ソフトウェア基礎論　３３－５
プログラミング言語　２３－５
（1989. 12. 8）

並列オブジェクト指向言語 Concurrent COBにおける例外処理と割り込み処理

細川　馨
日本アイ・ビー・エム株式会社　東京基礎研究所
102 東京都千代田区三番町5-19

あらまし　並列オブジェクト指向言語Concurrent COBにおける例外処理と割り込み処理について述べている．例外処理はターミネーションモデルに基づている．このモデルでは例外処理後，例外を受けたプログラムの制御フローは停止する．また，割り込み処理はリザンプションモデルに基づいている．割り込み処理後，割り込みを受けたプログラムの制御フローは継続される．この言語では，オブジェクトの継承を使ってデフォルトの例外処理ハンドラを記述することができる．さらに，関数呼出しと同期通信の際に起きる例外の伝播を統一的に扱うことができる．

Exception and Interrupt Handling in Concurrent COB

Kaoru Hosokawa
IBM Research, Tokyo Research Laboratory
5-19, Sanbancho, Chiyoda-ku
Tokyo 102, Japan

Abstract The design of exception and interrupt handling for a concurrent object oriented programming language, called Concurrent COB, is described. Exception handling is based on a termination model. In this model, after the exception is handled, the program that caught the exception terminates. Interrupt handling is based on a resumption model. Here, the interrupted program resumes execution. This language allows the specification of default handlers using inheritance.The propagation of exceptions are dealt with uniformly for sequential function calls and for synchronous communications between concurrent objects.

Introduction

A program written in a programming language without exception handling facilities must use gotos and flags. Although it is possible to use gotos in a structured manner, because gotos are too general, there is a tendency for programs with gotos to become difficult to understand. Thus to improve the readability of COB programs, the addition of exception handling support is considered.

Concurrent COB (C with OBjects) [Hosokawa and Kamimura (89)] is a concurrent object oriented programming language. One of the requirements of Concurrent COB (CCOB) is for programmers to able to write systems software. Hardware interrupt handling is often required when writing systems software, especially when writing device drivers. To help the programmers in this respect, interrupt handling is being considered as an inclusion to CCOB.

Exception handling for COB has already been designed [Yokouchi (89)]. Since exceptions and interrupts have some commonality, for example, both require a handler to handle the exception or interrupt, an attempt is made to incorporate interrupt handling into the design of COB exception handling. The design is also extended further to cope with concurrent programs dealing with exceptions and interrupts.

Concurrent COB

CCOB is an extension of COB with concurrency constructs. A process is introduced as a special class; its instance is a concurrent object that executes in parallel to the creator of the object. Synchronous communication is provided as a means of communication. Here is a buffer consumer system written in CCOB.

```
process decl consumer {
   void  init(void);
   void  put(int);
};

process decl producer {
   void  init(void);
};

consumer cons;
producer prod;

process impl consumer {
   int   i;
implementation:
   void init(void) {
      for (;;) {
         accept put(int);
         printf("%d\n", i);
      }
   }
   void put(int x) {
      i = x;
   }
};

process impl producer {
implementation:
   void init(void) {
      int   i  = 0;
      for (;;) {
         cons->put(i++);
```

```
        }
    }
};
void main(void) {
    cons = new();
    prod = new();
}
```

The accept statement signifies that the object is ready to engaged in a synchronous communication of the specified function. When both the requester and accepter of function are ready, then synchronous communication occurs. Until then the objects are in a waiting state. During synchronous communication, the accepted function is executed. When the execution of the accepted function completes, the synchronous communication terminates. The objects are in a synchronous communication state when they are engaged in a synchronous communication. Another state that a concurrent object can be in is called the autonomous state. In this state, a concurrent object is executing in parallel relative to other objects -- this is when it is not in neither waiting nor synchronous communication state.

COB Exception Handling

The following features characterizes COB exception handling.

