Tema 4. Fiabilidad. Tratamiento de Errores

4.1 Introducción

A diferencia del software científico y del comercial, el software de tiempo real tiene que funcionar muy a menudo ininterrumpidamente, aún en presencia de condiciones de error. Si una aplicación que calcula la solución de algún problema científico falla, puede ser razonable abortar el programa; lo único que habremos perdido es tiempo de computación. Sin embargo, en el caso de un sistema en tiempo real, abortar el programa puede no ser una solución aceptable. Por ejemplo, un computador encargado del control de una gran fábrica de gas no puede permitir su parada cuando se produce una situación de error. Por el contrario, debe intentar que la operación continúe de la mejor forma posible. La calidad del control puede quedar degradada, pero esto será generalmente preferible a iniciar una costosa operación de apagado.

Fiabilidad y seguridad son por tanto requisitos usualmente mucho más rigurosos para sistemas incrustados que para el resto de los sistemas computacionales. Cada día hay más funciones de control previamente realizadas por humanos o por sistemas analógicos que son ahora llevadas a cabo por computadoraa digitales. Por tanto, cada vez es más imperativo que estos sistemas no fallen.

Seguridad y fiabilidad pueden estar en conflicto: un sistema es fiable si cumple sus especificaciones; es seguro si no se pueden producir situaciones que causen daños, independientemente de que se cumpla la especificación o no.

Los fallos de funcionamiento de un sistema empotrado pueden tener su origen en:

- Una especificación inadecuada para el desarrollo del software.
- Errores en el diseño del software.
- Averías en el hardware.
- Interferencias transitorias o permanentes en las comunicaciones.

La **fiabilidad** (*reliability*) de un sistema se puede definir como la medida del éxito con que el sistema cumple las especificaciones fijadas para su comportamiento. Cuando el comportamiento del sistema se desvía del especificado, se dice que se produce una **avería** (*failure*). Es la manifestación externa de problemas internos en el sistema, llamados**errores** . Las causas mecánicas o algorítmicas de los errores se denominan **fallos** o faltas (*faults*). Un sistema está habitualmente formado por componentes, que pueden ser considerados a su vez como sistemas. Una avería en un sistema (componente) puede producir un fallo en el sistema que lo contiene, que dará lugar a un error y a una potencial avería en éste último:

Se pueden distinguir tres tipos de fallos:

1. Transitorios. Comienzan en un instante de tiempo, permanecen en el sistema durante un periódo de tiempo y luego desaparecen solos. Ejemplos de estos fallos son componentes hardware que presentan una reacción adversa ante alguna interferencia externa. Cuando la interferencia desaparece, también lo hace el fallo (aunque no necesariamente el error inducido). Muchas fallos en sistemas de comunicaciones son transitorios.
2. Permanentes. Comienzan en un instante de tiempo, y permanecen en el sistema hasta que son reparados. Por ejemplo, una línea rota, o un diseño de software erróneo.
3. Intermitentes. Fallos transitorios que ocurren de vez en cuando. Por ejemplo, componentes hardware sensibles a la temperatura: a partir de una temperatura, dejan de funcionar. Si ésta baja, vuelven a funcionar.

4.1.1 Prevención y tolerancia de fallos

Se puede distinguir entre dos aproximaciones que ayudan a los diseñadores a mejorar la fiabilidad de los sistemas, es decir, a asegurar que un fallo potencial no cause una avería del sistema. La primera es conocida como prevención de fallos; ésta intenta eliminar la posibilidad de fallos antes de que el sistema sea operacional. La segunda es la tolerancia de fallos (del inglés *fault tolerance*). Se trata de conseguir que el sistema continúe funcionando aunque se produzcan fallos.

a) Prevención de fallos

Hay dos niveles en la prevención: evitar fallos y eliminar fallos.

Evitar fallos implica limitar la introducción de componentes potencialmente defectuosos en la construcción del sistema. Para componentes hardware, esto conlleva la utilización de los componentes más fiables (usualmente más caros), y las mejores técnicas de interconexión. En cuanto al software, su calidad puede mejorarse mediante:

- Especificaciones rigurosas de requisitos.
- El uso de buenas metodologías de diseño.
- El uso de lenguajes con facilidades para modularidad y abstracción de datos.
- Utilización de entornos de desarrollo con computador (CASE) adecuados para gestionar los componentes software.

A pesar de estas técnicas, las faltas van a estar presentes inevitablemente en el sistema después de su construcción.

El segundo nivel en la prevención es por tanto la eliminación de tales fallos o faltas. Esto consiste normalmente en métodos para encontrar y eliminar las causas de los errores, con especial énfasis en los tests al sistema. Desafortunadamente, los tests nunca pueden ser exhaustivos y eliminar todas las potenciales causas de errores. En particular, se presentan los siguientes problemas:

- Un test puede ser usado para mostrar la presencia de defectos (fallos), no su ausencia.
- Muchas veces es imposible testear bajo condiciones reales. Los tests se hacen bajo simulación, y es difícil garantizar que la simulación cubra todos los casos reales.
- Los errores introducidos en la etapa de especificación de requisitos pueden no ser detectados.

