Session Summary: Exceptions and Concurrency

Chair: Alexander Romanovsky
Rapporteur: Jörg Kienzle

1 Motivations

The session chair started the session by pointing out that Ada has very elaborate features for handling concurrency, but that the exception handling model is basically sequential. Some of the position papers and also his own experience provide evidence that concurrent systems often require concurrency-aware exception handling features, or even means to express collaborative exception handling.

William Bail defined the situation very nicely: in sequential programming, any statement might depend on the previous statement. Sequential exception handling addresses this situation. In a concurrent setting, where exceptions might be raised in some other collaborating task, the dependency may be realized only later on, namely at the point where one of the tasks wants to talk to or use the results of the task that has got the exception.

2 Complex Concurrent Systems

2.1 Contributions

The papers "Using Ada Exceptions to Implement Transactions" and "Implementing Exceptions in Open Multithreaded Transactions based on Ada 95 Exceptions" address exception handling in transactions involving multiple tasks. Both models require that when an exception is raised in one of the tasks participating in a transaction, the other participants are informed of this event. The model described in the first paper even performs exception resolution in case of concurrently raised exceptions.

Both papers present an implementation in Ada, in which they rely on the programmer to follow programming guidelines that allow the underlying run-time support to get control when an exception is raised. The procedural interface requires two nested Ada blocks to associate an exception context with a transaction, which is quite cumbersome. Unfortunately, controlled types, which would allow a more elegant interface, are not sufficiently powerful.

Juan Antonio de la Puente mentioned a project, in which a set of tasks, named a *recovery group*, could cooperate to perform a joint activity. If one of the tasks encountered a problem, all tasks of the group were suspended, and a new single task would try and perform recovery.

Alexander Romanovsky mentioned that in some collaborative schemes such as atomic actions, an exception raised in one of the participant tasks should affect the execution of the other participating tasks. In such a situation, preemptive models will interrupt all other participating tasks and initiate cooperative error recovery. Such systems could benefit from a feature that allows a task to raise an exception in some other task. Alexander presented excerpts from the Ada 83 Rationale that discuss a special `Failure` exception to be raised in any task using the following syntax (this idea did not find its way into Ada 83):

```
raise Task.Failure;
```

2.2 Discussion

Alan Burns stated that the current Ada model of concurrency does not provide explicit features for the higher level cooperation schemes such as conversations, atomic actions, or transactions. It would therefore seem strange to provide features for the high level collaborative exception handling. Currently, the only direct interaction between (two) tasks are the rendezvous and the creation of a task. During a rendezvous, the behavior of exceptions is clearly defined, e.g. they are raised in both the calling and the called task. An exception during task creation is signalled to the parent by raising `Tasking_Error`. A weak form of synchronization also exists during task termination, where a master waits for all its dependent tasks to complete.

After some discussion, the workshop attendees seemed to be convinced that since there are a lot of different higher level collaboration models, it is not helpful to provide support for some of them. It would be more beneficial for the language to provide basic features, and let the programmers construct higher level features on their own (see section 5).

The discussion on raising exceptions in other tasks did not last very long. Several workshop participants pointed out that providing the ability to asynchronously notify any task is a nightmare for a programmer. The resulting race conditions would make it close to impossible to produce a correct program. In addition, all local data would have to be stored in protected objects, just in case. This would be against the fundamental principle that the execution of the normal code should not be accompanied by expensive actions that are only necessary when an exception is raised. All participants felt that it would be wrong to interrupt any task at any point of its execution; we all agreed that losing exceptions is a bad idea and a sign of a bad design.

It was argued that Ada already offers a feature that is intended for interrupting tasks: if a task agrees to be notified by some external entity, it can use Asynchronous Transfer of Control (ATC). Although there are some concerns here because non exception features are used for handling exceptions. For example, in this case, the normal code is not well separated from the exception handling code and the entry calls are used for propagating both normal and abnormal events because exceptions are not triggering events in Ada. Moreover, involving several tasks in the handling would require complex error-prone programming, as a single triggering event cannot trigger multiple tasks.

3 Tasks and Exceptions

3.1 Contributions

The paper "Exception Support for the Ravenscar Profile", identified the need for clarifying some situations related to tasking and exceptions in the Ravenscar profile. In standard Ada, an exception during task elaboration will propagate `Tasking_Error` to the parent unit. In the Ravenscar profile only library level tasks are allowed. In case an exception is raised during elaboration of a task in Ravenscar, the task is silently terminated, which is clearly not an acceptable solution. A similar situation is encountered with respect to task termination. In the standard Ada, a task that terminates exceptionally just silently disappears, whereas in the Ravenscar profile, termination of a task is considered a bounded error, and must therefore be documented. Raising `Program_Error` is explicitly allowed as a possible outcome of such an error, but this occurrence cannot be handled, as the parent unit for any task is a library unit. Again, the task will just terminate silently.

