JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2007

Vol. 6, No. 10, November–December 2007

Exceptions in Concurrent Eiffel

Phillip J. Brooke, School of Computing, University of Teesside, Middlesbrough, TS1 3BA, U.K. P.J.Brooke@tees.ac.uk
Richard F. Paige, Department of Computer Science, University of York, Heslington, York, YO10 5DD, U.K. paige@cs.york.ac.uk

We describe the problem of asynchronous exceptions in Eiffel's Simple Concurrent Object-Oriented Programming (SCOOP). We discuss a range of possible solutions to further enable dependable computing in concurrent Eiffel. We propose a mechanism to handle aynchronous exceptions via a limited developer choice, including the notion of a failed or dead object, and necessarily introduce a small number of new exceptions. We additionally describe a number of mechanisms that were discarded as unsuitable.

1 INTRODUCTION

The Simple Concurrent Object-Oriented Programming (SCOOP) mechanism is proposed as a means to introduce inter-object concurrency into the Eiffel programming language [Mey97, EcI05]. SCOOP extends the Eiffel language by adding one keyword, **separate**, which can be applied to classes, entities and formal routine arguments. Application of **separate** to a class indicates that objects of that class execute in their own conceptual thread of control; application of **separate** to entities (variables) or arguments of routines indicate that these constructs are points of synchronisation. Thus, SCOOP aims at a minimal (syntactic) extension to the Eiffel language, through combining the notions of thread and synchronisation in **separate** classes and entities.

Aim, scope and limitations

We discuss the SCOOP mechanism as it relates to sequential Eiffel's existing exception handling. As we shall see, the existing exception handling mechanism is insufficient to deal with failures in SCOOP. We thus desire an extension to the exception handling mechanism that 1. permits the creation of reliable, predictable systems; 2. allows the use of existing (sequential) libraries; and 3. is not unduly burdensome to either the run-time system or the developer.

The aim of this paper is to propose an extension to Eiffel's sequential exception handling in order to satisfy these requirements and fully support SCOOP.

As the focus of this work is on exception handling in SCOOP (as SCOOP is currently documented [Mey97]), we do not aim to compare SCOOP itself with other approaches to concurrency such as those found in Java, active objects or POSIX threads, although we do examine some related work that directly deals with exceptions. Some discussion on these issues can be found in [PB06].

Overview

We start with brief overviews of Eiffel in Section 2 and SCOOP in Section 3. From this, we state the problem with exceptions in SCOOP in Section 4.

Next, we describe the related work in this area in Section 5. We then further discuss exceptions and describe our proposed solutions in Section 6.

Sections 7 and 8 describe the mechanisms we rejected, and additional sources of exceptions in the concurrent environment. The paper ends with a discussion and our conclusions in Sections 9 and 10.

2 EIFFEL

Eiffel is a pure object-oriented (OO) programming language [Mey97, EcI05] that provides constructs typical of the OO paradigm, including classes, objects, inheritance, associations, composite ("expanded") types, polymorphism and dynamic binding, and automatic memory management. Novelties with Eiffel include its support for full multiple inheritance, generic types (including constrained generics), agents (closures and iterators over structures), and strong support for assertions, via preconditions and postconditions of routines, and invariants of classes.

Routines may have preconditions (**require** clauses) and postconditions (**ensure** clauses). The former must be true when a routine is called (*i.e.*, it is established by the caller) while the latter must be true when the routine's execution terminates. Classes may have invariants specifying properties that must be true of all objects of the class at stable points in time, *i.e.*, after any valid client call on the object. An exception is raised if an assertion (precondition, postcondition or invariant) evaluates to false.

For more details on the language, see [Mey97] or [EcI05].

Exceptions

Exceptions in sequential Eiffel can occur when an assertion is violated (*e.g.*, a routine is called with precondition *false*), a called routine fails, an interrupt is sent by the operating system, or an operation fails (*e.g.*, creation of a new object fails, arithmetic overflow occurs).

Exceptions are handled by so-called **rescue** clauses. A rescue clause is attached to a routine (after the postcondition) and describes a sequence of instructions that should be executed when the routine is to recover from an undesirable run-time event. One new instruction that can be included in rescue clauses is **retry**, which is a directive to re-start execution of the routine body from the beginning.

