Stateful Behavioral Types for ABS

Eduard Kamburjan and Tzu-Chun Chen

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

Abstract. It is notoriously hard to correctly implement a multiparty protocol which involves asynchronous/concurrent interactions and constraints on states of multiple participants. To assist developers in implementing such protocols, we propose a novel specification language to specify interactions within multiple object-oriented actors and the side-effects on heap memory of those actors. A behavioral-type-based analysis is presented for type checking. Our specification language formalizes a protocol as a *global* type, which describes the procedure of asynchronous method calls, the usage of *futures*, and the heap side-effects with a first-order logic. To characterize runs of instances of types, we give a model-theoretic semantics for types and translate them into logical constraints over traces. We prove protocol adherence: If a program is well-typed w.r.t. a protocol, then every trace of the program adheres to the protocol, i.e., every trace is a model for the formula of the protocol's type.

1 Introduction

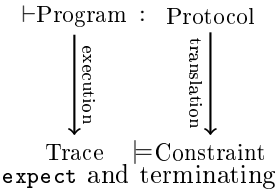
The combination of actors [25] with futures [4] in object-oriented languages (e.g., Scala [34] and ABS [28]), sometimes called *Active Objects* [12], is an active research area for system models and is frequently used in practice [37]. Processes of Active Objects communicate internally within an object via the object's heap memory. External communication works via asynchronous method calls with futures: constructs for synchronizing executions invoked by those calls. Encapsulated heap memory and explicit synchronization points make it easy to locally reason about Active Objects, but hard to specify and verify *global* protocols.

The main obstacle is to bridge the gap between local perspectives of single objects and global perspectives of the whole system. As Din and Owe [15] pointed out, it is non-trivial to precisely specify the communication within an object's heap memory from a global perspective [16]. Multiparty session types (short as MPST) [27], one important member of behavioral types [3, 19], are established theories for typing *globally stateless* concurrent interactions (i.e., method calls) among multiple participants (i.e., objects) to ensure communication safety. Current works in MPST [6, 38] have attempted to specify state in communication by using global values and assuming channels as the only communication concept. Global values are not sufficient to specify the non-trivial interplay of processes when taking the heap memory inside of an object into account. Furthermore, channels are not able to fully represent the usage of futures, because futures, unlike channels, could expose some inner state of their object. Namely, it exposes

that the computing process has terminated and the object was inactive before and after.

We integrate the stateful analysis and specification of traces of Din et al. [15] into MPST, where local verification of the endpoints compositionally guarantees the global specification of the whole system. Functional properties are specified as a part of the communication pattern. We ensure that from the perspective of each actor, its trace is not distinguishable from the global specification and that the whole system is deadlock free.

We specify passed data and modifications of heap memory with first-order logic (FOL) formulas and transform behavioral types into logical constraints on traces. Moreover, from the model-theoretic perspective, we define *protocol adherence* as the property that every generated trace of a well-typed($\vdash$) program is a model($\models$) for the translation of the type. The running example below illustrates the challenges for protocols in Active Objects.

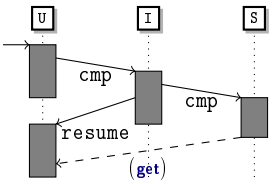


Consider a GUI `U`, a computation server `S`, and an interface server `I` such that `U`, without knowing `S`, wants to compute some data by sending it to `I` via a method call. After executing this call, `U` prepares for the next action by setting field `intern` to value `expect` and terminating its process to stay responsive. `I` delegates `U`'s task to `S` and remains responsive to other requests without waiting for `S`'s computation by invoking another method on `U` with future `x`, which will carry the computation result, back to `U`. The code and figure below implement this scenario:

```
1  object U{
2    TState intern = init;
3    Int resume(Fut⟨Int⟩ x){
4      if( this.intern!=expect ) return −1;
5      Int r = x.get; return r; }
6    Unit start(Int j){
7      Fut⟨Unit⟩ f = I!cmp(j);
8      this.intern = expect; }}
```

```
9  object I{
10   Unit cmp(Int dat){
11     Fut⟨Int⟩ f = S!cmp(dat);
12     Fut⟨Int⟩ f' = U!resume(f);}}
13
14 object S{ Int cmp(Int i){ ... }}
15
16 main { U!start(20); }
```

In the code, `!` denotes a non-blocking call, `I!cmp` calls method `cmp` of `I`, `U!start` calls `U.start`, `U!resume` calls method `resume` for continuation, and `S!cmp` starts the actual computation at `S`. The challenge for formal specifications is to express that (1) `I` is transparent to `U` and `S` such that `I` must pass the same data to `S` that it received from `U`, and `I` does not read the return value from `S`; and (2) `U` changes its heap to `expect` and reads the correct future.



Contributions. We propose (1) a specification language for actors' behaviors, that integrates FOL to specify heap memory, (2) model-theoretic semantics for protocol adherence, and (3) a static type system integrating a FOL validity calculus, which guarantees protocol adherence and deadlock freedom.

Roadmap. Section 2 provides an overview of our approach. Section 3 introduces a core language for Active Objects, Async, and its dynamic logic, Section 4 gives the types and operations on them and Section 5 gives the type system. Section 6 extends the concept to repetition. Section 8 concludes and discusses related work.

2 Scope, Challenges and an Overview of the Workflow

We aim to specify and verify *session-based* systems. A session-based system is a system which has a fixed, finite set of participating objects. Each object has an assigned *role* within the protocol of a session. Our analysis is fully static and is aimed at *system validation*: Ensuring that an existing system follows a certain specification.

We consider object-oriented actors, which use method calls, futures, and heap memory for communication. Every method call is asynchronous and starts a new process at the callee object. At each such call, the active *caller* obtains a *fresh* future identity, on which one may synchronize on the termination of the started process. An object may only switch its active process to another process if the currently active process terminates. The usage of futures provides programmers with the control of *when* synchronize – however, combining futures with object-oriented actors leads to the following complications:

Protocols with State In an object-oriented setting, one must take the heap memory into account when reasoning about concurrent computations. For one, the heap memory influences the behavior of objects. For another, changes of the heap memory (among coordinated actors) are not only a by-effect of communication but often the *aim* of a protocol. Actors enforce strong encapsulation and restrict communication between object to asynchronous method calls and future reads – coordinated memory changes must be part of the specification.

Unexposed State In the Active Object concurrency model, each process has exactly one future. Thus reading from a future is synchronizing with an unknown process *and depends on the state of the process's object.* To avoid deadlocks, futures cannot be analyzed in isolation — reading from a future must take the unexposed state of the object into account.

Mixed Communication Paradigms Processes inside an object communicate through the heap memory. This kind of communication is hard to describe with data types, as it requires fine-grained specification of computation and has no explicit caller or callee. Thus, it is difficult to isolate the parts of the program which realize the communication protocol. Furthermore, method calls are asynchronous, while future reads are synchronous.

Two-Fold Endpoints In the Active Object model, the callee endpoints of methods calls are *objects*, but the caller endpoints and the endpoints for future synchronization are *processes.* The interplay of multiple objects, which contain multiple processes, must be captured in the analysis by a two-fold notion of endpoints such that objects and processes are both endpoints.

In the following, we use the example from Section 1 to show how our approach works and addresses these issues.

Example 1: Specifying global types. Our specification language for side-effects is a FOL for specifying *local* memory instead of global values since (1) global values are not natural in an Active Object setting, and (2) a logic over memory

locations (variables and fields) allows us to use a well-established theory of first order dynamic logic [22] to capture the semantics of methods. We formalize the scenario in Section 1 by the following global type in our specification language:

$$\mathbf{G} = \mathbf{main} \to \mathsf{U} : \texttt{start}\langle \texttt{U.state} \doteq \texttt{expect} \rangle \;.\; \mathsf{U} \to \mathsf{I} : \texttt{cmp}\langle \top, \top \rangle \;.$$
$$\mathsf{I} \xrightarrow{\texttt{f}} \mathsf{S} : \texttt{cmp}\langle \texttt{i} \doteq \texttt{dat}, \mathbf{result} > 0 \rangle \;.\; \mathsf{I} \to \mathsf{U} : \texttt{resume}\langle \texttt{x} \doteq \texttt{f}, \top \rangle \;.\; \mathsf{U} \uparrow \texttt{x} \;.\; \mathsf{End}$$

We formally define the above syntax in Section 4 and only give the intuition here: $\top$ denotes true. $\mathsf{U} \to \mathsf{I} : \texttt{cmp}$ denotes a message `cmp` from U to I, i.e., the call to a method `cmp`. Formula $\mathsf{U}.\texttt{state} \doteq \texttt{expect}$ is the postcondition for the process *started* by this call at the callee object. If two formulas are provided, the first is the precondition describing the state of the caller and the second is the postcondition describing the state of the callee and the return value, which is denoted by keyword **result**. The annotation `f` denotes the memory location where the future of the denoted call is stored. Formula $\texttt{i} \doteq \texttt{dat}$ states that `dat`, the parameter of $\mathsf{S}.\texttt{cmp}$, carries the same value as received by $\mathsf{I}.\texttt{cmp}$ on parameter `i`, while formula $\texttt{x} \doteq \texttt{f}$ requires that parameter `x` of the call at method resume carries the future of the previous call to cmp. Finally, $\mathsf{U} \uparrow \texttt{x}$ describes a read of `U` on the future stored in the location `x`. Note that we specify locations in formulas and avoid a situation where an endpoint must guarantee an obligation containing values that it cannot access. Other approaches (e.g., Bocchi et al. [6]) allow this situation and thus require additional analyses of history-sensitivity and temporal-satisfiability.

For the analysis, we adopt an approach similar to MPST: We project a global type on endpoints defined inside it, to automatically derive local specifications for all objects and methods. Additionally, formulas, which are used to specify conditions on the heap memory, are projected on the logical substructure of the callee, because the callee cannot access the caller's fields.

Two-phase Analysis. The analysis requires that the protocol is encoded as a global type, which defines the order of method calls and future reads between objects, annotated with FO specifications of heap memory and passed data. Our analysis has two phases. In Phase 1, the global type is used to generate local types for all endpoints. In Phase 2, the endpoints are type checked against their local types and a causality graph is generated for checking for deadlocks. The workflow of Phase 1 is based on MPST's approach, but is adjusted to the Active Object concurrency model:

Phase 1. The workflow of Phase 1 is shown in Fig. 1.

- *Step 1*: The global type is projected onto the participating objects and generates *object* types. Such a type specifies the obligation of an object for running methods in a certain order, and for guaranteeing the FOL specifications of the object's state. During projection, the FO-specifications are projected onto the substructure of the object in question.
- *Step 2*: FO-specifications are propagated within an object type: as the order of method executions is specified by the specification, the postcondition of a method can be assumed as a *pre*condition for the next method.

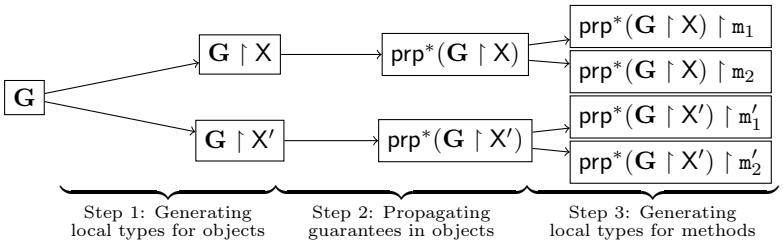


Fig. 1. Workflow for Phase 1: $\mathbf{G}$ is a global type and $\upharpoonright$ denotes projection on object X resp. method $\mathtt{m}$. Function prp^* is the function propagating guarantees.

– *Step 3*: An object type is projected on its methods, producing *method* types.

A global type encodes the following obligations (short as Obl.) for the implementation: (*Obl. a*) for each object, the observable events (calls and reads) are ordered as specified in the global type, (*Obl. b*) for each method, the observable events are ordered as specified in the local type derived from the global type and (*Obl. c*) the whole system does not deadlock, and adhere to the FO-specifications.

In the following, we demonstrate the workflow of Phase 1 for the global type in Example 2. We do not formally introduce the syntax at this point.

Step 1: Object Types. Projecting $\mathbf{G}$ from Example 2 on object `U` results in
$?\mathtt{start}\langle\top\rangle.\mathsf{I}!\mathtt{cmp}\langle\top\rangle.\mathsf{Put}\ \mathtt{state} \doteq \mathtt{expect}.?\mathtt{resume}\langle\exists f.\ \mathtt{x} \doteq f\rangle.\mathsf{Read}\ \mathtt{x}.\mathsf{Put}\ \mathbf{result} > 0$

Type $?\mathtt{start}\langle\top\rangle$ denotes a starting point for runtime execution. Type $\mathsf{I}!\mathtt{cmp}\langle\top\rangle$ denotes an invocation of method `cmp`. Type $\mathsf{Put}\ \varphi$ specifies the termination of the currently active process in a state where φ holds. Position and postcondition of $\mathsf{Put}\ \mathtt{state} \doteq \mathtt{expect}$ are automatically derived. The position is just *before the next* method start and the postcondition is taken from the call in the global type. The analysis ensures that no method executes in-between. The precondition of `resume` is *weakened*, since field `f` is not visible to `U` and callee `U` cannot use all information from caller `I`. Weakening ensures that all locations in φ are visible to `U`. Type $\mathsf{Read}\ \mathtt{x}$ specifies a synchronization on the future stored in `x`.

Step 2: Propagation. In the next step we propagate the postcondition of the last process to the precondition of the next process. No process is specified as active between `start` and `resume`, so the heap is not modified — thus, the postcondition of `start` still holds when `resume` starts. Adding $\mathtt{state} \doteq \mathtt{expect}$ to the precondition of `resume` strengthens the assumption for the type checking of `resume`. The propagation of conditions results in:

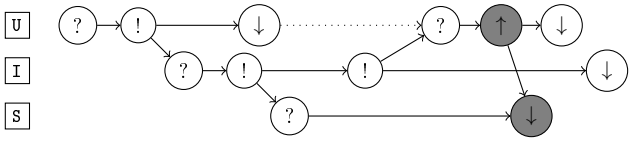
$$\mathsf{prp}^*(\mathbf{G} \upharpoonright \mathtt{U}) = ?\mathtt{start}\langle\top\rangle\ .\ \mathsf{I}!\mathtt{cmp}\langle\top\rangle\ .\ \mathsf{Put}\ \mathtt{state} = \mathtt{expect}\ .\ ?\mathtt{resume}\langle\exists f.\ \mathtt{x} \doteq f \wedge \mathtt{state} \doteq \mathtt{expect}\rangle\ .\ \mathsf{Read}\ \mathtt{x}\ .\ \mathsf{Put}\ \mathbf{result} > 0$$

Step 3: Method Types. We generate a *method type* to specify a method in isolation. Projecting the object type in Step 2 on method `resume` generates:
$\mathsf{prp}^*(\mathbf{G} \upharpoonright \mathtt{U}) \upharpoonright \mathtt{resume} = ?\mathtt{resume}\langle\exists f.\ \mathtt{x} \doteq f \wedge \mathtt{state} \doteq \mathtt{expect}\rangle.\mathsf{Read}\ \mathtt{x}.\mathsf{Put}\ \mathbf{result} > 0$

Method types share the syntax with object types. Projection from object types splits the object type at positions where one method ends and another one starts.

