Exceptions and Exception Handling

Rajesh Gupta
University of California, San Diego
System Characteristics

- Complexity in function (and in size)
- Concurrent control of separate components
  - devices operate in parallel
- Facilities to interact with special purpose hardware
  - need to program devices in a reliable and abstract way
- **High reliability and safety**
  - failure has severe life, environmental and economic consequences
  - Exception handling
  - Explore various models of exception handling and to show how exception handling can be used as a framework for implementing fault-tolerant systems.
- Guaranteed response times
  - predictability is important; sometimes more so than efficiency.
Outline

• Language requirements for exception handling
• Exception handling in “classical” languages
• Modern exception handling
  – Representation, domain, propagation of exceptions
• Exception handling and OS support
• Recovery blocks and exceptions

Reference: Burns and Wellings, Chapter 6
Desired Features in Language Facilities for Exception Handling

R1: simple and easy to understand language facilities
R2: separation of normal and exception processing, unobtrusive code
R3: runtime overheads are incurred only when handling an exception
    exception handling does not affect normal operation
    in some cases, speed of recovery may be of primary importance
R4: uniform treatment of exceptions detected by the environment and by the program
    for example, arithmetic overflow (in hardware) handled same way as an assertion failure in program
R5: allow recovery blocks to be programmed.
Exception Handling in Classical and Modern Languages

• Following techniques have been used in classical languages for handling exceptions:
  1. Unusual return value
  2. Forced branch
  3. Non-local goto
  4. Procedure variable

• More recently, programming languages use
  1. explicit representation and handling of exceptions
  2. exception domains and exception propagation
  3. resumption techniques
1. Unusual Return Value

• An “unusual” or error value is returned from a procedure or a function, e.g., in C
• No new mechanism
• Exception handling part of the normal processing:

```c
if(function_call(parameters) == AN_ERROR) {
    -- error handling code
} else {
    -- normal return code
}
```

• Syntactic sugar can ease coding, clarity
  ```c
  #define Sys_EHandler(param) if (sys_call(param) == Error) error();
  ```
Unusual Return

- Simple and allows recovery actions to be programmed.
- R1: Simplicity: OK
- R2: Unobtrusive: Not OK
- R3: No overheads in normal processing: Not OK
- R4: Uniform treatment: Not OK
- R5: Recovery blocks: OK
2. Forced Branch

- Used mainly in assembly languages
  - typical mechanism is for subroutines to skip return
  - that is, the instruction following the subroutine call is skipped to indicated the presence/absence of an error
    - this is done by the subroutine incrementing its return address (program counter) by the length of a simple jump instruction
  - more than one exceptional return is possible by manipulating the PC accordingly

```
jsr pc, PRINT_CHAR
jmp IO_ERROR
jmp DEVICE_NOT_ENABLED
# normal processing
```

- here the subroutine, for a normal return, would increment the return address by two jmp instructions.
Forced Branch

• Little overhead, and enables recovery actions to be programmed, but obscure programs.

• R1: Simplicity: Not OK
• R2: Unobtrusive: Not OK
• R3: No overheads in normal processing: OK
• R4: Uniform treatment: Not OK
• R5: Recovery blocks: OK
3. Non-local Goto

- A high-level language version of a forced branch which uses label variables
- Use different labels to be passed as parameters to procedures to transfer control
- For example: RTL/2 provides a non-local goto
  - RTL/2 uses “bricks” to structure its programs
  - a brick can be data, a procedure or a stack
  - a special type of data brick defined by the system is called svcdata
  - one such brick (rrerr) provides a standard error-handling facility by using a error variable erl
svc data rrerr
    label erl; %a label variable %
enddata

proc WhereErrorIsDetected();
    ...
    goto erl;
    ...
endproc;

proc Caller();
    ...
    WhereErrorIsDetected();
    ...
endproc;

proc main();
    ...
restart:
    ...
    erl := restart;
    ...
    Caller();
    ...
end proc;
Non-local Goto

• The goto is more than just a jump!
  – it implies an abnormal return from a procedure
    • that means, the stack must be unwound until the
      environment is restored to that of the procedure
      containing the declaration of the label (erl)
    – the penalty for this unwinding is incurred only when an
      error has occurred
    – very flexible, but can lead to obscure programs.
• R1: Simplicity: Not OK
• R2: Unobtrusive: Not OK
• R3: No overheads in normal processing: OK
• R4: Uniform treatment: OK
• R5: Recovery blocks: OK
4. Procedure Variable

