Who is Accountable for Asynchronous Exceptions?

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

***Abstract*—Large parts of today's software systems are devoted to detecting and recovering from failures, making exception handling a critical issue in software development. Concurrent software complicates this issue: most concurrent programming languages require a mechanism to deal with asynchronous exceptions, but because of the diverse design choices underlying each language, no approach fits all situations. We introduce a classification of possible approaches to guide the development of asynchronous exception mechanisms, and we show its applicability by deriving a sound and comprehensible mechanism for SCOOP, an object-oriented programming model for concurrency. We describe the key idea of the mechanism using the accountability framework, which precisely defines the obligations of client and supplier regarding the reporting of exceptions. The framework not only provides the necessary intuition to apply the mechanism correctly, it is also useful to comprehend other approaches.**

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

I. Introduction

In concurrent programming languages, multiple *computational units* execute in an interleaved, potentially parallel manner. Many of these languages offer asynchronous (non-blocking) calls, permitting a *client* to continue while the *supplier* executes the call. As long as the call terminates normally, the client does not need to be informed about the success. If the supplier *raises* an exception, however, some computational unit must *propagate* the exception, i.e., react to it by changing the control flow, and subsequently *handle* it by executing a dedicated sequence of instructions. The client might already have continued its execution and no longer be in a position to propagate the exception. An exception such as this is known as an *asynchronous exception*. Its context consists of two parts: the *failed context* is the supplier's context; the *responsible context* is the context in which the client made the call. The exception is the result of a *failure* – the inability of the supplier to fulfill its *promise* towards its client.

Because of different design choices in each language, no approach to deal with asynchronous exceptions fits all situations. This paper classifies existing approaches in order to guide the development of new asynchronous exception mechanisms. It then presents the *accountability* framework to describe, comprehend, and compare asynchronous exception mechanisms: only within a well-defined range is the supplier obliged to report failures; it only does so if it is *held accountable*. Based on the classification and the accountability framework, this paper derives a mechanism for SCOOP (Simple Concurrent Object-Oriented Programming) [1], [2], an object-oriented programming model for concurrency. The contributions are:

- A classification of approaches to deal with asynchronous exceptions. The classification embeds existing classifiers into the larger picture and extends them with new classifiers for restricted asynchrony and propagation locations. It also discusses the strengths and weaknesses of the approaches.
- The accountability framework. The framework offers concepts to describe asynchronous exception mechanisms. Its applicability is shown on a number of mechanisms: μC++ [3], Erlang [4], and SaGE [5].
- A mechanism for asynchronous exceptions in SCOOP. The mechanism is derived from the classification and described using the accountability framework.

Section II presents the classification, and Section III introduces the accountability framework. Section IV gives information on SCOOP and its existing exception mechanism for non-concurrent programs. Section V presents the derived mechanism for SCOOP, and Section VI concludes with future work.

II. Classification of Approaches

The literature contains a wide variety of asynchronous exception mechanisms. Some mechanisms support multiple approaches to combine the advantages and compensate for the disadvantages of these approaches. Figure 1 shows a classification of approaches. On the top level, the classification differentiates between locations where an asynchronous exception can propagate. In case *the supplier propagates* the exception, the client does not have to expect an exception; it can be assured that the supplier will take care of any exceptions. Moreover, the supplier can handle the exception in the failed context. As a disadvantage, the client has no direct way of telling whether its call succeeded. Furthermore, the supplier might not always be able to draw the right conclusions on behalf of the client, as it does not have access to the responsible context. The handler can either be defined in the supplier or it can come from the client via an argument. Supplier-specific handlers are common: any mechanism where a supplier can deal with an exception in a local handler falls into this category; one example is SaGE [5]. Call-specific handlers are less common, but there are mechanisms, such as JR [6], using this approach. Call-specific handlers contain clean-up code specified by the client; in general, however, the client does not know how to clean up

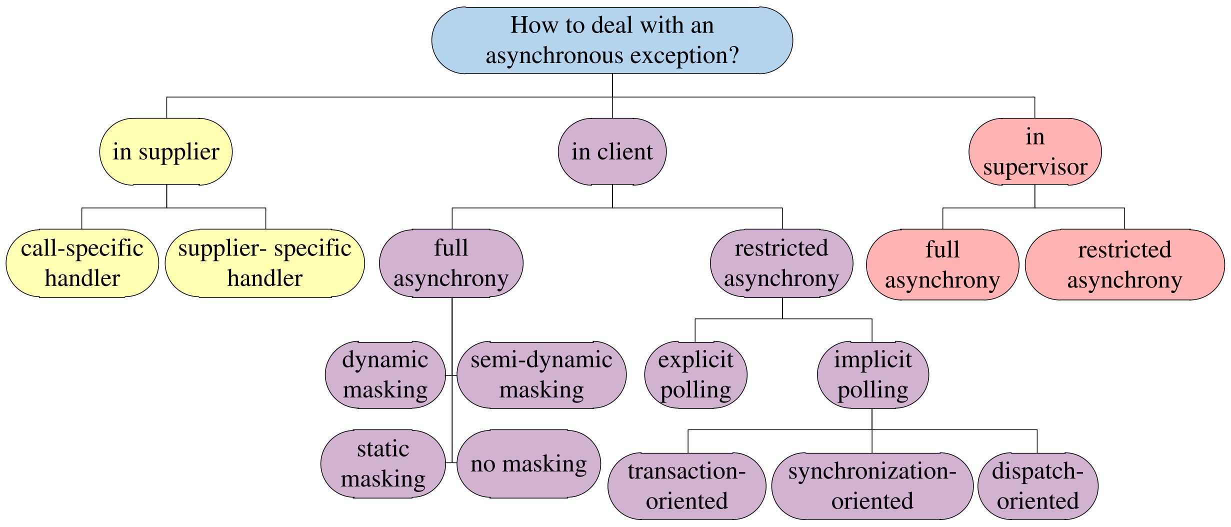

Fig. 1. Classification of approaches to deal with an asynchronous exception. The figure omits repeating the subdivisions for the supervisor-based approaches.

the supplier without violating information hiding principles. Supplier-specific handlers do not have this problem.

