

Asynchronous Exceptions in Concurrent Object-Oriented Programming

Volkan Arslan, Bertrand Meyer

Chair of Software Engineering, ETH Zurich
{Volkan.Arslan, Bertrand.Meyer}@inf.ethz.ch
<http://se.ethz.ch>

Abstract. Exceptions in concurrent object-oriented languages with asynchronous call semantics may raise a serious problem in certain situations. Since separate calls are asynchronous it might happen that the context of the enclosing routine, from which the asynchronous call was launched, has been already left and hence any exception raised by the asynchronous call can not be handled anymore by the enclosing routine. In this paper we present a practical solution for this problem, which relies on the notion of busy processors.

1 Introduction

Exceptions play an important role in programming robust software systems. According to [1] informally, an exception is an abnormal event that disrupts the execution of a system. In sequential (object-oriented) programming languages relying on Design by Contract technique, exceptions have a very clear and precise semantics. The situation changes dramatically as soon as one moves from sequential to concurrent programming, especially if the underlying concurrency model relies on so-called asynchronous calls; that is calls that are not blocking.

The rest of this paper is organized as follows: Section 2 explains shortly the semantics of exceptions in sequential object-oriented languages such as Eiffel. Section 3 first outlines shortly the concurrency model used in this paper and then describes the exact problem with exceptions in concurrent object-oriented programming and afterwards presents our proposed exception mechanism. Section 4 draws conclusions and discusses possible extensions of the proposed exception mechanism.

2 Exceptions in sequential programming

The semantics of exceptions in sequential object-oriented languages such as Eiffel relying on the Design by Contract technique is extremely easy to understand and to explain. Before we give a precise definition for the term exception, we have to give the definitions for routine success and failure.

Definition: success, failure

A routine call succeeds if it terminates its execution in a state satisfying the routine's contract. It fails if it does not succeed.

The routine's contract consists of its precondition and postcondition. Additionally the implementation of the routine can have assertion checkings through the **check** keyword. It should be noted that a class has also a class invariant which must hold after the execution of any publicly exported routine. Now we can give the definition for the term exception:

Definition: exception

An exception is a run-time event that may cause a routine call to fail.

Having given the definitions for routine success and failure, we note that a routine call will fail if and only if an exception occurs during its execution and the routine does not recover from the exception. In general there are various ways how to deal with the occurrence of exceptions, but in Eiffel the exception mechanism supports the Disciplined Exception Handling Principle:

Disciplined Exception Handling Principle

There are only two legitimate responses to an exception that occurs during the execution of a routine:

- 1 • **Retrying**: attempt to change the conditions that led to the exception and to execute the routine again from the start
- 2 • **Failure**: (also known as **organized panic**): clean up the environment, terminate the call and report failure to the caller.

The above disciplined exception handling principle is supported through the **rescue** and **retry** clauses.

```

r
require
    precondition
local
    ... local entity declarations
do
    body
ensure
    postcondition
rescue
    rescue_clause
end

```

A routine *r* might have a rescue clause. In case of an exception during the execution of the normal routine *body*, the execution in the body part will stop and the *rescue_clause* will be executed instead. Inside the *rescue_clause* there can be a **retry** instruction whose execution will force to re-start the routine body from the beginning without repeating the initialization of the routine. As stated above a routine such as *r* might either fail after executing the *rescue_clause* if there is no **retry** instruction or

succeed after a **retry** instruction. It should be noted that the execution of the *rescue_clause* has to establish the class invariant and additionally the precondition if the **retry** instruction is executed. If a routine fails the exception will be propagated to the caller routine, that is it to the routine which called *r*.

3 Exceptions in concurrent programming

3.1 The SCOOP model

The SCOOP model (Simple Concurrent Object-Oriented Programming) [1], [2] offers a comprehensive approach to building high-quality concurrent and distributed systems. The idea of SCOOP is to take object-oriented programming as given, in a simple and pure form based on the concepts of Design by Contract, which have proved highly successful in improving the quality of sequential programs, and extend them in a minimal way to cover concurrency and distribution. The extension indeed consists of just one keyword **separate**; the rest of the mechanism largely derives from examining the consequences of the notion of contract in a non-sequential setting. The model is applicable to many different physical setups, from multiprocessing to multithreading, network programming, Web services, highly parallel processors for scientific computation, and distributed computation. For application programmers, writing concurrent applications with SCOOP is extremely simple, not requiring the usual baggage of concurrent and multithreaded programming (semaphores, rendezvous, conditional critical regions etc.). The model takes advantage of the inherent concurrency implicit in object-oriented programming to provide programmers with a simple extension enabling them to produce concurrent applications with little more effort than sequential ones.