• Using non-local goto results in a break in the control flow of the program

• as a result, error label is generally used for unrecoverable errors

• instead, an error procedure variable is used when the control should be returned to the point where the error originated
svc data rerr;
    label erl;
    proc(int) erp; % erp is a procedure variable %
enddata;

proc recover(int);
    ...
endproc;

proc WhereErrorIsDetected();
    ...
    if recoverable then
        erp(n)
    else
        goto erl
    end;
end;

proc Caller();
    ...
    WhereErrorIsDetected();
    ...
endproc;

proc main();
    ...
    erl := fail;
    erp := recover;
    ...
    Caller();
    ...
    fail:
    ...
end proc
Procedure Variable

• Programs can become very difficult to understand and maintain

• Recently, languages like C++ provide default functions (within the context of language-level exception handling)
  – called when no handler for an exception can be found
  – these functions can be redefined by the programmer.
Modern Exception Handling

• Traditional approaches intermingle normal and exception processing

• An alternative is to add exception handling as a part of the language facility
  – “first class” treatment, part of the language, more structured code

• To do that, one has to understand
  1. the nature of exceptions and how to represent these
  2. the domain of exception handler
  3. how exceptions propagate
Modern Exception Handling

1. the nature of exceptions and how to represent these
2. the domain of exception handler
3. how exceptions propagate
Exceptions and Their Representation

• Exceptions can be detected by
  1. program (application)
  2. environment (hardware, software)

• Exceptions can be raised
  1. synchronously as an immediate result of a process attempting an inappropriate operation
  2. asynchronously, some time after the operation which resulted in the error occurrence.
    • May be raised in the process which executed the operation or in another process.

• Accordingly there are four classes of exceptions
  – ES, PS, EA, PA
Classes of Exceptions

1. ES: detected by the environment and raised synchronously
   – e.g., array bounds violation, divide by zero
2. PS: detection by program and raised synchronously
   – e.g., failure of a program defined assertion check
3. EA: detected by the environment and raised asynchronously
   – e.g., an exception raised due to the failure of some health monitoring mechanism, power failure
4. PA: detected by program and raised asynchronously
   – e.g., one process may recognize that an error condition has occurred and will result in another process not meeting its deadline or not terminating correctly

• Asynchronous exceptions are called signals and handled part of concurrency control mechanisms.
  – Concurrency control addressed later in the course.
Synchronous Exceptions

• Declared in one of two ways:
  – an explicitly declared constant name, or
  – use the typing mechanism. An exception is an object of a particular type that may or may not be explicitly declared

• For example: ADA
  – exceptions declared like constants
  – exceptions that can be raised by the runtime are declared in a package that is visible to all ADA programs.

• For example: C++
  – exceptions can be thrown without predeclaration.
  – exceptions can be any object type and “caught” by a handler that names the object type or its parent.
Modern Exception Handling

1. the nature of exceptions and how to represent these
2. the domain of exception handler
3. how exceptions propagate
The Domain of an Exception Handler

• Within a program, there may be several handlers for a particular exception
• Associated with each handler is a domain
  – domain specifies the region of computation during which, if an exception occurs, the handler will be activated.
• The domain specification determines how precisely the source of an exception is located
  – useful in detecting what caused an error
  – statement or block level
  – Not all blocks can have exception handlers
• In a block structured language, the domain is the block

Raised by the Ada RTS

```
declare
  subtype Temperature is Integer range 0 .. 100;
begin
  -- read temperature sensor and calculate its value
  exception
    -- handler for Constraint_Error
end;
```
The Domain

• When blocks form the basis of other units, such as procedures and functions, the domain of an exception handler is usually that unit.
• In some languages (C++, Modula-3) not all blocks can have exception handlers
  – domain of an exception handler is explicitly indicated
  – e.g., a guarded block or a try block

```plaintext
try {
    // statements which may raise exceptions
}
catch (exception_name) {
    // handler for exception_name
}
```
Granularity of a Domain