Phase 2. After generating method types, Phase 2 of the analysis checks the implementation of methods against their method types, and checks the formulas for validity. The type checking of method types guarantees the correct local order of events (*Obl. b*). State specifications are checked by integrating a validity calculus [15] into the type system. To guarantee (*Obl. a* and *c*), we require the following analyses:

Causality Graph. We generate a causality graph to ensure deadlock freedom (*Obl. c*): A deadlock free causality graph for Active Objects is cycle-free [17, 24]. A causality graph is also used to ensure that methods of one object are executed in the order specified in the global type that the object obeys to (*Obl. a*).



The nodes are the local types from the projected object types. A solid edge connecting two nodes models that the statement for the first type directly causes the statement for the second type; for example, there are edges from a call to the corresponding receiving type. The graph is partially generated from **G**, and partially generated from the code: The edge connecting the gray nodes is added by a *Points-To* analysis, which maps a location of a future to the methods resolving this future. The termination of a method causes the start of the next (as the object cannot switch the active process otherwise), but does not select the next method itself. A dotted edge models such *indirect* causality: Indirect causality edges are considered when checking cycle-freedom check for deadlock freedom, but not for checking the method order.

Model-theoretic Semantics. One of our contributions is the definition and verification of *protocol adherence* from a model-theoretic point of view: The property that a program follows a specified scenario (the protocol) if every generated trace is a model for the translation of the global type. We thus define protocol adherence through a *logical* characterization of global types and translate types into constraints over *traces*, which are sequences of configurations generated by the program.

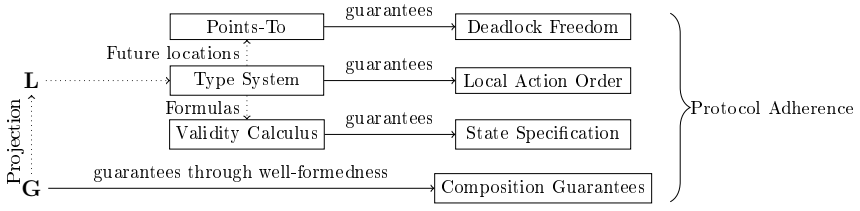


Fig. 2. Workflow for Phase 2 in our analysis.

This *declarative* approach for defining protocol adherence allows us to connect the FO properties embedded in the type to the execution of methods by using a dynamic logic: For a statement `s` the dynamic logic formula $\varphi \Rightarrow [\mathtt{s}]\psi$ expresses that the first-order formula ψ holds after executing `s`, if φ holds in the beginning. From the perspective of the trace logic, FOL describes a single configuration in the trace, while the modality [s] relates the configuration before executing `s` with the configuration after executing `s`. We use modalities during type checking.

3 Async, a Core Actor-Based Language using Futures

We introduce Async, a simple Active Object language based on ABS [28]. Due to space limitations, we only present the basic constructs of Async below. For branching constructs we refer to Section 7; repetition is introduced in Sec. 6. An Async-program consists of a `main` statement and a set of actors, which are objects that have fields and method but do not share state. Inside an object, processes do not interleave and the currently active process must terminate before another one is scheduled. Therefore, single methods can be considered sequential for analysis. We assume standard operations, literals and types for booleans, integers, lists and Object.

Definition 1 (Async-Syntax). *Let* e *denote expressions,* `T` *denote data types,* `x` *denote variable and field names,* X *denote object names, and* **Fut**<`T`> *denote future types.* $\bar{\cdot}$ *represents possibly empty lists and* $[\cdot]$ *represents optional elements.*

$$\mathsf{Prgm} ::= \overline{\mathsf{O}}\ \mathtt{main}\{\mathsf{X}!\mathtt{m}(\bar{\mathsf{e}})\} \quad \mathsf{O} ::= \mathtt{object}\ \mathsf{X}\ \{\overline{\mathsf{M}}\ \overline{\mathtt{T}\ \mathtt{x} = \mathsf{e}}\} \quad \mathsf{M} ::= \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}\ \mathtt{x}})\{\mathtt{s};\mathbf{return}\ \mathsf{e}\}$$
$$\mathtt{s} ::= \big[[\mathbf{Fut}\texttt{<}\mathtt{T}\texttt{>}]\ \mathtt{x} = \big]\mathsf{X}!\mathtt{m}(\bar{\mathsf{e}}) \mid [\mathtt{T}]\ \mathtt{x} = \mathsf{e} \mid \big[\mathtt{T}\big]\ \mathtt{x} = \mathsf{e}.\mathbf{get} \mid \mathbf{skip} \mid \mathtt{s};\mathtt{s}$$

Objects communicate only by asynchronous method calls using futures. Upon a method call, a fresh future is generated on callee side and is passed to the caller. The callee writes the return value into the future upon termination of the corresponding process; anyone with the access to the future can read, but not write, into it. We only consider static sessions, in which all objects are created before the start of the system. Async is a standard imperative language with two additional statements: (1) $\mathtt{x} = \mathsf{X}!\mathtt{m}(\bar{\mathsf{e}})$ calls method `m` with parameters $\bar{\mathsf{e}}$ on object X. The generated future is stored in `x`. The caller continues execution, while the callee is computing the call on `m` or scheduling `m` for later execution if another process is currently active. (2) e.**get** reads a value from the future stored in e. If the process computing this future has not terminated, the reading process blocks.

To define a small-step reduction relation over events for the semantics of Async, we first define an event as a process action with visible communication:

Definition 2 (Events). *Let* f, f' *range over futures. An* event, *denoted by* ev, *is defined by the following grammar:*

$$\mathsf{ev} ::= \mathsf{iEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \bar{\mathsf{e}}) \mid \mathsf{iREv}(\mathsf{X}, f, \mathtt{m}) \mid \mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e}) \mid \mathsf{fREv}(\mathsf{X}, f, \mathsf{e}) \mid \mathsf{noEv}$$

An *invocation* $\mathsf{iEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \bar{\mathsf{e}})$ models that X calls $\mathsf{X}'.\mathtt{m}$ using f and passes $\bar{\mathsf{e}}$ as parameters. An *invocation reaction* $\mathsf{iREv}(\mathsf{X}, f, \mathtt{m})$ models that X starts executing

$\mathtt{m}$ to resolve f. A *resolving* $\mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})$ models that X resolves f, which contains e at the moment, by finishing the execution of $\mathtt{m}$. A *fetch* $\mathsf{fREv}(\mathsf{X}, f, \mathsf{e})$ models that X reads value e from f. Finally, noEv models no visible communication.

A configuration is composed of *processes* and *objects*. A process has a unique future f, a store σ which maps variables to literals, and the name X of its object. An object has a unique name X, an active future f, and a store ρ which maps fields to literals.

Definition 3 (Runtime Syntax of Processes and Objects). *The following grammar defines runtime processes and objects as configurations* C*:*

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

A process either is executing a method $\mathtt{m}$ for a request carried by f at some object X, represented by $\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathsf{s}), \sigma)$, or has returned e, represented by $\mathbf{prc}(\mathsf{X}, f, \mathsf{val}(\mathsf{e}), \sigma)$. An object $\mathbf{ob}(\mathsf{X}, f, \rho)$ has its name X, the future of the active process f and the heap ρ. We write $\mathbf{ob}(\mathsf{X}, \bot, \rho)$ to indicate that X is inactive. Composition of configurations is commutative and associative, i.e., $\mathsf{C}\ \mathsf{C}' = \mathsf{C}'\ \mathsf{C}$ and $\mathsf{C}\ (\mathsf{C}'\ \mathsf{C}'') = (\mathsf{C}\ \mathsf{C}')\ \mathsf{C}''$. We denote the initial configuration of a program Prgm with $\mathbb{I}(\mathsf{Prgm})$. If all processes of a configuration C have terminated, the configuration also terminates. The body of method $\mathtt{m}$ is denoted by $M(\mathtt{m})$. We write $\widehat{M}(\mathtt{m}, \overline{\mathsf{e}})$ for the initial local store of a task executing $\mathtt{m}$ with parameters $\overline{\mathsf{e}}$.

We use *traces*, sequences of pairs of events and configurations, to capture the behavior of a program. We only consider terminating runs and define big-step semantics $\mathsf{Prgm} \Downarrow \mathbf{tr}$ for *finite* traces:

Definition 4 (Run and Big-Step Semantics). *A run from* C_1 *to* C_n *is a sequence of configurations* $\mathsf{C}_1, \ldots, \mathsf{C}_n$ *with events* $\mathsf{ev}_1, \ldots, \mathsf{ev}_{n-1}$ *such that:*

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

The trace $\mathbf{tr}$ *of a run is a sequence* $(\mathsf{ev}_1, \mathsf{C}_1), \ldots, (\mathsf{ev}_m, \mathsf{C}_m)$ *where for every* $1 \leq j < m \leq n$ *there is a* C *such that* $\mathsf{C}_j \rightarrow_{\mathsf{ev}_j} \mathsf{C}$ *is in the run and* $\mathsf{ev}_j \neq \mathsf{noEv}$*. An* Async *program* Prgm *generates* $\mathbf{tr}$*, written* $\mathsf{Prgm} \Downarrow \mathbf{tr}$*, if there is a run from its initial configuration to a terminated configuration such that* $\mathbf{tr}$ *is the trace of this run.*

Fig. 3 defines the reduction relation $\rightarrow_{\mathsf{ev}}$ for the semantics. $[\![\mathsf{e}]\!]_{\sigma,\rho}$ denotes the evaluation of an expression e under stores σ and ρ. Rule (call) executes a method call on the object stores in e by generating a fresh future f' and an invocation event. The new process is not set as active upon creation by (call). By rule (start), the object X must be inactive, when the process is started. An invocation reaction event is generated. Rule (sync) synchronizes on a future f' stored in e, by checking whether the configuration contains $\mathbf{prc}(\mathsf{X}', f', \mathsf{val}(\mathsf{e}'), \sigma')$, i.e. f' is resolved, and reads the return value e'. Rule (end) terminates a process. In all other rules, the ev parameter is noEv.

Dynamic Logic. A dynamic logic combines FO-formulas over the heap with symbolic executions [1, 32] of statements. A symbolic execution uses symbolic values to describe a possible set of actual values. It does not reason about one execution of the statement, but describes a *set* of executions.

$$\textsf{(call)}\ \frac{\mathsf{C}\text{ does not contain } f' \quad [\![\mathsf{e}]\!]_{\sigma,\rho} = \mathsf{X}' \quad \mathsf{C} = \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}' \quad \mathsf{ev} = \mathsf{iEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, [\![\overline{\mathsf{e}'}]\!]_{\sigma,\rho})}{\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathsf{e}!\mathtt{m}'(\overline{\mathsf{e}'}); \mathtt{s}), \sigma)\ \mathsf{C} \to_{\mathsf{ev}} \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)\ \mathbf{prc}(\mathsf{X}', f', \mathtt{m}'(M(\mathtt{m}')), \widehat{M}(\mathtt{m}, [\![\overline{\mathsf{e}'}]\!]_{\sigma,\rho}))\ \mathsf{C}}$$

$$\textsf{(start)}\ \frac{\mathsf{ev} = \mathsf{iREv}(\mathsf{X}, f, \mathtt{m})}{\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathsf{C} \to_{\mathsf{ev}} \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}}$$

$$\textsf{(sync)}\ \frac{\mathsf{C} = \mathbf{prc}(\mathsf{X}', f', \mathsf{val}(\mathsf{e}'), \sigma')\ \mathsf{C}' \qquad [\![\mathsf{e}]\!]_{\sigma,\rho} = f' \quad \mathsf{ev} = \mathsf{fREv}(\mathsf{X}, f', \mathsf{e}')}{\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{x} = \mathsf{e}.\mathbf{get}; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C} \to_{\mathsf{ev}} \mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathtt{x} = \mathsf{e}'; \mathtt{s}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C}}$$

$$\textsf{(end)}\ \frac{\mathsf{ev} = \mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})}{\mathbf{prc}(\mathsf{X}, f, \mathtt{m}(\mathbf{return}\ \mathsf{e}), \sigma)\ \mathbf{ob}(\mathsf{X}, f, \rho)\ \mathsf{C} \to_{\mathsf{ev}} \mathbf{prc}(\mathsf{X}, f, \mathsf{val}([\![\mathsf{e}]\!]_{\sigma,\rho}), \sigma)\ \mathbf{ob}(\mathsf{X}, \bot, \rho)\ \mathsf{C}}$$

Fig. 3. The selected semantics rules. Full rules are provided in [30].

Example 2. Formula $\exists \texttt{Int}\ a.\ \big(a > 0 \land \texttt{i} > a\big) \to [\texttt{j = i*2;}]\texttt{j} > 0$ describes that if there is a number a bigger than 0 and smaller than the value stored in `i`, then after executing `j = i*2;`, variable `j` contains a positive value.

Based on ABSDL [14], we present Async Dynamic Logic (short as ADL), which extends first-order logic over program variables and heap memory with modalities that model the effect of statements. In this logic, method parameters are special variables and a modality is a formula $[\mathtt{s}]\varphi$ which holds in a configuration, say C, if φ holds in every configuration reached from C after executing `s`. We focus on the semantics of *modality-free* formulas, which have configurations as models; the semantics of modalities is a transition relation between configurations.

Definition 5 (Formulas φ). *We define the set of formulas φ and terms* t *by the following grammar, where* p *ranges over predicate symbols,* f *ranges over function symbols,* x *ranges over logical variables, and* v *ranges over logical and program variables. The set of formulas is denoted by* ADL.

$$\varphi ::= \mathbf{tt} \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{p}(\mathsf{t}\ldots\mathsf{t}) \mid \mathsf{t} \geq \mathsf{t} \mid \mathsf{t} \doteq \mathsf{t} \mid \exists \mathtt{T}\ \mathsf{x}; \varphi \mid [\mathtt{s}]\varphi \qquad \mathsf{t} ::= \mathsf{v} \mid \mathsf{f}(\mathsf{t}\ldots\mathsf{t})$$

Local program variables (i.e., `v`) are modeled as special function symbols. To model heap accesses, following Schmitt et al. [36], we use two function symbols store and select with (at least) the axiom $\mathsf{select}(\mathsf{store}(\mathit{heap}, \mathtt{f}, \mathtt{o}, \mathit{value}), \mathtt{f}, \mathtt{o}) = \mathit{value}$ where *heap* is a special local program variable modeling the heap explicitly. A special function symbol **result** is interpreted as the return value of a method, and a logical variable is *free* if it is not bound by any quantifier.

Definition 6. *A formula φ is* valid *if it evaluates to true in every configuration.*

Formulas are *global* or X*-formulas*. Global formulas refer to the heap of multiple objects, while X-formulas refer only to X. The latter contains only the function symbols for elements from X and the special function symbol **self** modeling the reference to X. For proving that an X-formula holds for a given state, if suffices to locally check the code of X. A validity calculus for ADL is presented in [15].

