Issues and problems of software fault tolerance

1. Similar errors
2. Consistent comparison problem (CCP)
3. Domino effect
4. Overheads
http://www.cse.psu.edu/~kandemir/cse598d-2004.html
Similar errors and lack of diversity

1. Lack of independence of programming effort will assure that residual software design faults will lead to an erroneous decision by causing similar errors to occur at the same decision point in two or more versions.

2. NVP’s success depends on whether the residual software faults in each version are distinguishable.

Why do errors need to be distinguishable?
Tolerance

1. Use of floating-point arithmetic (FPA) in general computing produces a result that is accurate only within a certain range.

2. The use of design diversity can also produce individual variant results that differ within a certain range.

3. A *tolerance* is a variance allowed by a decision algorithm.
A taxonomy of variant results

Variant results

- Outside tolerance
  - Dissimilar results
- Within tolerance
  - Similar results
Dissimilar results

Correct

Multiple correct results (MCR)

Probable decision mechanism failure

[Undetected success]

Incorrect

Multiple incorrect results

E.g., finding the roots of an $n^{\text{th}}$ order equation

Independent failure

[Detected failure]
Similar results

- Correct
  - Correct results
    - [Success]

- Incorrect
  - Similar errors (IAW)
    - Same input case
  - Coincident failure
    - Occurs more frequently than by chance
  - Correlated or dependent failures
    - [Undetectable failures]
Example of similar results

A

Variant 1

Variant 2

Variant 3

$r_1$  $r_2$

Tolerance

$r_3$

$r^* = r_1$ or $r_2$ (or combination of $r_1$ and $r_2$)
Consistent comparison problem (CCP)

1. The CCP occurs as a result of finite-precision arithmetic and different paths taken by variants based on specification-required computations.

2. The difficulty is that if \( N \) versions operate independently, then whenever the specification requires that they perform a comparison, it is not possible to guarantee that the versions arrive at the same decision.

3. These isolated comparisons can lead to output values that are completely different from each other.
Finite-precision arithmetic (FPA) function, $A$

- If $A(x) > C_1$?
  - True
  - FPA function, $B$
    - If $B(A(x)) > C_2$?
      - True
      - FPA function, $D$
        - $D(B(A(x)))$
      - False
      - $E(B(A(x)))$
    - False
    - $C(A(x))$
- False

$X$

$A(x)$

$B(A(x))$

$E(B(A(x)))$

$C(A(x))$
The problem lies in the specification

1. CCP does not result from software faults
2. Specifications do not (and most probably cannot) describe required results down to the bit level for every computation and every input
3. This level of detail is necessary if the specification is to describe a function in which one, and only one, output is valid for every input
4. Without communication between the variants, there is no solution to the CCP (proven!)
Impractical avoidance techniques for CCP

1. Approximate comparison: regards two numbers as equal if they differ by less than a tolerance $\delta$
   - The problem arise again with C+ $\delta$

2. Rounding

3. Random selection of a result

4. Exact arithmetic

5. Cross-check points: forces agreement among variants on their floating-point values before any comparisons are made that involve the values
CCP avoidance techniques depend on system history maintenance

System

No history
CCP → Transient effects
Avoidance → Confidence signals

Convergent states
CCP → Temporary discrepancy
Avoidance → Confidence signals

History

Nonconvergent states
CCP → May never reach consensus
“Avoidance” → Revert to backup or fail-safe system
Systems with no history

1. The effects of the CCP in systems with no history are transient

2. An avoidance approach using confidence signals
   - Each variant determines for itself whether the values used in comparisons were close enough to warrant suspicion of inconsistency
   - It signals the voter if inconsistency is possible
   - The voter can vote using the results from the variants that indicated confidence in their results

3. This approach requires extensive modifications to the system structure — large overhead
Systems with convergent states

1 Inconsistent comparisons may cause a temporary discrepancy among variant states in systems with convergent states

1 A confidence signal approach may also be used with these systems
   - Each variant must maintain confidence information as part of its state
   - If part of the system’s state information is based on a comparison that may be inconsistent, then the variant must indicate a “No confidence” signal to the voter for its results
   - The no confidence state for this variant remains until the system state is reevaluated
System with nonconvergent states

1. Once the variants in a system with nonconvergent states acquire different states, inconsistency may persist indefinitely.
2. In the worst case, the NVP system may never again reach consensus on a vote.
3. No simple avoidance technique that can be used for systems with nonconvergent states.
4. Only practical approach is to revert to a backup or fail-safe system.
Domino effect

1. The *domino effect* refers to the successive rolling back of communicating processes when a failure is detected in anyone of the processes.