• Sometimes block level granularity may not be adequate for embedded systems
• For example, consider a handler that has to determine which calculation caused exception to be raised:
  – even more difficult in case of environmental errors (arithmetic overflow/underflow)

```plaintext
declare
    subtype Temperature is Integer range 0 .. 100;
    subtype Pressure is Integer range 0 .. 50;
    subtype Flow is Integer range 0 .. 200;
begin
    -- read temperature sensor and calculate its value
    -- read pressure sensor and calculate its value
    -- read flow sensor and calculate its value
    -- adjust temperature, pressure and flow
    -- according to requirements
exception
    -- handler for Constraint_Error
end;
```
Solution 1: Decrease the block size or nest them.

```plaintext
declare
define subtype Temperature is Integer range 0 .. 100;
define subtype Pressure is Integer range 0 .. 50;
define subtype Flow is Integer range 0 .. 200;
begin
  begin
    -- read temperature sensor and calculate its value
  exception
    -- handler for Constraint_Error for temperature
  end;
begin
  begin
    -- read pressure sensor and calculate its value
  exception
    -- handler for Constraint_Error for pressure
  end;
begin
  begin
    -- read flow sensor and calculate its value
  exception
    -- handler for Constraint_Error for flow
  end;
  -- adjust temperature, pressure and flow according
  -- to requirements
exception
  -- handler for other possible exceptions
end;
```
Solution 2: Statement level exception handling

• Hypothetically can be done as:
  • While ADA does not allow it
    – it has been tried in CHILL programming language
  • But then it messes up code by intermingling normal an exception processing
    – R2

By now, it must be clear what the preferred approach should be?
Solution 3: Parameters

• Allow parameters to be passed within the exceptions

• Automatically done in C++
  – exception is an object and, therefore, can contain as much information as the programmer wants

• Ada provides a predefined procedure `Exception_Information` which returns implementation defined details on the occurrence of an exception.
Modern Exception Handling

1. the nature of exceptions and how to represent these
2. the domain of exception handler
3. how exceptions propagate
Exception Propagation

• If there is no handler associated with the block or procedure, the exception can be treated one of two ways:
  – treat it as a programmer error or propagate the exception

• Programmer error
  – reported at compile time
  – does not always work since an exception can be raised in a procedure that can only be handled within the context from which the procedure was called
    • that is, it is not possible to have the handler local to the procedure
    • e.g., an exception raised in a procedure as a result of a failed assertion involving parameters

• compile time error generation requires that a procedure specify which exceptions it may raise (i.e., not handle locally).
  – The compiler can then check the calling context for an appropriate handler. (CHILL)
  – C++ allows a fn to define its exceptions. Unlike CHILL it does not require a handler to be available in the calling context.
Second Approach: Propagation

- If no local handler is found, look for handlers up the chain of invokers at runtime
  - I.e., propagate the exception
- Ada and C++ allow exception propagation
- However, this may be a problem when the language requires exceptions to be declared (and thus given scope)
  - It is possible for an exception to be propagated outside its scope, thereby making it impossible for a handler to be found.
  - To cope with this situation, most languages provide a catch all exception handler
    - Also a programming convenience (instead of enumerating many exception names)
- An unhandled exception causes a sequential program to be aborted (or a process to be aborted)
- Handlers can also be dynamically associated
  - More overhead, more flexible, better propagation.
Resumption versus Termination

- What happens after an exception has been handled?
- Resumption or Notify model
  - if the handler has “cured” the problem, the invoker of the exception can continue its execution
  - useful in asynchronous exceptions (exception has little to do with current process execution)
  - can also be a “retry” -- local variables must not be reinitialized
- Termination or Escape model
  - control is not returned to the invoker, instead it goes to the calling block/procedure/process
  - if the programmer were to attempt recovery here, must provide additional facilities to manipulate program state
- Hybrid model
  - handler can decide whether to resume or to terminate.
Resumption Model

- Handler is an implicit procedure which is called when the exception is raised.
- However, it is difficult to repair errors raised by the runtime system:
  - e.g., arithmetic OVO in the middle of a complex expression, results in several registers holding partial evaluations.
  - handler may overwrite these registers.
- Strict resumption is difficult:
  - “retry” as a part of exception handling (however, do not reinitialize local vars.)
Termination Model

