Fault Model Design Space for Cooperative Concurrency *

Ivan Lanese[1], Michael Lienhardt[1], Mario Bravetti[1], Einar Broch Johnsen[2], Rudolf Schlatte[2], Volker Stolz[2], and Gianluigi Zavattaro[1]

[1] Focus Team, Università di Bologna/INRIA, Italy
{lanese,lienhard,bravetti,zavattar}@cs.unibo.it
[2] Department of Informatics, University of Oslo, Norway
{einarj,rudi,stolz}@ifi.uio.no

Abstract. This paper critically discusses the different choices that have to be made when defining a fault model for an object-oriented programming language. We consider in particular the ABS language, and analyze the interplay between the fault model and the main features of ABS, namely the cooperative concurrency model, based on asynchronous method invocations whose return results via futures, and its emphasis on static analysis based on invariants.

1 Introduction

General-purpose modeling languages exploit *abstraction* to reduce complexity [20]: modeling is the act of describing a system succinctly by leaving out some aspects of its behavior or structure. Software models primarily focus on the functional behavior and the logical composition of the software. Modeling formalisms can have varying levels of detail and can express structural properties (for example UML diagrams), interactions (π-calculus), or the effects of functions or methods (pre- and post-conditions), etc.

Concurrent and distributed systems demand flexible communication forms between distributed processes. While object-orientation is a natural paradigm for distributed systems [15], the tight coupling between objects traditionally enforced by method calls may be criticized. Concurrent (or active) objects have been proposed as an approach to concurrency that blends naturally with object-oriented programming [1, 22, 32]. Several slightly differently flavored concurrent object systems exist for, e.g., Java [3, 30], Eiffel [5, 26], and C++ [25]. Concurrent objects are reminiscent of Actors [1] and Erlang processes [2]: objects are inherently concurrent, conceptually each object has a dedicated processor, and there is at most one activity in an object at any time. Thus, concurrent objects encapsulate not only their state and methods, but also a single (active) thread of control. In the concurrent object model, *asynchronous method calls* may be used to better combine object-orientation with distributed programming by reducing the temporal coupling between the caller and callee of a method, compared

* Partly funded by the EU project FP7-610582 ENVISAGE

to the tightly synchronized (remote) method invocation model (of, e.g., Java RMI [27]). Intuitively, asynchronous method calls spawn activities in objects without blocking execution in the caller. Return values from asynchronous calls are managed by *futures* [14,23,32]. Asynchronous method calls and futures have been integrated with, e.g., Java [11,19] and Scala [13] and offer a large degree of potential concurrency for deployment on multi-core or distributed architectures.

ABS is a modeling language targeting distributed systems [17]; the language combines concurrent objects and asynchronous method calls with *cooperative scheduling* of method invocations. In ABS the basic unit of computation is the *concurrent object group* (cog): a cog provides to a group of objects a shared processor. Method invocations on an object of a cog instantiate a new task that requires the cog's processor in order to execute. Cooperative scheduling allows tasks to suspend in a controlled way at explicit points in the code, so that other tasks of the object can execute. The **suspend** and **await** commands are used to explicitly release the processor: the difference between the two commands is that **await** has an associated boolean guard expressing under which condition the task should be re-activated by the scheduler. Asynchronous method invocations are used among objects belonging to different cogs; at each asynchronous method invocation a *future* is instantiated to store the return value. Futures are first class citizens in ABS and are accessed via a **get** command; **get** is blocking because a task, executing **get** on a future of a method invocation which has not yet completed, blocks and keeps the processor until the future is written. To avoid keeping the processor, one can use an **await** `f?` to ensure that future `f` contains a value.

ABS has a formal, executable semantics; ABS models can be run on a variety of backends and can be verified using the KeY proof checker [4]. In particular, asynchronous method calls and cooperative scheduling allow the verification of distributed and concurrent programs by means of sequential reasoning [8]. In ABS this is reflected in a proof system for local reasoning about objects where the class invariant must hold at all scheduling points [9]. Although ABS targets distributed systems, a notable abstraction of the language design is that faults are currently not considered part of the behavior to be modeled. On the other hand, dealing with faults is an essential and notoriously difficult part of developing a distributed system; this difficulty is exacerbated by the lack of clear structuring concepts [7]. A well-designed model is essential to understand potential faults and reason about robustness, especially in distributed settings. Thus, it is interesting to extend a modeling language such as ABS in order to model faults and how these can be resolved during the system design.