Processors

SCOOP uses the basic scheme of the object-oriented computation: the feature call, e.g. *x.f(args)*, which should be understood in the following way: the client object calls feature *f* on the supplier object attached to *x*, with the argument *args*. In a sequential setting, such calls are synchronous, i.e. the client is blocked until the supplier has terminated the execution of the feature. To introduce concurrency, SCOOP allows the use of more than one processor to handle execution of features. A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. It can be implemented by a piece of hardware (CPU), a process, a single thread in a multithreaded environment, or an application domain in Microsoft .NET, etc. If different processors are used for handling the client and the supplier objects, the feature call becomes asynchronous: the computation on the client object can move ahead without waiting for the call to terminate. Processors are the principal concept that SCOOP adds to the sequential object-oriented framework. Contrary to a sequential system, a concurrent system may have any number of processors, independently of the number of available CPUs.

Separate calls

A declaration of an entity or function, which normally appears as *x: X* may now also be of the form *x: separate X*. Keyword **separate** indicates that entity *x* is handled by a different processor, so that calls on *x* should be asynchronous and can proceed in parallel

with the rest of computation. With such a declaration, x becomes a separate entity. If the target of a call is a separate expression, i.e. a separate entity or an expression involving at least one separate entity, such call is referred to as separate call.

Synchronization

No special mechanism is required for a client to resynchronize with its supplier after a separate call $x.f(args)$ has gone off in parallel. The client will wait if and only if it needs to, i.e. when it requests information on the object through a query call, as in $value := x.some_query$. This automatic mechanism is known as wait by necessity [3]. SCOOP ensures that the separate calls made by the client to each supplier are executed in the correct order (FIFO).

Contracts and preconditions

SCOOP relies largely on the principles of Design by Contract. In particular, it introduces a new semantics for preconditions. The semantics of preconditions is different in sequential and concurrent setting. In sequential programs, preconditions are assertions that have to be fulfilled by the client object before calling the routine of the supplier object. If one or more preconditions are not met, the contract is broken and an exception is raised in the client object. In a concurrent context, the preconditions which do not involve any separate entities (e.g. $value_specified$ in the example routine *store* below) keep their original semantics: they are correctness conditions.

The preconditions involving calls on separate objects (e.g. $buffer_not_full$) change their semantics:

```

store (b: separate BUFFER [G]; v: G)
  -- Store v in b.
  require
    value_specified: v /= Void
    buffer_specified: buffer /= Void
    buffer_not_full: not b.is_full
  do
    b.put (v)
  ensure
    buffer_not_empty: not b.is_empty
  end

```

They become wait conditions. If such precondition is not satisfied, it does not result in an exception raised in the client; it only causes the client to wait until the precondition is satisfied.

3.2 The problem with exceptions in asynchronous feature calls

The basic problem with exceptions in concurrent programming with asynchronous feature calls such as $x.f(args)$ where x is a separate entity, is that it might happen, that the context of the enclosing routine, from which the asynchronous call was launched, has been already left and hence any exception raised by the asynchronous call can not be handled anymore by the enclosing routine. To illustrate the problem in more detail con-

sider class *X* (see /1/). Class *X* consists of the routines *f*, *g*, and *establish_invariant* which are commands and of *query* which is a query returning a boolean value.

```
class X feature /1/  
  
f  
  require  
    precondition_1  
  do  
    ...  
  ensure  
    postcondition_1  
  end  
  
g  
  require  
    precondition_2  
  do  
    ...  
  ensure  
    postcondition_2  
  end  
  
query: BOOLEAN  
  do  
    ...  
  end  
  
establish_invariant  
  do  
    ...  
  end  
  
end
```

As a side note commands can change the state of objects, whereas queries return information about objects. Furthermore /2/ lists a class *C1* which uses the class *X*. *C1* is said to be a client of *X*, and *X* a supplier of *C1*.

In routine *start* in class *C1* the call *r (my_x)* causes to reserve the separate object, to

```

class C1 feature /2/

start
  do
    r (my_x)
    ...
    z := z + 1
    ...
    s (my_x)
  end

my_x: separate X
z: INTEGER

r (x: separate X)
  do
    x.f
    x.g
  end

s (x: separate X)
  require
    x.query
  do
    x.f
  rescue
    x.establish_invariant
  retry
end

end

```

which *my_x* is attached, and afterwards the asynchronous call of the feature *f* on the formal argument *x*. Since this call *x.f* is a non-blocking call the next instruction *x.g* can also be launched immediately after the first asynchronous call. The routine *r* will then terminate since there is no other instruction in *r* and one thread of program execution will continue with the next instruction *z := z + 1* in routine *start* of class *C1*.

Now it can happen that one or both of the feature calls *x.f* and *x.g* fail due to an exception raised either in *f* or *g* of class *X*. Since the caller of the feature calls *x.f* and *x.g* (which is *r*) has already left the context, it is clear that *r* cannot handle anymore the exceptions propagated by *f* or *g* of class *X*. The sketched scenario above is a severe

problem in concurrent object-oriented programming. It should be noted that the above problem would not appear in the following modified routine r of class $C1$:

```

r(x: separate X)
  do
    x.f
    x.g
    res := x.queryf
  end

```

Since the last instruction in the routine r now is a query instead of a command, whose result is assigned to an entity res , the call $x.query$ will be thanks to wait by necessity synchronous and hence the context will be not left in case of an exception.

