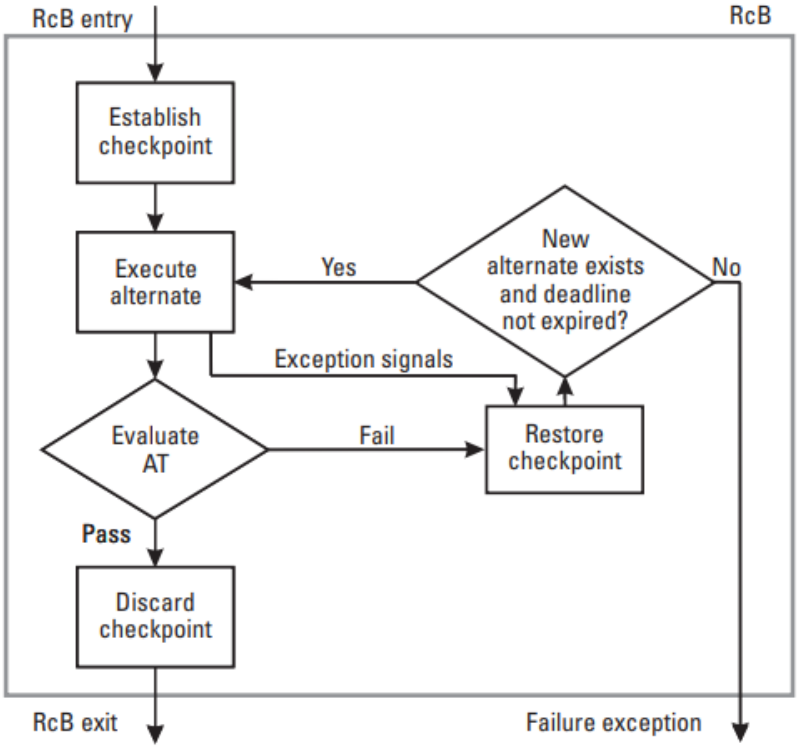

Figure 4.1 Recovery block structure and operation.

3- Augmentations to the Recovery Block Basic Technique

RcB operation continues until acceptable results are produced, there are no new alternates to try, or the deadline expires without an acceptable result. Several augmentations to the basic RcB technique have been suggested. One, of course, is the addition of a WDT, which we examined above. Another augmentation is to use an alternate routine execution counter. This counter is used when the primary fails and execution is switched to an alternate. The counter indicates the number of times to execute the alternate (on this and subsequent entries to the RcB) before reverting to executing the primary. The counter is incremented once the primary fails and prior to each execution of the alternate(s). The benefit of using the

alternate routine execution counter is that it provides the ability to take the primary out of service for repair and/or replacement while the alternate continues the algorithm execution tasks. The basic RcB technique may also be augmented by the use of a more detailed AT comprised of several tests.

One AT is invoked before the primary routine execution and checks the format and parameters of the call. If either of these fails this AT, the alternate is called instead of the primary.

Another AT executed before the call to the primary alternate tests the validity of the input data. If the input data fails this AT, then neither the primary nor secondary alternates are called. In this case, default or backup data may be used or the RcB can be further augmented with an alternate source for input data.

The final AT of this AT set is the one commonly used with RcB—the AT that checks the results of the primary or alternate try blocks.

4- Recovery Block Example

Lets look at an example for the RcB technique. Suppose we have software that incrementally sorts a list of integers (see Figure 4.2). Also suppose the input is the list (8, 7, 13, −4, 17, 44). If the software operates correctly, we expect the result to be (−4, 7, 8, 13, 17, 44). However, unbeknownst to us, our sort algorithm implementation produces incorrect results if one or more of the inputs is negative. How can we protect our system against faults arising from this error? A first thought may well be to test the software more thoroughly or to use other fault prevention techniques. This would be appropriate, especially with such a simple problem to solve. Of course, sound software engineering practice must be used in developing fault-tolerant software as it is (or should be) with

non-fault-tolerant software. However, most software algorithms used in real applications are not likely to be as simple and well documented as the sort problem. The more complex the system—the algorithm, data, operating environment, and so on—the less able we are (with current technology) to thoroughly test the program or prove its correctness. Another thought likely to arise with regards to the example used is to question why it is so simple and unrepresentative of the actual software one might incorporate with software fault tolerance techniques. We use a simple algorithm to illustrate this and other techniques so that the illustration of the techniques operation is not obscured by the complexities of the algorithm. Now, on with the example. Figure 4.3 illustrates an approach to using recovery blocks with the sort problem. Note the additional components needed for RcB technique implementation an executive that handles checkpointing and orchestrating the technique,