A pesar de todos los tests y verificaciones para eliminar fallos, los componentes hardware fallarán. La prevención de fallos no servirá cuando la duración del tiempo de reparación sea inaceptable, o cuando el sistema esté inaccesible para operaciones de mantenimiento (naves espaciales sin tripulación).

b) Tolerancia de fallos, o tratamiento de errores (*fault tolerance*)

A causa de la limitaciones inevitables de la prevención de fallos, los diseñadores de sistemas incrustados deben considerar el uso de la tolerancia de fallos. Esta puede presentar diferentes grados:

- Tolerancia completa. El sistema continúa operando en presencia de errores, aunque sea durante un tiempo limitado, sin una pérdida significativa de su funcionalidad.
- Degradación suave o aceptable. El sistema continúa operando en presencia de fallos, con una degradación parcial en su funcionamiento, hasta su reparación o recuperación.
- Parada segura. El sistema mantiene su integridad, siempre que sea admisible una parada temporal en su funcionamiento. Ante un fallo, se deja al sistema a controlar en un estado estable (seguro) y el sistema de control se detiene hasta que se repare el fallo.

Aunque en teoría la mayor parte de los sistemas críticos (en cuanto a seguridad) requieren tolerancia completa, en la práctica muchos aceptan una degradación suave, o varias formas de degradación suave.

4.2 Tolerancia de fallos

El grado de tolerancia necesario dependerá de la aplicación. Todas las técnicas de tolerancia de fallos se apoyan en la introducción de elementos extras en el sistema, para detectar y recuperarse ante fallos. Estos componentes extras son redundantes, en el sentido de que no son necesarios para un modo normal de funcionamiento. El fin de la tolerancia de fallos es minimizar la redundancia para una fiabilidad máxima.

La introducción de componentes redundantes inevitablemente aumenta la complejidad del sistema, y a su vez puede introducir fallos adicionales. Es conveniente separar los componentes tolerantes del resto del sistema.

4.2.1 Redundancia hardware

Redundancia estática.

Los componentes redundantes se utilizan en el sistema para enmascarar u ocultar los efectos de los fallos. Estos componentes están siempre activos. Un ejemplo es la redundancia módulo N (*NMR, N-Modular Redundancy*). Consiste en N componentes idénticos y unos circuitos encargados de comparar las salidas de los N componentes. Si uno difiere del resto, su salida se enmascara.

Redundancia dinámica.

Proporciona facilidad para detección de errores, en lugar de enmascaramiento de fallos. Unos componentes se encargarán de la detección (por ejemplo, bits de paridad en las memorias) y otros de la recuperación ante esos errores.

4.2.2 Tolerancia de fallos en software

A destacar dos técnicas de tolerancia de fallos en software: programación con N versiones, análoga a la redundancia estática en hardware, y detección y recuperación ante errores, análoga a la redundancia hardware dinámica.

4.2.2.1 Programación con N versiones

Se define como la generación independiente de N (N>=2) programas funcionalmente equivalentes, a partir de las mismas especificaciones iniciales. La generación independiente significa que N individuos o grupos producen las N versiones del software SIN INTERACCIONAR. Una vez diseñados y escritos, estos programas se ejecutan concurrentemente con las mismas entradas, y otro proceso (proceso guía) se encarga de controlar su ejecución y de comparar sus resultados. Como fruto de la comparación, el proceso guía indicará las acciones que cada programa debe seguir. Posibles acciones:

- Continuación.
- Terminación de una o más versiones.
- Continuación tras cambiar uno o más resultados al valor del resultado mayoritario.

La programación con N versiones asume que no hay relación entre los fallos en una versión y los fallos en otra. Esto puede quedar invalidado si todas las versiones están escritas en el mismo lenguaje, o utilizando el mismo

compilador o el mismo entorno soporte. Por ejemplo, en el sistema de control de vuelo del Boeing 777 se utiliza un único programa Ada, pero tres diferentes procesadores y 3 diferentes compiladores.

La correcta aplicación de este método depende de:

- Especificación inicial. Un error de especificación se manifestará en todas las versiones.
- Desarrollo independiente. No debe haber interacción entre los equipos.
- Presupuesto suficiente. Los costes de desarrollo se multiplican, y el mantenimiento también.

4.2.2.2 Redundancia dinámica en software

Los componentes redundantes se ejecutan sólo cuando se detecta un error. Se distinguen cuatro etapas:

1. Detección de errores. La mayor parte de los fallos se manifestarán en forma de errores. Ha de ser el primer paso en la utilización de una técnica de tolerancia de fallos.
2. Evaluación y confinamiento de los daños. El intervalo de tiempo entre la ocurrencia de un fallo y la detección del error indica que información errónea se ha podido extender por el sistema.
3. Recuperación ante errores. Es probablemente la etapa más importante. Se trata de situar al sistema erróneo en un estado desde el que pueda continuar funcionando normalmente, o quizás con una funcionalidad degradada.
4. Reparación de fallos. Un error es el síntoma de un fallo. Aunque el sistema funcione, el fallo puede persistir, y hay que repararlo.

1. La detección de errores ocurre normalmente a dos niveles distintos:

a) **En el ámbito de la implantación**. Errores detectados por el entorno en el que se ejecuta el programa. Incluye los detectados por el hardware (instrucción ilegal, overflow aritmético,...) y por el sistema soporte del lenguaje (puntero nulo, valor fuera de rango,...).

b) **En el ámbito de la aplicación**. Se pueden detectar errores mediante comprobaciones (tests) explícitamente programados, por ejemplo de la forma if CONDICION then ERROR ...

