Program Verification for Exception Handling on Active Objects Using Futures

Crystal Chang Din[1](✉), Rudolf Schlatte[1], and Tzu-Chun Chen[2]

[1] Department of Informatics, University of Oslo, Oslo, Norway
{crystald,rudi}@ifi.uio.no

[2] Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany
tc.chen@dsp.tu-darmstadt.de

Abstract. For implementing correct systems, handling and recovering from exceptional situations is important but challenging for ensuring correct interactions among distributed objects which are processing concurrently. To focus on exploring novel handling constructs for actor-based programming languages, we study ABS, an actor-based concurrent modeling language with an underlying executable formal semantics. This paper introduces multi-party session blocks with recovery handlers for exceptions into ABS. With this novel construct, we verify the correctness of interactions among objects within a session block. Program correctness is ensured by specifying invariants as pre- and post-conditions, called session contracts, for such a block, which is more expressive than the existing class invariant proof system for ABS. We present the extension of ABS with a try-catch-finally construct and class session recovery blocks that handle uncaught exceptions.

1 Introduction

Properly handling and recovering from exceptional situations is an important part of specifying and implementing robust and correct systems, especially for distributed systems where correctness must take partial failure scenarios into account [17]. Therefore, modeling languages should include means of specifying exceptional situations and how to recover from them. This paper presents a new approach to expressing multi-party exception transmission and recovery for active object languages [4]. We designed the approach for the modeling language ABS [13]. This paper adds standard language constructs to specify, raise and handle exceptional situations, as well as a novel construct, the *session block*, for reestablishing object invariants after unhandled exceptions.

Existing class invariant-based proof theories for ABS [7] are restricted in expressivity, specifically in the area of upholding guarantees of protocols involving series of message exchanges between multiple participants. The problem is that the semantics of ABS process interleaving and scheduling cannot forbid arbitrary messages to be processed in-between the expected ones, requiring whole-program analysis. This paper addresses this problem by introducing the concept of *sessions*, which temporarily restrict the scheduling behavior to the parts of a model participating in the session. In this work, we define *session contracts* to express the desired properties of a session based on the new session construct. A proof system for session contracts is introduced.

The rest of the paper is structured as follows. Section 2 describes the main characteristics of the ABS language. Section 3 introduces the new language constructs. Section 4 introduces session contracts and provides a proof system for verifying session contracts. Section 5 discusses related work. Section 6 discusses future work and concludes the paper.

2 A Short Introduction to ABS

The ABS language was developed to model distributed, parallel systems. Its design makes it amenable to both formal analysis and simulation (execution). The syntax is similar to languages in the C/Java family tree. ABS is an actor-based active object language, with interface inheritance and code reuse via traits. Being an active object language means that objects are "heavy-weight": method calls create processes on the target object, which are scheduled cooperatively in each concurrent object group (cog). Process switching occurs only when the current process terminates or at clearly marked program locations (*await* statements); this makes models of concurrent and distributed systems amenable to compositional analysis and proof. Data is modeled via a functional sub-language consisting of algebraic datatype definitions and side effect-free functions.

2.1 A Brief Example

Figure 1 shows a complete ABS model simulating bank accounts and transactions involving multiple accounts. The `Account` interface and `CAccount` class model a bank account with the usual deposit, withdrawal and balance inspection methods. Methods of type `Unit`, e.g., `deposit`, can omit an explicit `return Unit;` statement. The `Transaction` interface and `CTransaction` class model the control flow that models a transaction involving transferring some funds from one account to another, with a small commission transferred to a third account. The method `transfer` of the `Transaction` class (Line 17) first deducts the given amount from the sender account, then calculates the commission and deposits the proper amounts in the receiver and commission accounts.[1]

ABS object references are typed via interfaces, which describe the set of messages that an object can process (lines 3, 8). Classes (lines 11, 16) implement zero or more interfaces and contain method definitions. Method calls (e.g., Line 20) are *asynchronous*, written `o!m()`, and create a new process in the callee.

[1] The slightly awkward calculation of `profit` is used to introduce a runtime error.

```
 1  module BankAccount;
 2
 3  interface Account {
 4     Unit deposit (Rat amount);
 5     Unit withdraw (Rat amount);
 6     Rat getBalance();
 7  }
 8  interface Transaction {
 9     Unit transfer (Account f, Account t, Rat amount);
10  }
11  class CAccount(Rat balance) implements Account {
12     Unit deposit (Rat amount) { balance = balance + amount; }
13     Unit withdraw (Rat amount) { balance = balance - amount; }
14     Rat getBalance() { return balance; }
15  }
16  class CTransaction(Account commission, Rat factor) implements Transaction {
17     Unit transfer (Account sender, Account receiver, Rat amount) {
18        await sender!withdraw(amount);
19        Rat profit = amount / factor;
20        commission!deposit(profit);
21        receiver!deposit(amount - profit);
22     }
23  }
24  {
25     Account f = new CAccount(50);
26     Account t = new CAccount(50);
27     Account c = new CAccount(0);
28     Transaction trans = new CTransaction(c, 10);
29     await trans!transfer(f, t, 10);
30     Fut<Rat> fp = c!getBalance();
31     await fp?;
32     Rat profit = fp.get;
33     println("Profit: " + toString(profit));
34  }
```

Fig. 1. A motivating example

Execution in the caller continues in parallel with the new process. The value of a method call is a *future* (see Line 30), which can be used to synchronize with the resulting process (Line 31) and to obtain the result (Line 32). Abbreviated syntax makes it possible to omit an explicit future definition to synchronize with the callee and, optionally, obtaining the result (see Line 18).