- Control does not return to the point where the exception occurred
- Instead, the block or procedure containing the handler is terminated, and control is passed to the calling block or procedure
  - an invoked procedure, therefore, may terminate in one of a number of conditions.
  - Normal condition, exception condition.
- When the handler is inside a block, control is given to the first statement following the block after the exception handling
  - with procedures, the control flow may change dramatically.
  - Supported by C++, Ada.
declare
   subtype Temperature is Integer range 0 .. 100;
begin
   begin
      -- read temperature sensor and calculate its value,
      -- may result in an exception being raised
   exception
      -- handler for Constraint_Error for temperature,
      -- once handled this block terminates
   end;

   -- code here executed when block exits normally
   -- or when an exception has been raised and handled.

exception
   -- handler for other possible exceptions
end:
Exception Handling and OS

• OS often provide supports for detection of certain synchronous error conditions
  – e.g., memory violation, illegal instruction
• These usually result in the process being terminated
  – however, some systems allow programmer to attempt recovery
• For instance, POSIX allows the program to handle these exceptions by way of signals by associating handlers to these
  – the handler is called when the error is detected. Once finished, the process resumes where it was interrupted.
• If a language supports the termination model, then the runtime system must
  – catch the error and
  – undertake the necessary manipulation of the program state so that the program can use the termination model.
Exception Handling in C

- no exception handling facilities
- possible to provide exception handling using MACROs
- to implement termination model, it is necessary to save the status of a program’s registers etc on entry to an exception domain and restore them if an exception occurs
- support from UNIX: setjump, longjmp
  - setjump save the program status and returns a 0
  - longjmp restores the program status and results in the program abandoning its current execution and restarting from the position where setjump was called.
    - This time setjump returns the values passed by longjump
typedef char *exception;
    /* a pointer type to a character string */
exception error = "error";
    /* the representation of an exception named "error" */

if(current_exception = (exception) setjmp(save_area) == 0) {
    /* save the registers and so on in save_area */
    /* 0 is returned */

    /* the guarded region */

    /* when an exception "error" is identified */
    longjmp(save_area, (int) error);
    /* no return */
}
else {

    if(current_exception == error) {
        /* handler for "error" */
    }
    else {
        /* re-raise exception in surrounding domain */
    }
}
Exception Handling in C++

• Supports a termination model of exception handling
• Allows arbitrary objects to represent exceptions
• Does not require explicit declaration of exceptions
  – any instance of a class can be “thrown” as an exception
• There are no predefined exceptions.
• Example: Temperature Sensor
  – declare an object type to represent constraint error for integer subtypes

```cpp
class integer_constraint_error {
  public:
    int lower_range;
    int upper_range;
    int value;

    integer_constraint_error(int L, int U, int V) {
      /* constructor */
      lower_range = L;
      upper_range = U;
      value = V;
    }
};
```
class temperature {
    int T;
    public:
    temperature(int) throw(integer_constraint_error);
        /* constructor */
    int operator=(int) throw(integer_constraint_error);
        /* both the constructor and the "=" operator can throw */
        /* an integer_constraint_error */
};

class temperature_controller {
    temperature T;
    public:
    class actuator_dead {
    void set_temperature(temperature)
        throw(actuator_dead, integer_constraint_error);
    temperature read_temperature();
    temperature_controller(temperature);
    };
C++ Exception Handling

- In general each function may specify
  - a list of throwable exceptions,
  - the function may throw any exception in the list
  - an empty list of throwable exceptions throw ()
    - the function will throw no exceptions
  - no list of throwable exceptions
    - the function can throw any exception
- If a function attempts to throw an exception not in its throw list, the function unexpected is called.
- The default for unexpected is to call the terminate function whose default is to abort the program

```c++
typedef void(*PFV)();

PFV set_unexpected(PFV new_handler);
/* sets the default action to new_handler and */
/* returns the previous action */

PFV set_terminate(PFV new_handler);
/* sets the default action to new_handler and */
/* returns the previous action */
```
class temperature {
    int T;
public:
    temperature (int initial) throw (int_const_error){
        check (initial);
    }
    int operator = (int i) throw (int_cont_error){
        check (i); return T;
    }
private:
    void check (int value) {
        if (value > 100 || value < 0) {
            throw int_const_error (0, 100, value);
        } else {
            T = value;
        }
    }
};
Exception Handling Domain

