Programming for Reliability

- Programming techniques for building reliable software systems.

Objectives

- To describe programming techniques for reliable systems development
- To discuss fault avoidance by error-prone construct minimization
- To describe fault tolerant system architectures
- To show how exception handling constructs may be used to create robust programs and as part of a defensive approach to programming

Topics covered

- Fault avoidance techniques
- Fault tolerance and fault tolerant architectures
- Exception handling and management
- Defensive programming
- Program examples are presented in both Ada and C++

Software reliability

- In general, software customers expect all software to be reliable. However, for non-critical applications, they may be willing to accept some system failures
- Some applications, however, have very high reliability requirements and special programming techniques must be used to achieve this

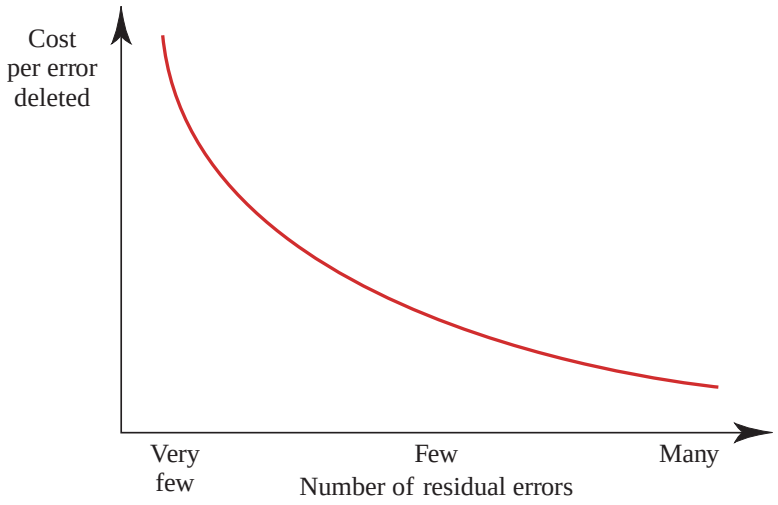
Reliability achievement

- Fault avoidance
  - The software is developed in such a way that it does not contain faults
- Fault detection
  - The development process is organised so that faults in the software are detected and repaired before delivery to the customer
- Fault tolerance
  - The software is designed so that faults in the delivered software do not result in complete system failure

Fault avoidance

- Current methods of software engineering now allow for the production of fault-free software.
- Fault-free software means software which conforms to its specification. It does NOT mean software which will always perform correctly as there may be specification errors.
- The cost of producing fault free software is very high. It is only cost-effective in exceptional situations. May be cheaper to accept software faults

Fault removal costs



Fault-free software development

- Needs a precise (preferably formal) specification.
- Information hiding and encapsulation in software design is essential
- A programming language with strict typing and run-time checking should be used
- Extensive use of reviews at all process stages
- Requires an organizational committment to quality.
- Careful and extensive system testing is still necessary

Ada and C++

- Ada was designed for large-scale software engineering and is a strictly typed language. However, few compilers available for personal computers
- However, C++ is becoming increasingly widely used for development.. Combines the efficiency of a low-level language (C) with object-oriented programming constructs. Better type checking than C but not so good as Ada

Structured programming

- First discussed in the 1970's
- Programming without gotos
- While loops and if statements as the only control statements.
- Top-down design.
- Important because it promoted thought and discussion about programming.

Error-prone constructs

- ⊗ Floating-point numbers
  - Inherently imprecise. The imprecision may lead to invalid comparisons
- ⊗ Pointers
  - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change
- ⊗ Dynamic memory allocation
  - Run-time allocation can cause memory overflow
- ⊗ Parallelism
  - Can result in subtle timing errors because of unforseen interaction between parallel processes

Error-prone constructs

- ⊗ Recursion
  - Errors in recursion can cause memory overflow
- ⊗ Interrupts
  - Interrupts can cause a critical operation to be terminated and make a program difficult to understand. they are comparable to goto statements.
- ⊗ It is NOT suggested that these constructs should always be avoided but they must be used with great care.