It is common in the literature to distinguish errors due to the software design (sometimes called *faults*) from random errors due to hardware (sometimes called *failures*). For software deployed on a single machine, such hardware failures entail a crash of the program. A characteristic of distributed systems is that failures may be *partial* [31]; i.e., the failure may cause a node to crash or a link to be broken while the rest of the system continues to operate. In our setting, a strict separation between faults and failures may seem contrived, and

we will refer to unintended behavior caused by both the software and hardware as faults. A fault is *masked* if the fault is not detected by the client of the service in which the fault occurs. In hierarchical fault models, faults can propagate along the path of service requests; i.e., a fault at the server level can result in a (possibly different) fault at the client level. In a synchronous communication model, a client object can only send one method call at the time whereas in an asynchronous communication model, the client may spawn several calls. Thus, it need not be clear for a client object which of the spawned calls resulted in a specific fault in the asynchronous case. However, asynchronous method calls in ABS allow results to be *shared* before they are returned: futures are first-class citizens of ABS and may be passed around. First-class futures give rise to very flexible patterns of synchronization, but they further obfuscate the path of service requests and thus of fault propagation.

This paper discusses an extension of the semantics of the ABS modeling language to incorporate a robust fault model that is both amenable to formal analysis and familiar to the working programmer. The paper considers how faults can be introduced into ABS in a way which is faithful to its syntax, semantics, and proof system, and discusses the appropriate introduction of faults along three dimensions: fault representation (Section 2), fault behavior (Section 3), and fault propagation (Section 4).

2 How Are Faults Represented?

Exceptions are the language entities corresponding to faults in an ABS program's execution. ABS includes two kinds of entities which in principle can be used to represent faults: *objects* and *datatypes* (datatypes [16] are part of the functional layer of ABS, and abstract simple, common structures like lists and sets).

Exceptions as Objects. Representing exceptions as objects allows for a very flexible management of faults. Indeed, in this setting exceptions would have both a mutable state and a behavior. Also, one could define new kinds of exceptions using the interface hierarchy. Finally, exceptions would have identities allowing to distinguish different instances of the same fault. However, most of these features are not needed for faults: faults are generated and consumed, but they are static and with no behavior. Representing them as objects would allow a programming style not matching the intuition and difficult to understand. Furthermore, in ABS static verification is a main concern, and semantic clarity is more needed than in other languages. For this reason we think that in the setting of ABS exceptions should not be objects.

Exceptions as Datatypes. Datatypes fit more with the intuition of exceptions as described before: they are simple values with no identity nor behavior. However, in ABS datatypes are closed, meaning that once a datatype has been declared, it is not possible to extend it with new constructors. This is a potential problem in using them to represent exceptions. Indeed, we would like the datatype for

exceptions to include system-defined exceptions such as Division by Zero or Array out of Bound, and to be extended to accommodate user-defined exceptions. Also, for modularity reasons, programmers of an ABS module should be able to declare their own exceptions, thus exception declaration cannot be centralized. User-defined exceptions are not only handy for the programmer, but may also help the definition of invariants by tracking the occurrence of specific conditions. We discuss below a few possible design choices related to the definition of user-defined exceptions.

Allow open datatypes in ABS. In this setting exception would be an open datatype [24], and other ABS datatypes could be open as well. The declaration of system-defined exceptions can be done as:

```
open data Exception = NullPointerException
                    | RemoteAccessException
```

where the keyword **open** specifies that the datatype is open (in principle open and closed datatypes may coexist). Then one can add user-defined exceptions as:

```
open data Exception = ... | MyUserDefinedException
```

However, this is a major modification of datatypes, a key component of ABS, and introducing this additional complexity only to accommodate exceptions may not be a good choice. In fact, handling open datatypes is in contrast with the fact that ABS type system is nominal. One would need to resort to a structural type system (similar to, e.g., OCaml's variants [29]) to ensure that a pattern matching is complete, which is far less natural.