The case that the *client propagates* the exception supports a response in the responsible context, but in turn the client does not have access to the failed context. Table I lists the mechanisms that support this kind of propagation. The classification differentiates two cases: full asynchrony and restricted asynchrony. With *full asynchrony*, the exception can propagate anytime unless the client masked the exception. This approach is well-suited for reactive systems. However, developers have to program under the assumption of interruptibility. Furthermore, there is a risk of race conditions: the timing of the propagation depends on the relative speed between the client and the supplier. For masking, Krischer [7] differentiates several possibilities:

- *Dynamic masking*. Propagation is enabled until it is disabled by an opposing call. One example is Haskell [8] with its block and unblock operations.
- *Semi-dynamic masking*. Propagation is enabled only within a dedicated program block. Examples include Haskell [8] with its scopes, μC++ [3], and SR [9].
- *Static masking*. Propagation is enabled within a dedicated program block, but only when the client is statically in the block. Examples include Krischer's mechanism [7] and RTSJ [10].
- *No masking*. Propagation is always enabled. Examples include SaGE [5] with its client-based handlers, Erlang [4] with its exit signals over process links, Arche [11] where a client propagates exceptions as a result of incoming method calls, and ARMI [12] with its callback mechanism.

With *restricted asynchrony*, the exception can only propagate when the client polls the supplier, thus alleviating the interruptibility issue. This comes at the price of a delay between the moment the supplier raises the exception until the client propagates it, which might be unacceptable for reactive systems. Moreover, polling adds to the execution time even when there are no exceptions. This polling overhead can be diminished, in case the client waits for the supplier anyway to first finish its workload upon polling; as a further advantage of this, race conditions cannot occur because the supplier always knows about its failures before answering. The classification differentiates between *explicit polling*, where the client polls using dedicated polling constructs, and *implicit polling*, where the client polls in the course of executing constructs that are not solely dedicated to exceptions. Mechanisms that offer explicit polling include μC++ [3], Krischer's mechanism [7], Erlang [4] with its exit signal traps, ABCL/1 [13] where the client can wait for exception messages, Modula-3 [14] with its alert tests, and ProActive [15] with its exception queries and try block barriers. With the explicit polling approach, it is clear when the client polls. By restricting the propagation to dedicated points, developers can first focus on the normal behavior and deal with exceptions in a second step. Furthermore, developers are free to synchronize as often as necessary, thus mitigating unwanted delays. As a drawback, the language or its library must introduce a primitive for this purpose. The implicit polling approach does not suffer from this; developers must however be aware of the constructs that imply polling. To poll implicitly, there are three categories:

- *Transaction-oriented polling*. The client polls at specific points in a transaction. Examples include Argus [16] with its coenter statement and Avalon/C++ [17] with its costart statement. Arslan and Meyer's busy processor approach [18] for SCOOP also falls into this category: the client polls for exceptions while locking the supplier. This approach fits well for languages with transactions; furthermore, it is clear when the client must expect an exception. However, transaction-oriented polling alone is too coarse-grained for scenarios where the client has to

TABLE I
CLASSIFICATION OF MECHANISMS WITH PROPAGATION IN THE CLIENT. THE MARKS IN EACH ROW INDICATE THE SUPPORTED APPROACHES. THE HORIZONTAL LINES GROUP MECHANISMS WITH A SIMILAR FOCUS.

mechanism	full asynchrony				restricted asynchrony			
	dynamic masking	semi-dynamic masking	static masking	no masking	explicit polling	implicit polling		
						transaction	synchronization	dispatch
Haskell	X	X						
μC++		X			X		X	
SR		X						
Krischer			X		X			
RTSJ			X					
SaGE				X				
Erlang				X	X			
Arche				X				
ABCL/1					X		X	
Modula-3					X		X	
Argus						X	X	
Avalon/C++						X		
Dirty object						X		
X10							X	
RMIX							X	
Fortress							X	
Cilk Plus							X	
ProActive					X		X	
Rintala							X	
ARMI				X			X	
Failed object							X	X

know about exceptions at a location that is not related to a transaction.

