Modular Verification of a Modular Specification: Behavioral Types as Program Logics

Modulare Verifikation einer modularer Spezifikation: Behavioral Types als Programmlogiken
Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von Eduard Kamburjan, M.Sc. aus Alma-Ata, Kasachstan
Tag der Einreichung: 18.02.2020, Tag der Prüfung: 31.03.2020
Darmstadt — D 17

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Prof. Dr. Frank de Boer

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Software Engineering

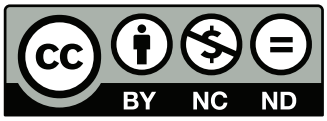
Modular Verification of a Modular Specification: Behavioral Types as Program Logics
Modulare Verifikation einer modularer Spezifikation: Behavioral Types als Programmlogiken

Genehmigte Dissertation von Eduard Kamburjan, M.Sc. aus Alma-Ata, Kasachstan

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Prof. Dr. Frank de Boer

Tag der Einreichung: 18.02.2020
Tag der Prüfung: 31.03.2020

Darmstadt — D 17



The Alexandrians knew of course,
that this was all mere words, all theatre.
But the day was warm and poetic,
the sky a pale blue.

— C. P. Cavafy
Alexandrian Kings

Abstract

Verification of distributed systems is a challenging problem, especially if the distributed system allows entities to interact in multiple ways. The complexity of the task is mirrored in the complexity of the verification system. The design of such verification systems itself is a challenge on the theoretical level. The main engineering tool to handle complex systems is modularization. While modularity in the target system is exploited by verification systems, it is rarely applied to their theory.

This thesis is a study of modularity in deductive verification with program logics and the main tool we introduce is the *Behavioral Program Logic* (BPL). As concurrency model we use Active Objects, because, for one, they allow multiple ways for entities to interact, for another, because their strong encapsulation enables modularization on the semantic level. Modularity is applied on multiple layers:

- We present a modular, new Locally Abstract, Globally Concrete semantics for Active Objects, that allows us to sharply distinguish between local and global behavior.
- We introduce a modular specification principle with object invariants, asynchronous method contracts, Effect Types and Session Types. Each of these specifications itself is modular and describes the behavior of an encapsulated entity. E.g., object invariants describe single objects, method contracts describe single methods, Session Types describe single methods in a protocol context.
- We design a modular verification system with BPL. BPL allows one to design one logic per verification aspect, such as presence of object invariants, as well as interfacing with external static analyses, such as pointer analyses. BPL enables us not only the simple design of new program logics, it also allows *reuse* on the rule level: The calculi of all aspects are composed before being applied. For composition, we give a mostly automatic soundness argument for the composed rule.

  BPL has structural similarities with behavioral types, which simplify calculus design — the most prominent is that the calculus modifies program and specification in one step, after syntactically matching them. This is uncommon in other dynamic logics.

The lack of modularity in verification of distributed systems and the need for a more structural approach in the design of program logics are identified as shortcomings of state-of-the-art approaches by applying existing tools to the `FormbaR` model of railway operations, the biggest verification study with Active Objects to date. This thesis concludes with a discussion that exemplifies how our approach can express and verify properties that were not possible before.

Zusammenfassung

Verifikation verteilter Systeme birgt viele Herausforderungen, insbesondere wenn das verteilte System mehrere Möglichkeiten der Interaktion zwischen Entitäten ermöglicht. Die Komplexität dieser Aufgabe spiegelt sich in der Komplexität des Verifikationssystems wider; der Entwurf solcher Verifikationssysteme ist selbst ein nicht minder herausforderndes Problem aus theoretischer Sicht. Das Hauptwerkzeug des Ingenieurwesens zum Entwurf komplexer Systeme ist Modularisierung. Modularität im Zielsystem wird von Verifikationssystemen zwar ausgenutzt, die Theorie der Verifikation ist aber selten selbst modular.

Diese Dissertation ist eine Studie über Modularität in deduktiver Verifikation mit Programmlogiken. Der Hauptbeitrag zu diesem Zwecke ist die *Behavioral Program Logic* (BPL). Als Nebenläufigkeitsmodell verwenden wir Active Objects, die auf der einen Seite mehrere Möglichkeiten der Interaktion zwischen Entitäten zulässt und auf der anderen Seite durch ihre strikte Kapselung bereits auf Semantikebene Modularität ermöglichen. Modularität wird auf mehreren Ebenen benutzt:

- Wir entwerfen eine modulare, neue Locally Abstract, Globally Concrete Semantik für Active Objects, welche lokales Verhalten von globalem Verhalten abkapselt.
- Wir entwerfen ein modulares Spezifikationsprinzip mit Objektinvarianten, asynchronen Methodverträgen, Effect Types und Session Types. Jede dieser Spezifikationen ist selbst modular und beschreibt das Verhalten einer abgekapselten Entität. Beispielsweise beschreiben Objektinvarianten einzelne Objekte, Methodverträge einzelne Methoden und Session Types einzelne Methoden im Kontext eines Protokolls.
- Wir entwerfen ein modulares Verifikationssystem mit BPL. BPL erlaubt es eine Programmlogik pro Verifikationsaspekt, zum Beispiel Objektinvarianten, zu entwerfen. BPL ermöglicht auch die Interaktion mit externen, statischen Analysen, z.B., Zeigeranalysen. BPL ermöglicht nicht nur den einfachen Entwurf neuer Programmlogiken, sondert erlaubt es Teile von *Regeln wiederzuverwenden*: die Kalküle aller Aspekte werden vor ihrer Anwendung zusammengesetzt. Basierend auf den Teilregeln konstruieren wir ein Korrektheitsargument für die zusammengesetzte Regel.

  BPL hat strukturelle Ähnlichkeiten zu Behavioral Types, was den Entwurf von Kalkülen erleichtern. Im Gegensatz zu anderen Dynamischen Logiken werden Programm und Spezifikation in einem Schritt manipuliert und syntaktisch verglichen.

Der Mangel an Modularität in Verifikationssystemen für verteile Systeme und die Notwendigkeit eines sturkturierteren Ansatzes im Entwurf von Programmlogiken sind als Defizite existierender Ansätze durch die Anwendung dieser Ansätze auf das `FormbaR` Modell für Eisenbahnbetrieb, der momentan größten Verifikationsstudie mit Active Objects, ermittelt worden. Diese Thesis schließt mit einer Diskussion die exemplarisch zeigt, dass BPL Eigenschaften spezifizieren und verifizieren kann, die über die Möglichkeiten bisheriger Ansätze hinausgehen.

Contents

1 Introduction

Reasoning about distributed programs and models for distributed systems is notoriously hard. Direly needed formal support requires more elaborate theories, techniques and tools than formal reasoning for sequential programs.

Under reasoning we understand three aspects: Specification, analysis and verification. Specification provides a way to describe possible properties of a program, analysis derives some property of a program and verification checks whether a program satisfies a given property [29]. A formal system implementing specification and either the analysis or verification aspect is a *reasoning system*.

The properties in question can be informally grouped into functional properties, non-functional properties[1] and generic properties [7]. A *functional* property states that the program implements the correct computational behavior, described by some specification in terms of the program. A functional property states *what* the program is computing in terms of the program itself, e.g., that it computes the correct result or implements the correct protocol. In contrast, a *non-functional* property is a statement about *how* a program realizes its behavior in term of the program [118]. An example for a non-functional property is the time complexity of a program, which is stated in terms of its input parameters. A generic property can be stated with a specification that is independent of the program, e.g., deadlock freedom.

In the following, we state two challenges for formal reasoning for distributed programming: Interdependence of properties and specification of heterogeneous structure.

Interdependence of Properties.

The increased complexity of concurrent systems is due to their inherent non-determinism. Non-determinism has the effect that for concurrent systems, non-functional/generic properties, in particular the order of events is more important than for sequential systems, when verifying functional properties. Functional and non-functional/generic properties already interact for sequential programs (for example, termination must be proven for total correctness), but their interactions are amplified in a concurrent setting.

Example 1.1. *Consider the following code snippet with two method calls that initialize and use some service at location* `0` *with the (functional) specification that the value at* `res` *at the end of the execution is positive:*

```
0.init(); res = 0.service(5);
```

In a sequential program, to establish (partial) functional correctness it may suffice to (1) check whether `service` *returns a positive value for input* `5` *if some predicate holds in its prestate and (2) whether* `init` *establishes this predicate. In a distributed setting complications may occur that require additional information, depending on the concurrency model.*

1. *Endpoint* `0` *may start both method calls in parallel, causing a data race – now it is necessary to check that method* `init` *always establishes the prestate predicate before method* `service` *relies on it.*

2. *If the methods are atomic themselves, the calls may be still be reordered at callee-side – now it is additionally necessary to check that method* `init` *is always* executed *first.*

[1] There is no formal definition to distinguish functional and non-functional properties, for the problems defining such a classification we refer to Glinz [60]. Glinz' discussion is from the related field of requirements engineering, but the general arguments are applicable, e.g., time complexity is a (non-functional) performance requirement. We do *not* distinguish them by deciding whether a property describes *behavior* or not, as put forward by, e.g., Eckhardt et al. [47].

The above example shows how functional properties rely on non-functional properties. However, non-functional properties may also rely on functional ones.

Example 1.2. *Following Ex. 1.1, consider the prestate predicate* `j > 0` *and the following method body for* `0.service(Int i)`*. The* **await** `j > 0` *statement waits until field* `j` *is positive and then continues execution:*

```
await j > 0; return i*j
```

To verify the non-functional property that the **return** *statement of* `0.service` *is executed after* `0.init`*, one must, among other things [78], check the functional property that* `0.init` *indeed finishes its computation correctly and establishes* `j > 0`*. Another example of a non-functional property is non-interference [61].*

Reasoning systems for distributed programs mirror the structure of interactions between multiple properties and are complex programs themselves, which are hard to understand, develop and communicate. As an example, the original formalization of Session Types [71, 72] verifies a generic property (deadlock freedom) by using a functional specification (some global protocol) and applies a non-functional analysis (linear usage of channels) in the process. This *interdependence of properties* forces reasoning systems to combine multiple analyses and verification techniques to verify one property.

It is in particular challenging to design reasoning systems for functional behavior, if the targeted concurrency model allows multiple ways for parts of the system or program to interact. Using multiple ways of interaction is observed widely in current programs, which mix multiple communication paradigms provided by multiple libraries [122]. Each communication paradigm may require to use its own analyses expressed in its own formalism. Reasoning about the whole system requires to combine results in an appropriate way by an overarching, more general formalism. Alternatively, only one formalism is used and all analyses must comply to it. In any case, encoding or combination of results adds complexity to the overall reasoning system.

Specifying Heterogeneous Structure.

The above issues, multiple analyses for one property and different formalisms for different analyses, are raised for the verification and analysis of properties — a similar issue however, arises for specification: To define the functional behavior of a program, one may first define the functional behavior of its subprograms — however, the concurrency model may have a heterogeneous structure and have different kinds of subprograms, all requiring a different way of specification. E.g., in an object oriented model, the specification for a single method differs from the specification for an object. Both differ from the specification of the whole program.

Example 1.3. *Consider the pseudo-code below that defines an object* `0` *as the endpoint of Ex. 1.1 and wraps the caller in an object* `C`*. We can provide specifications for several parts of the code. First, we can specify for* `service` *that if the input parameter* `i` *and field* `j` *are both positive, then so is the return value. Second, we can define an object invariant for* `0` *stating that the field* `j` *is always positive or zero. Lastly, we may specify the protocol realized by this system when* `C.run` *is called from the outside: First* `C` *calls* `init`*, then it calls* `service`*. Object* `0` *respects this order of calls in its scheduler and the final return value of* `C.run` *is positive.*

```
1  object 0 {
2      Int j = 0;
3      Int service(Int i){
4          return i*j
5      }
6      Unit init(){ j = j + 1; }
7  }
```

```
8  object C {
9    Int run(){
10       await 0.init();
11       Int res = 0.service(5);
12       return res;
13   }
14 }
```

The specifications in the above example are heterogeneous, following the heterogeneous structure of the program. They differ not only in their scope, but also in further details: Object invariants express a

constraint at every point in time[2] over all fields of an object – the precondition of a method additionally ranges over the input parameters of the method, the postcondition over a special symbol for the return value. Finally, the protocol adherence is a more complex temporal property.

Relevance and Active Objects.
The challenges are not merely of theoretical nature, but hamper verification for complex concurrency models. We performed a large industrial case study with available tools on a model of railway operations and the lack of a theory to deal with interdependency of properties and lack of heterogeneous specification is identified as a major issue in the verification studies and the development of new reasoning systems. The study was performed with an Active Object language. The *Active Object* [37] concurrency model for distributed programs exhibits both challenges, i.e., multiple ways to interact and a heterogeneous structure. An Active Object program is hierarchical and contains as subprograms objects communicating with each other, methods within the objects and statements within the methods. To interact, objects may use asynchronous method calls and futures [65]. Methods communicate within their object via the heap memory and cooperative scheduling, and with methods within other objects also via futures. Active Objects are, thus, suitable for the study of reasoning systems for complex concurrency models.

1.1 Approach

In this work, we present a reasoning system for functional behavior of Active Objects which addresses all of the above points. We are able to specify and verify functional properties beyond the state of the art of reasoning for Active Objects, by an emphasis on *modularity* in the design of the reasoning system. Modularity is used threefold to address the above issues.

Modular Specification We use four different specification languages to specify the behavior of different parts of the program in different situations. The four specification languages are hierarchical in the sense that each language is specifying a more narrow context than the language below it in the hierarchy. A narrower context allows more precision, but at the cost of a less general specification.

- Session Types specify the behavior of the whole *system*.
- Object Invariants are properties of the state of an *object*, that every method must preserve.
- Method Contracts specify the required input and the guaranteed output of a single *method*, its anticipated concurrency context (i.e., a certain constraint on scheduling required in addition to the precondition) and its suspension points.
- Postcondition reasoning is used to specify the postcondition of a *statement* in isolation.

Fig. 1.1 gives an overview. This addresses the problem of specifying heterogeneous structure.

Modular Verification Calculi Instead of developing a calculus that aims to verify all these functional specifications of a program, each specification is verified by its own calculus. These calculi, however, are calculi for the same program logic: The *Behavioral Program Logic* (BPL).

BPL is a dynamic logic with behavioral modalities $[\mathtt{s} \Vdash \tau]$, where the behavioral specification τ specifies trace properties of the statement $\mathtt{s}$. Each specification aspect above introduces its own behavioral specification and a calculus for it. When verifying a method, we do, however, not use the single calculi directly. Instead, they are composed before verification — each aspect can use the context derived by the calculus of another aspect to increase its precision. The composition of calculi is defined on the rule level to avoid the notoriously complex rules used for systems

[2] Or more precisely: At every point in time where the memory is read by anybody but the currently active process (that is aware that it possibly broke the invariant itself temporarily).

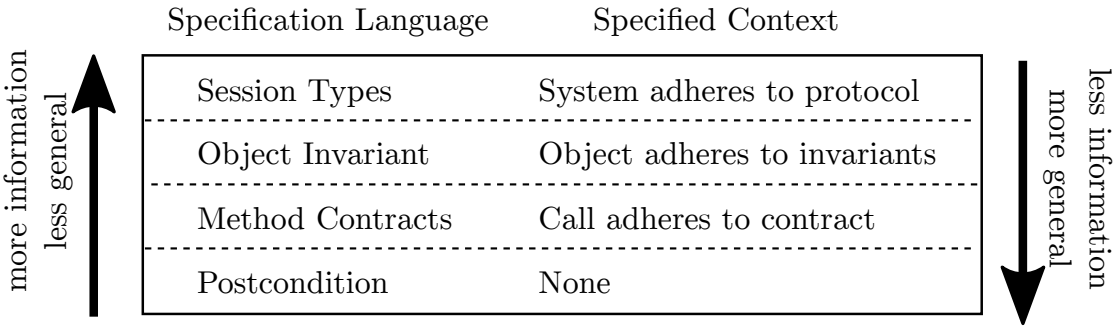


Figure 1.1: Hierarchy of Specifications

with multiple possibilities of interaction. This addresses the problem of interdependent functional properties.

Modular Integration of Analyses As discussed, verification of functional specifications may require the analysis of non-functional properties. We use the modality of BPL not only to express the four functional properties mentioned above, but any property of traces of single methods. To verify these properties, an external system can be used, that needs not to be based on a program logic. As we only specify the result of the analysis, its implementation is treated as a blackbox but its result can be used inside the program logic. In particular, this allows *deductive reasoning* about the results of static analyses. This addresses the problem of interdependency of functional and non-functional/generic properties.

BPL specifies statements, i.e., the method bodies in a program — specifications thus require composition principles to combine the local properties of the methods to a global property of the whole program. This holds also for seemingly non-global method contracts – however, they require the global property that every method call *in the whole program* observes the given contracts. Furthermore, Session Types require a *de*composition principle to generate the specifications for methods, which matches the composition principle.

Some of such (de-)composition principles have been developed for a different verification technique, *behavioral types*. Behavioral types are defined as descriptions of programs, "*in terms of the sequences of operations that allow for a correct interaction among the involved entities*" [1]. Similar to BPLs, behavioral types are reasoning systems that verify programs by giving a calculus to verify methods[3], such that the results of the verification are compositional. Indeed, we demonstrate that the type systems of some behavioral types can be interpreted as a sequent calculus for dynamic logic and, thus, a behavioral type can be interpreted as a program logic.

Summary of Main Contributions.

The reasoning system we present is for the *Core Active Object* language (CAO), a language similar to ABS [75] and, to a lesser degree, Rebeca [117]. The main contributions of this thesis are the following:

- *Behavioral Program Logic* is a dynamic trace logic that (1) allows us to verify properties of active object traces by deductive reasoning about multiple behavioral specifications by enclosing them in *behavioral modalities* (2), has a compositional calculus that allows us to compose multiple calculi (for different behavioral modalities and behavioral specifications) on the rule level, and (3) allows us to interpret behavioral types as program logics.

- *Cooperative Method Contracts* generalize rely-guarantee reasoning for methods to a setting with asynchronous method calls and cooperative scheduling. This is achieved by (1) specifying the

[3] Or other subprograms, in non-object-oriented concurrency models.

non-functional concurrency context in addition to the pre-/postcondition of the method, (2) analogously, specifying suspension points with pre-/postconditions and non-functional concurrency context and (3) making use of a propagation mechanism to ensure coherent specification.

