Asynchronous Cooperative Contracts for Cooperative Scheduling

Eduard Kamburjan[1], Crystal Chang Din[2],
Reiner Hähnle[1], and Einar Broch Johnsen[2]

[1] Department of Computer Science, Technische Universität Darmstadt, Germany
{kamburjan,haehnle}@cs.tu-darmstadt.de
[2] Department of Informatics, University of Oslo, Norway
{crystald,einarj}@ifi.uio.no

Abstract. Formal specification of multi-threaded programs is notoriously hard, because thread execution may be preempted at any point. In contrast, abstract concurrency models such as actors seriously restrict concurrency to obtain race-free programs. Languages with *cooperative scheduling* occupy a middle ground between these extremes by explicit scheduling points. They have been used to model complex, industrial concurrent systems. This paper introduces *cooperative contracts*, a contract-based specification approach for asynchronous method calls in presence of cooperative scheduling. It permits to specify complex concurrent behavior succinctly and intuitively. We design a compositional program logic to verify cooperative contracts and discuss how global analyses can be soundly integrated into the program logic.

1 Introduction

Formal verification of complex software requires decomposition of the verification task to combat state explosion. The *design-by-contract* [41] approach associates with each method a declarative contract capturing its behavior. Contracts allow the behavior of method calls to be *approximated* by static properties. Contracts work very well for sequential programs [4], but writing contracts becomes much harder for languages such as Java or C that exhibit a low-level form of concurrency: contracts become bulky, hard to write, and even harder to understand [10]. The main culprit is *preemption*, leading to myriads of interleavings that cause complex data races which are hard to contain and to characterize.

In contrast, methods in actor-based, distributed programming [7] are executed atomically and concurrency only occurs among actors with disjoint heaps. In this setting behavior can be completely specified at the level of interfaces, typically in terms of behavioral invariants jointly maintained by an object's methods [16, 19]. However, this restricted concurrency forces systems to be modeled and specified at a high level of abstraction, essentially as protocols. It precludes the modeling of concurrent behavior that is close to real programs, such as waiting for results computed asynchronously on the same processor and heap.

Active object languages [15] occupy a middle ground between preemption and full distribution, based on an actor-like model of concurrency [3] and *futures* to handle return values from asynchronous calls (e.g., [9, 13, 16, 21, 24, 40, 45]). ABS [33] is an active-object language which supports *cooperative scheduling* between asynchronously called methods. With cooperative scheduling, tasks may explicitly and voluntarily suspend their execution, such that a required result may be provided by another task. This way, method activations on the same processor and heap *cooperate* to achieve a common goal. This is realized using a guarded command construct **await** `f?`, where `f` is a reference to a future. The effect of this construct is that the current task suspends itself and only resumes once the value of `f` is available. Although only one task can execute at any time, several tasks may depend on the same condition, which may cause internal non-determinism.

The aim of this paper is to generalize method contracts from the sequential to the active object setting with asynchronous method calls, futures and cooperative scheduling. This generalization raises the following challenges:

1. **Call Time Gap.** There is a delay between the asynchronous invocation of a method and the activation of the associated process. During this delay, the called object ("callee") may execute other processes. To enter the callee's contract the precondition must hold. But even when that precondition holds at invocation time, it does not necessarily hold at activation time.
2. **Strong Encapsulation.** Each object has exclusive access to its fields. Since the caller object cannot access the fields of the callee, it cannot ensure the validity of a contract precondition that depends on the callee's fields.
3. **Interleaving.** In cooperative scheduling, processes interleave at explicitly declared scheduling points. At these points, it is necessary to know which functional properties will hold when a process is scheduled and which properties must be guaranteed when a process is suspended.
4. **Return Time Gap.** Active objects use futures to decouple method calls from local control flow. Since futures can be passed around, an object reading a future `f` knows in general neither to which method `f` corresponds nor the postcondition that held when the result value was computed.

The main contributions of this paper are (i) a formal *specification*-by-contract technique for methods in a *concurrency context* with asynchronous calls, futures, and cooperative scheduling; and (ii) a contract-based, compositional *verification* system for functional properties of asynchronous methods that addresses the above challenges. We call our generalized contracts *cooperative contracts*, because they cooperate through propagation of conditions according to the specified concurrency context. Their concrete syntax is an extension of the popular formal specification language JML [39]. We demonstrate by example that the proposed contracts allow complex concurrent behavior to be specified in a succinct and intelligible manner. Proofs can be found in our accompanying report [38].

2 Method Contracts for Asynchronous Method Calls

We introduce the main concepts of active object (AO) languages and present the methodology of our analysis framework in an example-driven way. AO languages model loosely coupled parallel entities that communicate by means of asynchronous method calls and futures (i.e., mailboxes). They are closely tied to the OO programming paradigm and its programming abstractions. We go through an example implemented in the ABS language [2,33], an AO modeling language with cooperative scheduling which has been used to model complex, industrial concurrent systems [5].

Running Example. We consider a distributed computation of *moving averages*, a common task in data analysis that renders long-term trends clearer in smoothened data. Given data points $x_1, \dots, x_n$, many forms of moving average $\mathsf{avg}(x_1, \dots, x_n)$ can be expressed by a function cmp that takes the average of the first $n-1$ data points, the last data point and a parameter α:

$$\mathsf{avg}(x_1, \dots, x_n) = \mathrm{cmp}(\mathsf{avg}(x_1, \dots, x_{n-1}), x_n, \alpha)$$

For example, an exponential moving average demands that α is between 0 and 1 and is expressed as $\mathsf{avg}(x_1, \dots, x_n) = \alpha * x_n + (1-\alpha) * \mathsf{avg}(x_1, \dots, x_{n-1})$.

Figure 1 shows the central class `Smoothing`. Each `Smoothing` instance holds a `Computation` instance `comp` in `c`, where the actual computation happens and cmp is encapsulated as a method. A `Smoothing` instance is called with `smooth`, passes the data piecewise to `c` and collects the return values in the list of intermediate results `inter`. During this time, it stays responsive: `getCounter` lets one inquire how many data points are processed already. Decoupling list processing and value computation increases usability: one `Smoothing` instance may be reused with different `Computation` instances. There are several useful properties one would like to specify for `smooth`: (i) `c` has been assigned before it is called and is not changed during its execution, (ii) no two executions of `smooth` overlap during suspension and (iii) the returned result is a smoothened version of the `input`.