- *Synchronization-oriented polling*. The client polls upon synchronization with the supplier. Examples include Modula-3 [14] with its alert joins and alert wait, Argus [19] with its synch statement, X10 [20] with its finish statement, RMIX [21], Fortress [22] with its val method, and Cilk Plus [23] with its sync statement. One example for SCOOP is Brooke and Paige's failed object mechanism [24]: after an object caused an exception, it is marked as failed; any future client propagates an exception upon a synchronous call. Further examples include languages with futures: μC++ [3], ABCL/1 [13], Argus [19] with its promises, ProActive [15], Rintala's C++ futures [25], and ARMI [12] with its delayed delivery mechanism. 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 the exception. With the synchronization-oriented approach, the client polls when it has to wait anyway; hence, it can diminish parts of the polling time. The approach also helps to mitigate unwanted delays: developers are free to poll as often as necessary. Race conditions cannot occur with this approach because the supplier finishes its workload upon polling, and hence, it always knows about its failures before answering. On the downside of synchronization-oriented polling, developers have to remember all possible implicit synchronization points.
- *Dispatch-oriented polling*. The client polls whenever it dispatches a message to the supplier. Brooke and Paige's failed object mechanism [24] also falls into this category; the client propagates an exception upon an asynchronous call as well. This approach helps to mitigate unwanted delays because every dispatch is a poll point. However, this approach can introduce race conditions because a dispatch does not wait for the supplier to finish the issued workload. Furthermore, developers have to be aware that each call causes polling.

Besides using the supplier or the client, a *supervisor* can be used to propagate the exception. The supervisor monitors the supplier to act upon exceptions. This approach supports the separation of normal behavior from exceptional behavior. Supervisors neither have access to the responsible context nor to the failed context; they can, however, have a global view on the system, as they can be supervising multiple suppliers. Similar to propagation in the client, the supervisor can be based on *full asynchrony* or *restricted asynchrony*. In μC++ [3], a supervisor comes into place when a supplier raises an exception in a task different from the client. Such a supervisor is in general not solely dedicated to monitoring; hence, μC++ makes use of full asynchrony with semi-dynamic masking and restricted asynchrony with explicit polling. In Erlang

[4], supervisors are dedicated to monitoring; hence, they are based on full asynchrony with no masking. The same is true for Ajanta [26] with its guardian agents. In Arche [11], a client specifies supervisors upon calling; the supervisors treat the exceptions as incoming method calls. In ABCL/1 [13], each call specifies a number of supervisors through complaint destinations; the supervisors use explicit polling to propagate exceptions. Modula-3 [14] supports supervisors with explicit polling and implicit polling.

Cooperative concurrent systems [27] such as guardian groups [28] are not discussed here. The classification can, however, be extended with cooperations as another propagation location.

III. Accountability

This section introduces the accountability framework and uses it to present a number of mechanisms from Section II in more detail.

A. Definition

Each of the mechanisms discussed in Section II differs in some way. On an abstract level, however, each of them addresses the same set of essential questions. This section introduces the accountability framework with concepts to answer these questions. In governance, Schedler [29] defines *accountability* as:

> A is accountable to B when A is obliged to inform B about A's (past or future) actions and decisions, to justify them, and to suffer punishment in the case of eventual misconduct.

In the context of asynchronous exceptions, a supplier is either accountable to its client, to a supervisor, or to no one, as it is the case with propagation in the supplier. For this reason, Definition 1 generalizes accountability for asynchronous exceptions: next to the client and the supplier, it introduces the *observer* as the computational unit to whom the supplier reports.

Definition 1: Consider a *client* that expects a *supplier* to perform some activities on its behalf. When the supplier fails, it performs a *local reaction*. An *observer* performs a *remote reaction* in response to the failure. The supplier is *accountable* to the observer for one of its activities if it is guaranteed to report any inability to fulfill the promise about the activity made. When the observer asks the supplier to report a failure, the observer is said to *hold* the supplier *accountable*. The guarantee is called an *accountability*. While the guarantee lasts, the accountability is said to be *alive*; afterwards it is *expired*.

The framework captures the essence of asynchronous exception mechanisms and provides concepts that can be instantiated to describe a particular mechanism. To instantiate the framework, the following questions must be answered:

- What is the supplier's local reaction?
- Who is the observer?
- When does an accountability become alive and when does it expire?
- When does the observer hold the supplier accountable?
- What is the observer's remote reaction?

For example, the framework can be used to describe the approaches from Section II. In future-based approaches (see Section II), an accountability becomes alive with the creation of a future, and it expires when the future gets disposed. The client is the observer; it holds the supplier accountable by accessing the future. For fully asynchronous mechanisms with semi-dynamic masking, an accountability exists between a client and each supplier; the client holds each supplier accountable while it executes blocks where propagation is enabled. With static masking, the client holds the suppliers only accountable while it is statically in a block with propagation. When exceptions propagate in supervisors, the supervisor is the observer. With propagation in the supplier, there is only a local reaction; hence there is no observer. These instantiations of the framework describe abstract approaches and not concrete mechanisms. To show how the framework can be applied to a concrete mechanism, Section III-B instantiates the framework for μC++ [3], Section III-C for Erlang [4], and Section III-D for SaGE [5].

The accountability framework and the classification from Section II can also be used as a guide to develop new mechanisms: the accountability framework defines the primary questions to be answered; the classification discusses possible answers from the literature. Section V demonstrates this on a new mechanism for SCOOP.

B. Accountability for μC++

μC++ [3] extends C++ [30] with language constructs for concurrency. A *task* can asynchronously spawn another task and retrieve results from it over a future. The supplier fails by executing a *throw* or *resume* statement.

What is the supplier's local reaction? Developers can enclose code blocks with two kinds of handlers: *termination* and *resumption* handlers. When a supplier executes a throw statement, it continues the execution in the nearest enclosing termination handler; after the supplier completes, it continues after the termination handler. When a supplier executes a resume statement, it executes the nearest enclosing resumption handler; after the supplier completes, it returns to the statement after the resume statement. A resume statement can optionally specify a different task to be informed about the failure, in which case the supplier remembers its failure and continues. In case the supplier issued a future to a client, it can additionally use the future to remember its failure.