- *Hierarchical Specification* for Active Objects that allows us to reason about the behavior of a method in multiple contexts: (1) in isolation, by postcondition reasoning, (2) in a consistent object, by invariant reasoning, (3) in a specific concurrency context, by cooperative method contracts and (4) in a protocol, by Session Types. All these specifications have their own BPL calculus and (de-)composition principle. The compositionality of BPL allows us to compose their calculi to use information derived by one calculus in another one to increase precision.
- We furthermore give a new formalization of dynamic frames, a new Session Type system for Active Objects and a new Locally Abstract, Globally Concrete semantics without continuation-markers.

1.2 Structure and Publications

This work is organized as follows. The remainder of this chapter describes the publications this work is based on and fixes basic notation used in it. Chapter 2 describes our work on `FormbaR` and analyzes the discovered shortcomings of state-of-the-art approaches in deductive verification of Active Objects. The following chapters introduce the technical contribution of this thesis.

- Chapter 3 introduces `CAO`, an Active Object language.
- Chapter 4 introduces BPL and gives the calculi for postconditions and object invariants.
- Chapter 5 introduces effect types in BPL, which we use to formalize dynamic frames.
- Chapter 6 presents Cooperative Method Contracts.
- Chapter 7 presents the Session Types for `CAO`.

In chapter 8 we illustrate how the challenges identified in chapter 2 have been overcome and conclude with discussion, related and future work in chapter 9. Each chapter contains first the technical contents and then a discussion on limitations and design decisions. These design decisions require forward references, as the preliminaries are designed to stress the novel points of the contribution. The discussions do not discuss possible extensions to overcome the limitations, this is done in chapter 9.

Included Publications.

The results of the following publications are part of the contributions of this work.

- [79] Kamburjan, E. Behavioral program logic. In *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings* (2019), S. Cerrito and A. Popescu, Eds., vol. 11714 of *Lecture Notes in Computer Science*, Springer, pp. 391–408

  This article [79] and the technical report [80] introduce BPL and LAGC semantics without continuation markers. I am the sole author of these papers. These publications are the basis for chapters 3 and 4.

- [82] Kamburjan, E., and Chen, T. Stateful behavioral types for active objects. In *Integrated Formal Methods - 14th International Conference, IFM 2018, Proceedings* (2018), C. A. Furia and K. Winter, Eds., vol. 11023 of *Lecture Notes in Computer Science*, Springer, pp. 214–235

  This publication [82] and the technical report [81] extend Session Types for Active Objects by (1) the ability to handle state and specification of communicated data by logical predicates (2) the integration of static analyses into the calculus and (3) denotational semantics for Session Types.

I am the main author of this paper, all the above ideas are mine. The Session Types in chapter 7 are based on this publication and [79]. BPL is a generalization of the calculus given in [82].

- [84] KAMBURJAN, E., DIN, C. C., HÄHNLE, R., AND JOHNSEN, E. B. Asynchronous cooperative contracts for cooperative scheduling. In *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings* (2019), P. C. Ölveczky and G. Salaün, Eds., vol. 11724 of *Lecture Notes in Computer Science*, Springer, pp. 48–66

  This paper extends method contracts to the model of Active Objects.

  I am the main author of this paper, the main ideas (suspension and resolving contracts, context sets and integration of static analyses) are mine. This publication is the basis for chapter 6.

- [85] KAMBURJAN, E., DIN, C. C., HÄHNLE, R., AND JOHNSEN, E. B. Behavioral contracts for cooperative scheduling. In *Deductive Verification: The Next 70 Years* (2020). To appear in LNCS

  This paper is an extended version of [84].

- [78] KAMBURJAN, E. Detecting deadlocks in formal system models with condition synchronization. *ECEASST 76* (2018)

  This publication presents a deadlock checker that can deal with condition synchronization and is the first deadlock checker which is sound for whole CoreABS. I am the sole author of this paper. This publication forms the basis for Example 4.7 in chapter 4. The deadlock checker itself is not part of this work.

- The following publications describe the `FormbaR` model, an ABS model of railway operations based on the rulebooks of DB Netz AG and the application of state-of-the-art approaches to this model. We use the `FormbaR` model for analyses of these approaches.
  - [87] KAMBURJAN, E., AND HÄHNLE, R. Deductive verification of railway operations. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification, RSSRail 2017, Proceedings* (2017), A. Fantechi, T. Lecomte, and A. B. Romanovsky, Eds., vol. 10598 of *Lecture Notes in Computer Science*, Springer, pp. 131–147
  - [86] KAMBURJAN, E., AND HÄHNLE, R. Uniform modeling of railway operations. In *Formal Techniques for Safety-Critical Systems - 5th International Workshop, FTSCS 2016, Revised Selected Papers* (2016), C. Artho and P. C. Ölveczky, Eds., vol. 694 of *Communications in Computer and Information Science*, pp. 55–71
  - [90] KAMBURJAN, E., HÄHNLE, R., AND SCHÖN, S. Formal modeling and analysis of railway operations with active objects. *Sci. Comput. Program. 166* (2018), 167–193

  [86, 90] describe the modeling itself and one verification study, [87] two further verification results for it, all proven in ABSDL, I am the main author of these papers. The verification studies in these publications are done by me, the modeling in ABS is done by me.

Further Publications.

The following publications were published during the writing of this thesis, but are not directly part of the contributions.

- The following publications describe further aspects of `FormbaR` which are not relevant for this thesis. [91] describes the visualization and integration into the workflow of rulebooks authors. [88] is a summary of [86, 90] in German, [116] describes possible uses cases for the prototype developed in [86, 90, 91], also in German.
  - [91] KAMBURJAN, E., AND STROMBERG, J. Tool support for validation of formal system models: Interactive visualization and requirements traceability. In *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE 2019* (2019), R. Monahan, V. Prevosto, and J. Proenca, Eds., vol. 310 of *EPTCS*

  - [88] Kamburjan, E., and Hähnle, R. Formalisierung von betrieblichen und anderen Regelwerken - Das FormbaR Projekt. In *Scientific Railway Signalling Symposium* (2017)
  - [116] Schön, S., and Kamburjan, E. The future use cases of formal methods in railways. In *Scientific Railway Signalling Symposium* (2018)

- The following publications describe the use of software product lines for interproduct code reuse. [31] describes the integration of traits with deltas and [32, 33] the extension of delta-oriented software product lines to handle multiple products in one code base.
  - [31] Damiani, F., Hähnle, R., Kamburjan, E., and Lienhardt, M. A unified and formal programming model for deltas and traits. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings* (2017), M. Huisman and J. Rubin, Eds., vol. 10202 of *Lecture Notes in Computer Science*, Springer, pp. 424–441
  - [32] Damiani, F., Hähnle, R., Kamburjan, E., and Lienhardt, M. Interoperability of software product line variants. In *Proceeedings of the 22nd International Systems and Software Product Line Conference, SPLC 2018* (2018), T. Berger, P. Borba, G. Botterweck, T. Männistö, D. Benavides, S. Nadi, T. Kehrer, R. Rabiser, C. Elsner, and M. Mukelabai, Eds., ACM, pp. 264–268
  - [33] Damiani, F., Hähnle, R., Kamburjan, E., and Lienhardt, M. Same same but different: Interoperability of software product line variants. In *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday* (2018), P. Müller and I. Schaefer, Eds., Springer, pp. 99–117

- [89] Kamburjan, E., and Hähnle, R. Prototyping formal system models with active objects. In *Proceedings 11th Interaction and Concurrency Experience, ICE 2018* (2018), M. Bartoletti and S. Knight, Eds., vol. 279 of *EPTCS*, pp. 52–67

  This publication proposes Active Objects as a tool for prototyping of formal models and presents a weak memory model in ABS.

A Note on Session Types for ABS.
Session Types for ABS have been developed by the author in the following two publications:

- [77] Kamburjan, E. Session Types for ABS. Tech. rep., TU Darmstadt, 2016
- [83] Kamburjan, E., Din, C. C., and Chen, T. Session-based compositional analysis for actor-based languages using futures. In *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Proceedings* (2016), K. Ogata, M. Lawford, and S. Liu, Eds., vol. 10009 of *Lecture Notes in Computer Science*, pp. 296–312

These publications are not part of the contributions of this work and the presented Session Type system differs from the one in [77, 83] by a new projection mechanism, a different passive choice operator, the ability to handle state and specification of communicated data by logical predicates, a different semantics/translation, a different verification system ([77] does not have a type system, only a translation into dynamic logic, the type system of [83] does not form the base for the type system shown here) and multiple smaller changes following from the above points. We do, however, extend their syntax and the general ideas of translation into logic, twofold projection and causality graphs.

1.3 Notation

We use the notation $\langle a_1, a_2, a_3, \ldots \rangle$ for finite sequences and ε for the empty sequence. Concatenation of two sequences A, A' is denoted by $A \circ A'$. The ith element of a sequence A is $A[i]$. The first element of a sequence has index 1. The subsequence from (including) index i to (including) index j is denoted by $A[i..j]$ and the length of a sequence with $|A|$. The subsequence from (including) index i to the end is denoted by $A[i..]$. Given a finite index set $I = \{i_1, i_2, \ldots, i_n\}$ with $I \subset \mathbb{N}$ we also use the notation $(a_i)_{i \in I} = \left\langle a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_n} \right\rangle$. In this case $a_j = (a_i)_{i \in I}[j]$ if $j \in I$. Given a sequence of sets $(S_i)_{i \in I} = \langle S_1, S_2, \ldots, S_n \rangle$ we use the shorthand notation $\bigcup_I (S_i)_{i \in I} = S_1 \cup S_2 \cup \cdots \cup S_n$ (analogously for $\bigcap$).

We give functions explicitly by $\{a \mapsto b, \ldots\}$ (if $f(a) = b$). Function update is denoted by

$$f[a \mapsto b](x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases}$$

We denote the substitution of t by t' in e with $e[t \setminus t']$, where e is a program (or a part, such as a statement or an expression), a logical formula or term and t and t' are from the same syntactical category.

2 State of the Art and Overview

In this chapter we describe the state of the art in verification and specification of Active Objects and the shortcomings that motivate our approach. To do so, we (1) describe the verification of safety properties in the FormbaR model, the currently biggest case study for verification of functional properties with Active Objects, (2) analyze which of the encountered problems are structural shortcomings of the theory and (3) derive properties that solve these problems.

2.1 Active Objects in a Nutshell

We introduce Active Objects in chapter 3, but to illustrate FormbaR we give an informal account here.

The *Abstract Behavioral Specification* (ABS) language used for FormbaR is an object-oriented language where all fields of the classes are object-private, i.e., only the object itself can access its own fields. Two objects may communicate via asynchronous method calls, where the caller continues execution after the call and does not wait for the callee to process the call when continuing execution. Instead, the caller generates a future, which identifies the process started by the call. The caller may pass the future around. Two synchronization operations can be performed on a future: (1) reading synchronization, which blocks the reading process until the process identified by the future terminates and then reads the result and (2) suspension, which deschedules the reading process until the process identified by the future terminates. An alternative form of suspension[1] is *condition synchronization*, where a process is descheduled until a boolean guard-expression evaluates to true. A process can only be descheduled if it suspends itself or terminates.

Example 2.1. *Consider the class in Fig. 2.1, whose* `run` *method synchronizes on the condition* `this.counter >= 0` *in line 7. Afterwards, it calls the* `J`*-object* `other` *repeatedly (line 9) and suspends using the resulting future (line 11). As the field* `lock` *is set to true before suspension, it is ensured that the* `counter` *field is not modified by the* `setCounter` *method, because it can only be scheduled if* `lock` *is set to* `False`*. However, the current counter can still be read via* `getCounter`*, as the process of* `run` *is suspended. Once the called* `J.call` *process has terminated, the object reads its return value (line 13) and subtracts it from the counter.*

Note that to enter the loop, the `setCounter` method must have been called before. The `run` process terminates, if `J.call` always returns a strictly positive number (and always terminates). We verify a similar pattern in FormbaR to formally show that during the execution of the loop in `run`, the `counter` field is not written to by any other process.

2.2 The FormbaR Case Study

We describe the FormbaR model, the verified safety properties and the application of deadlock checking. The modeling is only introduced as far as needed to establish our conclusion about the shortcomings of current approaches to deductive verification. This section summarizes [90].

FormbaR models the railway operations as described by the technical documents of DB Netz AG. These documents describe how train stations communicate with each other and with trains (in case of faults), how trains interact with the track-side infrastructure (e.g., signals or magnets) and the inner-workings

[1] In coreABS, i.e., without time. Timed ABS was used in FormbaR, but time plays no role in the verification study so we refrain from introducing it.

```
class C(J other) implements I {
  Int counter = -1;
  Bool lock = False;
  Unit setCounter(Int c) { await !this.lock; this.counter = c; }
  Int getCounter{ return this.counter;}
  Unit run(){
      await this.counter >= 0 && !this.lock;
      while(this.counter >= 0){
        Fut<Int> f = other!call();
        this.lock = True;
        await f?;
        this.lock = False;
        Int i = f.get;
        this.counter = this.counter - i;
      }
  }
}
```

Figure 2.1: Example for an Active Object in ABS.

of the train stations that coordinate the track-side infrastructure. The procedures are not described in a uniform way, instead each party involved has a local description of the global procedure.[2] One of the aims of FormbaR is to verify that the local descriptions suffice to establish safety of the global procedures.

The technical documents consist of legal regulations, the Eisenbahn-Bau- und Betriebsordnung (Law for Operating and Building Railways) [48], (2) public rulebooks managed by Deutsche Bahn (DB), in particular Ril. 408 [34] and 819 [35], (3) internal rulebooks for operations, (4) requirements specification for technical elements, (5) training documentation and (6) internal announcements. Most of the documents describe the behavior of humans, so it is not possible to use the model to generate provably correct controllers. The model can only be used to reason about an idealized world where humans make no mistakes. A rulebook assumes some conditions on the infrastructure: for one realistic physical behavior, e.g., that train do not skip over parts of the track they are driving on, and for another, restrictions from other rulebooks, e.g., that there is a certain minimal distance between two signals. Thus, the verification must hold for *any* infrastructure according to these conditions and not only for some fixed scenarios. In particular, we assume no bound on the (finite) size of the network.

Railway operations fit the Active Object concurrency model well: stations, track-side infrastructure and trains share no state and only interact with asynchronous messages. Messages may be acknowledged and we model this as synchronization on futures. Internally, entities (e.g., stations) wait on conditions before continuing with procedures. We model this as condition synchronization.

2.2.1 Formal Modeling of Railway Operations

The FormbaR model consists of infrastructure, communicating stations and trains. The stations issue commands to both infrastructure and trains. We first describe the infrastructure.

Layered Infrastructure

The infrastructure is modeled as a graph based on the concept of a *point of information flow* (PIF).

[2] Explaining material, e.g., the introduction of Pachl [110] to railway operations, contain a global description, but these documents only serve illustratory purposes.

Definition 2.1 (Point of Information Flow)**.** *A* point of information flow *(PIF) is an object at a fixed position on a track that participates in the information flow, if one of the following criteria applies:*

- *It is a structural element allowing a train to receive information, for example, a signal or a data transmission point of a train protection system.*
- *It has a critical distance before another PIF, where its information is transmitted at the latest. E.g., at the point where a signal is seen at the latest. We call points where this criteria applies, but no device is placed* virtual*.*
- *It is a structural element allowing a train to send information, for example, a track clearance detection device, such as axle counters or endpoints of switches.*

A PIF is a position at a track and an object that describes the information to be transmitted or relayed. Instead of modeling all features of a PIF in one object, we use a model of four layers to organize and separate its structure:

Topology Layer. The lowest layer is the *Topology Layer*, which consists of nodes and edges. Each node corresponds to the position of some PIF, the edge between two nodes models a track. Edges are undirected and have a length.

Physical Element Layer. The next layer is the *Physical Element Layer*. Its elements correspond to physical devices on the infrastructure, or virtual points with a certain distance (i.e., virtual points of information flow) to physical devices. Each element of this layer is assigned a node in the topological layer. Nodes can have multiple physical elements assigned, if several devices share one location, e.g., a presignal and a main signal,

Logical Element Layer. The next layer is the *Logical Element Layer*. Its elements group multiple physical elements from the lower layer. For instance, a presignal, a main signal, the visibility point of the presignal and the magnet in between form together one logical signal. Logical elements are the train dispatchers view on the infrastructure, as he cannot control its physical elements individually, while physical elements are the train driver's perspective, as he reacts to the physical elements he sees (or which are detected in another way by train-side components).

Each physical element can be assigned to one logical element. A physical element can be shared between multiple logical elements, e.g., a presignal can be used for two main signals. Physical elements can not be assigned to a logical element if their state never changes, e.g., gradient changes.

Interlocking Layer. Logical elements are summarized in the *Interlocking Layer*. Each element corresponds to one station. Each logical element is assigned to exactly one station, the one that controls it.

Fig. 2.2 shows the entry to a simple station in the layer model. The logical switch is assigned three physical elements, which correspond to the outermost points of the physical switch. The logical entry signal has a physical presignal, main signal, three magnets and two points of danger which are covered by the signal. These points of danger are physical devices: at this position some train detection device, such as axle counters, is placed. One of the magnets shares its position with the main signal.