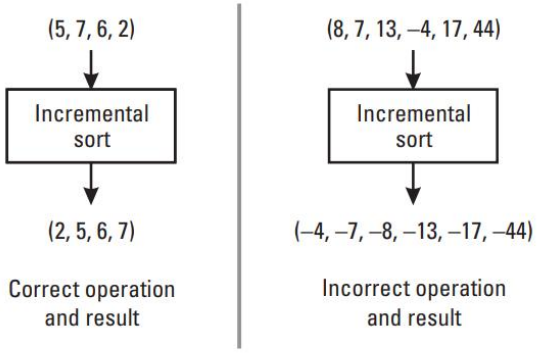

Figure 4.2 Example of original sort algorithm implementation.

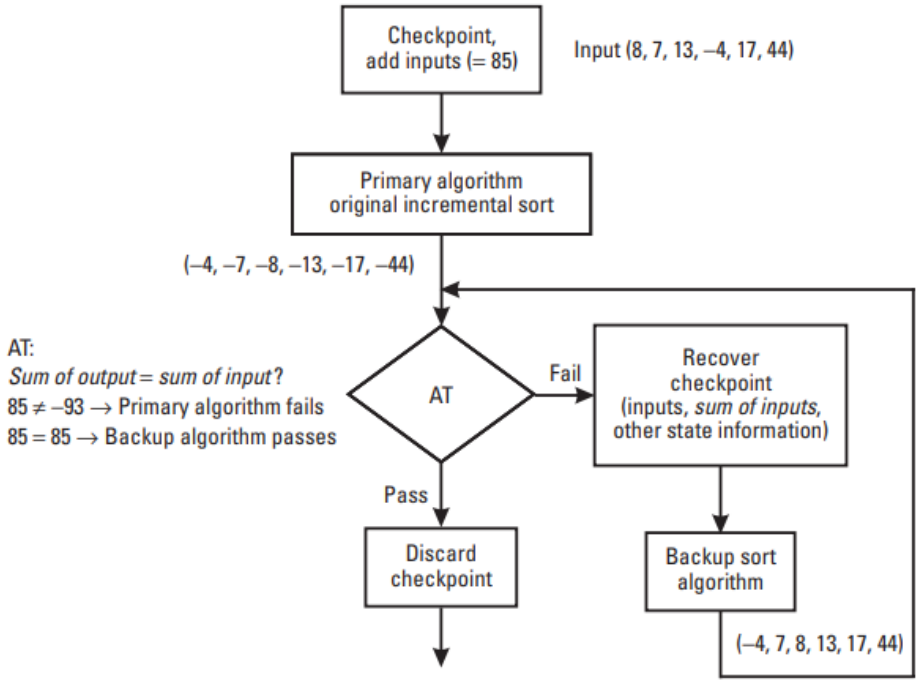

Figure 4.3 Example of recovery block implementation.

an alternate or backup sort algorithm, and an AT. In this example, no WDT is used. Also note the design of the AT. How can we determine if a sort algorithm is correct without performing an entire other sort to check it against? In this example, we test if the sum of the inputs is equal to the sum of the outputs. This test will not determine if the items in the input list were sorted correctly, but it will indicate if something blatantly incorrect has resulted from the sort. To use this AT, the executive sums the inputs upon entry to the RcB and sums the results of the alternates (primary or secondary) upon arrival at the AT. Now, lets step through the example:

• Prior to entering the RcB, a status module checks the primary module status. Since the primary has not yet failed, the RcBs operation proceeds. • Upon entry to the RcB, the executive performs the following: a checkpoint is established, calls to the primary and backup (alternate)

routines are formatted, and the inputs (8, 7, 13, −4, 17, 44) are summed. The sum of input = 85.

• The primary sort algorithm is executed. The results are (−4, −7, −8, −13, −17, −44).

• The results are submitted to the AT. The AT sums the items in the result vector. The sum of output = −93. The AT tests if the sum of input equals the sum of output. 85 ≠ −93, so the results of the primary sort algorithm fail the AT.

• Control returns to the executive. The executive sets the flag indicating failure of the primary algorithm and restores the checkpoint.