Definition 7. *Let φ be a formula. The weakened* X*-formula φ@X is obtained by replacing all function symbols in φ which are not exclusive to* X *(i.e., refer to the fields of other objects) by free variables and existentially quantifying over them.*

Example 3. Let fl be a field, X an object and $\mathtt{i}$ the parameter of some method in class X. Consider $\varphi = \mathsf{X.fl} > 0 \wedge \mathtt{i} > \mathsf{X.fl}$. The formula φ is an X-formula, as $\varphi = \varphi@\mathsf{X}$. The weakening for some object X' is $\varphi@\mathsf{X}' = \exists \mathtt{Int}\ a.a > 0 \wedge \mathtt{i} > a$. $\varphi@\mathsf{X}'$ does not reason about $\mathsf{X.fl}$, but still has the information of $\mathtt{i} > 1$.

4 Behavioral-Type-Based Stateful Specification

We define a specification language for global types to specify the behavior of the system. Following Sec. 3, we only represent the key constructs and leave branching to Section 7 and repetition to Sec. 6.

Definition 8 (Syntax of Global Types). Let φ, ψ range over *modality-free* ADL formulas and X_i range over object names. $[\cdot]$ denotes optional elements.

$$\mathbf{G} ::= \mathbf{main} \to \mathsf{X} : \mathtt{m}\langle\varphi\rangle.G \qquad G ::= \mathsf{X}_1 \xrightarrow{[\mathtt{x}]} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle.G \mid \mathsf{X}{\uparrow}\mathsf{e}.G \mid \mathsf{End}$$

The *calling type* $\mathsf{X}_1 \xrightarrow{[\mathtt{x}]} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle$ specifies a method call from X_1 to $\mathtt{m}$ at X_2. If $\mathtt{x}$ is not omitted above the arrow, the future of this call must be stored in location $\mathtt{x}$. The ADL-formula φ specifies (1) the call parameters passed to the callee and (2) the memory of X_1 at the moment of the call. Formula ψ is the postcondition of the callee process and specifies the state of X_2 and the return value once $\mathtt{m}$ terminates. The exact point of termination is derived during projection. The initial method call $\mathbf{main} \to \mathsf{X} : \mathtt{m}\langle\psi\rangle$ only specifies the postcondition of the process running $\mathsf{X.m}$. Type $\mathsf{X}{\uparrow}\mathsf{e}$ specifies a synchronization on the future, to which the expression e evaluates. Every synchronization must be specified. End specifies the end of communication.

$\mathbf{G}$ denotes a complete protocol with an initializing method call, while G denotes partial types. Even without fields in the formula, the implementation is referenced in the specification, as endpoints are object names. Object and method types share the same syntax. Together we call them *local types*. The grammar of local types is defined as follows:

Definition 9 (Syntax of Local Types). Let φ range over *modality-free* ADL formulas and X_i range over object names. $[\cdot]$ denotes optional elements.

$$\mathbf{L} ::= ?\mathtt{m}\langle\varphi\rangle.L \qquad L ::= ?\mathtt{m}\langle\varphi\rangle.L \mid \mathsf{X}!_{[\mathtt{x}]}\mathtt{m}\langle\varphi\rangle.L \mid \mathsf{Put}\ \varphi.L \mid \mathsf{Read}\ \mathsf{e}.L \mid \mathsf{skip}.L \mid \mathsf{End}$$

The type $?\mathtt{m}\langle\varphi\rangle$ denotes the start of a process computing $\mathtt{m}$ in a state where formula φ holds. Formula φ is the precondition of $\mathtt{m}$ and describes the local state and method parameters of $\mathtt{m}$. Type $\mathsf{Put}\ \varphi$ denotes the termination in a state where φ holds. Formula φ is a postcondition and describes the return value and the local store. Contrary to global types, a postcondition of a process is not annotated at the call, but at the point of termination because the point of termination is now explicit. Type $\mathsf{X}!_{[\mathtt{x}]}\mathtt{m}\langle\varphi\rangle$ corresponds to the caller side of $\mathsf{X}_1 \xrightarrow{[\mathtt{x}]} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle$. Type $\mathsf{Read}\ \mathsf{e}$ models a read from e and skip denotes no communication. As for global types, End models the end of communication. In our examples, we omit End for brevity's sake. We use $\mathbf{L}$ for complete local types and L for partial local types.

Projection has three steps: (1) projection of global types on objects, (2) condition propagation, and (3) projection of object types on methods.

Projection on Objects. The projection on objects ensures that every object can access all locations occurring in its specification and adds $\mathsf{Put}\ \varphi$ at the correct position. This requires an additional parameter in the projection to keep track of which process is specified to be active and what its postcondition is.

To track the postcondition of the last active method of an object, we use a partial function $\mathsf{ac} : \mathsf{O} \rightharpoonup \mathsf{ADL}$ to map objects to formulas. If no method was active on X yet, ac is undefined, written $\mathsf{ac}(\mathsf{X}) = \bot$. The projection of G on an object X is denoted by $G \upharpoonright_{\mathsf{ac}} \mathsf{X}$. The selected projection rules for methods calls and termination are given in Fig. 4. We write $\mathsf{ac}_\bot$ for the function defined by $\forall \mathsf{X}.\ \mathsf{ac}(\mathsf{X}) = \bot$. For updates, we write $\mathsf{ac}[\mathsf{X} \mapsto \psi](\mathsf{X}') = \begin{cases} \psi & \text{if } \mathsf{X} = \mathsf{X}' \\ \mathsf{ac}(\mathsf{X}') & \text{otherwise} \end{cases}$.

$$
\begin{aligned}
&(1)\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X}_1 = \mathsf{X}_2!_{\mathtt{x}}\mathtt{m}\langle\varphi\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) \text{ if } \mathsf{ac}(\mathsf{X}_1) \neq \bot \wedge \varphi = \varphi@\mathsf{X}_1 \\
&(2)\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X}_2 = \begin{cases} ?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) & \text{if } \mathsf{ac}(\mathsf{X}_2) = \bot \\ \mathsf{Put}\ \mathsf{ac}(\mathsf{X}_2).?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) & \text{if } \mathsf{ac}(\mathsf{X}_2) \neq \bot \end{cases} \\
&(3)\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X} = \mathsf{skip}.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}) \text{ if } \mathsf{X}_2 \neq \mathsf{X} \neq \mathsf{X}_1 \\
&(4)\mathbf{main} \to \mathsf{X}_2 : \mathtt{m}\langle\varphi\rangle.G \upharpoonright_{\mathsf{ac}_\bot} \mathsf{X}_1 = \begin{cases} ?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) & \text{if } \mathsf{X}_2 = \mathsf{X}_1 \\ \mathsf{skip}.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) & \text{if } \mathsf{X}_2 \neq \mathsf{X}_1 \end{cases} \\
&(5)\mathsf{End} \upharpoonright_{\mathsf{ac}} \mathsf{X} = \begin{cases} \mathsf{Put}\ \mathsf{ac}(\mathsf{X}).\mathsf{End} & \text{if } \mathsf{ac}(\mathsf{X}) \neq \bot \\ \mathsf{End} & \text{if } \mathsf{ac}(\mathsf{X}) = \bot \end{cases}
\end{aligned}
$$

Fig. 4. The selected rules for projection on objects.

When projecting on caller X_1, a sending local type is generated by (1) if X_1 has an active process ($\mathsf{ac}(\mathsf{X}_1) \neq \bot$) and the precondition can be proven by the caller ($\varphi = \varphi@\mathsf{X}_1$). If the callee has an active process (i.e., the last active postcondition exists: $\mathsf{ac}(\mathsf{X}_2) \neq \bot$), then the termination type for the active process is added by (2) before the receiving type. If the callee is specified as being inactive (i.e., no process was running before and no postcondition is tracked $\mathsf{ac}(\mathsf{X}_2) = \bot$), then only the receiving type is added by (2). When projecting on any other object, skip is added by (3). In any case, ac is updated and maps the callee to a new postcondition. Rules (4) and (5) are straightforward. As usual, projection is undefined if no rule matches, and we omit $\mathsf{ac}_\bot$ and write just $\mathbf{G} \upharpoonright \mathsf{X}$.

Propagation. In our concurrency model the heap does not change if no process is active. All guarantees from the last active process still hold for the next process. By propagation, formulas are added from the postcondition of one method to the precondition of the next. Propagation moves formulas from where they *must hold* to all points where they still are *assumed to hold*. Propagation replaces a partial local type, if the partial type matches the given pattern.

Definition 10 (Propagation). *The propagation function* prp *is defined via term* rewriting *(denoted* $\rightsquigarrow$*) as follows.* prp^* *denotes the fixpoint of rewriting.*

(1) $\mathsf{Put}\ \varphi.?\mathtt{m}\langle\psi\rangle \rightsquigarrow \mathsf{Put}\ \varphi.?\mathtt{m}\langle\psi \wedge \varphi@\mathsf{X}\rangle$ *where* X *is the target object*

Projection on Methods. The projection on a method, denoted by $L \upharpoonright_{\mathtt{m}'} \mathtt{m}$, results in a *set* of method types. A method may have multiple method types, as long as the method types are *distinguishable*, which means that they have non-overlapping preconditions. Formally, two preconditions φ_1 and φ_2, are distinguishable if the formula $\neg(\varphi_1 \wedge \varphi_2)$ is valid. In the case of overlapping preconditions, multiple preconditions can hold at the same time and it is not guaranteed that the correct type will be realized.

The rules for projection on a method are straightforward and we refer to the Section 2 for an example and to the appendix for full definitions.

Definition 11 (Well-Formedness). *A global type* $\mathbf{G}$ *is* well-formed*, if the projections on all methods are defined and all types of a method are distinguishable.*

Semantics of Types as Constraints on Traces. To formalize the behavioral types of the previous section, we transform them into first-order constraints over traces.

We define $\mathbb{C}$ as a function transforming global types to constraints on traces. Recall that we have defined C for configurations and ev for events. The primitive $\mathsf{C}(i)$ references the ith configuration and $\mathsf{ev}(i)$ references the ith event in a trace. We use events and formulas as colors and thus include futures, method names, literals and object names in the domain. Constraints refer to ADL formulas φ with $\mathsf{C}(i) \models \varphi$, meaning that in the ith configuration, φ holds.

To restrict a constraint to a subtrace, we use *relativization* [23], a *syntactic* restriction of constraint γ to a substructure described by another constraint χ.

Definition 12. *Let $\chi(x)$ be a constraint with a free variable x of data type $\mathtt{T}$ and γ another constraint. The relativization of γ with $\chi(x)$, written $\gamma[x \in \mathtt{T}/\chi]$, replaces all subconstraints of the form $\exists y \in \mathtt{T}.\gamma'$ in γ by $\exists y \in \mathtt{T}.\chi(y) \wedge \gamma'$.*

The main rules for translating $\mathbf{G}$ into a constraint $\mathbb{C}(\mathbf{G})$ are defined as follows.

Definition 13 (Semantics of Global Types). *Predicate* $\mathsf{res}(i)$ *holds if* $\mathsf{ev}(i)$ *is a resolving event and* $\mathsf{A}(i, \mathsf{X})$ *holds if* X *is active in* $\mathsf{C}(i)$.

$$
\begin{aligned}
&\textit{(1)}\mathbb{C}(\mathbf{main} \to \mathsf{X}_2 : \mathtt{m}\langle\psi\rangle.G) \;=\; \exists j, k.\ \exists f.\ \exists \mathsf{e}'.\ \mathsf{ev}(j) \doteq \mathsf{iREv}(\mathsf{X}_2, f, \mathtt{m}) \wedge \mathsf{C}(j) \models \varphi@\mathsf{X}_2 \wedge \\
&\qquad \mathsf{ev}(k) \doteq \mathsf{fEv}(\mathsf{X}_2, f, \mathsf{e}') \wedge \mathsf{C}(k) \models \psi \wedge \forall l.\, l \neq j \wedge l \neq k \Rightarrow \mathsf{res}(l) \wedge \mathbb{C}(G) \\
&\textit{(2)}\mathbb{C}(\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi, \psi\rangle) \;=\; \exists i, j, k.\ \exists f.\ \exists \mathsf{e}, \mathsf{e}'. \\
&\qquad \mathsf{ev}(i) \doteq \mathsf{iEv}(\mathsf{X}_1, \mathsf{X}_2, f, \mathtt{m}, \mathsf{e}) \wedge \mathsf{C}(i) \models \varphi \wedge \mathsf{ev}(j) \doteq \mathsf{iREv}(\mathsf{X}_2, f, \mathtt{m}) \wedge \mathsf{C}(j) \models \varphi@\mathsf{X}_2 \wedge \\
&\qquad \mathsf{ev}(k) \doteq \mathsf{fEv}(\mathsf{X}_2, f, \mathsf{e}') \wedge \mathsf{C}(k) \models \psi \wedge \mathsf{C}(i) \models (\mathsf{X}_1.\mathtt{x} \doteq f) \wedge \forall l.\ l \neq i \wedge l \neq j \wedge l \neq k \Rightarrow \mathsf{res}(l) \\
&\textit{(3)}\mathbb{C}(G_1.G_2) = \bigwedge_{\mathsf{X}} \left(\exists i \in \mathbb{N}.\ \mathbb{C}(G_1)[j \in \mathbb{N}/\mathsf{A}(j, \mathsf{X}) \Rightarrow j < i] \wedge \mathbb{C}(G_2)[j \in \mathbb{N}/\mathsf{A}(j, \mathsf{X}) \Rightarrow j \geq i]\right)
\end{aligned}
$$

The constraint (1) for the call type has three events modeling (1) a call, (2) the start of the process and (3) the existence of the termination of the process. Moreover, the projected formulas hold at the configurations for these events. Every other event is a fEv. The exact position of termination (i.e., fEv events) is not specified in global types, so we do not constrain them. Reading from a location is defined analogously. The translation of $G_1.G_2$ models that there is a

position i such that, for every object X, the events described in $\mathbb{C}(G_1)$ are in the subtrace before i and those in $\mathbb{C}(G_2)$ are in the subtrace after i.

The restriction is applied for every object, to ensure the following property: If a trace is a model for the translation of a type $\mathbf{G}$, then for each participating object (1) all events of this objects have the same order as specified in $\mathbf{G}$ and (2) at the moment of the event, the corresponding FO formula holds. The translation of, e.g., $\mathsf{X}_1 \to \mathsf{X}_2 : \mathtt{m}_2.\mathsf{X}_1 \to \mathsf{X}_3 : \mathtt{m}_3$ describes that $\mathsf{X}_2.\mathtt{m}_2$ is called before $\mathsf{X}_2.\mathtt{m}_2$, but does *not* describe that the execution start in the same order. Thus, there are multiple possible event order satisfying this constraint, but from *every local point of view* the differences between these traces are not visible.

5 Analysis

Verifying deadlock freedom requires a *Points-To* analysis in addition to a type system. Deadlock freedom is equivalent to cycle-freedom of causality graphs [17] in Active Objects. The *causality graph* of a global type $\mathbf{G}$ is $\mathbb{G}(\mathbf{G}) = (V, E)$. Each node $L \in V$ is a local type, and each edge $(L_1, L_2) \in E$ models that L_2 must happen after L_1.

