Software Fault Tolerance

Sequential Fault Tolerance Techniques

Jörg Kienzle
Software Engineering Laboratory
School of Computer Science
McGill University

Overview

- Robust Software (Pullum 2.1)
- Design Diversity
  - Recovery Blocks (Pullum 4.1)
  - Acceptance Tests (Pullum 7.2)
- Data Diversity
  - Retry Blocks (Pullum 5.1)
  - Data Re-expression Algorithms (Pullum 2.3)

Robust Software (1)

- Software that can continue to operate correctly despite the introduction of invalid inputs [IEEE82]
- Invalid inputs are defined in the specification
  - Out of range inputs
  - Inputs of the wrong type
  - Inputs in the wrong format
  - Corrupted inputs (detected using error-detecting codes)
  - Wrong invocation protocol
  - Violation of pre-conditions

Robust Software (2)

- Goal: No degradation of functionality (that does not depend on the invalid input)
- Detect wrong inputs, then
  - Request new input from the source (probably a human operator)
  - Use last acceptable value
  - Use a predefined default value
- Signal input error to the outside
- Means: (interface) exceptions

Design Diversity (Reminder)

- Identical copies (replicates) of software cannot increase reliability in the presence of software design faults
  ⇒ Design diversity:
  Provision of identical services through separate design and implementations
- Components providing identical functionality are called versions, variants, alternatives, modules
- Make versions as diverse and independent as possible
  - Low probability of common-mode failures:
    Variants should fail on disjoint subsets of the input space
  - High reliability: At least one variant should be operational all times

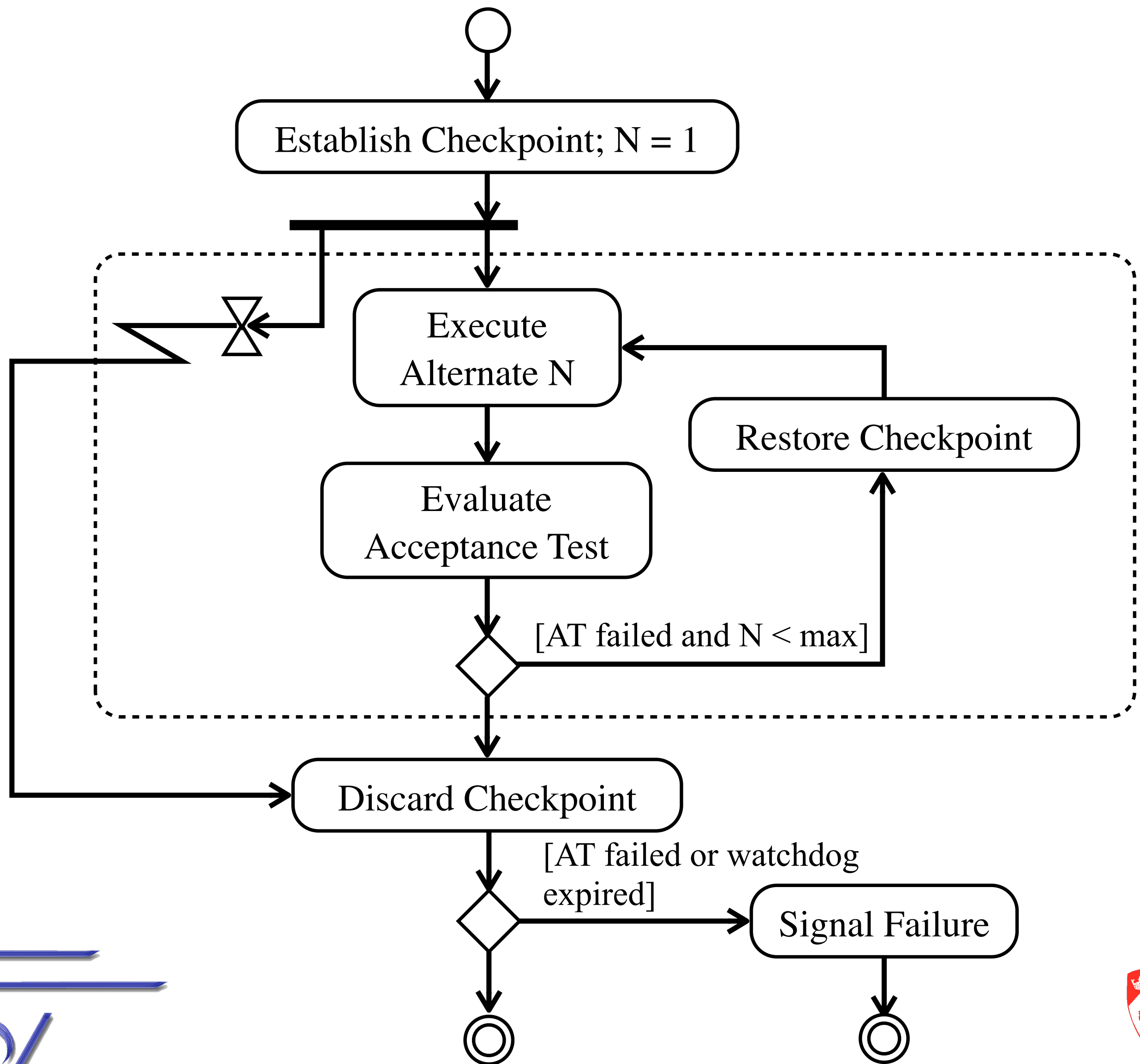
Recovery Blocks (1)

