Behavioral 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. We introduce *cooperative contracts*, a contract-based specification approach designed for cooperative scheduling. It permits to specify complex concurrent behavior succinctly. Cooperative contracts are formalized as behavioral contracts in a compositional behavioral program logic in which they can be formally verified.

1 Introduction

Specification contracts for methods are *the* pivotal concept that makes deductive verification of non-trivial programs feasible [33]. The main idea is simple: the behavior of each method of a program is described (precisely or approximately) by a declarative contract, consisting of (i) logical formulas characterizing its pre- and poststates and (ii) a set of memory locations that limit the method's frame (assignable locations). This allows a deductive verification system to replace a method call with a declarative description obtained from its contract. Verification of a large program is thus broken down into tasks of manageable size that consist in verifying that each method satisfies its contract. Formal specification languages based on contracts exist, for example, for the industrial languages Java [51] and C [11].

The contract-based approach works well for *sequential* programs and is supported in industrial-strength verification tools [4, 40, 50, 52], but it is notoriously difficult to specify *concurrent* programs. The reason is that in the sequential case specification and execution are based on the same unit of computation: a method. In standard concurrency models (such as in C, Java, or Scala) this is not the case: method execution can be interrupted ("preempted") at any time by another method with a possibly overlapping frame. To combat the myriad of possible interleavings with accordingly complex data races, it becomes necessary to encode program-wide assumptions about permitted scheduling sequences into method contracts. This in turn leads to contracts becoming bulky, hard to write, and even harder to understand, because their local nature is lost [12].

An extreme solution to this problem is to restrict the permitted form of concurrency radically, as in actor-based, distributed programming [8], where methods are executed atomically and concurrency only occurs among actors with disjoint heaps. In this setting, behavior—like in the sequential case—can be completely specified at the level of interfaces, typically in terms of behavioral invariants jointly maintained by an object's methods [22, 28]. However, this restricted concurrency enforces that systems are modeled and specified at a high level of abstraction, essentially as protocols. It precludes realistic modeling of concurrent behavior, such as waiting for results computed asynchronously on the same processor and heap.

Cooperative scheduling, as realized in *active object* (AO) languages [21], occupies a middle ground between preemption and full distribution. It is based on an actor-*like* model of concurrency [3] and *futures* to handle return values from asynchronous calls [10, 18, 22, 30, 34, 54, 61]. Programs voluntarily and in a *syntactically explicit* manner suspend their execution, such that a required result may be provided by another task: Method activations on the same processor and heap *cooperate* to achieve a common goal. The crucial point is that code locations where suspension may occur are explicitly marked and *only* at those locations preemption may occur. At the same time, *strong encapsulation* is enforced: an object may only access its own fields. In consequence, data races can only occur between tasks executing on the same object.

It was demonstrated in several case studies that cooperative scheduling permits realistic modeling of concurrent behavior of industrial software, allowing, for example, precise runtime prediction [6, 53] or the exhibition of performance bugs [60]. A still open problem is *deductive verification* of non-trivial programs with cooperative concurrency. The research question is: *Is there a generalization of sequential specification contracts to the cooperative setting that permits succinct, intelligible specifications and a compositional calculus for deductive verification?* In this paper we give an affirmative answer. We define a specification language called *behavioral contracts for cooperative scheduling*, "cooperative contracts" in short, that addresses fundamental problems encountered when generalizing sequential to cooperative concurrent behavior:

1. Due to strong object encapsulation, the specification of interfaces and of implementations diverges. The latter know implementation aspects that the former have no access to.
2. In the time gap between method invocation and activation or between a method's termination and reading the returned result, as well as at suspension points, it is possible that different tasks on the same object interleave. It must be possible to specify *locally* at such points which tasks can be relied upon to have finished and which might overlap.
3. As futures can be passed around as parameters and thus may be read in arbitrary code locations, it is not possible to determine statically the method that computes a future's value. Again, this information must be specified.

In addition, cooperative contracts must permit to specify frames, similar to sequential contracts.

The second main contribution of this paper is a program logic in which active object programs together with their specification can be expressed. The third contribution is a compositional deductive verification system that permits to formally prove the validity of logically rendered contracts. That calculus is sound relative to a formal semantics which is supplied as well.

To present our framework we need to fix a concrete AO language. We chose a subset[3] of ABS ("abstract behavioral specification") [41], because it has a stable and well-maintained ecosystem and its sequential fragment is similar to Java so that cooperative contracts can be presented as an extension of JML [51].

The following section introduces and explains the usage of cooperative contracts by way of a case study: we formally specify a concurrent publisher-subscriber model with dynamic allocation of proxy servers. Sect. 3 formally defines syntax and semantics of our active object language. Sect. 4 defines *behavioral program logic* (BPL) [43], a first-order dynamic logic that can represent active object programs together with their behavioral specification. Cooperative contracts are then rendered in BPL, the details are given in Sect. 5. We close with related work (Sect. 6) and a conclusion (Sect. 7).

The main ideas for specification and verification with cooperative contracts were first presented in [48]. Their encoding into BPL was first explored in [45]. The present account is rewritten from scratch. The publisher-subscriber case study is completely new, as is the formal semantics of the active object language. In contrast to the prior publications, frames are added and handled uniformly with method contracts. The program logic and, as a result the deductive verification system and the proof sketches, is simplified in contrast to both prior publications.

2 Cooperative Contracts for Active Objects

We introduce the main concepts of the active object (AO) language ABS [2, 41] as well as the methodology of our specification framework by way of a case study.

2.1 Case Study

Consider a publisher–subscriber model, where clients may subscribe to a service, while the service object is responsible for generating news and publishes each news update to the subscribing clients. To avoid bottlenecks when publishing, the service delegates publishing to a chain of proxy objects, where each proxy handles a bounded number of clients, as illustrated in Fig. 1.

The interfaces and implementations of service and proxies are shown in Fig. 2. The ABS code is fully executable and the complete model is found at `abs-models.org`. The `publish()` method of the `Proxy` class takes advantage of asynchronous method calls with futures: a proxy object first initiates publication down the chain of proxies via asynchronous, recursive calls on the `nextProxy`

[3] The restriction to a subset is purely for presentation purposes. A forthcoming implementation will support full ABS.

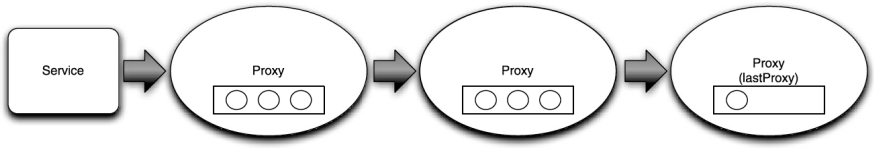

Fig. 1. A publisher-subscriber model with proxies handling at most three clients each

pointer, before it retrieves the news item `ns` and forwards that to its client list. If the current proxy object is the last in the chain, then the next news production cycle is initiated via a call to the `produce()` method of `Service`.

On the other hand, the recursive call in `add()` of `Proxy` is blocking: We use `r = o.m()` as a shorthand for `Fut<T> fut = o!m(); r = fut.get`. This is harmless since the implementation of `add()` does not deadlock and terminates after at most `limit` many calls.

Among several relevant properties one would like to specify for the `publish()` method are that the future passed to it is not null and that all clients of the current proxy object received the news update upon termination. In the following we illustrate that such properties can be easily and succinctly stated in our framework.

Recall that in the AO language ABS a task cannot be preempted, unless it is at a suspension point. The latter is marked explicitly by an **await** statement. Hence, the code between method activation, **await** statements, and method termination, respectively, runs uninterrupted. We speak of *atomic segments*. The scope of each atomic segment has a unique name, which is specified by the annotation `[atom: "name"]` at the **await** statement that closes it. The syntactic end of a method declares an implicit atomic segment whose name defaults to the method name. Therefore, a method without **await** statement has exactly one atomic segment named after the method. In addition, `sync` labels identify statements, where the value of a future is read. For example, the `publish()` method can be annotated as follows:

```
Unit publish(Fut<NewsI> fut) {
  NewsI ns = null;
  if (nextProxy != null) {
    nextProxy!publish(fut);
  }
  [sync: "getNews"]       ns = fut.get;
  [atom: "notifyClients"] await this!toClients(ns);
  if (nextProxy == null) {
    service!produce();
  }
}
```

```
1  interface ServiceI {
2    Unit setup(Int lm, ProducerI prod);
3    Unit subscribe(ClientI cl);
4    Unit produce();
5  }
6
7  class Service implements ServiceI {
8    ProducerI producer = null;
9    ProxyI proxy = null;
10   ProxyI lastProxy = null;
11   Int limit = 0;
12
13   Unit setup(Int lm, ProducerI prod) {
14     limit = lm;
15     producer = prod;
16     proxy = new Proxy();
17     proxy.setup(lm, this);
18     lastProxy = proxy;
19     this!produce();
20   }
21
22   Unit subscribe(ClientI cl) {
23     lastProxy = lastProxy.add(cl);
24   }
25
26   Unit produce() {
27     Fut<NewsI> f_news =
28       producer!detectNews();
29     await f_news?;
30     proxy!publish(f_news);
31   }
32 }
33
34 interface ProxyI {
35   Unit setup(Int lm, ServiceI s);
36   ProxyI add(ClientI cl);
37   Unit publish(Fut<NewsI> fut);
38 }
39
40 class Proxy implements ProxyI {
41   Int limit = 0;
42   List<ClientI> myClients = Nil;
43   ProxyI nextProxy = null;
44   ServiceI service = null;
```

```
45   Unit setup(Int lm, ServiceI s) {
46     limit = lm;
47     service = s;
48   }
49
50   ProxyI add(ClientI cl) {
51     ProxyI lastProxy = this;
52     if (length(myClients) < limit) {
53       myClients = append(myClients, cl);
54     } else {
55       if (nextProxy == null) {
56         nextProxy = new Proxy();
57         nextProxy.setup(limit, service);
58       }
59       lastProxy = nextProxy.add(cl);
60     }
61     return lastProxy;
62   }
63
64   Unit publish(Fut<NewsI> fut) {
65     NewsI ns = null;
66     if (nextProxy != null) {
67       nextProxy!publish(fut);
68     }
69     ns = fut.get;
70     await this!toClients(ns);
71     if (nextProxy == null) {
72       service!produce();
73     }
74   }
75
76   Unit toClients(NewsI ns) {
77     Int cnt = 0;
78     while (cnt < length(myClients)) {
79       ClientI cl = nth(myClients, cnt);
80       cl!signal(ns);
81       cnt = cnt + 1;
82     }
83   }
84 }
```

Fig. 2. Implementation of service and proxy interfaces and classes

2.2 Specifying State in an Asynchronous Setting

During the time gap between method invocation and activation, method parameters stay invariant (they are immutable in ABS), but the object's heap (value of fields) may change. This motivates to split the precondition of asynchronous method contracts into an interface part relating to method parameters and an implementation part that specifies heap values. The *parameter precondition* is guaranteed by the caller who knows the appropriate synchronization pattern. It is part of the callee's interface declaration and exposed to clients. (For parameterless methods the parameter precondition is `True` and can be omitted.) Vice

versa, the *callee* guarantees the heap precondition. This is so, because only the callee's implementation has sufficient knowledge of the internal state, Therefore, the heap precondition is declared in the class implementing an interface and not exposed to clients. Preconditions follow JML [51] convention and are marked by the *requires* keyword.

Postconditions (keyword *ensures*) work analogously. Like in JML, postconditions are evaluated in the poststate of a method. To access the value of a field at *activation* time of a method, the keyword *\old* is used.

In addition, like in JML, the implementation contract of a method specifies its *assignable* heap locations. This specifies either a set of memory locations or the shortcuts *\nothing*, *\everything* with obvious semantics. It is advisable to specify assignable locations, because their (safe) default is *\everything*.

Example 1. Parameters of `Proxy.setup()` must fulfill the precondition that the capacity `lm` of a proxy is larger than zero and the service object `s` is not null. The heap precondition expresses that field `limit` is zero and `service` is null. The heap postcondition expresses that after termination of `setup()` the value of `limit` is larger than zero and `service` is not null.

```
interface ProxyI {
  /*@ requires lm > 0 & s != null @*/
  Unit setup(Int lm, ServiceI s);
}
```

```
class Proxy implements ProxyI {
  /*@ requires limit == 0 && service == null;
      ensures  limit > 0   && service != null;
      assignable {limit, service}; @*/
  Unit setup(Int lm, ServiceI s) {
    limit = lm;
    service = s;
  }
}
```

2.3 Concurrency Context

A caller must fulfill the callee's parameter precondition, but the most recently completed process running on the callee object establishes its heap precondition. In other words, whether the callee's heap precondition holds, depends on the postconditions of the (atomic segments of) methods that run concurrently or that have just terminated. To express this, one specifies the *concurrency context* of a method, in addition to its *memory context* given by the heap precondition. The concurrency context is part of the interface contract and consists of two *context sets*, i.e. sets of atomic segment names:

- *succeeds*: Each atomic segment in this set must *ensure* the heap precondition when it terminates and at least one of them must have run before the specified method starts execution.
- *overlaps*: Each atomic segment in this set 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, but a class may extend context sets with private atomic segment names and methods. 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. Observe that context sets represent global information unavailable when a method is analyzed in isolation. If context sets are not specified in the code, they default to the set of *all* atomic segments in the class, whence the heap precondition degenerates into a class invariant and must be guaranteed by each process at each suspension point [27].

