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* [47] 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 being 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 [20, 23]. However, this restricted concurrency forces systems to be modeled

⋆ 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. Proofs can be found in our accompanying report [42].

and specified at a high level of abstraction, essentially as protocols. It precludes 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 [19] 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, 16, 20, 25, 29, 46, 52]). In this context, ABS [38] 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. The active object setting 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 do so 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: 1. A formal *specification*-by-contract technique for methods in a *concurrency context* with asynchronous calls, futures, and cooperative scheduling. 2. 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 [45].

We demonstrate by example that the proposed contracts allow complex concurrent behavior to be specified in a succinct and intelligible manner.

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 mailboxes, i.e. futures. They are closely tied to the OO programming paradigm and its programming abstractions. We go through an example implemented in the ABS language [2, 38], an AO language with cooperative scheduling that was used to model complex, industrial concurrent systems [5].

2.1 Description of the Example

We use a distributed computation of moving averages, a common task in data analysis that renders long-term trends clearer in smoothened data. Given data points $x_1, \ldots, x_n$, many forms of moving average $\mathsf{avg}(x_1, \ldots, x_n)$ 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:

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

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

Fig. 1 shows the central class `Smoothing`. Each `Smoothing` instance holds a `Computation` instance `comp` in `c`, where 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 a number of useful properties one would like to specify for `smooth`: 1. `c` has been assigned before it is called and is not changed during its execution. 2. No two executions of `smooth` overlap during suspension. 3. The returned result is a smoothened version of the `input`.

We explain some specification elements. We assign unique names to each *atomic segment* of statements between suspension points. They are labeled with the *annotation* `[atom: "string"]` at an await statement. The named scope "`str`" 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 also `sync`: labels at future reads. We use a ghost field [36] `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

2.2 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 break up the precondition of asynchronous method contracts into one part for parameters and separate one for the heap. The *parameter precondition* is guaranteed by the *caller* who knows the appropriate synchronization pattern. It is part of the interface declaration of the callee and exposed to clients. (Without parameters, the parameter precondition is `true`.) The *heap precondition* is guaranteed by the callee. It is declared in an implementing class and not exposed in the interface.

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* over atomic segment names:

- 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.
- 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 to add private methods. Observe that context sets represent *global information* unavailable when a method is analyzed in isolation. Undeclared context sets default to *all* atomic segments, whence the heap precondition degenerates into a class invariant and must be guaranteed by each process at each suspension point [22]. 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 [35]. 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 contexts and global protocols expressed with session types [40,41] must be proven consistent. Context sets can also be verified by static analysis once the whole program is available (see Sect. 2.5).

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.

```
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 context sets of the extended interface are implicitly included in 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.

2.3 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, which requires similar reasoning as for heap preconditions: What is guaranteed at 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 with 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 has to hold 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 continuation, the object is still locked and the `Computation` instance `c` must be present. During suspension, only method `getCounter` is allowed to run. By adding itself to the `succeeds` set, it is ensured that the suspension has to establish its own suspension assumption. The specification after *propagation* is shown below right. (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() { ... }
```

2.4 Specifying the Postcondition

In contrast to synchronous method contracts, the postcondition that specifies the return value need not be used immediately after a method terminates, but only when accessing the associated future. This future may be passed around, so the client accessing a future might not even know which method resolved it. To provide this context we use a *resolve contract*, a set containing those methods that can resolve a future. During the generation of proof obligations (after propagation), the postconditions of the resolving methods can be retrieved.

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. We extract that postcondition only to read the result at the client side, and define this postcondition in the interface of the corresponding method. The postcondition in the implementation of a method may contain properties of fields, parameters and results upon termination. 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.

Concerning inheritance, we follow the well-known subtyping rules for synchronous contracts in languages like Java [4].

Example 5. Consider the following specification of a computation server. It guarantees that if postive data is provided as input, then the output is positive.

The implementation specifies that the return value bounded by the two input values. This is encapsulated in the class—the process reading a future associated with this method has no access to the input. In our example, the call parameters are know so we are able to specify as follows and use the full information.

```
/*@ resolvedBy {TBound.compute} @*/
[sync: "sync"] Rat res = f.get;
```

2.5 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 requires only 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 [21] is used to show that *locally* single methods implement their pre- and postcondition 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.

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

Fig. 2. Syntax of the Async language.

Context sets can be verified by static analysis once the whole program is available. Such analyses build a graph on top of the call and control flow graphs of a program, so it is not modular. However, the enriched contracts are modular.