• The backup sort algorithm is executed using the same input vector. The results are (−4, 7, 8, 13, 17, 44).

• The results of the backup sort algorithm are submitted to the AT. The AT sums the items in the result vector. The sum of output = 85. The AT tests if the sum of input equals the sum of output. 85 = 85, so the results of the backup sort algorithm pass the AT. • Control returns to the executive.

• The executive discards the checkpoint, the results are passed outside the RcB, and the RcB is exited.

routines are formatted, and the inputs (8, 7, 13, −4, 17, 44) are summed. The sum of input = 85.

• The primary sort algorithm is executed. The results are (−4, −7, −8, −13, −17, −44).

• The results are submitted to the AT. The AT sums the items in the result vector. The sum of output = −93. The AT tests if the sum of input equals the sum of output. 85 ≠ −93, so the results of the primary sort algorithm fail the AT.

• Control returns to the executive. The executive sets the flag indicating failure of the primary algorithm and restores the checkpoint.

• The backup sort algorithm is executed using the same input vector. The results are (−4, 7, 8, 13, 17, 44).

• The results of the backup sort algorithm are submitted to the AT. The AT sums the items in the result vector. The sum of output = 85. The AT tests if the sum of input equals the sum of output. 85 = 85, so the results of the backup sort algorithm pass the AT.

• Control returns to the executive.

• The executive discards the checkpoint, the results are passed outside the RcB, and the RcB is exited.

5- Recovery Block Issues and Discussion

This section presents the advantages, disadvantages, and issues related to the RcB technique. In general, software fault tolerance techniques provide protection against errors in translating requirements and functionality into code, but do not provide explicit protection against errors in specifying requirements. This is true for all of the techniques described in this course.

Being a design diverse, backward recovery technique, the RcB subsumes design diversity's and backward recovery's advantages and disadvantages, too. These are discussed. While designing software fault tolerance into a system, many

considerations have to be taken into account. These are discussed. Issues related to several software fault tolerance techniques (e.g., similar errors, coincident failures, overhead, cost, redundancy, and so on) and the programming practices used to implement the techniques are described also. Issues related to implementing acceptance tests are discussed. There are a few issues to note specifically for the RcB technique. The technique runs in a sequential (uniprocessor) environment. When the results of the primary try block or alternate pass the AT, the overhead incurred (beyond that of running the primary alone, as in non-fault-tolerant software) includes setting the checkpoint and executing the AT. If, however, the primary's results fail the AT, then the time overhead also includes the time for recovering the checkpointed information, execution times for each alternate run until one passes the AT (or until all fail the AT), and run-time of the AT each time an alternate's results are checked. It is assumed that most of the time, the primary's results will pass the AT, so the expected time overhead is that of setting the checkpoint and executing the AT. This is little beyond the primary's execution time (unless an unusually large amount of information is being checkpointed). In the worst case, however, the recovery blocks execution time is the sum of all the module executions mentioned above (in the case where the primary's results fail the AT). This wide variation in execution time exposes the technique to timing errors that are likely unacceptable in real-time applications. One solution to the overhead problem is the DRB in which the modules and AT are executed in parallel. In RcB operation, when executing non-primary alternates, the service that the module is to provide is interrupted during the recovery. This interruption may be unacceptable in applications that require high availability. One advantage of the technique is that it is naturally applicable to software modules, as opposed to whole systems. It is natural to apply RcB to specific critical modules or processes in the system without incurring the cost and complexity of supporting fault

tolerance for an entire system. For the technique to be effective, a highly effective AT is required. Success of the RcB technique depends on all the error detection mechanisms used (including exception handling and others), not just on the AT. A simple, effective AT can be difficult to develop and depends heavily on the specification. If an error is not detected by the AT (or by the other error detection mechanisms), then that error is passed along to the module that receives the technique' s results. This undetected error does not, of course, trigger any recovery mechanisms and may cause system failure. A potential problem with the RcB technique is the domino effect, in which cascaded rollbacks can push all processes back to their beginnings. This occurs if recovery and communication operations are not coordinated, especially in the case of nested recovery blocks. Section 4.1.3.1 below presents the conversation scheme, which was proposed as a means to avoid the domino effect. To implement the RcB technique, the developer can use the programming techniques (such as assertions, checkpointing, atomic actions, idealized components). Also needed for implementation and further examination of the technique is information on the underlying architecture and performance. Table 4.1 lists several issues related to the RcB technique, indicates whether or not they are an advantage or disadvantage (if applicable), and points to where in the book the reader may find additional information. The indication that an issue in Table 4.1 can be a positive or negative (+/−) influence on the technique or on its effectiveness further indicates that the issue may be a disadvantage in general (e.g., cost is higher than nonfault-tolerant software) but an advantage in relation to another technique (e.g., the average cost of a two-try block RcB implementation is less than that for NVP with n = 3). In these cases, the reader is referred to the discussion of the issue (versus duplicating the discussion here).