Default handlers exist for the case no explicit termination or resumption handler has been defined. The default termination handler terminates the program; the default resumption handler initiates a search for a termination handler from the point of the initial resume statement.

Who is the observer? An observer comes into place when the supplier executes a resume statement that specifies another task; the specified task becomes the observer. This task can either be the client, or any other task that is dedicated to supervise the supplier. A task also becomes an observer when it receives a future from the supplier.

When does an accountability become alive and when does it expire? A supplier's accountability becomes alive with the creation of the supplier, and it expires when all its observers terminate. Consequently, the supplier might remain accountable after it terminates.

When does the observer hold the supplier accountable? An observer holds a supplier accountable when executing a block where masking is disabled, when executing a poll statement, or when querying a future from the supplier.

What is the observer's remote reaction? When the observer learns of a supplier's failure by accessing a future, it executes the nearest enclosing termination handler; the observer then continues after the termination handler. In all other cases, the observer behaves as if it executed a resume statement, i.e., it executes the nearest enclosing resumption handler and then returns to the statement after the resume statement.

C. Accountability for Erlang

Erlang [4] is a functional programming language whose concurrency support is based on the *actor model* [31]. An actor is a computational unit that responds to incoming asynchronous messages from other actors; in response to a message, the actor can either create other actors, send messages to other actors, or determine new behavior for future messages. In Erlang, actors are termed processes; these processes can be linked to each other.

What is the supplier's local reaction? When a supplier fails, it remembers the failure, terminates, and waits until it is held accountable.

Who is the observer? Dedicated supervisors are responsible for monitoring processes. Each supervisor has a list of *child specifications* that specifies the processes to be monitored. A supervisor can also monitor another supervisor, permitting a supervision tree. In addition to supervisors, a linked client is also an observer.

When does an accountability become alive and when does it expire? An accountability becomes alive with the creation of a process; it expires with the termination of all observers.

When does the observer hold the supplier accountable? A supervisor holds its monitored processes permanently accountable; it uses full asynchrony without masking. A linked client can have a *trap* to convert the failure of a supplier into an *exit message*. In case a linked client has a trap, it holds the supplier accountable when it responds to the exit message; otherwise, it holds the supplier permanently accountable.

What is the observer's remote reaction? The list of child specifications determines for each monitored process how the supervisor will react upon a failure; possible reactions are: restart or shutdown. For both restarts and shutdowns, there are a number of strategies to choose from. For instance, the supervisor can either restart just the failed process or all monitored processes. A linked client with a trap processes the supplier's failure as a message; a linked client without a trap fails.

D. Accountability for SaGE

SaGE [5] is an exception mechanism for languages using the *active object* pattern [32]. The active object pattern provides a solution to decouple an invocation in one object from the resulting execution in another object by giving an own thread of control to each object; the pattern is often used to implement the actor model [31]. In SaGE, each object scans its received messages. When it decides to accept a request in a message, it spawns a *service* that processes the message concurrently. The service, i.e., the client, can then send messages to other objects, causing further services, i.e., the suppliers, to be created.

What is the supplier's local reaction? Developers can attach handlers to services or objects. When a supplier fails, it first searches for a matching handler attached to itself. If no such handler exists, it searches for a matching handler attached to its owner object. In case the supplier finds a handler in this way, it executes the handler and terminates along with its own suppliers. Otherwise, it remembers the failure and waits until it is held accountable. After it is held accountable, the supplier terminates along with its suppliers.

Who is the observer? The observer is the client that called the faulty supplier.

When does an accountability become alive and when does it expire? An accountability becomes alive when an object creates a service. The accountability expires when the service terminates.

When does the observer hold the supplier accountable? A client holds a called supplier permanently accountable, i.e., even after it terminates; it uses full asynchrony without masking.

What is the observer's remote reaction? In addition to handlers attached to services and objects, developers can also attach handlers to requests. When a client learns about the failure of a supplier, it first searches for a matching handler that is attached to the corresponding request. If no such handler exists, the client evaluates a *resolution function*. The purpose of the resolution function is to aggregate and filter exceptions from different suppliers. If an exception remains, the client treats the supplier's failure as its own.

IV. SCOOP

This section gives an overview of SCOOP and its existing exception mechanism for non-concurrent programs, upon which the proposed mechanism builds.

A. Introduction to SCOOP

The starting idea of SCOOP is that every object is associated for its lifetime with a processor, called its *handler*. A *processor* is an autonomous thread of control capable of executing actions on objects. An object's class describes the possible actions as *features*. Any mechanism that can execute instructions sequentially, such as a CPU core or a thread, is suitable as a processor.

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*: the handler of *x* 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).

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. Assume that each object is handled by its own processor. One can then simplify the discussion using a single name to refer both to the object and its handler. For example, one can use "controller" to refer both to the controller object and its handler. 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. A *creation instruction* on such an entity creates an object on a new processor.

```
class CONTROLLER feature
  start (
    first_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
```

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. Such targets are called *controlled*. 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*; hence 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*).