- Introduced in 1974 [Hor74], first implementations by Randell [Ran75]
- Idea: Most program functions can be performed in more than one way
- Different algorithms and design, with varying degrees of efficiency in terms of memory utilization, execution time, reliability, etc…
  - Most efficient variant: primary alternate (or try block)
  - Less efficient: secondary alternate (or try block)

Recovery Blocks (2)

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

Recovery Block Execution



Recovery Blocks (3)

- Based on acceptance test and backward error recovery
- Dynamic technique
  (selection of what output / result is to be used is made during execution based on the result of the acceptance test)
- May include a watchdog to support real-time

Recovery Block Discussion (1)

- Runs in a sequential environment
- Overhead in fail-free mode:
  - Establishing a checkpoint
  - Running the acceptance test
  - Discard the checkpoint
- Additional overhead for every alternate failure:
  - Restoring the checkpoint, executing the alternate, and running the acceptance test again
- Although unlikely, potential overhead is huge
  - Without watchdog not suitable for real-time applications

Recovery Block Discussion (2)

- Can be applied to small, critical software modules
- Watchdog version can detect "infinite loops"
- Requires a highly effective acceptance test
  - Undetected error can cause severe damage
- Communication with the outside can cause domino effect

Acceptance Test (1)

- Basic approach to self-checking software
  - To check post-conditions of operations
- Must verify that the system behavior is acceptable based on an assertion on the anticipated system state
  - Returns true or false
- Used in recovery blocks, consensus recovery block, distributed recovery block, retry block, atomic actions, coordinated atomic actions

Requirements for Acceptance Tests

- Simple
  - Keep run-time overhead reasonable
- Effective
  - Detect anticipated faults
  - Does not incorrectly detect “unfaulty” behavior
- Highly Reliable
  - Does not introduce additional design faults

Acceptance Test Trade-Offs

	Cursory Test	Comprehensive Test
Error Detection Capability	Low	
Design Complexity		High
Design Fault Proneness		High
Development Cost		High
Execution Time		Long
Storage Requirements		High

Acceptance Test (2)

- Test for what a program should do, or for what a program should not do?
  - Testing for what a program should do may require computation of the same magnitude than the main algorithms
  - Possibility of dependence between the acceptance test and the main algorithms
  - Testing for a violation of safety conditions is often simpler

Testing for Satisfaction of Requirements

- Based on the program specification
  - In mathematical operations:
    - Test by applying the *inverse* operation (if it exists)
    - Example: square root
  - Sorting
    - Check that elements are in ascending order
    - Check that the result has the same number of elements
    - Check for the existence of each element in the original sequence
- Test must be independent in order to be effective
- Most effective when carried out on small segments of code [Hec79]

Accounting Tests

- Can handle larger sections of code than satisfaction of requirements tests
- *Checksum*
  - Number of records, sum of all fields
  - Invariants
- *Inventories*
  - Physically measurable (can be automated)
- Suits data-oriented applications with simple mathematical operations (banking systems, …)