We explain some specification elements. *Atomic segments* of statements between suspension points are assigned unique names, labeled by the *annotation* `[atom: "string"]` at an await statement. The named scope `"string"` is the code segment from the end of the previous atomic segment up to the annotation. The first atomic segment starts at the beginning of a method body, the final atomic segment extends to the end of a method body and is labeled with the method name. There are `sync` labels at future reads, which are used to identify the statement. We use a ghost field [31] `lock` to model whether an invocation of `smooth` is running or not. A ghost field is not part of the specified code. It is read and assigned in specification annotations which are only used by the verification system.

```
interface ISmoothing
    extends IPositive {
 Unit setup(Computation comp);
 Int getCounter();
 List<Rat>
  smooth(List<Rat> input, Rat a);
}
class Smoothing
   implements ISmoothing {
 Computation c = null;
 Int counter = 1;
 //@ ghost Bool lock = False;
 Unit setup(Computation comp) {
     c = comp;
 }
 Int getCounter() {
     return counter;
 }
```

```
 List<Rat> smooth(List<Rat> input, Rat a) {
  //@ lock = True;
  counter = 1;
  List<Rat> work = tail(input);
  List<Rat> inter = list[input[0]];
  while (work != Nil) {
   Fut<Rat> f = c!cmp(last(inter), work[0], a);
   counter = counter + 1;
   [atom: "awSmt"] await f?;
   [sync: "sync"] Rat res = f.get;
   inter = concat(inter, list[res]);
   work = tail(work);
  }
  //@ lock = False;
  counter = 1;
  return inter;
 }
}
```

Fig. 1. ABS code of the controller part of the distributed moving average

2.1 Specifying State in an Asynchronous Setting

During the delay between a method call and the start of its execution, method parameters stay invariant, but the heap may change. This motivates breaking up the precondition of asynchronous method contracts into one part for parameters and a separate part for the heap. The *parameter precondition* is guaranteed by the *caller* who knows the appropriate synchronization pattern. It is part of the callee's interface declaration and exposed to clients. (Without parameters, the parameter precondition is `true`.) The *callee* guarantees the *heap precondition*. It is declared in the class implementing the interface and not exposed to clients.

Example 1. The parameters of method `smooth` must fulfill the precondition that the passed data and parameter are valid. The heap precondition expresses that a `Computation` instance is stored in `c`.

```
interface ISmoothing { ...
/*@ requires 1 > a > 0 && len(input) > 0 @*/
List<Rat> smooth(List<Rat> input, Rat a); }
```

```
class Smoothing { ...
/*@ requires !lock && c != null @*/
List<Rat> smooth( ... ) { ... } }
```

To handle inheritance we follow [4] and implement behavioral subtyping. If `ISmoothing` extended another interface `IPositive`, the specification of that interface is *refined* and must be implied by all `ISmoothing` instances:

```
interface IPositive{ ...
  /*@ requires \forall Int i; 0 <= i < len(input) ; input[i] > 0 @*/
  List<Rat> smooth(List<Rat> input, Rat a); }
interface  ISmoothing extends IPositive { ... } // inherits parameter precondition
```

A caller must fulfill the called method's parameter precondition, but the most recently completed process inside the callee's object establishes the heap precondition. To express this a method is specified to run in a *concurrency context*, in addition to the memory context of its heap precondition. The concurrency context appears in a contract as two *context sets*: sets with atomic segment names:

- *Succeeds:* Each atomic segment in the context set *succeeds* must guarantee the heap precondition when it terminates and at least one of them must run before the specified method starts execution.
- *Overlaps:* Each atomic segment in the context set *overlaps* must preserve the heap precondition. Between the termination of the last atomic segment from *succeeds* and the start of the execution of the specified atomic segment, only atomic segments from *overlaps* are allowed to run.

Context sets are part of the interface specification and exposed in the interface. Classes may extend context sets by adding private methods and atomic segment names. Observe that context sets represent *global information* unavailable when a method is analyzed in isolation. If context sets are not specified in the code, they default to the set of *all* atomic segments, whence the heap precondition degenerates into a class invariant and must be guaranteed by each process at each suspension point [18]. Method implementation contracts need to know their expected context, but the global protocol at the object level can be specified and exposed in a separate coordination language, such as session types [30]. This enforces a separation of concerns in specifications: method contracts are local and specify a single method and its context; the coordination language specifies a global view on the whole protocol. Of course, local method contracts and global protocols expressed with session types [36,37] must be proven consistent. Context sets can also be verified by static analysis once the whole program is available (see Sect. 2.3).

Example 2. The heap precondition of `smooth` is established by `setup` or by the termination of the previous `smooth` process. Between two sessions (and between `setup` and the start of the first session) only `getCount` may run. Recall that the method name labels the final atomic segment of the method body.

Postconditions (*ensures*) use two JML-constructs: *\result* refers to the return value and *\last* evaluates its argument in the state at the *start* of the method. We specify that the method returns a strictly positive list of equal length to the input, which is bounded by the input list. Furthermore, the object is not locked. For readability, irrelevant parts of the contracts are omitted.

```
interface ISmoothing { ...
/*@ succeeds {setup, smooth};
     overlaps {getCounter};  @*/
List<Rat> smooth(List<Rat> input, Rat a); }
class Smoothing { ...
/*@ ensures !lock && len(\result) == len(input) &&
              \forall Int i; 0 <= i < len(\result);
                \result[i] > 0 && min(input) <= \result[i] <= max(input); @*/
List<Rat> smooth(List<Rat> input, Rat a) { ... } }
```

The specified concurrency context is used to *enrich* the existing method contracts: the heap precondition of a method specified with context sets is implicitly *propagated* to the postcondition of all atomic segments in *succeeds*, and to pre- *and* postconditions of all atomic segments in *overlaps*.

Example 3. We continue Example 2. After propagation, the specifications of `setup`, `smooth` and `getCounter` are as follows. The origin of the propagated formula is indicated in comments.

```
/*@ ensures <as before> && !lock && c != null // succeeds smooth @*/
List<Rat> smooth(List<Rat> input, Rat a) { ... }
/*@ ensures !lock && c != null // succeeds smooth @*/
Unit setup(Computation comp) { ... }
/*@ ensures \last(!lock && c != null) -> !lock && c != null // overlaps smooth @*/
Int getCounter() { ... }
```

In case of inheritance, the context sets of the extended interface are implicitly included in those of the extending class or interface. A class may extend context sets with private methods not visible to the outside. It is the obligation of that class to ensure that private methods do not disrupt correct call sequences from the outside. From an analysis point of view, private methods are no different than public ones.

2.2 Specifying Interleavings

An **await** statement introduces a scheduling point where process execution may be suspended and possibly interleaved with the execution of other processes. From a local perspective, the **await** statement can be seen as a *suspension point* where information about the heap memory is lost. This can be addressed by similar reasoning as for heap preconditions: What is guaranteed at the release of control, what can be assumed upon reactivation, and who has the obligation to guarantee the heap property. Hence, each suspension point is annotated by a *suspension contract* containing the same elements as a method contract: An *ensures* clause for the condition that holds upon suspension, a *requires* clause for the condition which holds upon reactivation, a *succeeds* context set for the atomic segments which must have run before reactivation and an *overlaps* context set for atomic segments whose execution may interleave. (As method names label the final atomic segments, all such atomic segments contain a **return** statement. A name may refer to multiple atomic segments in case of, for example, loops.)

