Fault in the Future *

Einar Broch Johnsen[1], Ivan Lanese[2], and Gianluigi Zavattaro[2]

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

Abstract. In this paper we consider the problem of fault handling inside an object-oriented language with asynchronous method calls whose results are returned inside futures. We present an extension for those languages where futures are used to return fault notifications and to coordinate error recovery between the caller and callee. This can be exploited to ensure that invariants involving many objects are restored after faults.

1 Introduction

Concurrent and distributed systems demand flexible communication forms between distributed processes. While object-orientation is a natural paradigm for distributed systems [14], 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, 18, 27]. Several slightly differently flavored concurrent object systems exist for, e.g., Java [5, 25], Eiffel [8, 22], and C++ [21]. Concurrent objects are reminiscent of Actors [1] and Erlang processes [4]: 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 to the tightly synchronized (remote) method invocation model. Intuitively, asynchronous method calls spawn activities in objects without blocking execution in the caller. Return values from asynchronous calls are managed by so-called *futures* [12, 19, 27]. Asynchronous method calls and futures have been integrated with, e.g., Java [17] and Scala [11] and offer a large degree of potential concurrency for deployment on multi-core or distributed architectures.

In the event-driven communication model of Actors and Erlang processes, fault recovery is typically managed by linking processes together [4] or by monitors [2, 26]. These approaches do not address asynchronous method calls and

* Partly funded by the EU project FP7-231620 HATS and the ANR-2010-SEGI-013 project AEOLUS.

futures. In this paper, we extend the Java approach [17] with mechanisms for error recovery developed in the context of web services. Futures are used to identify calls, so they provide a natural means to distribute fault notifications and kill requests. We introduce also primitives for defining and invoking compensations allowing one to undo already completed method executions. In this way, we obtain a symmetric framework where caller and callee can notify their failure to the partner and manage the incoming notifications. This supports distributed error recovery policies programming.

The work reported in this paper is based on ABS, a formal modeling language for distributed concurrent objects which communicate by asynchronous method calls and futures, to take advantage of its formal semantics and simplicity. ABS is a variant of Creol [9, 16], and it is the reference language of the European Project HATS [13]. Creol has been shown to support compositional verification of concurrent software [3,9], in contrast to multi-threading. A particular feature of ABS is its cooperative scheduling of method activations inside concurrent objects, which allows different activities to be pursued within the object in a controlled way; in particular, active and reactive object behaviors are easily and dynamically combined. In ABS, any method may be called both synchronously and asynchronously. Recently, this notion of cooperative scheduling has been integrated in Java by means of concurrent object groups [25].

We work with an ABS kernel language for distributed concurrent objects in which asynchronous method calls and futures form the basic communication constructs. The proposed kernel language combines the concurrency model of ABS with explicit language constructs for error recovery. In particular, both the caller and callee may signal a failure: the caller by performing a $x := f.$**`kill`** operation (reminiscent of the **`cancel`** method of Java futures) on the future f identifying the call, while the callee by executing the **`abort`** n command (n describes the kind of failure). If the callee aborts, then it will definitely terminate its activities. On the contrary, if the caller performs $x := f.$**`kill`**, it expects that the callee will react by executing some compensating activity (in contrast to Java, where the call is just interrupted). Such activities are attached to the **`return`** statement, that we replace with the new command **`return`** e **`on compensate`** s (where s is the compensation code). This is the main novelty of our proposal: when a callee successfully completes, it has not definitely completed its activity as it will possibly have to perform its compensation activity in case of failure of the caller. This mechanism is inspired by the compensation mechanisms adopted in service orchestration languages like WS-BPEL [23] or Jolie [10]. A compensation can return to the caller some results: to this aim we use a new future which is freshly created and assigned to x by $x := f.$**`kill`**.

Paper structure. Section 2 introduces our ABS kernel language (without error handling) and presents its syntax and semantics. Section 3 proposes fault-handling primitives for ABS and discusses by simple examples the typical patterns of interaction between the caller and callee under our model of failures. Section 4 discusses the operational semantics of the new primitives and their impact on the ABS type system. Section 5 concludes the paper.

2 A Language for Distributed Concurrent Objects

We consider *ABS*, an abstract behavioral specification language for distributed concurrent objects (modifying Creol [9, 16] by, e.g., excluding class inheritance and dynamic class upgrades). Characteristic features of ABS are that: (1) it allows abstracting from implementation details while remaining executable; i.e., a *functional sub-language* over abstract data types is used to specify internal, sequential computations; and (2) it provides *flexible concurrency and synchronization mechanisms* by means of asynchronous method calls, release points in method definitions, and cooperative scheduling of method activations.

Intuitively, concurrent ABS objects have dedicated processors and run in a distributed environment with asynchronous and unordered communication. Communication between objects is based on asynchronous method calls. (There is no remote field access.) Calls are asynchronous as the caller may decide at runtime when to synchronize with the reply from a call. Method calls may be seen as triggers, spawning new concurrent activities (so-called *processes*) in the called object. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is *active*. The others are *suspended* in a process pool. Process scheduling is non-deterministic, but controlled by *processor release points* in a cooperative way.

An ABS *model* defines interfaces, classes, datatypes, and functions, and a `main` method to configure the initial state. We elide the definition of data types and functions to focus on the concurrency and communication aspects of ABS models. Objects are dynamically created instances of classes; their declared attributes are initialized to arbitrary type-correct values. This paper assumes that models are well-typed, so method binding is guaranteed to succeed.

