# Erlang-style Error Recovery for Concurrent Objects with Cooperative Scheduling *

Georg Göri[1], Einar Broch Johnsen[2], Rudolf Schlatte[2], and Volker Stolz[2]

[1] University of Technology, Graz, Austria
goeri@student.tugraz.at

[2] University of Oslo, Norway
{einarj,rudi,stolz}@ifi.uio.no

**Abstract.** Re-establishing a safe program state after an error occurred is a known problem. Manually written error-recovery code is both more difficult to test and less often executed than the main code paths, hence errors are prevalent in these parts of a program. This paper proposes a failure model for concurrent objects with cooperative scheduling that automatically re-establishes object invariants after program failures, thereby eliminating the need to manually write this problematic code. The proposed model relies on a number of features of actor-based object-oriented languages, such as asynchronous method calls, co-operative scheduling with explicit synchronization points, and communication via future variables. We show that this approach can be used to implement Erlang-style process linking, and implement a supervision tree as a proof-of-concept.

## 1 Introduction

Crashes and errors in real-world systems are not always due to faulty programming. Especially but not only in distributed systems, error conditions can arise that are not a consequence of the logic of the running program. Robust systems must be able to deal with and mitigate such unexpected conditions. At the same time, error recovery code is notoriously hard to test.

An influential approach to more robust systems is "Crash-Only Software" [4], i.e., letting system components fail and restarting them. Erlang [2,19] is a widely-used functional language which successfully adopts these ideas. However, inherent in such subsystem restarts is the accompanying loss of state. This is much less a problem with programs written in a functional style than with programs written using object-oriented techniques, where the objects themselves hold state. This paper describes an approach to crash-only software which can keep objects alive without explicit code to restore object invariants.

The approach of this paper is based on concurrent objects which communicate by means of *asynchronous method calls*; the caller allocates a *future* as a container for the forthcoming result of the method call, and keeps executing until the result of the call is needed. Since execution can get stuck waiting for a reply,

* Partially funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu).

we allow process execution to suspend by introducing processor release points related to the polling of futures. Scheduling is *cooperative* via release-points in the code, awaiting either a condition on the state of the object, or the availability of the result from a method call. To concretize the approach, we use some features from the abstract behavior specification language ABS [11], a statically typed object-oriented modeling language targeting distributed systems. ABS has a formal semantics implemented in the rewriting logic tool Maude [7], which can be used to explore the runtime behavior of specifications.

This paper introduces linguistic means to both abort a single computation without corrupting object state and to terminate an object with all its pending processes. We provide a formal semantics for how those faults propagate through asynchronous communication. Callers may decide to not care about faults and fail themselves when trying to access the result of a call whose computation aborted, or use a safe means of access that allows them to explicitly distinguish a fault from a normal result and react accordingly. We show the usefulness of the new language primitives by showing how they allow us to implement process linking and supervision hierarchies, the standard recovery features of Erlang.

The rest of the paper is organized as follows. Section 2 describes the ABS language, Section 3 the novel failure model. Section 4 presents an operational semantics of a subset of the language, and illustrates the new functionality by modeling Erlang's well-known supervision architecture, and Section 5 discusses related and future work.

## 2 Behavioral Modeling in ABS

ABS is an abstract, executable, object-oriented modeling language with a formal semantics [11], targeting distributed systems. ABS is based on concurrent objects [5, 13] communicating by means of asynchronous method calls. Objects in ABS support interleaved concurrency based on explicit scheduling points. This allows active and reactive behavior to be easily combined, by means of a cooperative scheduling of processes which stem from method calls. 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].

ABS combines functional and imperative programming styles with a Java-like syntax. Objects execute in parallel and communicate through asynchronous method calls. However, the data manipulation inside methods is modeled using a simple functional language based on user-defined algebraic data types and functions. Thus, the modeler may abstract from the details of low-level imperative implementations of data structures while maintaining an overall object-oriented design close to the target system.

*The Functional Layer.* The functional layer of ABS consists of algebraic data types such as the empty type `Unit`, booleans `Bool`, and integers `Int`; parametric