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 [38]. This enforces a separation of concerns in specifications: method contracts are local and specify a single method and its context; the coordination language specifies a global view on the whole protocol. Of course, local method contracts and global protocols expressed with session types [46, 47] must be proven consistent. Context sets can also be verified by static analysis once the whole program is available, see Sect. 2.6 for an example.

Example 2. Consider the interface and implementation specification of `add()` in Fig. 3. The heap precondition of `add()` is established by `setup()` which it must succeed. Between `setup()` and (re-)activation of `add()` only `add()` or `publish()` may run. This is expressed by the specified context sets (recall that method names label the final atomic segment of a method body).

The heap postcondition specifies that `add()` may increase the length of field `myClients`, but it will never exceed `limit`. It preserves the contents of `myClients`. If the `limit` was not reached at the beginning of method execution, then client `cl` is indeed in `myClients` upon method termination. In addition, we extend the context set of the interface specification with atomic segment `notifyClients` in the `publish()` method and the private method `toClients()` which also may interleave with `add()`.

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

Example 3. Continuing Expl. 2, after propagation, contracts of `setup()`, `add()` and `toClients()` are as follows[4] (redundant expressions not shown, for example, *limit > 0 && service != null* in `add()`'s postcondition are preserved from its precondition, because of the *assignable* clause):

```
/*@ ensures <as before> && len(myClients)<=limit @*/
Unit setup(Int lm, ServiceI s) { ... }
/*@ ensures <as before> && len(myClients)<=limit @*/
ProxyI add(ClientI cl) { ... }
/*@ requires <as before> && limit > 0 && len(myClients) <= limit && service != null;
    ensures <as before> && limit > 0 && len(myClients) <= limit && service != null
@*/
Unit toClients(NewsI ns) { ... }
```

[4] The specifications of `publish()` and `notifyClients` are shown in the next subsection in connection with specification of suspension points.

```
interface ProxyI {
  /*@ requires cl != null;
      succeeds {setup};
      overlaps {add, publish} @*/
  ProxyI add(ClientI cl);
}

class Proxy implements ProxyI {
  /*@ requires limit > 0 && len(myClients) <= limit && service != null;
      ensures 0 < len(myClients) && len(myClients) >= len(\old(myClients)) &&
              ∀ ClientI c; hasElement(\old{myClients}, c); hasElement(myClients, c) &&
              (len(\old(myClients)) < limit -> hasElement(myClients, cl)) &&
              (len(\old(myClients)) == limit -> len(myClients) == len(\old(myClients));
      overlaps {notifyClients, toClients};
      assignable {myClients, nextProxy} @*/
  ProxyI add(ClientI cl) {
    ProxyI lastProxy = this;
    if (length(myClients) < limit){
      myClients = append(myClients, cl);
    } else {
      if (nextProxy == null) {
        nextProxy = new Proxy();
        nextProxy.setup(limit, service);
      }
      lastProxy = nextProxy.add(cl);
    }
    return lastProxy;
  }
}
```

Fig. 3. Interface and implementation of `add()` in proxy

2.4 Resolve Contracts

Consider the **get** statement in Line 69 of Fig. 2. It is important to be able to prove properties about its resolved future, for example, `ns != null`. Such a property can be ensured by the postcondition of the method that computed the future, but (like in the example) it is not obvious which methods that could be. For this reason we attach a *resolve contract* to each **get** statement. It consists simply of the keyword *resolvedby*, followed by the set of methods that resolve its future.

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

Example 4. The **get** statement in Line 69 of Fig. 2 is resolved by `Producer.detectNews()`, the resolve contract is displayed in Fig. 4. Assuming that the interface postcondition of `Producer.detectNews()` contains *\result != null*, we can ensure that `ns != null` holds at this point.

2.5 Suspension Contracts

Each **await** statement introduces a scheduling point, where task execution may be suspended and possibly interleaved. From a local perspective, an **await** statement is a *suspension point* where information about heap memory is lost. This is similar to heap preconditions and can be addressed in the same manner: By specifying what is guaranteed at release of control, what can be assumed upon reactivation, and who has the obligation to guarantee the heap property. Accordingly, each suspension point is annotated by a *suspension contract* containing the *same* elements as a method contract: An *ensures* clause for the condition that is guaranteed upon suspension, a *requires* clause for the condition that must hold upon reactivation[5], a *succeeds* context set for the atomic segments that must have run before reactivation and an *overlaps* context set for atomic segments whose execution may interleave. The *assignable* clause always relates to the atomic segment whose suspension contract it is part of.

```
interface ProxyI {
  /*@ requires fut != null;
      succeeds {setup};
      overlaps {add, publish} @*/
  Unit publish(Fut<NewsI> fut);
}

class Proxy implements ProxyI {
  /*@ ensures  len(myClients) <= limit ;
      assignable \nothing; @*/
  Unit publish(Fut<NewsI> fut) {
    NewsI ns = null;
    if (nextProxy != null) { nextProxy!publish(fut); }
    /*@ resolvedby {Producer.detectNews}; @*/
    [sync: "getNews"] ns = fut.get;
    /*@ ensures ns != null && len(myClients) <= limit ;
        requires this.service != null;
        succeeds {toClients};
        overlaps {add, publish};
        assignable \nothing; @*/
    [atom: "notifyClients"] await this!toClients(ns);
    if (nextProxy == null) { service!produce(); }
  }
}
```

Fig. 4. Suspension and resolve contracts of `publish()` in proxy

Example 5. In Fig. 4 we specify the behavior of the suspension point at the **await** statement `"notifyClients"`: Upon suspension, the news `ns` passed to `toClients()` must not be null which can be derived from the resolve contract (see Sect. 2.4).

[5] The execution pattern is *inverse* to method execution: on suspension execution stops and is later reactivated. Therefore, it is intuitive to specify *ensures before requires*. Observe that the *ensures* clause of the top-level method contract specifies the final state upon method termination, i.e. the postcondition of the final atomic segment.

During suspension, only methods `add()`, `publish()` can interleave. By adding the method `toClients()` to the `succeeds` set, we ensure that the news will have been delivered to the clients upon reactivation.

Propagation from context sets into pre- and postconditions of suspension contracts is analogous to the procedure for method contracts. Fig. 4 highlights the propagated specification for `publish()` from the heap precondition of `add()`. The name "`publish`" represents the *final* atomic segment of the `publish()` method, so the postcondition of `publish()` is also the postcondition of that atomic segment. After propagation, the contracts express that both atomic blocks preserve the heap precondition of `add()`.

2.6 Service 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 postconditions of called methods, it is sufficient to analyze its owner class.

However, the correctness of context sets in the cooperation contracts clearly depends on the sequence in which methods are called by client classes. In other words, to verify that a program respects all context sets, requires global, system-wide information. This constitutes a separation of concerns between functional specification (state) and non-functional specification (call sequence). That makes it possible to decompose concurrent system verification into two phases: In the first phase, deductive verification [24] is used to *locally* show that each individual method implements its pre- and postconditions correctly. In the second phase, a *global* light-weight, fully automatic dependence analysis [7] is used to approximate possible call sequences. This approach has two advantages: first, the global analysis need not be done by expensive deduction; second, it is often possible to reuse contracts: if a method is changed with only local effects it is sufficient to re-prove its contract and re-run the dependence analysis. The proofs of all other method contracts remain unchanged.

The dependence analysis of context sets is detailed in the technical report [49]; we only give an example for rejected and accepted call sequences here.

Example 6. Consider the code fragments interacting with a `Proxy` instance `p` given below (use `await o!m();` as shorthand for `Fut<T> f = o!m(); await f?;`). The left fragment fails to verify the context sets specified above: even though called after `setup()`, `publish()` can be executed first due to reordering, failing its *succeeds* clause. The right fragment verifies, because of the `await` guard and the fact that `publish()` is included in its *overlaps* context set.

```
p!setup(3,s);
p!publish(f);
```

```
await p!setup(3,s);
p!publish(f1);
p!publish(f2);
```

3 Active Object Programs

We define formally the simple active object language Async, based on ABS [41], first the syntax, then the formal semantics.

3.1 Syntax

The Async syntax is shown in Fig. 5. The language has been informally explained with an extensive example in the previous section, so we keep this brief and focus on language features related to communication and synchronization—other features are standard.

Objects communicate by asynchronous method calls, written $\mathtt{e!m}(\overline{\mathtt{e}})$, with an associated future f. A future's value can be accessed by the statement $x = f.\mathbf{get}$ once it is resolved, i.e. when the task associated with f terminated. Futures can be shared among objects. Field access between different objects is indirect through (getter/setter) method calls, amounting to strong encapsulation. Cooperative scheduling is realized 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 inside atomic segments is sequential and cannot be preempted. For the set `Exp` of pure expressions we assume the valuation function to be known, where $\mathtt{e} \in \mathtt{Exp}$ is a value of `Exp`. We assume programs are well-typed.

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

Fig. 5. Syntax of the Async language

Compared to ABS, Async features optional `atom` 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.

3.2 Trace Semantics

We define a layered denotational semantics for Async, inspired by locally abstract globally concrete (LAGC) trace semantics [26]. The purpose of the *local* layer is to associate with each initial state and each Async statement a *set* of traces that record not only each possible evolution of computation states, but also synchronization events (call, suspension, future resolution, etc.). A set of traces is needed, because, locally, the value of, for example, a resolved future cannot be known. The global layer combines for a given program the local traces into all

possible traces such that the involved events and states correspond to a permitted task interleaving in Async. Main advantages of this semantics include (a) a modular separation between sequential and concurrent execution and (b) close correspondence to the rules of the verification calculus for Async.

3.2.1 States and Traces. Let `Object` and `Future` denote the sets of possible object and future identifiers in the semantics, with values o and f, respectively. Let `Var` denote the set of program variables (fields, local variables), `Val` the set of expression values `v` in Async and `State` the states, i.e., the mappings from `Var` to `Val`. Traces combine states $\sigma \in$ `State` with the following events, which capture synchronization and communication between active objects:

$$
\begin{aligned}
ev(\overline{e}) ::= \ & \mathsf{invEv}(o_1, o_2, f, \mathtt{m}, \overline{\mathtt{v}}) \mid \mathsf{invREv}(o_1, o_2, f, \mathtt{m}, \overline{\mathtt{v}}) \\
\mid \ & \mathsf{futEv}(o, f, \mathtt{m}, \mathtt{v}) \mid \mathsf{futREv}(o, f, \mathtt{v}, i) \\
\mid \ & \mathsf{suspBoolEv}(o, f, \mathtt{m}, i) \mid \mathsf{reacBoolEv}(o, f, \mathtt{m}, i) \\
\mid \ & \mathsf{suspFutEv}(o, f, \mathtt{m}, f', i) \mid \mathsf{reacFutEv}(o, f, \mathtt{m}, f', i) \\
\mid \ & \mathsf{blkEv}(o) \mid \mathsf{blkREv}(o) \\
\mid \ & \mathsf{newEv}(o_1, o_2, C, \overline{\mathtt{v}})
\end{aligned}
$$

Most events are organized as duals above, with one pair in each line. The event $\mathsf{invEv}(o_1, o_2, f, \mathtt{m}, \overline{\mathtt{v}})$ captures the asynchronous invocation of a method by an object o_1 to a method `m` of object o_2 with actual parameters $\overline{\mathtt{v}}$; the future f is the identifier of the future to which the return value from the method will be sent. The dual event $\mathsf{invREv}(o_1, o_2, f, \mathtt{m}, \overline{\mathtt{v}})$ represents the activation of the called method. The event $\mathsf{futEv}(o, f, \mathtt{m}, \mathtt{v})$ captures the resolution of the future with identifier f; here, o is the called object, `m` the called method and `v` the return value from the call which is passed to the future f. The dual event $\mathsf{futREv}(o, f, \mathtt{v}, i)$ represents the synchronization on this future by object o; note that i here captures the `sync` *annotation* on **get**-statements. The event $\mathsf{suspBoolEv}(o, f, \mathtt{m}, i)$ captures the suspension of the active method `m` identified by the future f in object o on a Boolean condition; i captures the `atom` *annotation* on **await**-statements. The dual event $\mathsf{reacBoolEv}(o, f, \mathtt{m}, i)$ represents the scheduling (or reactivation) of the method after suspension. Similarly, event $\mathsf{suspFutEv}(o, f, \mathtt{m}, f', i)$ captures suspension while waiting for a future f' to be resolved, with the dual event $\mathsf{reacFutEv}(o, f, \mathtt{m}, f', i)$. The event $\mathsf{newEv}(o_1, o_2, C, \overline{\mathtt{v}})$ captures the creation by an object o_1 of a new object o_2 of class C with actual parameters $\overline{\mathtt{v}}$.

The semantics ultimately maps programs into sets of traces over states and events, but in the local layer synchronization points are not yet resolved. For this reason, *local* traces may contain the symbol ⨾ which indicates points where interleaving or blocking can take place. Accordingly, we define another pair of events $\mathsf{blkEv}(o)$ and $\mathsf{blkREv}(o)$ to capture the blocking of an object o at a **get**-statement while *global* interleaving happens.