Communication

Stations only communicate with their logical element and with the adjacent station. Logical elements do not communicate with each other. Only one physical element communicates with other physical elements: points of visibility read the state of the physical device whose visibility they model. Trains interact with the topological layer. Thus, communication goes either up or down through the layers,

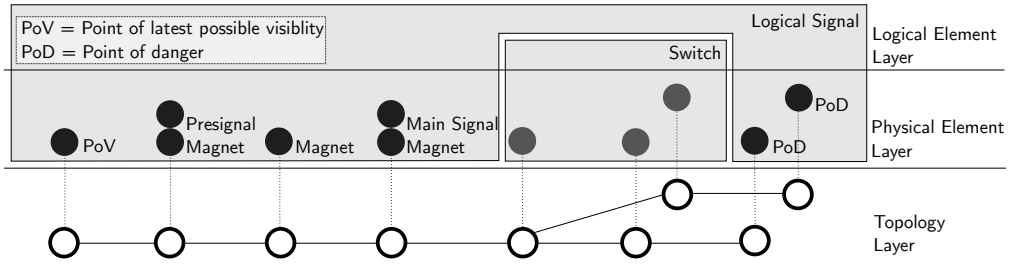


Figure 2.2: Illustration of the lower three layers of a station entry in the four-layer model. [87]

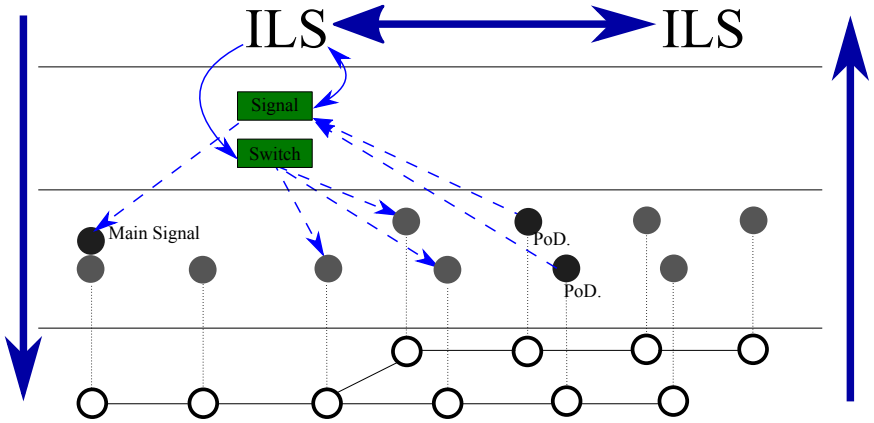


Figure 2.3: Communication in the four-layer model. The blue arrows show the standard information flow direction if no fault occurs.

as illustrated in Fig. 2.3. In the following we assume that no faults occur, which require additional communication.

The rulebooks differentiate between two kinds of stations: *Blockstellen* divide a track line into two parts to increase the possible number of trains on the line; and *Zugmeldestellen* (Zmst), which are able to "store" trains and rearrange their sequence. The generalization of both is *Zugfolgestelle* (Zfst). In the following, we use the term station for Zmst.

A Blockstelle is only responsible for the safety on the next section of a line behind one of its signals, while a Zmst is responsible for the safety of the whole line up to the next Zmst. To depart a train on a line L from A to B with the first section S several conditions have to hold[3]: (1) the station A has the permit to use the line, (2) the signal in A covering S can be set to "Go" and (3) the target station B is notified about departure and accepted the train. Condition (3) is, strictly spoken, not for safety, because trains cannot collide if this protocol is not observed, but it is used to ensure deadlock freedom.

Definition 2.2. *There are three protocols that ensure safe operations:*

Locking sections. *Each Zfst is responsible for several logical elements, but has its own state that depends on the neighboring Zfst: Each section has an additional Boolean state* locked.

This state is managed as follows: After a signal is set to "Go" and a train passes it, the covered section is automatically locked *and the message "*preblock*" is sent to the signal at the end of the section. A signal cannot be set to "Go" again, as long as the section it covers is locked. It must be unlocked by receiving the "*backlock*" message from the signal at the end of the section. The backlock message is sent whenever a signal is passed by the train.*

Permit token. *For each line there is exactly one token that allows a station to depart trains on this line. Without the token the signal that covers the track cannot be set to "Go".*

[3] These conditions are for safety, obviously there are additional conditions concerning the time table.

Accepting and reporting back trains. (Zugmeldegespräch) *Before a train leaves a station A with destination B, A* offers *the train and waits for B to accept. This ensures that B has (or will have) a track to park the train. Before the train departs, the departure is* announced *to B. The offering, announcement and acceptance of trains are modeled as methods—the current state of a Zmst is not only encoded in its fields, but also in the currently active (but possibly suspended) processes.*

2.2.2 Verifying Railway Operations

This section summarizes [87]. The following safety property is shown to hold for `FormbaR`:

> *For all stations A and B sharing a line between them, every train run from station A to station B is safe: during the run, no train will enter the line in the direction of A and whenever a signal is set to "Go", the next section is free.*

This is split into two properties: (1) When a train goes from A to B, no train enters the line towards A and (2) whenever a signal is set to "Go", the section it is covering is free. A general methodology developed in [87] is to split and formalize such properties.

First, the property is expressed as an informal statement over states. The second step is to reformulate the property as an informal statement of global traces. An informal global trace is the sequence of visible actions the system is performing. Afterwards, the property is expressed as a formal statement of global traces. Global traces consist of events and states. An event corresponds to some communication action, e.g., an invocation event models that an object called a method on another object with some call parameters and some future. An invocation reaction event with the same future then models the scheduling of this call.

Reasoning about such traces is possible by well-formedness axioms. E.g., if the ith event in the trace is an invocation reaction event with future f, then there is a $j < i$ such that the jth event is an invocation event with the same future. This means that a method is only started after being called. Finally, the global trace property is projected onto local endpoints. For each involved object, a local object invariant is generated. These invariants specify the events visible to the given object. A program logic (ABSDL) is available for such properties.

We give two examples of these invariants which imply the above safety property, but we stress that additionally to general axioms which hold for all ABS programs, such as the one sketched above, we can assume axioms specific to `FormbaR`. There are three classes of such axioms.

Program Structure. One can derive axioms from a given program, in our case the classes of the `FormbaR` model. E.g., if a method `m` is only called from a method `n`, then we can derive the axiom that if the ith event in the trace is an invocation event on method `m`, then there is a $j < i$ such that the j event is an invocation reaction event of `n`.

Well-Formed Infrastructure. In `FormbaR` the input infrastructure is encoded in the main block. We can assume that the input to the model is well-formed as well. E.g., if a section S ends at signal Sg then the handling Zfst Z has set its field `next` correctly: $Z.\texttt{next}(S) = Sg$. The signal Sg' that covers S has its `covers` field set correctly: $Sg'.\texttt{covers} = S$.

Abstraction of Layers. The safety properties we show are on the interlocking layers. To avoid reasoning about the whole stack beneath, its properties are expressed as additional axioms. E.g., that a train only passes a signal after the signal was set to "Go" [4] can be modeled as as axiom that expresses that before every invocation reaction event of the method modeling the arrival of the front of the train at the signal, there is a future event of the method which sets the very same signal to "Go" (and there are no actions on the signal in between).

[4] This is a valid assumption, because we assume train drivers who correctly follow their rules. The verification goal is to show that the protocols on the interlocking level ensure that this indeed avoids crashes.

Permit Token.

This section summarizes section 5.2 from [87]. Each line L has an associated token which gives the permission to dispatch trains on this line. In FormbaR, the token is implemented as a field `permit` in the Zmst class that maps the first section of the line to a Boolean value modeling the token. A Zmst has the token for a section S if `permit`$[S]$ is set to `True`. Before a station dispatches a train, it must first acquire the token for line L. This requires cooperation between the requesting station and the station having the token: The requesting station knows which trains are on the line in its direction, as all the trains are announced and saved as expected. It only requests the token if no trains are on the line in its direction. The station having the token only checks that the token is not locked, i.e., it is not in the process of dispatching a train using this token[5]. The informal state property is as follows:

> *"If station A acquires the permit token for line L from station B, then there is no train on L with arrival station A."*

Station A acquires the permit token when the call on B.`rqPerm` from method `setPreconditions` terminates. If we assume that all stations are connected correctly, the condition that there are no trains on L with arrival station A can be expressed as A.`expectIn`$[S]$ == `Nil`, where S is the first section of L from A. The field `expectIn` is a map from sections to the list of trains that have been accepted on the line ending at this section that have not arrived yet. We can rewrite the above property into the following property over the history.

> *"If station A reads from the future for B.*`rqPerm`*, then at this moment the following holds: A.*`expectIn`*[S] == Nil."*

Finally, we can now formalize the property into the following:

> *The following formula holds for all histories generated by the FormbaR model. Let A be a Zmst and L a line bordering A with S being the first section of L from A and A.*`other`*(S) the last.*
>
> $$\varphi_1(A,S) \equiv \forall i, f.\ h[i] = \mathsf{futREv}(A, \texttt{rqPerm}, f, [\texttt{True}, S]) \rightarrow \sigma[i](A) \models \texttt{expectIn}(A.\texttt{other}(S)) \doteq \texttt{Nil}$$
>
> *Here, $h[i]$ refers to the ith event in history h and $\sigma[i](A)$ to the state of A in the ith global state.*

In the four-layer model, this property specifies the interlocking layer. The lower layers, which are needed to register that a train arrived and is, thus, not expected anymore are abstracted into additional well-formed axioms. We refrain from giving the full decomposition into local invariants and the proof of the above statements. We give only the three object invariants of ABSDL proven in KeY-ABS mechanically:

$$\psi_1 \equiv \forall\, \texttt{Train}\ T.\ \forall\, \texttt{Section}\ S. last(h) = \mathsf{futEv}(\textbf{self}, \texttt{offer}, f, [T,S]) \rightarrow \textbf{self}.\texttt{allowed}[S] \doteq \texttt{True}$$

$$\psi_2 \equiv \forall\, \texttt{Train}\ T.\ \forall\, \texttt{Section}\ S.\ \forall\, \texttt{Station}\ B.\ last(h) = \mathsf{invEv}(\textbf{self}, B, \texttt{offer}, f, [T,S,\textbf{self}]) \rightarrow \textbf{self}.\texttt{unlocked}[S] \doteq \texttt{False}$$

$$\psi_3 \equiv \forall\, \texttt{Section}\ S.\ last(h) = \mathsf{futEv}(\textbf{self}, \texttt{rqPerm}, f, [\texttt{True}, S]) \rightarrow \textbf{self}.\texttt{unlocked}[S] \doteq \texttt{True}$$

Formula ψ_1 expresses that an `offer` process only terminates for a given section in states where the station is `allowed` to dispatch trains on it. Termination of `offer` models that the other station has accepted the train. Formula ψ_2 expresses that the `offer` method is only called, for a given section, in states where this section is locked, i.e., no other protocol on the exchange of the permit token is active. Formula ψ_3 expresses that the `rqPerm` method, which requests the token, unlocks the section. This is the same pattern as in example 2.1. All three invariants have been proven mechanically, but their derivation is manual.

[5] We verified a conservative fallback mechanism, where only the station having the token secures it, in [86] before.

This section rephrases sections 5.3 and 5.4 from [87].

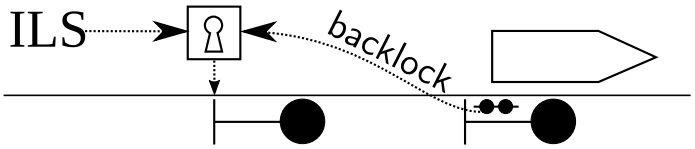


Figure 2.4: *Zugmitwirkung* ("Train Involvement"): The train has to trigger the second signal before the first can be set to "Go" again.

Railway signals are managed by interlocking systems, but are not detached from the actual movement of the trains: *Zugmitwirkung* ("Train Involvement") is an established concept in German railway operations and states that certain actions of the dispatcher are linked to actions of the train and their detection by the infrastructure. We show the following property, taken from [110]: A signal can only be set to "Go", if the train that passed it the last time has left the covered track. To ensure this, when a signal is set to "Halt", after a train passed it, the used line is *locked*. A signal can no longer be set to "Go" when the route is set to the line while the signal is locked. A signal can only be unlocked when the signal *at the end of the covered section* sends a *backlock* message. Fig. 2.4 illustrates the situation. The desired property, expressed as an informal statement over states is as follows:

> *"If an exit signal Sg is set to "Go", then the covered section is free of trains going away from it."*

Given the procedure described above, we can again rephrase this into a trace-based version. For presentation's sake, we do not consider the case that a signal may cover multiple sections.

> *"If a signal Sg is set to "Go" twice, then a train triggered the point of danger of the next signal at some time in between."*

Which is equivalent to the following, informally expressed in terms of methods and events.

> *"If there are two position i, j with $j < i$, such that $h[i]$ and $h[j]$ are invocation reaction events on* `setGo` *on some Signal Sg covering section S', then there is a k with $j < k < i$ such that $h[k]$ is an invocation reaction event on* `trigger` *on* `next`*(S')."*

And, finally, the formal global property:

> *The following formula holds for all histories generated by the* `FormbaR` *model. Let A be a Zmst and Sg a signal.*
>
> $$\varphi_2(A,S) \equiv \forall i.\ \Big(h[i] = \mathsf{invREv}(A, Sg, \texttt{setGo}, f, []) \rightarrow \forall j.\ \big(j < i \wedge h[j] = \mathsf{invREv}(A, Sg, \texttt{setGo}, f', []) \rightarrow \exists\, \texttt{DangerPt}\ P. \exists k.\ j < k < i \wedge h[k] = \mathsf{invREv}(P, \texttt{next}(Sg.\texttt{covers}), \texttt{trigger}, f'', [])\big)\Big)$$

The above statement does not rely on object invariants: well-formedness axioms suffice to prove it. The following property was additionally proven.

> *If a station A accepts a train t, then there is a track reserved for t. This means that the following is an ABSDL invariant for the* `ZugMelde` *class:*
>
> $$\forall\, \texttt{Train}\ T.\ \forall\, \texttt{Section}\ S.\ \forall\, \texttt{Station}\ B.\Big(\big(last(h) = \mathsf{futEv}(\mathbf{self}, B, \texttt{offer}, f, [T, S, B])\big) \rightarrow \exists\, \texttt{Signal}\ Sg.\ \mathbf{self}.\texttt{reserved}[Sg] \doteq T\Big)$$

```
1  class SignalImpl implements Signal{
2    Bool x = False;
3    Unit waitFree(){ await x; }
4    Unit unlock(){ x = True; }
5  }
6
7  class ZugImpl(Sig o, Sig o2) implements Zug {
8    Unit go(){
9      Fut<Unit> f = o!waitFree(); await f?;
10     o2!unlock();
11   }
12 }
```

```
13 {
14     Signal s1 = new SignalImpl();
15     Signal s2 = new SignalImpl();
16     Zug z1 = new ZugImpl(s1,s2);
17     Zug z2 = new ZugImpl(s2,s1);
18     z1!go(); z2!go();
19 }
```

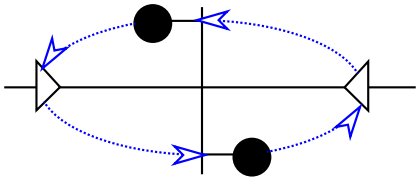


Figure 2.5: A deadlocked situation with condition synchronization. The code is a simple model of locked signals, the image is an illustration of the situation modeled by the main block. Interfaces in the code are omitted.

The general property shown is, thus, the following. We stress that this result holds for any well-formed infrastructure, without any limitation on the size.

> *For any well-formed infrastructure, every train departure is safe: when the exit signal Sg in station A is set to "Go" for train T on a line L to station B, then the first section of L is free, no train is on L in direction of A and a track is reserved for T in B.*

Deadlock Freedom.

Additionally to the above safety property, deadlock freedom of `FormbaR` is investigated. To do so, the DECO tool [51] is extended to deal with condition synchronization. We sketch the extension and experiment here, because the extension integrates a program logic, specifically designed for this application.

Example 2.2. *Consider Fig. 2.5. The code shown is a vastly simplified railway operations model. A signal has a state,* `x`*, which models whether it is set to "Go" or not. A signal can be observed from the outside until it is set to "Go" with the* `waitFree` *method. The* `unlock` *method sets the signal to "Go", i.e., the next section is free.*

A train waits for a given signal to show "Go" (line 9) and then unlocks the signal of the section it left (line 10). The main block shown sets up the situation pictured on the left of Fig. 2.5: both trains are waiting for a signal to signal "Go", and both signals are in turn waiting for the trains to unlock them.

Such a situation can occur if trains are accepted in the wrong order at the wrong time.

The original DECO tool cannot detect the deadlock in the given example, because it cannot handle condition synchronization, i.e., awaits with a boolean expression as their guard (in contrast to a future access). Instead, it returns that the program is deadlock-free. Thus, DECO was extended by me.

Dependency Graphs.

DECO is based on a *dependency graph* analyses. A concrete dependency graph can be derived from a concrete program state: its nodes are the tasks (i.e., processes) and objects in a program. We give an informal account here.

Let t, t' be tasks and o an object. A node (t, t') models that task t depends on task t', i.e., t cannot continue execution before t' has performed some action. In the non-extended DECO tool, the action in question is termination. An edge (t, t') models that a task t synchronizes on the future resolved by t' with an **await** statement of the form **await** `e?` or **get** statement. Additionally, edges (t, o) model that task

t is suspended and awaits to be scheduled on o. Edges (o,t) express that object o is currently blocked by a task t that is blocking at a **get** statement. A program state (a set of tasks and objects) is deadlocked iff its dependency graph contains a cycle.

The above definition does not handle **await** statements with boolean guards. To do so, we require an additional dependency.

Definition 2.3. *Let* $\mathtt{s}_1, \mathtt{s}_2$ *be two statements of tasks* t_1, t_2*, with* $\mathtt{s}_1 = \mathtt{await}\ \mathtt{e}_1;\ \mathtt{s}_3$ *for some* $\mathtt{s}_3$*. We say that* t_1 *0-depends on* t_2 *in some state, if* $\mathtt{e}_1$ *does not hold in the state and the execution of* $\mathtt{s}_2$ *results in a state where* $\mathtt{e}_1$ *holds. If* $\mathtt{s}_2$ *has the form* $\mathtt{await}\ \mathtt{e}_2;\ \mathtt{s}_4$*, i.e., it cannot be executed, then we say that* t_1 *n-depends on* t_2 *in state* σ *(where* $\mathtt{e}_1$,$\mathtt{e}_2$ *do not hold) if*

- *there is some state* σ' *which differs from* σ *only in the fields occurring in* $\mathtt{e}_2$*, such that* $\mathtt{e}_2$ *holds in* σ'
- *executing* $\mathtt{s}_4$ *results in a state* σ'' *where no further execution is possible or either* $\mathtt{e}_1$ *holds or* t_1 $(n-1)$*-depends on* t_2 *in* σ'' *for some task* t_2*.*

Intuitively, a task t_1 depends on another task t_2 if from some state which would make the guard of t_2 true, a state is reachable which also makes the guard of t_1 true. This is not decidable even for a given state, as it requires to reason about arbitrary Turing-complete statements.

E.g., a task executing `await this.i > 0; return` 1-depends on a task executing `await this.i < 0; this.i = 1; return` in a state with $\mathtt{i} = 0$. For a full formal definition, we refer to [78].

Analysis and Dispensable Dependencies.