One question is what the behavior of an asynchronous method call is, when it immediately followed by a `fp.get` expression, e.g., omitting Line 31 in the example. In this case, the `get` expression *blocks* until `fp` has a value. Blocking means that the cog will not schedule another process. There exists abbreviated syntax for this kind of call: instead of `f = o!m(); v = f.get;` one can write `v = o.m();`. This notation is used in examples later in this paper.

Finally, the behavior of a model is specified via its *main block* (Lines 24–34).

2.2 Asynchronous Method Calls, Scheduling Points, and Object Groups

The concurrency model of ABS merits some more explanation. The unit of concurrency in ABS is the *concurrent object group* (cog). Each cog contains a number of objects and cooperatively schedules the processes running on these objects such that at most one process per cog is running. As mentioned in Sect. 2.1, each asynchronous method call results in a process being created at callee-side that executes the method named in the call. Figure 2 shows the relation of processes, cogs, and sessions (sessions are introduced in Sect. 3.4). So, the two processes created by the method calls in Fig. 1, Line 20 and 21 can run in parallel provided they are not running in the same cog.

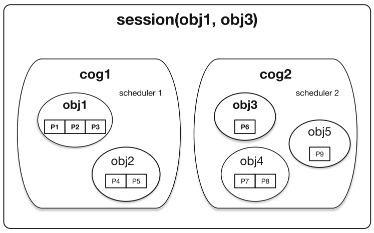

Fig. 2. Cogs contain objects, which run processes. A session temporarily "captures" its set of participants. The session names participating objects whose cogs join the session; other objects in the cog cannot join a different session at the same time.

A cog schedules a process to run when its currently running process reaches a *scheduling point*. A scheduling point occurs when a process terminates, either by executing its return statement or via an unhandled exception, or at the point of an `await` or `suspend` statement. The cog will choose the next process to run non-deterministically from its set of *runnable processes*. A process is runnable after it has been freshly created, after a `suspend` statement, and after an `await` statement if the condition in the `await` statement is true.

Cogs and cooperative scheduling makes modeling distributed concurrent systems easy and safe. Processes in different cogs are running in parallel, but do not have access to shared state. Processes within the same cog, on the other hand, can share state if they run on the same object, but are running interleaved, with scheduling points clearly visible at the source code level.

3 Exception Recovery in ABS

This section describes the new constructs added to the ABS language for modeling exceptional situations, handling exceptions and recovering from unhandled exceptions, and multi-party sessions.

The current ABS language documentation can be found at [1]. A formal semantics of ABS can be found in [13]. Figures 3 and 4 summarize the syntax of the ABS functional and imperative layer, respectively. Parts highlighted in yellow mark the elements added in this paper.

Syntactic categories	*Definitions*
T in Ground Type	$T ::= B \mid I \mid D \mid D\langle\overline{T}\rangle \mid E$
B in Basic Type	$B ::= \mathsf{Bool} \mid \mathsf{Int} \mid \cdots$
A in Type	$A ::= N \mid T \mid D\langle\overline{A}\rangle$
N in Name	$Dd ::= \mathtt{data}\ D[\langle\overline{A}\rangle] = Cons[\overline{\mid Cons}];$
E in Exception	$Cons ::= Co[(\overline{A})]$
x in Variable	$E ::= \mathtt{exception}\ Co[(\overline{A})];$
e in Expression	$F ::= \mathtt{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A\ x}) = e;$
b in Bool Expression	$e ::= b \mid x \mid t \mid \mathtt{this} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \mathtt{case}\ e\ \{\overline{br}\}$
t in Ground Term	$t ::= Co[(\overline{t})] \mid \mathtt{null}$
br in Branch	$br ::= p \Rightarrow e;$
p in Pattern	$p ::= \_ \mid x \mid t \mid Co[(\overline{p})]$

Fig. 3. Core ABS syntax for the functional level. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over corresponding syntactic categories, square brackets [ ] denote optional elements. (Color figure online)

Syntactic categories.	*Definitions.*
s in Stmt	$P ::= \overline{IF}\ \overline{CL}\ \{[\overline{T\ x};]\ s\}$
e in Expr	$IF ::= \mathtt{interface}\ I\ \{\ [\overline{Sg}]\ \}$
b in BoolExpr	$CL ::= \mathtt{class}\ C\ [(\overline{T\ x})]\ [\mathtt{implements}\ \overline{I}]\ [\mathtt{recover}\ \{\ \overline{cbr}\ \}]\ \{\ [\overline{T\ x};]\ \overline{M}\}$
g in Guard	$Sg ::= T\ m\ ([\overline{T\ x}])$
cbr in Catch branch	$M ::= Sg\ \{[\overline{T\ x};]\ s\ \}$
	$s ::= \{\ s\ \} \mid s; s \mid \mathtt{skip} \mid x = rhs \mid \mathtt{if}\ b\ \{\ s\ \}\ [\mathtt{else}\ \{\ s\ \}] \mid \mathtt{while}\ b\ \{\ s\ \}$
	$\mid \mathtt{suspend} \mid \mathtt{await}\ g \mid \mathtt{return}\ e$
	$\mid \mathtt{try}\ s\ \mathtt{catch}\ \{\ \overline{cbr}\ \}\ [\mathtt{finally}\ s] \mid \mathtt{throw}\ e$
	$\mid \mathtt{session}(\overline{e})\{\ s\ \}\ [\mathtt{recover}\ \{\ \overline{cbr}\ \}]$
	$rhs ::= e \mid e.m(\overline{e}) \mid e!m(\overline{e}) \mid x.\mathtt{get} \mid \mathtt{new}\ [\mathtt{local}]\ C\ (\overline{e})$
	$cbr ::= p\ \mathtt{=>}\ s$
	$g ::= b \mid x?$