The concurrent object language of ABS is given in Fig. 1. Here, an interface *IF* has a name I and method signatures Sg. A class implements interfaces specifying types for its instances. A class CL has a name C, interfaces $\overline{I}$, class parameters and state variables x of type T, and methods M. (The *attributes* of the class are its parameters and state variables.) A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, local variable declarations $\overline{x}$ of types $\overline{T}$, and a body with statement s. Statements may access attributes of the current class, locally defined variables, and the method's formal parameters.

Right-hand side expressions rhs include object creation **new** $C(\overline{e})$, communication constructs (discussed below), and expressions e. Expressions include Boolean expressions, the read-only self-reference **this**, references x to attributes and local variables, and functional terms (omitted here). Statements are standard for assignment $x := rhs$, sequential composition $s_1; s_2$, **skip**, **if**, **while**, and **return** constructs. The **release** statement unconditionally releases the processor, suspending the active process. In **await** $\overline{g}$ **do** $\{s\}$, the guards g control processor release and consist of Boolean conditions b and return tests $x?$ (see below). If all guards g evaluate to false, the processor is released and the process *suspended*. When the processor is idle, any enabled process from the object's pool of suspended processes may be scheduled.

Syntactic categories.	*Definitions.*
C, I, m in Names	$IF ::= \textbf{interface}\ I\ \{\ \overline{Sg}\ \}$
g in Guard	$CL ::= \textbf{class}\ C\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{I}]\ \{\ \overline{T\ x};\ \overline{M}\}$
s in Statement	$Sg ::= T\ m\ (\overline{T\ x})$
e in Expression	$M ::= Sg\{\overline{T\ x};\ s\}$
b in Bool Expression	$g ::= b \mid x? \mid g \wedge g \mid g \vee g$
	$e ::= b \mid x \mid \textbf{this} \mid \ldots$
	$s ::= s; s \mid x := rhs \mid \textbf{release} \mid \textbf{await}\ \overline{g\ \textbf{do}\ \{s\}} \mid \textbf{skip}$
	$\mid \textbf{if}\ b\ \textbf{then}\ \{\ s\ \}\ [\textbf{else}\ \{\ s\ \}] \mid \textbf{while}\ b\ \{\ s\ \} \mid \textbf{return}\ e$
	$rhs ::= e \mid \textbf{new}\ C[(\overline{e})] \mid e!m(\overline{e}) \mid x.\textbf{get}$

Fig. 1. ABS syntax for the concurrent object language.

Communication in ABS is based on asynchronous method calls, denoted $o!m(\overline{e})$. After an asynchronous call $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable, o is an object (an expression typed by an interface), and $\overline{e}$ are expressions. A future variable x refers to a return value which has yet to be computed. There are two operations on future variables, which control external synchronization in ABS. First, a return test $x?$ evaluates to false unless the reply to the call can be retrieved. (Return tests are used in guards.) Second, the return value is retrieved by the expression $x.\textbf{get}$, which blocks execution in the object until the return value is available. The statement sequence $x := o!m(\overline{e});\ v := x.\textbf{get}$ encodes a blocking, *synchronous call*, abbreviated $v := o.m(\overline{e})$ whereas the statement sequence $x := o!m(\overline{e});\ \textbf{await}\ x?\ \textbf{do}\ v := x.\textbf{get}$ encodes a non-blocking, *preemptable call*, abbreviated $\textbf{await}\ v := o.m(\overline{e})$.

2.1 Operational Semantics

The operational semantics of ABS is presented as a transition system in an SOS style [24]. The rules, given in Fig. 2, apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules (i.e., matching is modulo associativity and commutativity as in rewriting logic [20]). We denote by $[\![e]\!]_\sigma^{cn}$ a confluent and terminating reduction system which reduces expressions e to data values (from a set *Val*) in a substitution σ and a configuration *cn*. (In particular, $[\![x?]\!]_\sigma^{cn} = \text{true}$ if $[\![x]\!]_\sigma^{cn} = f$ and $\mathit{fut}(f, v) \in cn$ for some value $v \neq \bot$, otherwise $[\![x?]\!]_\sigma^{cn} = \text{false}$. The remaining cases are fairly straightforward.)

Configurations cn are sets of objects, invocation messages, and futures. The associative and commutative union operator on configurations is denoted by whitespace. Configurations live inside curly brackets; in the term $\{cn\}$, *cn* captures the *entire* configuration. An *object* is a term $ob(o, a, p, q)$ with identifier o, an attribute mapping a representing the object's fields, an *active process* p, and a *pool of suspended processes* q. A process p consists of a mapping l of local variable bindings and a list s of statements, denoted by $\{l|s\}$ when convenient. In an *invocation message* $\mathit{invoc}(o, f, m, \overline{v})$, o is the callee, f the future to which the call's result is returned, m the method name, and $\overline{v}$ the call's actual parameter

$$\textsc{(Assign1)}\quad \frac{x \in \mathrm{dom}(l) \quad v = \llbracket e \rrbracket^{\varepsilon}_{(a \circ l)}}{\begin{array}{c} ob(o, a, \{l | x := e; s\}, q) \\ \to ob(o, a, \{l[x \mapsto v] | s\}, q) \end{array}}$$