3 SCOOP

Outline

SCOOP introduces concurrency to Eiffel by the addition of the keyword **separate**. The **separate** keyword may be applied to the definition of a class or the declaration of an entity (a variable) or formal routine argument.

Access to a separate object, whether via an entity or formal argument indicates different semantics to the usual sequential Eiffel model. In the sequential model, a call to a routine causes execution to switch to the called object whereupon the routine executes; on completion, execution continues at the next instruction of the original object.

In SCOOP, procedure calls are asynchronous. The called object can queue multiple calls, allowing callers to continue concurrent execution. Function calls (*e.g.*, $a := x.f$) and reference access to attributes are synchronous — but may be subject to *lazy evaluation* (also known as *wait-by-necessity*).

Races are prevented by the enforced convention that a separate formal argument causes the object to be exclusively locked ('reserved') during that routine call. However, there are complications with locking, in that deadlocks may arise, or concurrency may not be maximised, unless some form of *lock passing* [Bro06a, Bro06c] is used.

Processors

SCOOP introduces the notion of a *processor*. When a separate object is created, a new processor is also created to *handle* its processing. This processor is called the object's *handler*. (Objects created as non-separate are handled by the creator's handler.) Thus, a processor is an autonomous thread of control capable of supporting *sequential* instruction execution [Mey97]. A system in general may have many processors associated with it.

Compton [Com02a] introduces the notion of a *subsystem*: a model of a processor and the set of objects it operates on. In his terminology, a separate object is any object that is in a different subsystem. In this paper, we will refer to subsystems rather than processors (to avoid possible confusion with real CPUs).

We additionally introduce the notion of a *partition* to describe any real processing resource, *e.g.*, a CPU, a POSIX thread or process [Bro06a]. Subsystems are assigned to partitions.

Preconditions and waiting

As described in Section 2, Eiffel uses **require** and **ensure** clauses for specifying the pre- and postconditions of routines. If a precondition or postcondition evaluates to false, an exception is raised.

There are two possible views on exceptions here:

1. The exception is due to a fault in the implementation that requires repair — it is a bug.
2. The exception is an unexpected condition at run-time that needs to be compensated for or tolerated.

In the first case, it is reasonable, at least during development, for the exception to cause an immediate system halt with suitable diagnostic information to allow the bug to be fixed. In the second case, catastrophic responses are likely problematic, particularly in systems required to be fault-tolerant.

In SCOOP, a **require** clause on a routine belonging to a separate object specifies a *wait* condition: if the routine's **require** clause evaluates to false, the processor associated with that object waits until the precondition is true before proceeding with routine execution.

There is then a third interpretation to be placed on exceptions that would otherwise arise from preconditions: that they are *guards* restricting when a feature may execute. Clearly, these exceptions are not bugs (although an unintended deadlock would clearly be a bug). However, we must handle exceptions that are raised while evaluating the precondition.

4 THE PROBLEM: EXCEPTIONS AND SCOOP

A major advantage of SCOOP is that it allows procedure calls on separate objects to progress asynchronously: that is, a caller can enqueue a procedure call on a callee and then continue processing. Similarly, function calls can, by lazy evaluation, issue a request and then continue until that return value is actually needed in the caller.

This brings with it a problem: suppose that the called routine fails. A routine *fails* when an exception is raised, and there is either no **rescue** clause, or the **rescue** clause does not lead to a **retry**. This failure will cause an exception in the caller in the sequential programming model, where the exception will be propagated through the list of callers until either a **rescue** clause succeeds with **retry**, or the root class's creation routine fails and the run-time system halts the whole program.

However, in SCOOP, if the caller and callee are separate, then the caller might already have completed when the callee fails. In this case, there is no obvious target to send the exception to. We cannot send the exception to another arbitrary object in the same subsystem or partition since there is not necessarily a call relationship between these objects.

So we ask: How should we handle asynchronous exceptions in SCOOP?

This is an instance of a more general problem: in distributed or concurrent setting, exception handling is difficult. The problem is particularly challenging in SCOOP with its complex underlying run-time structure, and because SCOOP does not disentangle locking and synchronisation. Note that this problem does not arise in *synchronous* languages such as Esterel [Est99], where the system can guarantee that a caller will still exist whenever a callee may raise an exception.