These concepts require execution-time support, known as the SCOOP runtime. The following description is abstract; actual implementations may differ. Each processor maintains a *request queue* for feature requests from other processors. When a client performs a separate feature call, it enqueues a *separate feature request* to the supplier's request queue. The supplier processes the feature requests in the order of queuing. A non-separate feature call can be processed right away without going through the request queue: the processor creates a *non-separate feature request* for itself and processes it right away using its call stack. Whenever a processor is ready to let go of obtained locks, it issues an unlock request to each locked processor. Each locked processor unlocks itself as soon as it processed all previous feature requests.

B. Exceptions in Non-Concurrent Programs

The mechanism presented in this article relies on the *rescue-retry* approach for non-concurrent programs [1], summarized in this section. While this approach originates in Eiffel, languages such as Ruby [33] or Jass [34] have adopted it too. In the rescue-retry approach there are two possible responses to a failure:

- *Organized panic*. The supplier admits that the promise cannot be fulfilled. It brings all affected objects to a consistent state and reports the failure by passing the exception to its client.
- *Resumption*. The supplier attempts to fix the reasons for the failure and retries the execution.

Developers decide on the appropriate response by providing a rescue clause (**rescue** keyword) for each feature; features without an explicit rescue clause implicitly have an empty one. Whenever the supplier fails, it executes the rescue clause. The main purpose of the rescue clause is to put the affected objects back into a consistent state. If executed until the end, the supplier reports the failure to the client (organized panic). A rescue clause may terminate by executing a retry instruction (**retry** keyword), which restarts the feature from the beginning (resumption); the idea is for the rescue clause to change some of the context to ensure that the new execution tries some other path.

Consider the class *SEARCHER*, whose feature *search* finds a solution using a random search with a new seed. If the new seed is invalid, the searcher raises an exception, and the rescue clause restores the seed before reporting the failure. The postcondition (**ensure** keyword) expresses the promise.

```
class SEARCHER feature
  seed: INTEGER -- The seed used in the random search.
  solution: STRING -- The result.

  search
    do
        -- Get the seed from atmospheric noise.
      seed := atmospheric_noise
      if seed >= 0 then
        solution := random_solution (seed) -- Search.
```

```
            else
                raise_exception -- Fail.
            end
        ensure not solution = Void -- The postcondition.
        rescue seed := 0 -- Restore consistency.
        end

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

V. Asynchronous Exception Mechanism for SCOOP

This section presents an asynchronous exception mechanism for SCOOP. The main goal of SCOOP is to provide concurrency mechanisms that are simple to understand and to use [1], [2]; the asynchronous exception mechanism must be designed in the same spirit. This is part of the following requirements:

- *Comprehensibility*. The mechanism must be easy to understand.
- *Compatibility*. It must be compatible with the existing mechanism for non-concurrent programs (see Section IV-B).
- *Consistency*. It must guarantee the consistency of a failed supplier.

This section first instantiates the accountability framework. It then presents an implementation, an example, and a comparison with related mechanisms.

A. Accountability for SCOOP

What is the supplier's local reaction? To satisfy the compatibility and the consistency requirements, the supplier must execute its rescue clause to reestablish its consistency, as explained in Section IV-B. The rescue clause is a supplier-specific handler. Without a retry instruction, the supplier remembers the failure and cleans up: it purges any remaining feature requests from the client because it no longer makes sense to execute them after the failure; it also releases the locks. It then waits until it is held accountable or until its accountability expires.

Who is the observer? The propagation in a supervisor approach would not be suitable for SCOOP because it is not compatible with the existing exception mechanism for non-concurrent programs; by definition, a non-concurrent program cannot have a supervisor working in parallel to the main execution. The client is a better observer; with this choice, the mechanism meets the compatibility requirement because the non-concurrent mechanism also displays a division of labor between the supplier and the client.

Lock passing complicates the situation. When a client p passes a lock on a supplier to another processor p', p' temporarily becomes the new client. However, as long as p' is not the observer, it cannot hold the supplier accountable. Yet, it has a legitimate need to do so. One way to resolve this is to *transfer the accountability* during lock passing, i.e., when p passes the lock on the supplier to p', then p also passes the supplier's accountability to p'; when p' returns the lock to p, p' also returns the accountability. For the duration of lock passing, p' is the observer. Accountability transfer extends the responsible context to include all feature executions along the lock passing chain.

When does an accountability become alive and when does it expire? For this question, it is helpful to consider when a client *can assume the promise* of a supplier to fulfill the postcondition of an issued feature call: this is the case as long as the client keeps a lock on the supplier; as soon as the client gives up the lock, it no longer has any guarantees because another processor could have modified the supplier. A client *relies on a promise* of a supplier by performing a synchronous feature call, while it can still assume the promise. We argue that a supplier's failure only matters when the client relies on the promise of the supplier. This view is compatible with the view on asynchrony in SCOOP: the supplier is not required to establish the promise of an asynchronous feature call immediately, but when the client relies on the promise [1], [2]. Therefore, a failure does not matter anymore when the client can no longer assume the promise. For the proposed mechanism, this means that a supplier is accountable from the moment the client locks the supplier to the moment the client gives up the lock. As a consequence of this, an accountability persists throughout a chain of lock passing operations. By letting an accountability expire, SCOOP programs can become more fault-tolerant. Because a supplier always restores the consistency of a failed object, this objects can safely be accessed again.

One question remains: does it make sense for a client not to rely on a promise? It does, for example, when the client spawns multiple suppliers but only wants the results from some of them. In Section IV, the log might not care about the second searcher if the first searcher found a satisfying solution. In absence of a cancellation mechanism, the log must ignore the second searcher by not synchronizing with it. Section V-C shows this example in more detail.

Mechanisms using futures, e.g., ProActive [15], also allow an accountability to expire: when the client does not query the future, the accountability expires [35]. For some programs, however, it might not be safe to forget about failures. The next paragraph includes a solution for these cases.

When does the observer hold the supplier accountable? Restricted asynchrony is suited for SCOOP because it alleviates the interruptibility issue and thus keeps the mechanism comprehensible. Next, implicit, synchronization-oriented polling fits well. SCOOP has three types of synchronization points: query calls, non-separate calls, and calls with lock passing. With this choice, each such call triggers the client to hold its supplier accountable. This choice makes the proposed mechanism compatible: in a non-concurrent program, every feature call is non-separate; hence every feature call becomes a poll point in the proposed mechanism, just as in the underlying mechanism. Furthermore, this choice ensures that the supplier finishes issued feature calls before reporting, thus preventing race conditions (see Section II). This would not be guaranteed with dispatch-oriented polling, where the client would hold the supplier accountable during asynchronous feature calls as well.

An accountability expires as soon as the client gives up the lock; with the accountability, any past failure disappears as well. There are some programs where failures must not disappear. For this reason, the mechanism has a *safe mode*, in which a client holds its locked suppliers accountable before their accountabilities expire. This mode is based on transaction-oriented polling. It ensures that no failure is lost. However, it also reduces the potential for concurrency: the client can no longer just asynchronously issue unlock requests and then continue; it first has to wait until the suppliers finished.

In summary, a client must synchronize with a supplier to learn about a failure. 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 local reaction, for example with a callback to the client.

What is the observer's remote reaction? When the client learns about a failure of its supplier, it treats the failure as its own failure. Consequently, it assumes the role of a supplier for its own client. This ensures compatibility.

B. Implementation

Figure 2 shows the interactions between a client p and suppliers $\{q_1, \ldots, q_n\}$ for a feature request f. The event-based notation follows Cachin et al. [36]. Every processor runs one instance of the algorithm along with its normal loop, in which it processes its feature requests in $requests$, executes a rescue clause upon a failure, and retries the execution after a retry instruction. The declaration **upon event** $\langle$Event_name $|$ arguments$\rangle$ **such that** *condition* **do** defines an event handler with arguments; a processor executes the handler in a mutually exclusive way when the named event occurred and the condition is satisfied. The instructions **trigger** and **wait** initiate respectively wait for an event; **rescue** terminates an event handler and initiates the rescue clause. Processors communicate with each other over reliable and ordered links; messages are enclosed in square brackets. For brevity, we omitted the scheduler and the bookkeeping of locks in the client.

Whenever a client locks a supplier (see Feature_application_start), the supplier remembers that it is locked and accountable (see $\langle$Get $|\, p$, [LOCK]$\rangle$). The supplier does not have to remember the observer because the existing runtime system ensures that only the observer can access the supplier. For the same reason, the accountability does not need to be transferred explicitly during lock passing. In case the supplier fails, it executes the associated rescue clause to reestablish its consistency. Without a retry instruction (see Rescue_clause_end), the supplier remembers the failure and cleans up: it purges any remaining feature requests from the client and releases the locks as it ends the feature application (see Feature_application_end). It then waits until the client holds it accountable or asks it to unlock. The client holds the supplier accountable in the course of a