- The termination model of exception handling is adopted. This means that when an exception is raised during the execution of a statement and upon completion of the exception handler, the statement terminates and does not resume execution.
- Exception handlers can be attached to statements using the try statement.

Try Statement

Exception handlers are attached to statements using the try statement. Here is an example,

```
try {
    y = 0;
    x = x / y;
}
with {
    zero_divide() =>
        fprintf(stderr,
            "zero divide\n");
    floating_point() =>
        fprintf(...);
};
```

The statement between `try` and `with` is called a monitored statement. The exception handlers for the monitored statement are defined after the `with`. In this example, `zero_divide` and `floating_point` are names of exceptions and statements after the `=>` are their respective handlers.

When an exception statement is raised, the following actions are taken.

1. The execution of the monitored statement terminates.
2. The `with` portion is searched for a matching exception name and when found the handler is executed.
3. Upon completion of the handler, the execution of the try statement completes.

In this example, a `zero_divide` exception is raised from `x = x / y;` and results in output of "`zero divide`".

When a handler is not found, the exception is propagated to its surrounding try statement.

Propagation of Exceptions

When the propagation of an exception reaches the outer most try statement and still no handler is found, the propagation of that exception continues to the caller of that function. This allows the caller of the function to handle exceptions. For example,

```
class decl calculator {
   int divide(int, int)
      raises zero_divide
         (void);
   ...
};
class impl calculator {
implementation:
   int div(int x, int y)
      raises zero_divide
         (void) {
         return x / y;
   }
   ...
};
calculator calc = new();
y = 0;
try
   calc->divide(x, y);
with {
   zero_divide() =>
      fprintf(stderr,
         "Oops!\n");
};
```

In this example, a `zero_divide` exception is raised within the `divide` function. Since no handler exists, the exception is propagated to the caller of the function. The call to `divide` is a monitored statement with the appropriate handler and so the handler is executed with the output of "`Oops!`".

COB Interrupt Handling

When a hardware interrupt occurs, the usual action taken is to suspend the current execution of the program, execute the interrupt handler and upon completion of the handler, resume the execution of the interrupted program. The original COB exception handling mechanism is based on a termination model, however the hardware interrupt handling mechanism requires a resumption model. The resumption model is a model where upon completion of the handler, the interrupted program is

resumed instead of terminated. Figure 1. shows the different models.

One way to approach the problem is to combine the two models of exceptions into one by allowing the handler to specify whether termination or resumption of the interrupted program is required [Goodenough (75)]. However, in this model, it is possible to write exception and interrupt handlers with a mixture of termination and resumption controls. One can imagine the difficulties in understanding such handlers when analysing propagation of exceptions for example. Thus the current design deals with exception handling and interrupt handling separately.

Interrupt Handling

Interrupt handlers are attached to objects. The reasoning for this is that, the point at which an interrupt can occur is not clearly defined as in exceptions and thus the need to attached interrupt handlers to statements is not great. For example, the occurrence of a zero divide exceptions can be narrowed to a divide statement, but a keyboard interrupt occurrence spans the whole program.

The original COB exception handling design does not allow one object to raise an exception of another object. Considering hardware interrupts, it is usually the case where one object