- Only from within a try-block
- Each handler specified using a catch statement
  - Catch statement is like a function declaration, the parameter of which identifies the type to be caught. Inside the handler, the object name behaves like a local variable.
  - Because of inheritance, exception hierarchies can be easily constructed.
    - Handler type T is the base class of Exception at the throw point
    - Declare a base type exception and derive various error types, the try block will catch all of these errors

```cpp
try {
    temperature_controller TC (20);
}
catch (int_const_error) {
    // exception caught
    cout << "Error using a constrained int type"
        << error.lower_range << error.upper_range << "Value " << error.value << "\n";
}
catch (temperature_controller::dead_actuator error) {
    cout << "Temp actuator not responding\n";
}
catch (...) {
```


Example

• Declare a base type ‘exception’ and derive multiple exceptions which are caught by a single try-catch

    Class exception {
        public: virtual void message () { cout << “exception raised\n”; };
    }

    Class integer_constraint_error : public exception {
        int lower_range; int upper_range; int value;
        public:
            integer_constraint_error (int L, int U, int V) {
                lower_range = L; upper_range = U; value = V;
            }
            virtual void message () {
                cout << “constraint error”; }
    }

    Class actuator_dead: public exception {
        public:
            virtual void message () {
                cout << “dead actuator”;
                exception:: message(); }
    }

Try {
    // statements which may raise exceptions...
}
Catch (exception E) {
    // the handler will catch all exceptions of type exception
}
Recovery Blocks Revisited

- The error detection facility is provided by the acceptance test.
- This test is simply the negation of a test which would raise an exception using forward error recovery.
- However, there is a problem in implementation of state save and restore.
  - In general, this may require RT and/or hardware support.
  - More desirable to allow more basic primitives and let the programmer do this.
- So what is the stuff RT is doing here?

```plaintext
package Recovery_Cache is
  procedure Save; -- save volatile state
  procedure Restore; -- restore state
end Recovery_Cache;
```
Recovery blocks can be used in an exception handling environment.

```plaintext
procedure Recovery_Block is
    Primary_Failure, Secondary_Failure,
    Tertiary_Failure: exception;
Recovery_Block_Failure: exception;
    type Module is (Primary, Secondary, Tertiary);

function Acceptance_Test return Boolean is
begin
    -- code for the acceptance test
    end Acceptance_Test;

procedure Primary is
begin
    -- code for primary algorithm
    if not Acceptance_Test then
        raise Primary_Failure;
    end if;
exception
    when Primary_Failure =>
        -- forward recovery to return environment
        -- to the required state
        raise;
    when others =>
        -- unexpected error
        -- forward recovery to return environment
        -- to the required state
        raise Primary_Failure;
end Primary;
-- similarly for Secondary and Tertiary

begin
    Recovery_Cache.Save;
    for Try in Module loop
        begin
            case Try is
                when Primary => Primary; exit;
                when Secondary => Secondary; exit;
                when Tertiary => Tertiary;
            end case;
            exception
                when Primary_Failure =>
                    Recovery_Cache.Restore;
                when Secondary_Failure =>
                    Recovery_Cache.Restore;
                when Tertiary_Failure =>
                    Recovery_Cache.Restore;
                    raise Recovery_Block_Failure;
                when others =>
                    Recovery_Cache.Restore;
                    raise Recovery_Block_Failure;
            end;
        end loop;
    end Recovery_Block;
```
Recovery Blocks

• By using exception handlers, forward error recovery can be achieved before restoring the state
• Recovery blocks can be implemented using a language with exceptions plus some RT support (e.g., raising of error flags)
Summary

• Models for exception handling must consider
  – exception representation, domain of handler, propagation, resumption/termination
• Parameter passing to handlers, if allowed, provides a greater flexibility in exception handling and its aftermath
• It is not clear if exception handling is entirely a programming language issue
  – to what extent the runtime system must participate in it?
• Or even if it should even be a programming language issue
  – e.g., C/OCCAM2 do not have any
    • “after all exception is a GOTO where the destination and source are unknown”
  – antithesis of structured programming?