Definition 14 (Causality Graph). *Let $\mathbf{G}$ be a well-formed global type. The nodes of its causality graph $\mathbb{G}(\mathbf{G})$ are all partial local types derived from projecting $\mathbf{G}$ on all endpoints. An edge (L_1, L_2) is added if either (1) $L_1 = L.L_2$ is a partial type for some L in some projection on some object or (2) L_1 is the sending type and L_2 the receiving type from the projection of a single calling type.*

Note that global types do not contain sufficient information to deduce all causality, e.g., the causality of **get** statements cannot be deduced from a global type because synchronizations on futures are specified over *locations*. We use a Points-To analysis for futures [17] instead. For generating a causality graph, we first derive a *partial* causality graph from the global type, and then we apply the Points-To analysis during type checking for the graph completion by deducing the missing edges. The Points-To analysis, defined below, determines which methods are responsible to resolve the futures in a given expression.

Definition 15 (Points-To). *The* Points-To analysis *determines the set* $\mathsf{p2}(\mathsf{e})$ *of methods, which may have resolved the future stored in an input expression* e. *We can express this using constraints, to integrate it into the type system:*

$$\forall i \in \mathbb{N}.\ \mathsf{C}(i) \doteq \mathbf{prc}(\mathsf{X}', f, \mathsf{val}(\mathsf{e}'), \sigma)\ \mathbf{prc}(\mathsf{X}, f', \mathtt{m}'(\mathtt{x} = \mathsf{e}.\mathbf{get}; \mathtt{s}''), \sigma'')\ \mathsf{C} \wedge [\![\mathsf{e}]\!]_{\sigma,\rho} = f \to$$
$$\exists j \in \mathbb{N}.\ j < i \wedge \mathsf{C}(j) \doteq \mathbf{prc}(\mathsf{X}', f, \mathtt{m}(\mathtt{s}), \sigma')\ \mathsf{C}' \wedge \mathtt{m} \in \mathsf{p2}(\mathsf{e})$$

Whenever a $\mathsf{e}.\mathbf{get}$-statement is checked against a type $\mathsf{Read}\ \mathsf{e}$, edges are added between the node of termination type of the methods which e can point to, and the node of the current type $\mathsf{Read}\ \mathsf{e}$. Although Points-To is undecidable, well-scaling tools which safely overapproximate are available [2].

Definition 16 (Admissibility). *A causality graph is* admissible *if (1) every path is cycle-free and (2) for every object X, and for any pair of receiving types of X, there exists a connecting path without an edge of the form* $(\mathsf{Put}\ \varphi, ?\mathtt{m}\langle\psi\rangle)$.

The graph on page 6 is admissible. With a non-admissible graph, methods may deadlock (violating (1)) or be executed in the wrong order (violating (2)).

Type System and Analysis. The auxiliary ADL-formula $\mathsf{post}(\mathsf{X}.\mathtt{m}, \varphi)$ models that the value in every future resolved by $\mathsf{X}.\mathtt{m}$ satisfies φ, while formula $\mathsf{Post}(\mathbf{G})$ represents the conjunction of all postconditions specified in $\mathbf{G}$. Figure 5 shows selected typing rules invoking the validity calculus [15] and Points-To analysis.

Before introducing the typing rules, we define $\mathsf{Roles}(\mathbf{G})$ as the set of objects in $\mathbf{G}$, $\mathbb{G}(\mathbf{G}) + E$ as the set of edges of $\mathbb{G}(\mathbf{G})$ and E (i.e., E is added into $\mathbb{G}(\mathbf{G})$), $\mathsf{term}(\mathtt{m})$ as the set of $\downarrow$ nodes of method $\mathtt{m}$, and $\mathsf{node}(\mathtt{s})$ as the set of nodes referring to the types that have typed $\mathtt{s}$. We define three kinds of type judgments:

(I) The Type Judgment for Programs. $\vdash \mathsf{Prgm} : \mathbf{G}$ checks Prgm against global type $\mathbf{G}$. The well-formedness of $\mathbf{G}$ (Def. 11) is ensured during type checking. Rule (T-Main) checks that every endpoint in $\mathbf{G}$ is implemented in Prgm, the main block makes the correct initializing call and checks each object against its object type. The edges collected from the typing rules for objects are added to the partial causality graph $\mathbb{G}(\mathbf{G})$ and the resulting graph is checked for admissibility.

(II) The Type Judgment for Objects. $\Phi \vdash \mathsf{O} : \mathbf{L} \rhd E$ checks whether O is well-typed by $\mathbf{L}$ under a given E with Φ. E is a set of causality edges and Φ is a set of ADL formulas. Rule (T-Object) projects $\mathbf{L}$ on all methods, checks each method $\mathtt{m}_i$ by $\mathbf{L} \upharpoonright \mathtt{m}_i$ and collects all resulting edges.

$$\textsf{(T-Main)}\ \frac{\begin{array}{c}\mathsf{O}_i = \mathtt{object}\ \mathsf{X}_i\{\dots\} \quad \mathsf{Roles}(\mathbf{G}) = \{\mathsf{X}_1, \dots, \mathsf{X}_n\} \qquad \mathbb{G}(\mathbf{G}) + \bigcup_{i \le n} E_i \text{ admissible} \\ \exists j \le n.\ \mathbf{G} = \mathtt{main} \to \mathsf{X}_j : \mathtt{m}\langle\varphi\rangle.G \quad \forall i \le n.\ \mathsf{Post}(\mathbf{G}) \vdash \mathsf{O}_i : \mathsf{prp}^*(\mathbf{G} \upharpoonright \mathsf{X}_i) \rhd E_i\end{array}}{\vdash \mathsf{O}_1 \quad \dots \quad \mathsf{O}_n \quad \mathtt{main}\{\mathsf{X}_j!\mathtt{m}()\} : \mathbf{G}}$$

$$\textsf{(T-Object)}\ \frac{\begin{array}{c}\forall i \le n.\ \mathbf{L} \upharpoonright_{\mathsf{ac}} \mathtt{m}_i = ?\mathtt{m}_i\langle\varphi_i\rangle.L_i \\ \forall i \le n.\ \Phi, \varphi_i, \textbf{skip} \vdash \mathtt{s}_i : L_i \rhd E_i \qquad E = \bigcup_{i \le n} E_i\end{array}}{\Phi \vdash \mathtt{object}\ \mathsf{X}\{\mathtt{T}_1\ \mathtt{m}_1(\overline{\mathtt{T}\ \mathtt{x}})\{\mathtt{s}_1\}\ \dots\ \mathtt{T}_n\ \mathtt{m}_n(\overline{\mathtt{T}\ \mathtt{x}})\{\mathtt{s}_n\}\quad \overline{\mathtt{T}\ \mathtt{x} = \mathtt{e}}\} : \mathbf{L} \rhd E}$$

$$\textsf{(T-Return)}\ \frac{\Phi \Rightarrow [\mathtt{s}; \textbf{return}\ \mathtt{e}]\varphi}{\Phi, \mathtt{s} \vdash \textbf{return}\ \mathtt{e} : \mathsf{Put}\ \varphi \rhd E} \qquad \textsf{(T-Call)}\ \frac{\begin{array}{c}\Phi, \mathtt{s}; \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\overline{\mathtt{e}}) \vdash \mathtt{s}' : L \rhd E \\ \Phi \Rightarrow [\mathtt{s}; \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\overline{\mathtt{e}})]\varphi\end{array}}{\Phi, \mathtt{s} \vdash \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\overline{\mathtt{e}}); \mathtt{s}' : \mathsf{X}!_{\mathtt{x}}\mathtt{m}\langle\varphi\rangle.L \rhd E}$$

$$\textsf{(T-Get)}\ \frac{\begin{array}{c}\Phi, \mathtt{s}; \mathtt{T}\ \mathtt{x} = \mathtt{e}.\textbf{get} \vdash \mathtt{s}' : L \rhd E' \\ E = E' \cup \{(n, n') | \exists \mathtt{m} \in \mathsf{p2}(\mathtt{e}).\ n \in \mathsf{term}(\mathtt{m}) \wedge n' \in \mathsf{node}(\mathtt{s}; \mathtt{e}.\textbf{get})\}\end{array}}{\Phi, \mathtt{s} \vdash \mathtt{T}\ \mathtt{x} = \mathtt{e}.\textbf{get}; \mathtt{s}' : \mathsf{Read}\ \mathtt{e}.L \rhd E}$$

Fig. 5. The selected typing rules.

(III) The Type Judgment for Statements. $\Phi, \mathtt{s} \vdash \mathtt{s} : L \rhd E$ checks whether $\mathtt{s}$ is well-typed by L under a given E with $\Phi, \mathtt{s}$. The environment $\mathtt{s}$ are the statements type-checked so far. Whenever an ADL formula is checked, a validity check is performed and $\mathtt{s}$ is added in the modality to consider the side-effects on the

heap memory so far. However, these are not recorded in E: The causality edges only record which method a **get** statement synchronizes on. Rule (T-Return) checks that after executing all the type-checked statements, the **return** statement results in a state where φ holds. Rule (T-Call) also checks the formula φ which describes the state when the call has to be executed. Rule (T-Get) additionally executes the Points-To analysis and adds all the edges as described in the previous section.

Theorem 1 (Deadlock Freedom and Protocol Adherence). *Let* Prgm *be a program and* $\mathbf{G}$ *be a global type. If* Prgm *is well-typed against* $\mathbf{G}$ *then (1)* Prgm *does not deadlock and (2) every generated trace from* Prgm *satisfies* $\mathbb{C}(\mathbf{G})$*:*

$$\vdash \mathsf{Prgm} : \mathbf{G} \to \left(\forall \mathbf{tr}.\ \mathsf{Prgm} \Downarrow \mathbf{tr} \to \mathbf{tr} \models \mathbb{C}(\mathbf{G})\right)$$

6 Loops and Repetition

In this section we present the whole workflow of the previous section for Async extended with repetition. The language is extended with loops and the types with *repetition* types $(G)^*_\varphi$ (resp. $(L)^*_\varphi$). A repetition type resembles a Kleene-star and models the finite repetition of the type G (resp. L). The formula φ is a loop invariant and has to be satisfied whenever a loop iteration starts or ends.

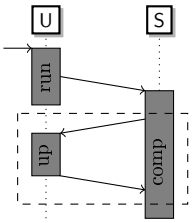
Definition 17 (Syntax with Repetition).

$$\mathtt{s} ::= \ldots \mid \textbf{while}(\mathsf{e})\{\mathtt{s}\} \qquad G ::= \ldots \mid (G)^*_\varphi.G \qquad L ::= \ldots \mid (L)^*_\varphi \,.\, L$$

By syntactic restrictions, the local type $\mathbf{L}$ of an object cannot have the form $(L)^*_\varphi.L'$, which forbids it to start with a loop. The intuition behind this restriction is that every loop has an invariant that an object must guarantee before executing the next iteration. If an object is not active before the loop, it cannot guarantee the invariant in the very beginning, thus repetition can start with the second action at the earliest. Below give an example for using invariants.

Example 4. Consider a big data analysis webtool with a client-side GUI U and a server-side computational server S. We model the following scenario:

U sends data to the computational server by calling `S.comp`. To stay responsive, U ends its initial process. U is called repeatedly on `G.up` by the server to update the progress. Whenever U is updated, the server also gets information by reading from the future of the last call to `U.up`. The sequence diagram to the right illustrates the protocol. During updating, U must stay in a state expecting to receive updates from the server. It is therefore important to specify that field `U.expect` is not `Nil`.



$$\mathbf{main} \to \mathsf{U}{:}\mathtt{run}\langle\top\rangle \,.\, \mathsf{U} \to \mathsf{S}{:}\mathtt{comp}\langle\top,\top\rangle \,.\, \left(\mathsf{S} \xrightarrow{\mathtt{x}} \mathsf{U}{:}\mathtt{up}\langle\top,\top\rangle \,.\, \mathsf{S}\uparrow\mathtt{x}\right)^*_{\mathsf{U}.\mathtt{expect}\neq\mathtt{Nil}} \,.\, \mathbf{end}$$

The invariant $\mathsf{U}.\mathtt{expect} \neq \mathtt{Nil}$ specifies the condition that field `U.expect` is a non-empty list. This is propagated during projection, which results in the following local type for `U.up`

$$?\mathtt{update}\langle \mathbf{self}.\mathtt{expect} \neq \mathtt{Nil}\rangle.\mathsf{Put}\ \mathbf{self}.\mathtt{expect} \neq \mathtt{Nil}$$

There is no repetition because *being repeatedly called* is only visible for the whole object, not a single process. The type of `S.cmp` however contains a repetition:

$$?\texttt{comp}\langle\top\rangle.\big(\texttt{U}!_{\texttt{x}}\texttt{up}\langle\top\rangle.\mathsf{Read}\ \texttt{x}\big)_{\exists l.\ l\neq\texttt{Nil}}.\mathsf{Put}\ \top$$

The workflow is the same as described above. We provide the projection, translation, propagation and typing rules as extension of the previous systems.

Definition 18 (Projection Rules for Loops).
The auxiliary predicate $rcv(\mathsf{X}, G)$ *holds if* X *is specified as being called in* G.

$$\big((G)^*_\varphi.G'\big)\upharpoonright_{\mathsf{ac}}\mathsf{X} = \begin{cases} \mathsf{Put}\ \mathsf{ac}(\mathsf{X}).(L'')^*_{\varphi@\mathsf{X}}.L' & \text{if } L\neq\mathsf{skip}\wedge\ \ rcv(\mathsf{X},G)\wedge\mathsf{ac}(\mathsf{X})\neq\bot\wedge cs(\varphi) \\ (L)^*_{\varphi@\mathsf{X}}.L' & \text{if } L\neq\mathsf{skip}\wedge\neg rcv(\mathsf{X},G)\wedge\mathsf{ac}(\mathsf{X})\neq\bot\wedge cs(\varphi) \\ L' & \text{if } L=\mathsf{skip}\wedge cs(\varphi) \end{cases}$$

$$\text{Where } G.\mathsf{End}\upharpoonright_{\mathsf{ac}[\mathsf{X}\mapsto\bot]}\mathsf{X} = L'', G\upharpoonright_{\mathsf{ac}}\mathsf{X} = L \text{ and } G'\upharpoonright_{\mathsf{ac}}\mathsf{X} = L'$$

The auxiliary formula $cs(\varphi)$ specifies that all weakenings of φ imply φ. This is necessary to reject invariants that connect multiple heaps: e.g., this condition would reject $\texttt{G.i} \doteq \texttt{S.i}$, as it cannot be guaranteed by `G` and `S` separately. This condition, however, admits $\texttt{G.i} \doteq 1 \wedge \texttt{S.i} = 1$. The first rule projects global types to object types. The first case is applied if the object participates in the repetition of the inner type G *by being repeatedly called*. The last active process must terminate first and the repeatedly called method must terminate within the repetition. The termination inside the loop is ensured by projecting the inner type with an appended End. The second case is applied if the object participates in the repetition ($L \neq \mathsf{skip}$) by any other repeated action then being called ($\neg rcv(\mathsf{X}, G)$). Finally, the last case skips the repetition if the object does not participate in it.