Table 4.1
Recovery Block Issue Summary

Issue	Advantage (+)/ Disadvantage (–)	Where Discussed
Provides protection against errors in translating requirements and functionality into code (true for software fault tolerance techniques in general)	+	Chapter 1
Does not provide explicit protection against errors in specifying requirements (true for software fault tolerance techniques in general)	–	Chapter 1
General backward recovery advantages	+	Section 1.4.1
General backward recovery disadvantages	–	Section 1.4.1
General design diversity advantages	+	Section 2.2
General design diversity disadvantages	–	Section 2.2
Similar errors or common residual design errors (RcB is affected to a lesser degree than forward recovery, voting techniques.)	–	Section 3.1.1
Coincident and correlated failures	–	Section 3.1.1
Domino effect	–	Section 3.1.3
Overhead for tolerating a single fault	+/–	Section 3.1.4
Cost (Table 3.3)	+/–	Section 3.1.4
Space and time redundancy	+/–	Section 3.1.4
Dependability studies	+/–	Section 4.1.3.3
Acceptance tests and discussions related to specific types of ATs	+/–	Section 7.2

6- Conversation Scheme

The conversation scheme is a means to avoid the domino effect. It is essentially a concurrent extension of the RcB technique in which a group of processes can interact safely. Each process entering a conversation is check pointed. To prevent spreading data that has not been validated, processes within the conversation can communicate only with other processes that are in the conversation. When all processes have completed their operations, a global AT occurs. If it fails, all processes roll back and execute their next alternates. If the AT is satisfied, all processes discard their checkpoints and exit synchronously from the conversation. The conversation scheme spans two or more processes and creates a boundary that process interactions may not cross. The boundary of a conversation consists of a recovery line, a test line, and two side walls defining the membership of the conversation. The basic program-structuring rules of the conversation scheme are the following.

1. A conversation defines a recovery line as a line that processes in roll back cannot cross.

2. Processes enter a conversation asynchronously.

3. A conversation contains one or more interacting processes whose goal is to obtain the same or similar computational results.

4. A conversation test line is an acceptability criterion for the results of interacting processes, acting as a single global acceptance test a conversation acceptance test.

5. Processes cooperate in error detection, regardless of where the error originates.

6. The processes participating in the conversation must neither obtain information from, nor leak information to, a process not participating in the conversation. The conversations side walls define the process membership relationship.

Limitations of the conversation scheme include the following:

• It burdens the designer with the responsibility of establishing the coordinated recovery points.

• Process desertion may occur when there are timing deadlines to meet. Desertion is the failure of a process to start a conversation or arrive at the AT when it is expected by other processes.

• The conversation scheme may involve a long time delay for synchronization.

• The goal requirement of the group of processes is combined with those of the individual participants in the AT.

• Process desertion may occur when there are timing deadlines to meet. Desertion is the failure of a process to start a conversation or arrive at the AT when it is expected by other processes.

• The conversation scheme may involve a long time delay for synchronization.

• The goal requirement of the group of processes is combined with those of the individual participants in the AT

6.1 Architecture

Structuring is required if we are to handle system complexity, especially when fault tolerance is involved. This includes defining the organization of software modules onto the hardware elements on which they run. The recovery block approach is typically uniprocessor implemented, with the executive and all components residing on a single hardware unit. All communications between the software components is done through function calls or method invocations in this architecture.

6.2 Performance

There have been numerous investigations into the performance of software fault tolerance techniques in general (e.g., in the effectiveness of software diversity) and the dependability of specific techniques themselves. Table 4.2 provides a briefly noted list of references for these dependability investigations. This list is by no means exhaustive, however, it does provide a good sampling of the types of analyses that have been performed and ample background for analyzing software fault tolerance dependability. The reader is encouraged to examine the references for details on assumptions made by the researchers, experiment design, and results interpretation.