$$\textsc{(Assign2)}\quad \frac{x \in \mathrm{dom}(a) \quad v = \llbracket e \rrbracket^{\varepsilon}_{(a \circ l)}}{\begin{array}{c} ob(o, a, \{l | x := e; s\}, q) \\ \to ob(o, a[x \mapsto v], \{l | s\}, q) \end{array}}$$

$$\textsc{(Bind-Mtd)}\quad \frac{p' = \mathrm{bind}(o, f, m, \overline{v})}{\begin{array}{c} ob(o, a, p, q)\ invoc(o, f, m, \overline{v}) \\ \to ob(o, a, p, \mathrm{enqueue}(p', q)) \end{array}}$$

$$\textsc{(Async-Call)}\quad \frac{o' = \llbracket e \rrbracket^{\varepsilon}_{(a \circ l)} \quad \overline{v} = \llbracket \overline{e} \rrbracket^{\varepsilon}_{(a \circ l)} \quad \mathrm{fresh}(f)}{\begin{array}{c} ob(o, a, \{l | x := e!m(\overline{e}); s\}, q) \\ \to ob(o, a, \{l | x := f; s\}, q) \\ invoc(o', f, m, \overline{v})\ fut(f, \bot) \end{array}}$$

$$\textsc{(Return)}\quad \frac{v = \llbracket e \rrbracket^{\varepsilon}_{(a \circ l)} \quad l(\mathrm{destiny}) = f}{\begin{array}{c} ob(o, a, \{l | \mathrm{return}\ e; s\}, q)\ fut(f, \bot) \\ \to ob(o, a, \{l | s\}, q)\ fut(f, v) \end{array}}$$

$$\textsc{(Await1)}\quad \frac{\llbracket g_i \rrbracket^{cn}_{(a \circ l)}}{\begin{array}{c} \{ob(o, a, \{l | \mathrm{await}\ \overline{g_i\ \mathbf{do}\ s_i}; s\}, q)\ cn\} \\ \to \{ob(o, a, \{l | s_i; s\}, q)\ cn\} \end{array}}$$

$$\textsc{(Await2)}\quad \frac{\forall i. \neg \llbracket g_i \rrbracket^{cn}_{(a \circ l)}}{\begin{array}{c} \{ob(o, a, \{l | \mathrm{await}\ \overline{g_i\ \mathbf{do}\ s_i}; s\}, q)\ cn\} \\ \to \{ob(o, a, \{l | \mathrm{release}; \mathrm{await}\ \overline{g_i\ \mathbf{do}\ s_i}; s\}, q)\ cn\} \end{array}}$$

$$\textsc{(Read-Fut)}\quad \frac{v \neq \bot \quad f = \llbracket e \rrbracket^{\varepsilon}_{(a \circ l)}}{\begin{array}{c} ob(o, a, \{l | x := e.\mathrm{get}; s\}, q)\ fut(f, v) \\ \to ob(o, a, \{l | x := v; s\}, q)\ fut(f, v) \end{array}}$$

$$\textsc{(Release)}\quad \begin{array}{l} ob(o, a, \{l | \mathrm{release}; s\}, q) \\ \to ob(o, a, \mathrm{idle}, \\ \qquad \mathrm{enqueue}(\{l | s\}, q)) \end{array}$$

$$\textsc{(Activate)}\quad \frac{p = \mathrm{select}(q, a, cn)}{\begin{array}{c} \{ob(o, a, \mathrm{idle}, q)\ cn\} \\ \to \{ob(o, a, p, q \backslash p)\ cn\} \end{array}}$$

Fig. 2. ABS semantics.

values. A *future* $fut(f, v)$ has an identifier f and a reply value v (which is $\bot$ when the reply value has not been received). Values are object and future identifiers, Boolean expressions, and null (as well as expressions in the functional language). For simplicity, classes are not represented explicitly in the semantics, but may be seen as static tables. We assume given a function $bind(o, f, m, \overline{v})$ which returns a process resulting from the activation of m in the class of o with actual parameters $\overline{v}$, callee o and associated future f; and a predicate $fresh(i)$ asserts that a name i is globally unique (where i may be an identifier for an object or a future). Let *idle* denote any process $\{l|s\}$ where s is an empty statement list.

Transition Rules. There are different assignment rules for expressions (*Assign1* and *Assign2*), method calls (*Async-Call*), and future dereferencing (*Read-Fut*). Rules *Assign1* and *Assign2* assign the value of expression e to a variable x in the local variables l or in the fields a, respectively. Here and in the sequel, the variable s will match any (possibly empty) statement list. (We omit the standard rules for skip, if-then-else, and while and the rule for object creation.)

Process Suspension and Activation. Three operations manipulate a process pool q: enqueue(p, q) adds a process p to q, $q \setminus p$ removes the process p from q, and select(q, a, cn) selects a process from q (if q is empty or no process is *ready*, this is the idle process [16]). The different possible definitions correspond to different process scheduling policies. Let $\emptyset$ denote the empty pool. Rule *Release* suspends the active process to the pool, leaving the active process idle. Rule *Await1* consumes the await statement if one of the guards evaluates to true in the current state and selects the related continuation, rule *Await2* adds a release to suspend the process if all the guards evaluate to false. Rule *Activate* selects a process from the pool for execution if this process is *ready* to execute, i.e., if it would not directly be resuspended or block the processor [16].