The second rule projects object types to methods. The rule distinguishes whether the whole process is inside the repetition or not. If the process is completely inside, the repetition is removed, as it is not visible to the method.

In presence of repetition, invariants have to be propagated inside the repeated, the previous, and the next types. The following definition summarizes gives the rules for repetition, additionally to rule (1) in Def. 10.

Definition 19 (Rules for Propagation for Repetition).

(2) $\mathsf{Put}\ \varphi.(L)^*_\psi \rightsquigarrow \mathsf{Put}\ \varphi\wedge\psi.(L)^*_\psi$ *(3)* $(L)^*_\psi.?\texttt{m}'\langle\varphi\rangle \rightsquigarrow (L)^*_\psi.?\texttt{m}'\langle\varphi\wedge\psi\rangle$

(4) $(L)^*_\varphi.(L)^*_\psi \rightsquigarrow (L)^*_{\varphi\wedge\psi}.(L)^*_\psi$ *(5)* $(?\texttt{m}'\langle\varphi\rangle.L.\mathsf{Put}\ \varphi')^*_\psi \rightsquigarrow (?\texttt{m}'\langle\varphi\wedge\psi\rangle.L.\mathsf{Put}\ \varphi'\wedge\psi)^*_\psi$

Since loop invariants have to hold *before* the first repetition, rule (2) ensures that the last process before a repetition satisfies the invariant when terminating. Rule (3) adds an invariant to the next process, as the invariant also holds *after* the last repetition. Rule (4) is another case of the first one, in case two repetitions are succeeding each other. Finally, rule (5) adds the invariant to the processes inside the repetition. This rule enables the use of the invariant in the first method of the repetition and ensures that the last method reestablishes the invariant.

For the translation into constraints, first-order constraints are not expressive enough. The Kleene star constraint resembles regular languages and we thus use

monadic second order logic (MSO) to capture repetition. MSO extends first-order logic with a quantifier $\exists Y \subseteq Z$ which quantities over subsets of Z and a $\in$ primitive to express membership of those sets. The extension of relativization is straightforward [23]. We now extend the semantics of types as constraints from Def. 13 to repetition:

Definition 20 (Semantics of Repetition). *The semantics of repeated types uses a set of boundary indices X, between which the inner translation must. Also, the invariant has to hold at every boundary.*

$$\begin{aligned}\mathbb{C}\big((G^*_\varphi)\big) \;=\; & \exists X \subseteq \mathbb{N}.\ \exists i, j \in X.\ \big(\forall k \in \mathbb{N}.\ i < k \leq j\big) \wedge \forall i \in X.\ \mathsf{C}(i) \models \varphi \wedge \\ & \forall i, j \in X.\ \Big(\big(\forall k \in X.\ k \geq j \vee k \leq i\big) \Rightarrow \big(\mathbb{C}(G)\big)[n \in \mathbb{N} / i < n \leq j]\Big)\end{aligned}$$

The typing rule for repetition resembles invariant rules from Hoare calculi [26]:

$$\frac{\varphi \wedge \mathsf{Post}(\mathbf{G}), \textsf{skip} \vdash \mathsf{s}' : L' \rhd E'' \qquad \Phi \Rightarrow [\mathsf{s}'']\varphi \qquad \varphi \wedge \mathsf{Post}(\mathbf{G}) \Rightarrow [\mathsf{s}]\varphi \qquad \varphi \wedge \mathsf{Post}(\mathbf{G}), \textsf{skip} \vdash \mathsf{s} : L \rhd E' \qquad E = E' \cup E''}{\Phi, \mathsf{s}'' \vdash \textsf{while}\ e\ \{\mathsf{s}\}; \mathsf{s}' : (L)^*_\varphi\ .\ L' \rhd E}\ (\textsf{T-While})$$

The first premise continues the type checking of the program, in an environment where only the information in the invariant (and the global information in Post, as defined in Section 5) is available. The second and third premises check that the invariant holds initially and is preserved by the loop body. The forth premise checks the loop body and the last premise combines the derived causality edges. The extension of the causality graph is described in [29].

Corollary 1. *Theorem 1 holds for the system with repetition.*

7 Branching

Active Object have multiple ways to communicate the choice how to continue the protocol and how an object reacts on it:

- (1) The choice is communicated via method selection, i.e., each branch corresponds to a different method call.
- (2) The choice is communicated via futures, i.e., other objects must react to the choice of an object by reading its future.
- (3) The choice is communicated via the heap memory, i.e., processes must behave according to some condition for the memory.

We aim to stick with standard imperative statements and must regard the restriction that an **if** statement can only choose between two branches, while a protocol may describe more than two. In our analysis of branching, choice is communicated:

- (1) method calls and condition on the passed data for new process running on other objects

– (2) the condition on future for already running processes running on (possibly) other objects.
– (3) via post-conditions to processes running later on the same object.

Definition 21 (Syntax with Branching).

$$\mathtt{s} ::= \ldots \mid \mathsf{if}(\mathsf{e})\mathtt{s}\ \mathsf{else}\ \mathtt{s}\ \mathsf{fi} \qquad G ::= \ldots \mid \mathsf{X}\{\langle\varphi_i\rangle, (\mathsf{X}_{ij}\langle\varphi_{ij}\rangle)_{j\in J}; G_i\}_{i\in I}$$
$$L ::= \ldots \mid \oplus\{L_i\}_{i\in I} \mid \&\{\mathsf{X}.\mathtt{m}\langle\varphi_i\rangle; L_i\}_{i\in I}$$

The global type $\mathsf{X}\{\langle\varphi_i\rangle, (\mathsf{X}_{ij}\langle\varphi_{ij}\rangle)_{j\in J}; G_i\}_{i\in I}$ describes that X chooses a branch G_i. The formulas φ_i are *additional* postconditions for the choosing process. Other process can read the choice by reading this future. In $\mathsf{X}_{ij}\langle\varphi_{ij}\rangle)_{j\in J}$, we describe that the currently active process of X_{ij} has the additional postcondition φ_{ij}. The local type $\oplus\{L_i\}_{i\in I}$ is an active choice and $\&\{\mathsf{X}.\mathtt{m}\langle\varphi_i\rangle; L_i\}_{i\in I}$ is a passive choice. The branch must be taken by reading the future from $\mathsf{X}.\mathtt{m}$ and evaluating φ_i.

Definition 22 (Projection Rules for Branching). *Given the ith branch* $\langle\varphi_i\rangle, (\mathsf{X}_{ij}\langle\varphi_{ij}\rangle)_{j\in J}; G_i$*, we denote the updated* ac *function with*

$$\mathsf{ac}_i = \mathsf{ac}[\mathsf{X} \mapsto \mathsf{ac}(\mathsf{X}) \wedge \varphi_i@\mathsf{X}][\mathsf{X}_{ij} \mapsto \mathsf{ac}(\mathsf{X}_{ij}) \wedge \varphi_{ij}@\mathsf{X}_{ij}]_{j\in J}$$

The auxiliary predicate allAct *states that all mentioned objects and occur in all branches are active and* dist *states that a set of formulas does not overlap.*

$$\mathsf{allAct} = \mathsf{ac}(\mathsf{X}) \neq \bot \wedge \bigwedge_{\substack{i\in I\\ j\in J}} \mathsf{ac}(\mathsf{X}_{ij}) \neq \bot \wedge \forall i, i' \in I.\ \forall j.\ \mathsf{X}_{ij} = \mathsf{X}_{i'j}$$

$$\mathsf{dist}(\{\varphi_1, \ldots, \varphi_n\}) = \forall i, j < n.\ i \neq j \rightarrow (\varphi_i \wedge \varphi_j \textit{ is unsatisfiable})$$

Figure 6 shows the projection rules for branching.

The projection rule from global to object-local types has four cases: the first two are straightforward for the choosing process and the currently active reacting processes. The third case handles objects which behave the same in all branches and the forth handles objects which are active in only one. The projection on the passive choice moves the Read type from its position after the choice in front of it: The global type has no explicit point where a process terminates, thus the read must be after the choice which adds the postcondition to the choosing process. However the **get** statement must be before the **if** statement, which relies on the read value in the guard.

Definition 23 (Translation into MSO for Branching). *For the translation into MSO constraints, we use the auxiliary predicate* $\mathsf{firstTerm}(i, \mathsf{X})$ *that states that the ith position in the trace refers to the first resolving event from* X *and the auxiliary predicate* $\mathsf{lastTerm}(i, \mathsf{X}.\mathtt{m})$ *that states that the ith position in the trace refers to the last resolving event of* $\mathsf{X}.\mathtt{m}$*.*

$$\mathsf{firstTerm}(i, \mathsf{X}) = \forall j.\ \big(\exists f.\ \exists \mathtt{m}.\ \exists \mathsf{e}.\ ev(j) \doteq \mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})\big) \rightarrow i \leq j$$
$$\mathsf{lastTerm}(i, \mathsf{X}.\mathtt{m}) = \forall j.\ \big(\exists f.\ \exists \mathsf{e}.\ ev(j) \doteq \mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e})\big) \rightarrow i \geq j$$

$$\left(\mathsf{X}\{\langle\varphi_i\rangle,(\mathsf{X}_{ij}\langle\varphi_{ij}\rangle)_{j\in J}; G_i\}_{i\in I}\right)\upharpoonright_{\mathsf{ac}}\mathsf{X}' =$$

$$\begin{cases}\oplus\{L_i\}_{i\in I} & \text{if } \mathsf{X}=\mathsf{X}'\wedge\mathsf{allAct}\wedge G_i\upharpoonright_{\mathsf{ac}_i}\mathsf{X}'=L_i\\ \&\{\mathsf{X}.\mathtt{m}\langle\varphi_i@\mathsf{X}'\rangle; L_i\} & \text{if } \mathsf{X}_{ij}=\mathsf{X}'\wedge\mathsf{allAct}\wedge G_i\upharpoonright_{\mathsf{ac}_i}\mathsf{X}'=L_i\\ L & \text{if } \mathsf{ac}(\mathsf{X}')=\bot\wedge\forall i.\ G_i\upharpoonright_{\mathsf{ac}_i}\mathsf{X}'=L\\ L & \text{if } \mathsf{ac}(\mathsf{X}')=\bot\wedge\exists i.\ G_i\upharpoonright_{\mathsf{ac}_i}\mathsf{X}'=L\\ & \qquad\wedge\forall j\neq i.\ G_j\upharpoonright_{\mathsf{ac}_j}\mathsf{X}'=\mathsf{skip}\end{cases}$$

$$\oplus\{L_i\}_{i\in I}\upharpoonright_{\mathtt{m}'}\mathtt{m} =$$

$$\begin{cases}\bigcup_{i\in I} L_i\upharpoonright_{\mathtt{m}'}\mathtt{m} & \text{if } \mathtt{m}\neq\mathtt{m}'\\ \{\oplus\{L_i\}_{i\in I}\} & \text{if } \mathtt{m}=\mathtt{m}'\wedge\forall i\in I.\ L_i\upharpoonright_{\mathtt{m}'}\mathtt{m}=L_i'\end{cases}$$

$$\&\{\mathsf{X}.\mathtt{m}\langle\varphi_i\rangle; L_i\}_{i\in I}\upharpoonright_{\mathtt{m}'}\mathtt{m} =$$

$$\begin{cases}\bigcup_{i\in I} L_i\upharpoonright_{\mathtt{m}'}\mathtt{m} & \text{if } \mathtt{m}\neq\mathtt{m}'\\ \{\mathsf{Read}\ \mathtt{e}.\&\{\mathsf{X}.\mathtt{m}\langle\varphi_i\rangle; L_i\}_{i\in I}\} & \text{if } \mathtt{m}=\mathtt{m}'\wedge\mathsf{dist}((\varphi_i')_{i\in I})\wedge\\ & \qquad\forall i\in I.\ L_i\upharpoonright_{\mathtt{m}'}\mathtt{m}=L_i'=\mathsf{Read}\ \mathtt{e}.L_i''\end{cases}$$

Fig. 6. Projection Rules for Branching

Additionally to the translation of the branches, it encodes that the choosing process terminates before any process that relies on the communication of its choice via the return value. The rules are as follows:

$$\mathbb{C}\left(\mathsf{X}\{\langle\varphi_i\rangle,(\mathsf{X}_{ij}\langle\varphi_{ij}\rangle)_{j\in J}; G_i\}_{i\in I}\right) =$$
$$\bigvee_{i\in I}\left(\mathbb{C}(G_i)\wedge\exists k.\ \mathsf{firstTerm}(k,\mathsf{X})\wedge\bigwedge_{j\in J}\left(\exists k_j.\ \mathsf{firstTerm}(k_j,\mathsf{X}_{ij})\wedge k\geq k_j\wedge\sigma(h)[k_j]\models\varphi_{ij}\right)\right)$$

$$\mathbb{C}(\oplus\{L_i\}) = \bigvee_i \mathbb{C}(L_i)$$

$$\mathbb{C}(\&\{\mathsf{X}.\mathsf{M}\langle\varphi_i\rangle; L_i\}) = \bigvee_i\left(\exists j\in\mathbb{N}.\ \mathsf{lastTerm}(j,\mathtt{p}.\mathtt{m})\wedge\sigma(j)\models\varphi_i\wedge\mathbb{C}(L_i)\right)$$

In the following we present the rules for branching. The typing rules split the branches into two disjoint sets and shows that the guard of the if statement together with the added choice-conditions of the branch selects the correct continuation of the type. Once the sets of branches are singletons, the choice operators can be removed.

Definition 24 (Typing Rules).

(T-Offer)

$$\frac{\begin{array}{c}I=I_1\cup I_2\qquad I_1\cap I_2=\emptyset\qquad E=E_1\cup E_2\\ \forall i\in I_1.\ \Phi\wedge\mathsf{post}(\mathsf{X}.\mathtt{m},\varphi_i)\Rightarrow e)\\ \forall i\in I_2.\ \Phi\wedge\mathsf{post}(\mathsf{X}.\mathtt{m},\varphi_i)\Rightarrow\neg e)\\ \Phi; e; \bigvee_{i\in I_1}\mathsf{post}(\mathsf{X}.\mathtt{m},\varphi_i), \mathtt{s}\vdash\mathtt{s}'; \mathtt{s}'''::\&\{\mathsf{X}.\mathtt{m}\langle\varphi_i\rangle; L_i\}_{i\in I_1}\rhd E_1\\ \Phi; \neg e; \bigvee_{i\in I_2}\mathsf{post}(\mathsf{X}.\mathtt{m},\varphi_i), \mathtt{s}\vdash\mathtt{s}'; \mathtt{s}'''::\&\{\mathsf{X}.\mathtt{m}\langle\varphi_i\rangle; L_i\}_{i\in I_2}\rhd E_2\end{array}}{\Phi, \mathtt{s}\vdash\mathsf{if}\ e\ \mathsf{then}\ \mathtt{s}'\ \mathsf{else}\ \mathtt{s}''\ \mathsf{fi}\ ; \mathtt{s}'''::\&\{\mathsf{X}.\mathtt{m}\langle\varphi_i\rangle; L_i\}_{i\in I}\rhd E}$$

$$\frac{\Phi, \mathtt{s}' \vdash \mathtt{s} : L \rhd E}{\Phi, \mathtt{s}' \vdash \mathtt{s} : \&\{\mathsf{X}.\mathtt{m} : \varphi; L\} \rhd E}\ \textit{(T-Offer-Single)} \qquad \frac{\Phi, \mathtt{s}' \vdash \mathtt{s} : L \rhd E}{\Phi, \mathtt{s}' \vdash \mathtt{s} : \oplus\{L\} \rhd E}\ \textit{(T-Select-Single)}$$

$$\textit{(T-Select)}\ \frac{I = I_1 \cup I_2 \qquad I_1 \cap I_2 = \emptyset \qquad E = E_1 \cup E_2 \qquad \Phi; e, \mathtt{s} \vdash \mathtt{s}'; \mathtt{s}''' : \oplus\{L_i\}_{i \in I_1} \rhd E_1 \qquad \Phi; \neg e, \mathtt{s} \vdash \mathtt{s}''; \mathtt{s}''' : \oplus\{L_i\}_{i \in I_2} \rhd E_2}{\Phi, \mathtt{s} \vdash \textsf{if } e \textsf{ then } \mathtt{s}' \textsf{ else } \mathtt{s}'' \textsf{ fi } ; \mathtt{s}''' : \oplus\{L_i\}_{i \in I} \rhd E}$$

The extension of the causality graph is described in [29].