Fig. 4. Syntax for the imperative layer of ABS. Notation as in Fig. 3 (Color figure online)

3.1 Exception Modeling in the Functional Layer

Algebraic data structures in ABS are defined with the keyword `data`, which defines both a type and a set of constructors. Exceptions are defined with the keyword `exception`, which introduces a named constructor for the new exception. The type of an exception is always `ABS.StdLib.Exception`, which is predefined in the ABS standard library. Exceptions can be used as data values. For example, they can be stored in lists and can be used in the `case` pattern-matching expression. Additionally, exceptions are used as argument to the `throw` statement and are pattern-matched in `catch` branches (see Sect. 3.2 below).

3.2 Exception Handling in the Imperative Layer

The imperative layer of ABS adds a `throw` statement for manually raising exceptions. Additionally, normal code execution can also lead to exceptions, like attempting to send a message to `null` or dividing by zero.

For handling exceptions, the imperative layer of ABS adds the familiar `try-catch-finally` construct. Exceptions raised in the statement(s) protected by `try` are pattern-matched by the branches in the `catch` block; the statements in the first matching branch are then executed ("the exception is *handled* by that branch"). Finally, all statements in the `finally` block are executed, regardless of how the `try` block was executed. In case no catch branch matches ("the exception is *unhandled*"), the `finally` block is executed and the exception is (hopefully) handled by an enclosing `try-catch` block. The scope of variables declared in the `try` block does not extend to the `catch` and `finally` blocks since they might not have been initialized yet when entering these blocks. To ensure progress, `finally` blocks cannot contain blocking operations or process suspension.

Unhandled exceptions terminate the current process and are stored in its future. As in [10], unhandled exceptions propagate across futures. When the callee process terminated with an exception, that exception will be raised when the caller tries to obtain the future's value via a `get` expression, and will thereby propagate along the chain of process invocations until it is handled.

Note that a process crash is effectively ignored if no other process tries to access its return value.

3.3 Recovery in the Object Layer

The compositional proof system of ABS [5,7] relies on class invariants; processes are responsible to establish these invariants at all of their scheduling points. Since with exceptions processes can terminate at arbitrary points, we introduce *recovery blocks* as a fall-back mechanism to reestablish class invariants.

All unhandled exceptions still lead to process termination, as above in Sect. 3.2, but additionally the unhandled exception is matched against the recovery block given in the class definition. If a matching branch is found, its statements are executed and the object is kept alive. If no matching branch is found in the recovery block, the object is killed. A dead object is marked as invalid, all processes running on it are terminated, and all further messages to that object result in an exception in the caller. This is not quite as draconian as it sounds, since models of distributed systems need to model this type of partial failure anyway.

3.4 Session Blocks

As discussed above, try-catch blocks and class recovery blocks help restore per-object class invariants in the face of exceptional situations. But they do not help in a systematic way for recovering invariants that span more than one object. In general, this requires corrective actions undoing or compensating from messages sent as part of an incomplete transaction. For example, see Fig. 1: when creating a `CTransaction` object with `factor`$= 0$, executing the `transfer` method will lead to a division by zero on Line 19, after sending a `withdraw` message to `sender` but before the corresponding `deposit` messages. Hence, the system-wide invariant ("the amount of money in the system is constant") is violated. To handle these cases, we introduce the `session block` construct.

A *session* is the analogue of a critical section over a group of cogs. During the lifetime of a session, the participating cogs will only run processes that "belong" to the session. Unrelated processes are not scheduled until the session has ended. Sessions are implemented and modeled via *session blocks*. The cog running the process that is executing the session block (the "session initiator") is a *session participant*, as are the cogs of all objects named in the session block parameter list. In Fig. 2 we see a session with two participants. For the duration of a session, all participants will only schedule processes that are created during the session by a session participant. There can be multiple active sessions in the system, but no cog can participate in more than one session at a time.

Figure 4 introduces the syntax of the `session` block. Figure 5 shows an example of this construct, in a revised `CTransaction` class. Note the use of local variables `start_sender` etc. is to record progress through the session and establish which actions in the `CAccount` objects can be undone. As with `try-catch-finally`, variables declared in the body of the session block go out of scope before entering the recovery block, since their value and status are uncertain.