Reasonableness Tests

- Based on physical constraints
  - Timing constraints
  - Physical laws
    - Temperature, Speed
    - Continuous rate of change
  - Boundary conditions in application environment
  - Sequencing of object states
- Suits process control / real-time applications
- Straightforward and efficient to implement

Run-time Tests

- Testing for anomalous states in the program
  - Divide-by-zero
  - Overflow / Underflow
  - Undefined operation code
  - Write-protection violation
- Range checks (e.g. Ada)
- Null pointer checks

Design Diversity Cost

- Cost for developing three-variant diversity is about twice that of single development [H88]
  - Cost for requirement specification, test specification and system test execution are not multiplied
  - Not all parts of a system are critical
  - Cost for design, coding and version testing is multiplied
- Recovery Blocks
  2 alternates: average cost 175%
  3 alternates: average cost 237 %
- N-Version Programming
  3 versions: average cost 225 %
  4 versions: average cost 301 % [L35]

Retry Blocks (1)

- Introduced in 1987 [AK87]
- Idea:
  Some algorithms fail on very specific input values (e.g. 0.0), but will succeed / be very efficient on related values
  - First try with original input
  - If attempt fails, re-express input and try again
- Data diverse complement of the recovery block

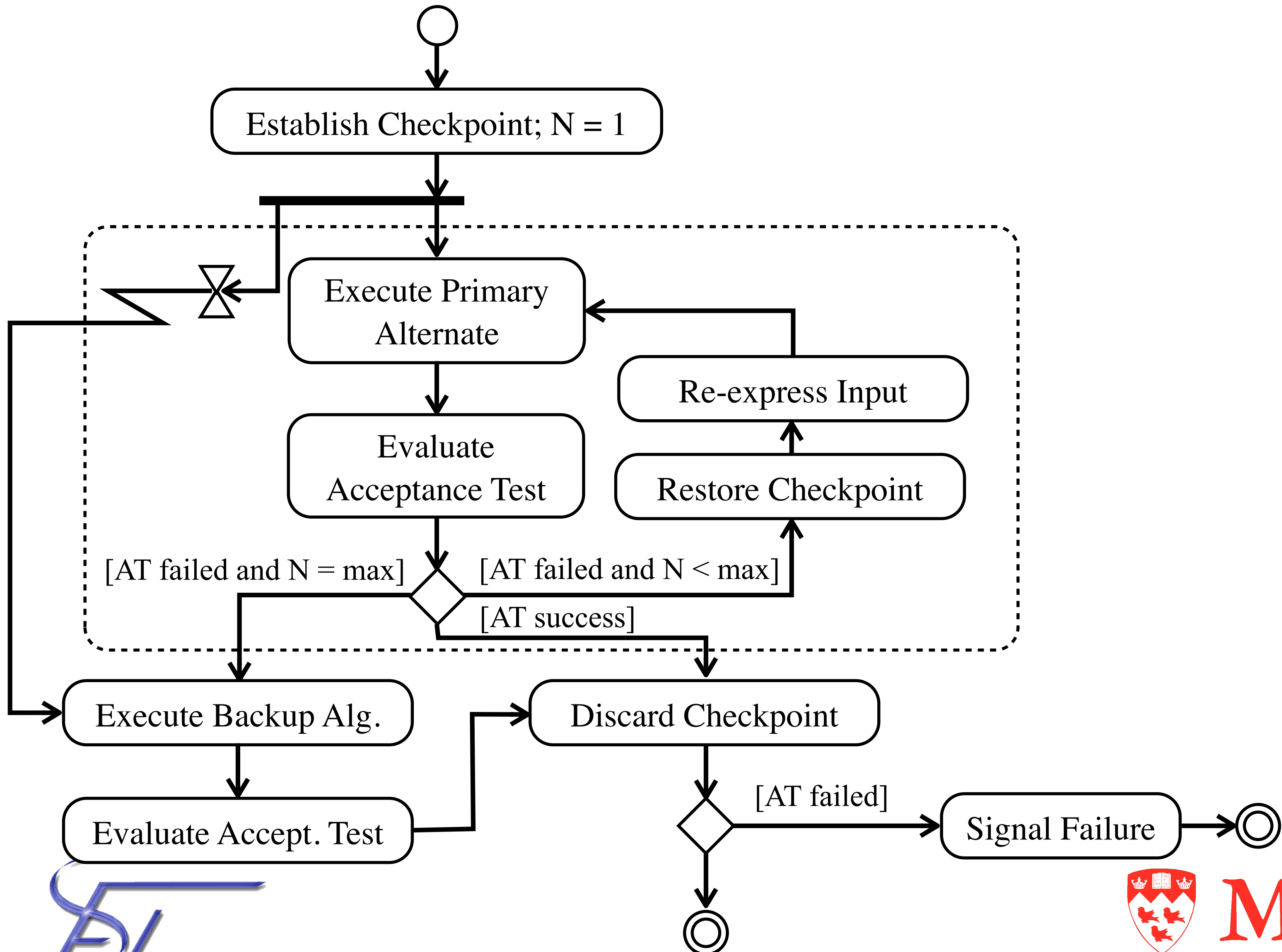
Retry Blocks (2)

```
ensure Acceptance Test
by Primary Algorithm (Original Input)
else by Primary Algorithm (Re-expr. Input)
else by Primary Algorithm (Re-expr. Input)
…
… [deadline expires]
else by Backup Algorithm (Original Input)
else signal failure exception
```