We use the following example to illustrate how we handle branching.

Example 5. Consider the scenario: Client X_1 wants to access data on server X_2 and sends its login data by calling method acc. Then X_2 decides. If the login data is invalid, X_2 logs the denied access by calling logging server S and returns -1 to X_1; if the access succeeds, it returns the data, a value > 0, to X_2. X_1 reacts on the return value and returns a boolean indicating whether the access was successful. This is formalized by the following type:

$$\mathbf{main} \to \mathsf{X}_1 : \mathsf{start} \,.\, \mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathsf{acc}.\mathsf{X}_2 \left\{ \begin{array}{l} \langle \mathbf{result} \doteq -1 \rangle\ \mathsf{X}_1 \langle \neg \mathbf{result} \rangle;\ \mathsf{X}_1 \uparrow \mathtt{x}.\mathsf{X}_2 \to \mathsf{S} : \mathsf{log}.\mathsf{End} \\ \langle \mathbf{result} > 0 \rangle\ \mathsf{X}_1 \langle \mathbf{result} \rangle;\ \mathsf{X}_1 \uparrow \mathtt{x}.\mathsf{End} \end{array} \right\}$$

The local type for X_1.start is the following. Note that the Read type is now before the branching.

$$?\mathsf{start}.\mathsf{X}_2 !_{\mathtt{x}} \mathsf{acc}.\mathsf{Read}\ \mathtt{x}.\& \left\{ \begin{array}{l} \mathsf{X}_2.\mathsf{acc}\langle \mathbf{result} \doteq -1 \rangle\ ;\ \mathsf{Put}\ \neg \mathbf{result} \\ \mathsf{X}_2.\mathsf{acc}\langle \mathbf{result} > 0 \rangle\ ;\ \mathsf{Put}\ \mathbf{result} \end{array} \right\}$$

8 Conclusion and Related Work

In this paper we generalize MPST for Active Objects to a two-phase analysis that handles protocols where information is not only transmitted between objects via asynchronous method calls but also inside the object through the heap memory of Active Objects. Additionally, we provide a model-theoretic semantics for MPST, which allows us to give a declarative definition of protocol adherence and integrate further static analyses. These analyses are used to reason about method order and future synchronization within a type system.

8.1 Discussion

Decidability and Types for Validation. The judgment $\vdash \mathsf{Prgm} : \mathbf{G}$ is undecidable if the validity of the FO logic used for specifying side-effects is undecidable. A developer can choose an FOL fragment with decidable validity to trade off expressiveness against analyzability, e.g., if the developer chooses a more restricted fragment, which may limit the expressiveness of the specification, then the validity of the FO logic used for specifying side-effects may become decidable.

When using an undecidable FOL fragment, our approach can be used as a *validation* tool to check whether the implemented (sub-)system will be behaving

as expected. Our approach can be integrated into the development process similarly as invariant-based approaches, and applies techniques proposed by MPST to connect global and local views of concurrent programs, a notoriously difficult problem when using contracts and invariants [15].

Protocol Adherence. Current work on MPST defines protocol adherence as a fidelity theorem, which states that every sequence of interactions in a session follows the scenario declared in MPST [27] as follows: An operational semantics for types is defined and it is shown that the semantics of the language is a refinement of the semantics of the types. Similarly, behavioral contracts [10] define protocol adherence by *compliance*, which compares the interaction of contracts. These are *operational* approaches to specification. We define protocol adherence from a *declarative* perspective by requiring a logical *property* to hold for all traces of a well-typed program. A declarative specification can be analyzed with tools for logical specification, and can enable easier integration of other static analysis tools (e.g., to consider state), because they are only required to have a logical characterization.

8.2 Related Work

This work extends our previous system for Active Objects [31], which could not specify and verify state, required an additional verification step for the scheduler and explicit termination points within the global type.

Actors and Objects. Crafa and Padovani [11, 35] investigate behavioral types for the object-oriented join calculus with typestate, a concurrency model similar to actors. Gay et al. [18] model channels as objects, integrating MPST with classes; Dezani-Ciancaglini et al. [13] use MPST in the object-oriented language `MOOSE`, where types describe communication through shared channels. We ensure deadlock freedom similarly to Giachino et al. [20, 21], who ensure deadlock freedom by inferring behavioral *contracts* and applying a cycle detection algorithm; however, they do not consider protocol adherence.

State and Contracts. Bocchi et al. [5–7] develop a MPST discipline with assertions for endpoint state. The work considers neither objects nor heap memory. The specifications use *global values* in global types and require complex checks for *history-sensitivity* and *temporal-sensitivity* to ensure that an endpoint proves its obligations. We evade this by specifying inherently class-local memory *locations*. They explicitly track values over several endpoints, while we implicitly do so by equations over locations. In a stateless setting, Toninho and Yoshida use dependent MPST [38] to reason about passed data.

Logics. Session types as formulas have been examined by Caires et al. [8] and Carbone et al. [9] for intuitionistic and linear logics as types-as-proposition for the π-calculus. Our work uses logic not for a *proof-theoretic* types-as-proposition theorem, but to use a *model-theoretic* notion of protocol adherence and to integrate static analysis and dynamic logic. Lange and Yoshida [33] also characterize

session types as formulas, but their characterization characterizes the *subtyping* relation, not the execution traces as in our work.

Acknowledgments This work is partially supported by `FormbaR`, part of the Innovation Alliance between TU Darmstadt and Deutsche Bahn AG.

References

1. 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.
2. 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, 2016.
3. D. Ancona, V. Bono, and M. Bravetti. *Behavioral Types in Programming Languages*. Now Publishers Inc., Hanover, MA, USA, 2016.
4. H. G. Baker and C. Hewitt. The incremental garbage collection of processes. *SIGART Newsletter*, 64:55–59, 1977.
5. L. Bocchi, R. Demangeon, and N. Yoshida. A multiparty multi-session logic. In *TGC'12*, volume 8191 of *LNCS*, pages 97–111. Springer, 2012.
6. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
7. 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.
8. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.
9. M. Carbone, S. Lindley, F. Montesi, C. Schürmann, and P. Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR'16*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
10. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5):19:1–19:61, 2009.
11. S. Crafa and L. Padovani. The chemical approach to typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 39(3):13:1–13:45, 2017.
12. F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017.
13. M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, and N. Yoshida. Objects and session types. *Information and Computation*, 207(5):595 – 641, 2009.
14. C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In *CADE*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer, 2015.
15. C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebr. Meth. Program.*, 83(5-6):360–383, 2014.
16. C. C. Din, S. L. T. Tarifa, R. Hähnle, and E. B. Johnsen. History-based specification and verification of scalable concurrent and distributed systems. In *ICFEM*, volume 9407 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2015.
17. A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FMOODS/FORTE*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2013.
18. S. J. Gay, N. Gesbert, A. Ravara, and V. T. Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015.

19. S. J. Gay, V. T. Vasconcelos, P. Wadler, and N. Yoshida. Theory and applications of behavioural types (dagstuhl seminar 17051). *Dagstuhl Reports*, 7(1):158–189, 2017.
20. E. Giachino, L. Henrio, C. Laneve, and V. Mastandrea. Actors may synchronize, safely! In *PPDP*, pages 118–131. ACM, 2016.
21. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in core ABS. *Software and System Modeling*, 15(4):1013–1048, 2016.
22. D. Harel. *First-Order Dynamic Logic*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1979.
23. L. Henkin. Relativization with respect to formulas and its use in proofs of independence. *Compositio Mathematica*, 20:88–106, 1968.
24. L. Henrio, C. Laneve, and V. Mastandrea. Analysis of synchronisations in stateful active objects. In *iFM'17*, volume 10510 of *LNCS*, pages 195–210. Springer, 2017.
25. C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.
26. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.
27. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, Mar. 2016.
28. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO'10*, pages 142–164, 2010.
29. E. Kamburjan. Session Types for ABS. Technical report, TU Darmstadt, 2016. http://formbar.raillab.de/en/techreportsessiontypesabs/.
30. E. Kamburjan and T. Chen. Stateful behavioral types for ABS. *CoRR*, abs/1802.08492, 2018.
31. E. Kamburjan, C. C. Din, and T. Chen. Session-based compositional analysis for actor-based languages using futures. In *ICFEM*, volume 10009 of *Lecture Notes in Computer Science*, pages 296–312, 2016.
32. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
33. J. Lange and N. Yoshida. Characteristic formulae for session types. In *TACAS*, volume 9636 of *Lecture Notes in Computer Science*, pages 833–850. Springer, 2016.
34. Odersky, M et al. Scala programming language. http://www.scala-lang.org.
35. L. Padovani. Deadlock-Free Typestate-Oriented Programming. Submitted to The Art, Science, and Engineering of Programming, preprint available under https://hal.archives-ouvertes.fr/hal-01628801/file/main.pdf, Nov. 2017.
36. P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In *FoVeOOS'10*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.
37. S. Tasharofi, P. Dinges, and R. E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer, 2013.
38. B. Toninho and N. Yoshida. Certifying data in multiparty session types. *J. Log. Algebr. Meth. Program.*, 90:61–83, 2017.

A Full Definitions

Definition 25 (Weakening). *Weakening is defined as*

$$\varphi@\mathsf{X} = \underbrace{\exists\ \mathtt{T}_1\ v_1, \ldots, \exists\ \mathtt{T}_n\ v_n.}_{\{v_1,\ldots,v_n\}=\mathsf{free}(\varphi)} \widehat{\varphi}_\mathsf{X}$$

where the set of all free variables in φ is denoted with $\mathsf{free}(\varphi)$ and $\widehat{\cdot}_\mathsf{X}$ is defined as:

$$\widehat{\exists v.\ \varphi}_\mathsf{X} = \exists v.\ \widehat{\varphi}_\mathsf{X} \quad \widehat{p(t,..,t_n)}_\mathsf{X} = p(\widehat{t}_\mathsf{X},..,\widehat{t_n}_\mathsf{X}) \quad \widehat{\neg\varphi}_\mathsf{X} = \neg\widehat{\varphi}_\mathsf{X} \quad \widehat{\varphi \vee \psi}_\mathsf{X} = \widehat{\varphi}_\mathsf{X} \vee \widehat{\psi}_\mathsf{X}$$

$$\widehat{\mathsf{f}}_\mathsf{X} = \begin{cases} \mathsf{f} & \textit{if } \mathsf{f} \textit{ is a function symbol of } \mathsf{X} \\ v_\mathsf{f} & \textit{otherwise, where } v_\mathsf{f} \textit{ is a fresh logical variable with the type of } \mathsf{f} \end{cases}$$

Example 6 (Weakening). Let fl be a field, X an object and `i` the parameter of some method in class X. Consider $\varphi = \mathsf{X.fl} > 0 \wedge \mathtt{i} > \mathsf{X.fl}$. The formula φ is a X-formula, as $\varphi = \varphi@\mathsf{X}$. The weakening for some object X' is $\varphi@\mathsf{X}' = \exists \mathtt{Int}\ a.a > 0 \wedge \mathtt{i} > a$. The following (valid) X'-formula describes that if $\varphi@\mathsf{X}'$ holds in some state, then after executing `j = i*2;` the program reaches a state, where the variable j contains a positive value:

$$\exists \mathtt{Int}\ a.\ \big(a > 0 \wedge \mathtt{i} > a \Rightarrow [\mathtt{j = i*2;}]\mathtt{j} > 0\big)$$

While X' can not reason about the value of $\mathsf{X.fl}$, the weakening allows to carry over the information that the parameter is larger than 1.

Definition 26 (Relativation). *Let φ be a MSO constraint with a free variable x of type `T` and ψ a MSO constraint. We denote the relativization of ψ with φ by $\psi[x \in Z/\varphi]$. For all quantifiers of type `T` in formula ψ, relativization adds $\varphi(x)$ as restrictions into ψ. The construction is defined with the following rules:*

$$\begin{aligned}
(\exists y \in Z.\psi)[x \in Z/\varphi] &= \exists y \in Z.\varphi(y) \wedge \psi[x \in Z/\varphi] \\
(\exists Y \subseteq Z.\psi)[x \in Z/\varphi] &= \exists Y \subseteq Z.\ \big((\forall y \in Y.\varphi(y)) \wedge \psi[x \in Z/\varphi]\big) \\
(\varphi \wedge \psi)[x \in Z/\varphi] &= \varphi[x \in Z/\varphi] \wedge \psi[x \in Z/\varphi] \\
(\neg\varphi)[x \in Z/\varphi] &= \neg(\varphi[x \in Z/\varphi]) \\
(p(t_1,\ldots,t_n))[x \in Z/\varphi] &= p(t_1[x \in Z/\varphi],\ldots,t_n[x \in Z/\varphi]) \\
(f(t_1,\ldots,t_n))[x \in Z/\varphi] &= f(t_1[x \in Z/\varphi],\ldots,t_n[x \in Z/\varphi])
\end{aligned}$$

Example 7 (Relativation). Consider a graph (V, E, c) with one predicate c over its nodes. I.e., at every node n, the predicate $\mathsf{c}(n)$ either holds or not. The formula $\psi = \forall n \in V.\ \mathsf{c}(n)$ expresses that c holds everywhere: The following formula expresses that x has an out-degree of at most 1:

$$\varphi(x) = \forall y, z \in V.\ E(x,y) \wedge E(x,z) \Rightarrow z \doteq y$$

The following formula restricts ψ on the subgraph described by φ, i.e. it expresses that at all nodes with an out-degree of at most 1, c holds:

$$\psi[x \in V \setminus \varphi] = \forall n \in V.\ \big(\forall y, z \in V.\ E(n,y) \wedge E(n,z) \Rightarrow z \doteq y\big) \Rightarrow \mathsf{c}(n)$$