5 RELATED WORK

Implementations of SCOOP

An incomplete prototype of the SCOOP mechanism was implemented by Compton [Com02a] by building upon the GNU SmartEiffel compiler and run-time system. A prototype preprocessor implementation was constructed by Fuks *et al.* for ISE Eiffel [Fuk04].

More recently, Nienaltowski *et al.* [Nie05] have produced the most complete implementation of SCOOP to date. This (in common with other prototypes) is a preprocessor that rewrites SCOOP-using classes. None of these prototypes can be considered a full implementation of the SCOOP specification in [Mey97].

Exceptions in SCOOP

Compton and Walker [Com02b] note that

> "Consider a case where processing had continued well beyond the point of a separate call, perhaps even exiting the routine in which the call was made. An asynchronous exception at this time could not possibly have the same semantics as in the usual case."

This clearly matches the problem as described earlier. They continued by discussing this with Meyer, and say

> "There seem to be two possible solutions. Either, that an exception between subsystems is considered as catastrophic and causes program termination, or that the object throwing the exception is flagged as dirty and any later attempt to use this object results in program termination."

However, we view both options as being overly aggressive to be the only solutions: they will result in trivial errors bringing down large, (hopefully) long-lived systems. (A highly concurrent system will surely have many cooperating subsystems.)

Adrian discusses an extension of SmallEiffel (itself now known as SmartEiffel) to include SCOOP [Adr02]. In that work, he quotes from the SmallEiffel mailing list, one of which notes that asynchronous exceptions would require all routines to be able to handle all possible exceptions (an undesirable outcome). Instead, Adrian says that SmallEiffel will adopt the following approach:

> "A subsystem is to me[sic] marked "dirty" if an exception reaches the root call of the subsystem. Any waiting command is (silently) discarded. The subsystem is then made unable to serve any request:
>
> - Any request (command or query) posted to a dirty subsystem immediately raises an exception in the caller;
> - Any query awaiting to be served also raises an exception in the blocked subsystem.
>
> Now, when we say "an exception", which exception? Should we raise the same as the dirty subsystem, or a new one? In this case, do we have access to the original exception code?"

However, this approach does not make clear how an exception would reach the "root call of a subsystem" in the first place: there is not necessarily a causal relationship between a caller and the first object to be created in a SCOOP subsystem.

Recently, Arslan and Meyer proposed a notion of 'busy processors' [Ars06]. In this work, an unhandled asynchronous exception causes the processor to be marked 'busy' until the processor that enqueued the failed call takes action to reestablish the invariant on the called object. However, we do not see that there is necessarily a causal relationship between particular processors and the object-call relationships.

We are unaware of any Eiffel/SCOOP implementation that handles asynchronous exceptions. We know of no other serious proposals other than Arslan and Meyer's busy processors.

Exceptions in other concurrent languages

Ada 95 [Taf97] introduces concurrency via tasks. Rendezvous is possible between tasks, and passive objects (protected types) allow sharing of data. An unhandled exception results in the termination of the enclosing task. Subsequent attempts to rendezvous with a terminated task cause an exception in the new caller.

Java's concurrency support is essentially thread-based, with block-based locking (synchronization) on a per-thread basis. There are few cases for asynchronous exceptions, so the problems that SCOOP encounters do not arise in general for Java.

Exceptions in object-oriented programming

Other more general work on exceptions in object-oriented programming has been undertaken, notably in the Exception Handling in Object Oriented Systems series [Rom05]. However, little of the work directly addresses the issue of asynchronous exceptions as the problem relates to SCOOP. Some from other areas is applicable as we discuss shortly.

Dony *et al.* [Don91] discuss a range of issues, and many contributions emphasise providing choice and developer-chosen links between exceptions and handlers. Issarny [Iss01]'s concurrent exception handling is another example of synchronous exception handling. Parallel procedures are grouped in multiprocedure blocks. It is not obvious how this approach can be extended to asynchronous concurrent systems. Romanovsky and Kienzle [Rom01] survey a number of approaches to exception handling in concurrency, and illustrate that exception handling is typically associated with the textual context of a program, *i.e.*, exceptions propogate outwards through the associated nested contexts. Again, there is no obvious solution offered to the problem of asynchronous, *i.e.*, out of context, exceptions. The discussions on transactions, however, suggest that changing Eiffel's concurrency model to be based around transactions may be compatible with exception handling approaches proven in other application domains.

