Programming Techniques

There are several programming or implementation techniques used by several software fault tolerance techniques such as: **Assertion, check pointing and Atomic actions.** Assertions can be used by any software fault tolerance technique, and by non-fault tolerant software. Check pointing is typically used by techniques that employ backward recovery. Atomic actions can also be used in non-faulttolerant software, but are presented here primarily in the context of software fault tolerance in concurrent systems.

1- Assertions

Assertions are a fairly common means of program validation and error detection. As early as 1975, Randell presented executable assertions as central to the design of fault-tolerant programs. An executable assertion is a statement that checks whether a certain condition holds among various program variables, and, if that condition does not hold, takes some action.

In essence, they check the current program state to determine if it is corrupt by testing for out-of-range variable values, the relationships between variables and inputs, and known corrupted states. These assertion conditions are derived from the specification, and the assertion can be made arbitrarily stringent in its checking. Assertions may be set up to only produce a warning upon detection of a corrupt state or they may take or initiate corrective action. For example, upon the detection of a corrupt state, the assertion may halt program execution or attempt to recover from the corrupt state. What the assertion does upon firing (detecting a corrupt state) is application-dependent. Assertions can be used as part of a reasonableness-checking AT such as a range bounds AT.

Traditional assertions produce warnings when the condition being checked is not met. They do not typically attempt recovery. Recovery assertions, on the other

hand, are forward recovery mechanisms that attempt to replace the current corrupt state with a correct state. As with check pointing, the entire state can be replaced or only specific variables, depending on the system constraints and overhead (e.g., time and memory) involved in the saving and restoration of the variables or states values. Assertions can also be used to reset variables periodically (i.e., without necessarily testing for a corrupt state) in, for example, safety-critical real-time systems to limit the propagation of corrupt data values. Some programming languages provide special constructs for executable assertions. However, executable assertions are essentially Boolean functions that evaluate to TRUE when the condition holds, and FALSE otherwise. Using a generic pseudo code language, we can present the simplest form of an executable assertion as:

if not assertion then action

where assertion is a Boolean expression and action is a method or procedure.

The most general form of an assertion must refer to the current state and to a previous state.

There are three reasons for which an intermediate state should be chosen over the initial state in an executable assertion:

a. **Modularity**: We can think of the assertion a as checking a local program segment b by referring to the state of the program before execution of b and after execution of b. The program segment b and its assertion-checking facilities then form a modular unit that is context independent it does not depend on where it is in the program.

b. **Time parsimony**: Block b can be arbitrarily short, and the function it computes arbitrarily simple. Hence the assertion that checks it can be arbitrarily easy to compute and arbitrarily time efficient. By contrast, referring to s0 means that, at

milestone m, we check expected past functional properties of program P at m, whose complexity we do not choose.

c. Space parsimony: Block b can be arbitrarily short, and the variables it affects arbitrarily few. Hence the memory space required to save the variables modified by block b is arbitrarily small. By contrast, referring to s0 means that sufficient memory space must be set aside to save all of s0, whose size we do not choose.

The initial state or intermediate state, the block b to be checked, and the statement that saves all or part of the previous state comprises an **elementary asserted block (EAB)**. The general form of an EAB is

```
$ = s;
b;      // modifies s, but not s$
if not a (s$, s) then action;
```

In the above EAB, the assignment statement $= s means saving states in s$. The expression a(s$, s) is the assertion. As stated earlier, we may only want to save some variables, such as those that are going to be modified by b and/or those that are involved in the expression of assertion a. for example: Suppose s □S is an integer. Also, suppose that program block b determines the square of s, that is, b = (s = s * s). The following three simple assertions illustrate different assertions we can use with the defined program block,

```
b. s$ = s;
b; if not (s = s$2 ) then action;
s$ = s; b;
if not (s$ > 1 ⇒ s > s$) then action; s$ = s; b;
if not (s > 0) then action;
```

In typical practice, b would be an intricate block of code that is difficult to analyze and a(s$, s) would be a simple assertion. When an error is detected in the current

state, action should be taken to notify the designer (so that corrective action fault removal can be taken) and a procedure is invoked to perform damage assessment and take appropriate recovery action. An assertion, sc, can be used to detect strict correctness (or freedom from errors) in the program. The following pseudo code sample illustrates the pattern for such an assertion.

```
perform_error_management
{ if not sc(s$ = s) then
{ // erroneous state
produce_warning(UI_or_errorfile, detected_error);
// UI - User Interface
perform_damage_assessment_and_recovery; }
 }
```

2- Checkpointing

Checkpointing is used in (typically backward) error recovery, which we recall restores a previously saved state of the system when a failure is detected. Recovery points, points in time during the process execution at which the system state is saved, are established. The recovery point is discarded when the process result is accepted, and it is restored when a failure is detected. Checkpoints are one of several mechanisms used to establish these recovery points. Other mechanisms include the audit trail and the recovery cache:

• Checkpoint: saves a complete copy of the state when a recovery point is established.