$$
\begin{array}{rl}
T & ::= I \mid D \mid D\langle \overline{T} \rangle \\
A & ::= X \mid T \mid D\langle \overline{A} \rangle \\
Dd & ::= \textsf{data}\ D[\langle \overline{A} \rangle] = [\overline{Cons}]; \\
Cons & ::= Co[(\overline{A})] \\
F & ::= \textsf{def}\ A\ fn[\langle \overline{A} \rangle](\overline{A}\ \overline{x}) = e; \\
e & ::= x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \textsf{case}\ e\ \{\overline{br}\} \\
v & ::= Co[(\overline{v})] \mid \textsf{null} \\
br & ::= p \Rightarrow e; \\
p & ::= \_ \mid x \mid v \mid Co[(\overline{p})]
\end{array}
\qquad
\begin{array}{rl}
P & ::= \overline{IF}\ \overline{CL}\ \{[\overline{T}\ \overline{x};]\ s\} \\
IF & ::= \textsf{interface}\ I\ \{\ [\overline{Sg}]\ \} \\
CL & ::= \textsf{class}\ C\ [(\overline{T}\ \overline{x})]\ [\textsf{implements}\ \overline{I}]\ \{\ [\overline{T}\ \overline{x};]\ \overline{M}\} \\
Sg & ::= T\ m\ ([\overline{T}\ \overline{x}]) \\
M & ::= Sg\ \{[\overline{T}\ \overline{x};]\ s\} \\
g & ::= b \mid x? \mid g \wedge g \\
s & ::= s; s \mid \textsf{skip} \mid \textsf{if}\ b\ \{\ s\ \}\ [\textsf{else}\ \{\ s\ \}] \mid \textsf{while}\ b\ \{\ s\ \} \\
 & \quad \mid \textsf{suspend} \mid \textsf{await}\ g \mid x = rhs \mid \textsf{return}\ e \\
rhs & ::= e \mid cm \mid \textsf{new}\ C\ (\overline{e}) \\
cm & ::= [e]!m(\overline{e}) \mid x.\textsf{get}
\end{array}
$$

**Fig. 1.** ABS syntax for the functional (left) and imperative (right) layers. The terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [] optional elements.

data types such as sets `Set<X>` and maps `Map<X>` (for a type parameter `X`); and functions over values of these data types, with support for pattern matching.

The syntax of the functional layer is given in Figure 1 (left). The ground types $T$ are interfaces $I$, type names $D$, and instantiated parametric data types $D\langle \overline{T} \rangle$. Parametric data types $A$ allow type names to be parameterized by type variables $X$. User-defined data types definitions $Dd$ introduce a name $D$ for a new data type, parameters $\overline{A}$, and a list of constructors $Cons$. User-defined function definitions $F$ have a return type $A$, a name $fn$, possible type parameters, a list of typed input variables $x$, and an expression $e$. Expressions $e$ are variables $x$, values $v$, constructor, functional, and case expressions. Values $v$ are constructors applied to values, or `null`. Case expressions match an expression $e$ to a list of case branches $br$ on the form $p \Rightarrow e$ which associate a pattern $p$ with an expression $e$. Branches are evaluated in the listed order, the (possibly nested) pattern $p$ includes an underscore which works as a wild card during pattern matching; variables in `p` are bound during pattern matching and are in the scope of the branch expression $e$. ABS provides a library with standard data types such as booleans, integers, sets, and maps, and functions over these data types.

The functional layer of ABS can be illustrated by considering naive *polymorphic sets* defined using a type variable `X` and two constructors `EmptySet` and `Insert`:

```
data Set<X> = EmptySet | Insert(X, Set<X>);
```

Two functions `contains`, which checks whether an item `el` is an element in a set `set`, and `take`, which selects an element from a non-empty set `set`, can be defined by pattern matching over `set`:

```
def Bool contains<X>(Set<X> set, X el) =
  case set {
    EmptySet => False ;
    Insert(el, _) => True;
    Insert(_, xs) => contains(xs, el); };

def X take<X>(Set<X> set) = case set { Insert(e, _) => e; };
```

*The Imperative Layer.* The imperative layer of ABS addresses concurrency, communication, and synchronization at the level of objects, and defines interfaces, classes, and methods. In contrast to mainstream object-oriented languages, ABS does not have an explicit concept of threads. Instead a thread of execution is unified with an object as the unit of concurrency and distribution, which eliminates race conditions in the models. Objects are *active* in the sense that their `run` method, if defined, gets called upon creation.

The syntax of the imperative layer of ABS is given in Figure 1 (right). A program $P$ lists interface definitions $IF$ and class definitions $CL$, and has a main block $\{[\overline{T}\ \overline{x};]\ s\}$ where the variables $x$ of types $T$ are in the scope of the statement $s$. Interface and class definitions, as well as signatures $Sg$ and method definitions $M$ are as in Java. As usual, **this** is a read-only field of an object, referring to the identifier of the object; similarly, we let **destiny** be a read-only variable in the scope of a method activation, referring to the future for the return value from the method activation. Below we focus on explaining the asynchronous communication and suspension mechanisms of ABS.

Communication and synchronization are decoupled in ABS. Communication is based on asynchronous method calls, denoted by assignments `f=o!m(e)` where `f` is a future variable, `o` an object expression, and `e` are (data value or object) expressions. After calling `f=o!m(e)`, the caller may proceed with its execution *without blocking* on the method reply. Two operations on future variables control synchronization in ABS. First, the statement **await** `f?` *suspends the active process* unless a return value from the call associated with `f` has arrived, allowing other processes in the same object to execute. Second, the return value is retrieved by the expression `f.`**get**, which *blocks all execution in the object* until the return value is available. Inside an object, ABS also supports standard synchronous method calls `o.m(e)`.

