Behavioral Program Logic*

Eduard Kamburjan

Department of Computer Science, Technische Universität Darmstadt, Germany
kamburjan@cs.tu-darmstadt.de

Abstract. We present Behavioral Program Logic (BPL), a dynamic logic for trace properties that incorporates concepts from behavioral types and allows reasoning about non-functional properties within a sequent calculus. BPL uses *behavioral modalities* $[\mathtt{s} \Vdash \tau]$, to verify statements $\mathtt{s}$ against *behavioral specifications* τ. Behavioral specifications generalize postconditions and behavioral types. They can be used to specify other static analyses, e.g., data flow analyses. This enables deductive reasoning about the results of multiple analyses on the same program, potentially implemented in different formalisms. Our calculus for BPL verifies the behavioral specification gradually, as common for behavioral types. This vastly simplifies specification, calculus and composition of local results. We present a sequent calculus for object-oriented actors with futures that integrates a pointer analysis and bridges the gap between behavioral types and deductive verification.

1 Introduction

When reasoning about concurrent programs, the intermediate states of an execution are of more relevance than when reasoning about sequential programs. In an object-oriented setting, it does not suffice to specify pre- and postcondition of some method `m`. Instead, the *traces* generated by `m` must be specified.

Recently, dynamic logics for trace properties have been developed [3, 7, 12] to leverage well-established verification techniques from dynamic logic [1] to a concurrent setting. The application of these approaches to real world models of distributed systems [13, 25] revealed two shortcomings: (1) the composition of method-local verification results to a guarantee for the whole system is not automatic and (2) the specification of trace properties is too complex. Thus, the current approaches are deemed as not practical for serious verification efforts.

Another group of verification techniques, *behavioral types*, aim "*to describe properties associated with the behavior of programs and in this way also describe how a computation proceeds.*" [20]. For object-oriented languages, behavioral types can also be seen as specifications of traces of methods. Behavioral types, especially session types [19], are restricted in their expressive power to easily compose their local results to global guarantees, and are natural specifications for protocols. However, they lack precision when handling state [5] or require

* This work is supported by the `FormbaR` project, part of AG Signalling/DB RailLab.

additional static analyses [23]. For Active Objects [9] (object-oriented actors with futures), a translation from session types to a trace logic has been given [23].

We introduce *Behavioral Program Logic* (BPL) to combine precise state reasoning from program logics with the relative simplicity of behavioral types and enable the integration of static analyses into deductive reasoning. The main difference to previous approaches in dynamic logic for trace properties is the *behavioral modality* $[\mathtt{s} \Vdash \tau]$, which expresses that all traces of statement `s` satisfy specification τ. The specification τ is not a formula, as the postcondition of modalities in classical dynamic logic, but is a specification translated into a monadic second order formula over traces. Similarly to behavioral types, τ may contain syntactic elements and allows to syntactically match with `s`. Sequent calculi for BPL may reduce `s` *and* τ in one rule. Contrary to previous dynamic logics for traces, behavioral specifications are more succinct and easier to compose and decompose by, e.g., using the projection mechanism of session types.

We distinguish between behavioral types, that have a sequent calculus of the above kind, and *behavioral specifications*, which do not. Behavioral specifications interface with external properties, such as a data-flow points-to analysis. Beyond integrating external analyses into the sequent calculus, this modularizes the sequent calculus by expressing different properties with different behavioral specifications. Behavioral specifications are clear interfaces that allow to close proofs once more context is known and generalize proof repositories [6].

Our main contributions are (1) BPL, a trace program logic that integrates deductive reasoning with static analyses (2) *method types*, a behavioral type in BPL that generalizes method contracts, object invariants and local types for Active Objects. Due to space constraints, we do not give (de-)compositions and full semantics and refer to [23] and our technical report [22] for full details. We introduce our programming language in Sec. 2 and BPL in Sec. 3. In Sec. 4 we introduce method types. Sec. 5 summarizes previous approaches and concludes.

2 Preliminaries: An Actor Language with Futures

We introduce Behavioral Program Logic using a Core Active Object language [9] (*CAO*) with futures, *CAO* uses strong encapsulation (i.e., all fields are object-private) and cooperative scheduling. *CAO* is based on ABS [21] and we use a locally abstract, globally concrete (LAGC) semantics [11]. An LAGC semantics consists of two layers: A locally abstract (LA) layer for statements and methods, and a globally concrete (GC) layer for objects and systems. The LA layer is a denotational semantics that abstractly describes the behavior of a method in every possible context, while the GC layer is an operational semantics that concretizes the LA semantics of processes in a concrete context. LA semantics enable one to analyze a method in isolation and Active Objects allow us to demonstrate that BPL is suited for complex concurrency models.

Definition 1 (Syntax). *Let* $\sim$ *range over* `&&`, `||`, `+`, `-`, `*`, `/`, `>=`, `>`, `<`, `<=`, `v` *over variables,* `f` *over fields,* `C` *over class names,* `m` *over method names,* i *and* `n` *over* $\mathbb{N}$. *The* syntax *of CAO is defined in Fig. 1, where* $\overrightarrow{\cdot}$ *denotes (possibly empty) lists.*

```
Prgm ::= Class* Main   Main ::= main{si}   Class ::= class C (C f*){Field* Meth*}   Field ::= D f = e;
Meth ::= D m(D v*){s;return e;}   D ::= Int | Bool | Fut<D>   si ::= C v = C(e*); si | v!m(e*)
  s ::= [D] l = e | [D] v = e.get_i | [D] v = f!m(e*) | skip | while(e){s} | if(e){s}else{s} | s;s
  e ::= l | n | True | False | e ~ e | !e | -e   l ::= this.f | v
```

Fig. 1. Syntax of *CAO*.

A program consists of a set of classes and a main block. The main block contains object instantiations and a method call to initialize the communication. All objects are created at once, not in the order of their instantiations. A class contains (1) parameter fields which reference other objects, (2) fields for data, initialized upon creation and (3) methods. Multiple instances can share their parameters. Parameters cannot be reassigned. As data types we use integers, booleans and parametric futures. Each class has a `run` method that is started upon creation. We omit `run` if it is empty.

```
1  class T(Comp S, Log L){
2   Int test(Int i){
3    Fut<Int> f = S!cmp(i);
4    Int r = f.get_0;
5    if(r < 0){
6     r = -r;
7     f = L!log(i);
8    }
9    return r;
10  }
11 }
```

Fig. 2. An example method

Statement `v = f!m(e*)` calls method `m` asynchronously on the object `f` with parameters `e*`. A fresh future is generated and stored in `v`. This future identifies the called process. We say that the called process will *resolve* the future by executing `return e` and storing the value of `e` in the future. The synchronizing statement `v = e.get`$_i$ reads from the future in `e` into `v`. Until the future is resolved, the reading process blocks its object. The identifier i is used to distinguish multiple synchronization points. The other statements, expressions and methods are standard.

Example 1. Fig. 2 shows a simple method that passes its input to `Comp.cmp` and reads the result. If the result is negative, its sign is inverted and the original input data is logged by `Log.log`. The possibly inverted result is returned.

The semantics of a method is a set of symbolic traces, to describe the behavior of the method in every possible context, i.e., for every possible heap, call parameters and accessed futures. Additionally to semantic values (semantic values are, e.g., object identifiers, rationals, futures etc.), symbolic traces contain symbolic expressions. Structurally, symbolic expressions mirror syntactic expressions, but do not contain variables or fields. Instead they contain symbolic values and symbolic fields. Symbolic values have no operations defined on them and act as placeholders. They are replaced by semantic values once the method is running and the context is known. Symbolic fields are special symbolic values that contain the name of the field they are abstracting.

Definition 2 (Symbolic Expressions). *Let v range over semantic values, $\underline{v}$ over symbolic values, i over $\mathbb{N}$ and* **this**`.f`$_i$ *over symbolic fields.* Symbolic expres-

sions $\underline{\mathtt{e}}$ *are defined below. We highlight symbolic elements by underlining.*

$$\underline{\mathtt{e}} ::= \underline{\mathtt{e}} \sim \underline{\mathtt{e}} \mid \mathtt{!}\underline{\mathtt{e}} \mid \mathtt{-}\underline{\mathtt{e}} \mid \underline{v} \mid v \mid \mathbf{this}.\mathtt{f}_i$$

To model the points where processes, objects and futures interact, traces contain events as markers for visible communication.