```
upon event ⟨Initialize⟩ do
    $locked$ := false; $accountable$ := false;
    $failed$ := false; $requests$ := ();

upon event ⟨Feature_application_start | $f$, $\{q_1, \ldots, q_n\}$⟩ do
    forall the $q_i \in \{q_1, \ldots, q_n\}$ do trigger ⟨Send | $q_i$, [LOCK]⟩;

upon event ⟨Feature_call | $f$, $q_i$, $is\_synchronous$⟩ do
    trigger ⟨Send | $q_i$, [FEATURE_REQUEST, f]⟩;
    if $is\_synchronous$ then
        trigger ⟨Send | $q_i$, [HOLD_ACCOUNTABLE]⟩;
        wait ⟨Get | $q_i$, [REPORT, s]⟩;
        if $s$ = fail then rescue;
    end

upon event ⟨Feature_body_end | $f$, $\{q_1, \ldots, q_n\}$, $in\_safe\_mode$⟩ do
    if $in\_safe\_mode$ then
        forall the $q_i \in \{q_1, \ldots, q_n\}$ do
            trigger ⟨Send | $q_i$,
            [HOLD_ACCOUNTABLE]⟩;
            wait ⟨Get | $q_i$, [REPORT, s]⟩;
            if $s$ = fail then rescue;
        end
    end

upon event ⟨Feature_application_end | $f$, $\{q_1, \ldots, q_n\}$⟩ do
    forall the $q_i \in \{q_1, \ldots, q_n\}$ do trigger ⟨Send | $q_i$, [UNLOCK]⟩;

upon event ⟨Get | $p$, [LOCK]⟩ do
    $locked$ := true; $accountable$ := true;

upon event ⟨Get | $p$, [FEATURE_REQUEST, $f$]⟩ do
    if $\neg failed$ then $requests$ := $requests \bullet f$;

upon event ⟨Rescue_clause_end⟩ do
    $failed$ := true; $requests$ := ();