3.2 Discussion

Most participants of the workshop felt that the fact that unhandled exceptions cause a task to terminate unnoticed is clearly wrong. Exceptions should not be forgotten or lost! A future version of Ada should provide a feature that allows a programmer to state what kind of actions should be taken in this case.

The following idea was briefly discussed: a child task might have in its specification the exceptions that will be propagated to the parent task when it terminates with an exception (this clearly extends the implicit interface existing between the child task and the parent). Another approach is to extend the language with more elaborate task attributes that make it possible to distinguish between tasks terminated with and without exceptions.

4 Comparison with Java

4.1 Contributions

Alfred Strohmeier presented the main concurrency features of Java, and how they address exceptions. In Java, threads are on a lower level of abstraction than Ada tasks, and therefore should not be compared directly.

Every Java object has an associated lock. Any synchronized statement must acquire the lock before it can execute. A wait set, on which threads waiting for the lock are queued, is therefore associated with each object. Obviously, such a low-level mechanism does not compare with the protected objects provided in Ada, but it can be used as a basic building block to construct higher level abstractions.

Java has a very interesting feature called *thread groups*. When a thread is created, it is possible to associate it with a thread group. Thread groups are organized as a hierarchy. When a thread terminates due to an unhandled exception, it implicitly invoke the `uncaughtException` method of the thread group it belongs to:

```
uncaughtException (Thread t, Throwable e);
```

This is the last action of the thread before it dies. By overriding the default behavior of this method, a programmer can for instance log the cause of death of the thread. Other members of the group can later on call other methods of the thread group to extract this information again.

4.2 Discussion

Of course it can be argued that this `uncaughtException` method is just some other form of a `when others` handler placed at the end of the task body, which then calls a library level procedure. The advantage of the Java model however is that whatever a programmer does, a thread can never die unnoticed. In Ada, a programmer can forget to include the `when others` clause.

Tullio Vardanega mentioned that the Java model can rely on garbage collection. An Ada solution should not rely on any implicit object allocation. Otherwise, since no one feels responsible for the implicitly allocated memory, no one will free it.

An idea of extending entry signatures with the exceptions that can be propagated to the caller was briefly discussed, but it was felt that this would require in the first instance resolving problems related to extending the subprogram signatures.

5 Exceptional `Finalize`

Controlled types allow a programmer to get control when an object goes out of scope. Before the object is deallocated, its `Finalize` procedure is invoked. This is done in case of normal termination of the block containing the object declaration, but also if an unhandled exception is propagated to the enclosing scope.

The workshop attendees are convinced that a feature that allows a programmer to distinguish between "normal" and "exceptional" finalization would provide a basic building block that addresses most problems mentioned in this session.

Such a feature would not only be used to implement higher level collaborative exception handling, but also to perform notification in case of task termination due to unhandled exceptions. Using the generic package `Ada.Task_Attributes`, a controlled object can be associated with each task in the system. When a task dies due to an unhandled exception, the "exceptional" finalize procedure of the controlled type is invoked automatically.

A possible extension to a future version of Ada could overload the `Finalize` procedure of controlled types:

```
procedure Finalize (Object: in out Controlled;
                    E: in Exception_Occurrence);
```

The `Exception_Occurrence` parameter contains the currently "pending" exception, if any.

Backward compatibility issues with Ada 95 were not discussed during the workshop, but there were a number of discussions during the breaks on the following days of the conference. Several solutions are possible, but the normal `Finalize` must be called in any case for backward compatibility. One solution would be to define two new procedures for controlled types, `Finalize_Normal` and `Finalize_Exceptional`. Only one of them is called during finalization, followed by a call to the old `Finalize`. Another possibility would be to always call the exceptional finalize, but pass a `Null_Occurrence` in the exception occurrence parameter in case of normal finalization, or just not call the exceptional finalize in case of normal finalization. Another possibility is to add new controlled types providing this new functionality, and leaving the old ones as they are. This, of course, would prevent extensions of existing controlled type hierarchies using this new feature.

6 Conclusion

The workshop attendees agreed that there are too many different higher level collaboration paradigms in concurrent programming, and it would be wrong to support some of them in the programming language. A better idea is to provide basic building blocks, making it possible to construct higher level models in an elegant and straightforward way. The participants felt that the exceptional `Finalize` procedure of controlled types could solve a number of the problems discussed during this workshop session. A concrete proposal must still be elaborated.