Information hiding

- Information should only be exposed to those parts of the program which need to access it. This involves the creation of objects or abstract data types which maintain state and operations on that state
- This avoids faults for three reasons:
  - the probability of accidental corruption of information
  - the information is surrounded by 'firewalls' so that problems are less likely to spread to other parts of the program
  - as all information is localised, the programmer is less likely to make errors and reviewers are more likely to find errors

Data typing

- Each program component should only be allowed access to data which is needed to implement its function
- The representation of a data type should be concealed from users of that type
- Ada, Modula-2 and C++ offer direct support for information hiding
- The type system can be used to enhance program readability by modelling real-world entities directly.

Type declarations

- C++ type declarations
  - typedef enum { red, redamber, amber, green} TrafficLightColour ;
    TrafficLightColour ColourShowing, NextColour ;
- Ada type declarations
  - **type** POSITIVE is INTEGER **range** 1..MAXINT ;
  - **type** OIL_STATUS **is new** BOOLEAN ;
    type DOOR_STATUS **is new** INTEGER ;
    type FUEL_STATUS **is new** BOOLEAN ;

Objects and abstract data types

- Implemented in C++ as objects, in Ada as packages
- The type name is declared within the object or ADT
- Type operations are defined as procedures or functions.
- The type representation is defined in the private part.
- Generic abstract data structures may be parameterised using the type name.

Ada specification of an integer queue

```
package Queue is
        type T is private ;
        procedure Put (IQ : in out T; X: INTEGER);
        procedure Remove (IQ : in out T; X : out INTEGER);
        function Size (IQ : T ) return NATURAL;
private
        type Q_RANGE is range 0..99 ;
        type Q_VEC is array ( Q_RANGE ) of INTEGER ;
        type T is record
                The_queue: Q_VEC ;
                front, back : Q_RANGE ;
        end record;
end Queue;
```


C++ Queue class declaration

```
class Queue {
public:
        Queue () ;
        ~Queue () ;
        void Put ( int x ) ; // adds an item to the queue
        int Remove () ; // this has side effect of changing the queue
        int Size( ) ;  // returns number of elements in the queue
private:
        int front, back ;
        int qvec [100] ;
} ;
```

Generics

- The behaviour of objects and ADTs which are composed of other objects or ADTs is often independent of the type of these included objects
- Generics are a way of writing generalised, parameterised ADTs and objects which may be instantiated later with particular types
- Both Ada and C++ have generic type or class definition facilities

Ada declaration of a generic queue

```
generic
        type ELEM is private ;
        type Q_SIZE is range <> ;
package Queue is
        type T is private ;
        procedure Put (IQ : in out T; X: ELEM );
        procedure Remove (IQ : in out T; X : out ELEM );
        function Size (IQ : in T ) return NATURAL ;
private
        type Q_VEC is array (Q_SIZE) of ELEM ;
        type T is record
                The_queue: Q_VEC ;
                Front : Q_SIZE := Q_SIZE'FIRST ;
                Back: Q_SIZE := Q_SIZE'FIRST ;
        end record;
end Queue;
```

C++ generic queue

```
template
        <class elem>
class Queue {
public:
        Queue ( int size = 100 ) ; // default to queue of size 100 elements
        ~Queue () ;
        void Put ( elem x ) ;
        elem Remove ( ) ; // this has side effect of changing queue
        int Size ( ) ;
private:
        int front, back ;
        elem* qvec ;
} ;
```


Generic instantiation

- Generics are instantiated at compile-time NOT at run-time so type checking is possible
- Ada
  - **type** IQ_SIZE **is range** 0..49 ; **type** LQ_SIZE **is range** 0..199 ;
    package Integer_queue **is new** Queue (ELEM => INTEGER,
    Q_SIZE => IQ_SIZE) ;
    package List_queue **is new** Queue (ELEM => List.T,
    Q_SIZE => LQ_SIZE) ;
- C++
  - //Assume List has been defined elsewhere as a type
    Queue <int> Int_queue (50) ;
    Queue <List> List_queue (200) ;

Fault tolerance

- In critical situations, software systems must be fault tolerant.
- Fault tolerance means that the system can continue in operation in spite of software system failure
- Even if the system has been demonstrated to be fault-free, it must also be fault tolerant as there may be specification errors or the validation may be incorrect

Fault tolerance actions

- Failure detection
  - The system must detect that a failure has occurred.
- Damage assessment
  - The parts of the system state affected by the failure must be detected.
- Fault recovery
  - The system must restore its state to a known safe state.
- Fault repair
  - The system may be modified to prevent recurrence of the fault. As many software faults are transitory, this is often unnecessary.