Definition 3 (Events). Events *are defined by the following grammar.*

$$\mathsf{ev} ::= \mathsf{invEv}(\underline{\mathtt{X}}, \underline{\mathtt{X}'}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{\mathtt{e}}}) \mid \mathsf{invREv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{\mathtt{e}}}) \mid \mathsf{futEv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \underline{\mathtt{e}}) \mid \mathsf{futREv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \underline{\mathtt{e}}, i) \mid \mathsf{noEv}$$

Event $\mathsf{invEv}(\underline{\mathtt{X}}, \underline{\mathtt{X}'}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{\mathtt{e}}})$ models a call from $\underline{\mathtt{X}}$ to $\underline{\mathtt{X}'}$ on method $\mathtt{m}$ with future $\underline{f}$ and call parameters $\overrightarrow{\underline{\mathtt{e}}}$. The future and the callee may be symbolic: locally it is not possible to know the used future and the called object. Event $\mathsf{invREv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{\mathtt{e}}})$ is the callee view on a call. The object $\underline{\mathtt{X}}$ here is the callee, the caller is not visible to the callee. Event $\mathsf{futEv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \underline{\mathtt{e}})$ models the termination of a process for future $\underline{f}$, computing method $\mathtt{m}$ in object $\underline{\mathtt{X}}$ and returning $\underline{\mathtt{e}}$. Event $\mathsf{futREv}(\underline{\mathtt{X}}, \underline{f}, \underline{\mathtt{m}}, \underline{\mathtt{e}}, i)$ models a $\mathbf{get}_i$ statement in object $\underline{\mathtt{X}}$ on the future $\underline{f}$, which was computed by $\mathtt{m}$ and returned $\underline{\mathtt{e}}$. Finally, noEv models an internal step.

Local traces consist of a *selection condition*, a set of symbolic expressions that express when a trace executes and a *history*, a sequence of events and states.

Definition 4 (Local Semantics and Traces). *A* heap ρ *maps from fields to symbolic expressions and a* local state σ *maps variables to symbolic expressions. Pairs of local states and heaps are* object states *and we write* $\binom{\sigma}{\rho}$*. The evaluation function* $[\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}}$ *maps a syntactic expression to a symbolic expression.*

A local trace θ *has the form* $\mathsf{sc} \triangleright \mathsf{hs}$*, where* sc *is a set of symbolic expressions, called* selection condition*, and* hs *is a non-empty sequence, called* history*, such that every odd-indexed element is an object state and every even-indexed element an event. The semantics of methods and statements is defined by a function* $[\![\cdot]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}}$ *where* $\mathtt{X}$ *is the object name,* f *the future the method is resolving,* $\mathtt{m}$ *the method name and* $\binom{\sigma}{\rho}$ *the current object state. Future, object name and state may be symbolic. The semantics of a method* $\mathtt{m}$ *with body* $\mathtt{s}$ *is, for a symbolic* $\binom{\sigma}{\rho}$*:*

$$[\![\mathtt{m}]\!]_{\underline{\mathtt{X}},\underline{f},\mathtt{m},\binom{\sigma}{\rho}} = \left\{ \emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{invREv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \overrightarrow{\mathtt{e}}) \right\rangle \circ \theta \mid \theta \in [\![\mathtt{s}]\!]_{\underline{\mathtt{X}},\underline{f},\mathtt{m},\binom{\sigma}{\rho}} \right\}$$

where $\overrightarrow{\mathtt{e}}$ *is extracted from the parameter names in* σ*. E.g., for a method* `Int m( Int a, Rat b)` *we set* $\overrightarrow{\mathtt{e}} = \langle \sigma(\mathtt{a}), \sigma(\mathtt{b}) \rangle$*. Fig. 3 shows selected rules. All variables are initialized, futures with* **no***, a special future that is never resolved.*

The rules for assignment both update the local state or the heap, and add a noEv event. The rule for branching evaluates both branches and adds the corresponding guard evaluation to the selection condition. The rule for **get** is similar to the variable assignment, but receives a fresh symbolic value and stores it in the local state. As the event, a resolving reaction event is added, which stores the accessed future and the fresh symbolic value. The rule for method

$$[\![\texttt{v = e}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{noEv}, \binom{\sigma[\mathtt{v} \mapsto [\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}}]}{\rho} \right\rangle\right\} \qquad [\![\textbf{this}.\mathtt{f}]\!]_{\binom{\sigma}{\rho}} = \rho(\mathtt{f})$$

$$[\![\mathtt{v}]\!]_{\binom{\sigma}{\rho}} = \sigma(\mathtt{v})$$

$$[\![\textbf{this}.\mathtt{f = e}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{noEv}, \binom{\sigma}{\rho[\mathtt{f} \mapsto [\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}}]} \right\rangle\right\} \qquad [\![\mathtt{-e}]\!]_{\binom{\sigma}{\rho}} = \begin{cases} -[\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}} \text{ if } [\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}} \text{ symbolic} \\ [\![-[\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}}]\!] \text{ otherwise} \end{cases}$$

$$[\![\textbf{return } \mathtt{e}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{futEv}\left(\mathtt{X}, f, \mathtt{m}, [\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}}\right), \binom{\sigma}{\rho} \right\rangle\right\} \qquad [\![\textbf{skip}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho} \right\rangle\right\}$$

$$[\![\textbf{if}(\mathtt{e})\{\mathtt{s}\}\textbf{else}\{\mathtt{s'}\}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\mathsf{sc} \cup \{[\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}}\} \triangleright \mathsf{hs} \middle| \mathsf{sc} \triangleright \mathsf{hs} \in [\![\mathtt{s}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}}\right\} \cup \left\{\mathsf{sc} \cup \{[\![\mathtt{!e}]\!]_{\binom{\sigma}{\rho}}\} \triangleright \mathsf{hs} \middle| \mathsf{sc} \triangleright \mathsf{hs} \in [\![\mathtt{s'}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}}\right\}$$

$$[\![\mathtt{v = e.}\textbf{get}_i]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{futREv}\left(\mathtt{X}, [\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}}, \mathtt{m}, \underline{v}, i\right), \binom{\sigma[\mathtt{v} \mapsto \underline{v}]}{\rho} \right\rangle\right\} \text{ where } \underline{v} \text{ is fresh}$$

$$[\![\mathtt{v = f!n}(\overrightarrow{\mathtt{e}})]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{invEv}\left(\mathtt{X}, [\![\mathtt{f}]\!]_{\binom{\sigma}{\rho}}, \underline{v}, \mathtt{n}, \overrightarrow{[\![\mathtt{e}]\!]_{\binom{\sigma}{\rho}}}\right), \binom{\sigma[\mathtt{v} \mapsto \underline{v}]}{\rho} \right\rangle\right\} \text{ where } \underline{v} \text{ is fresh}$$

$$[\![\textbf{while}(\mathtt{e})\{\mathtt{s}\}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = [\![\textbf{if}(\mathtt{e})\{\mathtt{s};\textbf{while}(\mathtt{e})\{\mathtt{s}\};\textbf{skip}\}\textbf{else}\{\textbf{skip}\}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}}$$

$$[\![\mathtt{s;s'}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\mathsf{sc} \cup \mathsf{sc}' \triangleright \mathsf{hs} \circ \mathsf{hs}' \middle| \mathsf{sc} \triangleright \mathsf{hs} \circ \left\langle \binom{\sigma'}{\rho'} \right\rangle \in [\![\mathtt{s}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}}, \mathsf{sc}' \triangleright \mathsf{hs}' \in [\![\mathtt{s'}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma'}{\rho'}}\right\}$$

Fig. 3. Selected rules of the LA semantics of statements and expression. Evaluation $[\![\mathtt{e}]\!]$ of semantic values has its natural definition.

$\{\underline{v}<0\}\triangleright$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \mathbf{no}, \mathtt{r} \mapsto 0\}, \rho), \mathsf{invREv}(\underline{X}, \underline{f}, \mathtt{T.test}, \langle \underline{i} \rangle)\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \mathbf{no}, \mathtt{r} \mapsto 0\}, \rho), \mathsf{invEv}(\underline{X}, \underline{S}, \underline{f'}, \mathtt{Comp.cmp}, \langle \underline{i} \rangle)\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \underline{f'}, \mathtt{r} \mapsto 0\}, \rho), \mathsf{futREv}(\underline{X}, \underline{f'}, \mathtt{Comp.cmp}, \underline{v}, 0)\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \underline{f'}, \mathtt{r} \mapsto \underline{v}\}, \rho), \mathsf{noEv}\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \underline{f'}, \mathtt{r} \mapsto -\underline{v}\}, \rho), \mathsf{invEv}(\underline{X}, \underline{L}, \underline{f''}, \mathtt{Log.log}, \langle \underline{i} \rangle)\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \underline{f''}, \mathtt{r} \mapsto -\underline{v}\}, \rho), \mathsf{futEv}(\underline{X}, \underline{f}, \mathtt{T.test}, -\underline{v})\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \underline{f''}, \mathtt{r} \mapsto -\underline{v}\}, \rho)\rangle$

$\{\underline{v}>=0\}\triangleright$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \mathbf{no}, \mathtt{r} \mapsto 0\}, \rho), \mathsf{invREv}(\underline{X}, \underline{f}, \mathtt{T.test}, \langle \underline{i} \rangle)\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \mathbf{no}, \mathtt{r} \mapsto 0\}, \rho), \mathsf{invEv}(\underline{X}, \underline{S}, \underline{f'}, \mathtt{Comp.cmp}, \langle \underline{i} \rangle)\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \underline{f'}, \mathtt{r} \mapsto 0\}, \rho), \mathsf{futREv}(\underline{X}, \underline{f'}, \mathtt{Comp.cmp}, \underline{v}, 0)\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \underline{f'}, \mathtt{r} \mapsto \underline{v}\}, \rho), \mathsf{futEv}(\underline{X}, \underline{f}, \mathtt{T.test}, \underline{v})\rangle \circ$
$\langle(\sigma \cup \{\mathtt{f} \mapsto \underline{f'}, \mathtt{r} \mapsto \underline{v}\}, \rho)\rangle$

Fig. 4. LA semantics of `T.test`, with $\sigma = \{\mathtt{i} \mapsto \underline{i}\}, \rho = \{\mathtt{S} \mapsto \underline{S}, \mathtt{L} \mapsto \underline{L}\}$.

calls is analogous, but uses a fresh future for the call instead of a fresh read value. The added event is an invocation event with the evaluated parameters. Fig. 4 shows the two traces in the semantics of Ex. 1.