The semantics of initiating and terminating a session demands synchronization among all participants. When the session initiator starts executing a session block, the list of participating cogs is calculated from the block's parameter list.

```
class CTransaction(Account commission, Rat factor) implements Transaction {
  Unit transfer (Account sender, Account receiver, Rat amount) {
    Maybe<Rat> start_sender = Nothing;
    Maybe<Rat> start_receiver = Nothing;
    Maybe<Rat> start_commission = Nothing;
    session(sender, receiver, commission) {
      Rat sb = sender.getBalance(); start_sender = Just(sb);
      Rat rb = receiver.getBalance(); start_receiver = Just(rb);
      Rat cb = commission.getBalance(); start_commission = Just(cb);
      sender.withdraw(amount);
      Rat profit = amount / factor;
      commission.deposit(profit);
      receiver.deposit(amount - profit);
    } recover {
      _ => {
        if (isJust(start_sender)) {
          Rat bal_sender = sender.getBalance();
          sender.deposit(fromJust(start_sender) - bal_sender);
        }
        if (isJust(start_commission)) {
          Rat bal_commission = commission.getBalance();
          commission.withdraw(bal_commission - fromJust(start_commission));
        }
        if (isJust(start_receiver)) {
          Rat bal_receiver = receiver.getBalance();
          receiver.withdraw(bal_receiver - fromJust(start_receiver));
        }
      }
    }
  }
}
```

Fig. 5. Error recovery in the transaction class via a session block

In Fig. 5, Line 6, there are four participants (the cogs of the three `Account` objects plus the cog of the `Transaction` object, which runs the session initiator). Execution of the initiating process blocks until all participants have (a) left any currently active sessions they might be in, and (b) have reached a scheduling point. Then, all participants acknowledge entering the session and receive the list of participants. When the session initiator reaches the end of the session body, either normally or via an exception, execution blocks until all participants have finished executing all processes that are part of the session. A final synchronization point is at the end of the recovery block, in case it is entered.

4 Program Analysis of Session Blocks with Exception Handlers

A session block, introduced in Sect. 3, is used to identify a special group of interactions in which (i) the states of participants in the interactions shall not be updated by other processes, and (ii) once an exception occurs but is not caught by `catch` block, the recovery block will recover the states of participants.

In this section, we give a session-contract based verification framework. This verification framework is inspired by the ABS class-invariant based verification [5], which, however, is not designed for verifying the preservation of invariants while exceptions are thrown or verifying properties across multiple objects, such as in the case in Fig. 1.

4.1 Session Contracts

In this section we first briefly explain the class-invariant based verification framework for ABS [7]. Then we will point out why this proof strategy is too strong for the language setting where exception handling is considered. The verification framework in [7] assumes formal specification at the class level, i.e. for each object implemented in a class C we aim to establish its class invariant I_{C}. We need to prove that C's initialization block establishes I_{C}, and I_{C} holds before process releasing at each `await` and `suspend` statements, as well as when a method on C returns. Thus, class invariants need to hold at each scheduling points but not necessary in between. Consequently, if an exception is thrown between two scheduling points, this may lead to a system ending in a state where class invariant does not hold. For instance, we define a specification for the banking example in Fig. 1.

$$\texttt{sender.balance} + \texttt{commission.balance} + \texttt{receiver.balance} = v \tag{1}$$

which says that the summation of balances of the sender's account, the receiver's account and the commission's account is a constant v. This property cannot be proven within the verification framework for ABS [7]. One reason is that the specification language used in [7] cannot express the state of the invoked objects, and this property does not hold at every suspension point as it should in [7], for instance, after the balance of the `sender` has been decreased but the balance of other accounts have not yet been changed. Besides, if there is any runtime error, for example division by zero, this property does not hold when an exception is thrown.

To overcome these restrictions, the concept of *session* is introduced in this work. The modified version of the banking example using *session* is presented in Fig. 5, for which we define Eq. (1) as a *session contract*. Session contracts express the state of the session or the communication pattern between objects in the same session. They are assumed at session entry and should be proven at session exit. Accordingly, the following statement should be proven upon session termination in Fig. 5.

$$v - \texttt{amount} + \texttt{profit} + (\texttt{amount} - \texttt{profit}) = v$$

In case of uncaught exceptions in the session block, the session contract should hold after the recovery block. In order to prevent the session state from being randomly modified at the process release points, we only allow process suspension outside the session blocks.

4.2 Proof System

In this section we introduce a modular proof system for proving session-based ABS programs. We first prove that each method satisfies its method contract and then prove that each session block satisfies its corresponding session contract.

4.2.1 Program Analysis at Method Level

We verify ABS methods against method contracts by applying the proof rules in Fig. 6, i.e. one rule for each program statement. The program logic is first-order dynamic logic for ABS (ABSDL) [2,5,7]. For a sequence of executable ABS statements $\mathcal{S}$ and ABSDL formulae P and Q, the formula $P \Rightarrow [\mathcal{S}]\,Q$ expresses: If the execution of $\mathcal{S}$ starts in a state where the assertion P holds and the program terminates normally, then the assertion Q holds in the final state. Gentzen-style sequent calculus is used to prove ABSDL formulae. In sequent notation, $P \Rightarrow [\mathcal{S}]\,Q$ is written $P \vdash [\mathcal{S}]Q$. A sequent calculus as realized in ABSDL essentially constitutes a symbolic interpreter for ABS. For example, the method rule in Fig. 6 expresses the proof of method `m` against its precondition p and postcondition q. In the assign rule, the assignment $v = e$ is an active statement in a modality $[\pi\ v = e; \omega]$, where v is a program variable and e is a pure (side effect-free) expression. The nonactive prefix π consists of an arbitrary sequence of opening braces "{", i.e. beginnings "$\mathtt{m}(\overline{\mathtt{x}})${" of method blocks, "`try{`" of `try-catch-finally` blocks, and "$\mathtt{session}(\overline{e})${" of session blocks. The remaining program is represented by ω. The assign rule generates a so-called update [2], as $\{v := e\}$ shown above, for the assignment statement, which captures state changes and is placed outside the modality box. Updates can be viewed as explicit substitutions that accumulate in front of the modality during symbolic program execution. We use $\mathcal{U}$ to represent the accumulated updates up to now. Updates can only be applied to formulae or terms. Once the program to be verified has been completely executed and the modality is empty (see the emptyBox rule in Fig. 6), the accumulated updates are applied to the formula after the modality, resulting in a pure first-order formula. Γ stands for (possibly empty) sets of side formulae, and ϕ the property required to be proven upon execution termination.