To approximate the concrete dependency graphs of all reachable states of a program, *abstract dependency graphs* are generated. The nodes of the abstract dependency graph are object creation sites c and pairs of object creation sites and method names $c,\mathtt{m}$. An edge from a pair $(c,\mathtt{m})$ to a pair $(c',\mathtt{m}')$ models that there is some reachable state with a dependency from a task t running on an object created at c executing $\mathtt{m}$ to a task t' running on an object created at c' executing $\mathtt{m}'$. Analogously for objects and object creation sites.

An edge for boolean guards is added between $(c,\mathtt{m})$ and $(c,\mathtt{m}')$ if $\mathtt{m}$ contains a statement `await e` which contains some field `f` that is written by $\mathtt{m}'$.

A program has a deadlock risk if its abstract dependency graph contains cycle. The analysis is shown sound, i.e., if there is a reachable state with a cycle in its concrete dependency graph, then its abstract dependency graph contains a cycle. Fig. 2.6 shows the abstract dependency graph of Fig. 2.5, which reveals the deadlock. Each edge (v,v') denotes that entity (guard or object) v waits for entity v' to finish, before it can continue.

The analysis is imprecise: there may be abstract edges which correspond to no concrete edges in any reachable state. E.g., consider a method `read` with the method body `await this.i > 0; return;` and a method `write` with method body `this.i = 1; await this.b; this.i = -1; return;`. An edge is added, but `read` does not depend on `write`: a deadlock occurs in a state where both tasks cannot continue execution. I.e., the tasks execute

`await this.i > 0; return;` and `await this.b; this.i = -1; return;`

in some state with $\mathtt{i} \leq 0$. Obviously, the second task can not result in a state where `this.i > 0` holds. To detect such *dispensable* edges we use a variant of the ABSDL logic. It has to be proven that $\mathtt{i} > 0$ is never established after a suspension.

This can *not* be modeled by using `this.i <= 0` as an invariant: (1) it does not have to be established at the first suspension point, or a termination before previous suspension and (2) it cannot be assumed at the method start. Thus, we developed a special modality $\langle\!|\mathtt{s}|\!\rangle\varphi$ which models that φ is established by termination and suspension *after the first suspension*. This is encoded in the calculus by the following rule, which reduces it to reasoning about invariants. The update $U_{\mathscr{A}}$ removes information from the heap [8].

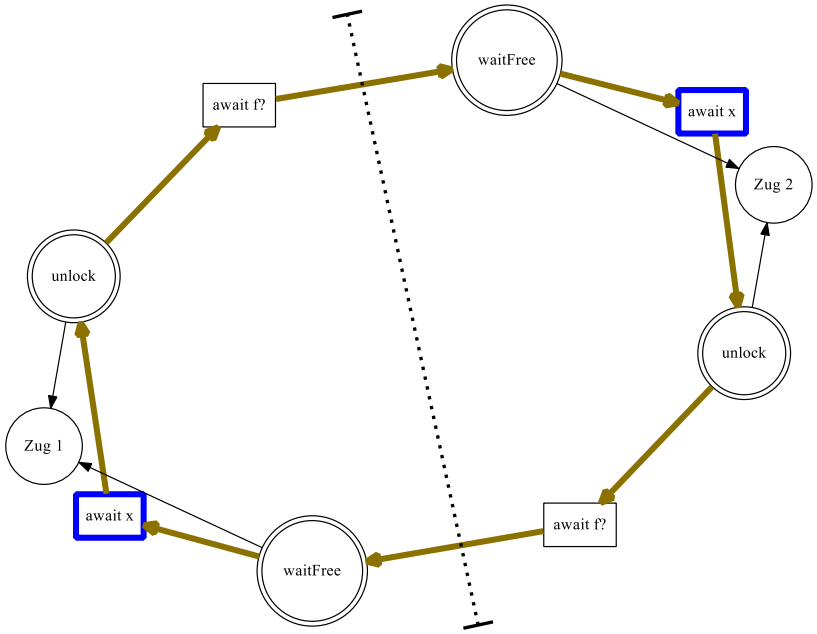


Figure 2.6: Abstract Dependency Graph of Fig. 2.5.

$$\textbf{(first-suspend)}\ \frac{\Gamma \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s}]\varphi, \Delta}{\Gamma \Rightarrow \{U\}\langle\!\langle \textbf{await}\ \mathtt{e?;\ s}\rangle\!\rangle\varphi, \Delta}$$

For each edge (stemming from a guard g) a proof obligation $\langle\!\langle \mathtt{s_m} \rangle\!\rangle \mathtt{g}$ is generated, where $\mathtt{s_m}$ is the method body of the reading method. If the proof obligation can be proven, then the edge is removed from the abstract dependency graph.

Conclusion.
The extended version of DECO can detect the deadlock in Fig. 2.5 and analyzes all industrial case studies available publicly with sufficient precision – except `FormbaR`. Applying the deadlock checker to `FormbaR` results in a memory error because too many potential deadlock cycles are found. Some reasons why `FormbaR` scales so badly is that it is (1) bigger than other models, using far more objects, (2) uses far more condition synchronization, with most guards containing multiple fields and (3) that the (modified) KeY-ABS tool cannot handle non-trivial proof obligations required to discharge the proof obligations stemming from such guards. The last point is due to the modification required for the program logic: It now models *only* the property required to detect dispensable edge and cannot use established object invariants anymore.

2.3 Analysis of Problems

We can observe the following problems with the above case study.

Homogeneous Specification and Homogeneous Verification. Even though the final proof obligations of safety are specifications of single methods, they are encoded as object invariants. Indeed, a general problem is that ABSDL requires that all specification aspects (contracts, invariants, protocols, frames, etc.) are uniformly described as object invariants — even if they are specific to a single method. This raises the question how to split specification into multiple aspects, how to design program logics for single aspects and how to combine these program logics.

This is not merely an inconvenience – specification has been identified as one of the main bottlenecks and challenges for formal methods [4, 64] and forcing the user to use a single paradigm

for any property hampers the usability and acceptance of the language. For example, the second property of the permit token study is a simple postcondition (in the sense of method contracts):

$$\exists \texttt{Section}\ st.\ \texttt{result} \doteq (\mathsf{True}, st) \rightarrow \texttt{unlocked}[st] \doteq \mathsf{True}$$

This does not require the notion of an event or history at all; domain experts can assess the adequacy of this postcondition with less knowledge about the underlying verification mechanism.

In previous work [77], stateless session types for ABS were also translated into object invariants. In both cases, the translation not only hampers automation by requiring a high number of quantifiers (and, thus, required instantiantions by the prover) but was also hard to understand – which makes it hard to understand for the user of an interactive theorem prover, such as KeY-ABS, to understand why a proof fails and to assess the quality of his specification. This homogeneous verification (only a calculus for object invariants) is a consequence of the homogeneous specification (only one specification language).

Projection and Handling State. There is no theory of automatically decomposing global specifications referring to state or method parameters, e.g., the formulas in the verification of the permit token: the only available decomposition is the limited Session Type system [83] that cannot specify passed data or heap memory.

This forces to project global properties manually.

Another problem with the history-based specification in ABSDL is that only events are recorded and the object invariant has to hold only at suspension points. This makes it impossible to express a property connecting a method call with the state of the caller. E.g., "*Whenever* `0.m(i)` *is called, then* **this**`.f == i` *holds*" or "*Whenever a method reads from a future, a flag in its memory is set.*". Similarly, it is not possible to express properties of the state which are not connected to properties of the history. E.g., that a field is not changed during the execution of a method.[6]

In summary: stateful global properties cannot be decomposed automatically, many stateful local properties cannot be expressed as object invariants.

Integration of Static Analyses and Deduction. The integration of KeY-ABS into the deadlock checker to detect dispensable edges required to modify the program logic to the specifics of the analyzed situation. There is no theory of developing, extending or combining program logics to verify new properties, thus, the extension for DECO is an ad-hoc extension that cannot use established object invariants. This is also a consequence of the homogeneous verification mechanism.

Conclusion of Analysis.

ABSDL lacks a theory of method contracts, as well as an integration of static analyses for global properties. Similarly, it lacks a theory of projection to derive local specifications from global specifications. While Session Types for ABS [83] have such a theory of projection, they are only able to specify communication patterns, not state. Thus, they could not be applied to `FormbaR`. The deadlock example shows that deductive verification may be integrated into static analyses, with a specific pattern (based on the static analysis in question) to analyze. Specification of distributed object-oriented programs is an open question and ABS lacks a theory how to handle new specification paradigms.

To overcome these problems we give a notion of contracts for Active Objects and integrate (and extend) Session Types for ABS into a multi-layer specification language. Under multi-layer specification we understand that method contracts, object invariants and session types are specified separately (instead

[6] We stress that this is *not* merely a question of saving the heap in the prestate and referring to it in the poststate (as it is done in JavaDL), because extra rules are required to keep track of the heap at all suspension points: the value of the field must be equal to the one in the state of the last activation, not necessarily the prestate. Such a mechanism is not available for ABSDL.

of specified homogeneously, by e.g., encoding them into object invariants as in ABSDL). They are also verified homogeneously, i.e., they are not encoded into object invariants for verification either.[7]

Furthermore, we give a new program logic that allows to use multiple specifications. This logic can (1) specify an interface to a used static analysis, (2) simplifies the design of projection mechanisms and (3) allows the design of new local specifications of trace properties of methods. Finally, we give a mechanism to compose multiple specifications into one overarching specification. This allows an aspect-oriented design of dynamic program logics: Instead of giving a modality that expresses all possible properties, each specification has its own modality. These modalities can be composed to express multiple specifications, but composition vastly simplifies design and soundness arguments. In particular, the program logic is easy to extend for new specifications, as we show with dynamic frames and effect types.

2.4 Overview over Approach

In this section we describe our approach in more detail.

Behavioral Program Logic

To enable us to tackle the problems describes in chapter 1, we introduce a new program logic: Behavioral Program Logic (BPL), an extension of first-order dynamic logic and related program logics, such as ABSDL and JavaDL. BPL allows for design and composition of multiple specification paradigms, as well as integration of static analyses into deduction.

The core of BPL is the *behavioral modality* $[\mathtt{s} \Vdash \tau]$ which expresses that every trace of $\mathtt{s}$ is a model for τ. The type τ is not a trace formula, but a representation of a trace formula. It can be used as an interface to static analyses (i.e., there are no rules to handle τ, or very few rules to reason about results). Then a proof of formulas containing such modalities must be closed by calling the static analysis corresponding to τ.

Alternatively, a calculus for all possible forms of the statement $\mathtt{s}$ can be given. In this case we show that many specification patterns are easily encodable: object invariants, postcondition reasoning, method contracts (see below), Session Types, effect types, dynamic frames. If multiple specification patterns are used, BPL allows reuse of the sound calculi of the used patterns to automatically generate a sound calculus that uses both patterns.

The separation of the representation of trace properties in the dynamic logic and their semantics allows us to design calculi that incorporate ideas from behavioral types. In particular, the calculi for symbolic execution (i.e., the rules concerned with reducing the statement of one modality) reduce both type and statement. This allows us to formulate all of the above patterns with explicit tracking of the trace. Furthermore, it simplifies the formulation of composition and decomposition patterns.

As a basis for BPL, we use a modified *locally abstract, globally concrete* (LAGC) semantics [43], which simplifies the existing semantics, yet allows to precisely analyze local traces of methods in isolation from the global system (or object).

BPL is the main tool to address the problem of integration of static analyses and deduction and homogeneous verification. It also allows the following contributions, which address homogeneous specification.

Cooperative Method Contracts

To enable local reasoning and exemplify the design of BPL we generalize bottom-up specification with method contracts to Active Objects. The main ideas are as follows:

[7] They are translated into trace formulas, but (1) not in the program logic used for verification and (2) these formulas are not object invariants over histories.

Preconditions. The precondition is split into two parts: a *parameter* precondition which has to be guaranteed by the caller of the method and a *heap* precondition which has to be guaranteed by the previously active process.

Context Sets. To specify the previously active processes responsible for the heap precondition, a method may specify *context sets*. These sets specify which methods may run before the specified one. This specification is *not* verified in BPL, but by a global causality analysis. Context sets are, however, encoded in the same formalism as the trace properties to allow us to use them for composition of method contracts.

Suspension Contracts. Each **await** statement is specified by a suspension contract that resembles a method contract. A suspension contract contains a suspension assertion (that has to hold before suspension), a suspension assumption (that has to hold before reactivation) and context sets (describing the method responsible for the suspension assumption). Again, only the suspension assertion and suspension assumption are used in BPL, the context sets are verified in a static analysis.

Postconditions. Each **get** statement is specified by a resolving contract: a set of methods. It models that the read future is resolved by a process executing one of these methods. This is specified as a behavioral modality, and either verified by an external points-to analysis or by some special rules that allow us to reason about the results of such analyses. In any case, this enables us to use the postcondition of the methods for the read value.

Additionally, Cooperative Method Contracts have an intuitive, JML-style syntax [96] and allow us to specify dynamic frames.

Cooperative Method Contracts address the problem of homogeneous specification by an intuitive way to specify local information. It also addresses the problem of unsatisfying handling of state in ABSDL.

Multi-Layer Specification

As hinted above and described in the introduction, we use multiple specification paradigms and reuse parts of their results (and rules from their calculi). The main ones are the following four: Postcondition reasoning, as it is standard in dynamic logic, is used for reasoning about the result of method without any specific context. If the object and its invariant is known, we include object invariants to reason about suspension contracts and more precise pre- and postcondition. If a method contract is given, we can use it to reason in an even more specific situation, if the object invariant is also given. If not, we can still use the method contract. Finally, Session Types for Active Objects are used for top-down specification of even more restricted situations.

This multi-layer specification addresses the problem of homogeneous specification and the extended Session Types additionally address the problem of projection and handling of state.

3 CAO: An Active Object Language with Locally Abstract, Globally Concrete Semantics

We use the Active Object concurrency model, an extension of the Actor [70] concurrency model. Actors are endpoints which communicate solely via asynchronous message-passing. An actor may receive messages at any time and each message is handled *preemption-free*: once the handler for a message has started, it may not be interrupted by any other handler. Messages arriving during execution of a handler are stored in a bag data structure.

As Actors do not share memory and handlers cannot be interrupted, reasoning about concurrent behavior in Actor-based systems is simpler than for C-style concurrency models with shared memory, where two process may interfere at any point, except when explicitly forbidden to do so. Indeed, compared to cloud systems based on other paradigms, real-world systems build upon pure actors sustain less coordination-based bugs [67]. Erlang [2] and the Akka [97] framework are some mainstream implementations of Actors, with library-support for further languages being available.

In an object-oriented setting, message sending can be encoded by asynchronous method calls and the handlers as the corresponding methods. Object-orientation also adds a further possibility for interaction: processes within one Actor may communicate by storing and reading data in the objects *heap memory*. However, there is still no preemption, so the points where interactions happen are clear, as they are explicit in the syntax.

Active Object languages are object-oriented languages that enforce the Actor concurrency model. Additionally, modern Active Object languages such as ABS [75], Encore [15] or ProActive [25] use *futures* [3, 65, 125, 99]. A future is an identifier for a method call that allows a process to synchronize on the called process and read its return value. Futures simplify the sending of a result back to the caller Actor. Instead of sending it to a second method via a callback, a downside of the pure Actor model, the result may now be handled in the same method by synchronizing on the called process. However, futures can cause Active Objects to deadlock by circular synchronization on futures.

Finally, Creol [76] and ABS introduced *cooperative scheduling*: a process may be descheduled, but only at a special **await** statement. The statements between two such suspension points still have exclusive access to the objects heap memory, data races may only occur at method start and **await** statements. The **await** statement has a guard that describes when the process may be rescheduled: the future guard suspends until a future is resolved, the condition guards suspends until a boolean expression evaluates to true.

We define the *Core Active Object* language (CAO) in this chapter, an Active Object language with futures, consequent strong encapsulation (i.e., all fields are object-private) and cooperative scheduling. For a further discussion of Active Object languages we refer to the survey of de Boer et al. [37].

3.1 Syntax

Definition 3.1 (Syntax)**.** *Let* v *range over program variables,* f *over field names,* I *over interface names,* C *over class names,* m *over method names,* p *over the set of program-point identifiers* P *and* n *over* $\mathbb{N}$ *and* $\sim$ *range over* &&, ||, +, -, *, /. *The* syntax *of* CAO *is defined by the grammar in Fig. 3.1. We use* $\overrightarrow{\cdot}$ *to denote (possibly empty) lists.*

A CAO program consists of a set of interfaces, set of classes and a main block. The main block is a list of object instantiations and one final method call to one of the instantiations to initialize the

Prgm ::= $\overrightarrow{\text{Inter}}$ $\overrightarrow{\text{Class}}$ Main Class ::= **class** C [**implements** I $\overrightarrow{\text{I}}$](I $\overrightarrow{\text{f}}$){$\overrightarrow{\text{Field}}$ $\overrightarrow{\text{Meth}}$}

Inter ::= **interface** I [**extends** I $\overrightarrow{\text{I}}$]{$\overrightarrow{\text{MSig}}$} MSig ::= D m($\overrightarrow{\text{D v}}$) Meth ::= MSig{s}

Main ::= **main**{si} Field ::= D f = e; D ::= Rat | Unit | Int | Bool | List<D> | Fut<D>

si ::= I v = C($\overrightarrow{\text{v}}$); si | v!m($\overrightarrow{\text{e}}$)

s ::= [D] l = e | [D] v = e.get$_p$ | **await** g$_p$ | [D] v = f!m($\overrightarrow{\text{e}}$) | **skip** | **return** e
| **while**(e){s} | **if**(e){s}**else**{s} | s;s

e ::= l | n | unit | Never | Nil | True | False | len(e) | hd(e) | tl(e) | Cons(e,e) | e ∼ e | !e | -e

l ::= **this**.f | v g ::= e | e?

Figure 3.1: Syntax of CAO.

communication. The object instantiations bind the created object to a name and take as parameters references to other objects. All objects are created at once and thus the order of object creations is not important. E.g., the following initializes a and b with references to each other.

```
main{ I a = C(b); I b = C(a); a!m(); }
```

An interface consists of a set of method signatures and may extend a set of interfaces. A class has (1) parameters, a list of references to other objects, (2) a list of fields which are initialized upon creation and (3) a list of methods. The references are regarded as fields that cannot be reassigned. Additionally, a class may implement a set of interfaces. Only the methods in the implemented interface may be called from other objects, other methods are *internal* and may only be called from the same object. CAO does not support inheritance. There are six kinds of data types: Rationals, the Unit type, Integers, Booleans, parametric lists and parametric futures. The expressions for Booleans and Integers are straightforward. `Never` is a unique future that is never resolved, `Nil` is the empty list, `hd` returns the first element of a list and `tl` the remainder. `Cons` is the list constructor and `len` returns the length of a list. The constant `unit` is the sole value of type `Unit`.