Objects locally sequentialize execution, resembling a monitor with release points but without explicit signaling. An object can have at most one active process. This active process can be unconditionally suspended by the statement **suspend**, adding this process to the queue of the object, from which an enabled process is then selected for execution. The guards `g` in **await** `g` control suspension of the active process and consist of Boolean conditions $b$ conjoined with return tests `f?` on future variables `f` and with time-bounded suspensions `duration(e1,e2)` which become enabled between a best-case `e1` and a worst-case `e2` amount of time. Just like functional expressions, guards `g` are side-effect free. Instead of suspending, the active process may *block* while waiting for a reply as discussed above, or it may block for some amount of time between a best-case `e1` and a worst-case `e2`, using the syntax `duration(e1,e2)` [3]. The remaining statements of ABS are standard; e.g., sequential composition $s_1; s_2$, assignment `x=rhs`, and **skip**, **if**, **while**, and **return** constructs. Right hand side expressions `rhs` include the creation of an object **new** `C(e)`, method calls, and future dereferencing `f.`**get**, in addition to the functional expressions `e`.

*Example.* To illustrate the imperative layers of ABS, let us consider an interface `Account`, with methods `deposit` and `withdraw`, which is implemented by a class

```
interface Account {
    Unit deposit (Int amount);
    Unit withdraw (Int amount);
}

class Account implements Account {
    List<Int> transactions = Nil; // log of transactions
    Int balance = 0; // current balance
    Unit deposit (Int amount) {
        transactions = Cons(amount, transactions);
        balance = balance + amount;
    }
    Unit withdraw (Int amount) {
        transactions = Cons(-amount, transactions);
        if (balance < amount) abort "Insufficient funds";
        balance = balance - amount;
    }
}
```

**Fig. 2.** Bank account with history in ABS

`BankAccount` (as shown in Figure 2). We see that expressions from the functional layer are used inside the method implementations; e.g., the constructor `Cons` is used in the right hand side of an assignment to extend the list of transactions, and infix functions `+` and `-` are similarly used to adjust the balance.

To approach the theme of the next section, the example does not resolve the case of negative balance on the account (ignoring the issue of a better design which checks the condition before updating the history). A call to `withdraw` will only succeed if the balance is sufficient; if the `balance` is less than `amount` it is unclear what would be meaningful behavior in order to restore a class invariant like `balance`$\geq$`0`, and the method activation will *abort*: the previous state of the object will be restored, and the future storing the implicit return value of `Unit` type will be filled with a value indicating that an error occurred.

## 3 Failure Models and Error handling

Apart from user-specified aborts, it is very common for programs to run into so-called *runtime errors*, i.e., abnormal termination in a case where the operational semantics does not prescribe how the system can proceed. Prominent representatives of this class of faults are division by zero, null pointer accesses in languages that allow pointer dereferences, and errors that are propagated from the runtime system in managed languages, like out of memory errors when no more objects can be allocated.

In the semantics of the ABS language, behavior in those situations is underspecified, even though those situations can be encountered by the backends when running the code generated from an ABS model. For example, in the Maude semantics, a *division by zero* does not allow further reduction of that process, which may go unnoticed in the overall system, or lead to a deadlock when other processes wait on the object. In the Java backend, the underlying Java runtime

will generate a Java exception through the primitive math operations, which will terminate the current (ABS) process, and lead to similar effects as in Maude.

### 3.1 Design considerations

*Invariants and the system.* On abrupt termination of a computation, we need to establish which reaction would be required. In a distributed, loosely coupled system, a local error should not affect the complete system. So clearly here the guiding point must be that we have to keep the effects *local*. In our actor-based setting, we can take the locality even further: Although a computation failed, we can limit the effects to the *current process*. The object may still be able to process pending and future requests (although the caller of the failing process needs to be notified). But what should be the basis for further executions within this object?

The underlying motivation for the explicit release points in the language are of course the class invariants that developers rely on when designing their programs. As such, each method call expects that its respective object invariant holds upon entry (and upon awakening). This is clearly not the case under abrupt termination, before which the fields of the object may have been arbitrarily manipulated—the next release point may not have been reached.

*Error handling in an object system.* The mechanism we propose, defines the behavior in case of an error:

- Propagate errors through futures. The caller receives an error when reading the future.
- Default to having no explicit error handling, in which case a *process* is terminated, yet the *object* stays alive.
- Revert any partial state modifications to the current object up to the last release point.

These concepts are introduced by extending futures to propagate a possible error in the callee to the caller, providing a method to detect and handle an error contained in a future, and to terminate the caller in the case an error in a future is accessed by the default mechanism.

*Linguistic support for error handling.* We consider the following linguistic support to enable the envisaged error handling:

- a notion of user-defined error types
- a generalization of futures to either return values or propagate errors
- a statement **abort** `e`, which raises an error `e` and terminates the process
- a statement `f.`**safeget**, which can receive errors and values from a future `f`
- a statement **die**, which terminates the current object and all its processes

*The occurrence of an error* is represented in the model by means of the statement **abort** `e`, where `e` is an user defined error. These errors are represented by a special data type (see [15] for an extensive discussion of the potential design decisions). Such an abort can either be explicit in the model or can occur implicit either in internals of the execution, to represent distribution, system (e.g. out of memory) or runtime (e.g. division through zero) errors.