Algunas técnicas son:

- Tests de tiempos: si un proceso no finaliza dentro de un intervalo de tiempo fijado se asume que se produce un error.
- Tests de datos: información redundante incluída con los datos, para comprobar su validez (por ejemplo, para comunicaciones).
- Tests inversos: con el valor de la salida se calcula cual debería ser el valor de la entrada, valor que se compara con el real.

2. Es importante confinar los daños causados por un fallo en una parte limitada del sistema. Algunas técnicas:

- **Descomposición modular.** El sistema debe romperse en componentes, cada uno de ellos representado por uno o varios módulos. Estos tendrán unos interfaces bien definidos, y los detalles internos no serán accesibles desde el exterior.
- **Acciones atómicas.** Para el resto del sistema, una acción atómica aparece como indivisible y se ejecuta instantáneamente.

3. Hay dos formas de llevar a cabo la recuperación ante errores:

- **Recuperación directa (hacia adelante).** Se avanza desde un estado erróneo, haciendo correcciones sobre partes del estado. Depende de una predicción correcta de la localización y causa de los posibles errores. La forma de hacerla es específica para cada sistema.
- **Recuperación inversa (hacia atrás).** Se retrocede a un estado correcto que se ha guardado previamente. El punto al que retorna el proceso es llamado **punto de recuperación**. Sirve para fallos imprevistos, pero no se pueden deshacer los efectos que el fallo haya causado en el entorno del sistema. Una técnica de recuperación inversa es la llamada **bloques de recuperación**. Un bloque de recuperación es un bloque tal que:
  - Su entrada es un punto de recuperación.
  - A su salida se efectúa una prueba de aceptación. Si el módulo primario del bloque termina en un estado correcto, se sale del bloque. Si la prueba de aceptación falla, se restaura el estado inicial en el punto de recuperación, y se ejecuta un módulo alternativo del mismo bloque. Si vuelve a fallar, se siguen intentanto alternativas. Cuando no quedan más, el bloque falla y hay que intentar la recuperación en un nivel más alto (bloques anidados).

4. La reparación automática de fallos es difícil y depende del sistema concreto. Hay dos etapas:

a) Localización del fallo. Se pueden utilizar técnicas de detección de errores.

b) Reparación del sistema. Los componentes hardware se pueden cambiar. Los componentes software se reparan haciendo una nueva versión. En algunos casos puede ser necesario reparar el sistema sin detenerlo.

4.3 Excepciones y sus manejadores

Son mecanismos de tratamiento de errores. Una excepción se puede definir como un suceso que causa la suspensión de la ejecución normal de un programa. Puede tratarse de un error o de una condición excepcional, que necesite un tratamiento especial. En un sistema en tiempo real fiable, no podemos permitir una terminación anormal en un programa. Debemos disponer de algún mecanismo para interceptar la excepción y responder adecuadamente. En el mejor caso, querríamos que el sistema siguiera operando sin pérdida significativa de funcionalidad. Si el sistema no puede recuperar su modo normal (caso de la avería de un periférico, por ejemplo), desearíamos una degradación paulatina, esto es, que que el sistema siguiera funcionando con una capacidad reducida, hasta su recuperación.

Cuando se produce un error, se alza la excepción correspondiente, lo que permitirá manejar la excepción. El mecanismo de manejo o tratamiento de excepciones puede ser considerado un mecanismo de recuperación directa: en lugar de retroceder a un estado previo, el control pasa al manejador, que se encargará de iniciar los procedimientos de recuperación. Aunque, como veremos más adelante, el manejo de excepciones también podrá ser usado para realizar recuperación inversa.

Hay una serie de requisitos generales que se deben dar en el tratamiento de excepciones:

a) El tratamiento debe ser sencillo de entender y de utilizar.

b) El código para el tratamiento de la excepción no debe oscurecer la comprensión de las operaciones normales del programa. La mezcla del código para el procesamiento normal con el código para el procesamiento de excepciones producirá un programa en general difícil de comprender y mantener, y por lo tanto un sistema menos fiable.

c) El mecanismo debería ser diseñado de forma que las sobrecargas (*overheads*) en tiempo de ejecución ocurran solamente durante el tratamiento de la excepción, y no durante la operación normal del programa.

d) El mecanismo debería permitir el tratamiento uniforme de las excepciones, tanto de las detectadas por el entorno como las detectadas por el programa.

e) El mecanismo de excepciones debería permitir programar acciones de recuperación.