upon event ⟨Get | $p$, [HOLD_ACCOUNTABLE]⟩ such that $requests$ = () do
    if $failed \wedge accountable$ then trigger ⟨Send | $p$, [REPORT fail ]⟩;
    else trigger ⟨Send | $p$, [REPORT success ]⟩;

upon event ⟨Get | $p$, [UNLOCK]⟩ do
    $accountable$ := false;

upon $\neg accountable \wedge$ $requests$ = () do
    $locked$ := false; $failed$ := false;
```

Fig. 2. Asynchronous exception mechanism

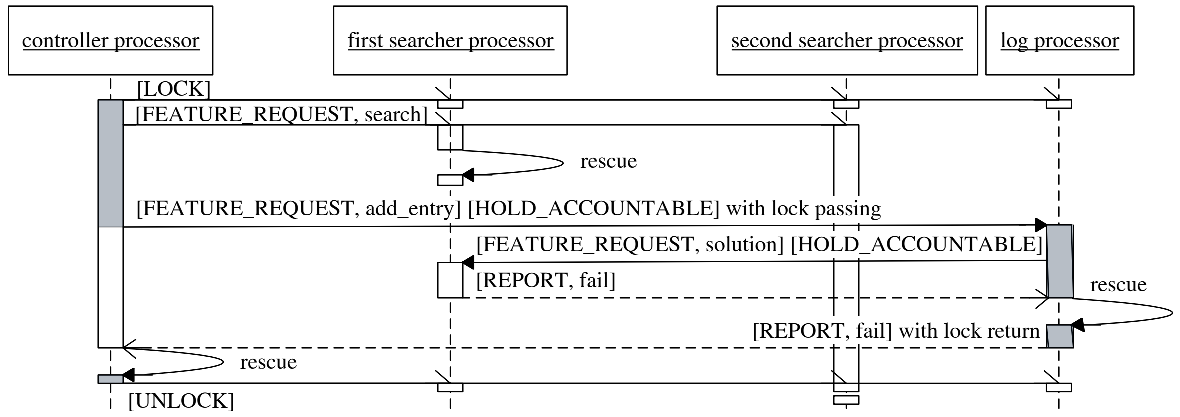
controller processor
first searcher processor
second searcher processor
log processor
[LOCK]
[FEATURE_REQUEST, search]
rescue
[FEATURE_REQUEST, add_entry] [HOLD_ACCOUNTABLE] with lock passing
[FEATURE_REQUEST, solution] [HOLD_ACCOUNTABLE]
[REPORT, fail]
rescue
[REPORT, fail] with lock return
rescue
[UNLOCK]

Fig. 3. The interactions between the controller, the searchers, and the log. The shaded areas indicate the searchers' accountabilities.

synchronous feature call (see Feature_call) or at the end of a feature body (see Feature_body_end) in safe mode; in both cases the supplier makes a report (see $\langle$Get $| p$, [HOLD_ACCOUNTABLE]$\rangle$). As soon as the client asks the supplier to unlock (see Feature_application_end), the supplier dismisses its accountability (see $\langle$Get $| p$, [UNLOCK]$\rangle$); when it unlocks (see $\neg accountable \wedge$ request_queue = ()), it forgets about its failure. Without a failure, the supplier must finish pending feature requests before reporting or unlocking.

C. Example

Consider again the controller from Section IV-A and the searchers from Section IV-B. Figure 3 shows an execution in which the controller calls the two searchers and passes them to the log. Meanwhile, the first searcher fails. Due to lock passing, the searchers' accountabilities persist throughout *start* and *add_entry*.

The following class describes the log that writes the solutions to disk:

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

By calling *solution* synchronously, the log holds the first searcher accountable, fails, and reports the failure to the controller, who is waiting for the locks to return.

D. Comparison

In Arslan and Meyer's busy processor mechanism [18], a supplier becomes busy when it fails; only the client that made the faulty feature call can relock the supplier, in which case it holds the supplier accountable. For this to work, the accountability persists beyond the locking context. This mechanism causes an exception to propagate in a context where the client cannot assume the promise of the corresponding feature call. Furthermore, it does not consider lock passing.

Brooke and Paige [24] propose three options from which developers can choose. The first two options are radical: the first option halts the entire system as a response to an asynchronous exception; the second option ignores any asynchronous exceptions. The third option marks failed objects permanently; any future client holds the failed object's handler accountable upon each feature call, i.e., each processor is an observer of each other processor, and an accountability persists until the end. The third option is limiting because it prevents cases where it would be valid to access the failed object once again. In particular, a different context might only assume that the failed object is consistent; it might not care about past failures. Hence, it would be valid to access the failed object again, provided that the failed object's consistency has been restored. Furthermore, the third option permits race conditions between the client and the supplier because the supplier is not permitted to finish its workload. On the plus side, polling at each feature call reduces the propagation delay.

VI. Conclusion

Deficiencies in exception handlers can cause severe failures. To support developers in writing correct error handlers, especially when reasoning about concurrent code, a comprehensible model of exception handling is critical. We introduced the accountability framework with concepts to describe asynchronous exception mechanisms, and we presented a classification of approaches along with a discussion. We used them to derive a mechanism for asynchronous exceptions in SCOOP. In future work, we want to extend the mechanism with exception resolution [37]. A client can have more than one supplier, and each supplier can raise an asynchronous exception. In the proposed mechanism, the client handles the exception of the first supplier with whom it synchronizes. With exception resolution, the client can collect exceptions from multiple suppliers and handle them together. We are also extending the formal semantics for SCOOP [38] with the mechanism.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA). Earlier work on the SCOOP model has also benefited from grants from the Swiss National Foundation and Microsoft (Multicore award).

REFERENCES

[1] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[2] P. Nienaltowski, "Practical framework for contract-based concurrent object-oriented programming," Ph.D. dissertation, ETH Zurich, 2007.