Definition 1 (Traces, Local Traces). *Let $\sigma \in$ `State` be a mapping from variables to values and $ev(\overline{e})$ be an event. A* trace τ *is defined by the following productions:*

$$\tau ::= \varepsilon \mid \tau \curvearrowright t$$
$$t ::= \sigma \mid ev(\overline{e})$$

A local trace π *is defined by the following production:*

$$\pi ::= \tau \mid \tau \fatsemi \pi$$

Here, ε denotes the empty trace. We only consider finite traces in this work. Let $\langle\sigma\rangle$ denote the *singleton trace* $\varepsilon \curvearrowright \sigma$. *Concatenation* of two traces τ_1, τ_2 is written as $\tau_1 \cdot \tau_2$ and only defined when τ_1 is finite. The final state of a non-empty, finite trace τ is obtained by $\mathrm{last}(\tau)$, and the first state of a non-empty trace τ is $\mathrm{first}(\tau)$; we lift these operations to local traces π in the obvious manner.

Since we use (local) traces to give a semantics to compound program statements, it is convenient to have an operator on (local) traces that reflects sequential composition. The technical issue it solves is as follows: Assume that τ_1 is a trace of a statement s_1 and τ_2 a trace of another statement s_2. To obtain the trace corresponding to the sequential composition of s_1, s_2, the last state of τ_1 and the first state of τ_2 must be identical, but the resulting trace should not contain a repeated state. This motivates the *semantic chop* operator $\underline{**}$ on local traces (inspired by [55]):

Definition 2 (Chop on Traces). *Let τ_1, τ_2 be non-empty traces, π_1, π_2 non-empty local traces and assume that τ_1, π, π_1 are finite. The semantic chop on traces is defined inductively as follows:*

$$\tau_1 \mathbin{\underline{**}} \tau_2 = \tau_1 \cdot \tau' \text{ where } \mathrm{last}(\tau_1) = \sigma,\ \tau_2 = \langle\sigma\rangle \cdot \tau'$$
$$(\pi \fatsemi \tau_1) \mathbin{\underline{**}} \tau_2 = \pi \fatsemi (\tau_1 \mathbin{\underline{**}} \tau_2)$$
$$(\pi_1 \fatsemi \tau_1) \mathbin{\underline{**}} (\tau_2 \fatsemi \pi_2) = ((\pi_1 \fatsemi \tau_1) \mathbin{\underline{**}} \tau_2) \fatsemi \pi_2$$

3.2.2 Local Semantics. We define the semantics of methods bottom-up, starting with individual statements, by a validation function $\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(s)$ that returns a set of the possible traces when executing a statement s on the current object X, where the future f is associated with the current instance of method $\mathtt{m}$, starting in initial state σ. We assume that a standard valuation function $\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(e)$ for side effect-free expressions e is given. Let the function $\widehat{C}(C, o, \overline{\mathtt{v}})$ return the initial state of an object o of class C with constructor parameters $\overline{\mathtt{v}}$, let $\widehat{M}(C, \mathtt{m}, \overline{\mathtt{v}})$ the initial local state of a method $\mathtt{m}$ of class C with actual parameters $\overline{\mathtt{v}}$, and $M(C, \mathtt{m})$ the body of a method $\mathtt{m}$ of class C.

The local trace semantics for statements is given in Fig. 6. The semantics of **skip** in state σ is the singleton trace $\langle\sigma\rangle$. An assignment results in a trace with two states where the latter updates σ with the new binding for the assigned variable. Conditionals and **while**-loops introduce two sets of local traces depending on whether the condition evaluates to true or false; here, only one of these sets can be non-empty.

Object creation introduces an event to reflect the creation of the new object and extends the state with the fields of the new object. We don't know the object

$$\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\textbf{skip}) = \{\langle\sigma\rangle\}$$

$$\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(x = e) = \{\langle\sigma\rangle \curvearrowright \sigma[x \mapsto \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(e)]\}$$

$$\begin{aligned}&\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\textbf{if}\,(\mathtt{e})\,\{\mathtt{s}_1\}\,\textbf{else}\,\{\mathtt{s}_2\})\\ &\quad= \{\pi \in \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\mathtt{s}_1) \mid \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(e)\} \cup \{\pi \in \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\mathtt{s}_2) \mid \neg\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(e)\}\end{aligned}$$

$$\begin{aligned}&\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\textbf{while}\,(\mathtt{e})\,\{\mathtt{s}\})\\ &\quad= \{\langle\sigma\rangle \mid \neg\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(e)\} \cup \{\pi \in \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\overline{\mathtt{s}};\,\textbf{while}\,(\mathtt{e})\,\{\mathtt{s}\}) \mid \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(e)\}\end{aligned}$$

$$\begin{aligned}&\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(x = \textbf{new}\ \mathtt{C}(\overline{\mathtt{e}}))\\ &\quad= \{\langle\sigma\rangle \curvearrowright \mathsf{newEv}(\mathsf{X}, o, C, \overline{v}') \curvearrowright \sigma \circ \widehat{C}(C, o, \overline{v}') \mid o \in \mathtt{Object} \wedge \overline{v}' = \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\overline{e})\}\end{aligned}$$

$$\begin{aligned}&\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(x = \mathtt{e}!\mathtt{m}(\overline{\mathtt{e}}))\\ &\quad= \{\langle\sigma\rangle \curvearrowright \mathsf{invEv}(\mathsf{X}, \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\mathtt{e}), f', \mathtt{m}, \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\overline{\mathtt{e}})) \curvearrowright \sigma[x \mapsto f'] \mid f' \in \mathtt{Future}\}\end{aligned}$$

$$\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\textbf{return}\ \mathtt{e}) = \{\langle\sigma\rangle \curvearrowright \mathsf{futEv}(\mathsf{X}, f, \mathtt{m}, \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(\mathtt{e})) \curvearrowright \sigma\}$$

$$\begin{aligned}&\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}([\mathtt{sync} : \text{“}i\text{”}]\ x = \mathtt{e}.\textbf{get})\\ &\quad= \{\langle\sigma\rangle \curvearrowright \mathsf{blkEv}(\mathsf{X}) \curvearrowright \sigma \fatsemi\\ &\qquad\langle\sigma'\rangle \curvearrowright \mathsf{blkREv}(\mathsf{X}) \curvearrowright \sigma' \curvearrowright \mathsf{futREv}(\mathsf{X}, \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma'}(\mathtt{e}), \mathtt{v}, i) \curvearrowright \sigma'[x \mapsto \mathtt{v}]\\ &\qquad\mid \mathtt{v} \in \mathtt{Val} \wedge \sigma' \in \mathtt{State} \wedge \forall \mathsf{X}.x \cdot \sigma(\mathsf{X}.x) = \sigma'(\mathsf{X}.x) \wedge \forall f.x \cdot \sigma(f.x) = \sigma'(f.x)\}\end{aligned}$$

$$\begin{aligned}&\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}([\mathtt{atom} : \text{“}i\text{”}]\ \textbf{await}\ \mathtt{e})\\ &\quad= \{\langle\sigma\rangle \curvearrowright \mathsf{suspBoolEv}(\mathsf{X}, f, \mathtt{m}, i) \curvearrowright \sigma \fatsemi \langle\sigma'\rangle \curvearrowright \mathsf{reacBoolEv}(\mathsf{X}, f, \mathtt{m}, i) \curvearrowright \sigma'\\ &\qquad\mid \sigma' \in \mathtt{State} \wedge \forall f.x \cdot \sigma(f.x) = \sigma'(f.x) \wedge \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma'}(e)\}\end{aligned}$$

$$\begin{aligned}&\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}([\mathtt{atom} : \text{“}i\text{”}]\ \textbf{await}\ \mathtt{e}?)\\ &\quad= \{\langle\sigma\rangle \curvearrowright \mathsf{suspFutEv}(\mathsf{X}, f, \mathtt{m}, f', i) \curvearrowright \sigma \fatsemi \langle\sigma'\rangle \curvearrowright \mathsf{reacFutEv}(\mathsf{X}, f, \mathtt{m}, f', i) \curvearrowright \sigma'\\ &\qquad\mid \sigma' \in \mathtt{State} \wedge \forall f.x \cdot \sigma(f.x) = \sigma'(f.x) \wedge f' = \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(e)\}\end{aligned}$$

$$\begin{aligned}&\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(C.\mathtt{m})\\ &\quad= \{(\langle\sigma\rangle \curvearrowright \mathsf{invREv}(\mathsf{X}', \mathsf{X}, f, \mathtt{m}, \overline{\mathtt{v}}) \curvearrowright \sigma) \mathbin{\underline{**}} \pi\\ &\qquad\mid \mathsf{X}' \in \mathtt{Object} \wedge \overline{\mathtt{v}} \in \overline{\mathtt{Val}} \wedge \pi \in \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma \circ \widehat{M}(\mathtt{m},f,\overline{\mathtt{v}})}(M(C,\mathtt{m}))\}\end{aligned}$$

$$\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(s_1; s_2) = \{\pi_1 \mathbin{\underline{**}} \pi_2 | \pi_1 \in \mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(s_1) \wedge \pi_2 \in \mathrm{val}_{\mathsf{X},\mathtt{m},f,\mathrm{last}(\pi_1)}(s_2)\}$$

Fig. 6. Local trace semantics for Async.

identifier o in the local semantics, hence it ranges over `Object`. Asynchronous method calls introduce an event to reflect method invocation, where f' ranges over `Future`, and **return** introduces a future resolution event into the trace.

The trace corresponding to the **get**-statement introduces a future reaction event and assigns the return value `v` to the corresponding variable. Since the value `v` fetched from the future is unknown, the traces range over the possible value for `v`. Observe that the fields of the object and the local variables of the method are unchanged when the statement is executed but that the state may otherwise change, reflecting that **get** is blocking on the object until it can execute, but other objects may execute in the meantime. We use $o.x$ to denote a field x of object o and likewise $f.x$ to denote a local variable of the method identified by the future f. In the semantics of the **await** e statement, the local traces are split between the suspension and reactivation events, with the corresponding state

change to $\sigma' \in \texttt{State}$; here, local variables are unchanged between these events, reflecting that the process is suspended. The condition $\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma'}(e)$ expresses that the guard must hold in state σ' for the process to be rescheduled. The **await** e? statement has a similar semantics, but the condition for reactivating the process will be captured in the well-formedness condition on traces.

The semantics of a method $\mathtt{m}$ of class C is given by an invocation reaction event for that method, composed with the semantics of its method body (in the state extended with the local variables). The calling object X' cannot be known and ranges over `Object`. Sequential composition is directly captured by the semantic chop operator.

3.2.3 Initial Configuration and Global Semantics. For a given program Pr in Async with main block $\{T_1\ x_1 = \mathtt{e}_2 \ldots, T_n\ x_n = \mathtt{e}_n; \mathtt{s}\}$, let $\mathit{classes}(Pr)$ denote the set of class names in Pr, $\mathit{methods}(C)$ the set of method names declared in class C and $\sigma_{main} \in \texttt{State}$ a mapping from the program variables $x_1, \ldots, x_n$ to some concrete (default) values of the corresponding types $T_1, \ldots, T_n$. For a future $f \in \texttt{Future}$, define the possible local traces of a program Pr to compute f by the following set:

$$\begin{aligned}\mathit{traces}(f, Pr) = \{\pi \mid \pi \in \mathrm{val}_{o,\mathtt{m},f,\sigma}(C.\mathtt{m}) \wedge o \in \texttt{Object} \wedge \sigma \in \texttt{State} \\ \wedge\ C \in \mathit{classes}(Pr) \wedge \mathtt{m} \in \mathit{methods}(C)\}\end{aligned}$$

Let $\mathcal{T}$ denote a set of sets of local traces. One such set is the set of all possible local trace sets for any future for a program Pr, defined by

$$\begin{aligned}\mathcal{T}_{Pr} = \{\mathit{traces}(f, Pr) \mid f \in \texttt{Future}\} \\ \cup \{\mathrm{val}_{o_{main},\mathtt{m}_{main},f_{main},\sigma_{main}}(x_1 = e_1; \ldots; x_n = \mathtt{e}_n; s)\}\ .\end{aligned}$$

The *initial configuration* of a program Pr is defined as the pair $\langle \sigma_{main} \rangle, \mathcal{T}_{Pr}$.

Let sh denote a *global trace*, i.e. a trace without synchronization markers "$\wp$". Global traces are produced by a transition system on configurations $sh, \mathcal{T}$ by means of the following rule:

$$(\textsc{Global})\quad \frac{\begin{array}{ccc}\tau \wp \pi \in \Omega & \Omega \in \mathcal{T} & \mathrm{last}(sh) = \mathrm{first}(\tau) \\ \mathit{wf}(sh \mathbin{\underline{**}} \tau) & \Omega' = \{\pi' \mid \tau \wp \pi' \in \Omega\} & \mathcal{T}' = (\mathcal{T} \setminus \Omega) \cup \Omega'\end{array}}{sh, \mathcal{T} \rightarrow sh \mathbin{\underline{**}} \tau, \mathcal{T}'}$$

The intuition behind rule (Global) is that only a local trace that happens to match the last state of the global trace sh can be selected to extend sh. The selection applies to the first segment of the local trace, after which all possible successor segments from $\mathit{traces}(f, Pr)$ are kept and all other possibilities for f are rejected in $\mathcal{T}'$. Whereas trace semantics for active objects which take a *local* perspective [5, 25] require the composition rule to enforce that local variables remain invariant over processor release points, our work takes a *global* perspective by letting the local trace sets range over all possible states. As a consequence, trace composition in (Global) can be captured directly by the semantic chop operator, which matches the states when composing two traces.