Allow any datatype to be an exception. In this setting any value of any datatype may be used as an exception (the fact of declaring which datatypes are actually used as exceptions does not change too much the setting). User-defined datatypes can be added by simply defining new datatypes. When the programmer wants to catch an exception, he has to specify which types of exceptions he can catch, and do a pattern matching both on the type and on its constructor to understand which particular fault happened. This produces a syntax like:

```
try { ... }
catch(List e) {
  case(e) {
    | Empty => ...
    | Cons(v,e2) => ...
} }
catch(NullPointerException e) { ... }
catch(_ e) { ... /* capture all exceptions */ }
```

where a special syntax _ is needed to catch exceptions of any type, since there is no hierarchy for datatypes in ABS. Note that in case the exception has type `List a` **case** is done to analyze its structure. A difficulty in applying this approach to ABS is due to the fact that in ABS values do not carry their type at runtime, but adding such an information seems not to have relevant drawbacks.

Exceptions as a new kind of value. In this setting exceptions are a separate kind of value, at the same level of objects and datatypes. The type Exception is open. New exceptions (both system- and user-defined) can be declared as follows:

```
exception NullPointerException
exception RemoteAccessException
...
exception MyUserDefinedException(Int, String)
```

Pattern matching can be used to distinguish different exceptions:

```
try { ... }
catch(e) {
  NullPointerException => ...
  MyUserDefinedException(n,s) => ...
  e2 => ...
  ...
}
```

Structural typing can be used if one wants to check that all exceptions possibly raised are caught, as, e.g., in Java.

Discussion. The simplest approach is to model exceptions as a closed datatype. However, if open exceptions are desired to increase the expressive power, the last solution is the one with minimal impact on the existing ABS language. Allowing any datatype as an exception also seems a viable option, but with a more substantial impact on the existing structure of ABS.

3 Which is the Behavior of Faults?

Faults interrupt the normal control flow of the program. A first issue concerning faults is how they are generated. Concerning fault management, it is a common agreement that faults are manipulated with a **try**/**catch** structure, and we do not see any reason to change this approach in our design for ABS. However, after this choice has been taken, the design space is still vast and many questions still need to be investigated.

Fault Generation. In programming languages, faults can be generated either by an explicit command such as **throw** `f` where `f` is the raised fault, or by a normal command. For instance, when evaluating the expression `x/y` a Division by Zero exception may be raised if `y` is 0. In this second case, which exception is raised is not explicit, but defined by the semantics of the command. After having been raised, the two kinds of exceptions are indistinguishable. A third kind of exception may be considered in ABS. Indeed, ABS is currently evolving towards having an explicit distribution, and in this setting localities or links may break. The only remote interaction in ABS is via asynchronous method invocation, and the corresponding **await**/**get** on the created future. In principle, network problems could be notified either during invocation, or during the **await**/**get**.

However, invocation is asynchronous, and will not check for instance whether the callee will receive and/or process the invocation message. For the same reason, it is not reasonable that it checks for network problems. Clearly, the **get** should raise the fault, since no return value is available.

The behavior of **await** depends on its intended semantics. If executing the statement **await** `f?` means that the process whose result will be stored in `f` has successfully finished, then the **await** needs to synchronize with the remote computation and should raise a pre-defined fault upon timeout and network errors. In this setting thus network faults are raised by both **await** and **get**. On the other hand, if executing **await** `f?` gives only the guarantee that a subsequent `f.`**get** will not block, then all faults, including network- and timeout-related faults, can be raised by **get** exclusively.

Fault Management. As discussed in the beginning of the section, we use the common **try**/**catch** structure to manage faults. This structure sometimes also features an additional block **finally**. The **finally** block specifies some code that must be executed both if no exception is raised and if it is. A common use of the **finally** block is to release resources which need to be freed whatever the result of the computation is.

```
try { ... }
catch(MyFirstException e) { ... }
catch(MySecondeException e) {... }
finally { P }
```

For instance, `P` may close a file used inside the **try** block. The **finally** block is very convenient for programming, but may not be needed in the core language. Indeed, in many cases it can be encoded. The encoding instantiated on the example above is as follows:

```
try {
  try { ... }
  catch(MyFirstException e) { ... }
  catch(MySecondeException e) { ... }
} catch(_ e) {
  P
  throw e;
}
P
```

Essentially, one has to catch all the exceptions, do `P` and rethrow the same exception. `P` also needs to be replicated at the end, so to be executed if no exception is raised. Note that this encoding relies on always having exactly one **return** statement per method, at its end (this is the recommended style of programming in ABS), and on the ability to catch all exceptions and to be able to rethrow them identically. Actually, in principle, one can also consider some uncatchable faults, but this seems not particularly relevant in practice.