$$\begin{array}{rcll} s & ::= & \ldots & \textit{Standard statements} \\ & | & \mathbf{abort}\ n & \text{(Abort)} \\ & | & \mathbf{return}\ e\ \mathbf{on\ compensate}\ s & \text{(Return)} \\ & | & \mathbf{on}\ x := f.\mathbf{get}\ \mathbf{do}\ s\ \overline{\mathbf{on\ fail}\ n\ s} & \text{(Get)} \\ & & & \\ rhs & ::= & \ldots & \textit{Standard rhs} \\ & | & f.\mathbf{kill} & \text{(Kill)} \end{array}$$

Fig. 3. Primitives for error handling

Communication. Rule *Async-Call* sends an invocation message to o' with the unique identity f (by the condition fresh(f)) of a new future, the method name m, and actual parameters $\overline{v}$. The return value of the new future f is undefined (i.e., $\perp$). Rule *Bind-Mtd* consumes an invocation method, placing the process corresponding to the method activation in the callee's process pool. A reserved local variable 'destiny' stores the identity of the future associated with the call. Rule *Return* places the return value in the call's associated future. Rule *Read-Fut* dereferences the future f if $v \neq \perp$, otherwise the object is *blocked.*

3 Primitives for Error Handling

In this section we describe the syntax and the informal semantics of the primitives we propose for distributed error handling. As already said, ABS communication is asynchronous and based on futures. Thus we extend this idea to allow also for error notification and management. We assume to have a set *Err* of fault names, ranged over by n. The sets *Err* of fault names and *Val* of values are disjoint. Consider a method invocation $x := o!m(\overline{e})$. The caller will use future x inside all primitives related to handling errors for this method invocation. In the callee instead the used future is implicit, since each method execution has an attached future, i.e. the one where the return value is put.

In order to deal with errors, we mainly have to extend statements s and right-hand sides rhs w.r.t. Fig. 1. Small extensions will be needed also for method signatures and types. The extended syntax for statements and right-hand sides is described in Fig. 3. One may have a look to Fig. 4 and Fig. 5, described in detail later, to see how those primitives can be used.

We have a new primitive, **abort** n, to be used by the callee to signal its failure to its caller. Name n is used to notify the kind of error, and is reminiscent of exception types in e.g. Java[3]. The **abort** statement concludes the execution of the method. Also, the primitive **return** is extended with the clause **on compensate** s. This clause declares that, after the **return** has been executed and the method's normal execution completed, if compensation of this method call is needed, code s has to be executed. No continuation different from a compensation is allowed after **return**. Compensation s will be executed in the same environment of the method body.

[3] Our approach can be generalized to exception types, we choose to have just names for simplicity.

The two primitives above are executed by the callee. The caller has primitives for detecting the result of an invocation and for killing/compensating a past invocation. For detecting the result of the invocation we extend the $x := f.\texttt{get}$ primitive of ABS (we use f for an expression that evaluates to a future). It becomes part of the construct $\texttt{on}\ x := f.\texttt{get}\ \texttt{do}\ s\ \overline{\texttt{on fail}\ n_i\ s_i}$, which executes $x := f.\texttt{get}$ as before, but then it executes s if the future f contains a value v, or the clause s_i if f contains a fault name n_i. In the first case the value v is assigned to variable x, otherwise x is unchanged.

The primitives described so far allow errors generated in the callee to be managed. On the other side, the caller may enter an error situation that requires to annul the effect of the call. This is done using the statement $x := f.\texttt{kill}$. Here f is the future corresponding to the method call to be annulled while x is a variable that will contain the fresh future f' to be used to interact with the compensation. The annul request is asynchronous, and the result can be tested by using the normal **await** and **get** primitives. Upon the execution of $x := f.\texttt{kill}$ the value of f becomes the special value $\mathit{kill}(f')$, denoting that a kill request has been sent, and a reply is expected in f'. The identity of future f' is stored in x and f' is initialized to $\perp$. The annul request will be managed by the target method call either before it starts, or at its end. In the first case the method is not executed at all. In the second case, if execution was successful then the compensation code is executed, otherwise no code is executed (only successful executions can be compensated). The value of future f' is changed to a normal value v or to an error notification n depending on the result of the compensation code. Since f' is a future, one may even ask to kill an ongoing compensation. Two special fault notifications may be returned in such a future: `Ann`, specifying that either the method call has been annulled before starting or that it aborted on its own, and `NoC`, specifying that the killed method defined no compensation.

Note that both values and fault notifications unlock the **await** statement.

We clarify the error handling style induced by these primitives with a simple bank transfer example and a speculative parallelism example[4].

Example 1 (Bank transfer). Assume that a bank A wants to transfer some amount of money *money* from one of its accounts $accNoA$ to an account $accNoB$ of a bank B. Clearly, the interaction has to guarantee that money is neither created nor destroyed, and this should hold even in case of failures.

The codes of the caller and callee are in Fig. 4 and Fig. 5, respectively. The caller asks for the transfer by invoking method MAKETRANSFER (Fig. 4, line 3). If later on it finds a problem (e.g., there is not enough money in the source account[5]) it kills the MAKETRANSFER computation (Fig. 4, line 5). If the computation has already failed then nothing has to be done, and the clause **on fail** `Ann` (Fig. 4, line 9) is executed. If the MAKETRANSFER computation

[4] For simplicity, we avoid in the examples the typing of faults: this is considered in the following. We also shorten $x := x + e$ (resp. $x := x - e$) into $x \mathrel{+}= e$ (resp. $x \mathrel{-}= e$).