The statements for assignment, branching, repetition and the empty statement are standard. The statement `v = f!m(`$\overrightarrow{\text{e}}$`)` is an asynchronous method call on method `m` on object `f` with parameters $\overrightarrow{\text{e}}$. A fresh future is generated and stored in program variable `v`. This future identifies the called process. We say that the called process will *resolve* this future. The statement **return** `e` terminates the process and stores the value of the expression `e` in the future it is resolving. The statement `v = e.`**get** synchronizes on the future in `e`. Once this future is resolved, the result stored in it is written into `v`. Until the future is resolved, the process blocks the object. The statement **await** `g` suspends the current process until the guard is resolved. A future guard `e?` becomes active, once the future stored in `e` becomes active. A boolean guard `e` becomes active once the boolean expression `e` evaluates to true. Once the guard is active, the process *may* be rescheduled, but if several process may be scheduled, one is chosen non-deterministically. The **await** and **get** statements have a program-point identifier `p`. This identifier is used later by the specification to distinguish multiple suspension and synchronization points. Synchronizations and method calls are only allowed to write into variables to keep the number of semantic rules low.

The treatment of methods is standard, e.g., as in Java, but parameters cannot be reassigned. We only consider CAO programs which fulfill the following constraints:

- They are data type checked. In particular, the expression in field initialization is not allowed to contain other fields.[1]

[1] We give no data type system: the system from [75] can be applied to the classes and checking the main block is trivial. We use the term “data type checking” to distinguish it from checking behavioral types.

```
interface IClient {
  Unit compute(List<Int> values);
}
class Client implements
      IClient(IServer server){
  Int res = 0;
  Unit compute(List<Int> values){
    List<Int> val = values;
    while(val != Nil){
      Int v = hd(val);
      Fut<Int> f = server!cmp(v);
      await f?;
      res = f.get;
      val = tl(val);
    }
  }
}
```

```
interface IServer {
    Int cmp(Int next);
}

class Server implements IServer{
  Rat intVal = 0;
  Rat cmp(Int next){
    this.intVal = this.intVal + next;
    return this.intVal;
  }
}

main{
  IServer sum = new Server();
  IClient c = new Client(sum);
  c!compute(Cons(1,Cons(2,Cons(3,Nil))));
}
```

Figure 3.2: Distributed sum in CAO.

- Each method has exactly one **return** statement, which is the very last statement of the method body.
- Each branch of the **if** statement and the loop body of **while** end in a **skip** statement.
- Variable, parameter and field names, as well as program-point identifiers are program-wide unique.

We use **skip** to (1) mark the end of branches and loop bodies and (2) have empty computations. Marking the end of loop bodies allows us to have less rules in later program logic calculi: we do not need to distinguish the cases where a statement is the final statement or not.

Convention 1. *We drop the final* **skip** *statement, empty* **else** *branches and (if not needed) the program-point identifiers in examples.*

Example 3.1. *Fig. 3.2 sets up a server* `sum` *that computes the sum of the parameters passed to* `cmp`*, and a client* `c` *sending a list of values to it. After termination, the value of* `c.res` *is the sum of the input list.*

3.2 Locally Abstract Semantics

We use a locally abstract, globally concrete (LAGC) semantics [43] for CAO. A LAGC semantics consists of two layers: A locally abstract (LA) layer for expressions, statements and methods, and a globally concrete (GC) layer for objects and systems. The LA layer is a denotational semantics that abstractly describes the behavior of a method in every possible context, while the GC layer is an operational semantics that concretizes the abstract behavior of the processes to a concrete context. LAGC semantics allows us to analyze a method in isolation and to precisely express possible context by concretizing the most abstract behavior bit by bit.

The semantics of a method is a set of *symbolic* traces, which all together describe the behavior of the method in every possible context, i.e., for every possible object heap, call parameters and accessed futures. Symbolic traces contain symbolic values and symbolic expressions, additionally to semantic values (a semantic value is e.g., an integer stored in a variable of `Int` type). Symbolic values have no operations defined on them, instead they act as placeholders. They are replaced by semantic values once the method is running and the object heap, call parameters etc. are known.

We first define the semantic values, i.e., the values occurring in a concrete run of a method. Additionally to method names, values of the data types etc., semantic values can be *semantic effects*.

In general, a semantic effect is some identifier of some action of the program, which may not be visible in the semantics otherwise. In this work, we use effects which are pairs of fields or program point identifiers, and effect (R for read access, W for write access, S for synchronization and E for execution). Given a program Prgm with n object creations in its main block, the set $\mathcal{X} = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ is the set of object names. $\mathtt{x}_i$ is the object name of the ith object creation. P is the set of all program-point identifiers (PPI).

Definition 3.2 (Semantic Values)**.** *The set of* semantic effects *is defined as*

$$\left\{l_{\mathsf{op}} \mid l \text{ is a field or variable }, \mathsf{op} \in \{\mathsf{R}, \mathsf{W}\}\right\} \cup \left\{\mathtt{p}_{\mathsf{E}} \mid \mathtt{p} \in \mathsf{P}\right\} \cup \left\{\mathtt{p}_{\mathsf{S}} \mid \mathtt{p} \in \mathsf{P}\right\}$$

The set of semantic values *contains all semantic effects, the unit value* **unit***, the boolean values* **True***,* **False***, the natural numbers n, all object names, all method names, all program-point identifiers and all future identities, including* **never***. Additionally, sequences and sets of semantic values are also semantic values.*

Futures are only identities, not containers. We say that a future contains a value, but this relation is established by the traces, not by some explicit state of the future. We distinguish between four semantic effects: $\mathtt{p}_{\mathsf{S}}$ models that the synchronization point p was executed and $\mathtt{p}_{\mathsf{E}}$ that the suspension point p was executed. l_{op} models the read (or write) of a location. A statement may have multiple effects, e.g., the statement `v = head(this.list).get`$_1$ has the effects $\{\mathtt{list}_{\mathsf{R}}, 1_{\mathsf{S}}\}$.

Effects may not be derivable from the semantics when considering only the stores, e.g., the assignment `this.f = this.f` has no effect on the memory yet still accesses memory. Similarly, the effects may not be precisely derivable syntactically. Consider the following statement.

```
Int i = 0; if(i>0){this.f = 0}else{this.g = 0}skip
```

Its effects are $\{\mathtt{g}_{\mathsf{R}}\}$, but in more complex situations it may not be statically decidable whether the effects are $\{\mathtt{g}_{\mathsf{R}}\}$ or $\{\mathtt{f}_{\mathsf{R}}\}$.

3.2.1 Semantics of Expressions

Symbolic expressions contain symbolic values and semantic values, but while their structure mirrors the syntax of CAO-expressions, they do not contain any reference to state, i.e., no variables, fields or constants. Instead, they contain symbolic values and symbolic fields. Symbolic fields act like symbolic values, but contain the name of the field they are abstracting and a counter. This is needed to later substitute a concrete value in a trace up to the next point where the heap may change.

Definition 3.3 (Symbolic Values and Expressions)**.** *Let i range over $\mathbb{N}$. A* symbolic expression $\underline{\mathtt{e}}$ *is defined by the grammar below, analogously to the grammar of Def. 3.1, with semantic values v, symbolic values $\underline{s}$ and symbolic fields* `this.`$\underline{\mathtt{f}_i}$ *instead of variables, fields or constants.*

$$\underline{\mathtt{e}} ::= \mathtt{len}(\underline{\mathtt{e}}) \mid \mathtt{hd}(\underline{\mathtt{e}}) \mid \mathtt{tl}(\underline{\mathtt{e}}) \mid \mathtt{Cons}(\underline{\mathtt{e}},\underline{\mathtt{e}}) \mid \underline{\mathtt{e}} \sim \underline{\mathtt{e}} \mid \mathtt{!}\underline{\mathtt{e}} \mid \mathtt{-}\underline{\mathtt{e}} \mid \underline{s} \mid v \mid \underline{\mathtt{this.f}_i}$$

To clearly separate symbolic values and fields from other expressions, we always denote them underlined, e.g., $\underline{s}$ is a symbolic value and $\underline{s} + 1$ is a symbolic expression. In contrast, `1+1` is a syntactic expression and 2 a semantic value. We stress that semantic values are a subset of symbolic expressions. Syntactic expressions and symbolic expressions share only their operators.

The local state of a method and the heap of an object are functions from variable names (or field names) to symbolic expressions.

$$[\![\texttt{len(e)}]\!]^{\sigma}_{\rho} = \begin{cases} \left[\!\left[\left|[\![\texttt{e}]\!]^{\sigma}_{\rho}\right|\right]\!\right] & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is a semantic value} \\ \texttt{len(}[\![\texttt{e}]\!]^{\sigma}_{\rho}\texttt{)} & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is symbolic} \end{cases}$$

$$[\![\texttt{-e}]\!]^{\sigma}_{\rho} = \begin{cases} \left[\!\left[-[\![\texttt{e}]\!]^{\sigma}_{\rho}\right]\!\right] & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is a semantic value} \\ -[\![\texttt{e}]\!]^{\sigma}_{\rho} & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is symbolic} \end{cases}$$

$$[\![\texttt{!e}]\!]^{\sigma}_{\rho} = \begin{cases} \left[\!\left[!\,[\![\texttt{e}]\!]^{\sigma}_{\rho}\right]\!\right] & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is a semantic value} \\ !\,[\![\texttt{e}]\!]^{\sigma}_{\rho} & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is symbolic} \end{cases}$$

$$[\![\textbf{this}\texttt{.f}]\!]^{\sigma}_{\rho} = \rho(\texttt{f}) \quad [\![\texttt{v}]\!]^{\sigma}_{\rho} = \sigma(\texttt{v})$$

$$[\![\texttt{n}]\!]^{\sigma}_{\rho} = n \quad [\![\texttt{Nil}]\!]^{\sigma}_{\rho} = \langle\rangle$$

$$[\![\texttt{True}]\!]^{\sigma}_{\rho} = \textbf{True}$$

$$[\![\texttt{False}]\!]^{\sigma}_{\rho} = \textbf{False}$$

$$[\![\texttt{Never}]\!]^{\sigma}_{\rho} = \textbf{never}$$

$$[\![\texttt{unit}]\!]^{\sigma}_{\rho} = \textbf{unit}$$

$$[\![\texttt{Cons(e,e}')]\!]^{\sigma}_{\rho} = \left\langle [\![\texttt{e}]\!]^{\sigma}_{\rho} \right\rangle \circ [\![\texttt{e}']\!]^{\sigma}_{\rho}$$

$$[\![\texttt{hd(e)}]\!]^{\sigma}_{\rho} = \begin{cases} \left[\!\left[[\![\texttt{e}]\!]^{\sigma}_{\rho}\texttt{[1]}\right]\!\right] & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is a semantic value and not empty} \\ \texttt{hd(}[\![\texttt{e}]\!]^{\sigma}_{\rho}\texttt{)} & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is symbolic} \end{cases}$$

$$[\![\texttt{tl(e)}]\!]^{\sigma}_{\rho} = \begin{cases} \left[\!\left[[\![\texttt{e}]\!]^{\sigma}_{\rho}\texttt{[2..]}\right]\!\right] & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is a semantic value and not empty} \\ \texttt{tl(}[\![\texttt{e}]\!]^{\sigma}_{\rho}\texttt{)} & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ is symbolic} \end{cases}$$

$$[\![\texttt{e} \sim \texttt{e}']\!]^{\sigma}_{\rho} = \begin{cases} \left[\!\left[[\![\texttt{e}]\!]^{\sigma}_{\rho} \sim [\![\texttt{e}']\!]^{\sigma}_{\rho}\right]\!\right] & \text{if } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ and } [\![\texttt{e}']\!]^{\sigma}_{\rho} \text{ are semantic values} \\ & \text{and no division by zero occurs} \\ [\![\texttt{e}]\!]^{\sigma}_{\rho} \sim [\![\texttt{e}']\!]^{\sigma}_{\rho} & \text{if either } [\![\texttt{e}]\!]^{\sigma}_{\rho} \text{ or } [\![\texttt{e}']\!]^{\sigma}_{\rho} \text{ are symbolic} \end{cases}$$

Figure 3.3: Locally Abstract Semantics of Expressions. $[\![\cdot]\!]$ without state denotes natural evaluation.

Definition 3.4 (Local State and Heap)**.** *Given a class* C*, a heap* ρ *is a partial function from the fields and parameters of* C *to symbolic expressions. Given a method* C.m*, a local state* σ *is a partial function from the variables of* C.m *to symbolic expressions. A local state or heap is* concrete *if all elements of its image are semantic values. We call a pair of a local state* σ *and a heap* ρ *an* object state *and write* $\binom{\sigma}{\rho}$*. If the local state is empty, we write* $\binom{\cdot}{\rho}$*.*

We denote the heap that maps every field f to the symbolic expression $\underline{\textbf{this}\texttt{.f}}_i$ and every object parameter according to ρ with $\rho@i$. This is used to model a heap that has unknown ,possibly new values.

We are now able to give a semantics to syntactic expressions, by evaluating them under a given local state and a heap to a symbolic expression. The evaluation $[\![\texttt{e}]\!]$ on semantic values (and operations on them) has its natural definition and we identify the operators of $\sim$ with their natural counterpart.

Definition 3.5 (Semantics of Expressions)**.** *Let* e *be an expression and* $\binom{\sigma}{\rho}$ *an object state. The* evaluation function $[\![\texttt{e}]\!]^{\sigma}_{\rho}$ *is defined in Fig. 3.3. The evaluation function is not defined if division by zero occurs or the head or tail of an empty list is accessed.*

The semantics of expressions is well-defined, i.e., given a syntactic expression the semantics produce a symbolic expression. Additionally, the semantics ensures that if a symbolic expression contains no symbolic value, then it is fully evaluated to a semantic value.

Lemma 3.1. *Let* e *be an expression,* σ *a local state and* ρ *a heap, such that* $\underline{\texttt{e}} = [\![\texttt{e}]\!]^{\sigma}_{\rho}$ *is defined. If* σ *and* ρ *contain no symbolic values, then* $\underline{\texttt{e}}$ *is a semantic value.*

Proof. See p. 166.

Example 3.2. *Consider the expression* `this.f+i+hd(Cons(2,Nil))`*, and the following states and heap for a class with no parameters and a field* `f` *of* `Int` *data type:*

$$\sigma = \{\mathtt{i} \mapsto 1\} \qquad \sigma' = \{\mathtt{i} \mapsto \underline{i}\} \qquad \rho = \{\mathtt{f} \mapsto 2\}$$

Evaluation with a concrete object state results in a semantic value:

$$[\![\mathtt{this.f+i+hd(Cons(2,Nil))}]\!]_{(\rho}^{\sigma)} = [\![\mathtt{this.f}]\!]_{(\rho}^{\sigma)} + [\![\mathtt{i}]\!]_{(\rho}^{\sigma)} + [\![\mathtt{hd(Cons(2,Nil))}]\!]_{(\rho}^{\sigma)} = 2 + 1 + (\langle 2\rangle \circ \langle\rangle)[1] = 5$$

Evaluation with concrete local state and symbolic heap results in a partially evaluated symbolic expression.

$$[\![\mathtt{this.f+i+hd(Cons(2,Nil))}]\!]_{(\rho@1}^{\sigma)} = [\![\mathtt{this.f}]\!]_{(\rho@1}^{\sigma)} + [\![\mathtt{i}]\!]_{(\rho@1}^{\sigma)} + [\![\mathtt{hd(Cons(2,Nil))}]\!]_{(\rho@1}^{\sigma)} = \underline{\mathtt{this.f}}_1 + 1 + (\langle 2\rangle \circ \langle\rangle)[1] = \underline{\mathtt{this.f}}_1 + 3$$

Evaluation with a fully symbolic object state results also in a partially evaluated symbolic expression.

$$[\![\mathtt{this.f+i+hd(Cons(2,Nil))}]\!]_{(\rho@1}^{\sigma'} = [\![\mathtt{this.f}]\!]_{(\rho@1}^{\sigma'} + [\![\mathtt{i}]\!]_{(\rho@1}^{\sigma'} + [\![\mathtt{hd(Cons(2,Nil))}]\!]_{(\rho@1}^{\sigma'} = \underline{\mathtt{this.f}}_1 + \underline{i} + (\langle 2\rangle \circ \langle\rangle)[1] = \underline{\mathtt{this.f}}_1 + \underline{i} + 2$$

3.2.2 Semantics of Statements and Methods

$$\begin{aligned} \mathsf{ev} ::= \quad & \mathsf{invEv}(\underline{\mathtt{X}}, \underline{\mathtt{X}'}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{\mathtt{e}}}, \overrightarrow{\mathsf{eff}}) \mid \mathsf{invREv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{\mathtt{e}}}, \overrightarrow{\mathsf{eff}}) \mid \mathsf{futEv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \underline{\mathtt{e}}, \overrightarrow{\mathsf{eff}}) \mid \mathsf{futREv}(\underline{\mathtt{X}}, \underline{\mathtt{e}}, \underline{\mathtt{m}}, \underline{\mathtt{e}}, \overrightarrow{\mathsf{eff}}) \\ & \mathsf{condEv}(\underline{\mathtt{X}}, \underline{f}, \overrightarrow{\mathsf{eff}}) \mid \mathsf{condREv}(\underline{\mathtt{X}}, \underline{f}, \overrightarrow{\mathsf{eff}}) \mid \mathsf{suspEv}(\underline{\mathtt{X}}, \underline{f}, \underline{\mathtt{e}}, \overrightarrow{\mathsf{eff}}) \mid \mathsf{suspREv}(\underline{\mathtt{X}}, \underline{f}, \underline{\mathtt{e}}, \overrightarrow{\mathsf{eff}}) \mid \mathsf{noEv}(\overrightarrow{\mathsf{eff}}) \end{aligned}$$

Figure 3.4: Syntax of Events.

The traces in the semantics of a method contain not only states, but also events: markers for visible communication (and the event noEv for uniformity). In the globally concrete semantics, the events are used to merge the local traces of a method first into the local trace of an object and then into a global trace of the whole system.