Example 4. We specify the behavior of the suspension point at the **await** statement with label `"awSmt"` (below left): At the continuation, the object is still locked and the `Computation` instance `c` must be present. During suspension, only the method `getCounter` is allowed to run. By adding the method itself to the `succeeds` set, we ensure that the suspension has to establish its own suspension assumption. The specification after *propagation* is shown below right. (The propagation from context sets into pre- and postconditions of suspension contracts is analogous to the procedure for method contracts.)

```
/*@ requires lock && c != null;
      ensures True;
      succeeds {awSmt};
      overlaps {getCounter}; @*/
[atom: "awSmt"] await f?;
```

```
/*@ requires lock && c != null;
      ensures lock && c != null;
      succeeds {awSmt};
      overlaps {getCounter}; @*/
[atom: "awSmt"] await f?;
```

The postcondition of `getCounter` is now as follows and encodes a case distinction.

```
/*@ ensures \last(!lock && c != null) -> !lock && c != null // overlaps smooth
          && \last( lock && c != null) ->  lock && c != null // overlaps awSmt @*/
Int getCounter() { ... }
```

2.3 Composition

The specification above is modular in the following sense: To prove that a method adheres to the pre- and postcondition of its own contract and respects the pre- and postcondition of called methods, only requires to analyze its own class. To verify that a system respects all context sets, however, requires global information, because the call order is not established by a single process in a single object. This separation of concerns between functional and non-functional specification allows to decompose verification into two phases that allow reuse of contracts. In the first phase, deductive verification [17] is used to *locally* show that single methods implement their pre- and postconditions correctly. In the second phase, a *global* light-weight, fully automatic dependency analysis is used to approximate call sequences. In consequence, if a method is changed with only local effects it is sufficient to re-prove its contract and re-run the dependency analysis. The proofs of the other method contracts remain unchanged.

The dependency analysis of context sets is detailed in the technical report [38]; we only give an example for rejected and accepted call sequences here.

Example 5. Consider the three code fragments interacting with a `Smoothing` instance `s` given below. The left fragment fails to verify the context sets specified above: although called last, method `smooth` can be executed first due to reordering, failing its *succeeds* clause. The middle fragment also fails: The first `smooth` needs not terminate before the next `smooth` activation starts. They may interleave and violate the `overlaps` set of the suspension. The right fragment verifies. We use **`await`** `o!m();` as a shorthand for **`Fut`**`<T> f = o!m();` **`await`** `f?;`.

```
s!setup(c);
s!smooth(l,0.5);
s!smooth(m,0.4);
```

```
await s!setup(c);
s!smooth(l,0.5);
s!smooth(m,0.4);
```

```
await s!setup(c);
await s!smooth(l,0.5);
s!smooth(m,0.4);
```

The client accessing a future might not be its creator, so properties of method parameters and class fields in the postcondition of the method associated to the future should be hidden. The postcondition in the implementation of a method may contain properties of fields, parameters and results upon termination. We abstract that postcondition into a postcondition for the corresponding method at the interface level, which only reads the result at the client side. In analogy to the split of precondition, we name the two types of postcondition *interface postcondition* and *class postcondition*, respectively. Only if the call context is known, the class postcondition may be used in addition to the interface postcondition.

$$
\begin{array}{rlrlrl}
\mathsf{Prgm} ::= & \overline{\mathsf{I}}\ \overline{\mathsf{C}}\ \mathtt{main}\{\mathtt{s}\} & \mathsf{I} ::= & \textbf{interface}\ \mathtt{I}\ \{\overline{\mathsf{S}}\} & \mathsf{C} ::= & \textbf{class}\ \mathtt{C}(\overline{\mathsf{T}\ x})\ \{\overline{\mathsf{M}}\ \overline{\mathsf{T}\ x = \mathtt{e}}\} \\
\mathsf{M} ::= & \mathsf{S}\{\overline{\mathtt{s}};\ \textbf{return}\ \mathtt{e}\} & \mathsf{S} ::= & \mathsf{T}\ \mathtt{m}(\overline{\mathsf{T}\ x}) & rhs ::= & \mathtt{e!m}(\overline{\mathtt{e}})\,|\,\mathtt{e}\,|\,\textbf{new}\ \mathtt{C}(\overline{\mathtt{e}}) \\
\mathtt{s} ::= & \multicolumn{5}{l}{\big[\mathtt{sync} : \text{“}\mathtt{string}\text{”}\big]x = \mathtt{e}.\textbf{get}\ |\ x = rhs\ |\ \big[\mathtt{atom} : \text{“}\mathtt{string}\text{”}\big]\ \textbf{await}\ \ \mathtt{g}} \\
& \multicolumn{5}{l}{|\ \textbf{if}\ (\mathtt{e})\ \{\overline{\mathtt{s}}\}\ \textbf{else}\ \{\overline{\mathtt{s}}\}\ |\ \textbf{while}\ (\mathtt{e})\ \{\overline{\mathtt{s}}\}\ |\ \textbf{skip} \qquad \mathtt{g} ::= \mathtt{e}\,|\,\mathtt{e}? \qquad x = \mathtt{v}\ |\ \textbf{this}.\mathtt{f}}
\end{array}
$$

Fig. 2. Syntax of the Async language.

3 An Active Object Language

Syntax. Consider a simple active object language Async, based on ABS [33]; the syntax is shown in Fig. 2. We explain the language features related to communication and synchronization, other features are standard. Objects communicate with each other by asynchronous method calls, written $\mathtt{e!m}(\overline{\mathtt{e}})$, with an associated future. The value of a future f can be accessed by a statement $x = f.\textbf{get}$ once it is resolved, i.e. when the process associated with f has terminated. Futures can be shared between objects. Field access between different objects is indirect through method calls, amounting to strong encapsulation. Cooperative scheduling is realized in Async as follows: at most one process is active on an object at any time and all scheduling points are *explicit* in the code using **await** statements. The execution between these points is sequential and cannot be preempted.

Objects in Async are active. We assume that all programs are well-typed, that their main block only contains statements of the form $\mathtt{v} = \textbf{new}\ \mathtt{C}(\overline{\mathtt{e}})$, and that each class has a `run()` method which is automatically activated when an instance of the class is generated. Compared to ABS, Async features optional annotations for atomic segments as discussed in Sect. 2. A *synchronize* annotation `sync` associates a label with each assignment which has a **get** right-hand side. We assume all names to be unique in a program.

Observable Behavior. A distributed system can be specified by the externally observable behavior of its parts, and the behavior of each component by the possible communication histories over its observable events [18, 29]. Theoretically this is justified because fully abstract semantics of object-oriented languages are based on communication histories [32]. We strive for *compositional* communication histories of asynchronously communicating systems and use separate events for method invocation, reaction upon a method call, resolving a future, fetching the value of a future, suspending a process, reactivating a process, and for object creation. Note that each of these events is witnessed by *exactly one object*, namely the generating object; different objects do not share events.

Definition 1 (Events).

$$
\begin{aligned}
\mathsf{ev} ::= & \mathsf{invEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \overline{\mathsf{e}})\ |\ \mathsf{invREv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \overline{\mathsf{e}})\ |\ \mathsf{newEv}(\mathsf{X}, \mathsf{X}', \overline{\mathsf{e}})\ |\ \mathsf{noEv} \\
& |\ \mathsf{suspEv}(\mathsf{X}, f, \mathtt{m}, i)\ |\ \mathsf{reacEv}(\mathsf{X}, f, \mathtt{m}, i)\ |\ \mathsf{futEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})\ |\ \mathsf{futREv}(\mathsf{X}, f, \mathsf{e}, i)
\end{aligned}
$$

An invocation event invEv and an invocation reaction event invREv record the caller X, callee X′, generated future f, invoked method `m`, and method parameters

$\overline{\mathsf{e}}$ of a method call and its activation, respectively. A termination event futEv records the callee X, the future f, the executed method $\mathtt{m}$, and the method result e when the method terminates and resolves its associated future. A future reaction event futREv records the current object X, the accessed future f, the value e stored in the future, and the label i of the associated **get** statement. A suspension event suspEv records the current object X, the current future f and method name $\mathtt{m}$ associated to the process being suspended, and the name i of the **await** statement that caused the suspension. Reactivation events reacEv are dual to suspension events, where the future f belongs to the process being reactivated. A new event newEv records the current object X, the created object X' and the object initialization parameters $\overline{\mathsf{e}}$ for object creation. The event noEv is a marker for transitions without communication.