[5] This particular problem could have been checked before the invocation, but this is useful to show in a small example most of the error recovery mechanisms.

```
1  bool TRANSFER (int accNoA, int accNoB, int money)
2  { bool x, y;
3    f := bankB!MAKETRANSFER (int accNoB, int money);
4    if accountsA[accNoA].balance < money then
5       { f' := f.kill;
6         on x := f'.get
7            do return false
8            on fail no-money abort lost-money
9            on fail Ann return false
10      }
11 else
12    { await f? do
13         on y := f.get
14            do accountsA[accNoA].balance -= money;
15               return x
16            on fail no-acc
17               return false
18 }  }
```

Fig. 4. Bank transfer example: caller.

has not started yet, it is annulled and the same clause line **on fail** Ann is executed on the caller. If the MAKETRANSFER computation has successfully terminated then a compensation has been installed and its execution is started (when the scheduler decides to schedule it). When executing the compensation one does not know whether the received money is still available. In fact, the lock has been released after completion of method MAKETRANSFER, and another method of bank B may have used the money. If the money is still available then compensation is successful (we will see that the invariant has been preserved), clause **do** is executed on the caller (Fig. 4, line 7), and value $false$ is given as a result. If the money is no longer available then the **abort** *no-money* (Fig. 5, line 12) statement triggers the **on fail** *no-money* clause on the caller (Fig. 4, line 8). In the caller this will cause an **abort** *lost-money* signaling at the upper level that error recovery has not been successful. This is the only case where money is not preserved, but this is notified by a failure to the upper level. Note that in case of successful termination of the caller (which may happen even if the call failed), the TRANSFER method returns true if the transfer has been performed, false otherwise.

Example 2 (Speculative parallelism). As an additional example of the usage of the error handling primitives, we consider a typical pattern of service composition —the so-called *speculative parallelism.* This pattern generalizes client-server interaction to cases in which several servers can provide to the client the required reply. In these cases, the client asynchronously calls all the possible (alternative) servers, and then waits for the replies. The first reply will be accepted, while the other calls will be killed.

```
bool MAKETRANSFER (int accNo, int money)
{ if not valid(accNo) then
     { abort no-acc }
  else
     { accountsB[accNo].amount += money;
       return true
          on compensate
             if accountsB[accNo].amount >= money then
                { accountsB[accNo].amount -= money;
                  return false }
             else
                { abort no-money }
}    }
```

Fig. 5. Bank transfer example: callee.

Consider, for instance, a concert ticket reservation method (the client of the pattern is specified in Fig. 6) that invokes two possible reservation services (Fig. 6, lines 2-3). The servers are specified in Fig. 7. When one of the two requests succeeds, the other is canceled. If the first one fails, the second one is waited for. Only if both of them abort (Fig. 6, lines 10 and 17), the failure *no_ticket* is propagated to the upper level. In case both of them will succeed, only one request will be considered (according to which of the two branches of the **await** clause will be selected): the other one will be compensate via the **kill** mechanism (Fig. 6, lines 6 and 13).

4 Semantics for Error Handling

In this section we extend the ABS semantics in Fig. 2 to include the error handling primitives discussed above. The rules defining the semantics for error handling in Fig. 8 are to be added to the ones of Fig. 2, but for rules *Return* and *Read-Fut*, which are supposed to replace the homonymous rules in Fig. 2.

In order to understand the rules, one has to keep in mind the different states a future can have. Futures are created with a value $\perp$, saying that no result from the invoked method is available yet. The callee can store in the future either a value $v \in Val$ specifying the return value of a successful method, or a failure notification $n \in Err$ in case of abort of the method call. On the other side the caller can store in the future the kill request $kill(f')$ where f' is the fresh future used for receiving the result of the kill request.

Rules *Return-Comp1* and *Return-Comp2* model successful return and installation of a compensation. The two rules differ since in the first case the return value is stored in the future, in the second case it is discarded because of a kill request. The compensation is installed by letting it precede by a **release**, forcing the terminated method to release the lock, and by an **onkill** f statement. This last is runtime syntax. Its semantics is defined by rules *Onkill1* and *Onkill2*. In case there is a $kill(f')$ inside future f the effect of **onkill** f is to

```
int CONCERT_TICKET (int concert_code)
{ f1 := service1!RESERVE_TICKET (int concert_code);
  f2 := service2!RESERVE_TICKET (int concert_code);
  await f1? do
     on x := f1.get
        do { f3 := f2.kill;
              return x }
        on fail no_ticket
           on y := f2.get do return y
                          on fail no_ticket abort no_ticket,
  f2? do
     on x := f2.get
        do { f3 := f1.kill;
              return x }
        on fail no_ticket
           on y := f1.get do return y
                          on fail no_ticket abort no_ticket
}
```

Fig. 6. Client in the “speculative parallelism” example.

```
int RESERVE_TICKET (int concert_code)
{ await x := this.AVAILABLE(concert_code)
          //returns the ticket code or -1
  if (x = -1)
     then abort no_ticket
     else return x on compensate f := this!CANCEL(concert_code,x)
}
```

Fig. 7. Server in the “speculative parallelism” example.

change the special local variable *destiny* to f' so specifying that the result of the compensation will be advertised on future f'. Otherwise it releases the lock and checks again later. Notice that the standard **return** statement corresponds just to a **return** installing the default compensation **abort** NoC (rule *Return*).

Rules *Abort1* and *Abort2* define the **abort** n primitive. Essentially, it stores the fault name n in the future and releases the lock. In case the future contains $kill(f')$ instead the fault name is not stored, and f' is set to fault name Ann.