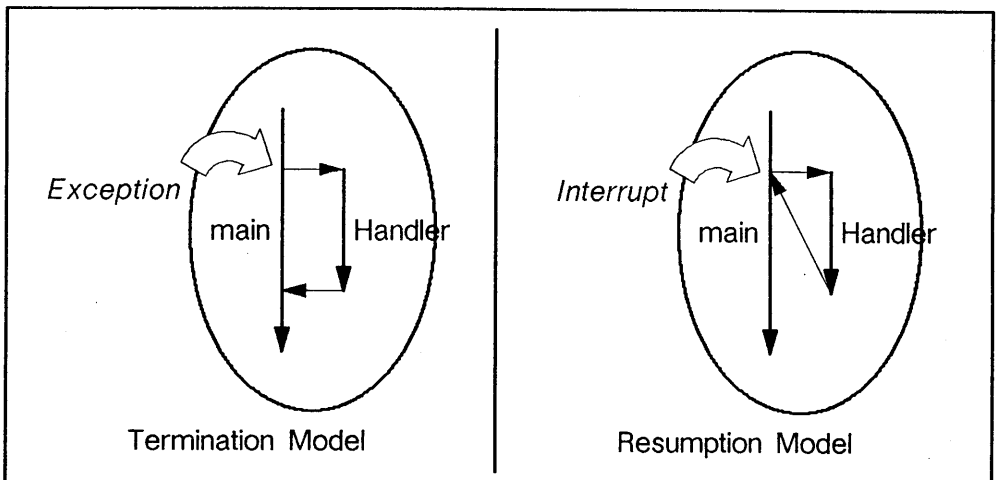

Figure 1. Termination and Resumption Models

raises an interrupt of another object, for example, a keyboard object may raise an interrupt of a terminal object, signifying that a key is depressed on the keyboard, and that the terminal object should retrieve that key value. Thus a way of raising exception/interrupt of other objects is added.

The raising of an exception/interrupt is considered as a special function call. Thus the form of declaration and definition is similar to that of a function of an object. It is possible to declare public and private exceptions/interrupts and exceptions/interrupts can be publicly and privately inherited. Public exceptions/interrupts are visible to the user of an object and private exceptions/interrupts are visible only within the object itself. Similarly, public inheritance is visible to the user and private inheritance is not. This means that exceptions/interrupts may be publicly inherited and visible to the user or they may be privately inherited and hidden from the user. Here is an example,

```
class decl time {
   void  reset(void);
   void  inc(void);
};

class decl
default_exception {
exception:
   void  hangup(void);
   void  interrupt(void);
   void  quit(void);
   ...
};

class decl stopwatch <
default_exception {
   void  init(void);
interruption:
   time  stop(void);
};
```

default_exception is publicly inherited by stopwatch. This means that the exception functions hangup, interrupt, quit, etc. can all be raised by the user of the stopwatch object. Note also that default handlers for system defined exceptions are used by inheriting them. The implementation of stopwatch is as follows.

```
class impl stopwatch {
   time  t  = new();
   int      go;
implementation:
   void init(void) {
      disable stop();
      go = 1;
      enable stop();
      while (go)
         t->inc();
      disable stop();
   }
   time stop(void) {
      go = 0;
      return t;
   }
};
```

The essence of stopwatch is to continuously increment time and stop when an interrupt is raised and return the value of the current time. Enable and disable allows and disallows the raising of the exception/in-

terrupt, respectively. If the interrupt function is called when that interrupt is disabled, the interrupt function simply returns as if an empty body has been executed.

Concurrent COB Exception Handling

The extended COB exception handling design is now applied to concurrent programs.

Here is an example of a simple terminal handler. The system consists of three concurrently executing objects, signified by the keyword `process`. The controller accepts a `get` operation whenever the buffer is not empty and waits for a synchronous communication of the `get` operation. When a key is depressed on the keyboard, the key value is stored in `c` and the `keyboard` object raises a `put` interrupt of the controller. When the put interrupt is raised, the key value is stored into `buffer` and also displayed onto the screen.

```
process decl keyboard {
   void  init(void);
   void  start(controller);
};

process decl screen {
   void  put(char);
};

process decl controller {
   void  init(void);
   char  get(void);
interruption:
   void  put(char);
};

process impl keyboard {
   controller  cont;
   char  c;
implementation:
   void init(void) {
      accept
         start(controller);
      for (;;) {
         /* when a key is
            hit c holds its
            value */
         cont->put(c);
      }
   }
   void start(controller
         cont) {
      self->cont = cont;
   }
};

process impl controller {
   int   size  = 0;
   char  buffer[512];
   keyboard key   = new();
   screen   scr   = new();
implementation:
   void init(void) {
      key->start(self);
         for (;;) {
            when (size > 0)
               accept
               get(void);
         }
      }
   }
   char get(void) {
      char  c;
      c = buffer[size];
      size--;
      return c;
   }
   void put(char c) {
      size++;
      buffer[size] = c;
```

```
        scr->put(c);
    }
};
```

The actions taken when an exception occurs in a concurrent object is similar to that of the sequential object. Since exceptions can be raised from several concurrent objects, the exceptions are queued and processed sequentially.