Definition 3.6 (Events)**.** *Let* eff *range over semantic effects,* $\underline{f}$ *over symbolic futures,* $\underline{\mathtt{X}}$ *over symbolic object names and* $\underline{\mathtt{m}}$ *over symbolic method names. Each type of event is paired with a reaction event, which either models that the communication has had its effect on the other party, or that the process has regained control. Fig. 3.4 shows the syntax of* events*.*

Every event has a sequence of effects as a parameter. These are the effects recorded during the execution of the statement that caused the event. We ignore the order of this sequence and consider it a set.

- *The invocation event* $\mathsf{invEv}(\underline{\mathtt{X}}, \underline{\mathtt{X}'}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{\mathtt{e}}}, \overrightarrow{\mathsf{eff}})$ *models a call from* $\underline{\mathtt{X}}$ *to* $\underline{\mathtt{X}'}$ *on method* `m`*. The future* $\underline{f}$ *is used and* $\overrightarrow{\underline{\mathtt{e}}}$ *are the call parameters. The future and the objects may be symbolic, because without global information it is not possible to know what future will be used, what object will be called and what object the local semantics are generated for.*

- *The invocation reaction event* $\mathsf{invREv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{\mathtt{e}}}, \overrightarrow{\mathsf{eff}})$ *is the callee view on the method call. The object* $\underline{\mathtt{X}}$ *here is the callee, the caller is not visible to the callee.*

- *The resolving event* $\mathsf{futEv}(\underline{\mathtt{X}}, \underline{f}, \mathtt{m}, \underline{\mathtt{e}}, \overrightarrow{\mathsf{eff}})$ *models the termination of a process for future* $\underline{f}$ *that computes method* `m` *in object* $\underline{\mathtt{X}}$ *and returns* $\underline{\mathtt{e}}$*.*

- *The resolving reaction event* $\mathsf{futREv}(\underline{x}, \underline{e}, \underline{m}, \underline{e'}, \overrightarrow{\mathsf{eff}})$ *models the read of a* **get** *statement in object* $\underline{x}$ *on the future* $\underline{e}$*, reading value* $\underline{e'}$*. The future was originally resolved by a process computing* $\underline{m}$*.*
- *The condition synchronization event* $\mathsf{condEv}(\underline{x}, \underline{f}, \overrightarrow{\mathsf{eff}})$ *models that the process computing the future* $\underline{f}$ *in object* $\underline{x}$ *suspends. The condition synchronization reaction event* $\mathsf{condREv}(\underline{x}, \underline{f}, \overrightarrow{\mathsf{eff}})$ *is analogous for continuing execution.*
- *The suspension event* $\mathsf{suspEv}(\underline{x}, \underline{f}, \underline{e}, \overrightarrow{\mathsf{eff}})$ *models that the process computing the future* $\underline{f}$ *in object* $\underline{x}$ *suspends and* $\underline{e}$ *is the future which is read.*
- *The suspension reaction event is analogous to the condition synchronization reaction event.*
- *Finally,* $\mathsf{noEv}(\overrightarrow{\mathsf{eff}})$ *models a step without visible communication.*

We say that $\mathsf{invEv}(\underline{x}, \underline{x'}, \underline{f}, \mathtt{m}, \overrightarrow{\underline{e}}, \overrightarrow{\mathsf{eff}})$ *introduces* $\underline{f}$ *and that* $\mathsf{futREv}(\underline{x}, \underline{e}, \underline{m}, \underline{e'}, \overrightarrow{\mathsf{eff}})$ *introduces* $\underline{e'}$ *and* $\underline{m}$*.*

Convention 2. *We omit the effect set parameter of events in examples where it is not relevant.*

Convention 3. *We omit all parameters of events in examples, if they are not relevant.*

In the globally concrete semantics of a program, the object parameters of the events, as well as the computed future, are not symbolic. They are, however, symbolic when we analyze a method in isolation.

Local traces consist of two parts. (1) A selection condition[2], a set of symbolic expressions that expresses when a trace can be selected to execute the next step and (2) a history, a sequence of events and object states.

Definition 3.7 (Local Traces)**.** *A local trace* θ *has the form* $\mathsf{sc} \triangleright \mathsf{hs}$*, where* sc *is a set of symbolic expressions, called* selection condition*, and* hs *is a non-empty sequence, called* history*.*

The history has odd length starts with a state and alternates between object states at odd positions and events (and the special symbol $\diamond$*) at even positions.*

The marker $\diamond$ models that at this point in the trace another process may run, but it is no event and has no semantic effects. We use two chopping operators to connect histories and traces.

Definition 3.8 (Chops)**.** *The standard* $**$ *merges two histories if they share the last and first object state [27, 106]. The extended chop* $***$ *operates like* $**$ *if the standard chop is defined. If only the heaps are equal, it concatenates the histories with a marker* $\diamond$*. Neither is defined if the heaps are different. Both chops are lifted to traces by chopping the histories and joining the selection conditions.*

$$\mathsf{hs} \circ \left\langle \begin{pmatrix} \sigma \\ \rho \end{pmatrix} \right\rangle ** \left\langle \begin{pmatrix} \sigma' \\ \rho' \end{pmatrix} \right\rangle \circ \mathsf{hs}' = \begin{cases} \mathsf{hs} \circ \left\langle \begin{pmatrix} \sigma \\ \rho \end{pmatrix} \right\rangle \circ \mathsf{hs}' & \text{if } \begin{pmatrix} \sigma \\ \rho \end{pmatrix} = \begin{pmatrix} \sigma' \\ \rho' \end{pmatrix} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{hs} \circ \left\langle \begin{pmatrix} \sigma \\ \rho \end{pmatrix} \right\rangle *** \left\langle \begin{pmatrix} \sigma' \\ \rho' \end{pmatrix} \right\rangle \circ \mathsf{hs}' = \begin{cases} \mathsf{hs} \circ \left\langle \begin{pmatrix} \sigma \\ \rho \end{pmatrix} \right\rangle \circ \mathsf{hs}' & \text{if } \begin{pmatrix} \sigma \\ \rho \end{pmatrix} = \begin{pmatrix} \sigma' \\ \rho' \end{pmatrix} \\ \mathsf{hs} \circ \left\langle \begin{pmatrix} \sigma \\ \rho \end{pmatrix}, \diamond, \begin{pmatrix} \sigma' \\ \rho' \end{pmatrix} \right\rangle \circ \mathsf{hs}' & \text{if } \rho = \rho', \sigma \neq \sigma' \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\left((\mathsf{sc} \triangleright \mathsf{hs}) ** (\mathsf{sc}' \triangleright \mathsf{hs}')\right) = (\mathsf{sc} \cup \mathsf{sc}') \triangleright (\mathsf{hs} ** \mathsf{hs}')$$

$$\left((\mathsf{sc} \triangleright \mathsf{hs}) *** (\mathsf{sc}' \triangleright \mathsf{hs}')\right) = (\mathsf{sc} \cup \mathsf{sc}') \triangleright (\mathsf{hs} *** \mathsf{hs}')$$

Before defining the semantics of statements and methods, we require some technical definitions.

[2] In a symbolic execution context this is called the path condition, but as we use it for selection and do not use paths, we adopt a slightly different terminology here.

Definition 3.9 (Freshness and Symbolic State)**.** *A symbolic value $\underline{s}$ is fresh, if it occurs nowhere else*[3] *A heap counter i is fresh, if no symbolic field with index i occurs anywhere else. A heap (or local state) is symbolic if it maps some field (or variable) to a symbolic expression that is not a semantic value. A heap (or local state) is fully symbolic if it maps every field (or variable) to a symbolic expression that is not a semantic value.*

To define the effects, we use the auxiliary function $\mathsf{loc}(\mathtt{e})$ which returns the set of all fields within $\mathtt{e}$. The set $\mathsf{loc}_R(\mathtt{e})$ is defined as the read effects of all fields and variables within $\mathtt{e}$:

$$\mathsf{loc}_R(\mathtt{e}) = \{l_R \mid l \in \mathsf{loc}(\mathtt{e})\}$$

We add two additional local variables to the local variables of all methods: **old** and **last** are variables of `Heap` type (which is not declarable otherwise) which are not accessible by syntactic means. Instead, they keep track of the heap when the process starts (**old**) or when it was scheduled the last time (**last**). A variable of `Heap` type maps field names to symbolic expressions. Keeping track of old heap states in implicit program variables has proven useful for specification in JavaDL and ABSDL, for `CAO` we must handle them explicitly when giving the semantics of the language.

Definition 3.10 (Local Semantics for Methods)**.** *The* semantics *of a method $\mathtt{m}$ with method body $\mathtt{s}$ is defined by a function $[\![\mathtt{m}]\!]^{\vec{\mathtt{e}}}_{\mathtt{X},dest,\mathtt{m},\rho}$ where $\mathtt{X}$ is the object name, dest the future the method is resolving, $\mathtt{m}$ the method name, ρ the current object heap and $\vec{\mathtt{e}}$ the call parameters. .Everything but the method name may be symbolic.*

$$[\![\mathtt{m}]\!]^{\{\langle \mathtt{e}_1,\ldots,\mathtt{e}_n\rangle\}}_{\mathtt{X},dest,\mathtt{m},\rho} = \left\{ \emptyset \triangleright \left\langle \binom{\sigma'}{\rho}, \mathsf{invREv}(\mathtt{X}, dest, \mathtt{m}, \langle \mathtt{e}_1,\ldots,\mathtt{e}_n\rangle, \emptyset), \binom{\sigma'}{\rho} \right\rangle ** \theta \mid \theta \in [\![\mathtt{s}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma'}{\rho}} \right\}$$

The method parameters are extracted from the parameter names into σ, e.g., if the signature of a method is `Int m(Int a, Rat b)` *and the parameters are $\langle 1, 2\rangle$ then*

$$\sigma = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2\}$$

The local state σ' is defined as $\sigma' = \sigma[\textbf{old} \mapsto \rho][\textbf{last} \mapsto \rho]$.

Definition 3.11 (Local Semantics for Statements)**.** *The* semantics *of statements is defined by a function $[\![\cdot]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}}$ where $\mathtt{X}$ is the object name, dest the future the method is resolving, $\mathtt{m}$ the method name and $\binom{\sigma}{\rho}$ the current object state. Fig. 3.5 shows the rules for statements. If the semantics of an expression is not defined, then neither is the semantics of a statement.*

- *The rule for assignment of variables updates the local state, adds a* noEv *event and is followed by the traces of the following statement, generated with the updated store.*
- *The rule for variable declaration just skips over the declaration and only evaluates the initialization. The local state has all local variables in its domain, the type system ensures that the variables are not used before the declaration.*
- *The rule for assignment of fields is analogous, but updates the heap instead of the local state.*
- *The rule for branching adds the guard to the selection condition of the first branch and its negation to the selection condition of the second branch.*

[3] To be more precise: nowhere in the state of the system, which we define in the next section. We refrain from introducing local and global freshness, as it offers no insights.

$$[\![\texttt{v = e}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{noEv}(\{\mathtt{v}_\mathsf{W}\} \cup \mathsf{loc}_\mathsf{R}(\texttt{e})), \binom{\sigma[\texttt{v} \mapsto [\![\texttt{e}]\!]_{\binom{\sigma}{\rho}}]}{\rho} \right\rangle \right\}$$

$$[\![\texttt{D v = e}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} = [\![\texttt{v = e}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}}$$

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

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

$$\begin{aligned}[\![\textbf{if}\texttt{(e)\{s\}}\textbf{else}\texttt{\{s'\}}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} = \quad & \left\{\{[\![\texttt{e}]\!]_{\binom{\sigma}{\rho}}\} \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{noEv}(\mathsf{loc}_\mathsf{R}(\texttt{e})), \binom{\sigma}{\rho} \right\rangle ** \theta \mid \theta \in [\![\texttt{s}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} \right\} \\ & \cup \left\{\{[\![\texttt{!e}]\!]_{\binom{\sigma}{\rho}}\} \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{noEv}(\mathsf{loc}_\mathsf{R}(\texttt{e})), \binom{\sigma}{\rho} \right\rangle ** \theta \mid \theta \in [\![\texttt{s'}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} \right\}\end{aligned}$$

$$[\![\texttt{v = e.}\textbf{get}_i]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} =$$

$$\left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{futREv}\left(\mathtt{X}, [\![\texttt{e}]\!]_{\binom{\sigma}{\rho}}, \underline{\mathtt{n}}, \underline{v}, \{i_\mathsf{E}, \mathtt{v}_\mathsf{W}\} \cup \mathsf{loc}_\mathsf{R}(\texttt{e})\right), \binom{\sigma[\texttt{v} \mapsto \underline{v}]}{\rho} \right\rangle \right\}$$

where $\underline{\mathtt{n}}, \underline{v}$ are fresh

$$[\![\texttt{v = f!n(e}_1\texttt{,...,e}_n\texttt{)}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} =$$

$$\left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{invEv}\left(\mathtt{X}, [\![\texttt{f}]\!]_{\binom{\sigma}{\rho}}, \underline{f}, \mathtt{n}, \left\langle [\![\texttt{e}]\!]_{\binom{\sigma}{\rho}}, \ldots, [\![\texttt{e}_n]\!]_{\binom{\sigma}{\rho}} \right\rangle, \{\mathtt{v}_\mathsf{W}, \mathtt{f}_\mathsf{R}\} \cup \bigcup_{1..n} \mathsf{loc}_\mathsf{R}(\texttt{e}_i)\right), \binom{\sigma[\texttt{v} \mapsto \underline{f}]}{\rho} \right\rangle \right\}$$

where $\underline{f}$ is fresh

$$\begin{aligned}[\![\textbf{await}\ \texttt{e?p}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} = \quad & \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{suspEv}\left(\mathtt{X}, dest, [\![\texttt{e}]\!]_{\binom{\sigma}{\rho}}, \{\mathtt{p}_\mathsf{E}\} \cup \mathsf{loc}_\mathsf{R}(\texttt{e})\right), \binom{\sigma}{\rho}, \diamond \right\rangle \right. \\ & \left. \circ \left\langle \binom{\sigma[\textbf{last} \mapsto \rho@j]}{\rho@j}, \mathsf{suspREv}\left(\mathtt{X}, dest, [\![\texttt{e}]\!]_{\binom{\sigma}{\rho}}, \emptyset\right), \binom{\sigma[\textbf{last} \mapsto \rho@j]}{\rho@j} \right\rangle \right\}\end{aligned}$$

where j is fresh

$$\begin{aligned}[\![\textbf{await}\ \texttt{e}_\mathtt{p}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} = \quad & \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, \mathsf{condEv}\left(\mathtt{X}, dest, \{\mathtt{p}_\mathsf{E}\} \cup \mathsf{loc}_\mathsf{R}(\texttt{e})\right), \binom{\sigma}{\rho}, \diamond \right\rangle \right. \\ & \left. \circ \left\langle \binom{\sigma[\textbf{last} \mapsto \rho@j]}{\rho@j}, \mathsf{condREv}\left(\mathtt{X}, dest, \emptyset\right), \binom{\sigma[\textbf{last} \mapsto \rho@j]}{\rho@j} \right\rangle \right\}\end{aligned}$$

where j is fresh

$$[\![\textbf{skip}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\emptyset \triangleright \left\langle \binom{\sigma}{\rho} \right\rangle \right\}$$

$$[\![\texttt{s;s'}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} = \left\{\theta ** \theta' \mid \theta \in [\![\texttt{s}]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}} \wedge \theta' \in [\![\texttt{s'}]\!]_{\mathtt{X},dest,\mathtt{m},\mathsf{last}(\theta)} \right\}$$

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

Figure 3.5: Locally Abstract Semantics of Statements. The rules for variable initialization are always analogous to the rule for variable assignment.

- *The rule for* **get** *is similar to the variable assignment, but gets a fresh symbolic value and stores it in the local state. As the event, a resolving reaction event is added, which stores the accessed future and the fresh symbolic value. Note that the accessed future may also be symbolic.*
- *The rule for method calls is analogous, but uses a fresh future for the call instead of a fresh read value. The added event is an invocation event with the evaluated parameters.*
- *The two rules for* **await** *are the only ones that add a marker* $\diamond$*. In both rules, the next statement is evaluated with a fresh symbolic heap. I.e., all fields map to fresh symbolic fields. Additionally, the* **last** *variable is update and has the value of the heap before the suspension.*
- *The rule for* **skip** *generates a trace that can always be selected and consists solely of the current state.*
- *The rule for sequential composition composes the traces of its substatements with the chop operator.*
- *Finally,* **while** *is defined recursively.*

The rule for **while** turns the used equations into a fixpoint definition, but we refrain from formally introducing fix points, because we only analyze terminating systems with finite traces which can be obtained by an unbounded, but finite number of applications of the **while** rule. The formalism needed behind infinite traces can be found in [17].