Operational Semantics. The operational semantics of Async is given by a transition relation $\rightarrow_{\mathsf{ev}}$ between configurations, where ev is the event generated by the transition step. We first define configurations and their transition system, before defining terminating runs and traces over this relation. A configuration C contains processes, futures, objects and messages:

$$\mathsf{C} ::= \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma) \mid \mathbf{fut}(f, \mathsf{e}) \mid \mathbf{ob}(\mathsf{X}, f, \rho) \mid \mathbf{msg}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \overline{\mathsf{e}}) \mid \mathsf{C}\ \mathsf{C}$$

In the runtime syntax, a process $\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)$ contains the current object X, the future f that will contain its execution result, the executed method $\mathtt{m}$, statements $\mathtt{s}$ in that method, and a local state σ. A future $\mathbf{fut}(f, \mathsf{e})$ contains the future's identity f and the value e stored by the future. An object $\mathbf{ob}(\mathsf{X}, f, \rho)$ contains the object identity X, the future f associated with the currently executing process, and the heap ρ of the object. Let $\perp$ denote that no process is currently executing at X. A message $\mathbf{msg}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \overline{\mathsf{e}})$ contains the caller object identity X, the callee object identity X', the future identity f, the invoked method $\mathtt{m}$, and the method parameters $\overline{\mathsf{e}}$.

A selection of the transition rules is given in Fig. 3. Function $[\![\mathsf{e}]\!]_{\sigma,\rho}$ evaluates an expression e in the context of a local state σ and an object heap ρ. Rule **async** expresses that the caller of an asynchronous call generates a future with a fresh identifier f' for the result and a method invocation message. An invocation event is generated to record the asynchronous call. Rule **start** represents the start of a method execution, in which an invocation reaction event is generated. The message is removed from the configuration and a new process to handle the call in created. Function M returns the body of a method, and $\widehat{M}$ returns the initial local state of a method by evaluating its parameters. Observe that a process can only start when its associated object is idle. Rule **return** resolves future f with the return value from the method activation. A termination event is generated. Rule **get** models future access. Provided that the accessed future is resolved (i.e., the future occurs in the configuration), its value can be fetched and a future reaction event generated. In this rule x is a local variable and is modified to e'. If the future is not resolved, the rule is not applicable and execution in object X is blocked.

$$\textsf{(async)}\ \frac{f' \text{ is fresh in } \mathsf{C}}{\begin{array}{l}\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(x = \mathsf{e}!\mathtt{m}'(\overline{\mathsf{e}'}); \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C} \rightarrow_{\mathsf{invEv}(\mathsf{X}, \llbracket \mathsf{e} \rrbracket_{\sigma,\rho}, f', \mathtt{m}', \llbracket \overline{\mathsf{e}'} \rrbracket_{\sigma,\rho})} \\ \quad \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma[x := f'])\ \mathbf{msg}(\mathsf{X}, \llbracket \mathsf{e} \rrbracket_{\sigma,\rho}, f', \mathtt{m}', \llbracket \overline{\mathsf{e}'} \rrbracket_{\sigma,\rho})\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}\end{array}}$$

$$\textsf{(start)}\ \begin{array}{l}\mathbf{msg}(\mathsf{X}', \mathsf{X}, f, \mathtt{m}, \overline{\mathsf{e}})\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathsf{C} \rightarrow_{\mathsf{invREv}(\mathsf{X}', \mathsf{X}, f, \mathtt{m}, \overline{\mathsf{e}})} \\ \quad \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(M(\mathtt{m})), \widehat{M}(\mathtt{m}, \overline{\mathsf{e}}))\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}\end{array}$$

$$\textsf{(return)}\ \begin{array}{l}\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\textbf{return}\ \mathsf{e}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C} \rightarrow_{\mathsf{futEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})} \\ \quad \mathbf{fut}(f, \llbracket \mathsf{e} \rrbracket_{\sigma,\rho})\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathsf{C}\end{array}$$

$$\textsf{(get)}\ \begin{array}{l}\mathbf{prc}(\mathsf{X}, f, \mathtt{m}([\mathtt{sync}:\ \text{“}i\text{”}]x = \mathsf{e}.\textbf{get}; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathbf{fut}(\llbracket \mathsf{e} \rrbracket_{\sigma,\rho}, \mathsf{e}')\ \mathsf{C} \\ \quad \rightarrow_{\mathsf{futREv}(\mathsf{X}, \llbracket \mathsf{e} \rrbracket_{\sigma,\rho}, \mathsf{e}', i)}\ \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma[x := \mathsf{e}'])\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathbf{fut}(\llbracket \mathsf{e} \rrbracket_{\sigma,\rho}, \mathsf{e}')\ \mathsf{C}\end{array}$$

Fig. 3. Selected Operational Semantics Rules for Async. Further rules are in [38].

Definition 2 (Big-Step Semantics). *Let* Prgm *be an* Async *program with initial configuration* C_1. *A* run *from* C_1 *to* C_n *is a finite sequence of transitions*

$$\mathsf{C}_1 \rightarrow_{\mathsf{ev}_1} \mathsf{C}_2 \rightarrow_{\mathsf{ev}_2} \dots \rightarrow_{\mathsf{ev}_{n-1}} \mathsf{C}_n.$$

The trace *of the run is the finite sequence* $(\mathsf{ev}_1, \mathsf{C}_1), \dots, (\mathsf{ev}_{n-1}, \mathsf{C}_{n-1}), (\mathsf{noEv}, \mathsf{C}_n)$ *of pairs of events and configurations. Program* Prgm *generates a trace* tr *if there is a run to some configuration with* tr *as the trace, such that the final configuration is terminated, i.e., has no process* ***prc***.

4 Formalizing Method Contracts

To reason about logical constraints, we use *deductive verification* over *dynamic logic* (DL) [27]. It can be thought of as the language of Hoare triples, syntactically closed under logical operators and first-order quantifiers; we base our account on [4]. Assertions about program behavior are expressed in DL by integrating programs and formulas into a single language. The big step semantics of statements $\mathtt{s}$ is captured by the *modality* $[\mathtt{s}]\mathrm{post}$, which is true provided that the formula post holds in any terminating state of $\mathtt{s}$, expressing partial correctness. The reserved program variable *heap* models the heap by mapping field names to their value [4,44]. The variable *heapOld* holds the heap the most recent time the current method was scheduled. DL features symbolic state updates on formulas of the form $\{v := t\}\varphi$, meaning that v has the value of t in φ.

We formalize method contracts in terms of constraints imposed on runs and configurations. Their semantics is given as first-order constraints over traces, with two additional primitives: the term $\mathsf{ev}^{tr}[i]$ is the i-th event in trace tr and the formula $\mathsf{C}^{tr}[i] \models \varphi$ expresses that the i-th configuration in tr is a model for the modality-free DL formula φ. To distinguish DL from first-order logic over traces, we use the term *formula* and variables φ, ψ, $\chi, \dots$ for DL and the term *constraint* and variables α, $\beta, \dots$ for first-order logic over traces.