Example 6. Consider different code fragments interacting with a `Smoothing` instance `s`. 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 process for `smooth` 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);
```

Resolve contracts can be verified statically by a points-to analysis. They are natural when reasoning about futures, because at every synchronization point one must establish what is being synchronized, not just properties of the future value.

3 An Active Object Language

Syntax. Async is a simple active object language, based on ABS [38]; 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 $\mathsf{e}!\mathtt{m}(\overline{\mathsf{e}})$, with an associated future. The value of a future f can be accessed by a statement $x = f.\mathbf{get}$ 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. 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 $\mathtt{v} = \mathbf{new}\ \mathtt{C}(\overline{\mathsf{e}})$, 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 [22, 34]. Theoretically this is justified because fully abstract semantics of object-oriented languages are based on communication histories [37]. 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).

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

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 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 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 m, statements 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 are 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. Rule **(await)** releases control and generates a suspension event. Rule **(react-expr)** generates a reactivation event for an idle object and a satisfied Boolean guard. Rule **(react-fut)** similarly generates a reactivation event for an idle object and a satisfied future query guard, reflecting that the future has been resolved.

Definition 2 (Runs and Traces). *Let* $\mathsf{C}_1, \ldots, \mathsf{C}_n$ *be configurations and let* $\mathsf{ev}_1, \ldots, \mathsf{ev}_{n-1}$ *be events. A* run *from* C_1 *to* C_n *is a finite sequence of transitions*

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

Given a run, its trace *is the finite sequence* $(\mathsf{ev}_1, \mathsf{C}_1), \ldots, (\mathsf{ev}_{n-1}, \mathsf{C}_{n-1}), (\mathsf{noEv}, \mathsf{C}_n)$ *of pairs of events and configurations.*

A configuration C is *terminating* if every process in C has terminated, i.e., all **prc** have been removed from the configuration due to method termination.

Definition 3 (Big-Step Semantics and Partial Traces). *An* Async *program* Prgm *generates a trace tr, written* $\mathsf{Prgm} \Downarrow tr$*, if and only if there is a run from its initial configuration to a terminating configuration with tr as the trace of this run. A trace tr′ of* Prgm *is* partial *if it is a prefix of a trace tr with* $\mathsf{Prgm} \Downarrow tr$.

We use two program analysis techniques: *deductive verification and static analysis.*[3] Before formalizing their role, we give an informal account.

[3] Obviously, deductive verification is static as well, however, it is common [11] to distinguish between *deductive verification*, a heavy-weight analysis based on expressive program logics, and *static analyses*, light-weight static inference systems, based on data flow analysis, variable dependence, etc.

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

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

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

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

$$\textsf{(await)}\ \begin{array}{l}\mathbf{prc}(\mathsf{X}, f, \mathtt{m}([\mathtt{atom}:\ \text{“}i\text{”}]\ \textbf{await}\ \ \mathsf{g}; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C} \rightarrow_{\mathsf{suspEv}(\mathsf{X}, f, \mathtt{m}, i)} \\ \quad \mathbf{prc}(\mathsf{X}, f, \mathtt{m}([\mathtt{atom}:\ \text{“}i\text{”}]\ \textbf{await}\ \ \mathsf{g}; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathsf{C}\end{array}$$

$$\textsf{(react-expr)}\ \frac{\llbracket \mathsf{e} \rrbracket_{\sigma,\rho} = \mathtt{True}}{\begin{array}{l}\mathbf{prc}(\mathsf{X}, f, \mathtt{m}([\mathtt{atom}:\ \text{“}i\text{”}]\textbf{await}\ \ \mathsf{e}; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathsf{C} \rightarrow_{\mathsf{reacEv}(\mathsf{X}, f, \mathtt{m}, i)} \\ \quad \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}\end{array}}$$

$$\textsf{(react-fut)}\ \begin{array}{l}\mathbf{prc}(\mathsf{X}, f, \mathtt{m}([\mathtt{atom}:\ \text{“}i\text{”}]\ \textbf{await}\ \ \mathsf{e}?; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathbf{fut}(\llbracket \mathsf{e} \rrbracket_{\sigma,\rho}, \mathsf{e}')\ \mathsf{C} \\ \quad \rightarrow_{\mathsf{reacEv}(\mathsf{X}, f, \mathtt{m}, i)}\ \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathbf{fut}(\llbracket \mathsf{e} \rrbracket_{\sigma,\rho}, \mathsf{e}')\ \mathsf{C}\end{array}$$

Fig. 3. Selected Operational Semantics Rules for Async.

Deductive Verification. Enriched method contracts comprise *logical constraints* (pre- and postconditions) and *behavioral constraints* (context sets and resolve contracts). Semantically, these are treated uniformly as *constraints* over execution traces. Their analysis differs considerably, however: for pre- and postconditions we use a program logic that follows the successful contract-based deductive verification paradigm for sequential object-oriented programs [4] (later extended to invariant reasoning on active objects [22]). Decomposing global system specifications into method contracts is crucial for the scalability of deductive verification, as it allows large programs to be verified by reasoning about one method at a time. In contrast, behavioral constraints pertain by their very nature to the *complete* system. To reason about these in the program logic would render deductive verification of active object programs non-modular, hence, non-scalable. Fortunately, the information needed in context sets and resolve contracts can be obtained efficiently, and with sufficient precision, using lightweight static analysis techniques on the global program to be verified.

Static Analysis. May-Happen-In-Parallel (MHP) analysis [6] identifies statements that may possibly run in parallel or can be interleaved. The analysis produces all pairs (i, j) of named program points with the following semantics: If two processes on the *same* object have their active statements at i and j, respectively, then their order of execution is determined by the (non-deterministic) scheduler. If two processes are active on two *different* objects, they may run in parallel at statements i and j of the active objects. MHP analysis also computes *Must-Have-Happened* (MHH) pairs. The analysis produces all pairs (i, j) such that if a process has j as its active statement, then statement i must have been executed already ("i before j"). Points-To (PT) analysis for futures [26] takes as argument a field or variable of future type and returns the set of methods that may resolve futures stored at this location. The MHP, MHH, and PT relations are all undecidable. The algorithms from [6] safely over-approximate. MHP may have false positives, but no false negatives: the analysis returns a superset of the true set of MHP pairs. The same holds for PT. MHH may have false negatives, but no false positives: the analysis returns a subset of the true set of MHH pairs.

4 Formalizing Method Contracts

To reason about logical constraints, we use *deductive verification* over *dynamic logic* (DL) [32]. 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 `s` is captured by the *modality* $[\mathtt{s}]\mathrm{post}$ which is true provided that post holds in any terminating state of `s`, expressing partial correctness. To model the heap, the reserved program variable *heap* is used. It maps field names to their value [4, 51]. Variable *heapOld* holds the heap at the time when the current method was scheduled most recently. 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, $\chi, \ldots$ for DL and the term *constraint* and variables α, $\beta, \ldots$ for first-order logic over traces.

Definition 4 (Method Contract). *Let* B *be the set of names for all atomic segments and methods in a given program. A contract for a method* `C.m` *has the following components:*

Context clauses. *1. A heap precondition* $\varphi_{\mathtt{m}}$ *over field symbols for* `C`*; 2. a parameter precondition* $\psi_{\mathtt{m}}$ *over formal parameters of* `C.m`*; 3. a class postcondition* $\chi_{\mathtt{m}}$ *over formal parameters of* `C.m`*, field symbols for* `C`*, and the reserved program variable* $\backslash result$*; 4. an interface postcondition* $\zeta_{\mathtt{m}}$ *only over the reserved program variable* $\backslash result$*. 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* `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* $\mathsf{succeeds}_j$*,* $\mathsf{overlaps}_j \subseteq \mathsf{B}$.

Resolve contracts *for each synchronization point named* j *in* `m`*: a set of method names qualified with their class name:* $\mathsf{resolvedBy}_j \subseteq \mathsf{C} \times \mathsf{M}$

Each `run` method has the contract $\varphi_{\texttt{run}} = \psi_{\texttt{run}} = \mathbf{True}$ and $\mathsf{succeeds}_{\texttt{run}} = \emptyset$. Methods without a specification have the default contract $\varphi_{\texttt{m}} = \psi_{\texttt{m}} = \chi_{\texttt{m}} = \zeta_{\texttt{m}} = \mathbf{True}$ and $\mathsf{succeeds}_{\texttt{m}} = \mathsf{overlaps}_{\texttt{m}} = \mathsf{B}$. 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.

4.1 Method Contracts as Constraints over Traces

Let $\mathcal{M}_{\texttt{m}}$ be the method contract for `m`. The semantics of $\mathcal{M}_{\texttt{m}}$ are four constraints over traces (formalized in Defs. 5–7 below), expressing different aspects of contracts: (i) $\mathsf{assert}(\mathcal{M}_{\texttt{m}}, tr)$ expresses that the postcondition and all suspension assertions hold in tr; (ii) $\mathsf{assume}(\mathcal{M}_{\texttt{m}}, tr)$ that the precondition and all suspension assumptions hold in tr; (iii) $\mathsf{context}(\mathcal{M}_{\texttt{m}}, tr)$ that context sets describe the behavior of the object in tr; and (iv) $\mathsf{resolve}(\mathcal{M}_{\texttt{m}}, tr)$ that methods resolving futures adhere to the resolving contracts in tr. If the method name is clear from the context, we write $\mathcal{M}$ instead of $\mathcal{M}_{\texttt{m}}$. All unbound symbols in the constraints, such as f, e, X, etc., are implicitly universally quantified.

Definition 5 (Semantics of Context Clauses). *Let* $\mathcal{M}_{\texttt{m}}$ *be a method contract,* tr *a trace, and* $\mathsf{susp}(\texttt{m})$ *the set of suspension points in* `m`*:*

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

The 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 trace 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 6 (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 $\mathcal{M}_\mathtt{m}$.

The constraint $\mathsf{resolve}$ expresses that for each synchronization point i, its resolve contract $\mathcal{R}_i$ contains all methods that can resolve it: For every reaction event on a future behind i, there must be a future event on that future which terminates a method from $\mathsf{resolve}_i$.

Definition 7 (Semantics of Resolve Contracts). *Let $\mathcal{M}_\mathtt{m}$ be a method contract, tr a trace, and* $\mathsf{resp}(\mathtt{m})$ *the set of resolve points in* $\mathtt{m}$*. Then the resolve constraint is defined as* $\mathsf{resolve}(\mathcal{M}_\mathtt{m}, tr) = \bigwedge_{i \in \mathsf{resp}(\mathtt{m})} \mathsf{resolve}(\mathsf{resolve}_i, tr)$*, where*

$$\begin{aligned}\mathsf{resolve}(\mathsf{resolve}_i, tr) = &\forall j \in \mathbb{N}.\ \mathsf{ev}^{tr}[j] \doteq \mathsf{futREv}(\mathsf{X}, f, \mathsf{e}, i) \rightarrow\\ &\exists k \in \mathbb{N}.\ k < j \wedge \mathsf{ev}^{tr}[k] \doteq \mathsf{futEv}(\mathsf{X}', f, \mathtt{m}', \mathsf{e}) \rightarrow \bigvee_{\mathtt{m} \in \mathsf{resolve}_i} \mathtt{m} \doteq \mathtt{m}'\end{aligned}$$

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 8 (Coherence). *Let* $\mathsf{CNF}(\varphi)$ *be the conjunctive normal form of φ, such that all function and relation symbols also adher 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.