Rules *Kill1* and *Kill2* perform **kill**. In *Kill1* future f (containing a value, possibly $\perp$) is set to $kill(f')$ and future f' is created and set to $\perp$. In *Kill2* instead f contained a failure notification, thus f' is set to Ann. Rule *Kill3* deals with killing of an already killed method call: simply the existing reference to the result of the **kill** is assigned to the variable. Thus **kill** is essentially idempotent. Rule *Pre-Kill* discards a method invocation which has not started yet and which has to be killed. The future waiting for the result is set to Ann.

The last two rules are used for getting the result of a method invocation (or of a kill). If the value in the future is a non $\perp$ data value (rule *Read-Fut*) then it is

(Return)

$$\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{return } e\}, q) \\ \to \mathit{ob}(o, a, \{l \mid \text{return } e \text{ on compensate abort } \texttt{NoC}\}, q) \end{array}$$

(Return-Comp1)

$$\frac{v = [\![e]\!]^{\varepsilon}_{(a \circ l)} \quad l(\text{destiny}) = f}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{return } e \text{ on compensate } s\}, q)\ \mathit{fut}(f, \bot) \\ \to \mathit{ob}(o, a, \{l \mid \text{release}; \text{onkill } f; s\}, q)\ \mathit{fut}(f, v) \end{array}}$$

(Return-Comp2)

$$\frac{l(\text{destiny}) = f}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{return } e \text{ on compensate } s\}, q)\ \mathit{fut}(f, \text{kill}(f')) \\ \to \mathit{ob}(o, a, \{l \mid \text{release}; \text{onkill } f; s\}, q)\ \mathit{fut}(f, \text{kill}(f')) \end{array}}$$

(Onkill1)

$$\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{onkill } f; s\}, q)\ \mathit{fut}(f, \text{kill}(f')) \\ \to \mathit{ob}(o, a, \{l[\text{destiny} \mapsto f'] \mid s\}, q)\ \mathit{fut}(f, \text{kill}(f')) \end{array}$$

(Onkill2)

$$\frac{y \neq \text{kill}(f')}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{onkill } f; s\}, q)\ \mathit{fut}(f, y) \\ \to \mathit{ob}(o, a, \{l \mid \text{release}; \text{onkill } f; s\}, q)\ \mathit{fut}(f, y) \end{array}}$$

(Abort1)

$$\frac{l(\text{destiny}) = f}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{abort } n\}, q)\ \mathit{fut}(f, \bot) \\ \to \mathit{ob}(o, a, \{l \mid \text{release}\}, q)\ \mathit{fut}(f, n) \end{array}}$$

(Abort2)

$$\frac{l(\text{destiny}) = f}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{abort } n\}, q)\ \mathit{fut}(f, \text{kill}(f'))\ \mathit{fut}(f', x) \\ \to \mathit{ob}(o, a, \{l \mid \text{release}\}, q)\ \mathit{fut}(f, \text{kill}(f'))\ \mathit{fut}(f', \texttt{Ann}) \end{array}}$$

(Kill1)

$$\frac{f = [\![e]\!]^{\varepsilon}_{(a \circ l)} \quad \text{fresh}(f') \quad v \in \mathit{Val}}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid x := e.\text{kill}; s\}, q)\ \mathit{fut}(f, v) \\ \to \mathit{ob}(o, a, \{l \mid x := f'; s\}, q)\ \mathit{fut}(f, \text{kill}(f'))\ \mathit{fut}(f', \bot) \end{array}}$$

(Kill2)

$$\frac{f = [\![e]\!]^{\varepsilon}_{(a \circ l)} \quad \text{fresh}(f') \quad n \in \mathit{Err}}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid x := e.\text{kill}; s\}, q)\ \mathit{fut}(f, n) \\ \to \mathit{ob}(o, a, \{l \mid x := f'; s\}, q)\ \mathit{fut}(f, \text{kill}(f'))\ \mathit{fut}(f', \texttt{Ann}) \end{array}}$$

(Kill3)

$$\frac{f = [\![e]\!]^{\varepsilon}_{(a \circ l)}}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid x := e.\text{kill}; s\}, q)\ \mathit{fut}(f, \text{kill}(f')) \\ \to \mathit{ob}(o, a, \{l \mid x := f'; s\}, q)\ \mathit{fut}(f, \text{kill}(f')) \end{array}}$$

(Pre-Kill)

$$\begin{array}{c} \mathit{invoc}(o, f, m, \overline{v})\ \mathit{fut}(f, \text{kill}(f'))\ \mathit{fut}(f', \bot) \\ \to \mathit{fut}(f, \text{kill}(f'))\ \mathit{fut}(f', \texttt{Ann}) \end{array}$$

(Read-Fut)

$$\frac{v \in \mathit{Val} \quad v \neq \bot \quad f = [\![e]\!]^{\varepsilon}_{(a \circ l)}}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{on } x := e.\text{get do } s' \ \overline{\text{on fail } n_i \ s_i}; s\}, q)\ \mathit{fut}(f, v) \\ \to \mathit{ob}(o, a, \{l \mid x := v; s'; s\}, q)\ \mathit{fut}(f, v) \end{array}}$$