$$\textsf{assign}\ \frac{\Gamma \vdash \mathcal{U}\{v := e\}[\pi\ \omega]\phi}{\Gamma \vdash \mathcal{U}[\pi\ v = e; \omega]\phi} \qquad \textsf{skip}\ \frac{\Gamma \vdash \mathcal{U}[\pi\ \omega]\phi}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{skip}; \omega]\phi}$$

$$\textsf{new}\ \frac{\Gamma, cl(o, C) \vdash \mathcal{U}(\mathit{fresh}(o) \Rightarrow \{v := o\}[\pi\ \omega]\phi)}{\Gamma \vdash \mathcal{U}[\pi\ v = \texttt{new}\ C(\overline{e}); \omega]\phi} \qquad \textsf{method}\ p \vdash [\mathsf{m}(\overline{\mathsf{x}})\{s\}]\,q$$

$$\textsf{ifElse}\ \frac{\begin{array}{c}\Gamma \vdash \mathcal{U}(b \Rightarrow [\pi\ s_1\ \omega]\phi)\\ \Gamma \vdash \mathcal{U}(\neg b \Rightarrow [\pi\ s_2\ \omega]\phi)\end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{if}\ b\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\ \omega]\phi} \qquad \textsf{while}\ \frac{\begin{array}{l}\Gamma \vdash \mathcal{U}I\\ \Gamma, I \wedge b \vdash [s]I\\ \Gamma, I \wedge \neg b \vdash [\pi\ \omega]\phi\end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \texttt{while}\ b\ \{\ s\ \}; \omega]\phi}$$

$$\textsf{return}\ \frac{\Gamma, C.\mathsf{m}(\overline{\mathsf{x}}) : (p, q) \vdash (\mathcal{U}\{\mathtt{r} := e\}q) \wedge \mathcal{U}[\pi\ \omega]\phi}{\Gamma, C.\mathsf{m}(\overline{\mathsf{x}}) : (p, q) \vdash \mathcal{U}[\pi\ \texttt{return}\ e; \omega]\phi} \qquad \textsf{emptyBox}\ \frac{\Gamma \vdash \mathcal{U}\phi}{\Gamma \vdash \mathcal{U}[\ ]\phi}$$

$$\textsf{asyncCall}\ \frac{\begin{array}{c}\Gamma, cl(o, C), C.\mathtt{m}(\overline{\mathsf{x}}) : (p, q) \vdash \mathcal{U}\{\mathsf{this} := o\}\{\overline{\mathsf{x}} := \overline{e}\}p\\ \Gamma, cl(o, C), C.\mathtt{m}(\overline{\mathsf{x}}) : (p, q), bt(fr, o, C.\mathtt{m}(\overline{e})) \vdash \mathcal{U}\{fr := fr'\}[\omega]\phi\end{array}}{\Gamma, cl(o, C), C.\mathtt{m}(\overline{\mathsf{x}}) : (p, q) \vdash \mathcal{U}[fr = o!\mathtt{m}(\overline{e}); \omega]\phi}$$

$$\textsf{get}\ \frac{\begin{array}{l}\Gamma, bt(fr, o, C.\mathtt{m}(\overline{e})), C.\mathtt{m}(\overline{\mathsf{x}}) : (p, q), \mathit{isValue}(fr), \mathit{fresh}(v') \vdash\\ \qquad\qquad (\mathcal{U}\{\mathsf{this} := o\}\{\overline{\mathsf{x}} := \overline{e}\}\{\mathtt{r} := v'\}q) \Rightarrow \mathcal{U}\{v := v'\}[\pi\ \omega]\phi\\ \Gamma, bt(fr, o, C.\mathtt{m}(\overline{e})), C.\mathtt{m}(\overline{\mathsf{x}}) : (p, q), \neg\mathit{isValue}(fr), \mathit{fresh}(v') \vdash \mathcal{U}\{v := v'\}[\pi\ \texttt{throw}\ v;\ \omega]\phi\end{array}}{\Gamma, bt(fr, C.\mathtt{m}(\overline{e})), C.\mathtt{m}(\overline{\mathsf{x}}) : (p, q) \vdash \mathcal{U}[\pi\ v = fr.\mathtt{get}; \omega]\phi}$$

$$\textsf{syncCall}\ \frac{\Gamma, \mathcal{U}\mathit{fresh}(fr') \vdash \mathcal{U}[\pi\ fr' = o!\mathtt{m}(\overline{e}); v = fr'.\mathtt{get}; \omega]\phi}{\Gamma \vdash \mathcal{U}[\pi\ v = o.\mathtt{m}(\overline{e}); \omega]\phi}$$

Fig. 6. Proof rules for statements.