Auxiliary Predicates

$$\mathsf{res}(i) = \exists \mathsf{A}.\ \exists f.\ \exists \mathsf{M}.\ \exists e \in D.\ ev(i) \doteq \mathsf{fREv}(\mathsf{A}, f, e)$$

Projection on Objects

$$\mathbf{main} \to \mathsf{X}_2 : \mathtt{m}\langle\varphi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X}_1 = ?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) \text{ if } \mathsf{ac} = \mathsf{ac}_\perp \wedge \mathsf{X}_2 = \mathsf{X}_1$$

$$\mathbf{main} \to \mathsf{X}_2 : \mathtt{m}\langle\varphi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X}_1 = \mathsf{skip}.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) \text{ if } \mathsf{ac} = \mathsf{ac}_\perp \wedge \mathsf{X}_2 \neq \mathsf{X}_1$$

$$\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi, \psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X}_1 = \mathsf{X}_2 ! \mathtt{x}\, \mathtt{m}\langle\varphi\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) \text{ if } \mathsf{ac}(\mathsf{X}_1) \neq \perp \wedge \varphi = \varphi@\mathsf{X}_1$$

$$\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi, \psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X}_2 = \begin{cases} ?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) & \text{if } \mathsf{ac}(\mathsf{X}_2) = \perp \\ \mathsf{Put}\ \mathsf{ac}(\mathsf{X}_2).?\mathtt{m}\langle\varphi@\mathsf{X}_2\rangle.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}_1) & \text{if } \mathsf{ac}(\mathsf{X}_2) \neq \perp \end{cases}$$

$$\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi, \psi\rangle.G \upharpoonright_{\mathsf{ac}} \mathsf{X} = \mathsf{skip}.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}) \text{ if } \mathsf{X}_2 \neq \mathsf{X} \neq \mathsf{X}_1$$

$$(\mathsf{X}_1 \uparrow \mathsf{e}.G') \upharpoonright_{\mathsf{ac}} \mathsf{X} = \begin{cases} \mathsf{Read}\ \mathsf{e}.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}) & \text{if } \mathsf{ac}(\mathsf{X}_1) \neq \perp \wedge \mathsf{X}_1 = \mathsf{X} \\ \mathsf{skip}.(G \upharpoonright_{\mathsf{ac}[\mathsf{X}_2 \mapsto \psi]} \mathsf{X}) & \text{if } \mathsf{X}_1 \neq \mathsf{X} \end{cases}$$

$$\mathsf{End} \upharpoonright_{\mathsf{ac}} \mathsf{X} = \begin{cases} \mathsf{Put}\ \mathsf{ac}(\mathsf{X}).\mathsf{End} & \text{if } \mathsf{ac}(\mathsf{X}) \neq \perp \\ \mathsf{End} & \text{if } \mathsf{ac}(\mathsf{X}) = \perp \end{cases}$$

$$\big((G)^*_\varphi.G'\big) \upharpoonright_{\mathsf{ac}} \mathsf{X} =$$

$$\begin{cases} \mathsf{Put}\ \mathsf{ac}(\mathsf{X}).(L'')^*_{\varphi@\mathsf{X}}.L' & \text{if } L \neq \mathsf{skip} \wedge rcv(\mathsf{X}, G) \wedge \mathsf{ac}(\mathsf{X}) \neq \perp \wedge cs(\varphi) \\ (L)^*_{\varphi@\mathsf{X}}.L' & \text{if } L \neq \mathsf{skip} \wedge \neg rcv(\mathsf{X}, G) \wedge \mathsf{ac}(\mathsf{X}) \neq \perp \wedge cs(\varphi) \\ L' & \text{if } L = \mathsf{skip} \wedge cs(\varphi) \end{cases}$$

Where $G.\mathsf{End} \upharpoonright_{\mathsf{ac}[\mathsf{X} \mapsto \perp]} \mathsf{X} = L'', G \upharpoonright_{\mathsf{ac}} \mathsf{X} = L$ and $G' \upharpoonright_{\mathsf{ac}} \mathsf{X} = L'$

$$cs(\varphi) = (\bigwedge_{\mathsf{X} \in objects(\varphi)} \varphi@\mathsf{X}) \to \varphi$$

Projection on Methods

$$?\mathtt{m}\langle\varphi\rangle \upharpoonright_{\mathtt{m}'} \mathtt{m} = \begin{cases} \{(?\mathtt{m}\langle\varphi\rangle, \mathtt{m})\} & \text{if } \mathtt{m}' = \perp \\ \{(\mathsf{skip}, \mathtt{m}')\} & \text{otherwise} \end{cases} \qquad \mathsf{Put}\ \varphi \upharpoonright_{\mathtt{m}'} \mathtt{m} = \begin{cases} \{(\mathsf{Put}\ \varphi, \perp)\} & \text{if } \mathtt{m} = \mathtt{m}' \\ \{(\mathsf{skip}, \perp)\} & \text{otherwise} \end{cases}$$

$$\mathsf{Read}\ e \upharpoonright_{\mathtt{m}'} \mathtt{m} = \begin{cases} \{(\mathsf{Read}\ e, \mathtt{m}')\} & \text{if } \mathtt{m} = \mathtt{m}' \\ \{(\mathsf{skip}, \mathtt{m}')\} & \text{otherwise} \end{cases}$$

$$\mathsf{X}_1 ! _v \mathtt{m}''\langle\varphi\rangle \upharpoonright_{\mathtt{m}'} \mathtt{m} = \begin{cases} \{(\mathsf{X}_1 ! _v \mathtt{m}''\langle\varphi\rangle, \mathtt{m}')\} & \text{if } \mathtt{m} = \mathtt{m}' \\ \{(\mathsf{skip}, \mathtt{m}')\} & \text{otherwise} \end{cases}$$

$$\mathsf{skip} \upharpoonright_{\mathtt{m}'} \mathtt{m} = \{(\mathsf{skip}, \mathtt{m}')\} \qquad \mathsf{End} \upharpoonright_{\mathtt{m}'} \mathtt{m} = \{(\mathsf{skip}, \mathtt{m}')\} \text{if } \mathtt{m}' = \perp$$

$$(L_1.L_2) \upharpoonright_{\mathtt{m}'} \mathtt{m} = \begin{array}{l} \{(L_1'.L_2, \mathtt{m}''') \mid (L_1', \mathtt{m}'') \in L_1 \upharpoonright_{\mathtt{m}'} \wedge \mathtt{m}'' \neq \perp \wedge (L_2', \mathtt{m}''') \in L_2 \upharpoonright_{\mathtt{m}''}\} \\ \cup \{(L_2', \mathtt{m}''') \mid (L_1', \perp) \in L_1 \upharpoonright_{\mathtt{m}'} \wedge (L_2', \mathtt{m}''') \in L_2 \upharpoonright_{\perp}\} \end{array}$$

$$(L)^*_\varphi \upharpoonright_{\mathtt{m}''} \mathtt{m}' = \begin{cases} \big\{\big((L \upharpoonright_{\mathtt{m}''} \mathtt{m}')^*_\varphi, \mathtt{m}''\big)\big\} & \text{if } \mathtt{m}'' = \mathtt{m}' \wedge L \upharpoonright_{\mathtt{m}''} \mathtt{m}' \neq \{(\mathsf{skip}, \mathtt{m}''')\} \\ L \upharpoonright_{(methodname'' \mathtt{m}'} & \text{if } \mathtt{m}'' \neq \mathtt{m}' \wedge L \upharpoonright_{\mathtt{m}''} \mathtt{m}' \neq \{(\mathsf{skip}, \mathtt{m}''')\} \\ \{(\mathsf{skip}, \mathtt{m}'')\} & L \upharpoonright_{\mathtt{m}''} \mathtt{m}' = \{(\mathsf{skip}, \mathtt{m}''')\} \end{cases}$$

Translation into Constraints

$$\mathbb{C}(\mathbf{main} \to \mathsf{X}_2 : \mathtt{m}\langle\psi\rangle.G) \;=\; \exists j,k.\ \exists f.\ \exists \mathsf{e}'.\ \mathsf{ev}(j) \doteq \mathsf{iREv}(\mathsf{X}_2, f, \mathtt{m}) \wedge \mathsf{C}(j) \models \varphi@\mathsf{X}_2 \wedge$$
$$\mathsf{ev}(k) \doteq \mathsf{fEv}(\mathsf{X}_2, f, \mathsf{e}') \wedge \mathsf{C}(k) \models \psi \wedge \forall l.\, l \neq j \wedge l \neq k \Rightarrow \mathsf{res}(l) \wedge \mathbb{C}(G)$$

$$\mathbb{C}(\mathsf{X}_1 \xrightarrow{\mathtt{x}} \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle) \;=\; \exists i,j,k.\ \exists f.\ \exists \mathsf{e},\mathsf{e}'.$$
$$\mathsf{ev}(i) \doteq \mathsf{iEv}(\mathsf{X}_1, \mathsf{X}_2, f, \mathtt{m}, \mathsf{e}) \wedge \mathsf{C}(i) \models \varphi \wedge \mathsf{ev}(j) \doteq \mathsf{iREv}(\mathsf{X}_2, f, \mathtt{m}) \wedge \mathsf{C}(j) \models \varphi@\mathsf{X}_2 \wedge$$
$$\mathsf{ev}(k) \doteq \mathsf{fEv}(\mathsf{X}_2, f, \mathsf{e}') \wedge \mathsf{C}(k) \models \psi \wedge \mathsf{C}(i) \models \mathsf{X}_1.\mathtt{x} \doteq f \wedge \forall l.\ l \neq i \wedge l \neq j \wedge l \neq k \Rightarrow \mathsf{res}(l)$$

$$\mathbb{C}(\mathsf{X}_1 \to \mathsf{X}_2 : \mathtt{m}\langle\varphi,\psi\rangle) \;=\; \exists i,j,k.\ \exists f.\ \exists \mathsf{e},\mathsf{e}'.$$
$$\mathsf{ev}(i) \doteq \mathsf{iEv}(\mathsf{X}_1, \mathsf{X}_2, f, \mathtt{m}, \mathsf{e}) \wedge \mathsf{C}(i) \models \varphi \wedge \mathsf{ev}(j) \doteq \mathsf{iREv}(\mathsf{X}_2, f, \mathtt{m}) \wedge \mathsf{C}(j) \models \varphi@\mathsf{X}_2 \wedge$$
$$\mathsf{ev}(k) \doteq \mathsf{fEv}(\mathsf{X}_2, f, \mathsf{e}') \wedge \mathsf{C}(k) \models \psi \wedge \wedge \forall l.\ l \neq i \wedge l \neq j \wedge l \neq k \Rightarrow \mathsf{res}(l)$$

$$\mathbb{C}(\mathsf{X}{\uparrow}\mathsf{e}) \;=\; \exists i.\ \exists f.\ \exists \mathsf{e}'.\ \exists \mathsf{X}'.\ \mathsf{ev}(i) \doteq \mathsf{fREv}(\mathsf{X}, \mathsf{X}', f, \mathsf{e}') \wedge \mathsf{C}(i) \models \mathsf{e} \doteq f \wedge \forall l.\ l \neq i \Rightarrow \mathsf{res}(l)$$

$$\mathbb{C}(G_1.G_2) = \bigwedge_{\mathsf{X}} \big(\exists i \in \mathbb{N}.\ \mathbb{C}(G_1)[j \in \mathbb{N}/\mathsf{A}(j,\mathsf{X}) \Rightarrow j < i] \wedge \mathbb{C}(G_2)[j \in \mathbb{N}/\mathsf{A}(j,\mathsf{X}) \Rightarrow j \geq i]\big)$$

$$\mathbb{C}\big((G_\varphi^*)\big) \;=\; \exists X \subseteq \mathbb{N}.\ \exists i,j \in X.\ \big(\forall k \in \mathbb{N}.\ i < k \leq j\big) \wedge \forall i \in X.\ \mathsf{C}(i) \models \varphi \wedge$$
$$\forall i,j \in X.\ \Big(\big(\forall k \in X.\ k \geq j \vee k \leq i\big) \Rightarrow \big(\mathbb{C}(G)\big)[n \in \mathbb{N}/i < n \leq j]\Big)$$

$$\mathbb{C}(\mathsf{End}) = \mathbf{true}$$

The local translation for some object X is:

$$\mathbb{C}(L_1\ .\ L_2) \;=\; \exists i \in \mathbb{N}.\ \mathbb{C}(L_1)[n \in \mathbb{N}/n < i] \wedge \mathbb{C}(L_2)[n \in \mathbb{N}/n \geq i]$$
$$\mathbb{C}(?\mathtt{m}\langle\varphi\rangle) = \exists i.\ \forall j.\ i = j \wedge \exists f.\ \mathsf{iREv}(\mathsf{X}, f, \mathtt{m}) \wedge \mathsf{C}(i) \models \varphi@\mathsf{X}$$
$$\mathbb{C}(\mathsf{X}'!_{\mathtt{x}}\mathtt{m}\langle\varphi\rangle) = \exists i.\ \forall j.\ i = j \wedge \exists f, \mathsf{e}.\ \mathsf{iEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \mathsf{e}) \wedge \mathsf{C}(i) \models \varphi@\mathsf{X} \wedge \mathsf{C}(i) \models (\mathtt{x} \doteq f)$$
$$\mathbb{C}(\mathsf{X}'!\mathtt{m}\langle\varphi\rangle) = \exists i.\ \forall j.\ i = j \wedge \exists f, \mathsf{e}.\ \mathsf{iEv}(\mathsf{X}, \mathsf{X}', f, \mathtt{m}, \mathsf{e}) \wedge \mathsf{C}(i) \models \varphi@\mathsf{X}$$
$$\mathbb{C}(\mathsf{Put}\ \varphi) = \exists i.\ \forall j.\ i = j \wedge \exists f, \mathsf{e}.\ \mathsf{fEv}(\mathsf{X}, f, \mathsf{e}) \wedge \mathsf{C}(i) \models \varphi@\mathsf{X}$$
$$\mathbb{C}(\mathsf{Read}\ \mathsf{e}) = \exists i.\ \forall j.\ i = j \wedge \exists f.\ \mathsf{fREv}(\mathsf{X}, \mathsf{X}', f, \mathsf{e}') \wedge \mathsf{e} = f$$
$$\mathbb{C}(L_\varphi^*) \;=\; \exists X \subseteq \mathbb{N}.\ \exists i,j \in X.\ \big(\forall k \in \mathbb{N}.\ i < k \leq j\big) \wedge \forall i \in X.\ \mathsf{C}(i) \models \varphi \wedge$$
$$\forall i,j \in X.\ \big(\forall k \in X.\ k \geq j \vee k \leq i\big) \to \big(\mathbb{C}(L)\big)[x \in \mathbb{N}/i < x \leq j]$$
$$\mathbb{C}(\mathsf{End}) = \mathbf{true}$$