Symbolic traces represent a set of concrete traces, which contain only semantic values and correspond to possible behaviors of the statement. The set of concrete traces represented by a symbolic trace is a vast overapproximation and we only consider *selected* traces: concrete traces used in some terminating run of a given program. For a formal definition and the GC semantics, we refer to [22].

Definition 5 (Selected Traces). *A trace θ continues trace θ', written $\theta \preceq \theta'$, if its history is a suffix of the history of θ', with all symbolic elements replaced by concrete values, such that this substitution evaluates all expressions in the selection condition to true. A trace θ is* selected *in a program* Prgm*, if it is used during some run of* Prgm*. Let* `m` *be the method containing* `s`*.*

$$[\![\mathtt{s}]\!]^{\mathsf{Prgm}}_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \{\theta \in [\![\mathtt{s}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} \mid \exists \theta' \in [\![\mathtt{m}]\!]_{\underline{\mathtt{X}},\underline{f},\mathtt{m},\binom{\sigma}{\rho}}.\ \theta \preceq \theta' \wedge\ \theta \textit{ used in a terminating run of } \mathsf{Prgm}\}$$

We use a first-order state (FOS) logic to express properties of states and a monadic second-order (MSO) logic to express properties of traces. The MSO logic embeds the FOS by using FOS formulas similar to predicates on states. Similarly, it uses terms that allow to specify events.

Definition 6 (FOS Syntax). *Let p range over predicate symbols, f over function symbols, x over logical variable names and S over sorts. As sorts we take all data types $\mathtt{D}$, all class names and additionally $\mathbb{N}$ and* **Heap**. *The logical heaps are functions from field names to semantic values. Formulas φ and terms $\mathtt{t}$ are defined by the following grammar, where $\mathtt{v}$ are program variables, consisting of local variables and the special variables* `heap` *and* `result`, *and* `f` *are all field names.*

$$\varphi ::= p(\overrightarrow{\mathtt{t}}) \mid \mathtt{t} \doteq \mathtt{t} \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x \in S.\ \varphi \qquad \mathtt{t} ::= x \mid \mathtt{v} \mid \mathtt{f} \mid f(\overrightarrow{\mathtt{t}})$$

We demand the usual constants, (e.g., 0, `True`) and that each operator defined in syntactic expressions `e` is a function symbol, so one can directly translate a syntactic expression into a FOS term. We additionally assume the following function symbols to handle heaps: $\mathsf{select}(\mathtt{t},\mathtt{t}) \mid \mathsf{store}(\mathtt{t},\mathtt{t},\mathtt{t})$, where $\mathsf{select}(h,\mathtt{f})$ reads field $\mathtt{f}$ from heap h and $\mathsf{store}(h,\mathtt{f},\mathtt{t})$ stores the value of $\mathtt{t}$ in field $\mathtt{f}$ of heap h. As only one object is considered, we do not require an object parameter.

Definition 7 (FOS Semantics). *Interpretation I maps function names to functions and predicate names to predicates. Assignment β maps logical variable to semantic values of the resp. sort. Evaluation of terms in state $\binom{\sigma}{\rho}$ is defined as a function $[\![\mathtt{t}]\!]_{\binom{\sigma}{\rho},I,\beta}$ and satisfiability of formulas by a relation $\binom{\sigma}{\rho},\beta,I \models \varphi$.*

For the special variable `heap` we set $[\![\mathtt{heap}]\!]_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} = \rho$ and for the heap functions we follow JavaDL [1] and demand, e.g., the following connection axiom for all heaps h, all fields $\mathtt{f}$ and terms $\mathtt{t}$: $I(\mathsf{select})(I(\mathsf{store})(h,\mathtt{f},\mathtt{t}),\mathtt{f}) = \mathtt{t}$.

The models for the MSO logic are local traces and the whole semantic domain. This allows to quantify over method names etc. – it is not a logic over finite sequences. In additional to standard MSO constructs, we use $[\mathsf{t_{tr}}] \doteq \mathtt{t}$ to say that the event at position $\mathsf{t_{tr}}$ of the trace is equal to the term $\mathtt{t}$. Similarly, $[\mathsf{t_{tr}}] \vdash \varphi$ expresses that the state at position $\mathsf{t_{tr}}$ is a model for the FOS formula φ.

Definition 8 (MSO Syntax). *Let p,f,x range over the same sets as before, S over sorts. As sorts we take all data types $\mathtt{D}$ and additionally* **I**, *the set of trace indices,* O, *the set of all object names,* **Fut**, *the set of all futures of all types, the supertype* Any, *the set of all well-typed expressions and* M, *the set of all method names. Formulas ψ are defined as follows. Terms $\mathsf{t_{tr}}$ are standard.*

$$\psi ::= p(\overrightarrow{\mathsf{t_{tr}}}) \mid \psi \vee \psi \mid \neg\psi \mid \mathsf{t_{tr}} \subseteq \mathsf{t_{tr}} \mid \exists x \in S.\ \psi \mid \exists X \subseteq S.\ \psi \mid [\mathsf{t_{tr}}] \doteq \mathsf{t_{tr}} \mid [\mathsf{t_{tr}}] \vdash \varphi$$

The predicate $\mathsf{isEvent}(i)$ that holds iff $\theta[i]$ is an event. For each type of event, there is a function symbol that maps its parameters to an event of its type and a predicate that holds iff the given position is an event of that kind, e.g.,

$$\mathsf{isfutEv}(i) \iff \exists f \in \mathbf{Fut}.\ \exists o \in \mathsf{O}.\ \exists m \in \mathsf{M}.\ \exists v \in \mathtt{Any}.\ [i] \doteq \mathsf{futEv}(o,f,m,v)$$

Definition 9 (MSO Semantics). *The semantics of terms and event terms is defined by a function* $[\![\cdot]\!]_{I,\beta}$*. The satisfiability of MSO-formulas is defined by a relation* $\theta, I, \beta \models \psi$*. The semantics of our extensions follows.*

$$\theta, I, \beta \models [\mathsf{t}_{\mathsf{tr}1}] \doteq \mathsf{t}_{\mathsf{tr}2} \iff 1 \leq [\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta} \leq |\theta| \wedge \theta[[\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}] = [\![\mathsf{t}_{\mathsf{tr}2}]\!]_{I,\beta}$$
$$\theta, I, \beta \models [\mathsf{t}_{\mathsf{tr}}] \vdash \varphi \iff 1 \leq [\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta} \leq |\theta| \wedge \theta[[\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta}] \textit{ is a state } \wedge \theta[[\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta}], I, \beta \models \varphi$$

Example 2. Let `r = f.get`$_0$ be the statement from Ex. 1. The following MSO-formula expresses that if all values read from futures of `cmp` is positive, and every future read at point 0 is from `cmp`, then after the read the value of `r` is positive.

$$\begin{aligned}&(\forall i \in \mathbf{I}.\ (\forall v \in \texttt{Int}.\ [i] \doteq \mathsf{futREv}(\_, \_, \texttt{cmp}, v, \_) \rightarrow v > 0) \wedge\\ &\forall i \in \mathbf{I}.\ (\forall m \in \mathsf{M}.\ [i] \doteq \mathsf{futREv}(\_, \_, m, \_, 0) \rightarrow m \doteq \texttt{cmp}))\\ &\rightarrow \forall i \in \mathbf{I}.\ ([i] \doteq \mathsf{futREv}(\_, \_, \_, \_, 0) \rightarrow [i+1] \vdash \texttt{r > 0})\end{aligned}$$

Relativization [17], an established technique in abstract model theory [15], syntactically restricts a formula ψ on a substructure defined by another formula ψ'. It is denoted $\psi[x \in S \setminus \psi']$, where x is a free variable in ψ' of S sort. Each quantifier of S sort is restricted to elements that fulfill ψ'.