Figure 6 also provides rules for other statements. Rules skip, new, return, and get are for `skip` statements, object creation, return statements, and `get` statements, respectively. In rule `new`, *fresh*(o) expresses that there is no object reference equals o up to now. Object o belongs to class C is captured by predicate $cl(o, C)$. The rule ifElse is for conditional statements. The rule while proves that a while loop preserves loop invariant I. In the rule asyncCall we assume that method contract of the invoked method is provided. We formulate method contract in the form of $C.\mathsf{m}(\overline{\mathsf{x}}) : (p, q)$, where (p, q) is a pair of pre- and post-condition of method $\mathsf{m}(\overline{\mathsf{x}})$ in class C. For brevity, we skip the case of multiple implementations of a given interface but they can be handled in the standard way using adaptation rule [6]. The asyncCall rule has two premises. The first one proves that the precondition p of m holds. The update substitutes this with callee o, and formal parameters $\overline{\mathsf{x}}$ with actual parameters $\overline{e}$. In the second premise, a fresh future fr' is generated and added into an update clause. The environment carries information about the callee of fr', i.e. predicate $bt(fr, o, C.\mathtt{m}(\overline{e}))$ expresses that future fr belongs to method $\mathtt{m}(\overline{e})$ which is executed on object o of class C. In rule return, the keyword r captures the return value and the postcondition q is required to be proven. Note that we consider partial correctness, so for the get rule we assume it is possible to fetch the data from the future at the get statements eventually. Since it is not possible to know the exact fetched data while applying the get rule, we follow the same principle as for the new rule and assign a fresh value, i.e. v', to variable v. However, if the environment carries information about the callee of fr, we can use the post condition q to restrict the possible values v'. If such information is unavailable, we assume $q = true$. If future fr in the get statement does not contain value, i.e. $\neg isValue(fr)$, but an exception, an exception will be thrown. This is captured by the second premise. The rule syncCall is syntactic sugar to an asynchronous call plus a get statement. Note that we do not present the proof rules for await and suspend statements in this paper, because we only allow process suspension outside the session blocks.

$$\textsf{try-catch-finally}\ \frac{\begin{array}{c}\Gamma, e = t \vdash \mathcal{U}[\pi\ \mathtt{try}\{\overline{s_2}\}\ \mathtt{catch}\{\}\ \mathtt{finally}\{\overline{s_3}\}\ \omega]\phi \\ \Gamma, e \neq t \vdash \mathcal{U}[\pi\ \mathtt{try}\{\mathtt{throw}\ e;\ \overline{s_1}\}\ \mathtt{catch}\{\overline{cbr}\}\ \mathtt{finally}\{\overline{s_3}\}\ \omega]\phi\end{array}}{\Gamma \vdash \mathcal{U}[\pi\ \mathtt{try}\{\mathtt{throw}\ e;\ s_1\}\ \mathtt{catch}\{t \Rightarrow \overline{s_2}; \overline{cbr}\}\ \mathtt{finally}\{\overline{s_3}\}\ \omega]\phi}$$

$$\textsf{try-emptyCatch-finally}\ \frac{\Gamma \vdash \mathcal{U}[\pi\ \overline{s_2}; \mathtt{throw}\ e\ \omega]}{\Gamma \vdash \mathcal{U}[\pi\ \mathtt{try}\{\mathtt{throw}\ e;\ \overline{s_1}\}\ \mathtt{catch}\{\}\ \mathtt{finally}\{\overline{s_2}\}\ \omega]\phi}$$

$$\textsf{emptyTry}\ \frac{\Gamma \vdash \mathcal{U}[\pi\ \overline{s}\ \omega]\phi}{\Gamma \vdash \mathcal{U}[\pi\ \mathtt{try}\{\}\ \mathtt{catch}\{\overline{cbr}\}\ \mathtt{finally}\{\overline{s}\}\ \omega]\phi}$$

Fig. 7. Proof rules for `try-catch-finally` statements.

In Fig. 7 we provide proof rules for try-catch-finally statements. Runtime exceptions are handled in the proof rules (see the example of the get rule in Fig. 6). Errors created during evaluation of expressions, for example division by zero, are handled in a similar way. The rule try-catch-finally has two premises. If the thrown exception matches the first case listed in the catch block, statements s_2 from the catch clause are then executed. Otherwise, the exception is thrown again and the first case in the catch block is eliminated. Note that the `finally` block can be empty, i.e. s_3 is an empty list. The rule try-emptyCatch-finally expresses that the exception cannot be caught by the `catch` clause so it executes the `finally` clause and then throws the exception to the outer scope. Maybe there will be another `try-catch` clause around it, i.e. contained in the nonactive code π and the remaining program ω. The rule emptyTry says if a `try` clause is completely executed without throwing any exceptions, then the `finally` clause and the remaining program will be executed.