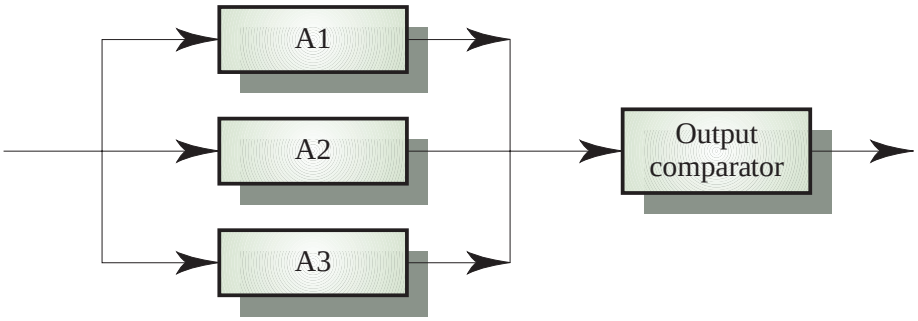
Fault occurrence

- Many software failures are transient and dependent on individual data. Operation may continue by re-starting the system
- If this is impossible, dynamic system re-configuration may be necessary where software components are replaced without stopping the system

Hardware fault tolerance

- Depends on triple-modular redundancy (TMR)
- There are three replicated identical components which receive the same input and whose outputs are compared
- If one output is different, it is ignored and component failure is assumed
- Based on most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure

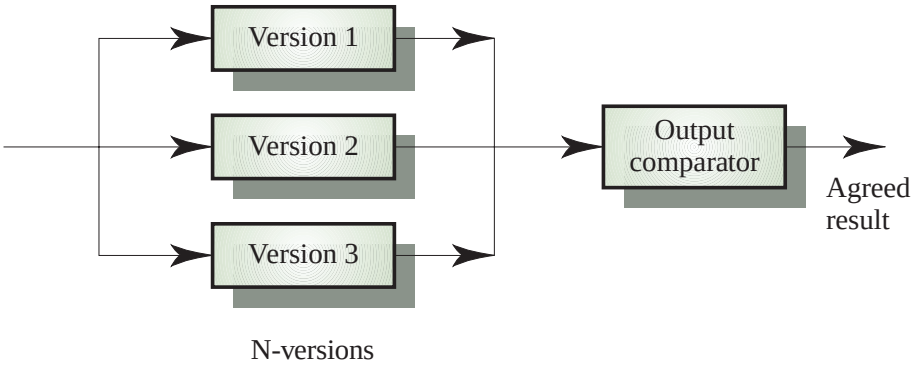
Hardware reliability with TMR



Software analogies

- N-version programming
  - The same specification is implemented in a number of different versions. All versions compute simultaneously and the majority output is selected.
    This is the most commonly used approach e.g. in Airbus 320. However, it does not provide fault tolerance if there are specification errors.
- Recovery blocks
  - Versions are executed in sequence. The output which conforms to an acceptance test is selected. The weakness in this system is writing an appropriate acceptance test.