Definition 3 (Method Contract). *Let* B *be the set of names for all atomic segments and methods in a given program. A contract for a method* $\mathtt{C.m}$ *has the following components:*

Context clauses. *1. A heap precondition* $\varphi_{\mathtt{m}}$ *over field symbols for* $\mathtt{C}$*; 2. a parameter precondition* $\psi_{\mathtt{m}}$ *over formal parameters of* $\mathtt{C.m}$*; 3. a class postcondition* $\chi_{\mathtt{m}}$ *over formal parameters of* $\mathtt{C.m}$*, field symbols for* $\mathtt{C}$*, and the reserved program variable \result; 4. an interface postcondition* $\zeta_{\mathtt{m}}$ *only over the reserved program variable \result. All context clauses may also contain constants and function symbols for fixed theories, such as arithmetic.*

Context sets. *The sets* $\mathsf{succeeds}_{\mathtt{m}}, \mathsf{overlaps}_{\mathtt{m}} \subseteq \mathsf{B}$*.*

Suspension contracts. *For each suspension point* j *in* $\mathtt{m}$*, a suspension contract containing 1. a suspension assumption* φ_j *with the same restrictions as the heap precondition; 2. a suspension assertion* χ_j *with the same restrictions; 3. context sets* $\mathsf{succeeds}_j$*,* $\mathsf{overlaps}_j \subseteq \mathsf{B}$*.*

Each $\mathtt{run}$ method has the contract $\varphi_{\mathtt{run}} = \psi_{\mathtt{run}} = \mathbf{True}$ and $\mathsf{succeeds}_{\mathtt{run}} = \emptyset$. Methods without a specification have the default contract $\varphi_{\mathtt{m}} = \psi_{\mathtt{m}} = \chi_{\mathtt{m}} = \zeta_{\mathtt{m}} = \mathbf{True}$ and $\mathsf{succeeds}_{\mathtt{m}} = \mathsf{overlaps}_{\mathtt{m}} = \mathsf{B}$. As its default contract, the main block can only create objects. A method's entry and exit points are implicit suspension points: the precondition then becomes the suspension assumption of the first atomic segment, and the postcondition becomes the suspension assertion of the last atomic segment. A suspension point may end in several atomic segments.

Contracts as Constraints. Let $\mathcal{M}_{\mathtt{m}}$ be the method contract for $\mathtt{m}$. The semantics of $\mathcal{M}_{\mathtt{m}}$ consists of three constraints over traces (formalized in Defs. 4 and 5 below): (i) $\mathsf{assert}(\mathcal{M}_{\mathtt{m}}, tr)$ expresses that the postcondition and all suspension assertions hold in tr; (ii) $\mathsf{assume}(\mathcal{M}_{\mathtt{m}}, tr)$ that the precondition and all suspension assumptions hold in tr; (iii) $\mathsf{context}(\mathcal{M}_{\mathtt{m}}, tr)$ that context sets describe the behavior of the object in tr. If the method name is clear from the context, we write $\mathcal{M}$ instead of $\mathcal{M}_{\mathtt{m}}$. In the constraints, all unbound symbols are implicitly universally quantified, such as f, e, X, etc.

Definition 4 (Semantics of Context Clauses). *Let* $\mathcal{M}_{\mathtt{m}}$ *be a method contract,* tr *a trace, and* $\mathsf{susp}(\mathtt{m})$ *the set of suspension points in* $\mathtt{m}$*:*

$$\begin{aligned}
\mathsf{assert}(\mathcal{M}_{\mathtt{m}}, tr) \;=\;& \forall i \in \mathbb{N}.\ \mathsf{ev}^{tr}[i] \doteq \mathsf{futEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e}) \rightarrow \mathsf{C}^{tr}[i] \models \chi_{\mathtt{m}} \wedge \zeta_{\mathtt{m}}\\
& \wedge\ \forall j \in \mathsf{susp}(\mathtt{m}).\ \forall i \in \mathbb{N}.\ \mathsf{ev}^{tr}[i] \doteq \mathsf{suspEv}(\mathsf{X}, f, \mathtt{m}, j) \rightarrow \mathsf{C}^{tr}[i] \models \chi_j\\
\mathsf{assume}(\mathcal{M}_{\mathtt{m}}, tr) \;=\;& \forall i \in \mathbb{N}.\ \mathsf{ev}^{tr}[i] \doteq \mathsf{invREv}(\mathsf{X}', \mathsf{X}, f, \mathtt{m}, \overline{\mathsf{e}}) \rightarrow \mathsf{C}^{tr}[i] \models \varphi_{\mathtt{m}} \wedge \psi_{\mathtt{m}}\\
& \wedge\ \forall j \in \mathsf{susp}(\mathtt{m}).\ \forall i \in \mathbb{N}.\ \mathsf{ev}^{tr}[i] \doteq \mathsf{reacEv}(\mathsf{X}, f, \mathtt{m}, j) \rightarrow \mathsf{C}^{tr}[i] \models \varphi_j
\end{aligned}$$

The third constraint $\mathsf{context}$ models context sets and is defined for both method and suspension contracts. In contrast to context clauses, it constrains the order of events belonging to different processes. The constraint $\mathsf{context}(\mathcal{S}_n, tr)$ formalizes the context sets of a suspension contract $\mathcal{S}_n$ for suspension point n: Before a reactivation event at position i in tr, there is a terminating event at a position $k < i$ on the same object from the *succeeds* set, such that all terminating events on the object at positions k' with $k < k' < i$ are from the *overlaps* set.

Definition 5 (Semantics of Context Sets). *Let $\mathcal{S}_n$ be a suspension contract, tr a trace, and $\mathsf{termEvent}(i)$ the terminating event of i, where i may be either a method name or the name of a suspension point. The predicate $\mathsf{isClose}(\mathsf{ev}^{tr}[i])$ holds if $\mathsf{ev}^{tr}[i]$ is a suspension or future event. The semantics of context sets of a suspension contract $\mathcal{S}_n$ is defined by the following constraint $\mathsf{context}(\mathcal{S}_n, tr)$:*

$$\begin{aligned}&\forall i, i' \in \mathbb{N}.\ \big(\mathsf{ev}^{tr}[i] \doteq \mathsf{reacEv}(\mathsf{X}, f, \mathtt{m}, n) \wedge \mathsf{ev}^{tr}[i'] \doteq \mathsf{suspEv}(\mathsf{X}, f, \mathtt{m}, n)\big) \rightarrow\\ &\quad \exists k \in \mathbb{N}.\ i' < k < i \wedge \Big(\bigvee_{j' \in \mathsf{succeeds}_n} \mathsf{ev}^{tr}[k] \doteq \mathsf{termEvent}(j') \wedge\\ &\quad \forall k' \in \mathbb{N}.\ k < k' < i \wedge \mathsf{isClose}(\mathsf{ev}^{tr}[k']) \rightarrow \big(\bigvee_{j' \in \mathsf{overlaps}_n} \mathsf{ev}^{tr}[k'] \doteq \mathsf{termEvent}(j')\big)\Big)\end{aligned}$$

The predicate $\mathsf{context}(\mathcal{M}_{\mathtt{m}}, tr)$ for method contracts is defined similarly, but includes an extra conjunction of the $\mathsf{context}(\mathcal{S}_n, tr)$ constraints for all $\mathcal{S}_n$ in $\mathcal{M}_{\mathtt{m}}$.