[3] P. A. Buhr, "μC++ annotated reference manual," University of Waterloo, Tech. Rep., 2012.

[4] Ericsson, "Erlang/OTP system documentation," Ericsson, Tech. Rep., 2012.

[5] C. Dony, C. Urtado, and S. Vauttier, "Exception handling and asynchronous active objects: Issues and proposal," in *Advanced Topics in Exception Handling Techniques*, 2006, pp. 81–100.

[6] R. A. Olsson and A. W. Keen, *The JR Programming Language*. Kluwer Academic Publishers, 2004.

[7] R. Krischer, "Advanced concepts in asynchronous exception handling," Ph.D. dissertation, University of Waterloo, 2010.

[8] S. Marlow, S. Peyton-Jones, A. Moran, and J. Reppy, "Asynchronous exceptions in Haskell," *SIGPLAN Notices*, vol. 36, no. 5, pp. 274–285, 2001.

[9] D. T. Huang and R. A. Olsson, "An exception handling mechanism for SR," *Computer Languages, Systems & Structures*, vol. 15, pp. 163–176, 1990.

[10] G. Bollella, J. Gosling, B. M. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[11] V. Issarny, "An exception handling mechanism for parallel object-oriented programming: towards reusable, robust distributed software," *Journal of Object-Oriented Programming*, vol. 6, no. 6, pp. 29–39, 1993.

[12] R. R. Raje, J. I. Williams, and M. Boyles, "Asynchronous remote method invocation (ARMI) mechanism for Java," *Concurrency and Computation: Practice and Experience*, vol. 9, no. 11, pp. 1207–1211, 1997.

[13] Y. Ichisugi and A. Yonezawa, "Exception handling and real time features in an object-oriented concurrent language," in *Concurrency: Theory, Language, and Architecture*, 1991, pp. 91–109.

[14] G. Nelson, *Systems Programming With Modula-3*. Prentice Hall, 1991.

[15] D. Caromel and G. Chazarain, "Robust exception handling in an asynchronous environment," in *Workshop on Exception Handling in Object-Oriented Systems*, 2005.

[16] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl, "Argus reference manual," Massachusetts Institute of Technology, Tech. Rep., 1995.

[17] J. M. Wing, M. Herlihy, S. Clamen, D. Detlefs, K. Kietzke, R. Lerner, and S.-Y. Ling, "The Avalon/C++ programming language (version 0)," Carnegie Mellon University, Tech. Rep., 1988.

[18] V. Arslan and B. Meyer, "Asynchronous exceptions in concurrent object-oriented programming," in *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-Like Languages*, 2006, pp. 62–70.

[19] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," *SIGPLAN Notices*, vol. 23, pp. 260–267, 1988.

[20] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification," IBM, Tech. Rep., 2011.

[21] D. Kurzyniec and V. S. Sunderam, "Semantic aspects of asynchronous RMI: the RMIX approach," in *International Parallel and Distributed Processing Symposium*, 2004.

[22] E. Allen, D. Chase, J. Hallet, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, "The Fortress language specification," Sun, Tech. Rep., 2008.

[23] Intel, "Cilk Plus language extension specification," Intel, Tech. Rep., 2011.

[24] P. J. Brooke and R. F. Paige, "Exceptions in concurrent Eiffel," *Journal of Object Technology*, vol. 6, no. 10, pp. 111–126, 2007.

[25] M. Rintala, "Handling multiple concurrent exceptions in C++ using futures," in *Advances in Exception Handling Techniques*, 2006, pp. 62–80.

[26] A. R. Tripathi and R. Miller, "Exception handling in agent-oriented systems," in *Advances in Exception Handling Techniques*, 2000, pp. 128–146.

[27] J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery," in *Symposium on Fault-Tolerant Computing*, 1995, pp. 499–508.

[28] R. Miller and A. R. Tripathi, "The guardian model and primitives for exception handling in distributed systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 1008–1022, 2004.

[29] A. Schedler, "Conceptualizing accountability," in *The Self-Restraining State: Power and Accountability in New Democracies*, A. Schedler, L. Diamond, and M. F. Plattner, Eds. Lynne Rienner, 1999, pp. 13–28.

[30] B. Stroustrup, *The C++ Programming Language*, 3rd ed. The C++ Programming Language, 1997.

[31] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *International Joint Conference on Artificial Intelligence*, 1973, pp. 235–245.

[32] H. R. F. B. Douglas Schmidt, Michael Stal, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[33] Information-technology Promotion Agency, "Ruby," Information-technology Promotion Agency, Tech. Rep., 2010.

[34] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass - Java with assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, 2001.

[35] B. Morandi, S. Nanz, and B. Meyer, "Can asynchronous exceptions expire?" in *Workshop on Exception Handling*, 2012, pp. 4–6.

[36] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer, 2011.

[37] R. H. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE Transactions on Software Engineering*, vol. 12, pp. 811–826, 1986.

[38] B. Morandi, S. Nanz, and B. Meyer, "A formal reference for SCOOP," in *Empirical Software Engineering and Verification*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds. Springer, 2012, vol. 7007, pp. 89–157.