In addition to this matching on states, only those local traces can be selected that conform to the well-formedness predicate $wf(sh \underline{**} \tau)$ over traces. Well-formedness imposes constraints on the ordering of events, expressing legal scheduling and synchronization during execution. For a global trace sh, let $\mathit{Objects}(sh)$ and $\mathit{Futures}(sh)$ denote the sets of objects and futures that are introduced by the events reflecting object creation and asynchronous method calls in the semantics (including o_{main} and f_{main}). Let $sh|_f$ denote the projection of sh to the trace of events involving the process with future f. Let $sh|_o$ denote the projection of sh to the trace of events of object o and $sh\ ew\ ev$ that ev is the final event occurring in sh ("ends with"). These functions all have straightforward inductive definitions. The well-formedness predicate $wf(sh)$ itself is defined inductively in Fig. 7. We abbreviate existentially quantified variables whose value is irrelevant with "_". One can assume a trace to end with an event, because the states after the last event in a trace do not affect well-formedness.

$$wf(\sigma_{main}) = \mathit{true}$$

$$wf(sh \frown \mathsf{invEv}(o_1, o_2, f, \mathtt{m}, \overline{\mathsf{e}})) = \{o_1, o_2\} \subseteq \mathit{Objects}(sh) \wedge f \notin \mathit{Futures}(sh) \wedge wf(sh)$$

$$wf(sh \frown \mathsf{invREv}(o_1, o_2, f, \mathtt{m}, \overline{\mathsf{e}})) = sh|_f\ ew\ \mathsf{invEv}(o_1, o_2, f, \mathtt{m}, \overline{\mathsf{e}}) \wedge wf(sh)$$

$$\begin{aligned} wf(sh \frown \mathsf{futEv}(o_2, f, \mathtt{m}, \mathsf{e})) = {} & \mathsf{invREv}(\_, o_2, f, \mathtt{m}, \_) \in sh \\ & \wedge\ \mathsf{futEv}(o_2, f, \mathtt{m}, \_) \notin sh \wedge wf(sh) \end{aligned}$$

$$wf(sh \frown \mathsf{futREv}(\_, f, \mathsf{e}, \_)) = \mathsf{futEv}(\_, f, \_, \mathsf{e}) \in sh \wedge wf(sh)$$

$$\begin{aligned} wf(sh \frown \mathsf{suspBoolEv}(o_2, f, \mathtt{m}, i)) = {} & sh|_f\ ew\ ev \\ & \wedge\ ev \in \{\mathsf{invREv}(\_, o_2, f, \mathtt{m}, \_), \mathsf{reacBoolEv}(o_2, f, \mathtt{m}, \_), \mathsf{reacFutEv}(o_2, f, \mathtt{m}, \_, \_)\} \\ & \wedge\ wf(sh) \end{aligned}$$

$$wf(sh \frown \mathsf{reacBoolEv}(o, f, \mathtt{m}, i)) = sh|_f\ ew\ \mathsf{suspBoolEv}(o, f, \mathtt{m}, i) \wedge wf(sh)$$

$$\begin{aligned} wf(sh \frown \mathsf{suspFutEv}(o_2, f, \mathtt{m}, f', i)) = {} & sh|_f\ ew\ ev \\ & \wedge\ ev \in \{\mathsf{invREv}(\_, o_2, f, \mathtt{m}, \_), \mathsf{reacBoolEv}(o_2, f, \mathtt{m}, \_), \mathsf{reacFutEv}(o_2, f, \mathtt{m}, \_, \_)\} \\ & \wedge\ wf(sh) \end{aligned}$$

$$\begin{aligned} & wf(sh \frown \mathsf{reacFutEv}(o_2, f, \mathtt{m}, f', i)) = \\ & \quad \mathsf{futEv}(\_, f', \_, \_) \in sh \wedge sh|_f\ ew\ \mathsf{suspFutEv}(o_2, f, \mathtt{m}, f', i) \wedge wf(sh) \end{aligned}$$

$$wf(sh \frown \mathsf{blkREv}(o)) = sh|_o\ ew\ \mathsf{blkEv}(o) \wedge wf(sh)$$

$$wf(sh \frown \mathsf{newEv}(o_1, o_2, C, \overline{x}, \overline{e})) = o_1 \in \mathit{Objects}(sh) \wedge o_2 \notin \mathit{Objects}(sh) \wedge wf(sh)$$

Fig. 7. Well-formedness predicate over traces

It is easy to see that the well-formedness predicate enforces that only the main block can be selected initially, as all other local traces are excluded by the well-formedness condition, that methods can only start after they have been called, that suspension events can only occur on executing processes, etc.

3.3 Semantic Logic

In the following, we aim to state and prove properties about all possible local traces of a method or a statement in a method. To formalize this, it is convenient to allow some reflection of the trace semantics into a logical language. We call this a *semantic* logic, its models are traces. In such a logic it is natural to relate indices in traces to states and events. The construct to relate traces and states has the form $[k] \vdash \varphi$, where k is an index term, denoting the k-th element of the trace that is the current model, and φ is a first-order formula over *states*. Here "$\vdash$" is a symbol of the logic. The meaning is that in the k-th state of a model trace the predicate φ holds. Similarly, traces and events are related by $[k] \doteq t$ which says that the k-th element of the model trace is equal to the event modeled by term t.

Example 7. The following formula in semantic logic expresses that there is a future resolve event that returns `Unit` and in the preceding state the field `fl` was positive. The type $\mathbf{I}$ models trace indices, `Method` is the set of all method names.

$$\exists\, k \in \mathbf{I}.\ \exists\, \mathtt{m} \in \mathtt{Method}.\ \exists\, o \in \mathtt{Object}.\ \exists f \in \mathtt{Future}.$$
$$[k] \doteq \mathsf{futEv}(o, f, \mathtt{m}, \mathtt{Unit}) \wedge [k-1] \vdash \mathbf{this}.\mathtt{fl} > 0$$

The formal definition of semantic logic is straightforward, for a full treatment, we refer to [44]. Given a statement s, we assume from now on to implicitly know the method that contains it, the object it is executed on and the future is resolves. Thus we abstract the evaluation function $\mathrm{val}_{\mathsf{X},\mathtt{m},f,\sigma}(s)$ defined Sect. 3.2.2 and write $[\![s]\!]_\sigma$ for the set of all local traces starting in σ.

4 Behavioral Program Logic

We state the verification system as behavioral contracts in Behavioral Program Logic (BPL) [43]. This simplifies the calculus compared to [48], as well as interaction with external static analyses.

BPL is a first-order dynamic logic with a *state*-based semantics: Its models are single states inside a Kripke structure, as in standard dynamic logics such as JavaDL [14]. Nonetheless BPL allows to verify *trace* properties of statements. To enable this, the representation and semantics of trace properties are separated and encapsulated in *behavioral modalities*. A behavioral modality can either be handled by a validity calculus, ensuring that every trace of a given statement follows the specification, or it can be passed to an external analysis. Hence, a BPL *behavioral specification* consists of at least two elements: syntax and semantics. If it also has a validity calculus and an obligation schema that assigns proof obligations to methods, then it is a *behavioral type*. We first define behavioral specifications. The semantics of terms in the *behavioral language* θ maps elements of the language into formulas of the semantic logic defined Sect. 3.3. We call elements of the behavioral language *behavioral words*.

Definition 3 (Behavioral Specification). *A* behavioral specification $\mathbb{T}$ *is a pair* $(\theta_{\mathbb{T}}, \alpha_{\mathbb{T}})$*, where* $\theta_{\mathbb{T}}$ *is a non-empty behavioral language and* $\alpha_{\mathbb{T}}$ *maps elements of* $\theta_{\mathbb{T}}$ *into formulas of semantic logic.*

It is straightforward to define, for example, postconditions as behavioral specifications, because they are already a trace property, where only the final element of a trace is relevant, but we aim for more complex properties, for example:

Example 8 (Points-To Behavioral Specification). The behavioral specification of a *points-to (p2) analysis* specifies that the next statement reads a future resolved by a method from a set M. The constant $\mathbf{1}$ refers to the index of the first event.

$$\begin{aligned}\mathbb{T}_{\mathsf{p2}} &= (\mathcal{P}(\mathsf{M}), \mathsf{p2}) \text{ with}\\ \mathsf{p2}(M) &= \exists \mathsf{X} \in \texttt{Object}.\ \exists f \in \texttt{Future}.\ \exists \mathtt{m} \in \texttt{Method}.\ \exists v \in \texttt{Val}.\ \exists i \in \mathbb{N}.\\ &\qquad \left([\mathbf{1}] \doteq \mathsf{futREv}(\mathsf{X}, f, \mathtt{m}, v, i) \wedge \bigvee_{\mathtt{m}' \in M} \mathtt{m} \doteq \mathtt{m}'\right)\end{aligned}$$

Intuitively, a behavioral language is a representation of a fragment of semantic logic. Such fragments are embedded into BPL with behavioral modalities of the form $[\mathtt{s} \Vdash^{\alpha_{\mathbb{T}}} \theta_{\mathbb{T}}]$. Following JavaDL, BPL uses *updates* [13] to keep track of state changes.

Definition 4 (Syntax of BPL). *Let* prd *range over predicate symbols,* fct *over function symbols,* x *over first-order variable names and* S *over sorts. As sorts we take all data types* `D`*, all interfaces, all class names and additionally* `Field`*,* `Future`*,* `Val`*, and* `Object`*. Formulas* φ*, updates* U *and terms* `t` *are defined by the following grammar, where* `v` *ranges over program variables, consisting of local variables and the special variables* `heap`*,* `heapOld`*,* `heapLast` *and* `result`*. Let* `fl` *range over all field names and* `eh` *range over expressions without direct field access, but with the extra program variable* `heap`*. Let* `s` *range over statements and* $(\theta_{\mathbb{T}}, \alpha_{\mathbb{T}})$ *over behavioral specifications.*

$$\begin{aligned}\varphi &::= prd(\bar{\mathtt{t}}) \mid \mathtt{t} \doteq \mathtt{t} \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x \in S.\ \varphi \mid [\mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \theta_{\mathbb{T}}] \mid \{U\}\varphi\\ \mathtt{t} &::= x \mid \mathtt{v} \mid \mathtt{fl} \mid fct(\bar{\mathtt{t}}) \mid \mathtt{eh} \mid \mathsf{if}\ \varphi\ \mathsf{then}\ \mathtt{t}\ \mathsf{else}\ \mathtt{t} \mid \{U\}\mathtt{t}\\ U &::= \epsilon \mid U||U \mid \{U\}U \mid \mathtt{v} := \mathtt{t}\end{aligned}$$

A behavioral modality says that all traces of `s` starting in the current state are models of $\alpha_{\mathbb{T}}(\theta_{\mathbb{T}})$. An update describes a specific state change. Updates are *delayed substitutions*: During the proof procedure, the state changes of the statement in question (inside a behavioral modality) are accumulated in updates and simplified. The resulting substitution is then applied *once per proof path* on the modality-free verification conditions resulting from the behavioral modality.

Example 9. The following BPL formula expresses that the **get** statement reads from a future that is resolved by `Producer.detectNews()` (see Expl. 4).

$$\left[\texttt{ns = fut.}\mathbf{get} \overset{\mathsf{p2}}{\Vdash} \{\texttt{Producer.detectNews}\}\right]$$

Definition 5 (Semantics of BPL). *Let I be a first-order model, i.e. a mapping from function names to functions and from predicate names to predicates. Let β be a variable assignment from first-order variables to semantic values.*

- *The evaluation of terms in a state σ is a function $[\![\mathtt{t}]\!]_{\sigma,\beta,I}$ that maps terms to domain elements.*
- *The evaluation of updates in a state σ is a function $[\![U]\!]_{\sigma,\beta,I}$ that maps updates to functions from states to states.*
- *The evaluation of formulas in a state σ is given as a satisfiability relation $\sigma, \beta, I \models \varphi$.*

Semantic evaluation is standard as in first-order logic, except for updates and behavioral modalities which are given in Fig. 8.

$$[\![\mathtt{v} := \mathtt{t}]\!]_{\sigma,\beta,I}(\sigma') := \sigma'[\mathtt{v} \mapsto [\![\mathtt{t}]\!]_{\sigma,\beta,I}]$$

$$[\![\epsilon]\!]_{\sigma,\beta,I}(x) := x \qquad [\![U_1||U_2]\!]_{\sigma,\beta,I}(x) := [\![U_2]\!]_{\sigma,\beta,I}\left([\![U_1]\!]_{\sigma,\beta,I}(x)\right)$$

$$[\![\{U_1\}U_2]\!]_{\sigma,\beta,I} := [\![U_2]\!]_{[\![U_1]\!]_{\sigma,\beta,I}\beta,I} \qquad [\![\{U\}t]\!]_{\sigma,\beta,I} := [\![t]\!]_{[\![U]\!]_{\sigma,\beta,I},\beta,I}$$

$$\sigma, \beta, I \models \{U\}\varphi \iff [\![U]\!]_{\sigma,\beta,I}, \beta, I \models \varphi$$

$$\sigma, \beta, I \models [\mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \theta_{\mathbb{T}}] \iff \forall \theta \in [\![\mathtt{s}]\!]_\sigma \,.\, \theta, \beta, I \models \alpha_{\mathbb{T}}(\theta_{\mathbb{T}})$$

Fig. 8. Semantics of updates and behavioral modalities.