Example 3. Formula φ expresses that every trace-element is either an event, or a state with `r > 0`. The relativization with ψ expresses that φ holds for every index above 9. Both traces of Fig. 2 satisfy $\varphi[j \in \mathbf{I} \setminus \psi]$, neither satisfies φ.

$$\begin{aligned}\varphi &= \forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \vee [i] \vdash \texttt{r > 0} \qquad \psi = j \geq 9\\ \varphi[j \in \mathbf{I} \setminus \psi] &= \forall i \in \mathbf{I}.\ i \geq 9 \rightarrow (\mathsf{isEvent}(i) \vee [i] \vdash \texttt{r > 0})\end{aligned}$$

We use common abbreviations, e.g., $\forall x \in S.\ \varphi$ for $\neg\exists x \in S.\ \neg\varphi$ and true and shorten comparisons of `Bool` terms by writing, e.g., $i > j$ instead of $i > j \doteq \mathbf{True}$.

3 Behavioral Program Logic

Behavioral Program Logic (BPL) is an extension of FOS with *behavioral modalities* $[\mathtt{s} \Vdash^{\alpha} \tau]$ that contain a statement `s` and a behavioral specification (τ, α). A behavioral specification consists of (1) a syntactic component (the type τ) and (2) a translation α of the type into an MSO formula that has to hold for all traces generated by the statement. Behavioral specifications can be seen as representations of a certain class of MSO formulas, which are deemed useful for verification of distributed systems. For the rest of this section, we assume fixed parameters Prgm, `X`, f, `m` for evaluation.

Definition 10 (Behavioral Program Logic). *A* behavioral specification $\mathbb{T}$ *is a pair* $(\tau_{\mathbb{T}}, \alpha_{\mathbb{T}})$*, where* $\alpha_{\mathbb{T}}$ *maps elements of* $\tau_{\mathbb{T}}$ *to MSO formulas.*

BPL*-formulas* φ*, terms* t *and updates* U *are defined by the following grammar, which extends Def. 6. The meta variables range as in Def. 6. Additionally let* `s` *range over statements and* $(\tau_{\mathbb{T}}, \alpha_{\mathbb{T}})$ *over behavioral specifications.*

$$\varphi ::= \ldots \mid [\mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau_{\mathbb{T}}] \mid \{U\}\varphi \quad \mathtt{t} ::= \ldots \mid \{U\}\mathtt{t} \quad U ::= \epsilon \mid U||U \mid \{U\}U \mid \mathtt{v} := \mathtt{t}$$

$$[\![\{U\}t]\!]_{\binom{\sigma}{\rho},I,\beta} = [\![t]\!]_{[\![U]\!]_{\binom{\sigma}{\rho},I,\beta},I,\beta} \quad [\![\epsilon]\!]_{\binom{\sigma}{\rho},I,\beta}(x) = x \quad \binom{\sigma}{\rho}, I, \beta \models \{U\}\varphi \Leftrightarrow [\![U]\!]_{\binom{\sigma}{\rho},I,\beta}, I, \beta \models \varphi$$

$$[\![\mathtt{v} := \mathtt{t}]\!]_{\binom{\sigma}{\rho},I,\beta}\left(\binom{\sigma'}{\rho'}\right) = \begin{cases} \binom{\sigma'}{\rho''} & \text{if } \mathtt{v} = \mathtt{heap},\ \rho'' = [\![\mathtt{t}]\!]_{\binom{\sigma}{\rho},I,\beta} \\ \binom{\sigma''}{\rho'} & \text{otherwise, } \sigma'' = \sigma'[\mathtt{v} \mapsto [\![\mathtt{t}]\!]_{\binom{\sigma}{\rho},I,\beta}] \end{cases}$$

$$[\![U||U']\!]_{\binom{\sigma}{\rho},I,\beta}(x) = [\![U']\!]_{\binom{\sigma}{\rho},I,\beta}\left([\![U]\!]_{\binom{\sigma}{\rho},I,\beta}(x)\right) \qquad [\![\{U\}U']\!]_{\binom{\sigma}{\rho},I,\beta} = [\![U']\!]_{[\![U]\!]_{\binom{\sigma}{\rho},I,\beta}I,\beta}$$

$$\binom{\sigma}{\rho}, I, \beta \models [\mathtt{s} \Vdash^{\alpha_{\mathbb{T}}} \tau_{\mathbb{T}}] \Leftrightarrow \forall \theta \in [\![\mathtt{s}]\!]^{\mathsf{Prgm}}_{\mathtt{X},f,\mathtt{m},\binom{\sigma}{\rho}} .\ \theta, I, \beta \models \alpha_{\mathbb{T}}(\tau_{\mathbb{T}})$$

Fig. 5. Semantics of BPL. The satisfiability relation on the right of the semantics of behavioral modalities is the one of MSO.

The semantics of a behavioral modality $[\mathtt{s} \Vdash^{\alpha_{\mathbb{T}}} \tau_{\mathbb{T}}]$ is that all traces generated by $\mathtt{s}$ *selected within* Prgm are models for $\alpha_{\mathbb{T}}(\tau_{\mathbb{T}})$. We use updates $[2, 1]$ to keep track of state changes, their semantics is a state transition. Update $\mathtt{v} := \mathtt{t}$ changes the state by updating $\mathtt{v}$ to $\mathtt{t}$. The parallel update $U||U'$ applies U and U' in parallel, with U' winning in case of clashes. ϵ is the empty update and application $\{U\}$ evaluates the term (resp. formula) in the state after applying U.

Definition 11 (Semantics of BPL). *The semantical extension of FOS to* BPL *is given in Fig. 5. The interpretation I has the properties described above. A formula φ is* valid *if every $\binom{\sigma}{\rho}$ and every β make it true.*

Object, program, method name, resolved future and type of `result` are implicitly known, but we omit them for readability's sake. We use a sequent calculus to reason about BPL (resp. FOS).

Definition 12 (Sequents and Rules). *Let Δ, Γ be sets of* BPL*-formulas. A* sequent $\Gamma \Rightarrow \Delta$ *has the semantics of $\bigwedge \Gamma \to \bigvee \Delta$. Γ is called the antecedent and Δ the succedent. Let C, P_i be sequents. A rule has the form*

$$(\textsf{name})\ \frac{P_1 \quad \dots \quad P_n}{C}\ cond$$

Where C is called the conclusion and P_i the premise, while cond is a side-condition. Side-conditions are always decidable. For readability's sake, we apply side conditions containing equalities directly in the premises.

Rules may contain, in addition to expressions, schematic variables. Their handling is standard [1]. We assume the usual FO rules for the FOS part of BPL handling all FO operators such as quantifiers.

Definition 13 (Soundness). *A rule is* sound *if validity of all premisses implies validity of the conclusion.*

Soundness implicitly refers to a program Prgm, as behavioral modalities are defined over Prgm-selectable traces. Rewrite rules $\tau_1 \rightsquigarrow \tau_2$ syntactically replace one type τ_1 by another, τ_2 (and vice versa) and are sound if $\alpha(\tau_1) \equiv \alpha(\tau_2)$.

Discussion. Before we introduce method types, a particular behavorial specification, we illustrate BPL with further examples. To reason about postconditions, as standard modal logics, we define a behavioral specification that only uses the last state of a trace (denoted by the function symbol *last*) for its semantics.

Example 4. The specification for postconditions is the pair of the set of all FOS sentences and the function pst, defined below. T is the type of `result`. The first case accesses the return value stored in the futEv when `result` is used.

$$\mathsf{pst}(\varphi) = \begin{cases} \exists v \in \mathsf{T}.\ [last-1] \doteq \mathsf{futEv}(\_,\_,\_,v) \wedge [last] \vdash \varphi[\texttt{result} \backslash v] & \text{if } \varphi \text{ contains } \texttt{result} \\ [last] \vdash \varphi & \text{otherwise} \end{cases}$$

A Hoare triple $\{\varphi\}\mathtt{s}\{\psi\}$ has the same semantics as the formula $\varphi \rightarrow [\mathtt{s} \Vdash^{\mathsf{pst}} \psi]$. A standard dynamic logic modality $[\mathtt{s}]\psi$ has the same semantics as the behavioral modality $[\mathtt{s} \Vdash^{\mathsf{pst}} \psi]$ [1]. Behavioral modalities generalize these systems and can be used to express any (MSO) trace property, independent of the form of its verification system. The following defines a points-to analysis for futures [14] (for the next statement), normally implemented in a data-flow framework.