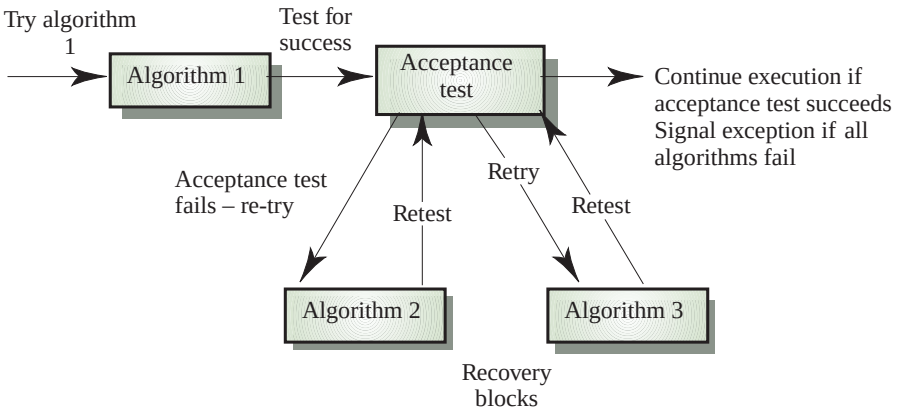
N-version programming



N-version programming

- The different system versions are designed and implemented by different teams. It is assumed that there is a low probability that they will make the same mistakes
- However, there is some empirical evidence that teams commonly misinterpret specifications in the same way and use the same algorithms i their systems

Recovery blocks



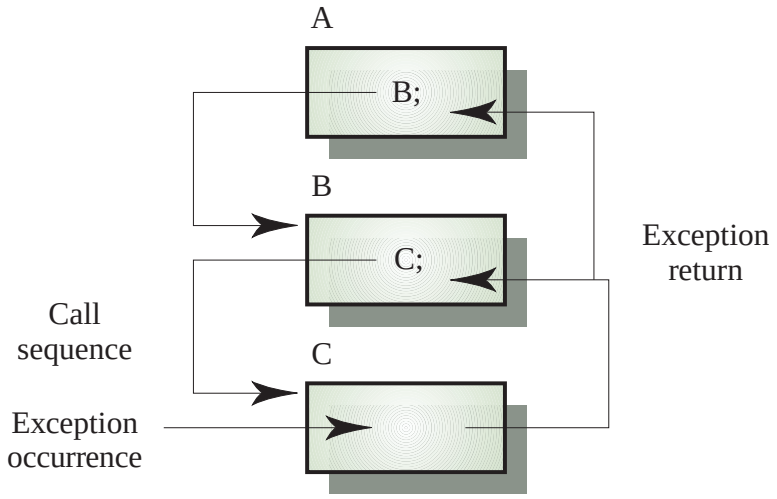
Recovery blocks

- Force a different algorithm to be used for each version so they reduce the probability of common errors
- However, the design of the acceptance test is difficult as it must be independent of the computation used
- Like N-version programming, susceptible to specification errors

Exception handling

- A program exception is an error or some unexpected event such as a power failure.
- Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- Using normal control constructs to detect exceptions in a sequence of nested procedure calls needs many additional statements to be added to the program and adds a significant timing overhead.

Exceptions in nested procedure calls



Ada exception handling

- Ada has a built-in type exception and names can therefore be associated with exceptions
- Drawing attention to an exception is called raising the exception (keyword **raise**)
- An Ada program unit can have an exception handler which is a block of code defining how exceptions should be processed
- Code is automatically switched to the exception handler when an exception is raised

Ada's built-in exceptions

- CONSTRAINT_ERROR
  - Raised when an attempt is made to assign an out of range value to a variable e.g. array access out of bounds.
- NUMERIC_ERROR
  - Raised when an error occurs in an arithmetic operation (e.g. division by zero)
- PROGRAM_ERROR
  - Raised when a control structure is violated.
- STORAGE_ERROR
  - Raised when dynamic store is exhausted.
- TASKING_ERROR
  - Raised when inter-task communication fails.

C++ exception handling

- Keyword **throw** means raise an exception. Handler is indicated by the keyword **catch**
- Exceptions are defined as classes so may inherit properties from other exception classes
- Normally, exceptions are completely handled in the block where they arise rather than propagated for handling
- All exceptions are user-defined. No built-in exceptions.