Domino effect

P1

R1 → C1 → R2
R4 → C2 → R3
R6 → C3 → R5

P2

C4 → R5
C5 → R7

Error

Time

T₀ → T₁
System consistent states

1. The avoidance of the uncontrolled rolling back evidenced by the domino effect is achieved if system consistent states, which serve as recovery points, can be established.

2. A consistent state allows the system to achieve an error-free state that leads to no contradictions and conflicts within the system and its interfaces.

3. Consistent states can be determined statically or dynamically.

4. Restrictions on the communication system to support consistency:
   - Communication delay is negligible and can be considered zero.
   - Communication maintains a partial order of data transfer. All messages sent between a particular pair of processes are received at the destination in the order they were sent.
Static versus dynamic approaches

1 Static
   – A recovery line is set at compile time comprising a set of recovery points
   – Conversation scheme: processes establish a recovery point when they enter a conversation, and all processes leave the conversation together

1 Dynamic
   – Uses stored information about communication and recovery points to set up a recovery line only after an error occurs
   – Programmer-transparent coordination scheme
Overheads

1. Software fault tolerance incurs overhead in terms of space, time, and cost.
2. All the software fault tolerance techniques require diversity in some form and this diversity in turn requires additional space or time, or both.
### Structural overhead for tolerating one fault

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Structural Overhead</th>
<th>Mechanisms (Layers Supporting the Diversified Software Layer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RcB</td>
<td>One variant and one AT</td>
<td>Recovery cache</td>
</tr>
<tr>
<td>NSCP</td>
<td>Error detection by ATs One variant and two ATs</td>
<td>Result switching</td>
</tr>
<tr>
<td></td>
<td>Error detection by comparison</td>
<td>Three variants</td>
</tr>
<tr>
<td></td>
<td>NVP</td>
<td>Two variants</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Voters</td>
</tr>
</tbody>
</table>
## Operational time overhead for tolerating one fault

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Operational Time Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Systematic</td>
</tr>
<tr>
<td></td>
<td>Decider</td>
</tr>
<tr>
<td></td>
<td>Variants Execution</td>
</tr>
<tr>
<td></td>
<td>On Error Occurrence</td>
</tr>
<tr>
<td>RcB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Acceptance test execution</td>
</tr>
<tr>
<td></td>
<td>Accesses to recovery cache</td>
</tr>
<tr>
<td></td>
<td>One variant and AT execution</td>
</tr>
<tr>
<td>NSCP</td>
<td>Error detection by ATs</td>
</tr>
<tr>
<td></td>
<td>Input data consistency and variants execution synchronization</td>
</tr>
<tr>
<td></td>
<td>Possible result switching</td>
</tr>
<tr>
<td>NVP</td>
<td>Two variants</td>
</tr>
<tr>
<td></td>
<td>Vote execution</td>
</tr>
<tr>
<td></td>
<td>Usually negligible</td>
</tr>
</tbody>
</table>
Space and time redundancy in software fault tolerance

Recovery blocks

Conditionally sequential execution

E.g., Sequential NVP

Time (sequential execution)

E.g., NVP t/(n-1)-VP NSCP

Possible region of dynamic space-time trade-offs

Software variant

Space (Parallel execution)
Cost-effectiveness of software

1. Is it better to devote the extra effort to develop the additional variants for diverse software or to devote that effort to the verification and validation of one “really good” piece of software?

This is the question!
Cost of diversity - observations

1. The development and maintenance costs for three-variant diversity can be twice that of a single development and less than double for two-variant diversity.

2. Not all parts of the software’s functionality are critical (Ericsson’s railway interlocking system).

3. Software fault tolerance may also be less expensive than alternative means of assurance.

4. Safety assessment is a big issue for aircraft and nuclear power industries.
Programming techniques for software fault tolerance

1 **Assertions** — can be used by any software fault tolerance technique

1 **Checkpointing** — is typically used by techniques that employ backward recovery

1 **Atomic actions** — are primarily used in the context of software fault tolerance in concurrent systems
Assertions

1. 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.

2. Assertion conditions are derived from the specification, and can be made arbitrarily stringent in its checking.

3. Executable assertions are essentially Boolean functions that evaluate to TRUE when the condition holds, and FALSE otherwise.

\[
\text{if not assertion then action}
\]
Referring to states

1. The most general form of an assertion must refer to the current state and to a previous state. An elementary asserted block (EAB):

   \[
   s' = s; \\
   b; \quad // \text{modifies } s, \text{ but not } s' \\
   \text{if not } a(s', s) \text{ then } \text{action};
   \]

1. Primary choices for the previous state
   - Initial state, \( s_0 \)
   - An intermediate between \( s_0 \) and current state that was reached along the path the program execution has taken
Reasons for which an intermediate state should be chosen over the initial state in an assertion

1. **Modularity**: The program segment and its assertion-checking facilities form a modular unit that is context independent.