4.2.2 Proof of the Example at Method Level

In this section, we provide method contracts for the example shown in Sect. 3.4. The method contract for `withdraw` is

$$\texttt{CAccount.withdraw(Rat amount}_1\texttt{):(this.balance} = \texttt{balance}',\ \texttt{this.balance} = \texttt{balance}' - \texttt{amount}_1\texttt{)}$$

in which this.balance accesses the field balance, and logical variable balance′ stores the value of balance at the prestate. This contract expresses that if the value of `balance` at prestate is `balance`$'$, then it updates to `balance`$' -$ `amount`$_1$ upon method termination. The method contract for `deposit` is:

$$\texttt{CAccount.deposit(Rat amount}_2\texttt{):(}\mathsf{this.balance} = \mathsf{balance}''\texttt{, }\mathsf{this.balance} = \mathsf{balance}'' + \mathsf{amount}_2\texttt{)}$$

This contract expresses that if the value of `balance` at prestate is `balance`$''$, then it updates to `balance`$'' +$ `amount`$_2$ upon method termination. Finally, a method contract for `getBalance` is given below:

$$\texttt{CAccount.getBalance():(true, r} = \mathsf{this.balance}\texttt{)}$$

There is no requirement for the precondition. In the postcondition the value of `balance` is assigned to variable `r`. We can prove these three method contracts by using rules method, assign, return and emptyBox in Fig. 6.

4.2.3 Program Analysis at Session Level

Session contract must be proven at the end of the session if there is no exception left unhandled, assuming the session contract holds at the session entry. The proof rules in Fig. 8 together with the ones in Figs. 6 and 7 build up a proof system for verifying session blocks. The rules in Figs. 6 and 7 will be used when the session block is not empty and the current active statement is not a `throw` statement for exception. A session block may contain `try-catch-finally` clauses but cannot be nested within another session block.

$$\mathsf{sessionRecover}\ \frac{\begin{array}{c}\Gamma, e = t \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\overline{s_2}\}\ \texttt{recover}\{\}]\phi \\ \Gamma, e \neq t \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\texttt{throw}\ e;\ \overline{s_1}\}\ \texttt{recover}\{\overline{cbr}\}]\phi\end{array}}{\Gamma \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\texttt{throw}\ e;\ \overline{s_1}\}\ \texttt{recover}\{t \Rightarrow \overline{s_2}; \overline{cbr}\}]\phi}$$

$$\mathsf{emptyRecover}\ \frac{\Gamma \vdash \texttt{false}}{\Gamma \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\texttt{throw}\ e;\ r\}\ \texttt{recover}\{\}]\phi}$$

$$\mathsf{emptySession}\ \frac{\Gamma \vdash \mathcal{U}\phi}{\Gamma \vdash \mathcal{U}[\texttt{session}(\overline{e})\{\}\ \texttt{recover}\{\overline{s}\}]\phi}$$

$$\mathsf{sessionStart}\ SC, \overline{C.\mathsf{m}(\overline{\mathsf{x}})\ :\ (p, q)}, init \vdash [\texttt{session}(\overline{e})\{\overline{s_1}\}\ \texttt{recover}\{\overline{s_2}\}]SC$$

Fig. 8. Rules for proving session contracts.

The rule sessionRecover has two premises. If the thrown exception matches the first case listed in the `recover` block, statements s_2 from the `recover` clause are then executed. Otherwise, the same exception is thrown again and the first case in the `recover` block is eliminated. The rule emptyRecover says if none of the cases in the `recover` block matches the exception thrown from a `session` block, this proof branch cannot be closed. This means the `recover` block needs to be re-implemented until the session contract can be successfully proven in the end of the `recover` block. The rule emptySession says if a session block is completely executed without throwing any exceptions, then the session contract should be proven at this session exit. The rule sessionStart captures the proof obligation of a session. We use this rule to prove that a session preserves the session contract *SC*. Contracts for all the methods in the system are assumed known from the beginning. The set *init* are variables defined before session entry but used in the session block.

4.2.4 Proof of the Example at Session Level

In this section, we explain the proof outline for the example in Fig. 5, which presents the cases when exceptions are thrown in a session. Equation (1) is the corresponding session contract. Since the session involves method invocations, the proof requires knowledge of all the invoked methods from the session. This knowledge is formalized as the method contracts presented in Sect. 4.2.2.

An exception can be thrown at any execution point of the session block. Since there is an execution barrier between the session body and the recovery block (see Sect. 3.4), all the processes executed in or related to the session block are finished before the recovery block can be executed. The recovery block rescues all the possible failing cases and makes sure the program is back to the state as if this particular transaction has never been executed. Below we present the proof outline for the recovery block and show that the session contract holds at the end of the recovery block.

Session participants, i.e. objects, and their method invocations are known in a session. According to this knowledge, we instantiate each method contract of the invoked methods in the session as follows: In the method contract for the `deposit` method of the `sender` object, we instantiate this to `sender` and parameter $\texttt{amount}_2$ to start_sender − bal_sender. In the method contract for the `withdraw` method of the `commission` object, we instantiate this to `commission` and parameter $\texttt{amount}_1$ to bal_commission − start_commission; In the method contract for the `withdraw` method of the `receiver` object, we instantiate this to `receiver` and parameter $\texttt{amount}_1$ to bal_receiver − start_receiver.

```
sender.deposit(start_sender − bal_sender):
  (sender.balance = bal_sender,
    sender.balance = bal_sender + (start_sender − bal_sender))

commission.withdraw(bal_commission − start_commission):
  (commission.balance = bal_commission,
    commission.balance = bal_commission − (bal_commission − start_commission))

receiver.withdraw(bal_receiver − start_receiver):
  (commission.balance = bal_receiver,
    commission.balance = bal_receiver − (bal_receiver − start_receiver))
```