(Read-Err)

$$\frac{n_j \in \mathit{Err} \quad f = [\![e]\!]^{\varepsilon}_{(a \circ l)} \quad (\text{on fail } n_j \ s_j) \in \overline{\text{on fail } n_i \ s_i}}{\begin{array}{c} \mathit{ob}(o, a, \{l \mid \text{on } x := e.\text{get do } s' \ \overline{\text{on fail } n_i \ s_i}; s\}, q)\ \mathit{fut}(f, n_j) \\ \to \mathit{ob}(o, a, \{l \mid s_j; s\}, q)\ \mathit{fut}(f, n_j) \end{array}}$$

Fig. 8. ABS semantics for error handling

assigned to the variable x and clause **`do`** is executed. If it is an error notification (rule *Read-Err*) instead the corresponding clause **`on fail`** is executed.

$$\frac{\Gamma \vdash x : \texttt{fut}\langle T\rangle}{\Gamma \vdash x.\texttt{get} : T}\;(\textsc{Get}) \qquad \frac{\Gamma \vdash e : \Gamma(\text{return})}{\Gamma \vdash \texttt{return}\ e}\;(\textsc{Return}) \qquad \frac{\Gamma \vdash e : T' \qquad T' \preceq \Gamma(v)}{\Gamma \vdash v := e}\;(\textsc{Assign})$$

Fig. 9. Sample typing rules for ABS.

4.1 Typing

ABS relies on a type system guarenteeing that method binding always succeeds. One can extend the type system to additionally ensure that faults are managed in a correct way, in particular that all the faults that may be raised by a method invocation are managed by the caller.

While referring to [15] for a full description of the type system, we report in Fig. 9 the more interesting rules, and extend them to deal with error handling.

We use typing contexts which are mappings from names (of variables, interfaces and classes) to types. The reserved name *return* is bound to the return type of the current method. Relation $\preceq$ is the subtyping relation.

To check the correctness of error management, one has essentially to tag a method with the list of failures it can raise. Also, one has to specify the behavior of the compensation, including (recursively) its ability to throw faults. According to this idea, the signature Sg of a method m becomes:

$$\begin{aligned} Sg &::= T\ m\ (\overline{T\ x})\ ED \\ ED &::= \texttt{throws}\ \overline{n}\ [\texttt{on comp}\ T\ ED] \end{aligned}$$

Here $\overline{n}$ is the list of names of faults method m may throw. The optional clause **on comp** T ED specifies the typing of the compensation. It is omitted if the compensation is not present. In this case it stands for the (infinite) unfolding of the type **on comp null throws** NoC **rec** X.**on comp null throws** ε X where **null** is a subtype of any data type and ε the empty list.

As an example, the signatures of methods TRANSFER in Fig. 4 and MAKE-TRANSFER in Fig. 5 are respectively:

```
bool TRANSFER(int accNoA, int accNoB, int money)
     throws lost-money
bool MAKETRANSFER(int accNo, int money)
     throws no-acc on comp bool throws no-money
```

Similarly, futures have to declare the kinds of faults they are supposed to manage. The type declaration of a future becomes:

$$T ::= \ \ldots\ \mid\ \texttt{fut}\langle T\rangle\ ED \qquad \text{where } ED \text{ is as before.}$$

We show in Fig. 10 the main typing rules for error recovery. We need two reserved names: *faults*, bound to the list of faults that the current method can throw, and *comp*, bound to the typing of the current compensation. Rule *T-Abort* simply checks that the thrown fault is allowed. Rule *T-Get* verifies that the returned value has the correct type, and that all faults that may be raised by the callee are managed. Rule *T-Return* checks the type of the returned value,

$$\frac{n \in \Gamma(\text{faults})}{\Gamma \vdash \texttt{abort}\ n}\ (\textsc{T-Abort}) \qquad \frac{\Gamma \vdash x : T \quad \Gamma \vdash f : \texttt{fut}\langle T' \rangle\ \texttt{throws}\ \overline{n_i}\ CM \quad T' \preceq T \quad \Gamma \vdash s \quad \Gamma \vdash s_i}{\Gamma \vdash \texttt{on}\ x := f.\texttt{get do}\ s\ \overline{\texttt{on fail}\ n_i\ s_i}}\ (\textsc{T-Get})$$

$$\frac{\Gamma \vdash e : \Gamma(\text{return}) \qquad \Gamma(comp) = T\ \texttt{throws}\ \overline{n}\ CM \qquad \Gamma[return \mapsto T, faults \mapsto \overline{n}, comp \mapsto CM] \vdash s}{\Gamma \vdash \texttt{return}\ e\ \texttt{on compensate}\ s}\ (\textsc{T-Return})$$

$$\frac{\Gamma \vdash x : T' \qquad \texttt{fut}\langle T \rangle\ \texttt{throws}\ \overline{m_i}, \texttt{Ann}\ CM \preceq T' \qquad \Gamma \vdash f : \texttt{fut}\langle T'' \rangle\ \texttt{throws}\ \overline{n_i}\ \texttt{on comp}\ T\ \texttt{throws}\ \overline{m_i}\ CM}{\Gamma \vdash x := f.\texttt{kill}}\ (\textsc{T-Kill})$$

Fig. 10. Sample typing rules for error management in ABS.

and ensures that the compensation has the expected behavior. Finally rule *T-Kill* controls that variable x can store the result of the **kill**, including the possibility for it to be Ann.