Asynchronous exceptions have received attention. Caromel and Chazarain [Car05] describe a Java approach. However, although the calls are asynchronous, the end of each try/catch block waits for incomplete asynchronous calls, thus the problem that we enounter of the caller expiring before the callee responds with an exception does not arise.

An interesting point that has not arisen directly so far relates to callbacks. This issue arises in sequential Eiffel as well as SCOOP. Suppose object a passes a reference to feature f of object c to object b, with the intention that object b subsequently executes $c.f$, a callback. If $c.f$ throws an exception, then arguably a should receive the report, not b; in other cases, it may make more sense in terms of the system for b to receive the exception, as it would at this time. This particular programming idiom is popular in event-based systems such as GUIs. Ploski and Hasselbring [Plo05] suggest firstly handling such exceptions locally; otherwise propogating the exception to "the invoked callback's clients, that is, modules affected by the callback's failure". However, it is not trivial to identify the affected modules.

Finally, the most relevant related work is due to Rintala, and deals with multiple concurrent and asynchronous exceptions in C++ [Rin05]. Results from asynchronous calls are initially substituted by *futures*; these act as placeholders for the eventual result from the call. Exceptions that arise during the asynchronous call are routed through the future; essentially, they are the return value of the future. Subsequent references then obtain the exception rather than the normal result. Finally, multiple exceptions can be grouped into compound exceptions.

6 FIXING SCOOP EXCEPTIONS

The main problem with exceptions in SCOOP is that the sequential model of a chain of callers does not hold. The client call that should be told of an exception in the called object may no longer exist; the lack of synchrony that makes SCOOP so appealing means that the model of exceptions needs updating.

While fixing this, we require that existing sequential code continues to work without modification. This section applies to exceptions arising from unhandled exceptions in objects that were called by another separate object.

There are three options, that could be chosen for groups of objects and/or classes by the developer:

- the entire system halts on asynchronous exception; or
- the object fails (dies) on asynchronous exception; or
- (groups of) asynchronous exceptions are ignored for particular objects.

We suggest that the first should be the default response. Additionally, a compile-time configuration mechanism or run-time system class should allow objects to be designated for a specific policy (via, say, 'set_exception_policy').

Halting as a response to unhandled asynchronous exceptions

An unhandled exception in the sequential model results in the entire system halting. We justify this as halting an entire SCOOP system in response to an unhandled asynchronous exception is analogous to the sequential case.

Thus we make *halt the entire system* the default response to an unhandled (asynchronous or synchronous) exception. This is a simple, low-overhead solution. Moreover, it is consistent with the Eiffel design-by-contract philosophy: exceptions are usually viewed as being a fault in the implementation that needs repair.

We choose this as the default response to be conservative: our next option as a default may allow objects to silently fail. By forcing the developer to explicitly choose an alternative response, they should then design their system to respond appropriately.

As we remark in Section 5, Compton and Walker in discussion with Meyer offer this as one of two possible solutions. We offer other options to the system designer as we view halting the entire system as too aggressive to be the only solution: it prevents any form of fault tolerance.

Unhandled asynchronous exceptions cause the object to fail and die

As an alternative to halting the system when a routine of an object fails and there is no sequential caller, we may instead mark the object as *failed*. It will never interact with any other object again nor will it process any further calls: this means that any calls already enqueued will not be processed. If a caller considers it important that a procedure call completes, then this caller must ensure that it checks a return value or status.

Future attempts to enqueue work for that object or to access it in any way result in the caller receiving a 'separate_object_failure' exception. An implication of this proposal is that *enqueueing a call to a subsystem's FIFO is synchronous.*

The only information that needs to be retained is the full name (*i.e.*, sufficient information to unambiguously identify it); the type of the object; and the exception object. All other data, including its part of the call queue, can be garbage collected. The exception object can be returned as part of the separate_object_failure exception allowing other objects to possibly determine an appropriate response.

When it is determined that an object will die due to an unhandled asynchronous exception, the system will call the procedure identified by 'set_asynchronous_exception_handler' (or else 'default_asynchronous_exception_handler', a feature that can be overridden in descendent classes). This is treated similarly to default_rescue and offers the failed object a 'last gasp' to carry out some work. The intention is that an object can use this call to set up a handler that makes an appropriate response, *e.g.*, calling a different object to report its failure, or to create a replacement object and indicate to other clients that they should fail-over to the replacement.