Example 5 (Points-To). The behavioral specification of a *points-to analysis* specifies that the next statement reads a future resolved by a method from set M.

$$\mathbb{T}_{\mathsf{p2}} = (\mathcal{P}(\mathsf{M}), \mathsf{p2}) \text{ with}$$

$$\mathsf{p2}(M) = \exists \mathtt{X} \in \mathsf{O}. \exists f \in \textbf{Fut}. \exists \mathtt{m} \in \mathsf{M}. \exists v \in \mathsf{Any}. \exists i \in \mathbb{N}.\ [1] \doteq \mathsf{futREv}(\mathtt{X}, f, \mathtt{m}, v, i) \wedge \bigvee_{\mathtt{m}' \in M} \mathtt{m} \doteq \mathtt{m}'$$

The following formula expresses that the **get** statement reads a positive number, if the future is resolved by `Comp.cmp`. This is the case if `Comp.cmp` always returns positive values. The identifier connects the two modalities semantically.

$$\varphi_p = [\texttt{r = f.}\textbf{get}_0 \overset{\mathsf{p2}}{\Vdash} \{\texttt{Comp.cmp}\}] \rightarrow [\texttt{r = f.}\textbf{get}_0 \overset{\mathsf{pst}}{\Vdash} \texttt{r} > 0]$$

It is not necessary to include postcondition reasoning. Rule **(ex1)** in Fig. 6 expresses that if the next read from `s` is from some set E' and it is required to show that the next read is from E, it suffices to check whether E is a subset of E'. Rule **(ex-⊩)** connects two analyses and generalizes Ex. 2: one may assume some formula ψ for a read value, if this synchronization always reads from method `Comp.cmp` and that the method body of `Comp.cmp` establishes ψ.

The above example illustrates the difference between modalites and typing judgments. Modalites are formulas and can be used for deductive reasoning about a type judgment (which, in our case, is encoded into $\Vdash$). While a calculus for pst is easily carried over from other sequent calculi, this is not possible for all behavioral specifications. The proof can still be closed in two ways.

– There may be some rules, such as **(ex1)** above, that enable to reason about the analysis without reducing the statement at all.

[1] This justifies our use of the term "modality". Contrary to standard modalities, behavioral modalities are not formulas that express modal statements about *formulas*, but formulas that express a modal statement about *more general specifications*.

$$\textbf{(ex1)}\ \frac{}{\Gamma, [\mathtt{s} \overset{p2}{\Vdash} E'] \Rightarrow [\mathtt{s} \overset{p2}{\Vdash} E], \Delta}\ E \subseteq E' \qquad \textbf{(ex-}\Vdash\textbf{)}\ \frac{\Gamma, \psi(v)\ \Rightarrow\ \{\mathtt{v} := v\}[\mathtt{s} \overset{pst}{\Vdash} \varphi], \Delta \quad \Rightarrow\ [\mathtt{v = f.get}_0 \overset{p2}{\Vdash} \{\mathtt{Comp.cmp}\}] \wedge [\mathtt{s}_{\mathtt{Comp.cmp}} \overset{pst}{\Vdash} \psi]}{\Gamma\ \Rightarrow\ [\mathtt{v = f.get}_0\mathtt{;s} \overset{pst}{\Vdash} \varphi], \Delta}\ v \text{ fresh}$$

Fig. 6. Two example rules for behavioral specifications. $\psi(v)$ replaces `result` by v and we assume that ψ contains no fields.

– If the proof contains only open branches containing behavioral specification, one may run a static analysis to evaluate them to true or false directly. E.g., if for the formula φ_p above the pointer analysis returns that the synchronization point 0 reads from `L.log`, the first behavioral modality evaluates to false and the whole formula to true.

Using external analyses increases modularity: (1) the BPL-calculus is simpler because it does not need to encode the implementation and (2) one may verify functional correctness of a method *up to its context*. Open branches are then a description of the context which the method requires. This may be verified once more context is known, thus extending proof repositories [6] to external analyses.

4 A Sequent Calculus for BPL: Behavioral Types

In this section we characterize behavioral types as behavioral specifications with a set of sequent calculus rules and a constraint on the proof obligations of the methods within a program. Before we formalize this in general, we introduce method types [23, 24], a behavioral type for Active Objects that suffices to generalize method contracts and object invariants by integrating the behavioral specifications for postcondition reasoning and points-to analysis. The method type of a method describes the local view of a method on a protocol.

Definition 14. *The local protocol* $\mathbf{L}$ *and method type* L *of a method are defined by the grammar below. The behavioral specification for method types is* $\mathbb{T}_{\mathsf{met}} = (\mathsf{L}, \alpha_{\mathsf{met}})$*. Let* $\mathtt{X}_0, \ldots, \mathtt{X}_n$ *be roles, and* $\mathtt{f}_{\mathtt{X}_0}, \ldots, \mathtt{f}_{\mathtt{X}_n}$ *fields of fitting type.* $\alpha_{\mathsf{met}}(\mathsf{L})$ *is defined as* $\exists \mathtt{X}_0, \ldots, \mathtt{X}_n \in \mathsf{O}.\ \bigwedge_{i \leq n} \mathtt{X}_i \doteq \mathtt{f}_{\mathtt{X}_i} \wedge \alpha'_{\mathsf{met}}(\mathsf{L})$*. The first part models the (generated [24]) assignment of roles (as function symbols) to fields.*

$$\mathbf{L} ::= ?\mathtt{m}(\varphi).\mathsf{L} \qquad \mathsf{L} ::= \mathtt{X}!\mathtt{m}(\varphi) \mid \downarrow(\varphi) \mid \mathsf{skip} \mid \mathsf{L}.\mathsf{L} \mid \mathsf{L}^* \mid \oplus\{\mathsf{L}_i\}_{i \in I} \mid \&(\overrightarrow{\mathtt{m}}, \varphi)\{\mathsf{L}, \mathsf{L}\}$$

The local protocol of a method contains the receiving action $?\mathtt{m}(\varphi)$, which models that the parameters satisfy the predicate φ. The method body is checked against the method type – there is no statement corresponding to receiving. Roles keep track of an object through the protocol. We stress that statements and method types share syntactic elements – it is possible to pattern match on statements/expressions on one side and a method type on the other side in rules.

Calls are specified with the call action $\mathtt{X}!\mathtt{m}(\varphi)$, where `X.m` is the receiver and the predicate φ has to hold. Here, φ does not only specify the sent data but

also local variables and fields. It can express properties such as "the sent data is larger then some field". The termination action $\downarrow(\varphi)$ models termination in a state satisfying φ (which again may include `result`). The empty action skip models no visible actions and $\mathsf{L}_1.\mathsf{L}_2$ to sequential composition: all interactions in L_1 must happen before L_2. Repetition L^* corresponds to the Kleene star (and loops) and models zero or more repetitions of the interactions in L.

There are two choice operators: $\oplus\{\mathsf{L}_i\}_{i\in I}$ is the active choice, the method must select one branch L_i. It is not necessary to implement all branches, the method may choose to never select some branches. The index set I must not be empty. $\&(\overrightarrow{\mathtt{m}}, \varphi)\{\mathsf{L}_1, \mathsf{L}_2\}$ is the passive choice: some other method made a choice and this method has to follow the protocol according to this choice. The choice is communicated via a future which has to be resolved by one of the methods in $\overrightarrow{\mathtt{m}}$. If the choice condition φ, which may only include the program variable `result`, is fulfilled by the read data, L_1 has to be followed, otherwise L_2 has to be followed. Both branches have to be implemented.

The semantics of the call and termination actions specify a trace with at least three elements with the correct event on second position and a state fulfilling the given predicate on the third position. Every other event is noEv. The semantics of the empty action and active choice are straightforward. Sequential composition uses relativization: some position i is chosen, such that the left translation holds before i and the right translation afterwards. Note that i is included in both relativization, to uphold the invariant that a trace always starts and ends with a state. The semantics of repetition are the only point where we require second order quantifiers: set I is a set of indices, such that the first and last position are included and for every consecutive pair k, l of elements of I, the translation of the repeated type holds in the relativization between k and l. Passive choice specifies that the first event is a read on a correct future (i.e., resolved by the correct method) and the suffix afterwards follows the communicated choice correctly.

Example 6. The following formalizes the behavior described informally in Ex. 2:

$$?\mathtt{T.test}(\mathsf{true}).\mathsf{S}!\mathtt{Comp.cmp}(\mathtt{data} \doteq \mathtt{i}).\&(\{\mathtt{Comp.cmp}\}, \mathtt{result} < 0)\left\{\begin{array}{l}\mathsf{L}!\mathtt{Log.log}(\mathtt{data} \doteq \mathtt{i}),\\ \mathsf{skip}\end{array}\right\}.\downarrow(\mathtt{result} \geq 0)$$

The `result` variable in the guard of the passive choice is referring to the result of the read value, not the specified method.

We define behavioral types from a program logic perspective[2] by a type system, which is a set of sequent calculus rules that match on behavioral modalities and a obligation scheme, that maps every method to a proof obligation

Definition 15 (Behavioral Types). *A* behavioral type $\mathbb{T}$ *is a behavioral specification* $(\tau_{\mathbb{T}}, \alpha_{\mathbb{T}})$ *extended with* $(\gamma_{\mathbb{T}}, \iota_{\mathbb{T}})$.