For resource management, an alternative to the **finally** block is the autorelease mechanism of Java 7 [28], which automatically releases its resource at the end of

the block. Encoding such a mechanism in ABS could be done using an approach similar to the one above for the **finally** block.

Fault Effects. We have discussed how to catch faults. However, it may happen that a fault is not caught inside the method raising it. Then, as already said, the fault should interrupt the normal flow of computation, i.e. killing the running process. However, one may decide to kill a larger entity. Suitable candidates in ABS are the object where the fault has been raised, or its cog. Now, remember that in ABS there is a strong emphasis on correctness proofs based on invariants, and that whenever a process releases the lock of an object the class invariant must hold. An uncaught fault releases the lock by killing the running process. This means that whenever an uncaught fault may be raised, the invariant must hold. Since faults may be raised by many constructs, including expressions and **get**, ensuring this may be particularly difficult, and may require in practice to manage all the faults inside the method raising them. However, this is undesirable since a method may not have enough information to correctly manage a given fault. One can try to define a weaker invariant, but this may be difficult. A solution is to decide that a fault may not only kill the process, but also the object whose invariant may be no more valid. An even more drastic solution is to kill the whole cog. This may be meaningful if invariants involving different objects (of the same cog) are considered. However, this kind of invariant is currently not considered in ABS, thus the introduction of mechanisms for killing a whole cog seems premature.

Effect Declaration. In classic programming languages, the only effect of an uncaught fault is to kill the running process. However, we just discussed that also killing the whole object (or cog) is a possible effect. One may want to have different effects for different faults. More in general, different faults may have different properties. Another possible property may describe whether a fault can be caught or not. Whatever the set of possible properties is, an important issue is where those properties are associated with the raised fault. One can have a keyword **deadly** specifying that a given exception will kill the whole object if uncaught, while the behavior of just killing the process can be considered the default behavior. We can see three possibilities here. Properties may be specified:

when an exception is declared: for instance, one may write

```
deadly exception NullPointerException
```

A main drawback of this approach is that the same exception will behave the same everywhere. Intuitively, an exception may be deadly for an object where the invariant cannot be restored, and not for another one where the fault has no impact on the invariant. Note also that if any datatype can be an exception, then one has to specify properties for each datatype, e.g. **deadly** `Int`. Actually, this second drawback is mitigated by choosing suitable default values for properties.

when an exception is raised: for instance, one may write

```
throw deadly NullPointerException
```

or also shorten it into **die** `NullPointerException`. Clearly, this approach is only reasonable for exceptions raised by an explicit throw (unless one wants to write something like `x=y/0` **deadly**). The approach is also less compositional, thus less suitable for static analysis. In fact, to understand the behavior of an exception it is not enough to look at declarations. For instance, the same exception may be either deadly or not for the same method, depending on how it has been raised. Note also that this approach would break the encoding of **finally** above, since there is no way to rethrow an exception with the exact same properties.

in the signature of the method raising the exception: for instance, one may write

```
Int calc(Int x) deadly: NullPointerException {...}
```

Clearly, this approach is viable only for properties relevant when the exception exits the method, such as deadly. It would not work for instance to specify whether an exception can be caught or not. Notice that this approach integrates well with the declaration of which faults a method may raise, useful to statically verify that all exceptions are caught. In fact, one could write

```
Int calc(Int x) throws: DivisionByZero,
                  deadly NullPointerException {...}
```

More in general, this approach is suitable for static analysis, since a method declaration also provides the information on the behavior of exceptions raised by the method itself. The same information is useful also for the programmer, in particular when using methods he did not write himself.