A temperature controller

- Controls a freezer and keeps temperature within a specified range
- Switches a refrigerant pump on and off
- Sets of an alarm is the maximum allowed temperature is exceeded
- Uses external entities Pump, Temperature_dial, Sensor, Alarm.
- External shared state is held in a package called Globals (in Ada)

Freezer controller (Ada)

- See portait slide

Freezer controller (C++)

- See portrait slide

Defensive programming

- An approach to program development where it is assumed that undetected faults may exist in programs
- The program contains code to detect and recover from such faults
- Does NOT require a fault-tolerance controller yet can provide a significant measure of fault tolerance

Failure prevention

- Type systems allow many potentially corrupting failures to be detected at compile-time
- Range checking and exceptions allow another significant group of failures to be detected at run-time
- State assertions may be developed and included as checks in the program to catch a further class of system failures

Ada range checking

- Types are declared as an allowed range e.g. 1..100
- Ada's range checking automatically raises a CONSTRAINT_ERROR exception if an assignment is out of range
- Range checking only applies to a single variable. Checks which apply across variables (e.g. if A=0 then B=1) cannot be applied.
- Out of range errors may require further processing to locate the error source

State assertions

- Logical predicates over the system state variables.
- May be incorporated directly in a language but this can cause compilation problems if quantifiers are used.
- Usually implemented as program checks.
- Simplifed if ALL state operations are through abstract data types. In many cases, predicates need only be associated with the ADT.

Even number type

- See portrait slide

Assertion checking

- See portrait slide

Damage assessment

- Analyse system state to judge the extent of corruption caused by a system failure
- Must assess what parts of the state space have been affected by the failure
- Generally based on 'validity functions' which can be applied to the state elements to assess if their value is within an allowed range

Damage assessment techniques

- Checksums are used for damage assessment in data transmission
- Redundant pointers can be used to check the integrity of data structures
- Watch dog timers can check for non-terminating processes. If no response after a certain time, a problem is assumed

Ada type with damage assessment

- See portrait slide

C++ class with damage assessment

```
template <class elem> class Robust_array {
public:
        Robust_array (int size = 20) ;
        ~Robust_array () ;
        void Assign ( int Index, elem Val) ;
        elem Eval  (int Index) ;

        // Damage assessment functions
        // Assess_damage takes a pointer to a  function as a parameter
        // It sets the corresponding element of Checks if a problem is
        // detected by the function Test
        void Assess_damage ( void (*Test ) (boolean*)) ;
        boolean Eval_state (int Index) ;
        boolean  Is_damaged () ;
private:
        elem* Vals ;
        boolean* Checks ;
} ;
```

Fault recovery

- Forward recovery
  - Apply repairs to a corrupted system state
- Backward recovery
  - Restore the system state to a known safe state
- Forward recovery is usually application specific - domain knowledge is required to compute possible state corrections
- Backward error recovery is simpler. Details of a safe state are maintained and this replaces the corrupted system state

Forward recovery

- Corruption of data coding
  - Error coding techniques which add redundancy to coded data can be used for repairing data corrupted during transmission
- Redundant pointers
  - When redundant pointers are included in data structures (e.g. two-way lists), a corrupted list or filestore may be rebuilt if a sufficient number of pointers are uncorrupted
  - Often used for database and filesystem repair

Backward recovery

- Transactions are a frequently used method of backward recovery. Changes are not applied until computation is complete. If an error occurs, the system is left in the state preceding the transaction
- Periodic checkpoints allow system to 'roll-back' to a correct state

Safe sort procedure

- Sort operation monitors its own execution and assesses if the sort has been correctly executed
- Maintains a copy of its input so that if an error occurs, the input is not corrupted
- Based on identifying and handling exceptions
- Possible in this case as ‘valid’ sort is known. However, in many cases it is difficult to write validity checks

Safe sort procedure (Ada)

- See portrait slide

Safe sort procedure (C++)

- See portrait slide

Key points

- Reliability in a system can be achieved through fault avoidance and fault tolerance
- Some programming language constructs such as gotos, recursion and pointers are inherently error-prone
- Data typing allows many potential faults to be trapped at compile time.
- Fault tolerant software can continue in execution in the presence of software faults

Key points

- Fault tolerance requires failure detection, damage assessment, recovery and repair
- N-version programming and recovery blocks are approaches to fault tolerance
- Exception handling mechanisms can be used to recover from failure
- Defensive programming can provide some fault tolerance without a special fault-tolerant controller