Context sets describe behavior required from other methods, so method contracts are not independent of each other. Each referenced method or method in a context set must have a contract which proves the precondition (or suspension assumption). Recall that method names are names for the last atomic segment, φ_i is the heap precondition/suspension assumption of atomic segment i and χ_i is its postcondition/suspension assertion. The following definition formalizes the intuition we gave about the interplay of context sets, i.e. that the atomic segments in the $\mathsf{succeeds}$ set establish a precondition/suspension assumption and the atomic segments in $\mathsf{overlaps}$ preserve a precondition/suspension assumption.

Definition 6 (Coherence). *Let $\mathsf{CNF}(\varphi)$ be the conjunctive normal form of φ, such that all function and relation symbols also adhere to some theory specific normal form. Let M be a set of method contracts. M is* coherent *if for each method and suspension contract $\mathcal{S}_i$ in M, the following holds:*

- *The assertion χ_j of each atomic segment j in $\mathsf{succeeds}_i$ guarantees assumption φ_i: Each conjunct of $\mathsf{CNF}(\varphi_i)$ is a conjunct of $\mathsf{CNF}(\chi_j)$*
- *Each atomic segment j in $\mathsf{overlaps}_i$ preserves suspension assumption φ_i: suspension assertion χ_j has the form $\chi'_j \wedge \big((\{heap := heapOld\}\varphi_i) \rightarrow \varphi_i\big)$.*

A program is coherent if the set of all its method contracts is coherent.

This notion of coherence is easy to enforce and to check syntactically.

Lemma 1 (Sound Propagation). *Given a non-coherent set of method contracts M, a coherent set $\widehat{M}$ can be generated from M, such that for every contract $\mathcal{M} \in M$ there is a $\widehat{\mathcal{M}} \in \widehat{M}$ with identical context sets and*

$$\forall tr. \big(\mathsf{assert}(\widehat{\mathcal{M}}, tr) \rightarrow \mathsf{assert}(\mathcal{M}, tr)\big) \wedge \big(\mathsf{assume}(\widehat{\mathcal{M}}, tr) \leftrightarrow \mathsf{assume}(\mathcal{M}, tr)\big)$$

The requirement for $\widehat{M}$ ensures that the new, coherent contracts extend the old contracts. In the border case where all context sets contain all blocks, all heap preconditions and suspension assumptions become invariants.

5 Verification

Method contracts appear in comments before their interface and class declaration, following JML [39]. Our specifications use DL formulas directly, extended with a *\last* operator referring to the evaluation of a formula in the state where the current method was last scheduled, i.e. the most recent reactivation/method start. Restrictions on the occurrence of fields and parameters are as above.

Definition 7. *Let* `str` *range over strings,* φ *over DL formulas. The clauses used for specification are defined as follows:*

$$\begin{array}{ll}\mathsf{Spec} ::= \mathit{/{*}@}\ \mathsf{Require}\ \mathsf{Ensure}\ \mathsf{Runs}\ \mathit{@{*}/} & \psi ::= \varphi \mid \mathit{\backslash last}(\varphi)\\ \mathsf{Require} ::= \mathit{requires}\ \psi; \quad \mathsf{Ensure} ::= \mathit{ensures}\ \psi; & \mathsf{Runs} ::= \mathit{succeeds}\ \overline{\mathtt{str}};\ \ \mathit{overlaps}\ \overline{\mathtt{str}};\end{array}$$

We do not consider loop invariants here which are standard. For ghost fields and ghost assignments, we follow JML [39].

As described above, our program logic for deductive verification is a dynamic logic based on the work of Din at al. [18]. The verification of context sets is *not* part of the program logic: our soundness theorem requires that the context sets are adhered to, in addition to proving the DL proof obligations. Context sets, however, can be verified with light-weight causality-based approaches, such as May-Happen-in-Parallel analysis [6]. Separating the DL proof obligation from the causual structure allows us to give a relatively simple proof system and reuse existing techniques to verify the context sets.

The DL calculus rewrites a formula $[\mathtt{s;r}]\mathrm{post}$ with a leading statement `s` into the formula $[\mathtt{r}]\mathrm{post}$ with suitable first-order constraints. Repeated rule application yields symbolic execution of the program in the modality. *Updates* (see Sect. 4) accumulate during symbolic execution to capture state changes; e.g., $[\mathtt{v\ =\ e;\ r}]\mathrm{post}$ is rewritten to $\{\mathtt{v} := \mathtt{e}\}[\mathtt{r}]\mathrm{post}$, expressing that `v` has the value of `e` during the symbolic execution of `r`. When a program `s` has been completely executed, the modality is empty and the accumulated updates are applied to the postcondition post, resulting in a pure first-order formula that represents the weakest precondition of `s` and post. We use a sequent calculus to prove validity of DL formulas [4, 17]. In sequent notation $\mathrm{pre} \rightarrow [\mathtt{s}]\mathrm{post}$ is written as $\Gamma, \mathrm{pre} \Longrightarrow [\mathtt{s}]\mathrm{post}, \Delta$, where Γ and Δ are (possibly empty) sets of side formulas. A formal proof is a tree of proof rule applications leading from axioms to a formula (a theorem). The formal semantics is described in [38].

We formulate DL proof obligations for the correctness of method contracts, given a method `m` with body `s` and contract $\mathcal{M}_{\mathtt{m}}$ as in Def. 3, as follows:

$$\begin{aligned}\varphi_{\mathtt{m}}, \psi_{\mathtt{m}}, \mathsf{wellFormed}(\mathit{trace}) \Longrightarrow \{\mathit{heapOld} := \mathit{heap}\}&\\ \{\mathbb{t} := \mathit{trace}\}\{\mathsf{this} := o\}\{\mathbb{f} := f\}\{\mathbb{m} := \mathtt{m}\}[\mathtt{s}]\widetilde{\chi_{\mathtt{m}}}&\end{aligned} \qquad \text{(PO)}$$

The heap and parameter preconditions $\varphi_{\mathtt{m}}$ and $\psi_{\mathtt{m}}$ of $\mathcal{M}_{\mathtt{m}}$ are assumed when execution starts, likewise it is assumed that the trace of the object up to now is well-formed. The class postcondition $\widetilde{\chi_{\mathtt{m}}}$ is modified, because *\last* is part of the specification language, but not of the logic: Any heap access in the argument of

$$\textbf{(local)}\ \frac{\Longrightarrow \{U\}\{\mathtt{v\ :=\ e}\}[\mathtt{s}]\chi}{\Longrightarrow \{U\}[\mathtt{v\ =\ e;s}]\chi} \qquad \textbf{(field)}\ \frac{\Longrightarrow \{U\}\{\mathit{heap} := \mathit{store}(\mathit{heap}, \mathtt{f}, \mathtt{e})\}[\mathtt{s}]\chi}{\Longrightarrow \{U\}[\textbf{this}\mathtt{.f\ =\ e;s}]\chi}$$

$$\textbf{(async)}\ \frac{\begin{array}{l}\Longrightarrow \{U\}\psi_{\mathtt{m}}(\bar{\mathtt{e}}) \\ \mathit{fresh}(\mathtt{f}, \mathbb{t}) \Longrightarrow \{U\}\{\mathtt{v:=f}\}\{\mathbb{t} := \mathbb{t} \cdot \mathsf{invEv}(\mathsf{this}, \mathsf{o}, \mathtt{f}, \mathtt{m}, \bar{\mathtt{e}})\}[\mathtt{s}]\chi\end{array}}{\Longrightarrow \{U\}[\mathtt{v\ =\ o!m(}\bar{\mathtt{e}}\mathtt{);s}]\chi}$$