Discussion. We think that in the context of ABS, a fault may have two different effects: either killing the process or the whole object, depending on whether the object invariant holds or not. Whether a fault should kill the whole object or not should be declared at the level of method signature to enhance compositionality. Note that in the most used object-oriented languages, objects are never killed as a result of an exception: indeed such a feature is relevant in ABS because of its emphasis on analysis based on invariants, and no widespread object-oriented language has been developed according to this philosophy. A possible alternative to kill the object would be to roll back state changes. A transparent rollback [10] in our setting could lead to the last release point, where one is sure the invariant holds. However such an approach, discussed in [12], is not always satisfying. Indeed, rolling back only locally may easily lead to inconsistencies between different local states (what corresponds to break invariants concerning multiple objects). On the other hand, global rollback as in [21] results in an overly complex semantics. Furthermore, if local rollback is needed in particular cases, it can usually be encoded. Similarly, the **finally** construct is not strictly needed, since with the choices we advocate it can be encoded.

4 How Do Faults Propagate?

We have discussed in the previous section the effect of a fault on the process or object where it is raised. However, in case of fault, in particular of uncaught fault, it is reasonable to propagate the exception also to other processes/objects related to it. In particular, possible targets for propagation are processes interested in the result of the computation, processes that have been invoked by the failed one, processes in the same object/cog of the failed one, processes trying to access an object after it died.

Propagation through the Return Future. In a language with synchronous method invocation the only process that can directly access the result of the computation is the caller. However, in languages with asynchronous method invocation any process receiving the future can directly access the result of the computation. The caller may be or may not be one of them, and indeed may even terminate before the result of the computation becomes ready. Thus we discuss here notification of faults to the processes synchronizing with the future. We have two possibilities: processes may synchronize with the future either with a **get** or with an **await** statement. The case of **get** is clear: those processes are interested in the result of the computation, in case of fault no correct result is available and those processes need to be notified so that they can decide how to proceed. The natural way of being notified is that the same exception is raised by **get**. A process doing an **await** is just interested in waiting for the computation to terminate, but not in knowing its result. Thus we claim that if the computation terminated, either with a normal value or with an exception, the **await** should not block and the exception, if any, should not be raised. The exception would be raised only if later on a **get** on the future is performed. This approach requires to put the fault notification inside the future, and has been explored in the context of ABS in [18]. Indeed, this is also the approach of Java future library (asynchronous computation with futures has been standardized in a Java library since Java SE 5 [11]). In contrast to ABS, Java's API does not distinguish between waiting for a future to become available, and retrieving the results. In fact, no primitive like **await** is available in Java. In addition, Java's futures do not faithfully propagate exceptions: the **get** method on a faulty future always raises the same exception `ExecutionException`.

An additional problem is related to concurrency. Indeed, in ABS, one may have multiple concurrent **get** and/or **await** statements on the same future containing an exception. Let us consider the case of multiple **get** statements. In this case, one has to decide whether they all raise the fault contained in the future or just one of them does. This second solution is more troublesome since to this end, the first process accessing the future would receive the exception and remove the fault from the future. The only possibility is to replace it with some default value, and this requires locking the future. However, this in turn changes the behavior of futures in a relevant way: Futures are understood as logical variables that change at most once, and this would no longer be true. Additionally, this creates a weak synchronization point between two processes

accessing the same future. Indeed, if a process knows that a future originally contains an exception, by accessing it he will know if another process accessed it before. These weak synchronization points between processes that would be independent otherwise make the concurrency model and thus the analysis more difficult. Note that concurrent **await** statements are not a problem, since they do not locally raise the fault, but just check whether the future is empty or not.