The semantic interpretation is dependent on the kind of ABS process the evaluation occurs in:

**Active Object** processes, represent the object's implicit execution of its `run` method. If in that process an **abort** `e` statement is evaluated, all current asynchronous calls to this object will abort with the error `e` and the references to this object will become invalid. Further synchronous or asynchronous calls to this object are equivalent to an **abort** `DeadObject` on the caller side. This mechanism was chosen, as the object behavior (its `run` method) is seen as an integral part of its correctness, and like an invalid state also an invalid termination of this behavior leads to an inconsistent object and therefore the object cannot be further used.

**Asynchronous Call** processes evaluated a method call in the called object. An **abort** `e` statement will terminate the process and return the error $e$ to the associated future. Moreover, the callee will perform a rollback (see below).

**Main Process.** The main process (similar to Java's `main`-method entry point) represents the begin of the execution, and an **abort** there will, by convention, lead to the runtime system being terminated (in principle, this could be handled uniformly like the normal case, but in practice we prefer termination).

*An automatic rollback* discards all changes to the object's values since the last scheduling point, which can be either an **await** or **suspend**. This guarantees that objects only evolve from one state at a scheduling point to another, and not leave in case of an error an object in a state, which could violate the object invariant.

*Extending futures* to contain either the computed value or a potential error raised either by an **abort** on the callee side (or from the runtime in a distributed setting), enables error propagation over invocations. Following this, also the semantics of the `Future.`**get** statement needs to be adjusted: a **get** will, in presence of an error `e` in the future, lead to an implicit **abort** `e` on the caller side.

The newly introduced `Future.`**safeget** stops this propagation and allows one to react on errors. **safeget** returns a value of the algebraic data type `Result<T>`, which is defined as `Result<T> = Value(T val)|Error(String s)`. In case the future contains an error `e`, the same is returned, otherwise the constructor `Value(T v)` wraps the result value `v`. Note that due to the lack of subtyping in the type system, currently the only way to communicate an error indication is through a value of type `String`, as we cannot define a common type for all possible (incl. user-defined) errors.

*The* **die** `e` *statement* allows in asynchronous calls to terminate the active object. Its semantic meaning is the same as an **abort** `e` in the execution context of an active object process or init block. In other words, all pending asynchronous calls and the active object's process are terminated. This statement allows to implement linking (see below), and can be used in distributed models to simulate a disconnect from an object.

*Discussion.* We come back to the banking example in Listing 2 to illustrate the point of rollbacks. The general contract is that the list of `transactions` should accurately reflect the current total in the account. As the body of `withdraw` needs to modify two fields, we clearly benefit from ABS's semantics of explicit release points which guarantees that only one process is executing within the object (e.g. in Java, we would be required to explicitly declare the method as `synchronized` to achieve the same effect).

Nonetheless, even though if only by construction of the example, an **abort** would leave the object in an undesired state, as after the modification of the list of transactions the balance is no longer in sync with the banking transaction history. If an **abort** would simply terminate execution of the current process, and start processing another pending call on the current state of the object, we would observe invalid results. But with the rollback before processing another call, this assumption can easily be re-established.

Note that the ABS methodology is only concerned with *object* invariants, and this mechanism does not give us *totality* in the sense that a method either completes successfully or not at all: a rollback will not undo changes in other objects that have (transitively) occurred as the result of method calls during execution of the current process, unlike e.g. in work on a higher-order $\pi$-calculus [16]. This means on the one hand that the developer still has to actively take into account the workings of error recovery when designing the system, but on the other hand allows us to implement this feature efficiently by only keeping track of fields in the current object that are actually touched.

Compared to traditional object-oriented programming, we note that this implicit error handling strategy frees the developers from restoring state explicitly in an exception handler. However, through the **safeget** mechanism, they still have this option open.

### 3.2 A practical application of error propagation: process linking

The previously presented primitives enable an implementation of Erlang-style linking between two objects in ABS. These links are part of the foundation for Erlang's well known and successful error handling [1]. Erlang's communication model is even more loosely coupled than ABS, in that it is based on asynchronous message passing. As such, there are no method calls or explicit returns, but rather the callee has to send back a response, which will be queued in the recipient until extracted from the mailbox. Thus, a failure in the recipient process will either go unnoticed if no response messages are used or otherwise lead to an expected

message not being sent/received, and in turn a corresponding potential blockage can occur in the initial sender.

*Erlang's links* enable mutual observation of processes. A process can link itself to another process. If one of the two processes terminates, the runtime environment sends an *EXIT* message to the other process, which contains an exit reason. Unless this exit reason is *normal* (termination because the process reached the end of the function), the linked process will terminate as well, and in consequence propagate its own *EXIT* message to its linked processes. With this *error propagation*, it is possible to let groups of processes up to the whole system terminate automatically and clean up components consisting of multiple processes.

To enable processes to observe exit messages or react on them, a process can be marked to be a system process with the *trap_exit* process flag. Such processes will not terminate when receiving an *EXIT* message, but can retrieve this message from their inbox.