Assume in the beginning of the session block the following holds

$$\texttt{sender.balance} = sb \wedge \texttt{commission.balance} = cb \wedge \texttt{receiver.balance} = rb$$

where sb, cb, rb are logical variables to record the initial balance of the accounts and $sb + cb + rb = v$. Besides, start_sender $= sb$ when established, start_commission $= cb$ when established, and start_receiver $= rb$ when established. Depending on which initial balance of the accounts are successfully accessed in the session block, the conditional branches in the recovery block will be selected for execution. In the end of the recovery block we show that

$$\begin{aligned}
&\texttt{sender.balance} + \texttt{commission.balance} + \texttt{receiver.balance}\\
&\quad = [sb \mid \mathsf{bal\_sender} + (\mathsf{start\_sender} - \mathsf{bal\_sender})] +\\
&\qquad [cb \mid \mathsf{bal\_commission} - (\mathsf{bal\_commission} - \mathsf{start\_commission})] +\\
&\qquad [rb \mid \mathsf{bal\_receiver} - (\mathsf{bal\_receiver} - \mathsf{start\_receiver})]\\
&\quad = sb + cb + rb = v
\end{aligned}$$

in which the symbol $[a \mid b]$ means either a is selected or b is selected. Thus, this session contract does hold in the end of the recovery block. From these proof results, we show that the execution of a session will always end in a safe state, with the session contract reestablished, irregardless of the presence of exceptions.

5 Related Work

The use of futures for transferring both result and exception values goes back to [15]. A specification of an exception handling system for active objects using one-way asynchronous communication and interacting via a request/response protocol was presented in [9]. Future-based communication and verification of such system were not considered. The first approach for exception recovery based on object state rollback for unhandled errors in ABS was proposed in [10]. This approach was implemented but ultimately rejected for inclusion in the main language for two reasons: the impact of object rollback on the ABS proof theory was too high, and the proposed approach did not handle notions of correctness that need to be expressed over multiple objects. The approach in this paper addresses both of these shortcomings. Reasoning about exception handling in Java is supported by PVS [12]. The invariant-based verification framework for ABS was provided by [5,7], in which formal specification was at the class level. KeY-ABS [5] is a theorem prover that realized the proof system for ABS. It was developed based on KeY [2], which supports deductive verification for exception handling in sequential Java programs. A new rule implemented in KeY to reason about exception handling in loops was introduced by [16]. A modular and scalable network-on-chip example is proven by KeY-ABS. The proof results are shown in [8]. In the work of [14], core ABS is extended with sessions and annotations to express scheduling policies based on required communication ordering. The annotation is statically checked against the session automata derived from the session types.

6 Conclusion

This paper shows an extension of the Active Object-based modeling language ABS with exception specifications, handling and recovery. We introduce several language constructs including the means to express coordinated multi-party sessions and recovery actions. To guarantee session correctness, we provide session contracts and an attendant proof system. We show that a system, in which exceptions are thrown, can be recovered back to a safe state, where session contract holds. Soundness proofs for the reasoning system with respect to the operational semantics are left as future work. Other planned future work includes (1) building up our type system to describe allowed scheduling during the lifetime of the session, and (2) tool support for the new language constructs, including an implementation of the type checking and runtime semantics, and a thorough evaluation of the usability of session contracts in the context of existing case studies utilizing ABS. Inspired by behavioral types [3,11], our type system will be designed to regulate the runtime behavior of objects and schedulers, and reduce the number of exceptions caused by undesired communication behaviour.

Acknowledgement. This work was supported by the research projects CUMULUS: Semantics-based Analysis for Cloud-Aware Computing, ERC project LiveSoft, and the SIRIUS Centre for Scalable Data Access.

References

1. The ABS Development Team. ABS Documentation. http://docs.abs-models.org
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Chen, T.-C., Viering, M., Bejleri, A., Ziarek, L., Eugster, P.: A type theory for robust failure handling in distributed systems. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 96–113. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_7
4. de Boer, F.S., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
5. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 517–526. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_35
6. Din, C.C., Johnsen, E.B., Owe, O., Yu, I.C.: A modular reasoning system using uninterpreted predicates for code reuse. J. Log. Algebraic Methods Program. **95**, 82–102 (2018)
7. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Aspects Comput. **27**(3), 551–572 (2015)
8. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 217–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_14
9. Dony, C., Urtado, C., Vauttier, S.: Exception handling and asynchronous active objects: issues and proposal. In: Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A. (eds.) Advanced Topics in Exception Handling Techniques. LNCS, vol. 4119, pp. 81–100. Springer, Heidelberg (2006). https://doi.org/10.1007/11818502_5

10. Göri, G., Johnsen, E.B., Schlatte, R., Stolz, V.: Erlang-style error recovery for concurrent objects with cooperative scheduling. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8803, pp. 5–21. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45231-8_2
11. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 31–336 (2016)
12. Jacobs, B.: A formalisation of Java's exception mechanism. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 284–301. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45309-1_19
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
14. Kamburjan, E., Din, C.C., Chen, T.-C.: Session-based compositional analysis for actor-based languages using futures. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 296–312. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_19
15. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI), pp. 260–267 (1988)
16. Steinhöfel, D., Wasser, N.: A new invariant rule for the analysis of loops with non-standard control flows. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 279–294. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_18
17. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. In: Vitek, J., Tschudin, C. (eds.) MOS 1996. LNCS, vol. 1222, pp. 49–64. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62852-5_6