The state of an concurrent object has an effect on exceptions are handled. When an exception is raised when a concurrent object is in a synchronous communication state and termination of the function is required, the accepted function is terminated and the new state is a ready state. When an object is in a waiting state and termination is required, the accept function request or synchronous communication request is removed from the scheduler, accept function is terminated and the new state of the object is the ready state. When the object is in the autonomous state and termination is requested, the statement that was executed when the exception was raised is terminated and the object state is made ready. A summary of the termination rules are given in table 1. The accept statement is undone, since termination without undoing accept statements will lead to deadlock because the requestor may wait for a terminated accept statement.

Propagation of Exceptions

Recall that in the sequential case, when a handler for an exception is not found within a function, the exception is propagated to the caller of

current state	new state	action
communi-cation	ready	terminate accepted function
waiting	ready	remove scheduling request terminate accept statement
autonomous	ready	terminate statement

Table 1. Termination rules

the function. This is the same for the concurrent case. When an exception is raised during a synchronous communication and no handler is found at the callee, the exception is propagated to its caller of the synchronous communication. However, when an exception is raise when the concurrent object is not engaged in a synchronous communication and when the handler is not found, the exception is propagated to the creator of the concurrent object, since no caller is involved.

A concurrent object may engage in a nested synchronous communication. This is a situation where an accepted function contains an accept statement. When an exception is raised and the exception is propagated, all synchronous communication must be unraveled. This means that the accept functions must be terminated and exceptions must be raised to the caller of the accept functions. This mechanism is required, since leaving an accepted function that will no longer be executed to completion will lead to deadlock.

Comparisons to Other Languages

In Ada [Ichbiah et al. (79)] an interrupt is considered as an entry call to a task and not as an exception. This has the advantage that a select statement can be used to choose between interrupts, but complicates the meaning of the select statement when normal entry calls and interrupt entry calls are specified together -- priorities, specifying the order of accepting an entry call, are attached to accept statements which reduce the clarity of the select statement.

Exceptions in ABCL/1 [Yonezawa and Ichisugi (89)] are supported by passing an object which handles exceptions. Since exceptions are handled by an exception object, modularity of the exception handler is improved, but it seems difficult to pass all the necessary exception information to the exception object since the exception object is usually out of scope from where the exception is raised.

Acknowledgements

I would like to thank H. Yokouchi for his COB exception handling design and T. Kamimura for his valuable comments and discussions.

References

[Goodenough (75)] Goodenough, J. B., *Exception Handling: Issues and a Proposed Notation*, Communications of the ACM, Vol. 18, No. 12, December 1975, pp. 683 - 696.

[Hosokawa and Kamimura (89)] Hosokawa, K. and Kamimura, T.,

Concurrent Programming in COB, distributed at the 2nd UK/Japan Workshop on Computer Science, September 25 – 27, 1989.

[Ichbiah et al. (79)] Ichbiah, J. D., Barnes J. G. P., Heliard, J. C., Krieg–Brueckner, B., Roubine, O., and Wichmann, B. A., *Rationale for the Design of the Ada Programming Language*, ACM SIGPLAN Notices, Vol. 14, No. 6, June 1979.

[Yokouchi (89)] Yokouchi, H., *Design of Exception Handling for COB*, IBM Tokyo Research Laboratory, Programming Languages Group Report, February 17, 1989.

[Yonezawa and Ichisugi (89)] Yonezawa, A., and Ichisugi, Y., *Exception Handling and Real Time Features in an Object–Oriented Concurrent Language*, distributed at the 2nd UK/Japan Workshop on Computer Science, September 25 – 27, 1989.