The obligation scheme $\iota_{\mathbb{T}}$ *maps method names* $\mathtt{m}$ *to* proof obligations*, sequents of the form* $\varphi_{\mathtt{m}} \Rightarrow [\mathtt{s_m} \Vdash^{\alpha_{\mathbb{T}}} \tau_{\mathtt{m}}]$*, which have to be proven.* $\mathtt{s_m}$ *is the method*

[2] Behavioral types are sometimes (informally) distinguished from data types by having a subject reduction theorem where the typing relation is preserved, but not the type itself [10]. In BPL this would correspond to the property that one of the rules has a premise where the type in the behavioral modality is different than in the conclusion.

Fig. 7. Semantics for $\mathbb{T}_{\mathsf{met}}$. Unbound variables are implicitly existentially quantified.

$$
\begin{aligned}
\alpha'_{\mathsf{met}}(\mathtt{X!m}(\varphi)) &= \forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \land [i] \not\doteq \mathsf{noEv} \to [i] \doteq \mathsf{invEv}(x, \mathtt{X}, f, \mathtt{m}, \overrightarrow{\mathtt{e}}) \land [i-1] \vdash \varphi(\overrightarrow{\mathtt{e}}) \\
&\quad \land \exists i \in \mathbf{I}.\ [i] \not\doteq \mathsf{noEv} \land \mathsf{isEvent}(i) \\
\alpha'_{\mathsf{met}}(\downarrow(\varphi)) &= \forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \land [i] \not\doteq \mathsf{noEv} \to [i] \doteq \mathsf{futEv}(x, f, m, \mathtt{e}) \land [i-1] \vdash \varphi[\mathtt{result} \setminus \mathtt{e}] \\
&\quad \land \exists i \in \mathbf{I}.\ [i] \not\doteq \mathsf{noEv} \land \mathsf{isEvent}(i) \\
&\quad \text{where } \varphi(\overrightarrow{\mathtt{e}}) \text{ replaces its free variables by } \overrightarrow{\mathtt{e}}.\ \varphi[\mathtt{result} \setminus \mathtt{e}] \text{ replaces } \mathtt{result} \text{ by } \mathtt{e}. \\
\alpha'_{\mathsf{met}}(\mathsf{skip}) &= \forall l \in \mathbf{I}.\ [l] \doteq \mathsf{noEv} \lor [l] \vdash \mathtt{true} \qquad \alpha'_{\mathsf{met}}(\oplus\{\mathsf{L}_i\}_{i \in I}) = \textstyle\bigvee_{i \in I} \alpha'_{\mathsf{met}}(\mathsf{L}_i) \\
\alpha'_{\mathsf{met}}(\mathsf{L}_1.\mathsf{L}_2) &= \exists i \in \mathbf{I}.\ \alpha'_{\mathsf{met}}(\mathsf{L}_1)[n \in \mathbf{I} \setminus n \leq i] \land \alpha'_{\mathsf{met}}(\mathsf{L}_2)[n \in \mathbf{I} \setminus n \geq i] \\
\alpha'_{\mathsf{met}}(\mathsf{L}^*) &= \exists I \subseteq \mathbf{I}.\ \exists a, b \in I.\ a < b \land \\
&\qquad \forall k \in \mathbf{I}.\ ((k < a \land \mathsf{isEvent}(i) \to [i] \not\doteq \mathsf{noEv}) \lor (a \leq k \land k \leq b)) \land \\
&\qquad \forall i_1, i_2 \in I.\ \left((\forall l \in I.\ l \leq i_1 \land i_2 \leq l) \to \alpha'_{\mathsf{met}}(\mathsf{L})[n \in \mathbf{I} \setminus i_1 \leq n \land n \leq i_2)]\right)
\end{aligned}
$$

$$
\begin{aligned}
&\alpha'_{\mathsf{met}}(\&(\{\mathtt{m}_l\}_{l \in I}, \varphi)\{\mathsf{L}_1, \mathsf{L}_2\}) = \exists i, j, k \in \mathbf{I}.\ i < j \land j < k \land \\
&\quad (\forall l \in \mathbf{I}.\ l \doteq j \lor l \geq k \lor (l \leq i \land ([l] \doteq \mathsf{noEv} \lor [l] \vdash \mathtt{true})) \land [j] \doteq \mathsf{futREv}(x, m, f, \mathtt{e}, n) \land \\
&\quad \textstyle\bigvee_{l \in I} m \doteq \mathtt{m}_l \land ([k] \vdash \varphi \to \alpha'_{\mathsf{met}}(\mathsf{L}_1)[n \in \mathbf{I} \setminus n \geq k]) \land ([k] \not\vdash \varphi \to \alpha'_{\mathsf{met}}(\mathsf{L}_2)[n \in \mathbf{I} \setminus n \geq k])
\end{aligned}
$$

body of $\mathtt{m}$. *The* type system $\gamma_{\mathbb{T}}$ *is a set of rewrite rules for* $\tau_{\mathbb{T}}$ *and sequent calculus rules with conclusions matching the sequent* $\Gamma \Rightarrow \{U\}[\mathtt{s} \Vdash^{\alpha_{\mathbb{T}}} \tau_{\mathbb{T}}], \Delta$.

We demand that obligation schemes are consistent, i.e., proof obligations do not contradict each other. This would be the case if, for example a method is called and its precondition φ is checked caller-side, then φ must truly be used as a precondition by the proof obligation for the called method.

Definition 16. *Let* $\mathbf{L}_{\mathtt{m}} = ?\mathtt{m}(\varphi_{\mathtt{m}}).\mathsf{L}_{\mathtt{m}}$ *be the local protocols in* Prgm. *We require that all* $\mathbf{L}_{\mathtt{m}}$ *are consistent: If* $\mathtt{m}$ *is called in the method type of any other method* $\mathtt{m}'$, *then the call condition implies* $\varphi_{\mathtt{m}}$. *Furthermore,* $\varphi_{\mathtt{X.run}} = \mathtt{true}$.

The extension of the behavioral specification $\mathbb{T}_{\mathsf{met}}$ *of method types to a behavioral type is given by the calculus in Fig. 8 and* $\iota_{\mathsf{met}}(\mathtt{m}) = \varphi_{\mathtt{m}} \land \Phi \Rightarrow [\mathtt{s}_{\mathtt{m}} \Vdash^{\alpha_{\mathsf{met}}} \mathsf{L}_{\mathtt{m}}]$. *Formula* $\Phi = \bigwedge_{\mathtt{X}} \mathtt{X} \doteq \mathit{select}(\mathtt{heap}, \mathtt{f}_{\mathtt{X}})$ *encodes the assignment of roles to fields.*

The call condition may contain fields of the other objects, but this is not an issue when checking consistency, as the precondition only contains fields of the own object and the fields are simply uninterpreted function symbols. The method in Fig. 2 can be typed with the type in Ex. 6.

Rule **(met–V)** translates a variable-assignment into an update and **(met–F)** is analogous for fields. Rule **(met–get)** has three premises: one premise checks via $\mathbb{T}_{\mathsf{p2}}$ that the correct methods are synchronized with. The two others use a fresh constant v for the read value and assign it to the target variable. The two premises differ in the branch that is checked afterwards, depending on whether or not the choice condition holds. Rule **(met–while)** is a standard loop invariant rule. An invariant I holds before the first iteration and is preserved by the loop to remove all other information afterwards. The loop body is checked against the repeated type and the continuation against the continuation of the type. Method types have no special action for the end of a statement, so $\mathbb{T}_{\mathsf{pst}}$ is used

$$\textbf{(met-V)}\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \mathtt{e}\}[\mathtt{s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v = e;\ s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}], \Delta} \qquad \textbf{(met-F)}\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{heap} := \mathsf{store}(\mathtt{heap}, \mathtt{f}, \mathtt{e})\}[\mathtt{s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}], \Delta}{\Gamma \Rightarrow \{U\}[\textbf{this}\mathtt{.f = e;\ s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}], \Delta}$$

$$\textbf{(met-get)}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\{\mathtt{v} := v\}(\neg\varphi(v) \to [\mathtt{s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}_2]), \Delta \qquad \Rightarrow [\mathtt{v = e.}\textbf{get}_i\mathtt{;s}] \overset{\mathsf{p2}}{\Vdash} \{\overrightarrow{\mathtt{m}}\}\\ \Gamma \Rightarrow \{U\}\{\mathtt{v} := v\}(\varphi(v) \to [\mathtt{s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}_1]), \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathtt{v = e.}\textbf{get}_i\mathtt{;s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \&(\overrightarrow{\mathtt{m}}, \varphi)\{\mathsf{L}_1, \mathsf{L}_2\}], \Delta}\ v \text{ fresh}$$