Example 3.3. *Consider the following method*

```
1  Int m(Int p){
2      this.f = this.f + 1;
3      Fut<Int> fut = o!n();
4      Int i = fut.get1;
5      if(this.f > i){
6          await this.f < 02;
7          if(this.f < p){
8              this.f = 0;
9          }
10     }
11     return this.f;
12 }
```

The semantics for a call with parameter `5` *and resolving future dest in a class that has object* $\mathtt{x}'$ *stored in its parameter* `o` *are three traces, shown in Fig. 3.6. For readability's sake we omitted the parameters* `p` *(for which we directly substituted* `5`*) and* `o` *from the object state and the variable* **old** *and* **last** *from the local state, and give the effects only for* θ_1*.*

Definition 3.12 (Applying Heaps)**.** *Let* ρ *be a heap and* hs *a history. We write* $\mathsf{hs}\rho$ *for the history that results from applying* ρ*, i.e., replacing all symbolic fields* $\underline{\texttt{this.f}_i}$ *with* $\rho(\mathtt{f})$ *up to the first* $\diamond$*. Given a trace* θ*, we apply heaps with* $\theta\rho = \mathsf{sc}\rho \triangleright \mathsf{hs}\rho$*. The selection condition set* $\mathsf{sc}\rho$ *only substitutes for those symbolic fields* $\underline{\texttt{this.f}_i}$ *which were substituted in* $\mathsf{hs}\rho$*. The effect set parameters of events are not substituted. We assume that resulting semantic values are evaluated directly after substitution.*

Example 3.4. *Consider the trace* θ_1 *from Fig. 3.6 and the heap* $\rho = \{\mathtt{f} \mapsto 2\}$*.*

$$
\begin{aligned}
\theta_1\rho = \ & \{2 > \underline{i}, \underline{\texttt{this.f}_2} < 5\}\triangleright \\
& \left\langle \begin{pmatrix} \mathtt{i}\mapsto 0, \mathtt{fut}\mapsto \textbf{never} \\ \mathtt{f}\mapsto 2 \end{pmatrix}, \mathsf{invREv}(\mathtt{x}, \mathit{dest}, \mathtt{m}, \langle 5\rangle), \begin{pmatrix} \mathtt{i}\mapsto 0, \mathtt{fut}\mapsto \textbf{never} \\ \mathtt{f}\mapsto 2 \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \mathtt{i}\mapsto 0, \mathtt{fut}\mapsto \textbf{never} \\ \mathtt{f}\mapsto 3 \end{pmatrix}, \mathsf{invEv}(\mathtt{x}, \mathtt{x}', \underline{\mathit{fut}}, \mathtt{n}, \langle\rangle) \right\rangle \\
& \circ \left\langle \begin{pmatrix} \mathtt{i}\mapsto 0, \mathtt{fut}\mapsto \underline{\mathit{fut}} \\ \mathtt{f}\mapsto 3 \end{pmatrix}, \mathsf{futREv}(\mathtt{x}, \underline{\mathit{fut}}, \underline{i}), \begin{pmatrix} \mathtt{i}\mapsto \underline{i}, \mathtt{fut}\mapsto \underline{\mathit{fut}} \\ \mathtt{f}\mapsto 3 \end{pmatrix}, \mathsf{condEv}(\mathtt{x}, \mathit{dest}), \begin{pmatrix} \mathtt{i}\mapsto \underline{i}, \mathtt{fut}\mapsto \underline{\mathit{fut}} \\ \mathtt{f}\mapsto 3 \end{pmatrix}, \diamond \right\rangle \\
& \circ \left\langle \begin{pmatrix} \mathtt{i}\mapsto \underline{i}, \mathtt{fut}\mapsto \underline{\mathit{fut}} \\ \mathtt{f}\mapsto \underline{\texttt{this.f}_2} \end{pmatrix}, \mathsf{condREv}(\mathtt{x}, \mathit{dest}), \begin{pmatrix} \mathtt{i}\mapsto \underline{i}, \mathtt{fut}\mapsto \underline{\mathit{fut}} \\ \mathtt{f}\mapsto \underline{\texttt{this.f}_2} \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \mathtt{i}\mapsto \underline{i}, \mathtt{fut}\mapsto \underline{\mathit{fut}} \\ \mathtt{f}\mapsto 0 \end{pmatrix}, \mathsf{futEv}(\mathtt{x}, \mathit{dest}, \mathtt{m}, \underline{\texttt{this.f}_2}), \begin{pmatrix} \mathtt{i}\mapsto \underline{i}, \mathtt{fut}\mapsto \underline{\mathit{fut}} \\ \mathtt{f}\mapsto 0 \end{pmatrix} \right\rangle
\end{aligned}
$$

3.3 Globally Concrete Semantics

This section gives the globally concrete semantics of CAO. While the locally abstract semantics are a denotational semantics, the GC semantics are operational semantics, where a reduction function is

$$
\begin{aligned}
\theta_1 = \ & \{\underline{\texttt{this}.\texttt{f}_1} > \underline{i}, \underline{\texttt{this}.\texttt{f}_2} < 5\}\triangleright \\
& \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} \end{pmatrix}, \mathsf{invREv}(\texttt{x}, \mathit{dest}, \texttt{m}, \langle 5 \rangle, \emptyset), \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} \end{pmatrix}, \mathsf{noEv}(\{\texttt{f}_\texttt{R}, \texttt{f}_\texttt{W}\}) \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{invEv}(\texttt{x}, \texttt{x}', \underline{\mathit{fut}}, \texttt{n}, \langle\rangle, \{\texttt{o}_\texttt{R}, \texttt{fut}_\texttt{W}\}), \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{futREv}(\texttt{x}, \underline{\mathit{fut}}, \underline{\texttt{n}}, \underline{i}, \{\texttt{i}_\texttt{W}, \texttt{fut}_\texttt{R}, 1_\texttt{E}\}) \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{noEv}(\{\texttt{i}_\texttt{R}, \texttt{f}_\texttt{R}\}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{condEv}(\texttt{x}, \mathit{dest}, \{2_\texttt{E}, \texttt{f}_\texttt{R}\}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \diamond \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2} \end{pmatrix}, \mathsf{condREv}(\texttt{x}, \mathit{dest}, \{2_\texttt{E}, \texttt{f}_\texttt{R}\}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2} \end{pmatrix}, \mathsf{noEv}(\{\texttt{f}_\texttt{R}, \texttt{p}_\texttt{R}\}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2} \end{pmatrix}, \mathsf{noEv}(\{\texttt{f}_\texttt{W}\}) \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto 0 \end{pmatrix}, \mathsf{futEv}(\texttt{x}, \mathit{dest}, \texttt{m}, 0, \{\texttt{f}_\texttt{R}\}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto 0 \end{pmatrix} \right\rangle
\end{aligned}
$$

$$
\begin{aligned}
\theta_2 = \ & \{\underline{\texttt{this}.\texttt{f}_1} > \underline{i}, \underline{\texttt{this}.\texttt{f}_2} \geq 5\}\triangleright \\
& \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} \end{pmatrix}, \mathsf{invREv}(\texttt{x}, \mathit{dest}, \texttt{m}, \langle 5 \rangle), \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{invEv}(\texttt{x}, \texttt{x}', \underline{\mathit{fut}}, \texttt{n}, \langle\rangle) \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{futREv}(\texttt{x}, \underline{\mathit{fut}}, \underline{\texttt{n}}, \underline{i}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{condEv}(\texttt{x}, \mathit{dest}) \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \diamond, \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2} \end{pmatrix}, \mathsf{condREv}(\texttt{x}, \mathit{dest}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2} \end{pmatrix}, \mathsf{noEv} \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2} \end{pmatrix}, \mathsf{futEv}(\texttt{x}, \mathit{dest}, \texttt{m}, \underline{\texttt{this}.\texttt{f}_2}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2} \end{pmatrix} \right\rangle
\end{aligned}
$$

$$
\begin{aligned}
\theta_3 = \ & \{\underline{\texttt{this}.\texttt{f}_1} \leq \underline{i}\}\triangleright \\
& \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} \end{pmatrix}, \mathsf{invREv}(\texttt{x}, \mathit{dest}, \texttt{m}, \langle 5 \rangle), \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{invEv}(\texttt{x}, \texttt{x}', \underline{\mathit{fut}}, \texttt{n}, \langle\rangle), \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{futREv}(\texttt{x}, \underline{\mathit{fut}}, \underline{\texttt{n}}, \underline{i}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{noEv} \right\rangle \\
& \circ \left\langle \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix}, \mathsf{futEv}(\texttt{x}, \mathit{dest}, \texttt{m}, \underline{\texttt{this}.\texttt{f}_1} + 1), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{\mathit{fut}} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_1} + 1 \end{pmatrix} \right\rangle
\end{aligned}
$$

Figure 3.6: Example Semantics.

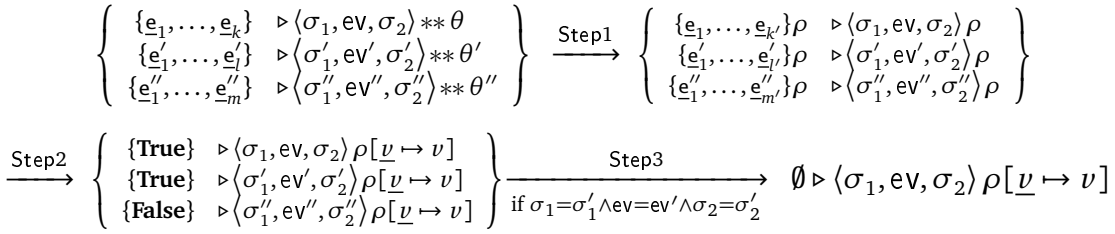


Figure 3.7: Structure of Agreement.

used to compute the next step in the computation. CAO has a two-layered GC semantics: there is a global semantics for objects and a global semantics for the whole system. The two semantics are not independent: the global system-semantics is defined using the object-semantics and ensures that the communication steps that the object makes are correct with respect to the whole system.

3.3.1 Semantics of Objects

Definition 3.13 (Object)**.** *An* object **O** *has the form* $\left(\mathsf{procpool}, \theta, \mathsf{proc}\right)_{\mathtt{X}}$*, where* $\mathtt{X}$ *is the object name,* proc *is the* active process*: a set of traces which may continue execution.* procpool *is the* process pool*, a set of processes, i.e., a set of sets of traces. One of the processes may be chosen to continue an execution once no process is active. Finally, the trace* θ *is the* object trace*. This is a local trace that models the execution of the object up to this point in time.*

The active process contains the traces that are *not* yet executed. The heap is not explicitly part of an object, instead the heap of the last object state of θ is the current heap.

Initially, all objects are initialized with an empty process pool, no active process and an object trace that has no local state and a heap initializing all fields to (semantic) default values. The object called by the main block has a non-empty process pool, with a single element: the semantics of the called method for some concrete future f and an accordingly instantiated local state.

Definition 3.14 (Initial Object)**.** *Let* $\mathtt{v!m}(\overrightarrow{\mathtt{e}})$ *be the initial call from the main block. The first* initial object *below is the initial object for all other objects, the second* initial object *for the called object.*

$$\left(\emptyset, \emptyset \triangleright \left\langle \begin{pmatrix} \cdot \\ \rho_i \end{pmatrix} \right\rangle, \emptyset\right)_{\mathtt{X}_i}$$

$$\left(\left\{ [\![\mathtt{m}]\!]_{\mathtt{X}, dest, \mathtt{m}, \rho_i} \right\}, \emptyset \triangleright \left\langle \begin{pmatrix} \sigma \\ \rho_i \end{pmatrix} \right\rangle, \emptyset\right)_{\mathtt{X}_i}$$

Where ρ_i *maps all fields to the default value of their data type (***unit***,* 0*,* **False***,* $\langle\rangle$*,* **never***) and the parameters according to the parameters of the object creation of* $\mathtt{X}_i$*. The heap* ρ_i *contains no symbolic expressions. The local state* σ *of the only process is initialized according to the parameters* $\overrightarrow{\mathtt{e}}$ *of the initial call. The variables* **old** *and* **last** *map to* ρ_i*.*

The GC semantics of objects is the point where the locally abstract traces are executed and concretized. This is done by *agreement*: the traces in a process agree on how to continue execution by generating a local trace *without symbolic values* that is appended to the object trace. This is one execution step. Informally, agreement is established in three steps, which are shown in Fig. 3.7 for an example with three traces.

1. For each trace in the process, a *candidate trace* is generated: The history of the candidates are the first three elements of the history of the trace. The selection condition of the candidate is the set of all expressions in the selection condition of the trace, which have either no symbolic values or only

symbolic values introduced by the event in the history of the candidate. Every trace has the same symbolic values. The heap of the current state is applied on this candidate trace. If a trace cannot generate a candidate, agreement fails. This is the case if the candidate traces after application of the heap contain a value that is not introduced by its event.

2. In the next step, the symbolic values are instantiated with some semantic values. The symbolic values are substituted with the semantic values and every expression is then fully evaluated, The candidate is then fully concrete.

3. In the final step, agreement is established. A candidate is agreed on, if it is the (concretized) candidate of all processes whose selection condition holds. I.e., all candidates who may be selected are identical. Then, the continuation process is computed: it is the suffix of all traces whose candidate was selected after the symbolic values are substituted.

Fig. 3.7 gives an informal overview how a process agrees on a trace: Step 1 generates the candidates. The sets $\{\underline{e}_1, \ldots, \underline{e}_{k'}\}$ are those expressions in the selection conditions, where the only symbolic values are occurring also in the candidates' history. Step 2 applies the heap. Step 3 ensures that all selectable candidates are equal. The continuation process is $\{\theta, \theta'\}$. Trace θ'' is discarded because its candidate is not selected. Before we give an example, we first introduce the required technical definitions.

Definition 3.15. *Let ρ be a heap. A trace $\theta = \mathsf{sc} \triangleright \mathsf{hs}$ is* selectable *if the natural evaluation of all expressions in the selection condition is* **True***:*

$$\forall \underline{e} \in \mathsf{sc}.\ [\![\underline{e}]\!] = \textbf{True}$$

*The ρ-*selection candidate *of a trace $\theta = \mathsf{sc} \triangleright \mathsf{hs}$ is a trace $\theta_C = \mathsf{sc}_C \triangleright \mathsf{hs}_C$ such that*

$\mathsf{hs}\rho = \mathsf{hs}_C \ast\ast \mathsf{hs}'$ *for some* hs' *and* $|\mathsf{hs}_C| = 3$*. If* hs_C *contains symbolic values, then it is introduced within it.*
$\mathsf{sc}_C = \{\underline{e} \in \mathsf{sc}\rho \mid \underline{e}$ *contains exactly the symbolic values introduced in* $\mathsf{hs}_C\}$

Example 3.5. *In Ex. 3.4, $\theta_1, \theta_2, \theta_3$ have the following $\{\mathtt{f} \mapsto 2\}$-selection candidates (ignoring parameter $\mathtt{o}$):*

$$\emptyset \triangleright \left\langle \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto \textbf{never} \\ \mathtt{f} \mapsto 2, \mathtt{o} \mapsto \mathtt{X}' \end{pmatrix}, \mathsf{invREv}(\mathtt{X}, \mathit{dest}, \mathtt{m}, \langle 5 \rangle, \emptyset), \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto \textbf{never} \\ \mathtt{f} \mapsto 2, \mathtt{o} \mapsto \mathtt{X}' \end{pmatrix} \right\rangle$$
$$\emptyset \triangleright \left\langle \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto \textbf{never} \\ \mathtt{f} \mapsto 2, \mathtt{o} \mapsto \mathtt{X}' \end{pmatrix}, \mathsf{invREv}(\mathtt{X}, \mathit{dest}, \mathtt{m}, \langle 5 \rangle, \emptyset), \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto \textbf{never} \\ \mathtt{f} \mapsto 2, \mathtt{o} \mapsto \mathtt{X}' \end{pmatrix} \right\rangle$$
$$\emptyset \triangleright \left\langle \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto \textbf{never} \\ \mathtt{f} \mapsto 2, \mathtt{o} \mapsto \mathtt{X}' \end{pmatrix}, \mathsf{invREv}(\mathtt{X}, \mathit{dest}, \mathtt{m}, \langle 5 \rangle, \emptyset), \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto \textbf{never} \\ \mathtt{f} \mapsto 2, \mathtt{o} \mapsto \mathtt{X}' \end{pmatrix} \right\rangle$$

Agreement generates not only the agreed upon trace and the continuation process, but also outputs the event within the candidate. This event is used to communicate the eventual choice for the symbolic value needed for agreement – the global semantics will not execute a step, if the agreement relies on an event that is not possible. E.g., if the process agrees to substitute 5 for the read value of a future f, but the future was resolved with 0.

Definition 3.16 (Agreement)**.** proc *ρ-agrees on a trace θ_F with continuation proc_F if the following holds.*
Let sel^ρ be the set of ρ-selection candidates of the traces in proc*.*

$$\mathsf{sel}^\rho = \{\theta \mid \exists \theta' \in \mathsf{proc}.\ \theta \text{ is the } \rho\text{-selection candidate of } \theta'\}$$

Let $\underline{v}, \underline{v'}$ be the symbolic values in sel[4]. Let v, v' be concrete values such that the following set is a singleton

$$\left\{\theta \mid \exists \theta' \in \mathsf{sel}^\rho.\ \theta = \theta'[\underline{v} \mapsto v][\underline{v'} \mapsto v'] \wedge \theta \text{ is selectable}\right\}$$

The trace θ_F is the sole element of that set. If the set is not a singleton, agreement fails.

[4] There can be at most two symbolic values introduced at the same time, as there is only one event in the selection candidate and each event introduces 0, 1 or 2 symbolic values.

$$\theta_1' = \{2 > \underline{i}, \underline{\texttt{this}.\texttt{f}_2} < 5\}\triangleright \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 2, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{invEv}(\texttt{x}, \texttt{x}', \underline{fut}, \texttt{n}, \langle\rangle) \right\rangle$$
$$\circ \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{futREv}(\texttt{x}, \underline{fut}, \underline{\texttt{n}}, \underline{i}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{suspEv}, \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \diamond \right\rangle$$
$$\circ \left\langle \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2}, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{suspREv}, \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2}, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto \underline{0}, \texttt{o} \mapsto \texttt{x}' \end{pmatrix} \right\rangle$$
$$\circ \left\langle \mathsf{futEv}(\texttt{x}, dest, \texttt{m}, 0), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto \underline{0}, \texttt{o} \mapsto \texttt{x}' \end{pmatrix} \right\rangle$$

$$\theta_2' = \{2 > \underline{i}, \underline{\texttt{this}.\texttt{f}_2} \geq 5\}\triangleright \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 2, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{invEv}(\texttt{x}, \texttt{x}', \underline{fut}, \texttt{n}, \langle\rangle) \right\rangle$$
$$\circ \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{futREv}(\texttt{x}, \underline{fut}, \underline{\texttt{n}}, \underline{i}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix} \mathsf{suspEv}, \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \diamond \right\rangle$$
$$\circ \left\langle \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2}, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{suspREv}, \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2}, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{futEv}(\texttt{x}, dest, \texttt{m}, \underline{\texttt{this}.\texttt{f}_2}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto \underline{\texttt{this}.\texttt{f}_2}, \texttt{o} \mapsto \texttt{x}' \end{pmatrix} \right\rangle$$

$$\theta_3' = \{2 \leq \underline{i}\}\triangleright \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 2, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{invEv}(\texttt{x}, \texttt{x}', \underline{fut}, \texttt{n}, \langle\rangle), \right\rangle$$
$$\circ \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{futREv}(\texttt{x}, \underline{fut}, \underline{\texttt{n}}, \underline{i}), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{futEv}(\texttt{x}, dest, \texttt{m}, 3), \begin{pmatrix} \texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{fut} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix} \right\rangle$$

Figure 3.8: First continuation process of Fig. 3.6.