We only consider models I that map each function symbol to its natural semantics. For example, heap functions and the `heap` program variable are mapped to the theory of arrays extended for objects [59]. Central is the following connection axiom, for all heaps h, fields `fl`, and terms `t`:

$$I(\mathsf{select})\big(I(\mathsf{store})(h, \mathtt{fl}, \mathtt{t}), \mathtt{fl}\big) = \mathtt{t}$$

For the full axiomatization we refer to Beckert et al. [14]. We further assume that all first-order variables are unique (no overloading) and that the type and number of parameters for functions and predicates is well-formed. We use common abbreviations such as $\forall x \in S.\ \varphi$ for $\neg\exists x \in S.\ \neg\varphi$. We shorten comparison expressions for terms of `Bool` type by writing, e.g., `i > j` instead of `i > j` $\doteq$ `True` and we render `select(heap, f)` as `this.f`. We use a sequent calculus to reason about validity of BPL-formulas.

Definition 6 (Sequent). *Let Γ, Δ be finite sets of* BPL*-formulas. A* sequent $\Gamma \Rightarrow \Delta$ *has the semantics $\bigwedge \Gamma \to \bigvee \Delta$. Γ is called the antecedent and Δ the succedent. A sequent $\{\gamma_1, \ldots, \gamma_n\} \Rightarrow \{\delta_1, \ldots, \delta_m\}$ is also written $\gamma_1, \ldots, \gamma_n \Rightarrow \delta_1, \ldots, \delta_m$.*

Definition 7 (Rules). *Let C, P_i be sequents. A rule has the form*

$$\textit{(name)}\ \frac{P_1 \quad \cdots \quad P_n}{C}\ \mathit{cond}$$

Where C is called the conclusion and P_i premises, while "cond" is a decidable side condition. Rules with zero premises are admissible and called axiom.

Definition 8 (Validity, Soundness, Calculus). *A BPL formula is* valid *if it is satisfied for all states σ, all variable assignments β and all models I (fixed as above). A rule is* sound *if validity of all premises implies validity of the conclusion. A calculus is a set of sound sequent rules.*

A behavioral type is a behavioral specification extended with (i) a proof obligation schema mapping every method in a given program to a behavioral word (its specification) and (ii) a set of sound validity rules for behavioral modalities containing the extended behavioral specification.

Definition 9 (Behavioral Type). *Let $\mathbb{T} = (\theta_{\mathbb{T}}, \alpha_{\mathbb{T}})$ be a behavioral specification. A* behavioral type *extending $\mathbb{T}$ is a quadruple $(\theta_{\mathbb{T}}, \alpha_{\mathbb{T}}, \iota_{\mathbb{T}}, \kappa_{\mathbb{T}})$. The calculus $\kappa_{\mathbb{T}}$ consists of rules over $\theta_{\mathbb{T}}$ (i.e. the conclusion must contain a behavioral modality with a behavioral word from $\theta_{\mathbb{T}}$). The* obligation schema *$\iota_{\mathbb{T}}$ is a map from method names $\mathtt{m}$ to pairs $(\varphi, \theta^{\mathtt{m}}_{\mathbb{T}})$, where φ is a first-order formula that may only contain field and local variables accessible by $\mathtt{m}$.*

Not every behavioral specification needs to be extended to a behavioral type, for example, $\mathbb{T}_{\mathtt{p2}}$ serves as an interface to an external analysis. To evaluate the formula in Expl. 9, the modalities can be compared against the result of a pointer analysis for futures [31]. For examples of other BPL behavioral types and a discussion on the notion of "Behavioral Type" we refer to [43].

5 Cooperative Contracts in BPL

We formulate cooperative contracts as a behavioral type in BPL, called *behavioral contract*[6]. We also define the extraction of annotated specifications to behavioral contracts and formulate the propagation of specifications described in Sect. 2.3 in terms of behavioral contracts.

Context sets are not part of behavioral contracts, but can be expressed analogously as (global) trace properties.

Compared to our previous account [48], the use of BPL vastly simplifies the verification system: First, the propagation does not require an additional intermediate representation. Second, the validity calculus operates directly on behavioral contracts and not on an *encoding* of contracts in postconditions and dedicated rules. In contrast to postcondition-based approaches [24], the calculus also does not have implicit parameters, such as call conditions.

In addition to the specification elements introduced in previous sections, we use class preconditions, which are essentially preconditions on the fields passed to the constructor.

[6] Due to their nature as contracts and behavioral types, not due to a relation to behavioral contracts as a subset of behavioral types as in [39].

Example 10. The following specifies that class `C` is initialized with a positive value for `param`.

```
/*@ requires this.param > 0; @*/
class C(Int param) { ... }
```

5.1 Cooperative Contracts as Behavioral Specifications

We first define suspension and resolve specifications. A suspension specification follows the structure of a suspension contract (without context sets) and consists of the frame, suspension assumption and suspension guarantee. Similarly, a resolve specification is a pair of the expected guarantee and the set of responsible methods.

Definition 10 (Suspension, Resolve Specification). *A* suspension specification Susp *is a triple* $(\mathbb{W}, ens, req)$*, where* $\mathbb{W} \subseteq \mathsf{Field}$ *is a set of fields and* $ens, req \in \mathsf{FOL}$ *are FOL formulas. A* resolve specification $\mathsf{Res} \in \mathcal{P}(\mathsf{M}) \times \mathsf{FOL}$ *is a pair* $(mtds, cond)$ *of a set of method names and a FOL formula.*

Example 11. The suspension specification for `notifyClients` in Expl. 5 is shown below, with the prettified syntax for heap access.

$$\begin{aligned}&(\mathbb{W} = \emptyset,\\ &ens = \texttt{ns} \neq \textbf{null} \wedge \texttt{len(}\textbf{this}\texttt{.myClients)} \leq \textbf{this}\texttt{.limit}\\ &req = \textbf{this}\texttt{.service} \doteq \textbf{null})\end{aligned}$$

A behavioral contract itself mirrors the structure of the cooperative contract for a method. To identify specific statements inside a method, we use program point identifiers (PPI), a generalization of the annotated names. The PPI for **await** and **get** statements is their name annotation, recorded as i in Sect. 3.2.1, the PPI for **return** statements is implicit and not exposed.

Definition 11 (Syntax for Behavioral Contracts). *The behavioral language* met *for contracts is a tuple* $(\mathbb{W}, post)_{\mathbb{S},\mathbb{C}}$ *with the following components:*

$\mathbb{S} : \mathsf{PPI} \mapsto (\mathsf{Susp} \cup \mathsf{Res} \cup \mathsf{FOL})$	*Program Point Specifications*
$\mathbb{C} : \mathsf{Classes} \cup \mathsf{Methods} \mapsto \mathsf{FOL}$	*Call Conditions*
$\mathbb{W} \subseteq \mathsf{Field}$	*Assignable fields*
$post \in \mathsf{FOL}$	*Postcondition*

A program-point specification $\mathbb{S}$ maps (i) every PPI of a suspension to a suspension specification, (ii) every PPI of a reading statement to a resolve contract, (iii) every PPI of a terminating statement to a method postcondition. Call conditions $\mathbb{C}$ map method names to the call precondition of the called method and class names to the class precondition.

The pair $(\mathbb{S}, \mathbb{C})$ does not change during symbolic execution: with top-down contract generation it can be seen as a lookup table for global properties. The set

$\mathbb{W}$ is the currently active dynamic frame. Finally, *post* is the current statement postcondition. It is *not* necessarily the postcondition of the method, but used to establish postconditions of arbitrary statements *not ending in suspension or termination*. This is needed to express loop invariants.

We say $(\mathbb{W}, \mathit{post})$ is the *active* part of the contract and $(\mathbb{S}, \mathbb{C})$ the *passive* part: The active part must be adhered to by the *currently active* statement, while the passive part serves as a specification repository for program points encountered during symbolic execution.

Example 12. Consider the following behavioral modality

$$\left[\texttt{this.f = this.g+1; [atom: "a"] await this.f > 0; s;} \overset{\mathsf{met}}{\Vdash} \left(\{\texttt{f}\}, \texttt{this.g} \doteq 1\right)_{\mathbb{S},\mathbb{C}}\right]$$

The behavioral contract expresses that before the next suspension (or termination), only writes to `f` are allowed. The formula $\texttt{this.g} \doteq 1$ is the statement postcondition. It does *not* need to hold at the suspension point.

5.2 Extraction and Propagation

Given a specified program P, the verification workflow consists of the following steps:

1. For each method a specification triple is extracted. Such a triple contains heap and parameter preconditions, as well as behavioral contracts. Context sets are recorded globally.
2. The behavioral contracts are propagated according to the context sets and the context sets are passed to external analyses to check whether the program adheres to them.
3. Each methods of the program is verified against its propagated behavioral contract.

5.2.1 Specification Extraction All abbreviations (such as `await o!m()`) are expanded before extraction. Syntactically, the formulas in the specification are an extension of the first-order fragment of BPL that only uses `heap` and the local variables as program variables. These may contain functions *old(·)*, *last(·)* that allow its argument (either a term or a formula) to be evaluated in the heap at the prestate of the current method call and at the most recent suspension, respectively. These constructs are removed by substitutions

$$\varphi[\texttt{heap} \setminus \texttt{heapOld}] \text{ and } \varphi[\texttt{heap} \setminus \texttt{heapLast}]$$

that replace occurrences of `heap` in the argument of *old(·)*, *last(·)* with ghost variables.

Given a program P, we define the extracted *passive* specification $\mathbb{S}^{\mathsf{P}}, \mathbb{C}^{\mathsf{P}}$:

$$\mathbb{C}^{\mathsf{P}}(\texttt{I.m}) = \textit{requires}\text{ clause of }\texttt{m}\text{ in interface }\texttt{I}$$
$$\mathbb{C}^{\mathsf{P}}(\texttt{C}) = \textit{requires}\text{ clause of class }\texttt{C}$$
$$\mathbb{S}^{\mathsf{P}}(\texttt{p}) = \begin{cases} \mathsf{res}(\texttt{p}) & \text{if p is the PPI of a } \textbf{get} \text{ statement} \\ \mathsf{susp}(\texttt{p}) & \text{if p is the PPI of a } \textbf{await} \text{ statement} \\ \mathsf{term}(\texttt{p}) & \text{if p is the PPI of a } \textbf{return} \text{ statement} \end{cases}$$

For brevity, write $\mathbb{S}^{\mathbb{W}}(\texttt{p})$ for the $\mathbb{W}$ component of $\mathbb{S}(\texttt{p})$ (if defined) and similar for the other possible components and $\mathbb{C}$. The postcondition of a return statement with PPI `p` is accessed simply with $\mathbb{S}(\texttt{p})$. The functions res, susp and term are generated as follows. For a resolve contract (the read statement possibly prefixed by an assignment) of the form

```
/*@ resolvedBy M; @*/
[atom: "name"] e.get;
```

we extract the resolve specification $\mathsf{res}(\texttt{name}) = \big(M, \mathsf{true}\big)$. If no resolve contract is given, M defaults to set of all method names.

Given a suspense contract, where A_1, A_2 are sets of method and atomic segment names,

```
/*@ ensures  ens;
    requires req;
    succeeds A1;
    overlaps A2;
    assignable 𝕎; @*/
[sync: "name"] await g;
```

we extract the suspension specification. $\mathsf{susp}(\texttt{name}) = \big(\mathbb{W}, \mathit{ens}, \mathit{req}\big)$. The default for state clauses is true, the default for context sets is the set of all method names and atomic block names in the class containing the specified statement, the default for the frame is the set of all fields of the class containing the statement.

Given a **return** statement with PPI `p`, we set its specification $\mathsf{term}(\texttt{p})$ to the conjunction of the *ensures* clauses of (i) the contract annotated to the containing method implementation in the class and (ii) the contract annotated to the method signature declared in an interface (if such a declaration exists and true otherwise). The following is the specification of a method `C.m` with contracts in its class and interface `I`:

```
interface I {
  /*@ requires pre^param;
      ensures post^param;
      succeeds A1;
      overlaps A2; @*/
  T m(...);
}
```

```
class C implements I {
  /*@  requires pre^heap;
       ensures post^heap;
       assignable 𝕎; @*/
  T m(...) {...}
}
```

The overall behavioral contract is the following triple, where the default of formulas is true and the default for sets is the set of all names (fields):

$$\Big(\mathit{pre}^{\mathsf{heap}}, \mathit{pre}^{\mathsf{param}}, \big(\mathbb{W}, \mathsf{true}\big)_{\mathbb{S}^{\mathsf{P}}, \mathbb{C}^{\mathsf{P}}}\Big)$$

In addition to the behavioral contract we also extract a global specification:

Definition 12 (Extraction of Global Specification). *From a program* P *we extract the* global specification $\mathbb{G}$ *that maps each atomic segment name and each method name to a pair of sets of atomic segment and method names. Each method name is mapped to the context set pair of its method contract and each atomic segment name is mapped to the context set pair of its suspension contract. For a name* a *we denote the first element of such a pair with* $\mathbb{G}^{\mathsf{Succ}}(a)$ *and the second with* $\mathbb{G}^{\mathsf{Over}}(a)$.

5.2.2 Specification Propagation Propagation uses the global specification to propagate heap preconditions. It is, however, *frame-aware*: if the propagated formula cannot possibly change its evaluation during executing an atomic segment, because it does not share any fields with the assignable clause, then the atomic segment needs not prove that it preserves the formula. It is ignored in the overlaps specification for propagation. It must, however, be taken into account when it is part of the succeeds set.