There may be some complications resulting from this part of the mechanism: consider the case where the called handler makes a series of subsequent calls, including calls to the dying object. Should they be processed? Similarly, how should subsequent exceptions during this handler be managed?

This part of our solution also handles the case where objects may not be contacted again by another object (we call these independent objects below). This can arise because some objects may have been created with the express purpose of carrying out a task independently, so they may never need to be contacted by any other separate object again, *e.g.*, some Internet service daemons where a client asks for service and a new process is started. In SCOOP, a new separate object could be used for each client. Our proposal allows such objects to report to another object that they have failed.

This solution is similar but not identical to the second option suggested by Compton and Walker in discussion with Meyer (Section 5). In particular,

1. contacting a failed object does not necessarily result in program termination; and

2. we handle the case of independent objects and other required 'tidying-up' by providing a facility to set an asynchronous exception handler.

This solution is also related to Rintala's approach [Rin05], in the sense that Rintala's futures would be replaced by the exception.

Ignoring asynchronous objects

Some objects may not encapsulate any state, or for other reasons, an unhandled exception within that object may not be cause for it to fail. Such exceptions could simply be ignored: the call fails, and the object continues to process its work queue.

The recent ECMA standard for Eiffel [EcI05] alludes to (but does not describe) a mechanism for ignoring exceptions. There is no need for special treatement for SCOOP: the 'ignore an exception' mechanism for the sequential model should apply equally to the concurrent case.

7 DISCARDED MECHANISMS

We considered a wide range of possible mechanisms and discarded all but those described above.

Halting the subsystem or partition We might further suggest that affected subsystem or partition is halted, instead of just the failed object.
However, we view each object as being an independent part of the system; there is not necessarily a causal relationship between objects on a given subsystem. This applies even more to partitions. Thus we argue that there is little reason why an exception in one should cause the other (unrelated) objects to halt, so we dismiss this as an option.

Allowing clearing of an asynchronous exception A failed object may, or may not, have a valid invariant. Alternatively, the object may not encapsulate any state and might be able to continue without any remedial action. Instead of destroying a failed object, we might allow it to be 'resurrected' by another object taking action that achieves both: 1. the invariant is restored (unless the invariant is already valid); and 2. the exception is 'cleared' on the failed object (perhaps by a specific routine of the *CONCURRENCY* class). The first condition requires that a creation procedure is executed on the object — this must be done if the invariant is false, and may be if the invariant is true. The second condition is an indication to the system that the problem has been handled.
However, the restoration of the invariant and the clearing of the exception both require routines to be executed on the failed object. As the object may have other calls already enqueued, these restoration routines must run ahead of the queue. Modifying the object ahead of other enqueued calls may change the result of those calls. Thus we reject this option as overly complicated, and semantically dangerous.

Queues of exceptions for polling or waiting We could arrange that failed objects place their exception into a queue. This queue could be periodically polled by other objects, or could be waited on (*i.e.*, blocking behaviour). But what should happen then? We must arrange that the exception is handled, and also deal with the case when the exception is not handled for a long time (or ever). This option has all the problems of allowing resurrection of a failed object: the lack of a causal connection and the issues of clearing exceptions result in us rejecting this option.

8 ADDITIONAL SOURCES OF EXCEPTIONS

There are three special sources of exception we should consider:

Deadlock It is well known that non-trivial concurrent systems can easily admit deadlock. Nothing in SCOOP directly prevents that, although the philosophy of locking the objects that are needed by listing them as formal arguments may help structure systems such that deadlock is less likely. So other approaches are needed:

- **Deadlock detection at run-time** SCOOP could be implemented to detect cycles of reservation requests. The response to this is difficult: one possibility is that a randomly chosen object has its reservations taken away and an exception raised, perhaps 'deadlock_recovery_attempt'.
- **Deadlock avoidance during design** The CSP model in [Bro06b, Bro06c] could be developed to the point where Eiffel compilers write out CSP suitable for input to FDR [FSE95] or an alternative CSP tool; the tool then checks the CSP model of that particular system for sequences of events leading to deadlock.

Problems in the infrastructure Our system comprises one or more partitions. Suppose that the CPU executing one of these partitions fails or is disconnected from its communications network. In this case, no further calls can be made to or from that partition. What should be done in this case? Clearly, this is the type of problem that exceptions should manage. An attempt to communicate with a partition that has failed or is unreachable should result in an exception. It is meaningful to have a distinct exception from the one proposed earlier; we suggest 'partition_communication_failure'.