*Implementation in the concurrent object model.* The implementation idea is to represent a link by two asynchronous calls, one to each of the objects. Each call will only terminate upon termination of the object, and thus enables the caller to take an action.

In Figure 3 a sample implementation is shown, which assumes that each class implements code similar to the `Linkable` class. A link can be established by creating a new object of class `Link`, where the link gets initialized with references to both objects (referred to as `s` and `f`), and then calling `setup` on this new link. The `setup` method will initiate the calls between the objects, by calling `waitOn` and then wait until both calls are processed, where finished calls can be seen by the counter `done`.

The `waitOn` method implemented in the `Linkable` class places the normally non-terminating asynchronous call in line 3 to the other `Linkable` it should link to. The non-termination is achieved by a simple **await** `false`, as can be seen in the `wait` method. After those calls are made, the `waitOn` method reports back to the `Link` that it succeeded, and will afterwards await the termination of the call in line 3. The only possibility for a call to `wait` to return is when the object dies. Should now this future ever contain a value it must be an error, where in line 7 we can now take an action in case that the other object terminated, which will be in the default case a subsequent termination of the local object, by executing **die** `e`.

*Linking in a producer consumer environment* can be used to bind both objects together, so that a termination of the producer or consumer leads to the termination of the other party as well. In Figure 4 we see a `Producer` and `Consumer`, modeled as ABS classes, where the `Producer` sends a new input to the `Consumer` via an asynchronous call. Both classes have to implement the `Linkable` interface and include the shown default implementation of `wait` and `waitOn`.

Setting up a Link between `Producer` and `Consumer` is performed by the first two lines in the `Producer`'s `run` method. We construct the `Link` object and ini-

```
class Link(Linkable f,Linkable s)
  implements Link{
  Int done=0;
  Unit setup(){
   f!waitOn(this,s);
   s!waitOn(this,f);
   await done==2;
  }

  Unit done(){
   done=done+1;
  }
}
```

```
class Linkable() implements Linkable{
  Unit waitOn(Link l,Linkable la){
    Fut<Unit> fut=la!wait();
    l!done();
    await fut?;
    case fut.safeget {
     Error(e) => die e;
    }
  }
  Unit wait(){
    await false;
  }
}
```

**Fig. 3.** Implementation of links in ABS

```
class Producer(Consumer c)
  implements Linkable{
 Unit run(){
  Link lConsumer= new Link(this,c);
  await lConsumer!setup();
  // produce
  c!consume(X);
 }
 // include wait and waitOn
}
```

```
class Consumer()
  implements Linkable{

 Unit consume(String x){
    // consume
 }

 // include wait and waitOn,
}
```

**Fig. 4.** Links between a Producer and a Consumer

tialize the link via the `setup` method. A more detailed view of asynchronous calls and their lifetime is presented in the sequence diagram in Figure 5, arrows represent an invocation and a possible return value, and boxes represent the duration of the call on the callee side. First, the link is setup, two inputs are produced, and after that the `Consumer` aborts, which also terminates the `Producer`.

Before the `consume` calls, all necessary invocations to establish the `wait` calls, which can be seen as a monitor if the object is still alive, are shown. After that we see that two inputs from the `Producer` are sent to the `Consumer`, where the `wait` calls are still pending. In the end, the `Consumer`, and in consequence also the `wait` call, terminate. The termination leads to the retrieval of the exit reason (in form of an error) by the `Producer` from the associated future, which results in its termination as well.

One of the current limitations of this design is that due to the lack of subclassing, the boiler-plate implementation of the methods `wait` and `waitOn` in any class needs to be replicated (such as `Producer` and `Consumer` above). ABS offers so-called *deltas* to support assembly of *software product lines*. Although this feature can be used here in principle to inject code into a class, according to the current syntax of deltas, the method bodies would still have to be replicated in *each* delta. A potential improvement would be an extension of ABS which would allow injecting code into all classes implementing a particular interface.

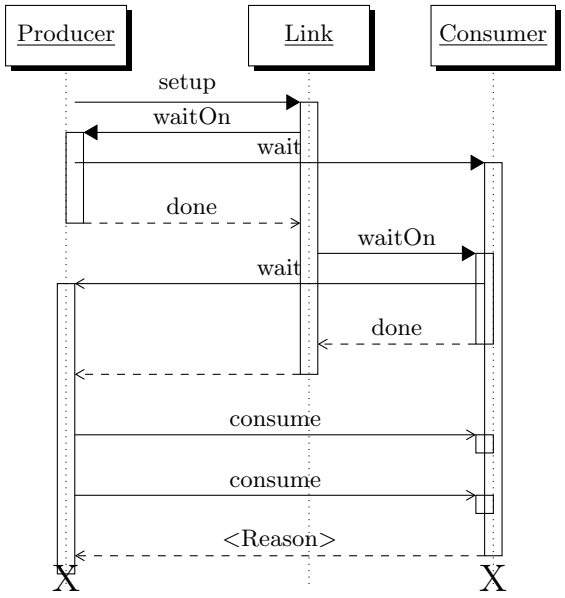


**Fig. 5.** Asynchronous calls in the Producer-Consumer example

Such functionality is well-known in *aspect-oriented programming*, and the ABS compiler should be easy to extend with a similar feature.