Definition 13 (Propagated Specification). *Let* P *be a program,* $\mathbb{C}$, $\mathbb{S}$ *the passive specification of the extracted behavioral contracts,* $\mathbb{G}$ *the extracted global specification. The* propagated *passive specification* $\mathbb{C}$, $\mathbb{S}_{\mathsf{prop}}$ *is generated as follows. Let* a *be any atomic segment name.*

- *For each* $\mathtt{m} \in \mathbb{G}^{\mathsf{Succ}}(\mathtt{m}')$*, let* $\mathtt{p}$ *be the PPI of the* **return** *statement of* $\mathtt{m}$ *and* ψ *the heap precondition of* $\mathtt{m}'$*. Then set* $\mathbb{S}_{\mathsf{prop}}(\mathtt{p}) = \mathbb{S}(\mathtt{p}) \wedge \psi$.
- *For each* $\mathtt{m} \in \mathbb{G}^{\mathsf{Succ}}(a)$*, let* $\mathtt{p}$ *be the PPI of the* **return** *statement of* $\mathtt{m}$ *and* ψ *the* requires *clause of* $\mathbb{S}(a)$*. Then set* $\mathbb{S}_{\mathsf{prop}}(\mathtt{p}) = \mathbb{S}(\mathtt{p}) \wedge \psi$.
- *For each* $\mathtt{m} \in \mathbb{G}^{\mathsf{Over}}(\mathtt{m}')$*, let* $\mathtt{p}$ *be the PPI of the* **return** *statement of* $\mathtt{m}$*,* $\mathbb{W}$ *its* assignable *set and* ψ *the postcondition of* $\mathtt{m}'$*, if* $\mathsf{fields}(\psi) \cap \mathbb{W} \neq \emptyset$*, we set*

$$\mathbb{S}_{\mathsf{prop}}(\mathtt{p}) = \mathbb{S}(\mathtt{p}) \wedge \big((\{\mathtt{heap} := \mathtt{oldHeap}\}\psi) \rightarrow \psi\big)$$

- *For each* $\mathtt{m} \in \mathbb{G}^{\mathsf{Over}}(a)$*, let* $\mathtt{p}$ *be the PPI of the* **return** *statement of* $\mathtt{m}$*,* $\mathbb{W}$ *its* assignable *set and* ψ *the postcondition of* $\mathbb{S}(a)$*, if* $\mathsf{fields}(\psi) \cap \mathbb{W} \neq \emptyset$*, we set*

$$\mathbb{S}_{\mathsf{prop}}(\mathtt{p}) = \mathbb{S}(\mathtt{p}) \wedge \big((\{\mathtt{heap} := \mathtt{oldHeap}\}\psi) \rightarrow \psi\big)$$

- *Analogously for atomic segment names* $a' \in \mathbb{G}^{\mathsf{Over}}(a)$ *and* $a' \in \mathbb{G}^{\mathsf{Over}}(\mathtt{m}')$*, where the* ensures *clause instead of the postcondition is manipulated.*

This construction is applied recursively for all atomic segment and method names. Afterwards, the postconditions of the interface contract are made available in the resolve contracts. Let $\mathtt{r}_{\mathtt{C.m}}$ *be the PPI of the* **return** *statement of* $\mathtt{C.m}$ *and* $\mathbb{S}^{\mathsf{param}}_{\mathsf{prop}}(\mathtt{r}_{\mathtt{C.m}})$ *its conjunct from the interface contract. Then set*

$$\mathbb{S}^{cond}_{\mathsf{prop}}(\mathtt{p}) = \bigvee_{\substack{\mathtt{r}_{\mathtt{C.m}} \\ \mathtt{C.m}\ implements\ \mathtt{m} \in \mathbb{S}^{mtds}(\mathtt{p})}} \mathbb{S}^{\mathsf{param}}_{\mathsf{prop}}(\mathtt{r}_{\mathtt{C.m}}) \ .$$

5.3 Semantics of Behavioral Contracts

We require a few auxiliary definitions to locate events in traces. For each event type ev, for example, futEv, there is a predicate $\mathsf{isEv}(i)$ that holds iff the i-th element of the current trace is an event of type ev. Also predicate $\mathsf{isInvEv}(i, \mathtt{m})$ holds iff the i-th element of the current trace is an invEv on method m, and analogously for isNewEv. Similar definitions include predicate $\mathsf{isSuspEv}(i)$ that holds iff the i-th element of the current trace is an SuspBoolEv or SuspFutEv; predicate $\mathsf{isSuspEv}(i, \mathtt{p})$ holds iff the i-th element of the current trace is an SuspBoolEv or SuspFutEv at PPI p; predicate $\mathsf{isInvEvW}(i, \mathtt{m}, \mathsf{e})$ holds iff the i-th element of the current trace is an invEv on method m with value e, and analogously for isNewEvW, isFutEvW and isFutREvW.

Given a specification φ with parameters (for example, method parameters), write $\varphi(\overline{\mathsf{e}})$ for the substitution of the parameters with the corresponding terms from $\overline{\mathsf{e}}$. Given a field f, we denote the declared type of its values with $T_{\mathtt{f}}$. Predicate $\mathsf{isLast}(i, j)$ holds iff j is either the final state of the current trace after i or the state before the first suspension event after i:

$$\begin{aligned}\mathsf{isLast}(i,j) \equiv\ & j > i \wedge \\ & \Big(\big(\mathsf{isSuspEv}(j+1) \wedge \forall k \in \mathbf{I}.\ (i < k < j \rightarrow \neg \mathsf{isSuspEv}(k))\big) \vee \\ & \big(\forall k \in \mathbf{I}.\ k \leq j \wedge (i < k < j \rightarrow \neg \mathsf{isSuspEv}(k))\big)\Big)\end{aligned}$$

Definition 14 (Semantics of Behavioral Contracts). *The semantics α of a frame $\mathbb{W}$ starting at trace position $i \in \mathbf{I}$ is that the trace has the same value in all fields not in $\mathbb{W}$ at the subsequent suspension event (or final state, if the trace has no suspension):*

$$\begin{aligned}&\alpha(\mathbb{W}, i) = \\ &\quad \exists j \in \mathbf{I}.\ \mathsf{isLast}(i,j) \wedge \bigwedge_{\mathtt{f} \notin \mathbb{W}} \exists t_{\mathtt{f}} \in T_{\mathtt{f}}.\ \Big(\big([i] \vdash \mathbf{this}.\mathtt{f} \doteq t_{\mathtt{f}}\big) \wedge \big([j] \vdash \mathbf{this}.\mathtt{f} \doteq t_{\mathtt{f}}\big)\Big)\end{aligned}$$

The semantics α of a postcondition φ is that the final state is a model for φ, unless the last event is a resolve event:

$$\alpha(\varphi) = \exists max \in \mathbf{I}.\ \big(\forall i \in \mathbf{I}. i \leq max\big) \wedge \neg \mathsf{isFutEv}(max - 1) \rightarrow [max] \vdash \varphi$$

The semantics α of the call conditions $\mathbb{C}$ is that each call and each object creation has parameter values that are a model for the call (creation) precondition. The instantiation of the precondition contains no variable or field of the callee, but is evaluated in a state:

$$\begin{aligned}\alpha(\mathbb{C}) = & \bigwedge_{\mathtt{m} \in \mathbf{dom}(\mathbb{C})} \forall i \in \mathbf{I}.\ \mathsf{isInvEv}(i, \mathtt{m}) \rightarrow \\ & \qquad \big(\exists \overline{\mathsf{e}} \in \texttt{List<Any>}.\ \mathsf{isInvEvW}(i, \mathtt{m}, \overline{\mathsf{e}}) \wedge [i+1] \vdash \mathbb{C}(\mathtt{m})(\overline{\mathsf{e}})\big) \\ & \bigwedge_{\mathtt{C} \in \mathbf{dom}(\mathbb{C})} \forall i \in \mathbf{I}.\ \mathsf{isNewEv}(i, \mathtt{m}) \rightarrow \\ & \qquad \big(\exists \overline{\mathsf{e}} \in \texttt{List<Any>}.\ \mathsf{isNewEvW}(i, \mathtt{m}, \overline{\mathsf{e}}) \wedge [i+1] \vdash \mathbb{C}(\mathtt{C})(\overline{\mathsf{e}})\big)\end{aligned}$$

The semantics α of the suspension specification is that at every suspension (i) the frame semantics holds until the subsequent suspension point, (ii) the ensures *clause is established, and (iii) the* requires *clause holds. Additionally, at every return statement the corresponding postcondition holds and at every future read the correct method has been read and the conditions on the read value hold:*

$$\begin{aligned}\alpha(\mathbb{S}) = &\bigwedge_{\mathtt{p}\in\mathbf{dom}(\mathbb{S})} \forall i \in \mathbf{I}.\ \mathsf{isSuspEv}(i,\mathtt{p}) \rightarrow \quad [i+1] \vdash \mathbb{S}^{ens}(\mathtt{p}) \wedge [i+3] \vdash \mathbb{S}^{req}(\mathtt{p}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \alpha(\mathbb{S}^{\mathbb{W}}(\mathtt{p}, i+4) \\ \wedge &\bigwedge_{\substack{\mathtt{p}\in\mathbf{dom}(\mathbb{S})\\ \mathtt{p}\ is\ at\ \mathbf{return}}} \forall i \in \mathbf{I}.\ \forall \mathsf{e} \in \mathtt{Any}.\ \mathsf{isFutEvW}(i,\mathtt{p},\mathsf{e}) \rightarrow [i-1] \models \mathbb{S}(\mathtt{p})(\mathsf{e}) \\ \wedge &\bigwedge_{\substack{\mathtt{p}\in\mathbf{dom}(\mathbb{S})\\ \mathtt{p}\ is\ at\ \mathbf{get}}} \forall i \in \mathbf{I}.\ \forall \mathsf{e} \in \mathtt{Any}.\ \mathsf{isFutREvW}(i,\mathtt{p},\mathsf{e}) \rightarrow [i-1] \models \mathbb{S}^{cond}(\mathtt{p})(\mathsf{e})\end{aligned}$$

The complete semantics is

$$\mathsf{met}\big((\mathbb{W},\varphi)_{\mathbb{S},\mathbb{C}}\big) = \exists mi \in \mathbf{I}.\ \big(\forall i \in \mathbf{I}.\ i \geq mi\big) \wedge \alpha(\mathbb{W}, mi) \wedge \alpha(\varphi) \wedge \alpha(\mathbb{S}) \wedge \alpha(\mathbb{C})\ .$$

The formula is finite, as for any specification the domains of $\mathbb{C}$, $\mathbb{S}$ are finite.

5.4 Method Contracts as Behavioral Types

We proceed to proof obligation generation and proof calculus. The proof obligation schema is straightforward. In particular, it requires neither an encoding of frames in the postcondition like JavaDL [59] nor statements involving the event history like ABDSL [27].

Definition 15 (Proof Obligation Schema). *Let* P *be a specified program and* $\mathsf{Mtd_P}$ *the set of its methods. Let* $\big(\mathsf{pre}^{\mathsf{param}}_{\mathtt{m}}, \mathsf{pre}^{\mathsf{heap}}_{\mathtt{m}}, \theta_{\mathtt{m}}\big)_{\mathtt{m}\in\mathsf{Mtd_P}}$ *be the extracted and propagated set of behavioral contract triples from* P*. Further, given a method* $\mathtt{m}$*, let* $\mathtt{s_m}$ *be the method body of* $\mathtt{m}$*. The proof obligation schema is*

$$\iota(\mathtt{m}) = \big(pre^{\mathsf{param}}_{\mathtt{m}} \wedge pre^{\mathsf{heap}}_{\mathtt{m}}, \theta_{\mathtt{m}}\big)$$

which characterizes the following proof obligations:

$$\{\mathtt{lastHeap} := \mathtt{heap}\}\{\mathtt{oldHeap} := \mathtt{heap}\}\big(\big(pre^{\mathsf{param}}_{\mathtt{m}} \wedge pre^{\mathsf{heap}}_{\mathtt{m}}\big) \rightarrow [\mathtt{s_m} \overset{\mathsf{met}}{\Vdash} \theta_{\mathtt{m}}]\big)\ .$$

The two updates save the prior state for the specifications connecting two different heaps: `oldHeap` is the heap at the beginning of method execution, `lastHeap` is the heap from the last (re-)activation. In the beginning, all heaps (`heap`, `lastHeap`, `oldHeap`) coincide.

To formulate the proof system we need the auxiliary definitions stated in Fig. 9:

– Applying the update $U_{\mathcal{A}}$ removes ("anonymizes") all state information from the `heap` variable and saves the old state in `lastHeap`.

- Similarly, $U^{\mathbb{W}}_{\mathcal{F}}$ removes all information from all program variables in $\mathbb{W}$ and all heaps except `oldHeap`.
- $\mathbb{W}'_{\mathtt{s}}$ is the intersection of the currently active frame $\mathbb{W}$ and all frame specifications in `s`. This is needed for loops, because the loop body `s` has to be in the frame from before the loop, as well as in the frame at the end of every iteration. These sets may differ when the loop body contains an **await**.
- fields extracts all field accesses from a set of expressions.

$$U_{\mathcal{A}} = \{\mathtt{lastHeap} := \mathtt{heap}\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}$$

$$U^{\mathbb{W}}_{\mathcal{F}} = \{\mathtt{lastHeap} := \mathsf{anon}(\mathtt{lastHeap})\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\mathtt{v}_1 := v_1\}\dots\{\mathtt{v}_n := v_n\}$$
for each non-heap program variable $\mathtt{v}_i \in \mathbb{W}$ and fresh symbols v_i.

$$\mathbb{W}'_{\mathtt{s}} = \begin{cases} \mathbb{W} & \text{if } \mathtt{s} \text{ contains no } \textbf{await} \\ \mathbb{W} \cap \bigcap_{\text{PPI } \mathtt{i} \text{ in } \mathtt{s}} \mathbb{S}^{\mathbb{W}}(\mathtt{i}) & \text{otherwise} \end{cases}$$

$$\mathsf{fields}(\mathtt{e}_1, \dots, \mathtt{e}_n) = \text{ fields within expressions } \mathtt{e}_1, \dots, \mathtt{e}_n$$

Fig. 9. Auxiliary definitions.