Duels Duels, or 'requesting special service', as described in Meyer's text [Mey97, section 30.8], are a way for one object to (attempt to) demand access to a reserved object. The result of this can sometimes be the exception 'is_concurrency_interrupt'. This itself may propogate causing a routine failure. If exception handling can be managed for separate objects as proposed in this work, then the use of duels becomes possible, although other ongoing work may provide a better solution for asynchronous transfer of control[1].

[1]RECOOP (Real-Time Concurrent Object-Oriented Programming), Wellings, Brooke, Paige, Jacob and Burns, with draft semantics by Jacob and Brooke. Publications in preparation.

9 DISCUSSION

The difficulties caused by asynchronous exceptions are a necessary evil: we cannot remove them except by losing the appeal of SCOOP's asynchronous procedures. We can, however, manage them. We compare our work with our aims from Section 1:

1. *Permits the creation of reliable, predictable systems:* By clearly specifying the options for a range of policies, we can build fault-tolerant (or if desired, intolerant) systems.
2. *Allows the use of existing (sequential) libraries:* The existing sequential behaviour of exceptions exists for intra-object calls and calls between objects with the same handler, so existing libraries can continue unaffected. Where an existing library is used as part of a concurrent program, the general solution here becomes applicable.
3. *Is not unduly burdensome to either the run-time system or the developer:* The run-time system must include some means of configuring the desired behaviour, but this is a very limited overhead compared to the rest of the work associated with managing concurrent objects. Developers must necessarily consider the behaviour of their system when it fails: the proposal here gives them a small, expressive but not excessive toolkit to manage this behaviour.

Our work is strongly influenced by the Ada approach to tasks, although all objects in SCOOP have a thread of control. In particular, we do not allow propogation of exceptions outside of an asynchronous 'frame', and the resulting failure (termination) of an object (task) results in further exceptions to callers.

The impact of this proposal on developers is minimal compared to the complexity of the intrinsic task. Developing concurrent systems is known to be subtle; the developer must necessarily decide how to handle exceptions. The default is a safe case, but now the developer can choose that particular classes (or individual objects) may fail without halting the entire system while still offering a mechanism for recovery. Developers must be aware that if a separate call must execute successfully, then they have to take action to ensure that it does occur: either by checking that a result is returned, or otherwise obtaining some form of acknowledgement from the separate object. Otherwise, a failed object could discard its job queue and the calls are never executed without the caller being aware of this failure.

If radical changes are considered, then the concept of compensation[2] may be appropriate: this provides 'forward recovery' in the case of aborted transactions where the effects of the aborted transaction cannot be undone. An approach that would cause substantial changes to Eiffel in general is to treat success as the exception. Thus the main flow of the code makes different or repeated attempts to succeed, while branches or exceptions are invoked when these attempts succeed.

[2]We first heard of compensation in the context of Eiffel from Volkan Arslan and Sebastien Vaucouleur in February 2005. Prof. Sir Tony Hoare suggested other possible mechanisms, notably that of treating success as the exceptional case at the 2006 UTP Symposium.

An alternative radical change is to use a more synchronous form, such as cobegin-coend or a construct that mimics Esterel's [Est99] parallel construct. This then removes the entire problem of asynchronous exceptions by ensuring that a synchronous rendezvous is still available whenever an exception may arise.

These approaches are interesting and may be very profitable in the long term: we have explicitly restricted ourselves to making minimal changes to Eiffel itself, so do not consider them further in this work.

10 CONCLUSION

Our contributions in this paper are

1. survey the current situation with exceptions in concurrent Eiffel;
2. identify suitable options for handling asynchronous exceptions with minimal disruption to the language and libraries;
3. introduce the concept of failed (or dead) objects and the consequences of failed objects;
4. identify mechanisms that we consider are unsuitable.

Our mechanism can be summarised thus: unhandled exceptions propagate as far as possible using the normal sequential mechanism. We have a number of options for handling asynchronous exceptions: halt the entire system (the default response); the object is marked as failed; or ignore the exception. As no single option is satisfactory in all cases, the developer can specify the appropriate response. Exceptions (for deadlocks, infrastructure problems and failed objects) and routines (for setting policies and 'last gasp' handlers) are introduced.