## 4 Operational Semantics and Application

A complete operational semantics of the core ABS language can be found in [11]. This section presents an operational semantics of the new language elements discussed in Section 3, omitting or simplifying parts that are not necessary for understanding the new error model. Figure 6 presents the runtime syntax of the language, while Figure 7 contains the operational semantics rules for the new rollback behavior, **abort** and **die** statements, and error propagation via futures.

The runtime state is a collection $cn$ of objects, futures and method invocations. Objects are denoted $o(a, a', p, q)$, where $a$ is the object state, $a'$ the safe state at the previous suspension point. Dead objects are represented by their identifier $o$ only. Object and process states $a$ are mappings from identifiers to values, $p$ is the currently running process or the symbol **idle** (denoting an object not currently running any process), and $q$ is the process queue. A process $p$ is written as $\{a|s\}$ with $a$ a mapping from local variable identifiers to values and $s$ a statement list.

In Figure 7 we elide the step of reducing expressions to values – evaluation is standard and can be seen in [11]. The SUSPEND rule saves the current state $a$, while the ABORT rule reinstates a saved state while also removing the current process and filling the future $f$ with an error term. The DIE rule deactivates

$$
\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \epsilon \mid fut \mid object \mid invoc \mid cn\ cn & & & \\
fut & ::= & f \mid f(val) & a & ::= & T\ x\ v \mid a, a \\
object & ::= & o(a, a', p, q) \mid o & p & ::= & process \mid \textsf{idle} \\
process & ::= & \{a \mid s\} & val & ::= & v \mid error \\
q & ::= & \epsilon \mid process \mid q\ q & v & ::= & o \mid f \mid data \\
invoc & ::= & m(o, f, \overline{v}) & error & ::= & e(val)
\end{array}
$$

**Fig. 6.** Runtime syntax. Overall program state is a set $cn$ of futures, objects and invocation messages. Literals $v$ are object identifiers $o$, future identifiers $f$, and number and string literals $data$.

the object and fills all futures of the object's processes with an error term. The Dead-Call rule provides a default error term as the result of a method call to a dead object. The other rules show the behavior of normal execution for these cases.

### 4.1 Discussion

From an implementation perspective, we note that the rollback mechanism appears reasonably cheap, as only that part of the state of the current object needs to be duplicated which is actually modified. This is easy to implement since ABS does not have destructive modification of data structures.

How to make best use of the rollback-mechanism is still up to the developer. We note that compared to traditional exception handling, a single method essentially corresponds to a **try**-block, whereas the caller specifies through a **safeget** and a subsequent case-distinction the possible **catch**-blocks, or decides to propagate any exceptions through **get**.

### 4.2 Application: Supervision

In Erlang the idea to let processes observe each other was taken further by constructing trees, where so called *supervisors start, observe and restart* their child processes. Supervision is one of the very important concepts, which is part of Erlang's highly regarded error handling capabilities [19]. Plugging in a supervisor as child of another supervisor generates a tree structure, which describes a structural view on components of a system. This tree structure enables both restarting of faulty leaves and of larger subtrees in case of repeated errors in a subsystem. So a faulty system with a supervisor tries to restart larger and larger parts of the whole system until enough faulty state is discarded and it is able to continue its operation.

*Supervision for concurrent objects.* Through linking, we can now apply the concept of supervision to concurrent objects. This enables modeling of a statically typed supervision tree that maintains active objects.

(Suspend)
$$o(a, a', \{l \mid \textsf{suspend}; s\}, q) \to o(a, a, \textsf{idle}, \{l \mid s\} \circ q)$$

(Activate)
$$\frac{p = \mathit{select}(q, a, cn)}{o(a, a', \textsf{idle}, q)\ cn \to o(a, a', p, (q \setminus p))\ cn}$$

(Await-Incomplete)
$$o(a, a', \{l \mid \textsf{await}\ f?; s\}, q)\ f \to o(a, a', \{l \mid \textsf{suspend}; \textsf{await}\ f?; s\}, q)\ f$$

(Await-Complete)
$$o(a, a', \{l \mid \textsf{await}\ f?; s\}, q)\ f(val) \to o(a, a', \{l \mid s\}, q)\ f(val)$$

(Return)
$$\frac{f = l(\textsf{destiny})}{o(a, a', \{l \mid \textsf{return}(v); s\}, q)\ f \to o(a, a, \textsf{idle}, q)\ f(v)}$$

(Abort)
$$\frac{f = l(\textsf{destiny})}{o(a, a', \{l \mid \textsf{abort}(v); s\}, q)\ f \to o(a', a', \textsf{idle}, q)\ f(e(v))}$$

(Async-Call)
$$\frac{\mathit{fresh}(f)}{o(a, a', \{l \mid x = o'!m(\overline{v}); s\}, q) \to o(a, a', \{l \mid x = f; s\}, q)\ m(o', f, \overline{v})\ f}$$