4.4 Declaración y generación de excepciones en Ada

Ada permite la declaración de excepciones definidas por el usuario. Estas declaraciones pueden hacerse en cualquier parte donde pueda ser declarado un objeto, excepto como parámetros de un subprograma:

```
ATASCO_VALVULA : exception;
ERROR_PARIDAD : exception;
```

El lenguaje tiene varias excepciones estándar predefinidas:

- CONSTRAINT_ERROR: se genera, por ejemplo, cuando se intenta asignar un valor a un objeto fuera de su rango, se intenta un acceso a un índice no existente en un array, o cuando una operación numérica produce un resultado fuera del rango del tipo. Incluye el 'divide by zero error'.
- STORAGE_ERROR: se intanta asignar memoria dinámica y no hay suficiente.
- PROGRAM_ERROR: un intento de saltarse una estructura de control. Por ejemplo, llegar a la sentencia END de una función sin haber ejecutado un RETURN.
- TASKING_ERROR: errores durante la comunicación entre tareas (tasks).

```
package Ada.Exceptions is
   type Exception_Id is private;
   Null_Id : constant Exception_Id;
   function Exception_Name (X : Exception_Id) return String;

   type Exception_Occurrence is limited private;
   type Exception_Occurrence_Access is access all Exception_Occurrence;
   Null_Occurrence : constant Exception_Occurrence;

   procedure Raise_Exception(E : in Exception_Id; Message : in String :=
"");
   function   Exception_Message      (X  :  Exception_Occurrence)  return
String;
   procedure Reraise_Occurrence   (X : Exception_Occurrence);
   function   Exception_Identity(X    :    Exception_Occurrence)    return
Exception_Id;
   function Exception_Name     (X : Exception_Occurrence) return String;
   --  Same as Exception_Name (Exception_Identity (X))
   function  Exception_Information  (X  :  Exception_Occurrence)  return
String;
   procedure Save_Occurrence (Target :    out Exception_Occurrence;
      Source : in     Exception_Occurrence);

   function Save_Occurrence (Source : in Exception_Occurrence)
      return Exception_Occurrence_Access;
private
     ...-- no especificado por el lenguaje
end Ada.Exceptions;
```

Las versiones de Ada incluyen el siguiente package con las correspondientes declaraciones de excepciones.

```
package Ada.IO_Exceptions is

   Status_Error : exception;
   Mode_Error   : exception;
```

```
   Name_Error   : exception;
   Use_Error    : exception;
   Device_Error : exception;
   End_Error    : exception;
   Data_Error   : exception;
   Layout_Error : exception;

end Ada.IO_Exceptions;
```

- STATUS-ERROR: Se alza si tratamos de hacer una operación en un fichero cerrado, o si tratamos de abrir un fichero ya abierto.
- MODE_ERROR: Se alza si tratamos de leer de un fichero con modo OUT_FILE, o de escribir en un fichero con modo IN_FILE.
- NAME_ERROR: Algo erróneo en el parámetro NAME en los procedures CREATE u OPEN.
- USE_ERROR: Cuando tratamos de aplicar una operación que no está permitida sobre el fichero físico (leer de una impresora, por ejemplo).
- DEVICE_ERROR: Dispositivo averiado o apagado.
- END_ERROR: Intento de leer después del final de un fichero.
- DATA_ERROR: El procedure GET no puede interpretar datos como valores del tipo deseado.
- LAYOUT_ERROR: Algo erróneo en las salidas de texto (package TEXT_IO), por ejemplo una llamada a PUT con un número de caracteres en el STRING que no caben en la línea.

- Sentencia **raise**

Suspende el procesamiento normal y pasa el control al manejador de excepciones apropiado:

raise NOMBRE_EXCEPCION;

4.5 Manejadores de excepciones en Ada

Cuando ocurre una excepción, si no se dispone del tratamiento adecuado, el procesamiento se suspende y el control retorna al sistema operativo.

Ada permite escribir manejadores de excepciones para los dos tipos: las predefinidas y las definidas en el programa. Estos tratamientos de excepciones pueden aparecer al final de un bloque, o al final del cuerpo de un subprograma, package o task.

```
bloque:
    ...
    declare  -- principio del bloque
        ...
    begin
        ...
        -- sentencias
     exception
         ...
             -- manejadores de excepciones
     end;    --fin del bloque
     ...
```

subprograma:

```
procedure NOMBRE(....) is
   ...
   -- declaraciones
begin
    ...
    -- sentencias
exception
    ...
   -- manejadores de excepciones
end NOMBRE;
```

package:

```
package body NOMPAC is
   ...
   -- declaraciones y cuerpos de subprogramas
begin
   ...
   -- inicialización
exception
   ...
   --- manejadores de excepciones
end NOMPAC;
```

Su forma es similar a la de una sentencia case. Ejemplo:

```
...
declare
   LOW_FLUID_LEVEL: exception;

begin
   ...

exception
   when LOW_FLUID_LEVEL =>
                       OPEN_VALVE;
                       SOUND_ALARM;
   when NUMERIC_ERROR =>
                       CLOSE_VALVE;
   when OTHERS =>
                        LOG_UNKNOW_ERROR;
end;
```

Si por ejemplo la excepción LOW_FLUID_LEVEL es alzada en el bloque, el manejador de excepciones respondería ejecutando las sentencias (procedures) OPEN_VALVE y luego SOUND_ALARM. Una vez el manejador completa su operación, el control no retorna al punto donde se produjo la excepción; finaliza la ejecución de la unidad en la que se encuentra el manejador de la excepción (modelo de terminación).