Retry Blocks (3)

- Based on acceptance test and backward error recovery
- Dynamic technique
  (selection of what output / result is to be used is made during execution based on the result of the acceptance test)
- May include a watchdog for handling real-time situations

Retry Block Execution



Retry Block Discussion

- Runs in a sequential environment
- Overhead in fail-free mode:
  - Establishing a checkpoint
  - Run the acceptance test
- Additional overhead in case of failure:
  - For each additional try: Restoring the checkpoint, executing the data re-expression algorithm, running the primary algorithm again, and running the acceptance test again
  - In case of deadline expiration or failure of all primary runs: Restoring the checkpoint, execution of the backup algorithm, running the acceptance test
- Although unlikely, potential overhead is huge
- Without watchdog not suitable for real-time applications

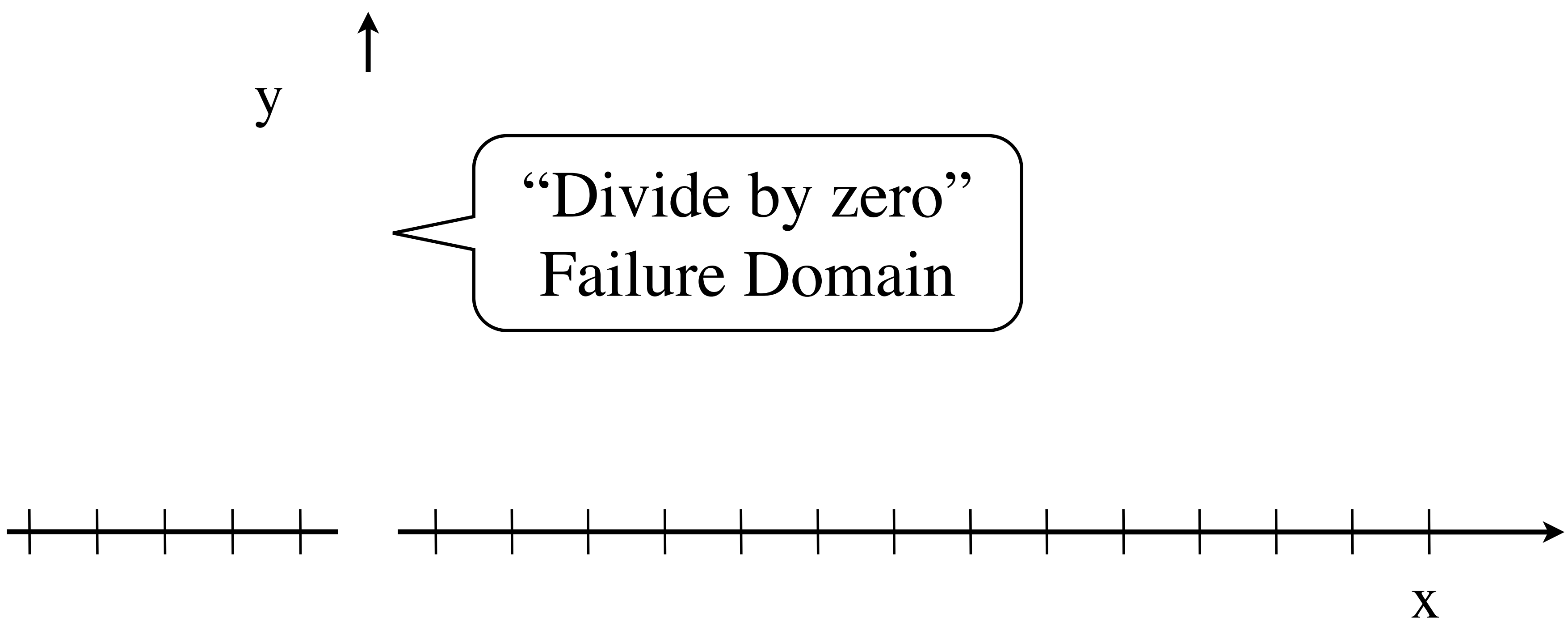
Retry Block Discussion (2)