2. **Time parsimony**: Block can be arbitrarily short, and the function it computes arbitrarily simple. Hence, the assertion that checks it can be arbitrarily easy to compute and efficient.

3. **Space parsimony**: Block can be arbitrarily short, and the variables it affects arbitrarily few. Hence, the memory space required to save the variables modified by it is small.
Example

1. \(s\) is an integer

1. Program block \(b\): \(b = (s = s \times s)\)

1. Three different assertions

   \[
   \begin{align*}
   s' &= s; \\
   b; \\
   \text{if not} \ (s=s'^2) \ \text{then} \ \text{action};&
   \\
   s' &= s; \\
   b; \\
   \text{if not} \ (s>1 \rightarrow s>s') \ \text{then} \ \text{action};&
   \\
   s' &= s; \\
   b; \\
   \text{if not} \ (s>0) \ \text{then} \ \text{action};
   \end{align*}
   \]
An assertion used to detect strict correctness

```
perform_error_management {
  if not sc(s=s’) then {
    // erroneous state
    produce_warning(UI_orerrorfile, detected_error);
    // UI – User Interface
    perform_damage_assessment_and_recovery;
  }
}
```
Recovery points

1 *Recovery points*: points in time during process execution at which the system state is saved
1 They are discarded when the process result is accepted, and are restored when a failure is detected
1 Three mechanisms used to establish recovery points
   - *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
1 Generic term “checkpoint” includes all three mechanisms
The information saved by checkpoints includes:

- Values of variables in the process
- Its environment
- Control information
- Register values
- Other...

The information should be saved on stable storage
Setting checkpoints for a single process

1. Different strategies
   - Randomly selected points
   - Maintain a specified time interval between checkpoints
   - Set checkpoint after a certain number of successful transactions have completed

2. There are tradeoffs between frequency and amount of information checkpointed, and various performance measures (e.g., information integrity, system availability, program correctness, and expected execution time)
Setting checkpoints for multiple processes

1. **Asynchronous checkpointing**: the checkpointing by the various nodes in the system is not coordinated.

2. **Synchronous checkpointing (or distributed checkpointing)**: establishing checkpoints is coordinated so that the set of checkpoints as a whole comprise a consistent system state.
Asynchronous checkpointing

- 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.
- (+) Cost of asynchronous checkpointing is lower than synchronous checkpointing.
- (-) Risk of unbounded rollback remains.
- (+) Simpler.
- (-) Many checkpoints for a given process may need to be saved.
- (-) Can be useful only where expected failures are rare and there is limited communication between system processes.
Synchronous checkpointing

1. (+) Amount of rollback required is limited
2. (-) Cost establishing the checkpoints is higher
   - Typically requires some form of coordination
3. (+) Fewer checkpoints of a process need to be saved at a time
Two approaches for implementing checkpointing

```java
try {
    T oldobject = object;
    alternate(object);
    if (accept(object)) {
        return;
    }
} catch (...) {
    object = oldobject;
    continue;
}
```

```java
try {
    T newobject = alternate(object);
    if (accept(newobject)) {
        object = newobject;
        return;
    }
} catch (...) {
    continue;
}
```
Atomic Actions

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**: 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 an action, the failure is detected and the action returns without changes on shared data.
Nested atomic actions

1. An action can be composed of other actions that are not necessarily primitive operations.
2. The structure of a nested action cannot be visible from outside the nested atomic action.
3. A nested atomic consists of subactions, which are seen as atomic actions to other subactions of the same action.
   - Within the nested atomic action, structure of a subaction is not visible to another subaction.
   - This enables a safe method of supporting concurrency within an action.
Supporting atomicity in single process environment

1. Supporting atomicity in a single process environment is straightforward.
2. Prior to beginning the execution an action, checkpoint the state of the system.
3. If no failure occurs before the completion of this action, then the “all” part of “all-or-nothing” is satisfied.
4. On the other hand, if a failure occurs, then restore the checkpointed state. This effectively satisfies the “nothing” part of the “all-or-nothing” property.
Some efforts to provide support for atomic actions in distributed systems

1. In PACT, atomic actions are used to achieve fully user-transparent fault tolerance with low run-time overhead.
2. Wellings and Burns show how atomic actions can be implemented in Ada 95, and how they can be used to implement atomic actions in object-oriented applications.
3. Avalon/C++ takes advantage of inheritance to implement atomic actions in object-oriented applications.
4. The Arjuna system uses inheritance, in a manner similar to Avalon/C++, and object extensions to implement atomic actions in object-oriented applications.