Propagation through Method Invocations. It may be the case that the failed computation has invoked methods in other objects, whose execution is no more necessary after the failure. Indeed, it may even be undesired. For instance, if you are planning a trip and the booking of the airplane fails, you do not want to complete the booking of the hotel. Thus a mechanism to cancel a computation originated by a past method call may be useful. Actually, cancel may have different meanings according to the state of the invocation. If the invocation has not started yet, one can simply remove the invocation message itself. If the invoked process is running, one may raise the exception. If the execution already completed, one may do nothing or execute some code to compensate the terminated execution. This second option has been explored in [18]. The most interesting case is the one where the invoked process is running. Indeed, in this case the fault may be raised in any point of the execution, thus dealing with it using a try-catch would require to have the whole method code, including the return command, inside the try block. A better approach is to define specific points in the code where the running process checks for exceptions from its invoker, and specifying there the code to be executed in this case. A more modular way is to separate the two issues. One may have a statement `check` to specify when to check for faults, and a statement **setHandler** `H` establishing that `H` is the handler to be used to deal with faults from the invoker from now on. `H` can be a simple piece of code, or a function associating pieces of code to exceptions. Pieces of code may have a return statement, to communicate the result of the fault management to the invoker. If the execution of the handler terminates successfully, the execution of the method code restarts. One may also decide that the last handler has to be used to compensate the execution if the cancellation occurs after the termination of the invoked process.

We have described the effect of propagation to invoked processes. However, one has to understand which invoked processes to consider. The simplest possibility is to let the programmer decide. We call this approach *programmed propagation*. This can be done through a statement `f1 = f.`**cancel**`(e)` where `f` is the future corresponding to the invocation to be canceled, `e` the exception to be raised and `f1` the future storing the result of exception management. Note that the future `f` is the right entity to individuate the invocation, since each invocation corresponds to a different future. Note also that with programmed cancel one may cancel twice the same invocation, and that cancel can be executed by any process on any future he knows of. Future `f1` may contain different values according to the outcome of the cancel. If the exception sent by the cancel is correctly managed, the handler returns a specific value to fill that future (potentially different from the value returned as a result of the method, which is in

future `f`). In all the other cases a system-defined exception is put inside future `f1` (one cannot put there a normal value, since this should depend on the type of the future):

- an exception `notStarted` if the **cancel** arrives before the invocation started (while the future `f` contains an exception `canceled`);
- an exception `terminated` if the **cancel** arrives after method termination, and compensations are not used (future `f` keeps its value);
- an exception `noCompensation` if the **cancel** arrives after method termination, compensations are allowed, but no compensation is specified for the target method (future `f` keeps its value);
- an exception `CancelNotManaged` if the exception arrives when the method is running, but it is not managed since there is no handler for it (while the future `f` stores the exception `e`).

In case of multiple cancellations, cancellations behave as above according to the state of the method when they are processed. Note also that the future `f` is not changed if it already contains the result of the method invocation.

An alternative approach is to have an *automatic propagation* of exceptions to invoked processes. First, one should decide whether to propagate only uncaught faults, or also managed faults. This last solution is not desirable in general, since most managed faults should not affect other processes, and can be dealt with by programmed propagation in case of need. Propagation of uncaught faults, if desired, should be necessarily done automatically. Now, the problem is to understand to which method invocations the fault needs to propagate. An upper bound is given by the futures known by the dying process. One may also consider that futures on which a **get** has already been performed are no more relevant. However, there is no fast and easy answer to this question. We think that a reasonable solution is to choose the futures which have been created by the current method execution and on which no **get** has been performed yet by the same method. One may also want to check whether the reference to the future has been passed to another method, and whether this method has performed a **get** on the future, but this would make the implementation and the analysis much more tricky. Similarly, one may want to consider also futures received as parameters, but again this needs to propagate the information on whether a future has been accessed or not from one method to the other. Note that in case of automatic propagation, no information on the result of the cancellation is needed, since the caller already terminated.

One may want to ensure that children can manage all the faults from their parent. To this end, each child should declare the exceptions that he can manage (at any point, since it may be the case that some exceptions can be managed only at some `check` due to handler modifications). Then, one can check that these include all the exceptions the parent may send to him. For automatic propagation, these coincide with the exceptions the parent may raise. For programmed propagation, these are the arguments of the various **cancel** of the corresponding future in the parent or in methods to which the future is passed.