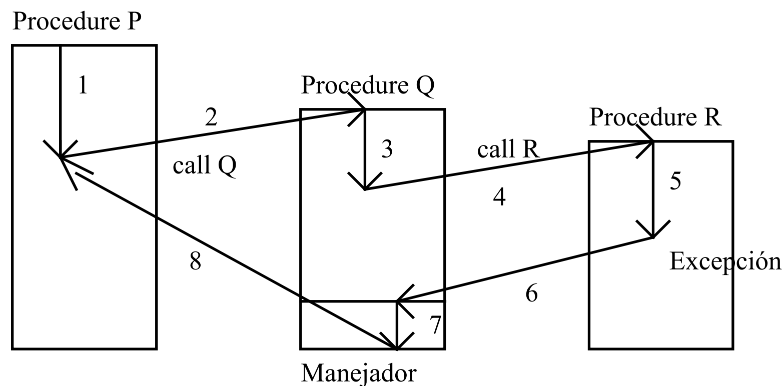
Si una unidad no proporciona un tratamiento para una excepción, ésta se propaga dinámicamente: la unidad termina su ejecución y la excepción se alza en el punto donde la unidad fue invocada. En el caso de un bloque, se buscará un manejador en la unidad que lo contiene:

```
procedure MAIN is
begin
   ...
   declare
      LOCAL_ERROR: exception;
```

```
        begin
           ...
        exception
           when LOCAL_ERROR => DO_SOMETHING;
        end;
        ...
     exception
        when CONSTRAINT_ERROR => DO_SOMETHING_ELSE;
        when NUMERIC_ERROR => DO_SOMETHING_MORE;
     end MAIN;
```

Si la excepción LOCAL_ERROR se genera durante la ejecución del bloque, el manejador de la excepción definido en el bloque se ocupará de ella (ejecutará la orden DO_SOMETHING). A continuación, el control pasará al final del bloque.

Si durante la ejecución del bloque se alza la excepción CONSTRAINT_ERROR, ésta se propaga a la unidad que lo contiene (procedure MAIN). Su parte de código para el manejo de excepciones se encargará del tratamiento de dicha excepción (se ejecutará la orden DO_SOMETHING_ELSE) y finalizará la ejecución de MAIN. Si la excepción PROGRAM_ERROR se alza durante la ejecución del bloque (o del subprograma MAIN), no se encontrará un tratamiento adecuado de la misma. Así, la excepción se propagará hasta el sistema operativo.



En el caso de subprogramas, si durante su ejecución se produce una excepción y el subprograma no tiene un manejador adecuado, la excepción se alzará en el punto de llamada del subprograma (por tanto, en la unidad que llama al subprograma, no en la que lo define).

Si la excepción ocurre en la parte declarativa del subprograma, la excepción se alzará en el punto de llamada del subprograma, independientemente de que éste incluya el manejador adecuado.

Los manejadores de excepciones definidos en el package sólo serán utilizados por las sentencias incluídas en la inicialización. Una excepción que se alce en el cuerpo de un subprograma del package y que no sea tratada en el mismo subprograma se propagará hasta el punto de llamada del subprograma; sólo será tratada por los manejadores de excepciones que aparecen en el package si la llamada se efectúa desde las sentencias incluídas en la inicialización del mismo.

4.6 Modelos de retorno

Una vez que la excepción ha sido tratada, existen 3 posibles acciones:

a) Modelo de reanudación. El manejador realiza el tratamiento oportuno y devuelve el control a la sentencia siguiente a aquella en la que se produjo el error.

b) Modelo de terminación. Asumido por el lenguaje Ada, y presentado en el apartado anterior. Una vez completadas las acciones del manejador, el bloque o subprograma que contiene a dicho manejador termina su operación, y el control pasa al bloque o subprograma que lo había invocado (o al sistema operativo, caso de ser el módulo más externo).

c) Modelo híbrido. Dentro del manejador de la excepción se puede emitir una sentencia de reanudación (con lo que estaríamos en el caso a), o no emitirla (caso b).

A primera vista, la solución b) no parece aconsejable para la recuperación ante errores. El mecanismo a) o el c) parecen ideales: el manejador de excepciones puede reanudar el cálculo como si nada hubiera pasado. En realidad, este modelo presenta graves inconvenientes. En primer lugar, puede ser muy difícil, o imposible, restaurar los cálculos que se estaban haciendo cuando se alzó la excepción: la llamada al manejador puede sobreescribir los registros que estaban siendo utilizados. En segundo lugar, los programas pierden claridad. Suponer el siguiente fragmento:

```
SUMA:=A+B;

X:=SUMA-A;

-- a partir de aquí, asumimos que X = B.
```

Si se produce un overflow en la evaluación de A+B, el manejador de la excepción puede asignar a SUMA otro valor, y luego retornar a la siguiente sentencia (X:=SUMA-A), de forma que X ≠ B.

El modelo de terminación no presenta estos problemas. El manejador de la excepción sustituye al resto del bloque o subprograma abortado. Las acciones de recuperación de errores, que pueden parecer la carencia de este método, también pueden programarse con este modelo, como veremos a continuación.

4.7 Recuperación ante errores

Uno de los requisitos de un mecanismo de tratamiento de errores para un sistema en tiempo real es que facilite la posibilidad de programar la recuperación ante errores.