The next steps in our work are:

1. embed a model of exception handling in our CSP model [Bro06b, Bro06c] and examine the behaviours under the various options — in particular, to establish suitable constraints on asynchronous exception handlers to ensure that they do not trigger further problems;
2. demonstrate and evaluate this solution by means of a preprocessor implementation or a simplistic compiler; and
3. further examine how the recent ECMA 367 standard for the Eiffel language [EcI05] affects this work.

ACKNOWLEDGEMENTS

- Bertrand Meyer, Volkan Arslan and others at the CORDIE'06 workshop for their comments on the purpose of exceptions in Eiffel.

REFERENCES

[Adr02] C. Adrian, SCOOP for SmallEiffel, draft, http://www.chez.com/cadrian/eiffel/scoop.html, 2002, last accessed 10th November 2005.

[Ars06] V. Arslan and B. Meyer. Asynchronous exceptions in concurrent object-oriented programming. In [PB06].

[Bro06a] P.J. Brooke and R.F. Paige. A critique of SCOOP. In [PB06].

[Bro06b] P.J. Brooke and R.F. Paige. An alternative model of concurrency for Eiffel. In [PB06].

[Bro06c] P.J. Brooke, R.F. Paige, and J.L. Jacob. A CSP model of Eiffel's SCOOP. To appear in Formal Aspects of Computing, 2007.

[Car05] D. Caromel and G. Chazarain. Robust exception handling in an asynchronous environment. In [Rom05], 2005.

[Com02a] M. Compton. *SCOOP: an Investigation of Concurrency in Eiffel*, MSc Thesis, Australian National University, 2000.

[Com02b] M. Compton and R. Walker. A Run-time System for SCOOP. *Journal of Object Technology* 1(3), special issue: TOOLS USA 2002 proceedings, pp. 119-157, 2002.

[Don91] C. Dony, J. Purchase and R. Winder. Exception handling in object-oriented systems. ECOOP'91, 1991.

[EcI05] ECMA-367: Eiffel Analysis, Design and Programming Language, Ecma International, June 2005.

[Est99] The Esterel v5 Language Primer. ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.pdf,

[FSE95] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR 2. http://www.formal.demon.co.uk/, December 1995.

[Fuk04] O. Fuks, J.S. Ostroff, and R.F. Paige. SECG: The SCOOP-to-Eiffel Code Generator. *Journal of Object Technology* 3(10), November/December 2004.

[Iss01] V. Issarny. Concurrent exception handling. In *Advances in Exception Handling Techniques*, LNCS 2022, 2001.

[Mey97] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 1997.

[Nie05] P. Nienaltowski and B. Meyer, SCOOPLI implementation, http://se.inf.ethz.ch/research/scoop.html, 2005.

[PB06] Richard F. Paige and Phillip J. Brooke, editors. *Proc. First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, number YCS-TR-405. University of York, July 2006.

[Plo05] J. Ploski and W. Hasselbring. The callback problem in exception handling. In [Rom05], 2005.

[Rin05] M. Rintala. Handling multiple concurrent exceptions in C++ using futures. In [Rom05], 2005.

[Rom01] A. Romanovsky and J. Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In *Advances in Exception Handling Techniques*, LNCS 2022, 2001.

[Rom05] A. Romanovsky, C. Dony, J.L. Knudsen, and A. Tripathi (editors). Developing Systems that Handle Exceptions, Proceedings of *ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems.* Technical Report No 05-050. Department of Computer Science. LIRMM. Montpellier-II University. 2005.

[Taf97] T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual.* Number 1246 in Lectures Notes in Computer Science. Springer-Verlag, 1997.

ABOUT THE AUTHORS

Phillip J. Brooke is a principal lecturer at the University of Teesside, United Kingdom, where he researches formal methods and security. He completed his D.Phil. in Computer Science at the University of York in 1999. Subsequently, he worked as a software engineer in the security domain for two years and then five years as a senior lecturer at the University of Plymouth. pjb@scm.tees.ac.uk

Richard F. Paige is a senior lecturer at the University of York, United Kingdom, where he works with the High-Integrity Systems Engineering Group. His research focuses on model-driven development, agile development, and building dependable systems. He completed his PhD in Computer Science at the University of Toronto in 1997. paige@cs.york.ac.uk