Can Asynchronous Exceptions Expire?

Benjamin Morandi, Sebastian Nanz, Bertrand Meyer
Chair of Software Engineering, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch
`http://se.inf.ethz.ch/`

***Abstract*—A multitude of asynchronous exception mechanisms have been proposed. They specify where and when an asynchronous exception propagates. We highlight another aspect that has largely been overlooked: can an asynchronous exception expire? We discuss scenarios where it is meaningful for an asynchronous exception to expire. We further elaborate on one of the scenarios, thereby outlining an asynchronous exception mechanism for the SCOOP concurrency model.**

***Keywords*-asynchronous exception; concurrent programming; SCOOP**

I. Introduction

Asynchronous exception mechanisms specify where an exception propagates: in the supplier [3], in the client [1]–[3], [7], [8], in a supervisor [9], or in a cooperation [5], [10]. Few of them, however, treat the question whether it is meaningful for an asynchronous exception to expire.

Expiration is not uncommon for synchronous exceptions: in the context of synchronous calls with time-outs for example, the synchronous call expires together with its exception when the time is up. The following section showcases scenarios where expiring of exceptions is valid in the asynchronous case as well.

II. Scenarios

End of a supplier: In μC++ [1], an asynchronous exception expires when the supplier ends its execution. At that point, the client can no longer communicate with the supplier; hence the exception becomes meaningless and can therefore expire with the supplier.

Cancellation: μC++ [1] also has a cancellation mechanism. Once a client is no longer interested in the outcome of a supplier, it can cancel the supplier; this causes the supplier to clean-up and terminate. With the termination, any pending exception expires because the client is no longer interested in the outcome.

Futures: Asynchronous exceptions can expire in languages that have futures, such as ProActive [2] and Rintala's C++ extension [7]. A future represents a result that a supplier is computing asynchronously. When the client needs the result, it accesses the future. This forces the client to wait until the result is available or the supplier raises an exception. In the latter case, the client propagates an exception. If the client does not access the future, it never propagates the exception. At some point the garbage collector disposes the future, and hence the exception expires. This behavior makes sense because when the client never uses the future, the exception is irrelevant. The supplier must, however, clean up by reverting its side effects.

Locks and Contracts: Expired exceptions are also meaningful in languages that combine resource locking with *contracts*, i.e., pre- and postconditions that specify a routine's behavior. A client *can assume the postcondition* of an issued call as long as the client keeps a lock on the supplier: while the client keeps the lock, it can be sure that no other client modifies the supplier after the call; as soon as the client gives up the lock, it no longer has any guarantees. A client *relies on a postcondition* by synchronizing with the supplier, while it can still assume the postcondition. Only then does a supplier's exception, i.e., a symbol for a failed postcondition, matter. Consequently, an exception does not matter anymore when the client can no longer assume the postcondition, i.e., after unlocking the supplier. The supplier must, however, clean up in reaction to an exception.

One question remains: does it make sense for a client not to synchronize with a supplier? It does, for example, when the client spawns multiple suppliers but only wants the results from some of them. The suppliers could be performing random searches with different seeds, and the client is happy with the first result. In absence of a cancellation mechanism, the client must ignore the second searcher by not synchronizing with it. In this case, the client not only ignores exceptions from the second searcher, but also normal results.

The next section elaborates on the last scenario in the context of the SCOOP concurrency model.

III. Mechanism for a Language with Locks and Contracts

A. Overview of SCOOP

SCOOP [4], [6] is an object-oriented programming model for concurrency, which combines resource locking with contracts. Every object is associated with a *processor*, an autonomous thread of control capable of executing actions on objects. An object's class describes the possible actions as *features*.

A variable x belonging to a processor can point to an object with the same handler (*non-separate object*), or to an object on another processor (*separate object*). In the first

case, a *feature call x.f* is *non-separate*: *x*'s handler executes the feature synchronously. In this context, *x* is called the *target* of the feature call. In the second case, the feature call is *separate*: the handler of *x*, i.e., the *supplier*, executes the call asynchronously on behalf of the requester, i.e., the *client*. The possibility of asynchronous calls is the main source of concurrent execution. The asynchronous nature of separate feature calls implies a distinction between a feature call and a *feature application*: the client logs the call with the supplier (feature call) and moves on; only at some later time will the supplier actually execute the body (feature application).

To illustrate these concepts, consider an application that explores a search space to find solutions to a problem. A controller triggers two concurrent searchers; a log records the solutions. In the following code for this example, note that the keyword **separate** is a type system extension to specify that an entity may reference an object on a different processor.

```
class CONTROLLER feature
  start (
    first_searcher: separate SEARCHER;
    second_searcher: separate SEARCHER;
    log: separate LOG
  )
    do
      -- Search concurrently.
      first_searcher.search; second_searcher.search
      -- Record the solutions.
      log.add_entry (first_searcher, second_searcher)
    end
end

class SEARCHER feature
  seed: INTEGER
  solution: STRING

  search
    do
      -- Get the seed from atmospheric noise.
      seed := atmospheric_noise
      -- Search if possible; otherwise, fail.
      if seed >= 0 then
        solution := random_solution (seed)
      else raise_exception
      end
    ensure not solution = Void -- The postcondition.
    rescue seed := 0 -- Restore consistency.
    end

invariant seed >= 0 -- The consistency criterion.
end
```

```
class LOG feature
  add_entry (
    first_searcher: separate SEARCHER;
    second_searcher: separate SEARCHER
  )
    do
      -- Has the first searcher found a solution?
      if not first_searcher.solution.is_empty then
        -- Yes, he has. Log the first solution.
        write (first_searcher.solution)
      else
        -- No, he has not. Log the second solution.
        write (second_searcher.solution)
      end
    end
end
```