3.3 Proposed asynchronous exception mechanism

The proposed exception mechanism relies on the notion of *busy processors*. A processor is called *busy* if an exception has been raised by any object handled by this processor. Let us illustrate the proposed solution again through a simple example. Assume the following declarations:

```

c1: separate C1
c2: separate C2
c3: separate C3

```

The class code of $C2$ (see /3/:

```

class C2 feature /3/
  r(x: separate X)
    do
      x.f
    end
  end
end

```

and $C3$ (see /4/) are similar to those of $C1$

```

class C3 feature /4/
  r(x: separate X)
    do
      x.f
    end
  end
end

```

We assume that on behalf of $c1$ the feature r (through the feature $start$) has been executed; similarly on behalf of $c2$ and $c3$ the feature r has been executed. Hence all three objects are competing for the processor PX of the shared separate object x . Assume that the processor $P1$ of $c1$ reserves or locks first the processor PX . This means that both the processors of $c2$ and $c3$ ($P2$ and $P3$) have to wait until the processor PX is free again. Now $P1$ asynchronously calls $x.f$ and $x.g$ and leaves the context of r and continues to execute the next instruction in the routine s , which is the assignment instruction. In the meanwhile assume that the asynchronous call $x.f$ fails due to an exception. In this case the processor PX is declared as “busy” meaning that only objects of the processor $P1$ can access the processor PX . This means in particular that an object of $P1$ has to bring the state of the processor PX from “busy” to “normal” since $P1$ was the originator of the exception through the routine r of $C1$. For the processors $P2$ and $P3$ the processor PX will still be busy meaning that the processor PX is not available for them. This is similar to the case where several separate processors are competing to lock a certain processor, but only one processor can succeed and the others have to wait.

The interesting question now is what should happen when $P1$, the originator of the exception in PX , again accesses PX . In this case $P1$ should get the pending exception. In our example this will be the case when $P1$ executes $s(my_x)$ and tries to lock PX . Since PX is in busy state, and $P1$ is the originator of this state, $P1$ will get the exception before it enters the body of the routine s without the need to wait until PX is free and without checking for the waitcondition $x.query$. $P1$ will immediately continue in the rescue clause of the routine s . There $P1$ has then the possibility to reestablish the invariant of the separate object attached to x by calling $x.establish_invariant$ and then to call `retry` to continue the execution in the body of the routine s . As soon as the pending exception is handled, the state of the processor PX will be set from “busy” to “normal”. Now when PX is again in “normal” state, other processors such as $P2$ and $P3$ can access PX as if nothing has happened.

The advantage of this solution lies in the fact that other processors such as $P2$ and $P3$ are not punished for the exception which has been not caused by any objects of $P2$ and $P3$. Since $P1$ is the originator of the exception, an object handled by $P1$ is the best suited one to resolve the problem. One major disadvantage for the processors $P2$ and $P3$ is of course the fact that in certain situations they have to wait indefinitely for the processor PX , if no other object of $P1$ accesses PX and hence brings PX into normal state. But again the problem can be solved by $P2$ and $P3$ by introducing timeouts. After a certain amount of waiting time, $P2$ and $P3$ can access PX , but then they will be confronted immediately with the pending exception.

4 Conclusions and ongoing work

We presented a simple solution for the exception mechanism in concurrent object-oriented languages relying on asynchronous calls. It should be noted that there are not many concurrent object-oriented languages relying on asynchronous calls. One previous work [4] relied on wait by rescue, meaning that whenever a routine has a rescue clause, the routine had to wait until all asynchronous calls terminated. This approach was not

very efficient from performance aspects, since all calls in such routines degenerated to synchronous calls losing the attractiveness of concurrent programming.

We are currently in the process of integrating the proposed exceptions mechanism into our SCOOP library called SCOOPLI [5]. We are also extending our SCOOP library with a timeout mechanism and specific support for periodic and aperiodic real-time tasks [6].

References

- [1] Meyer B.: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.
- [2] Nienaltowski P., Arslan V.: *SCOOPLI: A library for concurrent object-oriented programming on .NET*; in 1st International Workshop on C# and .NET 2003, University of West Bohemia, Plzen, Czech Republic.
- [3] Caromel D.: *Towards a Method of Object-Oriented Concurrent Programming*; in Communications of the ACM, Volume 36, Number 9, September 1993, 90-102
- [4] Nenning C.: *Exception Handling in SCOOP*, Master's thesis, 2004, ETH Zurich
- [5] SCOOP Library: available for download at <http://se.inf.ethz.ch/research/scoop>
- [6] Arslan V., Eugster P., Nienaltowski P.: *Modeling Embedded Real-Time Applications with Objects and Events*; in 12th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2006, San Jose, California , United States.