$$\textbf{(get-}\mathtt{m}\textbf{)}\ \frac{\begin{array}{c}\mathit{fresh}(\mathtt{r}, \mathbb{t}),\ \{U\}\left(\exists\, \mathtt{Int\ \ j};\ \mathsf{invocOn}(\mathbb{t}[j], \mathtt{f}, \mathtt{m}) \to \zeta_{\mathtt{m}}(\mathtt{r})\right) \Longrightarrow \\ \{U\}\{\mathtt{v:=r}\}\{\mathbb{t} := \mathbb{t} \cdot \mathsf{futREv}(\mathsf{this}, \mathtt{f}, \mathtt{r}, i)\}[\mathtt{s}]\chi\end{array}}{\Longrightarrow \{U\}[\mathtt{[sync:\ "i"]\ \ v\ =\ f.}\textbf{get}\mathtt{;s}]\chi}$$

$$\textbf{(await)}\ \frac{\begin{array}{l}\Longrightarrow \{U\}\{\mathbb{t} := \mathbb{t} \cdot \mathsf{suspEv}(\mathsf{this}, \mathbb{f}, \mathbb{m}, i)\}\chi_i \\ \mathit{fresh}(t, \mathbb{t}) \Longrightarrow \{U\}\{\mathbb{t} := \mathbb{t} \cdot \mathsf{suspEv}(\mathsf{this}, \mathbb{f}, \mathbb{m}, i)\}\{\mathit{heapOld} := \mathit{heap}\} \\ \qquad\quad \{\mathit{heap} := \mathit{heap}_A\}\{\mathbb{t} := \mathbb{t} \cdot t \cdot \mathsf{reacEv}(\mathsf{this}, \mathbb{f}, \mathbb{m}, i)\}(\varphi_i \to [\mathtt{s}]\chi)\end{array}}{\Longrightarrow \{U\}[\mathtt{[atom:\ "i"]}\ \textbf{await}\ \mathtt{f?;s}]\chi}$$

Fig. 4. Selected DL proof rules.

$\backslash\mathit{last}$ is replaced by *heapOld*. Reserved variables $\mathbb{t}$, this, $\mathbb{f}$, and $\mathbb{m}$ record the current trace, object, future, and method, respectively, during symbolic execution.

The above proof obligation must be proved for each method of a program using schematic proof rules as shown in Fig. 4. There is one rule for each kind of Async statement. We omit the standard rules for sequential statements. To improve readability, we leave out the sequent contexts Γ, Δ and assume that all formulas are evaluated relative to a current update U representing all symbolic updates of local variables, the heap, as well as $\mathbb{t}$, this, $\mathbb{f}$, $\mathbb{m}$ up to this point. These updates are extended in the premisses of some rules.

Rule **local** captures updates of local variables by side-effect free expressions. Rule **field** captures updates of class fields by side-effect free expressions. It is nearly identical to **local**, except the heap is updated with the *store* function. This function follows the usual definition from the theory of arrays to model heaps in dynamic logics [44]. Rule **async** for assignments with an asynchronous method call has two premisses. The first establishes the parameter precondition $\psi_{\mathtt{m}}$ of $\mathcal{M}_{\mathtt{m}}$. The second creates a fresh future `f` relative to the current trace $\mathbb{t}$ to hold the result of the call. In the succedent an invocation event recording the call is generated and symbolic execution continues uninterrupted. We stress that the called method is syntactically known.

For each method `m` we define a rule **get**-`m`. It creates a fresh constant `r` representing the value stored in future `f`. Per se, nothing about this value is known. However, the term in the antecedent of the premise expresses that *if* it is possible to show that the future stored in `f` stems from a call on `m`, then the postcondition of `m` can be assumed to show `r`. The predicate $\mathsf{invocOn}(\mathsf{ev}, f, \mathtt{m})$ holds if the event ev is an invocation reaction event with future f on method `m`.

Rule **await** handles process suspension. The first premise proves the postcondition χ_i of the suspension contract $\mathcal{S}_i$ in the current trace, extended by a suspension event. When resuming execution we can only use the suspension assumption φ_i of $\mathcal{S}_i$; the remaining heap must be reset by an “anonymizing update’ $heap_A$ [4,44], which is a fresh function symbol. Also a reaction event is generated. In both events $\mathbb{f}$ is not the future in the **await** statement, but the currently computed future that is suspended and reactivated.

Theorem 1 (Soundness of Compositional Reasoning). *Let $\widehat{M}$ be the coherent set generated from the method contracts M of a program* Prgm*. If*

(i) $\mathsf{context}(\mathcal{M}_{\mathtt{m}}, tr)$ *holds for all methods and generated traces, and*

(ii) for each $\mathcal{M}_{\mathtt{m}} \in \widehat{M}$, the proof obligation (PO) *for* $\mathtt{m}$ *holds,*

then the following holds for all terminating traces tr of Prgm*:*

$$\bigwedge_{\mathcal{M}_{\mathtt{m}} \in \widehat{M}} \big(\mathsf{assert}(\mathcal{M}_{\mathtt{m}}, tr) \wedge \mathsf{assume}(\mathcal{M}_{\mathtt{m}}, tr)\big)$$

6 Related Work and Conclusion

Related Work. Wait conditions were introduced as program statements (not in method contracts) in the pioneering work of Brinch-Hansen [25,26] and Hoare [28]. SCOOP [8] explores preconditions as wait/when conditions. Previous approaches to AO verification [16,18] consider only object invariants that must be preserved by every atomic segment of every method. As discussed, this is a special case of our system. Actor services [43] are compositional event patterns for modular reasoning about asynchronous message passing for actors. They are formulated for pure actors and do not address futures or cooperative scheduling. Method preconditions are restricted to input values, the heap is specified by an object invariant. A rely-guarantee proof system [1,34] implemented on top of Frama-C by Gavran et al. [22] demonstrated modular proofs of partial correctness for asynchronous C programs restricted to using the Libevent library.

Contracts for channel-based communication are partly supported by session types [11,30]. These have been adapted to the AO concurrency model [37], including assertions on heap memory [36], but they require composition to be explicit in the specification. Stateful session types for active objects [36] contain a propagation step (cf. Sect. 2.1): Postconditions are propagated to preconditions of methods that are specified to run subsequently. In contrast, the propagation in the current paper goes in the opposite direction, where a contract specifies what a method *relies* on and one propagates to the method that is obliged to prove it. Session types, with their global system view, specify an *obligation* for a method and propagate to the methods that can rely on it.