Propagation to Other Processes in the same Object/Cog. We already discussed the fact that it is important for processes to restore the invariant of the object or cog before releasing the lock, and in particular before terminating because of an uncaught exception. In case the invariant cannot be restored, we proposed as a solution the possibility of killing the whole object/cog. An alternative solution is to terminate only the process P that first raised the fault, and notifying the other processes about the uncaught fault, since they may be able to manage it. Note that when process P is terminated, there is no other running process in the same cog. Thus the other processes will get the fault notification when they will be scheduled again. This means that they may get a fault, either when they start, or when they resume execution at an **await** or **suspend** statement. The fault may then be managed, or propagated as discussed above. In particular, if not managed, will be propagated to the next process to be scheduled. In this setting objects never die, but method calls may receive an exception as soon as they start. If we raise the same exception that was raised by P, then it may be difficult to track which exceptions may be raised inside any method. A simpler solution is to have a dedicated exception, e.g., InvariantNotRestored. What said above for objects holds similarly for cogs, concerning cog invariants. However, as already said, they are not a main concern in ABS at the moment.

Propagation through Dead Objects/Cogs. Some of the approaches we discussed involve the killing of an object or cog. We have not yet discussed what happens when a dead object is accessed (through method invocation). An exception should be raised. We can follow either the approach discussed above for network errors, or the one for normal faults. In practice the only difference is whether also the **await** will raise such a fault or not. We do not see any particular advantages or disadvantages for the two approaches: which is the best solution depends on which one better fits the programmer intuition, which may be different from one programmer to the other. In both the cases, using a standard exception such as DeadObject, instead of propagating the exception that caused the death of the object, simplifies the management. Also, it allows the caller to know whether the object is dead because of its invocation or it was already dead before.

Discussion. Among the propagation strategies above, propagation through the return future is nowadays standard, since it is used, e.g., in Java and C#. The possibility of canceling a running process via a future is also available in Java and C#, but the possibility of doing it automatically and/or of defining handlers and compensations for managing cancel requests while the process is running are not considered. Indeed, these strategies are quite complex, and it is not yet clear how useful they are in real programming. Also propagation of the fault to other processes in the same object is not considered in mainstream languages as far as we know, but we think this is a viable strategy in ABS. In fact, when a process is not able to restore the object invariant, there are two possibilities: either destroying the whole object or leave to another method call the task of restoring it. This second strategy seems also less extreme.

5 Conclusion

We have discussed the design space for fault models to be included in the ABS modeling language. As future work, we will extend the formal semantics of ABS with appropriate datatypes for the representation of faults, primitives to raise and catch faults, and mechanisms to distribute faults to other objects and cogs.

We complete the paper with a review of related fault models (the comparison with Java has been already discussed throughout the paper).

Functional programming languages, like OCaml or Haskell, also include primitives for faults modeled as exceptions. Both languages allow user-defined exceptions, but they implement them in different manners. OCaml uses a special type (called *exn*) to type exceptions, and new exceptions can be declared using the syntax **exception** `e` **of** `data`. In [24], the introduction of open datatypes in Haskell to encode exceptions is discussed. However, the current implementation of GHC uses *typeclass* [6], which allow one to register new datatypes as being exceptions.

Message-Passing Interface (MPI) is a cross-language standard, used, e.g., for C and Fortran, to program distributed applications. MPI expresses communication via so-called MPI functions, the basic ones being SEND and a blocking RECV. The SEND can work with three modalities: (*buffered send*) buffering the data to be sent, thus returning immediately as we assumed in this paper; (*synchronous send*) waiting for a corresponding RECV to be posted by the destination before terminating; or (*ready send*) failing in the case a corresponding RECV has not yet been posted by the destination. Dealing with the network, MPI functions represent communication at a lower level than we do: in MPI also the process of data delivering is taken into account. SEND (in all its modalities) and RECV have asynchronous variants, called ISEND and IRECV (where the "I" stands for "initiation"), which indicate a buffer where to fetch/put data and return immediately. For each such asynchronous send/receive, functionalities similar to some of the ones considered in this paper can be used: WAIT makes it possible to wait for the completion of data sending/receiving (in addition in MPI there is also a function TEST which returns immediately the status of the data sending/receiving without waiting); failures (e.g., in the communication while sending/receiving data) are detected by calling WAIT or TEST; and it is possible to cancel a send/receive by a call to CANCEL. The semantics of the latter, however, is the removal of the send/receive, supposing that it has not completed yet, as if it never occurred (a matching receive/send would perceive, as well, the canceled send/receive as if it never occurred). By combining the mechanisms above it is possible to obtain the waiting and canceling mechanisms considered in this paper. For example an asynchronous method invocation can be modeled by executing both ISEND and IRECV, and cancellation by executing CANCEL both on the send and the receive in the case the data is still under transmission or just on the receive in the case the send has completed. In the latter case, if the invoked method performs a ready send at the end of method execution, it will be notified of the matching IRECV having been canceled.