Un modo de recuperación consiste en utilizar un manejador para proporcionar unos valores de resultado válidos o alternativos cuando falle un cálculo:

```
function MULTIPLICAR(X,Y:INTEGER) return INTEGER is
begin
   return X*Y;
exception
   when NUMERIC_ERROR => if X/Y <0 then
-- esta excepción está desfasada en Ada 95
-- ahora es englobada por CONSTRAINT_ERROR
                          return MIN_INT;
                       else
                   return MAX_INT;
                       end if;
end MULTIPLICAR;
```

Otro ejemplo. Caso de que exista un problema de comunicación entre tareas; podemos tratar de enviar el mensaje por una ruta alternativa:

```
...
begin
```

```
      SEND_MESSAGE_TO_PATH_1(MESSAGE);
   exception
      when TASKING_ERROR => SEND_MESSAGE_TO_PATH_2(MESSAGE);
   end;
```

4.7.1 Bloques de recuperación y excepciones

Otro modo de recuperación es intentar repetir una operación después de producirse una excepción. La técnica para ello consiste en declarar un bloque local que encapsule el algoritmo, y colocar el bloque dentro de un bucle que repita la operación:

```
      type RESPONSE is (UP, DOWN, LEFT, RIGHT);
      USER_REQUEST: RESPONSE;
      with TEXT_IO; use TEXT_IO;
      package RESP_IO is new ENUMERATION_IO(RESPONSE);
      use RESP_IO;
      ...
   begin
      ...
      loop       --en caso de funcionamiento correcto se ejecutará solo una
vez
         begin
            put(": ");
            get(USER_REQUEST);
            new_line;
            exit;          -- solo se ejecuta si todo fue bien
         exception
            when DATA_ERROR => new_line;
                   put_line("Invalid response. Enter only UP, DOWN, LEFT o
RIGHT);
         end;
      end loop;

   o bien

    begin
      ...
      for I in 1..5 loop
         begin
            put(": ");
            get(USER_REQUEST);
            new_line;
            exit;
         exception
            when DATA_ERROR => if I < 5 then
                   put_line("Invalid response. Enter only UP, DOWN, LEFT o
RIGHT);
                           else
                      put_line("The assumed response is UP");
                      USER_REQUEST:=UP;
                           end if;
         end;
      end loop;
```

Ejemplo:

```
   package RobustInput is
```

```
  -- Package for getting numeric input robustly.

  procedure Get (Item : OUT Integer;
                 MinVal : IN Integer;
                 MaxVal : IN Integer);
  -- Gets an integer value in the range MinVal..MaxVal from the terminal
  -- Pre: MinVal and MaxVal are defined
  -- Post: MinVal <= Item <= MaxVal

  procedure Get (Item : OUT Float;
                 MinVal : IN Float;
                 MaxVal : IN Float);

  -- Gets a float value in the range MinVal..MaxVal from the terminal
  -- Pre: MinVal and MaxVal are defined
  -- Post: MinVal <= Item <= MaxVal
end RobustInput;

with Ada.Text_IO;  use Ada.Text_IO;
package body RobustInput is
   package IIO is new Integer_IO(Integer); use IIO;
   package FIO is new Float_IO(Float); use FIO;

  procedure Get (Item : OUT Integer;
                 MinVal : IN Integer;
                 MaxVal : IN Integer) is

  -- Gets an integer value in the range MinVal..MaxVal from the terminal
  -- Pre: MinVal and MaxVal are defined
  -- Post: MinVal <= Item <= MaxVal

    subtype TempType IS Integer range MinVal..MaxVal;
    TempItem : TempType;     -- temporary copy of MinVal

  begin -- Get
    loop
      begin      -- exception handler block
        Put(Item => "Enter an integer between ");
        Put(Item => MinVal, Width => 0);
        Put(Item => " and ");
        Put(Item => MaxVal, Width => 0);
        Put(Item => " > ");
        Get(Item => TempItem);
        Item := TempItem;
        exit;    -- valid data
      exception  -- invalid data
        when Constraint_Error =>
          Put ("Value entered is out of range. Please try again.");
          New_Line;
          Skip_Line;
        when Text_IO.Data_Error =>
          Put ("Value entered not an integer. Please try again.");
          New_Line;
          Skip_Line;
      end;       -- exception handler block
    end loop;
```

```
    -- assert: Item is in the range MinVal to MaxVal
  end Get;

  procedure Get (Item : OUT Float;
                 MinVal : IN Float;
                 MaxVal : IN Float) IS

  -- Gets a float value in the range MinVal..MaxVal from the terminal
  -- Pre: MinVal and MaxVal are defined
  -- Post: MinVal <= Item <= MaxVal
    subtype TempType IS Float RANGE MinVal..MaxVal;
    TempItem : TempType;     -- temporary copy of MinVal

  begin -- Get
    loop
      begin       -- exception handler block
        Text_IO.Put(Item => "Enter a floating-point value between ");
        FIO.Put(Item => MinVal, Fore=> 1, Aft => 2, Exp => 0);
        Text_IO.Put(Item => " and ");
        FIO.Put(Item => MaxVal, Fore=> 1, Aft => 2, Exp => 0);
        Text_IO.Put(Item => " > ");
        FIO.Get(Item => TempItem);
        Item := TempItem;
        exit;     -- valid data
      exception   -- invalid data
        when Constraint_Error =>
          Text_IO.Put  ("Value  entered  is  out  of  range.  Please  try
again.");
          Text_IO.New_Line;
          Text_IO.Skip_Line;
        when Text_IO.Data_Error =>
          Text_IO.Put("Value  entered  not  floating  point.  Please  try
again.");
          Text_IO.New_Line;
          Text_IO.Skip_Line;
      end;        -- exception handler block
    end loop;
    -- assert: Item is in the range MinVal to MaxVal
  end Get;
end RobustInput;

with RobustInput;
procedure TestRobustInput is
  subtype SmallInt  is Integer range -10 ..10;
  subtype LargerInt is Integer range -100..100;
  subtype SmallFloat  is Float range -10.0 ..10.0;
  subtype LargerFloat is Float range -100.0..100.0;
  Small :  SmallInt;
  SmallF :  SmallFloat;
  Larger : LargerInt;
  LargerF : LargerFloat;
begin
  RobustInput.Get(Small,SmallInt'First,SmallInt'Last);
  RobustInput.Get(Larger,LargerInt'First,LargerInt'Last);
  RobustInput.Get(SmallF,SmallFloat'First,SmallFloat'Last);
  RobustInput.Get(LargerF,LargerFloat'First,LargerFloat'Last);
```

```
end TestRobustInput;
```