$$\textbf{(met-while)}\ \frac{\Gamma \Rightarrow \{U\}I, \Delta \qquad I, \mathtt{e} \Rightarrow [\mathtt{s} \overset{\mathsf{pst}}{\Vdash} I] \qquad I, \mathtt{e} \Rightarrow [\mathtt{s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}] \qquad I, \neg\mathtt{e} \Rightarrow [\mathtt{s'} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}'], \Delta}{\Gamma \Rightarrow \{U\}[\textbf{while}\ \mathtt{e}\ \textbf{do}\ \mathtt{s}\ \textbf{od}\ \mathtt{s'} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}^*.\mathsf{L}'], \Delta}$$

$$\textbf{(met-if)}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}(\mathtt{e} \to [\mathtt{s;s''} \overset{\alpha_{\mathsf{met}}}{\Vdash} \oplus\{\mathsf{L}_i\}_{i\in I_1}]), \Delta\\ \Gamma \Rightarrow \{U\}(\neg\mathtt{e} \to [\mathtt{s';s''} \overset{\alpha_{\mathsf{met}}}{\Vdash} \oplus\{\mathsf{L}_i\}_{i\in I_2}]), \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{if}\ \mathtt{e}\ \textbf{then}\ \mathtt{s}\ \textbf{else}\ \mathtt{s'}\ \textbf{fi}\ \mathtt{s''} \overset{\alpha_{\mathsf{met}}}{\Vdash} \oplus\{\mathsf{L}_i\}_{i\in I}], \Delta}\ I_1 \cup I_2 \subseteq I$$

$$\textbf{(met-call)}\ \frac{\Gamma \Rightarrow \{U\}\left(\varphi(\mathtt{e}) \wedge \mathit{select}(\mathtt{heap}, \mathtt{f}) \doteq \mathtt{X}\right), \Delta \qquad \Gamma \Rightarrow \{U\}\{\mathtt{v} := f\}[\mathtt{s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{L}], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v = f!m(e);\ s} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathtt{X!m}(\varphi).\mathsf{L}], \Delta}\ f \text{ fresh}$$

$$\textbf{(met-return)}\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{result} := \mathtt{e}\}\varphi, \Delta}{\Gamma \Rightarrow \{U\}[\textbf{return}\ \mathtt{e} \overset{\alpha_{\mathsf{met}}}{\Vdash} \downarrow(\varphi)], \Delta} \qquad \textbf{(met-skip)}\ \frac{}{\Gamma \Rightarrow \{U\}[\textbf{skip} \overset{\alpha_{\mathsf{met}}}{\Vdash} \mathsf{skip}], \Delta}$$

$$\mathsf{L} \leftrightsquigarrow \oplus\{\mathsf{L}\} \qquad \mathsf{skip}.\mathsf{L} \leftrightsquigarrow \mathsf{L} \qquad \mathsf{L}.\mathsf{skip} \leftrightsquigarrow \mathsf{L}$$

Fig. 8. Rules for $\mathbb{T}_{\mathsf{met}}$. We remind that the sets I_1, I_2 are defined as non-empty. For simplicity, we assume that every branch and every loop body implicitly ends in **skip**.

for checking that the loop preserves its invariant. Rule **(met–if)** splits the set of possible choices into two and checks each branch against one of these sets. These sets may overlap and do not need to cover all original choices, but may not be empty. Rule **(met–call)** checks the annotated condition of the called method and the correct target explicitly and that the correct method is called by matching call type and call statement. We remind that references are not reassigned, so call targets can be verified locally. The other rules are straightforward.

Contracts and Invariants. Method types generalize method contracts and object invariants as follows. An object invariant is encoded by adding it to the formula in the receiving and terminating actions of all method in an object – except the constructor `run`, where it is only added to the terminating action. A method contract (consisting of a precondition on the parameters and a postcondition) is encoded analogously by adding the precondition to the receiving and the postcondition to the terminating actions. However, one additional step is required: Method types are generated by projection of global types [23], so to use them for object invariants or method contracts requires to infer a method type first. This is done by mapping every call to a call action, every branching

```
1 class T(Comp S, Log L){
2  Int nr = 0;
3  Int test(Int i){
4   Fut<Int> f = S!cmp(i);
5   this.nr = this.nr + 1;
6   Int r = f.get_0;
7   if(r < 0 && i > 0){
8    r = -r; f = L!log(i);
9   }
10  return r;
11 }
12 }
```

Precondition: $\mathtt{i} \geq 0$
Postcondition: $\mathtt{result} \geq 0$
Invariant: $\mathbf{this}.\mathtt{nr} \geq 0$

$$\mathsf{L}_1 = \; ?\mathtt{T.test}(\mathtt{i} \geq 0 \wedge \mathbf{this}.\mathtt{nr} \geq 0).\mathtt{S!Comp.cmp}(\varphi_{\mathtt{cmp}}) \; . \; \&(\mathsf{M}, \mathsf{true}) \left\{ \begin{array}{l} \oplus \left\{ \begin{array}{l} \mathtt{L!Log.log}(\varphi_{\mathtt{log}}), \\ \mathsf{skip} \end{array} \right\}, \\ \mathsf{skip} \end{array} \right\} \; . \; \downarrow(\mathtt{result} \geq 0 \wedge \mathbf{this}.\mathtt{nr} \geq 0)$$

$$\mathsf{L}_2 = \; ?\mathtt{T.test}(\mathtt{i} \geq 0 \wedge \mathbf{this}.\mathtt{nr} \geq 0).\mathtt{S!Comp.cmp}(\mathtt{data} \doteq \mathtt{i} \wedge \varphi_{\mathtt{cmp}}) \; . \; \&(\{\mathtt{Comp.cmp}\}, \mathtt{result} < 0) \left\{ \begin{array}{l} \mathtt{L!Log.log}(\mathtt{data} \doteq \mathtt{i} \wedge \varphi_{\mathtt{log}}), \\ \mathsf{skip} \end{array} \right\} \; . \; \downarrow(\mathtt{result} \geq 0) \wedge \mathbf{this}.\mathtt{nr} \geq 0$$

Fig. 9. An example method and two method types for method contracts and invariants.

to an active choice, every loop to a repetition, termination to a terminating action and using true at every position where a formula is required, before adding precondition, postcondition or object invariant. The most complex construct is synchronization. Each such read is mapped to a passive choice with all methods as the method set and true as the choice condition. The following code is added in the first branch. The second branch is skip. Invariants require fields in the precondition and a fitting notion of consistency, which was developed in [23].

Example 7. Consider the code in Fig. 9, a variation of our running example. It tracks the number of calls to `T.test` and inverts the result if the input is positive. It adheres to the contract with precondition $\mathtt{i} \geq 0$ and postcondition $\mathtt{result} \geq 0$ and the invariant $\mathbf{this}.\mathtt{nr} \geq 0$. The algorithm above derives the following type:

$$?\mathtt{T.test}(\mathsf{true}) \; . \; \mathtt{S!Comp.cmp}(\mathsf{true}) \; . \; \&(\mathsf{M}, \mathsf{true}) \left\{ \begin{array}{l} \oplus \left\{ \begin{array}{l} \mathtt{L!Log.log}(\mathsf{true}), \\ \mathsf{skip} \end{array} \right\}, \\ \mathsf{skip} \end{array} \right\} \; . \; \downarrow(\mathsf{true})$$

Let $\varphi_{\mathtt{cmp}}$ and $\varphi_{\mathtt{log}}$ be the preconditions of the called methods. The final specification, after adding the contract and the invariant, is shown on the right in Fig. 9 as L_1. The inferred type is not the one we gave in Ex. 6: For one, it differs in its shape (two choice operators). For another, it neither keeps track of the passed data, nor specifies the relation between the return value of `Comp.cmp` and the taken branch. These properties are typical for protocol specifications and require a global view, contrary to the local view of method contracts and object invariants. However, one can add the pre- and postcondition and the object invariant also to the type given in Ex. 6 and combine local and global specification. The result is shown as L_2 in in Fig. 9. L_2 expresses that the method follows the protocol and adheres to contract and object invariant.

Theorem 1. $\mathbb{T}_{\mathsf{met}}$ *is sound for every program.*

The proof is standard [22]. Consistency of the obligation scheme is required to establish that all selected traces are models for the type of their method. The first two elements are not described by the method type and, thus, removed.

Corollary 1. *If (1) for every method* $\mathtt{m}$ *with type* $?\mathtt{m}(\varphi).\mathsf{L}_\mathtt{m}$ *the formula* $\iota_{\mathsf{met}}(\mathtt{m})$ *is valid and (2) the obligation scheme is consistent, then for every selected trace* θ *of any method* $\mathtt{m}$*, the trace after the invocation reaction event follows its type:*

$$\theta[2..|\theta|], I, \emptyset \models \alpha_{\mathsf{met}}(\mathsf{L}_\mathtt{m})$$

5 Conclusion and Related Work

This work presents BPL, a program logic for object-oriented distributed programs that enables deductive reasoning about the results of static analyses and integrates concepts from behavioral types by pattern-matching statement and specification. The *method type* behavioral type generalizes method contracts, session types and object invariants. In the following, we discuss related work.