In web applications the HTTP protocol is used to realize service invocations by means of request/response pairs over a TCP/IP connection, as happens in

the popular approaches of Java and Javascript, i.e. Asynchronous Javascript and XML (AJAX) invocations. In Java a method is used to initiate the HTTP request/response, which differently from the approach considered in this paper, may yield an error in the case the connection with the HTTP server cannot be established (timeout based). Then the client goes through a two phase process, where he first sends data over an output-stream and then, similarly, receives data. At the server side a symmetric process is followed. Java methods for reading pieces of response data are blocking as for the waiting function used in MPI and considered in this paper. Similarly, failures are notified via exceptions when reading response data (or while sending request data). Finally concerning cancellation, the HTTP request/response can be aborted as a whole by the client and this causes the server to detect the failure (an exception is raised) when it is in the phase of inserting data in the response, i.e. when returning data (or while reading request data). In Javascript (AJAX) request data are preliminarily put into memory (as in MPI) and then the request/response is initiated (again this can fail if connection cannot be established). Such an initiation function also installs a user-defined function which is expected to manage the data received once the response is completed (including also managing the case of failure). This mechanism is an alternative to the waiting function used in MPI and considered in this paper. Concerning cancellation, it is possible, as in Java, to abort the HTTP request/response (with the same effect at server side).

References

1. G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.
2. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
3. L. Baduel et al. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer, 2006.
4. B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer, 2007.
5. D. Caromel. Service, Asynchrony, and Wait-By-Necessity. *Journal of Object Oriented Programming*, pages 12–22, 1989.
6. K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. of LFP'92*, pages 170–181. ACM, 1992.
7. F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
9. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.
10. E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

11. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
12. G. Göri, E. B. Johnsen, R. Schlatte, and V. Stolz. Erlang-style error recovery for concurrent objects with cooperative scheduling. In *ISoLA*, LNCS. Springer, 2014. To appear.
13. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
14. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.*, 7(4):501–538, 1985.
15. International Telecommunication Union. Open Distributed Processing — Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.
16. B. Jay. Algebraic data types. In *Pattern Calculus*, pages 149–160. Springer, 2009.
17. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. of FMCO 2010*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
18. E. B. Johnsen, I. Lanese, and G. Zavattaro. Fault in the future. In *COORDINATION*, volume 6721 of *LNCS*, pages 1–15. Springer, 2011.
19. *JSR166: Concurrency utilities*. `http://java.sun.com/j2se/1.5.0/docs/guide/concurrency`.
20. J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.
21. I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In *CONCUR*, volume 6901 of *LNCS*, pages 297–311. Springer, 2011.
22. R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., 1996.
23. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, pages 260–267. ACM Press, 1988.
24. A. Löh and R. Hinze. Open data types and open functions. In *Proc. of PPDP'06*, pages 133–144. ACM, 2006.
25. B. Morris. CActive and Friends. Symbian Developer Network, November 2007. `http://developer.symbian.com/main/downloads/papers/CActiveAndFriends/CActiveAndFriends.pdf`.
26. P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, Department of Computer Science, ETH Zurich, 2007.
27. E. Pitt and K. McNiff. *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.
28. J. Ponge. Better resource management with Java SE 7: Beyond syntactic sugar, May 2011. `http://www.oracle.com/technetwork/articles/java/trywithresources-401775.html`.
29. D. Rémy. Type checking records and variants in a natural extension of ml. In *Proc. of POPL'89*, pages 77–88. ACM, 1989.
30. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.
31. J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall. A note on distributed computing. In *MOS'96)*, volume 1222 of *LNCS*, pages 49–64. Springer, 1997.
32. A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.