Definition 16 (Proof Calculus). *The proof system for contracts is in Fig. 10. The rules for writes to variables are similar to those for field writes to* `f`*: the generated update* $x := t$ *changes to* $\mathtt{heap} := \mathsf{store}(\mathtt{heap}, \mathtt{f}, t)$*. Additionally* $\mathtt{f} \in \mathbb{W}$ *is added, see* ***(assignF)****. In the remaining rules we only give the version for variables.*

The rules for all statements except **return** and **skip** are given with an active statement and a continuation `s`. The rules for **skip** ensure that if the statement has no continuation, a **skip** can be added.

- Rule **(assign)** turns an assignment to a variable into an update and **(assignF)** does the same for a field.
- The rule **(skI)** introduces **skip** as discussed above, **(skF)** removes a **skip** as the active statement, **(sk)** terminates symbolic execution evaluates the statement postcondition when **skip** is the sole remaining statement.
- Rule **(return)** proves the *method* postcondition at (the implicit) PPI `i` of the **return** statement. The statement postcondition needs *not* to hold here.
- Rule **(get)** has two premises: the first checks with a global points-to analysis that the correct method is synchronized. There are no rules for this modality—the branch is closed by an external analysis. The second premise generates a fresh symbol v and adds the propagated knowledge about it when symbolic execution is continued.
- Rule **(await)** checks first that the correct predicate holds before termination, and continues symbolic execution according to the specification of the suspension point. The statement postcondition needs *not* to hold here.

$$\textsf{(assign)}\ \frac{\Gamma \Longrightarrow \{U\}\{\mathtt{x} := \mathtt{e}\}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}{\Gamma \Longrightarrow \{U\}\big[\mathtt{x = e;\ s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}$$

$$\textsf{(assignF)}\ \frac{\Gamma \Longrightarrow \{U\}\{\mathtt{heap} := \mathsf{store}(\mathtt{heap}, \mathtt{f}, \mathtt{e})\}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}{\Gamma \Longrightarrow \{U\}\big[\mathtt{f = e;\ s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}\ \mathtt{f} \in \mathbb{W}$$

$$\textsf{(skI)}\ \frac{\Gamma \Longrightarrow \{U\}\big[\mathtt{s;\ skip} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}{\Gamma \Longrightarrow \{U\}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}\ \mathtt{s} \text{ is neither } \textbf{skip} \text{ nor composed}$$

$$\textsf{(skF)}\ \frac{\Gamma \Longrightarrow \{U\}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}{\Gamma \Longrightarrow \{U\}\big[\textbf{skip}\mathtt{;\ s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta} \qquad \textsf{(sk)}\ \frac{\Gamma \Longrightarrow \{U\}\varphi, \Delta}{\Gamma \Longrightarrow \{U\}\big[\textbf{skip} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}$$

$$\textsf{(return)}\ \frac{\Gamma \Longrightarrow \{U\}\{\mathtt{result} := \mathtt{e}\}\mathbb{S}(\mathtt{i}), \Delta}{\Gamma \Longrightarrow \{U\}\big[\textbf{return}\ \mathtt{e;} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}\quad \mathtt{i} \text{ is the PPI of this } \textbf{return}$$

$$\textsf{(get)}\ \frac{\begin{array}{c}\Longrightarrow \big[\mathtt{[sync: "i"]}\ x\ \mathtt{=\ e.}\textbf{get}\mathtt{;\ s} \overset{\mathsf{p2}}{\Vdash} \mathbb{S}^{mtds}(\mathtt{i})\big] \\ \Gamma, \{U\}\{\mathtt{result} := v\}\mathbb{S}^{cond}(\mathtt{i}) \Longrightarrow \{U\}\{x := v\}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta\end{array}}{\Gamma \Longrightarrow \{U\}\big[\mathtt{[sync: "i"]}\ x\ \mathtt{=\ e.}\textbf{get}\mathtt{;\ s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}\ v \text{ fresh}$$

$$\textsf{(await)}\ \frac{\begin{array}{c}\Gamma \Longrightarrow \{U\}\mathbb{S}^{\mathsf{ens}}(\mathtt{i}), \Delta \\ \Gamma, \{U\}U_{\mathcal{A}}(\mathbb{S}^{\mathsf{req}}(\mathtt{i}) \wedge \mathtt{g}) \Longrightarrow \Gamma, \{U\}U_{\mathcal{A}}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{S}^{\mathbb{R}}(\mathtt{i}), \mathbb{S}^{\mathbb{W}}(\mathtt{i}), \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta\end{array}}{\Gamma \Longrightarrow \{U\}\big[\mathtt{[atom: "i"]}\ \textbf{await}\ \mathtt{g;\ s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}$$

$$\textsf{(create)}\ \frac{\Gamma \Longrightarrow \{U\}\mathbb{C}(\mathtt{C})(\overline{\mathtt{e}}), \Delta \qquad \Gamma \Longrightarrow \{U\}\{x := o\}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}{\Gamma \Longrightarrow \{U\}\big[x\ \mathtt{=\ e.}\textbf{new}\ \mathtt{C(\overline{e});\ s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}\ o \text{ fresh}$$

$$\textsf{(call)}\ \frac{\Gamma \Longrightarrow \{U\}\big(\mathtt{e} \not\doteq \textbf{null} \wedge \mathbb{C}(\mathtt{m})(\overline{\mathtt{e}})\big), \Delta \quad \Gamma \Longrightarrow \{U\}\{x := f\}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}{\Gamma \Longrightarrow \{U\}\big[x\ \mathtt{=\ e!m(\overline{e});\ s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}\ f \text{ fresh}$$

$$\textsf{(loop)}\ \frac{\begin{array}{c}\Gamma \Longrightarrow \{U\}I, \Delta \qquad \Gamma, \{U\}U_{\mathcal{F}}^{\mathbb{W}'_{\mathtt{s}}}(I \wedge \mathtt{e}) \Longrightarrow \{U\}U_{\mathcal{F}}^{\mathbb{W}'_{\mathtt{s}}}\big[\mathtt{s} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}'_{\mathtt{s}}, I)_{\mathbb{S},\mathbb{C}}\big], \Delta \\ \Gamma, \{U\}U_{\mathcal{F}}^{\mathbb{W}'_{\mathtt{s}}}(I \wedge \neg\mathtt{e}) \Longrightarrow \{U\}U_{\mathcal{F}}^{\mathbb{W}'_{\mathtt{s}}}\big[\mathtt{s}' \overset{\mathsf{met}}{\Vdash} (\mathbb{W}'_{\mathtt{s}}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta\end{array}}{\Gamma \Longrightarrow \{U\}\big[\textbf{while}\ \mathtt{(e)}\ \{\mathtt{s}\}\ \mathtt{s}' \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}$$

$$\textsf{(branch)}\ \frac{\begin{array}{c}\Gamma, \{U\}\mathtt{e} \Longrightarrow \{U\}\big[\mathtt{s;s''} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta \\ \Gamma, \{U\}\neg\mathtt{e} \Longrightarrow \{U\}\big[\mathtt{s';s''} \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta\end{array}}{\Gamma \Longrightarrow \{U\}\big[\textbf{if}\ \mathtt{(e)}\ \textbf{then}\ \{\mathtt{s}\}\ \textbf{else}\ \{\mathtt{s}'\}\ \mathtt{s}'' \overset{\mathsf{met}}{\Vdash} (\mathbb{W}, \varphi)_{\mathbb{S},\mathbb{C}}\big], \Delta}$$

Fig. 10. Calculus for behavioral contracts

- Rules **(create)** and **(call)** handle object creation and method calls: the precondition has to be proven in one premise and a fresh symbol is used subsequently during symbolic execution. Rule **(call)** additionally checks that the target is not **null**.
- Rule **(loop)** is a standard loop invariant rule: The invariant formula I has to be proven when the loop is entered. This is done in the first premise. The second premise checks that I is preserved by the method body. The update mechanism removes all information from the accessed variables and heap, except that I and the guard holds. This is the only place where the statement postcondition is modified: it must be shown that the method body has the postcondition I. The mechanism for the frames is described above.
- Rule **(branch)** splits the proof into two branches, following the two branches of the **if** statement.

The rules assume a standard technique to translate expressions into terms and formulas, however, a guard `e?` is translated into `true`.

Frames are checked syntactically by computing the accessed fields in an assignment, yet our system is more precise than a purely syntactic approach: By embedding frames into symbolic execution, the check is flow- and value-sensitive. For example, it avoids to execute dead code. In addition, it is *contract-sensitive*: given multiple contracts we can check whether the frames hold for a given specification, while purely syntactic approaches cannot distinguish these cases.

We also do not require an explicit history variable keeping track of events, as the trace properties dealing with events are hidden in the behavioral modalities.

5.5 Context Sets as Global Trace Properties and Soundness

The main theorem states that in every global trace of a verified program, the projections to local traces are models for the semantics of the corresponding method contract. This requires a characterization of context sets as properties of global traces. In addition, we assume standard soundness proofs for most rules in the proof calculus. However, rules **(await)** and **(get)** *depend on non-local information*. For example, **(get)** is only sound if the methods that provide the read value indeed have verified soundness conditions. These rules, therefore, are part of the composition step of the main theorem.

Lemma 1. *Rules* ***(assign)***, ***(assignF)***, ***(skI)***, ***(skF)***, ***(sk)***, ***(return)***, ***(create)***, ***(call)***, ***(loop)***, ***(branch)*** *are sound, i.e. validity of the premises implies validity of the conclusion.*

Definition 17 (Semantics of Global Specification). *Let* $\mathtt{m}$ *be a method and* $\mathtt{p}$ *an atomic segment name. A trace sh adheres to global specification* $\mathbb{G}$ *if:*

- *For every invocation reaction event on* $\mathtt{m}$ *at index* i*, there is a future event from a method* $\mathtt{m}' \in \mathbb{G}^{\mathsf{Succ}}(\mathtt{m})$ *or a suspension event from some* $\mathtt{p}' \in \mathbb{G}^{\mathsf{Succ}}(\mathtt{m})$ *at index* $j < i$*. If these sets are empty, then* i *must be the first invocation reaction event on the object. Moreover, every future or suspension event with index* k*, such that* $j < k < i$*, is from some* $\mathtt{m}'' \in \mathbb{G}^{\mathsf{Over}}(\mathtt{m})$ *or* $\mathtt{p}'' \in \mathbb{G}^{\mathsf{Over}}(\mathtt{m})$*.*

– *For every suspension reaction event on* $\mathtt{p}$ *at index* i*, there is a future event from a method* $\mathtt{m}' \in \mathbb{C}^{\mathsf{Succ}}(\mathtt{p})$ *or a suspension event from some* $\mathtt{p}' \in \mathbb{C}^{\mathsf{Succ}}(\mathtt{p})$ *at index* $j < i$*. Moreover, every future or suspension event with index* k*, such that* $j < k < i$*, is from some* $\mathtt{m}'' \in \mathbb{C}^{\mathsf{Over}}(\mathtt{p})$ *or* $\mathtt{p}'' \in \mathbb{C}^{\mathsf{Over}}(\mathtt{p})$*.*

Given a future f in a global trace sh, let $\mathtt{m}_f$ be the method resolving f and $\theta_{\mathtt{m}}$ its behavioral contract.

Theorem 1. *Let* P *be a program. If all method contracts can be proven, i.e. there is a proof in the calculus for the sequents in* $\iota(\mathtt{m})$*, and all traces* sh *generated by* P *adhere to the global specification, then for each global trace generated by* P *and each future* f *within* sh*, the projection of* sh *on* f *is a model for* $\theta_{\mathtt{m}_f}$*:*

$$sh|_f \models \mathsf{met}\big(\theta_{\mathtt{m}_f}\big)$$

Proof Sketch. The proof is by induction over the number n of reactivations and future reads in the generated trace sh.

n = 0. In the first base case there are no suspensions and future reads, hence, rules **(get)** and **(await)** are irrelevant. So all sequents in $\iota(\mathtt{m})$ are valid. It remains to show that the proof obligations do not discard any traces, i.e. preconditions pre^{heap}, pre^{param} hold in the first state of every projected trace $sh|_f$.
Let i be the invocation reaction of a process and $\mathtt{m}_i$ its method. If the semantics of the global specification holds, then before i there was a position $k < i$ with a future event that terminates a process of a method $\mathtt{m}_k$ in the *succeeds* set of $\mathtt{m}_i$. By propagation, pre^{heap} is established here, as $\iota(\mathtt{m}_k)$ is valid. Between k and i only methods $\mathtt{m}'_k$ from the *overlaps* method run on the same object, so pre^{heap} is preserved until i because $\iota(\mathtt{m}'_k)$ is valid. Methods running on other objects are irrelevant, because they cannot access the heap of the object running $\mathtt{m}_i$ and pre^{heap} contains only fields of one object. Regarding pre^{param}, there is a $l < i$ with the invocation event corresponding to i. This l is issued by some other method $\mathtt{m}_l$. Since $\iota(\mathtt{m}_l)$ is valid, that call has been verified to adhere to the call conditions of $\mathbb{C}$.

n = 1. In the second base case there is exactly one future read or reactivation. Let i be the position of that event in sh. Let $\mathtt{m}_i$ be the method issuing the event. We distinguish the two cases:

Future Read. Let $\mathtt{p}$ be the PPI of the reading **get** statement. By assumption, the proof for $\iota(\mathtt{m}_i)$ has been provided, but this does not imply that $\iota(\mathtt{m}_i)$ is valid: Rule **(get)** is not sound, i.e., validity of its premises does not imply validity of its conclusion. But it is sound for the state σ before i: if its premises hold in σ, then its conclusion holds in σ. To establish this, we must show that $\{\mathtt{result} := v\}\mathbb{S}^{cond}(\mathtt{p})$ holds in the state before i. It is sufficient to show that the read value is described by $\mathbb{S}^{cond}(\mathtt{p})$.
By the first rule premise, $\mathtt{m}_i$ is in $\mathbb{S}^{mtds}(\mathtt{p})$. By the second premise, the code following the **get** statement is a model for the rest of the behavioral contract, if the read value is described by $\mathbb{S}^{cond}(\mathtt{p})$. By propagation, $\mathbb{S}^{cond}(\mathtt{p})$ is the disjunction of all postconditions (from the interface, so they contain only `result` as a program variable).