(Bind-Mtd)
$$\frac{p' = \mathit{bind}(m, o, \overline{v}, f)}{o(a, a', p, q)\ m(o, f, \overline{v}) \to o(a, a', p, p' \circ q)}$$

(Die)
$$\frac{f = l(\textsf{destiny})\ cn' = \mathit{abort\text{-}futures}(cn, q, v)}{o(a, a', \{l \mid \textsf{die}(v); s\}, q)\ f\ cn \to o\ f(e(v))\ cn'}$$

(Dead-Call)
$$o\ f\ m(o, f, \overline{v}) \to o\ f(e(\texttt{"dead object"}))$$

(Read-Fut)
$$o(a, a', \{l \mid x = f.\textsf{get}; s\}, q)\ f(v) \to o(a, a', \{l \mid x = v; s\}, q)\ f(v)$$

(Read-Fut-Error)
$$o(a, a', \{l \mid x = f.\textsf{get}; s\}, q)\ f(e(v)) \to o(a, a', \{l \mid \textsf{abort}(v); s\}, q)\ f(e(v))$$

(Safe-Read)
$$o(a, a', \{l \mid x = f.\textsf{safeget}; s\}, q)\ f(val) \to o(a, a', \{l \mid x = val; s\}, q)\ f(val)$$

**Fig. 7.** Operational semantics. The following helper functions are assumed: *bind* creates a new process given a method name $m$, object $o$, arguments $\overline{v}$ and future $f$; *abort-futures* transforms a configuration, filling all futures $f$ referenced from processes in queue $q$ with an error term $e(v)$ while returning all other parts of the configuration unchanged; *select* chooses a process from a queue $q$ that is ready to run.

```
1 Unit start(SupervisibleStarter child){
2  SupervisorLink sl=
3       new SupervisorLink(this,child);
4  Link l=new Link(sl,this);
5  await l!setup();
6  links=Cons(sl,links);
7  sl.start();
8 }
```

(a) Start a child

```
1 Unit died(SupervisibleStarter ss,
2         String error){
3  case strategy {
4   RestartAll => this.restart();
5   RestartOne => this.start(ss);
6   Prop => die error;
7  }
8 }
```

(b) Handle a deceased child

**Fig. 8.** Key methods of the `Supervisor`

To achieve a very generalized supervisor implementation we want to separate it from the concrete way of starting and linking children and want to be able to define different restart strategies. These strategies define the actions taken if a child terminates. Therefore we implemented a class `Supervisor` with following parameters: a list of `SupervisorStarter` objects, each of which specifies one child and implements the start and linking of this child; a strategy, which can be one of the following:

**Restart one:** Only the terminated child is restarted.
**Restart all:** If a child dies, it and all its siblings will be restarted.
**Propagate:** The supervisor and all children will terminate and the error will be thereby propagated to the next supervisor, ending at the root node of the runtime system.

This can be easily extended with other interesting strategies like rate limiting, e.g. propagating an error if a certain frequency of crashes is exceeded.

*The implementation of the supervisor* requires special considerations, as a supervisor has to start a list of children, keep track of them, has to detect a link failure and be able to forcefully terminate a child (for the *restart all* strategy). As the standard implementation of the link mechanism, shown in Figure 3, has on the error receiving side no indication about the source of the link error, every link to a child is represented by an object of class `SupervisorLink`. This object keeps the reference of the child specification (the `SupervisibleStarter` object) and passes it along to the `Supervisor`'s `died` method, which is depicted in Figure 8b. Furthermore this design allows one to forcefully kill one child, by terminating the associated `SupervisorLink`, which will—via linking—terminate the child.

For starting a child, a new `SupervisorLink` has to be created and linked to, so that in case the supervisor itself terminates (e.g. when the strategy is to propagate) all `SupervisorLink`s and children are terminated as well. This method is shown in Figure 8a.

## 5 Conclusion and Related Work

We have presented an extension to a concurrent object language, which incorporates automatic rollback to a "safe" (as conceptually defined by the developer

through a class invariant) state for the object that encountered an *abort*. Aborts either occur in the form of runtime errors, through an explicit call similarly to **throw**ing an exception, or from accessing a future which holds the result of an aborted computation.

The propagation- and detection mechanism for such faults allows us to model Erlang-like process linking, and the *safe* way of accessing futures corresponds roughly to exception handling with a distinction on the return result (normal return value vs. fault plus description).

We have implemented the proposed extension in a straight-forward manner in the prototypical (non-distributed) Erlang backend for ABS, and in the Maude simulator: The sources are publicly available in the ENVISAGE git repository at http://envisage-project.eu.

*Related Work.* Asynchronous computation with *futures* has been standardized in the Java API since Java SE 5 [10]. Due to the limitations of the so-called *generics* in the type system, no subtyping on futures is possible: this leads to the situation that (synchronous) method calls may make use of covariant return types, but for a type `B extending A`, a **Fut**`<B>` cannot be assigned to a **Fut**`<A>`. Our futures, based on ABS, do not have this limitation as futures stem from the functional data types and thus subtyping over parameterized types is safe due to the lack of destructive updates/writes. As first-class citizens, the ABS futures do not offer any cancellation and a process cannot affect another process except through sending messages (the Java API offers advisory cancellation, and—discouraged—forceful termination of threads).