4.7.2 Ultimas voluntades (*last wishes*)

Además de intentar asegurarse de que un subprograma o bloque devuelva resultados útiles, la recuperación ante errores implica también contrarrestar los efectos perniciosos causados como consecuencia de un error, antes de que el daño pueda extenderse a otras partes del sistema. Si se produce una excepción en un módulo y el manejador local desconoce su significado, la excepción se propagará al módulo llamador. Sin embargo, puede ser necesario que el módulo en el que se alza la excepción realice alguna operación para no propagar los daños, antes de su terminación (lo que se conoce como sus 'últimas voluntades'). Por ejemplo, consideremos el siguiente diseño de programa, en el que un procedure ANALIZAR utiliza dos procedures, PROCESAR_FICHERO y CALCULAR, para analizar el contenido de un fichero de datos:

```
procedure ANALIZAR (F: in out FILE_TYPE; RESULT: out DATOS) is
   E: exception;
   procedure CALCULAR(X: in DATOS; Y: out DATOS) is
   begin
      ...
      if ... then raise E;
      end if;
      ...
   end CALCULAR;

   procedure PROCESAR_FICHERO(F: in out FILE_TYPE; Y: out DATOS) is
      X: DATOS;
   begin
       abrir fichero(F)
       leer datos(F,X)
       calcular(X,Y) -- alza la excepción E
       cerrar fichero(F)
   exception
      when others => cerrar fichero(F)
                raise; -- vuelve a alzar la última excepción
   end PROCESAR_FICHERO;

begin
   ...
   PROCESAR_FICHERO(F,RESULT);
   ...
exception
   when E => ...
   ...
end ANALIZAR;
```

CALCULAR genera la excepción E, pero no dispone de un manejador local para la misma. Cuando se genera esta excepción, el procedure PROCESAR_FICHERO debería abandonar su labor y pasar el control al manejador de E, que se encuentra en ANALIZAR. De esta forma, el fichero F quedaría abierto. Para evitarlo, PROCESAR_FICHERO se diseña de modo que intercepte cualquier excepción y que cierre el fichero antes de volver a levantar la excepción en el módulo llamador.

Ejercicio:

En una aplicación de control, un gas es calentado en una cámara cerrada. La cámara está rodeada por un enfriador que reduce la temperatura del gas. Hay también una válvula que, cuando está abierta, libera gas a la atmósfera. La operación del proceso está controlada por un package en Ada, cuya especificación se da abajo. Por razones de seguridad, el package reconoce varias situaciones de error. La excepción HEATER_STUCK_ON es alzada por el procedure HEATER_OFF cuando es incapaz de apagar el elemento calentador. La excepción TEMPERATURE_STILL_RISING es levantada por el procedure INCREASE_COOLANT si es incapaz de bajar la temperatura del gas aumentando el flujo del enfriador. Finalmente, la excepción VALVE_STUCK es generada por el procedure OPEN_VALVE si no puede abrir la válvula que libera el gas.

```
package TEMPERATURE_CONTROL is
   HEATER_STUCK_ON, TEMPERATURE_STILL_RISING, VALVE_STUCK:exception;

   procedure HEATER_ON;
   --enciende el calentador

   procedure HEATER_OFF;
   --apaga el calentador
   --genera la excepción HEATER_STUCK_ON

   procedure INCREASE_COOLANT;
   --Incrementa el flujo del enfriador que rodea la cámara
   --hasta que la temperatura alcanza un valor seguro
   --genera la excepción TEMPERATURE_STILL_RISING

   procedure OPEN_VALVE;
   --abre la válvula que libera gas, evitando así
   --una explosión
   --alza VALVE_STUCK

   procedure PANIC;
   --hace sonar una alarma, y llama a los bomberos,
   --hospital y servicios policiales
end TEMPERATURE_CONTROL;
```

Escribir un procedure que intente apagar el calentador en la cámara. Si el calentador está atascado, debe incrementar el flujo de enfriador. Si la temperatura sigue aumentando, debe abrir la válvula para liberar gas. Si ésta falla, debe hacer sonar la alarma e informar a los servicios de emergencia.

Escribir el package TEMPERATURE_CONTROL, de modo que nos permita testear nuestro procedure. Estos procedures pueden emitir un mensaje señalando lo que están haciendo.