The subtyping relation $\preceq$ has to be defined also on the new types for futures. It can be defined by:

$$\frac{T \preceq T' \quad \overline{n} \supseteq \overline{n'} \quad T_1 \preceq T_1' \quad ED \preceq ED'}{\texttt{fut}\langle T \rangle\ \texttt{throws}\ \overline{n}\ [\texttt{on comp}\ T_1\ ED] \preceq \texttt{fut}\langle T' \rangle\ \texttt{throws}\ \overline{n'}\ [\texttt{on comp}\ T_1'\ ED']}\ (\textsc{Fut-Sub})$$

The type system is easily extended from statements to configurations. Then, a standard subject reduction theorem holds for configurations, ensuring that well-typed configurations evolve to well-typed configurations. Finally, it is possible to prove that in well-typed configurations whenever a **get** statement receives a fault n, it provides a corresponding **on fail** n s clause for managing it.

5 Conclusion and Future Work

Taking inspiration from models and languages for fault and compensation handling like the Sagas calculi [6] and the orchestration languages WS-BPEL [23] and Jolie [10], we have presented an extension of the concurrent object-oriented language ABS (the reference language in the European Projects HATS [13]) with primitives for error handling. Callee side faults are similar to exceptions as in e.g. Java, while the use of compensations for managing caller side kill requests is novel for the object-oriented world as far as we know. Our main contribution has been to combine these two mechanisms in a coherent way, suitable for a language with asynchronous communication based on futures. This approach has been developed by ensuring that the main principles underlying ABS were preserved, in particular concerning collaborative scheduling and asynchronous method execution. These features of ABS are needed so to allow compositional correctness proofs based on invariants, similarly e.g. to what done in [3, 9]. In fact, under collaborative scheduling processes may ensure that an invariant holds only at release points. If all pieces of code ensure that the invariant holds before any release point by assuming that it holds at the beginning of their execution, then the invariant holds under any possible scheduling, without any need to check for interferences.

Invariants shed some light on a typical problem of the compensation approach, namely compensation correctness. In fact, compensations are supposed to take the process back to a consistent state which is however different from the state where the process started. In other words, the rollback is not perfect. For instance, in [7] this is kept into account by relying on an user-defined equivalence on states: the compensations should lead the program to a state which is equivalent to the one where the program itself started. In a world where programs are equipped with invariants, compensation correctness becomes clearer: compensation should restore the invariant which has been broken by the failure.

Let us consider Example 1. There we have a distributed invariant specifying that the sum of the money in the source and target accounts should not change. Since this invariant involves two distinct objects, it may not hold when there are pending method invocations, kills or when at least one of the objects has not released the lock. It is easy to check that the invariant is preserved by the normal execution, where no fault happens. Interestingly, it is also preserved in case of faults which are handled. In fact, in case of fault in the callee the money is never removed from the starting account, and in case of failure in the caller the compensation withdraws the excess of money from the callee. The only case where the invariant is violated is if the caller wants to compensate the call, but this is no more possible because the money has already been used by the callee. This is also the only case where method TRANSFER aborts. Thus one can say that the code satisfies the invariant above in the sense that either it aborts, or the invariant holds when the call is terminated, independently on the number of (successfully managed) failures. In this sense we can say that the compensation code in the example is correct w.r.t. this specific invariant.

As future work, we plan to develop general techniques for proving correctness of compensations using invariants. Moreover, to better evaluate the practical impact of our proposal, we will implement the proposed primitives in ABS and we will investigate the possibility to apply our fault handling model in other object-oriented languages with futures.

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. G. Agha and R. Ziaei. Security and fault-tolerance in distributed systems: an actor-based approach. In *Proc. of CSDA'98*, pages 72–88. IEEE Computer Society Press, 1998.
3. W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2010. In press.
4. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
5. L. Baduel et al. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer, 2006.

6. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. of POPL '05*, pages 209–220. ACM Press, 2005.
7. L. Caires, C. Ferreira, and H. Vieira. A process calculus analysis of compensations. In *Proc. of TGC'08*, volume 5474 of *LNCS*, pages 87–103. Springer, 2008.
8. D. Caromel. Service, Asynchrony, and Wait-By-Necessity. *Journal of Object Oriented Programming*, pages 12–22, Nov. 1989.
9. 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.
10. C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. Dynamic error handling in service oriented applications. *Fundam. Inform.*, 95(1):73–102, 2009.
11. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
12. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.*, 7(4):501–538, 1985.
13. European Project HATS. `http://www.cse.chalmers.se/research/hats/`.
14. International Telecommunication Union. Open Distributed Processing — Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.
15. E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In *Proc. of FM'09*, volume 5850 of *LNCS*, pages 596–611. Springer, 2009.
16. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.
17. *JSR166: Concurrency utilities.* `http://java.sun.com/j2se/1.5.0/docs/guide/concurrency`.
18. 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.
19. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proc. of PLDI'88*, pages 260–267. ACM Press, 1988.
20. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
21. B. Morris. CActive and Friends. Symbian Developer Network, November 2007. `http://developer.symbian.com/main/downloads/papers/CActiveAndFriends/CActiveAndFriends.pdf`.
22. P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming.* PhD thesis, Department of Computer Science, ETH Zurich, 2007.
23. Oasis. *Web Services Business Process Execution Language Version 2.0.* `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.
24. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.
25. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. of ECOOP 2010*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.
26. N. Venkatasubramanian and C. L. Talcott. Reasoning about meta level activities in open distributed systems. In *Proc. PODC'95*, pages 144–152. ACM Press, 1995.
27. A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.