Typing Rules

$$\textsf{(T-Main)}\ \frac{\begin{array}{c}\mathsf{O}_i = \mathtt{object}\ \mathsf{X}_i\{\ldots\} \quad \mathit{Roles}(\mathbf{G}) = \{\mathsf{X}_1,\ldots,\mathsf{X}_n\} \qquad \mathbb{G}(\mathbf{G}) + \bigcup_{i\leq n} E_i \text{ admissible} \\ \exists j \leq n.\ \mathbf{G} = \mathtt{main} \to \mathsf{X}_j : \mathtt{m}\langle\varphi\rangle.G \quad \forall i \leq n.\ \mathsf{Post}(\mathbf{G}) \vdash \mathsf{O}_i : \mathsf{prp}^*(\mathbf{G} \upharpoonright \mathsf{X}_i) \rhd E_i \end{array}}{\begin{array}{c}\vdash \mathsf{O}_1 \quad \ldots \quad \mathsf{O}_n \quad \mathtt{main}\{\mathsf{X}_j!\mathtt{m}()\} : \mathbf{G} \\ \forall i \leq n.\ \mathbf{L} \upharpoonright_{\mathsf{ac}} \mathtt{m}_i = ?\mathtt{m}_i\langle\varphi_i\rangle.L_i \end{array}}$$

$$\textsf{(T-Object)}\ \frac{\begin{array}{c}\forall i \leq n.\ \Phi, \varphi_i, \mathbf{skip} \vdash \mathtt{s}_i : L_i \rhd E_i \qquad E = \bigcup_{i\leq n} E_i \end{array}}{\Phi \vdash \mathtt{object}\ \mathsf{X}\{\mathtt{T}_1\ \mathtt{m}_1(\overline{\mathtt{T}\ \mathtt{x}})\{\mathtt{s}_1\}\ \ldots\ \mathtt{T}_n\ \mathtt{m}_n(\overline{\mathtt{T}\ \mathtt{x}})\{\mathtt{s}_n\}\quad \overline{\mathtt{T}\ \mathtt{x} = \mathsf{e}}\} : \mathbf{L} \rhd E}$$

$$\textsf{(T-Return)}\ \frac{\Phi \Rightarrow [\mathtt{s}; \mathbf{return}\ \mathsf{e}]\varphi}{\Phi, \mathtt{s} \vdash \mathbf{return}\ \mathsf{e} : \mathsf{Put}\ \varphi \rhd E} \qquad \textsf{(T-Call)}\ \frac{\begin{array}{c}\Phi, \mathtt{s}; \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\bar{\mathsf{e}}) \vdash \mathtt{s}' : L \rhd E \\ \Phi \Rightarrow [\mathtt{s}; \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\bar{\mathsf{e}})]\varphi \end{array}}{\Phi, \mathtt{s} \vdash \mathtt{T}\ \mathtt{x} = \mathsf{X}!\mathtt{m}(\bar{\mathsf{e}}); \mathtt{s}' : \mathsf{X}!_{\mathtt{x}}\mathtt{m}\langle\varphi\rangle.L \rhd E}$$

$$\textsf{(T-Call-2)}\ \frac{\Phi, \mathsf{s}; \mathsf{X!m}(\bar{\mathsf{e}}) \vdash \mathsf{s}' : L \rhd E \qquad \Phi \Rightarrow [\mathsf{s}; \mathsf{X!m}(\bar{\mathsf{e}})]\varphi}{\Phi, \mathsf{s} \vdash \mathsf{X!m}(\bar{\mathsf{e}}); \mathsf{s}' : \mathsf{X!m}\langle\varphi\rangle.L \rhd E} \qquad \textsf{(T-Assign)}\ \frac{\Phi, \mathsf{s}; \mathtt{T\ x} = \mathsf{e} \vdash \mathsf{s}' : L \rhd E'}{\Phi, \mathsf{s} \vdash \mathtt{T\ x} = \mathsf{e}; \mathsf{s}' : L \rhd E}$$

$$\textsf{(T-Get)}\ \frac{\Phi, \mathsf{s}; \mathtt{T\ x} = \mathsf{e}.\textbf{get} \vdash \mathsf{s}' : L \rhd E' \qquad E = E' \cup \{(n, n') | \exists \mathtt{m} \in \mathsf{p2(e)}.\ n \in \mathsf{term}(\mathtt{m}) \wedge n' \in \mathsf{node}(\mathsf{s}; \mathsf{e}.\textbf{get})\}}{\Phi, \mathsf{s} \vdash \mathtt{T\ x} = \mathsf{e}.\textbf{get}; \mathsf{s}' : \mathsf{Read\ e}.L \rhd E}$$

$$\textsf{(T-While)}\ \frac{\varphi \wedge \mathsf{Post}(\mathbf{G}), \textbf{skip} \vdash \mathsf{s}' : L' \rhd E'' \qquad \Phi \Rightarrow [\mathsf{s}'']\varphi \qquad \varphi \wedge \mathsf{Post}(\mathbf{G}) \Rightarrow [\mathsf{s}]\varphi \qquad \varphi \wedge \mathsf{Post}(\mathbf{G}), \textbf{skip} \vdash \mathsf{s} : L \rhd E' \qquad E = E' \cup E''}{\Phi, \mathsf{s}'' \vdash \textbf{while}\ e\ \{\mathsf{s}\}; \mathsf{s}' : (L)^*_\varphi\ .\ L' \rhd E}$$

B Soundness

The proof for Theorem 1 is similar to the proof for Theorem 2 in [29], we thus only give a sketch and point out where the proofs differ.

Propagation

First, we state the correctness of propagation. Let $\mathbf{tr} \restriction_{\mathsf{X}}$ be the projection of trace $\mathbf{tr}$ on X, i.e., $\mathbf{tr} \restriction_{\mathsf{X}}$ results from $\mathbf{tr}$ by replacing all events not issued by X.

Lemma 1. *Let* Prgm *be a program and* $\mathbf{G}$ *a type for* Prgm*. If in all traces produced by* Prgm*, the order of invocation events is the same, then every trace that satisfies the translation of the propagated type iff it satisfies the translation of the original type:*

$$\vdash \mathsf{Prgm} : \mathbf{G} \to \forall \mathbf{tr}.\ \forall \mathsf{X}.\ \mathsf{Prgm} \Downarrow \mathbf{tr} \to \left(\mathbf{tr} \restriction_{\mathsf{X}} \models \mathbb{C}(\mathsf{prp}^*(\mathbf{G} \restriction \mathsf{X})) \leftrightarrow \mathbf{tr} \restriction_{\mathsf{X}} \models \mathbb{C}(\mathbf{G} \restriction \mathsf{X})\right)$$

Proof. We fix X and denote $\mathbf{G} \restriction \mathsf{X}$ with $\mathbf{L}$. We show this by induction on the number n of applications of prp for the fixpoint.

Induction Base, $n = 0$ Then $\mathsf{prp}^*(\mathbf{L}) = \mathbf{L}$ and the lemma holds trivially.

Induction Step, $n = n' + 1$ By induction hypothesis there is a type $\mathbf{L}' = \mathsf{prp}^{n'}(\mathbf{L})$ such that the desired property holds. We make a case distinction on the applied case in the definition of prp in its last application:

- **Case 1** $\mathsf{Put}\ \varphi.?\mathtt{m}\langle\psi\rangle \rightsquigarrow \mathsf{Put}\ \varphi.?\mathtt{m}\langle\psi \wedge \varphi@\mathsf{X}\rangle$

  In this case we have to show that the start of execution of method $\mathtt{m}$ the formula ψ as holds. Let $\mathtt{m}'$ be the method whose termination action $\mathsf{Put}\ \varphi$ is responsible for φ. By assumption, the order of invocation events is fixed and the program can be typed. Thus, there is no trace such that between the invocation action of $\mathtt{m}'$ and $\mathtt{m}$, there is another invocation event. Thus, each trace $\mathbf{tr}$ that contains pairs of the form $(\mathsf{iREv}(\mathsf{X}, f, \mathtt{m}), \mathsf{C})$, for some f, C s.t. $\mathsf{C} \models \psi$ contains this pair as part of a subtrace of the following form:

  $$\left[\left(\mathsf{fEv}(\mathsf{X}, f', \mathtt{m}', \mathsf{e}'), \mathsf{C}'\right), \left(\mathsf{iREv}(\mathsf{X}, f, \mathtt{m}), \mathsf{C}\right)\right]$$

  for some f', C s.t. $\mathsf{C} \models \varphi$. Every state change on X must be executed by some process on X, but as there is no such such process (as there would be an

invocation event for it) between the two events in the subtrace, the state-part of φ, i.e. $\varphi@\mathsf{X}$, still holds at the invocation event: $\mathsf{C} \models \varphi@\mathsf{X}$. This is exactly the condition captured by this propagation case.

- **Case 2** $\mathsf{Put}\ \varphi\ .\ (L)^*_\psi \rightsquigarrow \mathsf{Put}\ \varphi \wedge \psi\ .\ (L)^*_\psi$
  By the definition of $\mathbb{C}((L)^*)$ there is a set of indices X in every trace, such that every such position $i \in X$, the invariant holds and for every pair of consecutive positions $i, j\ \in X$, the subtrace $\mathbf{tr}[i..j]$ satisfies $\mathbb{C}(L)$. Now, $\mathsf{Put}\ \varphi$ is the last event before the repetition, thus before the very first position $i_0 \in X$ there is a pair
  $$\mathbf{tr}[i_0 - 1] = \big(\mathsf{fEv}(\mathsf{X}, f, \mathtt{m}, \mathsf{e}), \mathsf{C}\big)$$
  In C, no process is active at X. We only regard traces produced Prgm, thus $\mathbf{tr}$ is well-formed[1] and i_0 must be a invocation reaction event
  $$\mathbf{tr}[i_0] = \big(\mathsf{iREv}(\mathsf{X}, f', \mathtt{m}'), \mathsf{C}\big)$$
  Such that $\mathsf{C} \models \psi$. With the same argument as above, the condition at ψ must hold at $i_0 - 1$, as there was no process who could have changed it. Note, that $i_0 \neq 0$ as every local type starts with a receiving action, not a repetition.
- **Case 3** $(L)^*_\psi\ .\ ?\mathtt{m}'\langle\varphi\rangle \rightsquigarrow (L)^*_\psi\ .\ ?\mathtt{m}'\langle\varphi \wedge \psi\rangle$
  This case is analogous to case 2.
- **Case 4** $(L)^*_\varphi\ .\ (L)^*_\psi \rightsquigarrow (L)^*_{\varphi\wedge\psi}\ .\ (L)^*_\psi$
  This case is analogous to case 2.
- **Case 5** $(?\mathtt{m}'\langle\varphi\rangle\ .\ L\ .\ \mathsf{Put}\ \varphi')^*_\psi \rightsquigarrow (?\mathtt{m}'\langle\varphi \wedge \psi\rangle\ .\ L\ .\ \mathsf{Put}\ \varphi' \wedge \psi)^*_\psi$
  By the definition of $\mathbb{C}((L)^*)$, there is a set of indices X in every trace, such that every such position $i \in X$, the invariant holds and for every pair of consecutive positions $i, j\ \in X$, If the repetition start with a receiving action and ends with a termination, the chosen positions are those of the termination actions and the first action before (Again, the syntactic form guarantees such a position). This reduces this case to show that the same propagation as in case 1 holds and thus technical details are analogous to case 1. □

Main Theorem

Given a well-formed global type $\mathbf{G}$ we can say that another (possibly not well-formed) global $\mathbf{G}'$ is a prefix of $\mathbf{G}$ if we can extend $\mathbf{G}'$ to $\mathbf{G}$ by concatenating another global type:
$$\mathbf{G}' \sqsubseteq \mathbf{G} \iff \mathbf{G} = \mathbf{G}'.G$$
And similarly for local types $\mathbf{L}$ and traces $\mathbf{tr}$.

The main lemma is similar to subject reduction in non-model-theoretic semantics for types, as it connects types and operational semantics of the language. It states that each step in the execution preserves the property that the trace so far is a prefix of a a trace which is a model for the type.

Lemma 2. *Let* Prgm *be a program and* $\mathbf{G}$ *a well-formed type with* $\vdash \mathsf{Prgm} : \mathbf{G}$. *Every prefix of every trace of* Prgm *satisfies the translation of a prefix of* $\mathbf{G}$*:*

$$\vdash \mathsf{Prgm} : \mathbf{G} \to \forall \mathbf{tr}.\ \mathsf{Prgm} \Downarrow \mathbf{tr} \to \Big(\forall \mathbf{tr}'.\ \mathbf{tr}' \sqsubseteq \mathbf{tr} \to \big(\exists \mathbf{G}'.\ \mathbf{G}' \sqsubseteq \mathbf{G} \wedge \mathbf{tr}' \models \mathbb{C}(\mathbf{G}')\big)\Big)$$

[1] The well-formedness of traces is defined in [15, 29]

The property that the whole execution of the program satisfies the translation of the whole type and not some prefix follows from well-formedness of global types (Theorem 1 in [29]) and deadlock freedom. The main differences to the proof in [29] are the following:

All assumed conditions hold at the point they are used We distinguish between the following kinds of assumed conditions:

- The precondition at method start. The precondition is a conjunction $\varphi_1 \wedge \varphi_2$ where φ_1 is resulting from the projection and φ_2 from the propagation. That φ_2 holds follows from Lemma 1. That φ_1 holds follows from the fact that the precondition is projected from a formula ψ in the global call, which is fully proven by the caller, checked in rule (T-Call) with the condition that ψ is equal to its projection on the caller.
- The selection condition of the passive choice. It must connect that the additional condition executes the statement branch which is typed with the corresponding type branch. This follows directly from the two additional promises of (T-Offer) with respect to (T-Select).

Methods are executed in the right order Assume there is a method $\mathtt{m}_1$ that is executed on some object X before $\mathtt{m}_2$ in the type, but executed the other way around in the generated trace. Then $\mathtt{m}_2$ does not depend on $\mathtt{m}_1$. But by assumption the program has been typed. Rule (T-Main) checks, however, that the start of $\mathtt{m}_1$ causes $\mathtt{m}_2$ in its admissibility check and this means that $\mathtt{m}_2$ cannot be executed before $\mathtt{m}_1$ (Lemma 36 in [29]).

Deadlock Freedom A deadlocked configuration is a configuration which is not terminated, yet cannot continue execution. First we observe that every deadlock is caused by processes blocking at **get** statements. It cannot be a single process, because a process has no access on its own future. It can also not be stored in the heap between call and execution start, as this would mean that another method was active to store it and this would violate the condition that all methods are executed in the right order shown above.

Assume there would be a deadlock. W.l.o.g. we assume that only two processes are involved, p_1 executing $\mathtt{m}_1$ and p_2 executing $\mathtt{m}_2$. If p_1 blocks while attempting to read a future belonging to $\mathtt{m}_2$ then the Points-To analysis will include $\mathtt{m}_2$ in the set of possible method in rule (T-Get). As $\mathtt{m}_1$ has been type checked, this mean that an edge from the corresponding termination to the corresponding read would be added to the causality graph. The same holds for $\mathtt{m}_2$. The termination of $\mathtt{m}_i$ is in the same object type as the reading type, thus there is a path from the read in $\mathtt{m}_i$ to the termination in $\mathtt{m}_i$. The resulting graph is pictured below and contains a cycle. The absence of cycles is however checked in rule (T-Main). For a full formalization of deadlocks through causality graphs, we refer to [17].