• Recovery cache: saves only the original state of the objects whose values have changed after the latest recovery point.

• Audit trail: records all the changes made to the process state.

The generic term checkpoint will be used and will include all three mechanisms, unless otherwise stated. The information saved by checkpoints includes the values

of variables in the process, its environment, control information, register values, and so on. The information should be saved on stable storage so that even if the node fails, the saved checkpoint information will be safe. For single node, single process systems, checkpointing and recovery are simpler than in systems with multiple communicating processes on multiple nodes. For single process checkpointing, there are different strategies for setting the checkpoints. Some strategies use randomly selected points, some maintain a specified time interval between checkpoints, and others set a checkpoint after a certain number of successful transactions have been completed. For example, examine the location of checkpoints based on the principle of information reduction. There is a trade-off between the frequency and amount of information checkpointed, and various performance measures (e.g., information integrity, system availability, program correctness, and expected execution time). For example, the code size between checkpoints can be a determining factor in the effectiveness and cost of a fault tolerance strategy. If the intermediate results are checked after small pieces of code have been executed, then there is lower error latency, but also a higher execution time overhead. In addition, decision points limit design diversity, since an increase in the frequency or number of decisions requires the agreement of variants at a higher level of detail. However, a large modular decomposition (and thus larger code segments between decisions or checkpoints) ensures higher variant independence and lower execution overheads. Larger code segments between checkpoints and decisions may result in a cursory acceptance test that is incapable of localizing the errors that occur. Models of the various approaches to checkpointing have been compared and their effects on system performance examined. There are generally two approaches to multiprocess backward recovery **asynchronous and synchronous checkpointing**. In asynchronous checkpointing, the checkpointing by the various nodes in the system is not

coordinated. However, sufficient information is maintained in the system so that when rollback and recovery is required, the system can be rolled back to a consistent state. The cost of asynchronous checkpointing is lower than synchronous checkpointing, but the risk of unbounded rollback (the domino effect) remains. Many checkpoints for a given process may need to be saved because during rollback, a remote (in time) state can be restored. Asynchronous checkpointing is simpler than synchronous checkpointing, but can be useful only where expected failures are rare and there is limited communication between the system processes.. In synchronous checkpointing (or distributed checkpointing), establishing checkpoints is coordinated so that the set of checkpoints as a whole comprise a consistent system state. This limits the amount of rollback required, but the cost of establishing the checkpoints is higher than in asynchronous checkpointing because of the coordination required. Also, only a few checkpoints of a process need to be saved at a time.

3- Atomic Actions

Atomic actions are used for error recovery, primarily in concurrent systems (and are widely used in transaction systems. Critical concurrent systems must be structured so that their complex asynchronous activities, such as those related to fault tolerance, can be achieved. One way to approach this requirement is to use atomic actions, which have been shown to increase the quality and reusability of code and to reduce code complexity significantly. The activity of a group of components constitutes an atomic action if no information flows between that group and the rest of the system for the duration of the activity. An atomic action is an action that is:

• **Indivisible**: Either all the steps in the atomic action complete or none of them does, that is, the all-or-nothing property.

• Serializable: All computation steps that are not in the atomic action either precede or succeed all the steps in the atomic action.
• Recoverable: The external effects of all the steps in the atomic action either occur or not; that is, either the entire action completes or no steps are completed.

The property of atomicity guarantees that if an action successfully executes, its results and the changes it made on shared data become visible for subsequent actions. On the other hand, if a failure occurs inside of an action, the failure is detected and the action returns without changes on shared data. This enables easy damage containment and error handling, since the fault, error propagation, and error recovery all occur within a single atomic action. Therefore, the fault and associated recovery activities will not affect other system activities. If the activity of a system can be decomposed into atomic actions, fault tolerance measures can be constructed for each of the atomic actions independently.

The indivisibility property of atomic actions may seem to imply that an atomic action itself cannot have structure. However, an action can be composed of other actions that are not necessarily primitive operations. These are called nested atomic actions [81]. Since a procedure (or operation or method) may invoke other procedures, which may invoke other procedures, and so on, we can naturally get a nested atomic action. The structure of a nested action atomic cannot be visible from outside the nested atomic action. A nested atomic action consists of subactions (not visible from outside), which are seen as atomic actions to the other subactions of the same action. That is, within the nested atomic action, each subaction is an atomic action, and hence the structure of a subaction is not visible to another subaction. This enables a safe method of supporting concurrency within an action.