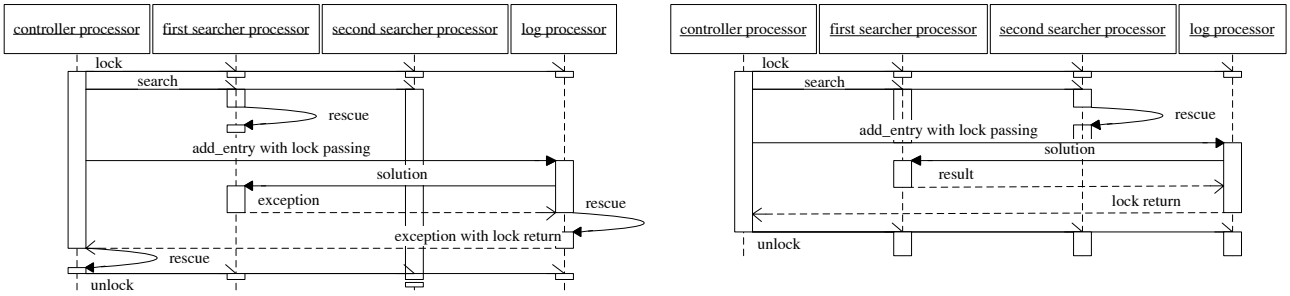
Locking requirements of a feature must be expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' handlers are locked for the duration of the feature execution, thus preventing data races. For instance, in *start*, *log* is a formal argument; the controller has exclusive access to the log while executing *start*.

Sometimes it is necessary to transfer the ownership of locks between processors through the *lock passing* mechanism. In *start*, the log takes the searchers as arguments. To be able to continue, the log requires the lock on the searchers, currently owned by the controller. To resolve the situation, the controller automatically *passes the locks* and waits until the *locks return*; the feature call becomes synchronous. There is another situation where a separate feature call becomes synchronous: when a client expects a result from a supplier, then the client must wait until the supplier provides the result (*wait by necessity*).

B. Asynchronous Exception Mechanism

In the proposed asynchronous exception mechanism, a supplier reacts to an exception by executing a *rescue clause* to re-establish its consistency. A rescue clause for a feature appears after the **rescue** keyword; features without this keyword have an implicit empty rescue clause. The exception persists while the client is executing the feature for which the supplier got locked. The client propagates the exception when it executes a synchronous feature call; these synchronization-based polling points are comprehensible and have also been used in other languages such as Ada [8]. When a client passes a lock to another client, the exception persists because the lock is still in place. Therefore, the next client also propagates the exception during a synchronous feature call. In case no client synchronizes until the supplier gets unlocked, the exception expires.

To demonstrate the working of the mechanism, assume in the example that the first searcher raises an exception and



(a) The first searcher raises an exception.

(b) The second searcher raises an exception.

Figure 1. The interactions between the controller, the searchers, and the log.

executes its rescue clause, as shown in Figure 1a. Because the feature call to the first searcher is asynchronous, the controller does not propagate the exception. In *log.add_entry*, the controller passes its locks to the log; the log calls the first searcher synchronously and propagates the exception. Assume now that the second searcher raises an exception instead, as shown in Figure 1b. Again, the controller passes its locks to the log; however, since the log does not call the second searcher, the log does not propagate the exception. After the controller gets back the passed locks, it unlocks the searchers, and the exception expires. This behavior makes sense because the controller does not rely on the second searcher's postcondition; hence the exception is irrelevant. By executing its rescue clause, the second searcher makes sure to re-establish its consistency.

For programs where exceptions must not be lost, the mechanism has a *safe mode*, in which a client propagates any pending exceptions just before unlocking the suppliers. This mode, however, reduces the potential for concurrency: the client can no longer asynchronously issue an unlock request and then continue; it has to wait until the supplier finished. For long-lived suppliers, this might not be convenient for the client. For these cases, failures in suppliers can also be handled entirely in the supplier's rescue clause, for example with a callback to the client.

IV. Conclusion

Expired exceptions are well-known in the context of synchronous calls. For asynchronous calls, however, exception expiration is not yet a central concept, even though there are benefits: by letting asynchronous exceptions expire, concurrent systems can become more fault-tolerant. To highlight this fact, we showed that expiration of asynchronous exceptions is meaningful in a number of scenarios.

Acknowledgments

We thank the anonymous reviewers for their detailed comments. This work is part of the SCOOP project at ETH Zurich, which has benefited from grants from the Hasler Foundation, the Swiss National Foundation, Microsoft (Multicore award), and ETH (ETHIIRA).

References

[1] Peter A. Buhr. μC++ annotated reference manual. Technical report, University of Waterloo, 2006.

[2] Denis Caromel and Guillaume Chazarain. Robust exception handling in an asynchronous environment. In *Workshop on Exception Handling in Object-Oriented Systems*, 2005.

[3] Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Exception handling and asynchronous active objects: Issues and proposal. In *Advanced Topics in Exception Handling Techniques*, pages 81–100, 2006.

[4] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[5] Robert Miller and Anand R. Tripathi. The guardian model and primitives for exception handling in distributed systems. *IEEE Transactions on Software Engineering*, 30(12):1008–1022, 2004.

[6] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, 2007.

[7] Matti Rintala. Handling multiple concurrent exceptions in C++ using futures. In *Advances in Exception Handling Techniques*, pages 62–80, 2006.

[8] S. Tucker Taft and Robert A. Duff. *Ada 95 Reference Manual*. Springer, 1997.

[9] Anand R. Tripathi and Robert Miller. Exception handling in agent-oriented systems. In *Advances in Exception Handling Techniques*, pages 128–146, 2000.

[10] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecilia M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.