Solución posible:

```
package TEMPERATURE_CONTROL is
   HEATER_STUCK_ON, TEMPERATURE_STILL_RISING, VALVE_STUCK:exception;

   procedure HEATER_ON;
   --turn on heater

   procedure HEATER_OFF;
   --turn off heater
   --raises HEATER_STUCK_ON

   procedure INCREASE_COOLANT;
   --Causes the flow of coolant which surrounds the chamber
   --to increase until the temperature reaches a safe level
   --raises TEMPERATURE_STILL_RISING
   procedure OPEN_VALVE;
   --opens a valve to release some of the gas thereby avoiding
   --an explosion
   --raises VALVE_STUCK

   procedure PANIC;
   --sounds an alarm and calls the fire,
   --hospital and police services

end TEMPERATURE_CONTROL;
with Ada.Text_IO; use Ada.Text_IO;
package body TEMPERATURE_CONTROL is

   procedure HEATER_ON is
   --turn on heater
   begin
      PUT_LINE("TURN ON HEATER");
   end HEATER_ON;

   procedure HEATER_OFF is
   --turn off heater
   --raises HEATER_STUCK_ON

      OPC: character;
   begin
      PUT_LINE ("TURN OFF HEATER");
      PUT("Is it OK (Y/N)? ");
      GET(OPC);
      if (OPC="Y") or (OPC="y")  then
         PUT_LINE("OK");
      else
         raise HEATER_STUCK_ON;
      end if;
   end HEATER_OFF;
   procedure INCREASE_COOLANT is
   --Causes the flow of coolant which surrounds the chamber
   --to increase until the temperature reaches a safe level
   --raises TEMPERATURE_STILL_RISING
```

```
        OPC: character;
        begin
           PUT_LINE("INCREASE COOLANT");
           PUT("Is it OK (Y/N)? ");
           GET(OPC);
           if (OPC="Y") or (OPC="y")  then
              PUT_LINE("OK");
           else
              raise TEMPERATURE_STILL_RISING;
           end if;
        end INCREASE_COOLANT;
        procedure OPEN_VALVE is
        --opens a valve to release some of the gas thereby avoiding
        --an explosion
        --raises VALVE_STUCK
        OPC: character;
        begin
           PUT_LINE("OPEN VALVE");
           PUT("Is it OK (Y/N)? ");
           GET(OPC);
           if (OPC="Y") or (OPC="y")  then
              PUT_LINE("OK");
           else
              raise VALVE_STUCK;
           end if;
        end OPEN_VALVE;
        procedure PANIC is
        --sounds an alarm and calls the fire,
        --hospital and police services
        begin
           PUT_LINE("ALARM. CALL THE HOSPITAL, POLICE, FIRE");
        end PANIC;
     end TEMPERATURE_CONTROL;
     with TEMPERATURE_CONTROL; use TEMPERATURE_CONTROL;
     with Ada.Text_IO; use Ada.Text_IO;
     procedure APAGARV1 is
        type ACCION is (TURN_OFF, INCREASE_FLOW, OPEN_V, ALARM);
     --cada acción corresponde a un módulo de recuperación: primaria
(TURN_OFF),
     --secundaria (INCREASE_FLOW), ...
     begin
     for A in ACCION loop
           begin
              case A is
                 when TURN_OFF => HEATER_OFF; exit;
                 when INCREASE_FLOW => INCREASE_COOLANT; exit;
                 when OPEN_V => OPEN_VALVE; exit;
                 when ALARM => PANIC;
              end case;
           exception
              when HEATER_STUCK_ON =>
                  PUT_LINE("HEATER STUCK ON");
                  PUT_LINE("NEXT ACTION: INCREASE COOLANT");
              when TEMPERATURE_STILL_RISING =>
                  PUT_LINE("TEMPERATURE STILL RISING");
                  PUT_LINE("NEXT ACTION: OPEN VALVE");
```

```
            when VALVE_STUCK =>
                PUT_LINE("VALVE STUCK");
                PUT_LINE("NEXT ACTION: PANIC");
        end;
        end loop;
  end APAGARV1;
```

```
with TEMPERATURE_CONTROL; use TEMPERATURE_CONTROL;
with TEXT_IO; use TEXT_IO;
procedure APAGARV2 is
   type ACCION is (TURN_OFF, INCREASE_FLOW, OPEN_V, ALARM);
begin
HEATER_OFF;
exception
   when HEATER_STUCK_ON =>
      begin
         PUT_LINE("HEATER STUCK ON");
         PUT_LINE("NEXT ACTION: INCREASE COOLANT");
         INCREASE_COOLANT;
      exception
         when TEMPERATURE_STILL_RISING =>
            begin
               PUT_LINE("TEMPERATURE STILL RISING");
               PUT_LINE("NEXT ACTION: OPEN VALVE");
               OPEN_VALVE;
            exception
               when VALVE_STUCK =>
                  PUT_LINE("VALVE STUCK");
                  PUT_LINE("NEXT ACTION: PANIC");
                  PANIC;
            end;
      end;
end APAGARV2;
```

BIBLIOGRAFIA

En la elaboración de este tema se han consultado los siguientes libros:

[1] A. Burns, A. Wellings; 'Real-Time Systems and Programming Languages, 2nd. Ed.'; ed. Addison-Wesley (1996). Caps. 5,6

[2] S.J. Young; *Lenguajes en Tiempo Real*; Ed. Paraninfo 87. Cap. ...

[3] G. Booch; Software *Engineering with Ada*; ed. Addison-Wesley (1994). Cap. 17