- Can be applied to small, critical software modules
- Watchdog version can detect “infinite loops”
- Requires a highly effective data re-expression algorithm and acceptance test
  - Undetected error can cause severe damage
- Communication with the outside can cause domino effect

Retry Block Example (1)

- Program calculates f(x,y)
  - The two inputs x and y are measured by sensors with a tolerance of ± 0.02
- Original algorithm should not receive x = 0.0 as an input, or else `Divide_By_Zero` exception is thrown
  - Input can be close to 0.0, but due to lack of precision in the floating point data type, values such as 1e-10 are rounded down to 0.0
- Acceptance test: $f(x,y) \geq 100.0$

Retry Block Example (2)



Retry Block Example (3)

- Calculate f(0.7e-10, 2.2)

  1. Retry block executive establishes a checkpoint
  2. Primary algorithm is executed with ($0.7e^{-10}$, 2.2)
     ⇒ Divide_By_Zero exception
  3. The executive catches the exception, sets a flag indicating the failure of the first run, and restores the checkpoint
  4. The executive re-expresses the inputs by calling the data re-expression algorithm

Retry Block Example (4)

5. The DRA modifies x within x's limits of accuracy: $R(x) = x + 0.0021$
6. The executive calls the primary algorithm with the re-expressed input. Execution returns 123.45
7. The executive submits the result to the acceptance test, which is passed successfully
8. The executive discards the checkpoint and returns the results

McGill

References (1)

- [IEEE92]
  IEEE Standard 729-1982: "IEEE Glossary of Software Engineering Terminology", 1982.
- [Hor74]
  Horning, J. J, et al.: "A Program Strucure for Error Detection and Recovery", in E. Gelenbe and C. Kaiser (eds.), Lecture Notes in Computer Science, Vol. 16, pp. 171-187, Springer, 1974.
- [Ran75]
  Randell, B.: "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 220-232, 1975.
- [Hec79]
  Hecht, H.: "Fault-Tolerant Software", IEEE Transactions on Reliability, Vol. R-28, No. 3, pp.227-232, 1979.
- [Avi84]
  Aviziensis, A.; Kelly, J. P. J.: "Fault Tolerance by Design Diversity: Concepts and Experiments", IEEE Computer, Vol. 17, No. 8, pp. 67-80, 1984.

McGill

References (2)

- [AK87]
  Ammann, P. E.; Knight, J. C.: “Data Diversity: An Approach to Software Fault Tolerance”, Proceedings of FTCS-17, Pittsburgh, PA, pp. 122-126, 1987.
- [Cri89]
  Cristian, F. : “Exception Handling”, in T. Anderson (ed.), Resilient Computing Systems, Vol. 2, Wiley & Sons, 1987.
- [AK88]
  Ammann, P. E.; Knight, J. C.: “Data Diversity: An Approach to Software Fault Tolerance”, IEEE Transactions on Computers, Vol. 37, pp. 122-126, 1988.