```
class C(Int f, J o) implements I {
 /*@ requires f == 0;
   @ ensures \result > i;
   @ overlaps {};
   @ succeeds {init}; @*/
Int m(Int i) {
 Fut<Int> ft = o!n();
 /*@ requires f >= 0;
   @ ensures f == 0;
   @ overlaps {up};
   @ succeeds {}; @*/
 [atom: "susp1"] await ft?;
 /*@ resolvedBy {J.n}; @*/
 [sync: "sync"] Int j = ft.get;
 return i + j + f;
 }
}
```

```
interface I {
  /*@ requires i > 0;
    @ ensures \result > 0 @*/
  Int m(Int i); }
interface J { Int n(); }
```

φm	`f == 0`
ψm	`i > 0`
ζm	`\result > 0`
χm	`\result > i`
$\mathsf{succeeds}_{\mathtt{m}}$	`{init}`
$\mathsf{overlaps}_{\mathtt{m}}$	`{}`
φsusp1	`f >= 0`
χsusp1	`f == 0`
$\mathsf{succeeds}_{\mathtt{susp1}}$	`{}`
$\mathsf{overlaps}_{\mathtt{susp1}}$	`{up}`
$\mathsf{resolve}_{\mathtt{sync}}$	`{J.n}`

Fig. 4. A method contract with naming conventions for contract components.

Lemma 1 (Sound Propagation [42]). *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*

$$\forall tr.\big(\mathsf{assert}(\widehat{\mathcal{M}}, tr) \to \mathsf{assert}(\mathcal{M}, tr)\big) \land \big(\mathsf{assume}(\widehat{\mathcal{M}}, tr) \leftrightarrow \mathsf{assume}(\mathcal{M}, tr)\big)$$
$$\land \big(\mathsf{context}(\widehat{\mathcal{M}}, tr) \leftrightarrow \mathsf{context}(\mathcal{M}, tr)\big) \land \big(\mathsf{resolve}(\widehat{\mathcal{M}}, tr) \leftrightarrow \mathsf{resolve}(\mathcal{M}, tr)\big)$$

Proof. To generate the coherent set, for every $\mathcal{S}_i$ in M, the suspension assumption φ_i is added by conjunction to the suspension assertion χ_j of each atomic segment j in $\mathsf{succeeds}_i$, and $(\{heap := heapOld\}\varphi_i) \to \varphi_i$ is added by conjunction to χ_j of each atomic segment j in $\mathsf{overlaps}_i$. This obviously enforces coherence. Coherence is only concerned with the precondition of atomic segments and the postcondition of atomic segments in their context sets, thus the equivalences hold. The implication of assert follows directly from the fact hat the added parts of postconditions are added by a conjunction. □

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.

4.2 On the Structure of Atomic Segments

Fig. 4 shows the specification of method `C.m` and its components. Class `C` implements interface `I` and we omit the specification and implementation of further methods of `I`.

```
Unit m() {
  s₁
  if (fl == 0) {
     s₂
     await f?;
  } else { s₃ }
  s₄
}
```

m has three atomic segments:

```
s₁ s₂ await f?
```

```
s₃ s₄
```

```
s₁ s₃ s₄
```

Fig. 5. Structure of atomic segments. The statements $\mathtt{s}_i$ contain no loops, conditionals, or suspension statements.

The structure of atomic segments in a method may be more complex than in Fig. 4. For example, in the presence of a conditional or loop between two suspension points, a suspension assertion of a suspension contract specifies the condition at the end of *multiple* atomic segments. The set of atomic segments of a method can be computed by generating all paths in the control flow graph that begin at the entry node or an await statement and end at an exit node or an await statement. In our calculus we do not need to compute these atomic segments explicitly, because that is implicitly achieved by symbolic execution. Fig. 5 shows a method and its atomic segments. The postcondition of `m` describes the state at the end of *three* atomic segments.

5 Verification

Like in JML [45] method contracts appear in comments before their interface and class declaration. 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 or method start. Restrictions on the occurrence of fields and parameters are as above.

Definition 9. *Let* `str` *range over strings,* φ *over DL formulas, and* `m` *over method names. The clauses used for specification are defined as follows:*

$$\begin{array}{rlrl}
\mathsf{ISpec} ::= & /{*}@\ \mathsf{Require}\ \mathsf{Ensure}\ \mathsf{Runs}\ @{*}/ & \mathsf{Spec} ::= & /{*}@\ \mathsf{Require}\ \mathsf{Ensure}\ \mathsf{Runs}\ @{*}/\\
\mathsf{ASpec} ::= & /{*}@\ \mathsf{Require}\ \mathsf{Ensure}\ \mathsf{Runs}\ @{*}/ & \mathsf{GSpec} ::= & /{*}@\ \mathsf{Resolve}\ @{*}/\\
\mathsf{Require} ::= & \mathit{requires}\ \psi; & \mathsf{Ensure} ::= & \mathit{ensures}\ \psi;\\
\mathsf{Runs} ::= & \mathit{succeeds}\ \overline{\mathtt{str}};\quad \mathit{overlaps}\ \overline{\mathtt{str}}; & \mathsf{Resolve} ::= & \mathit{resolvedBy}\ \overline{\mathtt{m}} \qquad \psi ::= \varphi \mid \backslash\mathit{last}(\varphi)
\end{array}$$

The parameter precondition and interface precondition are defined by ISpec in an interface declaration. The heap precondition and class postcondition is defined by Spec in a class declaration. The specification at suspension points is defined by ASpec, before the annotated atomic segment. The specification for **get** statements is defined by GSpec. We do not consider loop invariants here, which are standard. For ghost fields and ghost assignments, we follow standard JML [45].

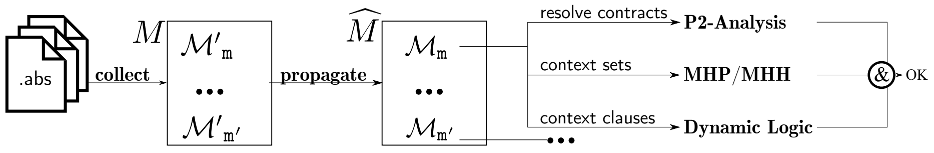

Fig. 6. Workflow of the Verification

5.1 Static Analysis

Our verification approach uses MHP, MHH, and PT static analyses as explained in Sect. 3. Their semantics is formalized using constraints. MHP and MHH guarantee the context constraint of a contract, while PT guarantees the resolve constraint. We use a sequent calculus to connect assert and assume constraints. Fig. 6 shows the overall workflow.

Points-To Analysis. Given a program location k, PT returns a set $\mathsf{p2}(k)$ of method names satisfying the constraint $\mathsf{points}(k, tr)$ defined as follows for each trace tr of the analyzed program:

$$\forall i \in \mathbb{N}.\ \mathsf{ev}^{tr}[i] \doteq \mathsf{futREv}(\mathsf{X}, f, \mathsf{e}, k) \to \exists j \in \mathbb{N}.\ j < i \wedge \bigvee_{\mathtt{m} \in \mathsf{p2}(k)} \mathsf{ev}^{tr}[j] \doteq \mathsf{futEv}(\mathsf{X}', f, \mathtt{m}, \mathsf{e})$$

Lemma 2. *For any named synchronization statement k the following holds: If* PT *returns a subset of the resolving contract, then we can assume the* resolve *constraint for the composition of any trace tr of the analyzed program:*

$$\forall tr.\ \mathsf{p2}(k) \subseteq \mathsf{resolve}_k \to (\mathsf{points}(k, tr) \to \mathsf{resolve}(\mathsf{resolve}_k, tr))$$

May-Happen-In-Parallel and Must-Have-Happened Analysis. Let Prgm be a program. MHP returns a set mhp of pairs of block names (b_1, b_2), such that the order of these blocks differs in some runs of Prgm, but the prefix up to this point is equal in both traces. We define the constraint $\mathsf{mayP}(b_1, b_2)$ as follows:

$$\exists i,j,k \in \mathbb{N}. \exists tr, tr'.\ \mathsf{Prgm} \Downarrow tr \wedge \mathsf{Prgm} \Downarrow tr' \wedge \mathsf{act}(\mathsf{ev}^{tr}[i], b_1) \wedge i < j \wedge \mathsf{act}(\mathsf{ev}^{tr}[j], b_2) \wedge$$
$$\mathsf{act}(\mathsf{ev}^{tr'}[i], b_2) \wedge i < k \wedge \mathsf{act}(\mathsf{ev}^{tr'}[k], b_1) \wedge \forall l \in \mathbb{N}.\ l < i \to \mathsf{ev}^{tr}[l] \doteq \mathsf{ev}^{tr'}[l]$$

where $\mathsf{act}(\mathsf{ev}^{tr}[i], b)$ models that $\mathsf{ev}^{tr}[i]$ is activating atomic segment b, i.e. it is invREv if b is the first block of a method, otherwise reacEv. Finally, $\mathsf{mayP}(\mathsf{MHP}) \equiv \forall (b_1, b_2) \in \mathsf{MHP}.\ \mathsf{mayP}(b_1, b_2)$.

The MHH analysis returns a set mhh of pairs (b_1, b_2) of block names. In every run of Prgm, there is one execution of b_1 before any execution of b_2. The definition of $\mathsf{mustH}(b_1, b_2)$ and $\mathsf{mustH}(\mathsf{MHH})$ follows a similar pattern as MHP. Let $\mathsf{mhp}(b)$ ($\mathsf{mhh}(b)$) be the maximal subset of mhp (mhh) such that each pair has b as one of its elements.

Lemma 3. *Let $\mathcal{M}$ be a method contract. If all sets $\{(b,\, b') \mid b' \in \mathsf{succeeds}_b\}$ are subsets of $\mathsf{mhh}(b)$ and all sets $\{(b,\, b') \mid b' \in \mathsf{overlaps}_b\}$ are supersets of $\mathsf{mhp}(b)$, then the* context *constraint holds for all traces.*

$$\forall b \in B.\, \{(b,b') \mid b' \in \mathsf{succeeds}_b\} \subseteq \mathsf{mhh}(b) \wedge \{(b,b') \mid b' \in \mathsf{overlaps}_b\} \supseteq \mathsf{mhp}(b) \rightarrow \Big(\forall tr.\, \mathsf{Prgm} \Downarrow tr \rightarrow \big(\mathsf{mayP}(\mathsf{MHP}) \wedge \mathsf{mustH}(\mathsf{MHH}) \rightarrow \mathsf{context}(\mathcal{M}, tr)\big)\Big)$$

MHP (MHH) is used as a black box represented by mayP (mustH). The framework can be extended to include other analyses or concurrency models, such as scheduling policies or FIFO messaging.

5.2 Sequent Calculus.

The DL calculus rewrites a program formula [`s;r`]post with a leading statement `s` into the formula [`r`]post plus suitable first-order constraints. Repated rule application yields symbolic execution of the program in the modality. *Updates* (see Sect. 4) accumulate during symbolic execution to capture state changes; for example, [`v = e;r`]post is replaced with $\{$`v` $:=$ `e`$\}$[`r`]post, expressing that `v` has the value of `e` during symbolic execution of `r`. When a program `s` has been completely executed, the modality is empty and accumulated updates are applied to the postcondition post, resulting in a pure first-order formula that represents the weakest precondition of `s` and post. Formulas are evaluated relative to a variable assignment and a trace.

Definition 10. *Let tr be a trace and β an assignment for logical variables. The* evaluation function $[\![\cdot]\!]_{tr,\beta}$ *maps DL formulas to truth values* $\{\mathbf{tt}, \mathbf{ff}\}$.

The definition of the evaluation function can by derived from the one given in [22] by using only the final configuration of tr.[4] We use a sequent calculus to prove validity of DL formulas [4, 21]. In sequent notation pre $\rightarrow$ [`s`]post is written as Γ, pre $\Longrightarrow$ [`s`]post, Δ, 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). A DL-based proof system for verifying class invariants of ABS programs is available [22].

We formulate DL proof obligations for the correctness of method contracts. Their soundness rests on the assumptions for the MHP, MHH, and PT analyses spelled out in Lemmas 2 and 3. The DL sequent to be proven for a method `m` with body `s` and contract $\mathcal{M}_{\mathtt{m}}$ as in Def. 4 has the form:

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

The heap and parameter preconditions $\varphi_{\mathtt{m}}$ and $\psi_{\mathtt{m}}$ of $\mathcal{M}_{\mathtt{m}}$ are assumed when execution starts, likewise that the trace of the object up to now is well-formed. The

[4] Technically, we extend traces to include configurations that issue no communication. We omit details as all rules and cases are covered in [22].

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

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

$$\textsf{(get)}\ \frac{fresh(\mathtt{r}, \mathbb{t}),\ \left(\bigvee_{\mathtt{m} \in \mathsf{resolve}(i)} \zeta_{\mathtt{m}}(\mathtt{r})\right) \Longrightarrow \{U\}\{\mathtt{v:=r}\}\{\mathbb{t} := \mathbb{t} \cdot \mathsf{futREv}(\mathsf{this}, \mathtt{f}, \mathtt{r}, i)\}[\mathtt{s}]\chi}{\Longrightarrow \{U\}[\mathtt{[sync:\ "i"]\ v\ =\ f.}\mathbf{get}\mathtt{;s}]\chi}$$

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

Fig. 7. Selected DL proof rules.

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 *\last* is replaced with *heapOld* and *\last* is removed. Reserved variables $\mathbb{t}$, this, $\mathbb{f}$, and $\mathbb{m}$ record the current trace, object, future, and method, respectively, during symbolic execution. The above sequent must be proved for each method of a program using schematic proof rules as shown in Fig. 7. There is one rule per statement kind in Async. The rules for sequential statements are standard and omitted. To improve readability, we leave out the sequent contexts Γ, Δ and assume 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 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. 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.

Rule **(get)** introduces a fresh constant `r` representing the value stored in future `f`. By Lemma 2 we can assume at least one of the interface postconditions of methods in $\mathsf{resolve}(i)$ to hold[5] for `r`. The current trace is extended with a future reaction event.

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

[5] We rule out the possibility that no suitable future was resolved and the current process blocks, because our version of the DL rules is for partial correctness. Blocking can be easily modeled by searching $\mathbb{t}$ for future events of methods in $\mathsf{resolve}(i)$.

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" [4, 51] $heap_A$, 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.

5.3 Soundness

Proving soundness follows a two-tier approach: First, we show that the rules themselves are sound. Then, we show that if the static analyses succeed and all proof obligations can be closed, each trace satisfies all constraints of all contracts.

We slightly modify the usual notion of rule soundness. To incorporate the results of the static analyses, we restrict the traces we reason about to those generated by the analyzed program: A rule is Prgm-sound if the premisses imply the conclusion in every trace of Prgm.

Definition 11 (Prgm-Soundness). *Let* Prgm *be a program. A rule with premisses $P_1 \ldots P_n$ and conclusion C is* Prgm-*sound if for every β and every partial trace tr of* Prgm *the following holds:* $\left(\bigwedge_{i\leq n}[\![P_i]\!]_{tr,\beta}\right) \to [\![C]\!]_{tr,\beta}$.

This relativized soundness notion is required to decompose program correctness from the correctness of the auxiliary static analyses. It does not compromise the modularity of our approach, because the verification rules do not assume anything about Prgm than its soundness relative to the results of the static analyses. Relative soundness makes it possible to connect method contracts and characterize them using the assert and assume constraints:

Lemma 4 (Contracts as Constraints). *Let* Prgm *be a coherent program and $\widehat{M}$ its method contracts. Let $\mathcal{M}_{\mathtt{m}} \in \widehat{M}$. If (i) the* PT, MHP *and* MHH *analyses succeed as described in Lemmas 2 and 3, (ii) for all traces tr of* Prgm *and $\mathcal{M}_{\mathtt{m}'} \in \widehat{M} \setminus \{\mathcal{M}_{\mathtt{m}}\}$ the constraint* $\mathsf{assert}(\mathcal{M}_{\mathtt{m}'}, tr)$ *holds, and (iii) the proof obligation* (PO) *for $\mathcal{M}_{\mathtt{m}} \in \widehat{M}$ can be shown; then the following holds:*

1. *All rules in Fig. 7 are* Prgm-*sound if all rules in [22] are sound*
2. $\forall tr.\ \mathsf{Prgm} \Downarrow tr \to \big(\mathsf{assume}(\mathcal{M}_{\mathtt{m}}, tr) \to \mathsf{assert}(\mathcal{M}_{\mathtt{m}}, tr)\big)$

As all analyses are described using constraints over traces, soundness can now be stated as follows:

Theorem 1 (Soundness of Compositional Reasoning). *Let* Prgm *be a program, M its set of method contracts, and $\widehat{M}$ the coherent set of method contracts from Lemma 1. If (i) the* PT, MHP *and* MHH *analyses succeed on $\widehat{M}$ as described in Lemmas 2, 3 and (ii) for each $\mathcal{M}_{\mathtt{m}} \in \widehat{M}$ the proof obligation can be shown, then the following holds for all tr with* $\mathsf{Prgm} \Downarrow tr$*:*

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

6 Conclusion, Related and Future Work

This paper generalizes rely-guarantee reasoning with method contracts to active objects with futures. This asynchronous setting challenges contract-based rely-guarantee reasoning, compared to the well-known synchronous setting: The delay between the invocation and activation of method calls means that preconditions cannot solely be guaranteed at call time. Similar delays exist for interleaving and method return. In addition, strong encapsulation means that preconditions that depend on local fields cannot be evaluated by the caller. These challenges restricted previous work on verification for active object languages to reason about the preservation of monitor invariants. To overcome these challenges, we separate the responsibilities of the caller and the callee by splitting the precondition into a parameter precondition and a heap precondition, and likewise, the postcondition into an interface postcondition and a class postcondition. The parameter precondition and interface postcondition can expose the method implementation. The heap precondition and class postcondition can be stronger than a class invariant, because they do not need to be maintained by all methods. Instead, context sets are introduced to specify the methods that have to establish or maintain the heap precondition. This separation of concerns entails modularity and allows to propagate preconditions within a class without having to access program-global information.

Related Work. Wait conditions were introduced as program statements (not in method contracts) in the pioneering work of Brinch-Hansen [30, 31] and Hoare [33]. SCOOP [8] explores preconditions as wait/when conditions. Previous approaches to AO verification [20, 22] 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 [49] 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, 39] implemented on top of Frama-C by Gavran et al. [27] 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 [13, 35]. These have been adapted to the active object concurrency model [41], including assertions on heap memory [40], but require composition to be explicit in the specification. Stateful session types for active objects [40] contain a propagation step (cf. Sect. 2.2): 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 which can rely on it.

Approaches to compositional specification of concurrent systems which do not follow rely-guarantee have been proposed. For example, *Separation Logic* [15, 48] separates shared memory regions and assigns responsibilities for regions to processes. Huisman et al. [12,53] have used permission-based separation logic to verify class invariants in multi-threaded programs, using barrier contracts. Shared regions [24] are related to heap preconditions and suspension assumptions in our work, in the sense that a heap precondition is a region associated with a predicate that must be stable. In contrast, context sets in our work describe precisely *when* and *by whom* the predicate must be stabilized. Although approaches to precisely specify regions have been developed [18, 24], their combination with additional modes of interactions than heap accesses (such as asynchronous calls and futures in our case) is not well explored.

Villard et al. [50] consider synchronous message passing in a setting where parameters are transmitted copyless and thus the heaps may leak. Kragl et al. [44] address asychronous procedure calls in a multi-threaded concurrency model over global state by identifying conditions for sequentialization such as commutativity of atomic actions. They do not address futures and cooperative concurrency, and their proof system is non-compositional, based on non-modular specifications for complete programs. Kloos et al. [43] provide an analysis tool that infers types for asynchronous tasks, but do not consider modular specification. Compared to our work, they focus on the correct usage of ownership types and heap invariants, not on functional properties of single methods.

It is worth noting that active objects do not require the notion of regions inside the logic, because strong encapsulation and cooperative scheduling ensure that two threads never run in parallel on the same heap. What is achieved by separation logic—separation of heaps—appears as a feature of the active object concurrency model. An extension of our approach where separation logic is used in preconditions is an interesting topic for future work that could address concurrency models with asynchronous calls and futures/channels, but less strict encapsulation. The Kappa type system [17] combines actors with ownership capabilities to enable sharing of data between active objects with weaker encapsulation propetries [14]. There is currently no proof system for such active object languages, which appear as an interesting application domain to integrate separation logic in our context.

Future Work. The correctness of context set specifications may be inferred by static analysis techniques such as May-Happen-in-Parallel [6]. Common features of synchronous method contracts, such as termination witnesses [28] or exceptions [4] should be integrated. The exact relationship of the present work to session types and regions ins separation logic remains to be formalized.

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. B. Beckert and R. Hähnle. Reasoning and verification. *IEEE Intelligent Systems*, 29(1):20–29, Jan.–Feb. 2014.
12. S. Blom, M. Huisman, and M. Mihelcic. Specification and verification of GPGPU programs. *Sci. Comput. Program.*, 95:376–388, 2014.
13. 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.
14. S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. Tapia Tarifa, T. Wrigstad, and A. M. Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, volume 9104 of *LNCS*, pages 1–56. Springer, 2015.
15. S. Brookes and P. W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, Aug. 2016.
16. 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.

17. E. Castegren and T. Wrigstad. Reference capabilities for concurrency control. In S. Krishnamurthi and B. S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 5:1–5:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
18. 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.
19. 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.
20. 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.
21. 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.
22. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
23. 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.
24. 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.
25. C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.
26. A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FMOODS/FORTE*, volume 7892 of *LNCS*, pages 273–288. Springer, 2013.
27. 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, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
28. D. Grahl, R. Bubel, W. Mostowski, P. H. Schmitt, M. Ulbrich, and B. Weiß. Modular specification and verification. In 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*, pages 289–351. Springer, Cham, 2016.
29. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.
30. P. B. Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, 1972.
31. P. B. Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.
32. D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. *SIGACT News*, 32(1):66–69, 2001.
33. C. A. R. Hoare. Towards a theory of parallel programming. *Operating System Techniques*, pages 61–71, 1972.

34. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
35. 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.
36. M. Huisman, W. Ahrendt, D. Grahl, and M. Hentschel. Formal specification with the java modeling language. In *Deductive Software Verification*, volume 10001 of *LNCS*, pages 193–241. Springer, 2016.
37. 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.
38. 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 International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer-Verlag, 2011.
39. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, Oct. 1983.
40. E. Kamburjan and T. Chen. Stateful behavioral types for active objects. In *IFM*, volume 11023 of *LNCS*, pages 214–235. Springer, 2018.
41. E. Kamburjan, C. C. Din, and T. Chen. Session-based compositional analysis for actor-based languages using futures. In *ICFEM*, volume 10009 of *LNCS*, pages 296–312, 2016.
42. 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/.
43. J. Kloos, R. Majumdar, and V. Vafeiadis. Asynchronous liquid separation types. In *ECOOP*, volume 37 of *LIPIcs*, pages 396–420. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
44. B. Kragl, S. Qadeer, and T. A. Henzinger. Synchronizing the asynchronous. In *CONCUR*, volume 118 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
45. 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.
46. 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, Atlanta, GE, USA, June 1988. ACM Press.
47. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.
48. 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.
49. A. J. Summers and P. Müller. Actor services - modular verification of message passing programs. In P. Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *LNCS*, pages 699–726. Springer, 2016.

50. J. Villard, É. Lozes, and C. Calcagno. Proving copyless message passing. In Z. Hu, editor, *Programming Languages and Systems*, pages 194–209. Springer Berlin Heidelberg, 2009.
51. B. Weiß. *Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.
52. 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.
53. M. Zaharieva-Stojanovski and M. Huisman. Verifying class invariants in concurrent programs. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 230–245, 2014.

Appendix

A Full Example

The full, unspecified code including of our running example is:

```
interface IPositive{
  List<Rat> smooth(List<Rat> input, Rat a):
}
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;
 }
}
```

The fully specified code is as follows. submin(`input`, i) returns the minimal element of `input` up to position i. submax(`input`, i) returns the maximal element of `input` up to position i. len(`input`) returns the length of `input`.

The loop invariant is

```
/*@ loop_invariant
  (\forall Int i; i >= 0 && i < length(inter); inter[i] > 0
     &&  (\exists int min, max; min >= 0 && max >= 0
                    && min < length(input) && max < length(input)
             && inter[i] <= input[max]
             && inter[i] >= input[min]))
   && \forall Int i; i >= 0
   && i < length(work); inter[i] <= submax(input,i)
   && inter[i] >= submin(input,i)
   && length(work) + length(inter) = length(input)
   &&  input != inter && input == \old(input);
   decreases length(work);
@*/
```

The specification of the interfaces is

```
interface Computation{
/*@ requires inew > 0 && iold > 0 && param > 0;
    ensures  \result > 0
      && (iold >= inew −> (\result <= iold && \result >= inew))
      && (iold <= inew −> (\result >= iold && \result <= inew));
@*/
Rat cmp(Rat iold, Rat inew, Rat param);
}

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{
 /*@ requires comp != null @*/
 Unit setup(Computation comp);
 Int getCounter();
 /*@ requires 1>a>0 && len(input)>0;
     ensures len(\result) == len(input) &&
            \forall Int i; 0 <= i < len(\result);
              \result[i] > 0 && min(input) <= \result[i] <= max(input);
   succeeds {setup, smooth}; overlaps {getCounter};  @*/
 List<Rat> smooth(List<Rat> input, Rat a);
}
```

The specification of the class is

```
class Smoothing implements ISmoothing {
 Computation c = null;
 Int counter = 1;
 //@ ghost Bool lock = False;
 Unit setup(Computation comp) { c = comp; }
 Int getCounter() { return counter; }
/*@ requires !lock && c!=null;
  ensures !lock; @*/
 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;
/*@ requires lock && c != null;
  ensures True;
  succeeds {awSmt}; overlaps {getCounter};
   @*/
   [atom: "awSmt"] await f?;
   /*@ resolvedBy {Computation.cmp} @*/
   [sync: "sync"] Rat res = f.get;
   inter = concat(inter,list[res]);
   work = tail(work);
  }
  //@lock = False;
  counter = 1;
  return inter;
 }
}
```

The fully specified code after progapation, slightly simplified, without context sets and after moving the pre- and postcondition to the class, is:

```
class Smoothing  implements ISmoothing {
 Computation c = null;
 Int counter = 1;
//@ ghost Bool lock = False;
/*@ requires !lock &&comp != null @*/
/*@ ensures !lock && c!=null @*/
 Unit setup(Computation comp) { c = comp; }
/*@ ensures \last(!lock && c != null) ==> !lock && c != null @*/
 Int getCounter() { return counter; }
/*@ requires !lock && c!=null @*/
/*@ requires \forall Int i; 0 <= i < len(input) ; input[i] > 0 @*/
/*@ requires 1>a>0 && len(input)>0 @*/
/*@ ensures len(\result) == len(input) &&
            \forall Int i; 0 <= i < len(\result);
             \result[i] > 0 && min(input) <= \result[i] <= max(input); @*/
/*@ ensures !lock && c!=null @*/
 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;
/*@ requires lock && c != null; @*/
/*@ ensures lock && c != null;  @*/
   [atom: "awSmt"] await f?;
   /*@ resolvedBy {Computation.cmp} @*/
   [sync: "sync"] Rat res = f.get;
   inter = concat(inter,list[res]);
   work = tail(work);
  }
  //@lock = False;
  counter = 1;
  return inter;
 }
}
```

B All Constraints and Examples

Definition 12 (Semantics of Context Clauses). *Let $\mathcal{M}_{\mathtt{m}}$ be a method contract and tr a trace:*

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

Definition 13 (Semantics of Context Sets). *Let $\mathcal{M}_{\mathtt{m}}$ be a method contract, $\mathcal{S}_n$ be a suspension contract, and tr a trace: The semantics of context sets of a suspension contract $\mathcal{S}_n$ is defined by the following constraint* $\mathsf{context}(\mathcal{S}_n, tr)$*:*

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

The constraint $\mathsf{context}(\mathcal{M}_{\mathtt{m}}, tr)$ *for the semantics of context sets in method contracts is defined analogously:*

$$\begin{aligned}\mathsf{context}(\mathcal{M}_{\mathtt{m}}, tr) = &\bigwedge_{n \in \mathsf{susp}(\mathtt{m})} \mathsf{context}(\mathcal{S}_n, tr) \wedge \forall i \in \mathbb{N}.\ \mathsf{ev}^{tr}[i] \doteq \mathsf{invREv}(\mathsf{X}', \mathsf{X}, f, \mathtt{m}, \overline{\mathsf{e}})\\ &\rightarrow \exists k \in \mathbb{N}.\ k < i \wedge \Big(\bigvee_{j' \in \mathsf{succeeds}_n} \mathsf{ev}^{tr}[k] \doteq \mathsf{termEvent}(j') \wedge\\ &\forall k' \in \mathbb{N}.\ k < k' < i \wedge \mathsf{isClose}(\mathsf{ev}^{tr}[k']) \rightarrow \big(\bigvee_{j' \in \mathsf{overlaps}_n} \mathsf{ev}^{tr}[k'] \doteq \mathsf{termEvent}(j')\big)\Big)\end{aligned}$$

Definition 14 (Semantics of Resolve Contracts). *Let $\mathcal{M}_{\mathtt{m}}$ be a method contract, tr a trace, and* $\mathsf{resp}(\mathtt{m})$ *the set of resolve points in* $\mathtt{m}$.

Define $\mathsf{resolve}(\mathcal{M}_{\mathtt{m}}, tr) = \bigwedge_{i \in \mathsf{resp}(\mathtt{m})} \mathsf{resolve}(\mathsf{resolve}_i, tr)$*, with*

$$\begin{aligned}\mathsf{resolve}(\mathsf{resolve}_i, tr) = &\forall j \in \mathbb{N}.\ \mathsf{ev}^{tr}[j] \doteq \mathsf{futREv}(\mathsf{X}, f, \mathsf{e}, i) \rightarrow\\ &\exists k \in \mathbb{N}.\ k < j \wedge \mathsf{ev}^{tr}[k] \doteq \mathsf{futEv}(\mathsf{X}', f, \mathtt{m}', \mathsf{e}) \rightarrow \bigvee_{\mathtt{m} \in \mathsf{resolve}_i} \mathtt{m} \doteq \mathtt{m}'\end{aligned}$$

Definition 15 (Semantics of Points-To). *Given the name k of a program location, the PT analysis returns a set* $\mathsf{p2}(k)$ *of method names that satisfy the following constraint* $\mathsf{points}(\mathsf{p2}(k), tr)$ *for each trace tr generated by the analyzed program:*

$$\forall i \in \mathbb{N}.\ \mathsf{ev}^{tr}[i] \doteq \mathsf{futREv}(\mathsf{X}, f, \mathsf{e}, k) \rightarrow \exists j \in \mathbb{N}.\ \bigvee_{\mathtt{m} \in \mathsf{p2}(k)} \mathsf{ev}^{tr}[j] \doteq \mathsf{futEv}(\mathsf{X}', f, \mathtt{m}, \mathsf{e})$$

Definition 16 (Semantics of MHP and MHF). *Let* Prgm *be a program. The MHP analysis returns a set* MHP *of pairs of block names (b_1, b_2), such that* *there are two runs of* Prgm*, where the order of these blocks is different, but the prefix up to this point is equal in both traces. We define constraint* $\mathsf{mayP}(b_1, b_2)$ *as follows:*

$$\exists i,j,k \in \mathbb{N}. \exists tr, tr'.\ \mathsf{Prgm} \Downarrow tr \land \mathsf{Prgm} \Downarrow tr' \land \mathsf{act}(\mathsf{ev}^{tr}[i], b_1) \land i < j \land \mathsf{act}(\mathsf{ev}^{tr}[j], b_2) \land \big(\mathsf{act}(\mathsf{ev}^{tr'}[i], b_2) \land i < k \land \mathsf{act}(\mathsf{ev}^{tr'}[k], b_1)\big) \land \forall l \in \mathbb{N}.\ l < i \rightarrow \mathsf{ev}^{tr}[l] \doteq \mathsf{ev}^{tr'}[l]$$

where $\mathsf{act}(\mathsf{ev}^{tr}[i], b)$ *models that* $\mathsf{ev}^{tr}[i]$ *is activating atomic segment b, i.e., it is a* invREv *or* reacEv*, depending on whether b is the first block of a method or not. Finally,* $\mathsf{mayP}(\mathsf{MHP}) = \forall (b_1, b_2) \in \mathsf{MHP}.\ \mathsf{mayP}(b_1, b_2)$*.*

The Must-Have-Finished (MHF) analysis returns a set MHH *of pairs* (b_1, b_2) *of block names, such that in every run of* Prgm*, there is one execution of* b_1 *before any execution of* b_2*. Predicate* $\mathsf{mustH}(b_1, b_2)$ *is defined as:*

$$\forall tr.\ \mathsf{Prgm} \Downarrow tr \rightarrow \big(\forall i. \mathsf{act}(\mathsf{ev}^{tr}[i], b_2) \rightarrow \exists j < i.\ \mathsf{act}(\mathsf{ev}^{tr}[j], b_1)\big)$$

And $\mathsf{mustH}(\mathsf{MHH}) = \forall (b_1, b_2) \in \mathsf{MHH}.\ \mathsf{mustH}(b_1, b_2)$*. Let* $\mathsf{MHP}(b)$ *(*$\mathsf{MHH}(b)$*) be the maximal subset of* MHP *(*MHH*) such that each pair has b as one of its elements.*

C Proofs

C.1 Proof for Lemma 4

We show Lemma 4 by showing (1) and (2) first. Property (3) follows from it.

The following Lemma states the soundness of the given proof rules and connects the proof obligation with the constraints. We require for the soundness of a fixed contract that the precondition, suspension-assumptions and resolving contracts of other method contracts hold.

Lemma 5 (Soundness of Single Method Contracts). *Let* Prgm *be a program, M its set of method contracts and* $\widehat{M}$ *the coherent set of method contracts from Lemma 1. Let* $\mathcal{M}_{\mathtt{m}} \in \widehat{M}$ *be a method contract. If*

1. *the Points-to, MHP and MHF analyses succeed on* $\widehat{M}$ *as described in Lemma 2 and Lemma 3,*
2. *For every partial trace tr' of* Prgm *and every* $\mathcal{M}_{\mathtt{m'}} \in \widehat{M} \setminus \{\mathcal{M}_{\mathtt{m}}\}$ *the assertion constraint holds:* $\mathsf{assert}(\mathcal{M}_{\mathtt{m'}}, tr)$*, and*
3. *all the other proof rules for statements are* Prgm*-sound [22].*

Then the rules in Figure 7 are Prgm*-sound.*

Proof. The rule **(return)** is also shown to be sound in [22]. Rule **(async)** is shown to be sound in [22] without the first premise.

Rule **(get)** is shown to be sound without the antecedent in the premise in [22]. It is thus only required to show that the antecedent holds in every partial trace. Let `i` be a fixed annotation for some **get** statement. First, by (1), a future location `f` only contains futures from methods in resolve(`i`).

Let tr be a partial trace of Prgm ending in a futREv event for the statement annotated with $\mathtt{i}$ and β a variable assignment, such that the rest of the premise holds. We must show that the following holds:

$$\forall r.\ \mathsf{resolvedIn}(r, tr, \mathsf{resolve}(i)) \rightarrow \left[\!\!\left[\bigvee_{\mathtt{m}' \in \mathsf{resolve}(i)} \widehat{\chi_{\mathtt{m}'}}(r) \right]\!\!\right]_{tr,\beta} = \mathbf{tt}$$