Compared to Java futures, the ABS futures are intended to scale massively: while due to the limitations in Java's thread model only a restricted (by memory/stack requirements) number of threads can be effectively active (the standard reference [10] gives a limit in the "few thousands or tens of thousands", usually scheduled by an execution service); their intended use in ABS clearly follows Erlang's notion of virtually unbounded, light-weight, disposable threads.

A related failure model for an ABS-like language has also been discussed in [12]. To enable coordinated rollbacks, *compensations* are attached to method *returns*, in case a later condition indicates that a rollback across method calls should be necessary. The authors illustrate however that the distributed nature of compensation still does not make it easier to maintain *distributed invariants* involving several objects. Rollbacks in a concurrent system and their intricacies have also been discussed in the context of a higher-order $\pi$-calculus by Lanese et al. [16]. The entire design space of fault handling in a loosely coupled system is discussed in [15], but focuses on a more traditional approach of exception handlers to give developers an explicit means of recovery, instead of the implicit rollbacks presented here.

Unlike Java Card's transactions [6] our extension does not allow selective *non-atomic* updates, where a persistent value is modified within a transaction and *not* rolled back with the transaction. Our implementations do not store the entire heap upon method activation, but only the state of the current object.

A corresponding proof-theory as developed by Mostowski [17] for JAVA CARD-support in the KeY system should likewise be feasible for our approach.

*Future work:* The current rollback mechanism should also be easy to extend to *transactions* through a combination of versioning the object state and speculative execution. Also, rollbacks for a group of objects should at least semantically be easy to model, yet maintaining object graphs as additional state may make this approach too costly: every method invocation on another object would make this object a member of the transaction, and all objects would have to reach release points simultaneously to commit. Additionally, a distributed implementation of checking for such a commit would most likely be prohibitive. Instead of arbitrary object groups derived again following the discussion in [15], one may instead take advantage of so-called *concurrent object groups* (which are already present in ABS, but not discussed in this paper). They are used in ABS to model groups of objects running e.g. on the same node or hardware. Because of the intentionally tight coupling, one consideration is that a **die**-statement may even have the consequence of terminating the processes of an entire group, instead of the limited effect on the local object only that we discussed here.

Although the asynchronous communication mechanism together with the introduced failure mechanisms allows us to describe the communication behavior in a distributed system, the current semantics treats all calls—whether remote or local—the same. While this location transparency is also a feature of the Erlang language, it would be useful to reflect the topology of the system and resource aspects (such as processing power and communication latency) of the different nodes in a model. To this end, in [14] *deployment components* were introduced, which give the modeler the possibility to specify where objects are created and consequently where their processes run. Note that in contrast to Erlang, *objects* are allocated at creation-time, whereas Erlang allocates processes. On top of deployment components, resource costs and capabilities can be modeled and execution times can be estimated under different resource and deployment models. Simulation can then be used to examine the behavior of the (distributed) system wrt. artificially injected faults and deadline misses.

With respect to the supervision trees, we note that in the Erlang community, since the tree structure is specified through code, there was an interest in reverse-engineering the actual hierarchy for purposes of static analysis from the source code [18]. We hope that for top-down development, specification of the hierarchy can be made independent of the code, and is conversely more amenable to verification.

## References


1. J. Armstrong. Erlang—a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, 1996.
2. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

3. J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.
4. G. Candea and A. Fox. Crash-only software. In M. B. Jones, editor, *HotOS*, pages 67–72. USENIX, 2003.
5. D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2005.
6. Z. Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, 2000.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. of ESOP'07*, 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. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
11. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
12. E. B. Johnsen, I. Lanese, and G. Zavattaro. Fault in the future. In W. D. Meuter and G.-C. Roman, editors, *COORDINATION*, volume 6721 of *LNCS*, pages 1–15. Springer, 2011.
13. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
14. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Modeling application-level management of virtualized resources in ABS. In *Proc. 10th Intl. Symp. on Formal Methods for Components and Objects (FMCO 2011)*, volume 7542 of *LNCS*, pages 89–108. Springer, 2013.
15. I. Lanese, M. Lienhardt, M. Bravetti, E. B. Johnsen, R. Schlatte, V. Stolz, and G. Zavattaro. Fault model design space for cooperative concurrency. In *Submitted to ISoLA'14*, 2014.
16. I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order Pi. In J.-P. Katoen and B. König, editors, *CONCUR*, volume 6901 of *LNCS*, pages 297–311. Springer, 2011.
17. W. Mostowski. Formal reasoning about non-atomic Java Card methods in dynamic logic. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 444–459. Springer, 2006.
18. J. Nyström and B. Jonsson. Extracting the process structure of Erlang applications. In *Erlang Workshop, Florence, Italy*, Sept. 2002. http://www.erlang.org/workshop/nystrom.ps.
19. S. Vinoski. Reliability with Erlang. *IEEE Internet Computing*, 11(6):79–81, 2007.