We observe that every method not containing **await** and **get** statements is a model for its type by the above argument. To read a future, a method $\mathtt{m}_i$ must have terminated with a future event at position $k < i$. So the read value v is described by $\{\mathtt{result} := v\}post^{\mathsf{param}}$ and

$$\{\mathtt{result} := v\}post^{\mathsf{param}} \rightarrow \{\mathtt{result} := v\}\mathbb{S}^{cond}(\mathtt{p}) \ .$$

Thus, the proof of $\iota(\mathtt{m}_i)$ describes all relevant states to conclude that the theorem statement holds.

Reactivation. Let $\mathtt{p}$ be the PPI of the reading **await** statement. This case is similar to the previous one, except one has to show that **(await)** is sound at i. It is sufficient to show that $\mathbb{S}^{\mathsf{req}}(\mathtt{p}) \wedge \mathtt{g}$ can be assumed. The guard condition $\mathtt{g}$ obviously holds, as it is directly part of the semantics. The argument for $\mathbb{S}^{\mathsf{req}}(\mathtt{p})$ is the same as in the proof in the case for $n = 0$, i.e. that pre^{heap} can be assumed at method start. The only difficulty arises when the **await** statement has to establish $\mathbb{S}^{\mathsf{req}}(\mathtt{p})$ itself. However, if the proof of $\iota(\mathtt{m}_i)$ has been closed, then the first premise has been shown and does *not* rely on the soundness of **(await)**: we may extract a partial proof by pruning the branch corresponding to the second premise to establish that at suspension $\mathbb{S}^{\mathsf{req}}(\mathtt{p})$ holds.

n > 1. Let i be the index of the last reactivation or future read event. For $sh[0 \ldots i-1]$ we can apply the induction hypothesis, i.e. every complete local trace so far was a model for its contract. Let $\mathtt{m}_i$ be the method issuing the event. We distinguish the same cases as above:

Future Read. The case for future reads is analogous to the one in base case $n = 1$. The only difference is that the read future may have been resolved by a method that contains a **get** statement itself. The soundness of **(get)** for those states, however, is established by the induction hypothesis.

Reactivation. This case is again analogous. The only difference is that other **await** statements may need to establish the suspension assumption (instead of the **await** statement in question), and the process of the other **await** statement has not terminated yet at i. This is covered by the same argument as in the base case $n = 1$ by extracting partial proofs for any relevant method. Otherwise, the induction hypothesis suffices. □

We integrate the pointer analysis as a behavioral specification that is obviously not complete. The appropriate notion of completeness for logics referring to external analyses remains an open question.

6 Related Work

Wait conditions were introduced as program statements for critical regions and monitors in the pioneering work of Brinch-Hansen [35, 36] and Hoare [37]. Reasoning approaches for monitors are discussed by Dahl [20]. SCOOP [9] explores preconditions as wait/when conditions.

The paper [23] provided a reasoning system for distributed communications between active objects where interleaved concurrency inside each object is based on explicit release points. This paper assumes preconditions to hold when methods are activated, but uses a set of invariants, i.e., one monitor invariant capturing interleaving at each release point. The proof system is used to prove that a class maintains a set of monitor invariants which describe its release points. Compared to the present work, the major difference is in how release points are handled. We have more expressive language to specify at the granularity of atomic segments at the interleaving points. Thus, the proof system of [23] is more expressive than reasoning merely over a single class invariant, but less expressive than ours.

Previous approaches to AO verification [22, 25, 27] consider only object invariants that must be preserved by every atomic segment of every method. As mentioned above, this is a special case of our system and makes it hard to specify different method behavior. Object invariants are also used in KeY-ABS [24, 27] to specify and verify behavioral contracts for ABS programs. KeY-ABS is based on a four-event semantics for asynchronous method calls, which introduces disjoint alphabets for the local histories of different objects, extending the work of Ahrendt & Dylla [5]: an invocation event in [5] is split into an invocation and invocation reaction event; a completion event is split into a completion and completion reaction event. Disjoint alphabets allow to reduce the complexity of reasoning about programs, because they significantly simplify formulas in terms of the number of needed quantifiers.

Actor services [58] 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 [1,42] implemented on top of Frama-C by Gavran et al. [32] demonstrated modular proofs of partial correctness for asynchronous C programs restricted to using the Libevent library.

A verification system for message passing programs written in Java and the MPJ library can be found in [57], where a *future* has a different meaning compared to our work. The authors model communication protocols in the mCRL2 process algebra. These algebraic terms were defined as *futures* to predict how components will interact during program execution. Permission-based separation logic and model checking were applied in [57] to reason about local and global correctness of a network, respectively. Specification and verification includes message sending, receiving and broadcasting, but no method contracts.

Contracts for channel-based communication are partly supported by session types [16, 38]. These have been adapted to the AO concurrency model [47], including assertions on heap memory [46], but require composition to be explicit in the specification. Stateful session types for AO [46] contain a propagation step (cf. Sect. 2.3): 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 then 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.

Compositional specification of concurrency models outside rely-guarantee was mainly proposed based on separation logic [17, 56], which separates shared memory regions [29] and assigns responsibilities for regions to processes. Shared regions relate predicates over the heap that must be stable, i.e. invariant, when accessed. Huisman et al. [15, 62] have used permission-based separation logic to verify class invariants in multi-threaded programs, using barrier contracts. Even though approaches to specify regions precisely have been developed [19,29], their combination with interaction modes beyond heap access (such as asynchronous calls and futures) is not well explored. It is worth noting that AO do not require the concept of regions in the logic, because strong encapsulation and cooperative scheduling ensure that two threads never run in parallel on the same heap. The central goal of separation *logic*—separation of heaps—is a design feature of the AO *concurrency model*.

7 Conclusion

Preemption interferes with specification contracts in concurrent programs, because the unit of computation here differs from the unit of specification. Cooperative scheduling introduces syntactically declared program points for preemption, occupying a middle ground between no preemption (as in actors and sequential programs) and full preemption, as in multi-threaded programs.

This paper addresses the problem of specification contracts for cooperative scheduling in active object languages. Because message passing does not correspond to transfer of control in the asynchronous setting, it is necessary to distinguish the responsibilities of the caller from the responsibilities of the callee in fulfilling the precondition of a task. We solve this problem by means of pre- and postconditions at the level of interfaces, reflecting that the caller must fulfill the *parameter precondition*, and at the level of implementations, reflecting that only the callee has enough knowledge of the implementation to fulfill the *heap precondition*. We further show that by exploiting the syntactic declaration of such preemption points, it is possible to specify locally how tasks depend on each other and when different tasks may safely overlap.

Technically, the paper develops a specification language for cooperatively scheduled active objects by specifying program behavior at the possible interleaving points between tasks in terms of a concurrency context with *succeeds* and *overlaps* sets. These sets enable fine-grained interleaving behavior to be specified when required, and otherwise default into standard invariants, which weaken the specification to allow any interleaving. We formalized reasoning about such specifications in a behavioral program logic over behavioral types. It relates the trace semantics generated by execution of programs with method contracts expressed as behavioral types.

Acknowledgments This work is supported by the `SIRIUS` Centre for Scalable Data Access and the `FormbaR` project, part of AG Signalling/DB RailLab in the Innovation Alliance of Deutsche Bahn AG and TU Darmstadt. The authors thank Wolfgang Ahrendt, Frank de Boer, and Henk Mulder for their careful reading and valuable feedback.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, 1995.
2. ABS Development Team. *ABS Documentation*, v1.8.2 edition, 2020. `http://abs-models.org/manual`.
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. W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. Science of Computer Programming, 77(12):1289–1309, 2012.
6. 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.
7. 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.
8. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2007.
9. 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.
10. 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.
11. P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 1.4 edition, 2010.
12. 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.
13. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In *Java Card Workshop*, volume 2041 of *LNCS*, pages 6–24. Springer, 2000.
14. B. Beckert, V. Klebanov, and B. Weiß. Dynamic logic for Java. 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*, volume 10001 of *LNCS*, pages 49–106. Springer, 2016.

15. S. Blom, M. Huisman, and M. Mihelcic. Specification and verification of GPGPU programs. *Sci. Comput. Program.*, 95:376–388, 2014.
16. 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.
17. S. Brookes and P. W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, Aug. 2016.
18. 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.
19. 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.
20. O.-J. Dahl. Monitors revisited. In A. W. Roscoe, editor, *A classical Mind: Essays in Honour of C.A.R. Hoare*, pages 93–103. Prentice Hall, 1994.
21. 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.
22. 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.
23. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. D. Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP, Braga, Portugal*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
24. 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.
25. C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. Journal of Logic and Algebraic Programming, 81(3):227–256, 2012.
26. C. C. Din, R. Hähnle, E. B. Johnsen, V. K. I. Pun, and S. L. Tapia Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In C. Nalon and R. Schmidt, editors, *Proc. 26th Intl. Conf. on Automated Reasoning with Tableaux and Related Methods*, volume 10501 of *LNCS*, pages 22–43. Springer, Sept. 2017.
27. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
28. 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.
29. 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.
30. C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.
31. 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.

32. I. Gavran, F. Niksic, A. Kanade, R. Majumdar, and V. Vafeiadis. Rely/Guarantee Reasoning for Asynchronous Programs. In L. Aceto and D. de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 483–496. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.
33. R. Hähnle and M. Huisman. Deductive verification: from pen-and-paper proofs to industrial tools. In B. Steffen and G. Woeginger, editors, *Systems*, volume 10000 of *LNCS*. Springer, 2017.
34. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.
35. P. B. Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, 1972.
36. P. B. Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.
37. C. A. R. Hoare. Towards a theory of parallel programming. *Operating System Techniques*, pages 61–71, 1972.
38. 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.
39. H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.
40. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.
41. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
42. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, Oct. 1983.
43. E. Kamburjan. Behavioral program logic. In *TABLEAUX*, volume 11714 of *LNCS*, pages 391–408. Springer, 2019.
44. E. Kamburjan. Behavioral program logic and LAGC semantics without continuations (technical report). *CoRR*, abs/1904.13338, 2019.
45. E. Kamburjan. *Modular Verification of a Modular Specification: Behavioral Types as Program Logics*. PhD thesis, Technische Universität Darmstadt, 2020.
46. E. Kamburjan and T. Chen. Stateful behavioral types for active objects. In *IFM*, volume 11023 of *LNCS*, pages 214–235. Springer, 2018.
47. 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.
48. E. Kamburjan, C. C. Din, R. Hähnle, and E. B. Johnsen. Asynchronous cooperative contracts for cooperative scheduling. In *SEFM*, volume 11724 of *LNCS*, pages 48–66. Springer, 2019.
49. 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/.

50. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: a software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
51. 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.
52. K. R. M. Leino and V. Wüstholz. The Dafny integrated development environment. In C. Dubois, D. Giannakopoulou, and D. Méry, editors, *Proc. 1st Workshop on Formal Integrated Development Environment, F-IDE, Grenoble, France*, volume 149 of *EPTCS*, pages 3–15, 2014.
53. J. Lin, I. C. Yu, E. B. Johnsen, and M. Lee. ABS-YARN: A formal framework for modeling hadoop YARN clusters. In P. Stevens and A. Wasowski, editors, *Fundamental Approaches to Software Engineering, 19th Intl. Conf. FASE, Eindhoven, The Netherlands*, volume 9633 of *LNCS*, pages 49–65. Springer, 2016.
54. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.
55. K. Nakata and T. Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of While. *Logical Methods in Computer Science*, 11(1):1–32, 2015.
56. 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.
57. W. Oortwijn, S. Blom, and M. Huisman. Future-based static analysis of message passing programs. In D. A. Orchard and N. Yoshida, editors, *Proceedings of the Ninth workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2016, Eindhoven, The Netherlands, 8th April 2016*, volume 211 of *EPTCS*, pages 65–72, 2016.
58. A. J. Summers and P. Müller. Actor services - modular verification of message passing programs. In P. Thiemann, editor, *Proceedings of the 25th European Symposium on Programming (ESOP 2016)*, volume 9632 of *LNCS*, pages 699–726. Springer, 2016.
59. B. Weiß. *Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.
60. P. Y. H. Wong, N. Diakov, and I. Schaefer. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study (invited paper). In *2nd International Conference on Formal Verification of Object-Oriented Software, Torino, Italy*, volume 7421 of *LNCS*. Springer, 2012.
61. 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.
62. 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.