Where $\mathsf{resolvedIn}(r, tr, \mathsf{resolve}(i))$ models that r is the value of some resolved future of a method in $\mathsf{resolve}(i)$ in tr. We are required to show that every execution where $\mathtt{m}$ has not yet been executed, preserves α. Thus, we proceed with an induction on the number n of termination events in the trace tr:

Base Case $n = 0$**.** In this case there is no r with $\mathsf{resolvedIn}(r, tr, \mathsf{resolve}(i))$, we can not execute a **get** statement and the program blocks. Note that we only consider partial correctness.

Induction Step $n = n' + 1$**.** I.e., there is a futEv for some method $\mathtt{m}' \in \mathsf{resolve}(i)$ at position j for some future containing r within tr. W.l.o.g. we assume that this position j is the last termination event in tr. By induction hypothesis:

$$\forall r.\ \mathsf{resolvedIn}(r, tr[1..j-1], \mathsf{resolve}(i)) \rightarrow \left[\!\!\left[\bigvee_{\mathtt{m}' \in \mathsf{resolve}(i)} \widehat{\chi_{\mathtt{m}'}}(r) \right]\!\!\right]_{tr[1...j-1],\beta} = \mathbf{tt}$$

By (2) we have closed the proof for all methods $\mathtt{m}'$ with contracts $\mathcal{M}_{\mathtt{m}'} \in \widehat{M} \setminus \{\mathcal{M}_{\mathtt{m}}\}$. It remains to show that the proof for the contract of $\mathtt{m}'$ implies $\widehat{\chi_{\mathtt{m}'}}(r)$. The obligation has the form $\varphi \rightarrow [\mathtt{s}]\chi_{\mathtt{m}'}$, where $\mathtt{s}$ is the corresponding method body. If the proof can be closed, than the last symbolic execution rule is **(return)** and thus $\chi_{\mathtt{m}'}$ holds in the final state. Thus one of the $\chi_{\mathtt{m}'}$ formulas holds for the value r. Note that the method body of $\mathtt{m}'$ may also contain **get** statements, we make a case distinction on the number m of futREv events in $tr[1...j-1]$.

Case $m = 0$**.** In this case the fetch i is the first, thus we do not rely on the soundness of **(get)** for the other contracts and the rule is sound.

Case $m > 0$**.** The previous rule applications of symbolic execution with **(get)** correspond to execution steps within $tr[1...j-1]$, where by induction hypothesis we can assume the above property holds - thus these rule applications were sound and so is the last one.

(await) is proven analogously (with a common induction with rule **(get)**). $\square$

A direct consequence of Lemma 5 and the form of the proof obligation is the following corollary, Lemma 4. We are thus able to abstract away from method contracts and only reason about them in terms of their logical characterization over traces.

Corollary 1. *If the proof obligation for* $\mathcal{M}_{\mathtt{m}} \in \widehat{M}$ *can be shown, then the following holds:*

$$\forall tr.\ \mathsf{Prgm} \Downarrow tr \rightarrow \big(\mathsf{assume}(\mathcal{M}_{\mathtt{m}}, tr) \rightarrow \mathsf{assert}(\mathcal{M}_{\mathtt{m}}, tr)\big)$$

C.2 Proof for Theorem 1

Let Prgm *be a program, M its set of method contracts and $\widehat{M}$ the coherent set of method contracts from Lemma 1. If (1) the Points-to, MHP and MHF analyses succeed on $\widehat{M}$ as described in Lemma 2 and Lemma 3 and (2) For each $\mathcal{M}_{\mathtt{m}} \in \widehat{M}$ the proof obligation can be shown then the following holds for all tr with* $\mathsf{Prgm} \Downarrow tr$*:*

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

Proof. We proof a slightly stronger statement for all *partial* traces of Prgm.

$$\bigwedge_{\mathcal{M}_{\mathtt{m}} \in \widehat{M}} \Big(\mathsf{assert}(\mathcal{M}_{\mathtt{m}}, tr) \land \mathsf{assume}(\mathcal{M}_{\mathtt{m}}, tr) \land \mathsf{context}(\mathcal{M}_{\mathtt{m}}, tr) \land \mathsf{resolve}(\mathcal{M}_{\mathtt{m}}, tr) \land$$
$$\big(\forall i \in \mathbb{N}.\ \mathsf{ev}^{tr}[i] \doteq \mathsf{invEv}(\mathsf{X}', \mathsf{X}, f, \mathtt{m}, \overline{\mathsf{e}}) \rightarrow \mathsf{C}^{tr}[i] \models \psi_{\mathtt{m}}\big)\Big)$$

We observe that the following holds by assumption (1):

$$\bigwedge_{\mathcal{M}_{\mathtt{m}} \in \widehat{M}} \big(\mathsf{context}(\mathcal{M}_{\mathtt{m}}, tr) \land \mathsf{resolve}(\mathcal{M}_{\mathtt{m}}, tr)\big)$$

Furthermore, by the restriction of the main block, each partial trace tr of runs of Prgm has the form

$$tr = [(\mathsf{C}_0, \mathsf{invEv}(\mathbf{main}, \mathsf{X}_1, f_1, \mathtt{run}, \epsilon), \ldots, (\mathsf{C}_0, \mathsf{invEv}(\mathbf{main}, \mathsf{X}_k, f_k, \mathtt{run}, \epsilon)] \circ tr'$$

Induction on $|tr'|$:

Induction Base $tr' = \epsilon$**:** The assume and assert constraints hold as no event of the correct form is in tr. The last part of the conjunction holds because $\psi_{\mathtt{run}} = \mathbf{True}$.

Induction Step $tr' = tr'' \circ [(\mathsf{C}, \mathsf{ev})]$**:** Case distinction on ev:

Invocation Event: I.e., $\mathsf{ev} = \mathsf{invEv}(\mathsf{X}', \mathsf{X}, f, \mathtt{m}, \overline{\mathsf{e}})$. We have to show that $\mathsf{C} \models \psi_{\mathtt{m}}(\overline{\mathsf{e}})$. Let $\mathtt{m}'$ be the method which is active on X' in C.
By assumption (2) the proof obligation of $\mathcal{M}_{\mathtt{m}'}$ has been proven. Thus the statement `v = e'!m'(e)` has been symbolically executed by rule (async). As the proof has been closed, the first premise has been shown.
Note that the first premise may rely on future reads before the asynchronous call statement, and on the precondition of $\mathtt{m}'$. This is covered by the Prgm-soundness of (async), where we use the induction hypothesis for assumption (2) of Lemma 5 and the Corollary 1. Thus $\mathsf{C} \models \psi_{\mathtt{m}}(\overline{\mathsf{e}})$ holds.

Invocation Reaction Event: I.e., $\mathsf{ev} = \mathsf{invREv}(\mathsf{X}', \mathsf{X}, f, \mathtt{m}, \overline{\mathsf{e}})$. We have to show that $\mathsf{C} \models \varphi_{\mathtt{m}} \land \psi_{\mathtt{m}}$.

State Precondition As the specification is coherent and by induction hypothesis we can assume that there is C' with $\mathsf{C}' \models \varphi_{\mathtt{m}}$ at some pair $(\mathsf{C}', \mathsf{ev})$ at position l in tr', where ev is either a suspension or termination event. By assumption (1), that $\mathsf{context}$ holds in tr'' and the object X was not active in $tr[l..|tr|]$. By a simple argument over the semantics it holds that if an object is not active, than its state does not change and $\mathsf{C} \models \varphi_{\mathtt{m}}$.

Parameter Precondition This case is analogous to the above, but we use the additional conjunct of the stronger assumption.

Termination Event: $\mathsf{ev} = \mathsf{futEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})$. This case is analogous to the Invocation Event case, except that the soundness of **(return)** is used.

Suspension Event: $\mathsf{ev} = \mathsf{suspEv}(\mathsf{X}, f, \mathtt{m}, j)$. This case is analogous to the Invocation Event case, except that the soundness of **(await)** is used.

Termination Reaction Event: I.e., $\mathsf{ev} = \mathsf{futREv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})$. This case is analogous to the Invocation Event case, except that the soundness of **(get)** is used.

Reactivation Event: $\mathsf{ev} = \mathsf{reacEv}(\mathsf{X}, f, \mathtt{m}, j)$. This case is analogous to the State Precondition of the Invocation Reaction case, but requires additionally the soundness of **(await)**.