Compositional specification of concurrency models outside rely-guarantee was mainly proposed based on separation logic [12,42]. Closest to our line of research are shared regions [20] which relate predicates over the heap that must be stable, i.e. invariant, when accessed. Even though approaches to specify regions precisely have been developed [14,20], their combination with interaction modes beyond heap access (such as asynchronous calls and futures) is not well explored. It is worth noting that AO do not require the concept of regions in the logic, because strong encapsulation and cooperative scheduling ensure that two threads never run in parallel on the same heap. The central goal of separation *logic*—separation of heaps—is a design feature of the AO *concurrency model.*

Conclusion. This paper generalizes rely-guarantee reasoning with method contracts from sequential OO programs to active objects with asynchronous method calls and cooperative scheduling. The main challenges are: the delay between the invocation and the actual start of a method, strong object encapsulation, and interleaving of atomic segments via cooperative scheduling. To deal with these issues, preconditions of contracts are separated into a caller specification (parameter precondition) and a callee specification (heap precondition); likewise, into an interface postcondition and a class postcondition. The heap precondition and the class postcondition can be stronger than a class invariant, because they do not need to be respected by all methods of a class. Instead, context sets specify those methods that establish or preserve the heap precondition. The context sets are justified separately via a global analysis of possible call sequences. This separation of concerns enables class-modular verification. Preconditions need not contain global information, rather, this is automatically propagated within a class with the help of external global analyses.

Future Work. In this paper, we did not consider all features present in synchronous method contracts, such as termination witnesses [23], and it is unclear how these can be used in an asynchronous setting due to interleaving. Other contract extensions, such as exceptional behavior [4], are largely orthogonal to concurrency and could be easily added. Furthermore, we plan to explore recursion: In this case, specifications working with program-point identifiers, i.e. at the statement-level, are not precise enough, because they cannot distinguish between multiple processes of the same method.

Beyond implementation and addition of features from synchronous method contracts, we plan to connect cooperative contracts with our work on session types [36] with the aim to integrate local and global specifications by formulating them in the framework of Behavioral Program Logic [35]. We also expect such a formalization to enable runtime verification.

Acknowledgments This work is supported by the SIRIUS Centre for Scalable Data Access and the FormbaR project, part of AG Signalling/DB RailLab in the Innovation Alliance of Deutsche Bahn AG and TU Darmstadt.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, 1995.
2. ABS Development Team. *The ABS Language Specification*, Jan. 2018. http://docs.abs-models.org/.

3. 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.
4. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *LNCS*. Springer, 2016.
5. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications*, 8(4):323–339, Dec. 2014.
6. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. May-Happen-in-Parallel Analysis for Actor-based Concurrency. *ACM Trans. Comput. Log.*, 17(2):11:1–11:39, 2016.
7. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2007.
8. V. Arslan, P. Eugster, P. Nienaltowski, and S. Vaucouleur. SCOOP - Concurrency made easy. In *Dependable Systems: Software, Computing, Networks, Research Results of the DICS Program*, pages 82–102, 2006.
9. H. G. Baker and C. E. Hewitt. The incremental garbage collection of processes. In *Proceeding of the Symposium on Artificial Intelligence Programming Languages*, number 12 in SIGPLAN Notices, page 11, August 1977.
10. C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Lessons learned from microkernel verification – specification is the new bottleneck. In F. Cassez, R. Huuck, G. Klein, and B. Schlich, editors, *Proc. 7th Conference on Systems Software Verification*, volume 102 of *EPTCS*, pages 18–32, 2012.
11. L. Bocchi, J. Lange, and E. Tuosto. Three algorithms and a methodology for amending contracts for choreographies. *Sci. Ann. Comp. Sci.*, 22(1):61–104, 2012.
12. S. Brookes and P. W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, Aug. 2016.
13. D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 123–134. ACM Press, 2004.
14. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Tada: A logic for time and data abstraction. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 207–231. Springer Berlin Heidelberg, 2014.
15. F. de Boer, C. C. Din, K. Fernandez-Reyes, R. Hähnle, L. Henrio, E. B. Johnsen, E. Khamespanah, J. Rochas, V. Serbanescu, M. Sirjani, and A. M. Yang. A survey of active object languages. *ACM Computing Surveys*, 50(5):76:1–76:39, Oct. 2017.
16. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.
17. C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In A. P. Felty and A. Middeldorp, editors, *Proceedings of the 25th International Conference on Automated Deduction (CADE 2015)*, volume 9195 of *LNCS*, pages 517–526. Springer, 2015.
18. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
19. C. C. Din, S. L. Tapia Tarifa, R. Hähnle, and E. B. Johnsen. History-based specification and verification of scalable concurrent and distributed systems. In M. Butler,

S. Conchon, and F. Zaïdi, editors, *Proc. 17th International Conference on Formal Engineering Methods (ICFEM 2015)*, volume 9407 of *LNCS*, pages 217–233. Springer, 2015.

20. T. Dinsdale-Young, P. da Rocha Pinto, and P. Gardner. A perspective on specifying and verifying concurrent modules. *Journal of Logical and Algebraic Methods in Programming*, 98:1 – 25, 2018.
21. C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.
22. I. Gavran, F. Niksic, A. Kanade, R. Majumdar, and V. Vafeiadis. Rely/Guarantee Reasoning for Asynchronous Programs. In L. Aceto and D. de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 483–496. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.
23. D. Grahl, R. Bubel, W. Mostowski, P. H. Schmitt, M. Ulbrich, and B. Weiß. Modular specification and verification. In Ahrendt et al. [4], pages 289–351.
24. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.
25. P. B. Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, 1972.
26. P. B. Hansen. *Operating System Principles.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.
27. D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. *SIGACT News*, 32(1):66–69, 2001.
28. C. A. R. Hoare. Towards a theory of parallel programming. *Operating System Techniques*, pages 61–71, 1972.
29. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., 1985.
30. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284, 2008.
31. M. Huisman, W. Ahrendt, D. Grahl, and M. Hentschel. Formal specification with the Java Modeling Language. In Ahrendt et al. [4], pages 193–241.
32. A. Jeffrey and J. Rathke. Java jr: Fully abstract trace semantics for a core Java language. In *ESOP*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.
33. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
34. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, Oct. 1983.
35. E. Kamburjan. Behavioral program logic. In S. Cerrito and A. Popescu, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, 28th Intl. Conf., TABLEAUX, London, UK*, LNCS. Springer, to appear, 2019. Technical report available under https://arxiv.org/abs/1904.13338.
36. E. Kamburjan and T. Chen. Stateful behavioral types for active objects. In C. A. Furia and K. Winter, editors, *Integrated Formal Methods, 14th Intl. Conf. IFM, Maynooth, Ireland*, volume 11023 of *LNCS*, pages 214–235. Springer, 2018.
37. E. Kamburjan, C. C. Din, and T. Chen. Session-based compositional analysis for actor-based languages using futures. In K. Ogata, M. Lawford, and S. Liu, editors, *Formal Methods and Software Engineering, 18th Intl. Conf. on Formal Engineering Methods, ICFEM, Tokyo, Japan*, volume 10009 of *LNCS*, pages 296–312, 2016.

38. E. Kamburjan, C. C. Din, R. Hähnle, and E. B. Johnsen. Asynchronous cooperative contracts for cooperative scheduling. Technical report, TU Darmstadt, 2019. http://formbar.raillab.de/en/techreportcontract/.
39. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. *JML Reference Manual*, May 2013. Draft revision 2344.
40. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.
41. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.
42. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, UK, 2001. Springer.
43. A. J. Summers and P. Müller. Actor services - modular verification of message passing programs. In P. Thiemann, editor, *Proceedings of the 25th European Symposium on Programming (ESOP 2016)*, volume 9632 of *LNCS*, pages 699–726. Springer, 2016.
44. B. Weiß. *Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.
45. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, Nov. 1986.