Dynamic Logics. Beckert and Bruns [3] use LTL formulas in dynamic logic modalities in their Dynamic Trace Logic (DTL) for Java. Given an LTL formula φ, the DTL-formula $[\mathtt{s}]\varphi$ expresses that φ describes all traces of $\mathtt{s}$. DTL uses a restricted form of pattern matching: its three loop invariant rules depend on the outermost operator of φ and other rules may consume a "next" operator. DTL does not use events and specifies patterns of state changes, not of interactions.

The Abstract Behavior Specification Dynamic Logic (ABSDL) of Din and Owe [12] is for the ABS language [21]. In ABSDL, a formula $[\mathtt{s}]\varphi$, where φ is a first-order formula over the program state, has the standard meaning that φ holds after $\mathtt{s}$ is executed. ABSDL uses a special program variable to keep track of the visible events. Its rules are tightly coupled with object-invariant reasoning. This makes it impossible to specify the state at arbitrary interactions.

Bubel et al. [7] define dynamic logic with coinductive traces (DLCT). In DLCT, a formula $[\mathtt{s}]\varphi$, where φ is a trace modality formula, containing symbolic trace formulas, has the meaning that every trace of $\mathtt{s}$ is a model for φ. Contrary to ABSDL, DLCT keeps track of the whole trace, not just the events. DLCT is not able to specify the property that between two states, some form of event does *not* occur, as symbolic trace formulas are not closed under negation.

Behavioral Types. A number of behavioral types deals with assertions [4, 5] or Actors [16, 18, 26]. Stateful Behavioral Types for Active Objects (STAO) [23] uses both and defines the judgment $\varphi, \mathtt{s}' \vdash \mathtt{s} : \tau$, that expresses that all traces of $\mathtt{s}$ are models for the translation of τ. φ and $\mathtt{s}'$ keep track of the chosen path so far. STAO is not able to reason about multiple judgments, but relies on external analyses for precision. Reasoning about these results happens on a meta-level.

Finally, Propositions-as-Types theorems (PaT) have been established [8, 27] between session types for the π-calculus and intuitionistic linear logic. They are specific to this setting and do not characterize general behavioral types. To our best knowledge, Def. 15 is the first formal characterization of behavioral types.

Future Work. An implementation of BPL for full ABS is ongoing and as future work, we plan to investigate further types and concurrency models, in particular systems with shared memory and effect type systems.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., and Ulbrich, M., Eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*, vol. 10001 of *Lecture Notes in Computer Science*. Springer, 2016.
2. Beckert, B. A dynamic logic for the formal verification ofjava card programs. In *Java on Smart Cards:Programming and Security* (Berlin, Heidelberg, 2001), I. Attali and T. Jensen, Eds., Springer Berlin Heidelberg, pp. 6–24.
3. Beckert, B., and Bruns, D. Dynamic logic with trace semantics. In *Automated Deduction - CADE 2013. Proceedings* (2013), M. P. Bonacina, Ed., vol. 7898 of *Lecture Notes in Computer Science*, Springer, pp. 315–329.
4. Berger, M., Honda, K., and Yoshida, N. Completeness and logical full abstraction in modal logics for typed mobile processes. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Proceedings, Part II* (2008), L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, Eds., vol. 5126 of *Lecture Notes in Computer Science*, Springer, pp. 99–111.
5. Bocchi, L., Lange, J., and Tuosto, E. Three algorithms and a methodology for amending contracts for choreographies. *Sci. Ann. Comp. Sci. 22*, 1 (2012), 61–104.
6. Bubel, R., Damiani, F., Hähnle, R., Johnsen, E. B., Owe, O., Schaefer, I., and Yu, I. C. Proof repositories for compositional verification of evolving software systems - managing change when proving software correct. *Trans. Found. Mastering Chang. 1* (2016), 130–156.
7. Bubel, R., Din, C. C., Hähnle, R., and Nakata, K. A dynamic logic with traces and coinduction. In *Automated Reasoning with Analytic Tableaux and Related Methods TABLEAUX 2015. Proceedings* (2015), H. de Nivelle, Ed., vol. 9323 of *Lecture Notes in Computer Science*, Springer, pp. 307–322.
8. Caires, L., and Pfenning, F. Session types as intuitionistic linear propositions. In *Concurrency Theory, CONCUR 2010. Proceedings* (2010), P. Gastin and F. Laroussinie, Eds., vol. 6269 of *Lecture Notes in Computer Science*, Springer, pp. 222–236.
9. de Boer, F. S., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C. C., Johnsen, E. B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., and Yang, A. M. A survey of active object languages. *ACM Comput. Surv. 50*, 5 (2017), 76:1–76:39.
10. Dezani-Ciancaglini, M. personal communication, 19.10.2018.
11. Din, C. C., Hähnle, R., Johnsen, E. B., Pun, K. I., and Tapia Tarifa, S. L. Locally abstract, globally concrete semantics of concurrent programming languages. In *Automated Reasoning with Analytic Tableaux and Related Methods - 26th International Conference, TABLEAUX 2017, Proceedings* (2017), R. A. Schmidt and C. Nalon, Eds., vol. 10501 of *Lecture Notes in Computer Science*, Springer, pp. 22–43.
12. Din, C. C., and Owe, O. A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebr. Meth. Program. 83*, 5-6 (2014), 360–383.
13. Din, C. C., Tapia Tarifa, S. L., Hähnle, R., and Johnsen, E. B. History-based specification and verification of scalable concurrent and distributed systems. In *International Conference on Formal Engineering Methods, ICFEM 2015. Proceedings* (2015), M. J. Butler, S. Conchon, and F. Zaïdi, Eds., vol. 9407 of *Lecture Notes in Computer Science*, Springer, pp. 217–233.

14. Flores-Montoya, A. E., Albert, E., and Genaim, S. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems* (Berlin, Heidelberg, 2013), D. Beyer and M. Boreale, Eds., Springer Berlin Heidelberg, pp. 273–288.
15. García-Matos, M., and Väänänen, J. Abstract model theory as a framework for universal logic. In *Logica Universalis* (Basel, 2005), J.-Y. Beziau, Ed., Birkhäuser Basel, pp. 19–33.
16. Giachino, E., Johnsen, E. B., Laneve, C., and Pun, K. I. Time complexity of concurrent programs - - A technique based on behavioural types -. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Revised Selected Papers* (2015), C. Braga and P. C. Ölveczky, Eds., vol. 9539 of *Lecture Notes in Computer Science*, Springer, pp. 199–216.
17. Henkin, L. Relativization with respect to formulas and its use in proofs of independence. *Compositio Mathematica 20* (1968), 88–106.
18. Henrio, L., Laneve, C., and Mastandrea, V. Analysis of synchronisations in stateful active objects. In *Integrated Formal Methods - 13th International Conference, iFM 2017. Proceedings* (2017), N. Polikarpova and S. Schneider, Eds., vol. 10510 of *Lecture Notes in Computer Science*, Springer, pp. 195–210.
19. Honda, K., Yoshida, N., and Carbone, M. Multiparty asynchronous session types. vol. 63, pp. 9:1–9:67.
20. Hüttel, H., Lanese, I., Vasconcelos, V. T., Caires, L., Carbone, M., Deniélou, P.-M., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H. T., and Zavattaro, G. Foundations of session types and behavioural contracts. *ACM Comput. Surv. 49*, 1 (Apr. 2016), 3:1–3:36.
21. Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010. Revised Papers* (2010), B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds., vol. 6957 of *Lecture Notes in Computer Science*, Springer, pp. 142–164.
22. Kamburjan, E. Behavioral program logic and LAGC semantics without continuations (technical report). *CoRR abs/1904.13338* (2019).
23. Kamburjan, E., and Chen, T. Stateful behavioral types for active objects. In *Integrated Formal Methods - 14th International Conference, iFM 2018. Proceedings* (2018), C. A. Furia and K. Winter, Eds., vol. 11023 of *Lecture Notes in Computer Science*, Springer, pp. 214–235.
24. Kamburjan, E., Din, C. C., and Chen, T. Session-based compositional analysis for actor-based languages using futures. In *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016. Proceedings* (2016), K. Ogata, M. Lawford, and S. Liu, Eds., vol. 10009 of *Lecture Notes in Computer Science*, pp. 296–312.
25. Kamburjan, E., and Hähnle, R. Deductive verification of railway operations. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Second International Conference, RSSRail 2017. Proceedings* (2017), A. Fantechi, T. Lecomte, and A. B. Romanovsky, Eds., vol. 10598 of *Lecture Notes in Computer Science*, Springer, pp. 131–147.
26. Neykova, R., and Yoshida, N. Multiparty session actors. *Logical Methods in Computer Science 13*, 1 (2017).
27. Wadler, P. Propositions as types. *Commun. ACM 58*, 12 (2015), 75–84.