Let $\overline{\mathsf{proc}}$ *be the set of traces, whose* ρ*-selection candidates are selected with the substitution* $[\underline{v} \mapsto v][\underline{v'} \mapsto v']$*.*

$$\mathsf{proc}_F = \left\{ \theta_C \mid \exists \theta' \in \overline{\mathsf{proc}}.\ \theta' = \left((\theta_F \ast\!\ast\!\ast \theta_C)[\underline{v} \mapsto v][\underline{v'} \mapsto v']\right)\rho \wedge |\theta_C| > 1 \right\}$$

The condition on the length of the trace in the last step produces an empty continuation (i.e., an empty set) upon termination of the execution. To stress the event we say that proc **ρ*-agrees on a trace* θ_F *with continuation* proc_F *under event* $\theta_F[2]$*. The trace* θ_C *in the above construction is said to continue the corresponding* $\theta \in \mathsf{proc}$*. If any of the above steps fails, agreement fails.*

Example 3.6. *The above three traces* $\{\texttt{f} \mapsto 2\}$*-agree on the following trace under* $\mathsf{invREv}(\texttt{x}, dest, \texttt{m}, \langle 5\rangle)$*. This corresponds to an execution step to schedule the process.*

$$\hat{\theta} = \emptyset\triangleright \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 2, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{invREv}(\texttt{x}, dest, \texttt{m}, \langle 5\rangle), \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 2, \texttt{o} \mapsto \texttt{x}' \end{pmatrix} \right\rangle$$

The continuation process $\{\theta_1', \theta_2', \theta_3'\}$ *consists of three traces, show in Fig. 3.8.*
Next, the continuation agrees on the following (for any heap): This corresponds to an execution step to execute the first statement.

$$\hat{\theta}' = \emptyset\triangleright \left\langle \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 2, \texttt{o} \mapsto \texttt{x}' \end{pmatrix}, \mathsf{noEv}, \begin{pmatrix} \texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never} \\ \texttt{f} \mapsto 3, \texttt{o} \mapsto \texttt{x}' \end{pmatrix} \right\rangle$$

$$\hat{\theta}''' = \{2 > 0\}\triangleright \left\langle \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto f \\ \mathtt{f} \mapsto 3, \mathtt{o} \mapsto \mathtt{x}' \end{pmatrix}, \mathsf{futREv}(\mathtt{x}, f, 0, \mathtt{n}, 1), \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto f \\ \mathtt{f} \mapsto 3, \mathtt{o} \mapsto \mathtt{x}' \end{pmatrix} \right\rangle$$
$$\hat{\theta}'''' = \{2 \leq 5\}\triangleright \left\langle \begin{pmatrix} \mathtt{i} \mapsto 5, \mathtt{fut} \mapsto f \\ \mathtt{f} \mapsto 3, \mathtt{o} \mapsto \mathtt{x}' \end{pmatrix}, \mathsf{futREv}(\mathtt{x}, f, 5, \mathtt{p}, 1), \begin{pmatrix} \mathtt{i} \mapsto 5, \mathtt{fut} \mapsto f \\ \mathtt{f} \mapsto 3, \mathtt{o} \mapsto \mathtt{x}' \end{pmatrix} \right\rangle$$

Figure 3.9: Two possible selected traces in Ex. 3.4.

The next selected candidate is the following, again for any heap and some future f:

$$\hat{\theta}'' = \emptyset\triangleright \left\langle \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto \mathbf{never} \\ \mathtt{f} \mapsto 3, \mathtt{o} \mapsto \mathtt{x}' \end{pmatrix}, \mathsf{invEv}(\mathtt{x}, \mathtt{x}', f, \mathtt{n}, \langle\rangle), \begin{pmatrix} \mathtt{i} \mapsto 0, \mathtt{fut} \mapsto f \\ \mathtt{f} \mapsto 3, \mathtt{o} \mapsto \mathtt{x}' \end{pmatrix} \right\rangle$$

They can agree on any future – the global system semantics later forces an agreement on a fresh future. Finally, there is a split, they can agree on, among others, one of the following two traces, depending on the value that is instantiated for $\underline{\mathtt{i}}$ and the resolving method name. Two possible traces are shown in Fig. 3.9. The difference now is that the first trace selects the first branch of the branching (and its continuation set has two elements), while the second trace selects the empty **else** *branch and has only one element in its continuation set. Again, the global system semantics ensure later that the values the process agrees on for substitution are indeed the values the future was resolved with.*

Definition 3.17 (Object Semantics)**.** *The semantics of an object is a relation $\xrightarrow{\mathsf{ev}}$ between objects, where* ev *is either an event or a marked event $\overline{\mathsf{ev}}$. We say that* ev *is communicated to the outside. The semantics consists of four rules, given in Fig. 3.10.*

- *Rule (**O-Schedule**) activates a process in the process pool, if the traces agrees on a trace under the current heap and no other process is active. This trace is executed by being appended to the object trace and making the continuation to the active process. The event under which the traces agree is communicated to the outside. It is not possible to activate a process that immediately suspends – if two* **await** *statement follow each other directly, then there is still a suspension reaction event between their suspension events.*
- *Rule (**O-Step**) is analogous to (**O-Schedule**), but the process is already active.*
- *Rule (**O-Deschedule**) is analogous to (**O-Step**), but if the communicated event is suspending, then the continuation is added to the process pool and no active process exists.*
- *Rule (**O-Add**) finally adds a called method to the process pool. This rule can always be executed, the system semantics ensure that every called method is added exactly once. This is, besides the initial state the* only *place the GC and LA layers interact. Note that it uses a labeled invocation event and not a invocation reaction event, because the called method is not scheduled yet.*

*Only the (**O-Schedule**) rules uses the $***$ operator, because this is the only point where the local state of the next step may not agree with the local state of the last step. There is no rule for process termination. If a process terminates, then the continuation is the empty set and this falls under the rule (**O-Step**).*

Example 3.7. *Fig. 3.11 shows how Ex. 3.6 can be formulated with the object semantics. For readability's sake we omitted the continuations and the last trace before suspension.*

3.3.2 Semantics of Programs

Objects are always reduced in the context of a system, that enforces that the steps a process agrees on to execute next adheres to the values already used in the system.

$$\textbf{(O-Schedule)}\ \frac{\mathsf{last}(\theta) = \binom{\sigma}{\rho} \qquad \Theta\ \rho\text{-agrees on } \theta' \text{ with continuation } \Theta' \text{ under } \mathsf{ev}}{\left((\Theta_i)_{i\in I} \cup \{\Theta\}, \theta, \emptyset\right)_{\mathtt{X}} \xrightarrow{\mathsf{ev}} \left((\Theta_i)_{i\in I}, \theta *** \theta', \Theta'\right)_{\mathtt{X}}}$$

$$\textbf{(O-Step)}\ \frac{\begin{array}{c}\mathsf{ev} \text{ is neither } \mathsf{condEv} \text{ nor } \mathsf{suspEv} \\ \mathsf{last}(\theta) = \binom{\sigma}{\rho} \qquad \Theta\ \rho\text{-agrees on } \theta' \text{ with continuation } \Theta' \text{ under } \mathsf{ev}\end{array}}{\left((\Theta_i)_{i\in I}, \theta, \Theta\right)_{\mathtt{X}} \xrightarrow{\mathsf{ev}} \left((\Theta_i)_{i\in I}, \theta ** \theta', \Theta'\right)_{\mathtt{X}}}$$

$$\textbf{(O-Deschedule)}\ \frac{\begin{array}{c}\mathsf{ev} \text{ is either } \mathsf{condEv} \text{ or } \mathsf{suspEv} \\ \mathsf{last}(\theta) = \binom{\sigma}{\rho} \qquad \Theta\ \rho\text{-agrees on } \theta' \text{ with continuation } \Theta' \text{ under } \mathsf{ev}\end{array}}{\left((\Theta_i)_{i\in I}, \theta, \Theta\right)_{\mathtt{X}} \xrightarrow{\mathsf{ev}} \left((\Theta_i)_{i\in I} \cup \{\Theta'\}, \theta ** \theta', \emptyset\right)_{\mathtt{X}}}$$

$$\textbf{(O-Add)}\ \frac{\mathtt{j} \text{ is fresh}}{\left((\Theta_i)_{i\in I}, \theta ** \left\langle \binom{\sigma}{\rho} \right\rangle, \Theta\right)_{\mathtt{X}} \xrightarrow{\overline{\mathsf{invEv}(\mathtt{X}', \mathtt{X}, f, \mathtt{m}, \vec{\mathtt{e}})}} \left((\Theta_i)_{i\in I} \cup \{[\![\mathtt{m}]\!]_{\mathtt{X}, dest, \mathtt{m}, \rho@j}\}, \theta ** \left\langle \binom{\sigma}{\rho} \right\rangle, \Theta\right)_{\mathtt{X}}}$$

Figure 3.10: Globally Concrete Semantics of Objects.

$$\begin{aligned}
&\left(\emptyset, \begin{pmatrix}\cdot \\ \mathtt{f} \mapsto 2\end{pmatrix}, \emptyset\right)_{\mathtt{X}} \xrightarrow{\overline{\mathsf{invEv}(\mathtt{X}'', \mathtt{X}, dest, \mathtt{m}, \langle 5\rangle)}} \left(\{\{\theta_1, \theta_2, \theta_3\}\}, \begin{pmatrix}\cdot \\ \mathtt{f} \mapsto 2\end{pmatrix}, \emptyset\right)_{\mathtt{X}} \\
\xrightarrow{\mathsf{invREv}(\mathtt{X}, dest, \mathtt{m}, \langle 5\rangle)} &\left(\emptyset, \begin{pmatrix}\cdot \\ \mathtt{f} \mapsto 2\end{pmatrix} *** \hat{\theta}, \{\theta_1', \theta_2', \theta_3'\}\right)_{\mathtt{X}} \xrightarrow{\mathsf{noEv}} \left(\emptyset, \begin{pmatrix}\cdot \\ \mathtt{f} \mapsto 2\end{pmatrix} *** \hat{\theta} ** \hat{\theta}', \{\ldots\}\right)_{\mathtt{X}} \\
\xrightarrow{\mathsf{invEv}(\mathtt{X}, \mathtt{X}', f, \mathtt{n}, \langle\rangle)} &\left(\emptyset, \begin{pmatrix}\cdot \\ \mathtt{f} \mapsto 2\end{pmatrix} *** \hat{\theta} ** \hat{\theta}' ** \hat{\theta}'', \{\ldots\}\right)_{\mathtt{X}} \\
\xrightarrow{\mathsf{futREv}(\mathtt{X}, f, \mathtt{n}, 0)} &\left(\emptyset, \begin{pmatrix}\cdot \\ \mathtt{f} \mapsto 2\end{pmatrix} *** \hat{\theta} ** \hat{\theta}' ** \hat{\theta}'' ** \hat{\theta}''', \{\ldots\}\right)_{\mathtt{X}} \\
\xrightarrow{\mathsf{suspEv}} &\left(\{\ldots\}, \begin{pmatrix}\cdot \\ \mathtt{f} \mapsto 2\end{pmatrix} *** \hat{\theta} ** \hat{\theta}' ** \hat{\theta}'' ** \hat{\theta}''' ** \ldots, \emptyset\right)_{\mathtt{X}} \to \cdots \to (\emptyset, \theta, \emptyset)_{\mathtt{X}}
\end{aligned}$$

Figure 3.11: Ex. 3.6 in the object semantics.

$$\textbf{(S-Internal)}\ \frac{\mathbf{O} \xrightarrow{\mathsf{ev}} \mathbf{O}' \qquad \mathsf{ev} \text{ is neither a invEv nor a futREv nor has the form } \overline{\mathsf{ev}}}{\mathsf{evs} \blacktriangleright \mathbf{O}\ \mathsf{obs} \xrightarrow{\mathsf{ev}} \mathsf{evs} \circ \langle \mathsf{ev} \rangle \blacktriangleright \mathbf{O}'\ \mathsf{obs}}$$

$$\textbf{(S-Get)}\ \frac{\mathbf{O} \xrightarrow{\mathsf{ev}} \mathbf{O}' \qquad \mathsf{ev} = \mathsf{futREv}(\mathtt{x}, f, \mathtt{m}, \mathtt{v}, i) \qquad \mathsf{futEv}(\mathtt{x}', f, \mathtt{m}, \mathtt{v}) \text{ is in evs for some } \mathtt{x}'}{\mathsf{evs} \blacktriangleright \mathbf{O}\ \mathsf{obs} \xrightarrow{\mathsf{ev}} \mathsf{evs} \circ \langle \mathsf{ev} \rangle \blacktriangleright \mathbf{O}'\ \mathsf{obs}}$$

$$\textbf{(S-Invoc)}\ \frac{\mathbf{O}_1 \xrightarrow{\mathsf{invEv}(\mathtt{X},\mathtt{X}',f,\mathtt{m},\overrightarrow{\mathtt{e}})} \mathbf{O}_1' \qquad \mathbf{O}_2 \xrightarrow{\overline{\mathsf{invEv}(\mathtt{X},\mathtt{X}',f,\mathtt{m},\overrightarrow{\mathtt{e}})}} \mathbf{O}_2' \qquad f \text{ is fresh in evs}}{\mathsf{evs} \blacktriangleright \mathbf{O}_1\ \mathbf{O}_2\ \mathsf{obs} \xrightarrow{\mathsf{invEv}(\mathtt{X},\mathtt{X}',f,\mathtt{m},\overrightarrow{\mathtt{e}})} \mathsf{evs} \circ \left\langle \mathsf{invEv}(\mathtt{x}, \mathtt{x}', f, \mathtt{m}, \overrightarrow{\mathtt{e}}) \right\rangle \blacktriangleright \mathbf{O}_1'\ \mathbf{O}_2'\ \mathsf{obs}}$$

Figure 3.12: Globally Concrete Semantics of Systems.

Definition 3.18 (Systems)**.** *A* system ***S*** *has the form* evs ▶ obs*, where* evs *is a sequence of events and* obs *consists of objects:*

$$\mathsf{obs} ::= \mathbf{O} \mid \mathsf{obs}\ \mathsf{obs}$$

The composition obs obs *is commutative and associative:* obs obs′ = obs′ obs *and* obs (obs′ obs″) = (obs obs′) obs″.

Definition 3.19 (Initial, Terminated and Stuck Systems)**.** *A system is terminated if all its objects have (1) an empty process pool and (2) no active process. A system is stuck if it cannot be reduced further, but is not terminated. Given a main block*

```
main{
    C1 v1 = new C1(e);
    ...
    Cn vn = new Cn(e);
    vj!m();
}
```

The initial system has the form

$$\langle\rangle \blacktriangleright \mathbf{O}_1\ \ldots\ \mathbf{O}_n$$

Where the initial objects are defined in Def. 3.14

Definition 3.20 (System Semantics)**.** *Fig. 3.12 shows the rules for system semantics.*

- *(**S-Internal**) applies one rule to some object and records the event in the global history.*
- *(**S-Get**) applies a rule to some object that communicates a resolving reaction event. It is checked that the read value that the process agreed on is the value that the read future is resolved with.*
- *Rule (**S-Invoc**) applies a rule for some object that communicates an invocation event. It adds the called method by applying (**O-Add**) on the called object. Only the communicated invocation event is added to the global history.*

Analogously to local traces, we define global traces. A global trace is a sequence of pairs, where each pair is the global state at this point of execution, and the last executed event. A global state only records the current heaps of all objects, not the traces, processes and process pools.

Definition 3.21 (Global Traces and Big-Step Semantics)**.** *A global state* $\mathsf{g|}$ *is a function from object names to heaps. Given a system*

$$\mathbf{S} = \mathsf{evs} \blacktriangleright \left(\mathsf{procpool}_1, \theta_1 \ast\!\ast\!\ast \left\langle \begin{pmatrix}\sigma_1\\ \rho_1\end{pmatrix} \right\rangle, \mathsf{proc}_1\right)_{\mathtt{X}_1} \ldots \left(\mathsf{procpool}_n, \theta_n \ast\!\ast\!\ast \left\langle \begin{pmatrix}\sigma_n\\ \rho_n\end{pmatrix} \right\rangle, \mathsf{proc}_n\right)_{\mathtt{X}_n}$$

The corresponding global state $\mathsf{g|}(\mathbf{S})$ *is* $\{\mathtt{X}_1 \mapsto \rho_1, \ldots, \mathtt{X}_n \mapsto \rho_n\}$.

A global trace γ *is a sequence of global states and events, analogous to local histories. Given a run of a system*

$$\mathbf{S}_1 \xrightarrow{\mathsf{ev}_2} \mathbf{S}_2 \ldots \xrightarrow{\mathsf{ev}_n} \mathbf{S}_n$$

its global trace is

$$\left\langle \mathsf{g|}(\mathbf{S}_1), \mathsf{ev}_2, \mathsf{g|}(\mathbf{S}_2), \ldots, \mathsf{ev}_n, \mathsf{g|}(\mathbf{S}_n) \right\rangle$$

Given a program Prgm*, we say that* Prgm *realizes a global trace* γ*, if there is a run from its initial state to a terminated state, such that* γ *is the trace of that run. We write this as* $\mathsf{Prgm} \Downarrow \gamma$.

Symbolic values do not escape from locally abstract semantics, neither the object traces nor the global trace contains symbolic values.

Theorem 1 (Global Symbolic Isolation of Locally Abstract Semantics)**.** *Let* Prgm *be a program and* γ *a global trace with* $\mathsf{Prgm} \Downarrow \gamma$*.* γ *contains no symbolic values.*

Proof. See p. 166.

3.4 Selectability

The local semantics of a method can be expressed as $[\![\mathtt{m}]\!]_{\underline{\mathtt{X}},\underline{dest},\mathtt{m},\left(\frac{\underline{\sigma}}{\underline{\rho}}\right)}$, where $\underline{\sigma}$ and $\underline{\rho}$ are fully symbolic[5]. The store $\underline{\sigma}$ also maps all parameters to symbolic object names. This set describes the behavior of a method in every possible context, but also vastly overapproximates it, especially when some context is known. Such overapproximation manifests in several ways:

- Some symbolic traces are never selected, because their selection condition can never hold.
- Some symbolic traces are not selected in some context, because in this context their selection condition can never hold.
- Some concrete instantiantions of symbolic traces are not possible in some or every context.

The first case is dead code. We give an example to illustrate the other cases.

Example 3.8. *Consider the following method in some class with a field* `Int fd = 10`.

```
1 Int m(Int p){
2     if( this.fd < 0 ) {
3         this.fd = -1;
4     } else {
5         if(p >= 0){ this.fd = this.fd + p; }
6         else { this.fd = 10; }
7     }
8     return this.fd;
9 }
```

[5] Strictly spoken, we misuse syntax here as the method is evaluated with a heap $\underline{\rho}$. Here, $\underline{\sigma}$ just denotes the initial local store.

The semantics contains three traces, one for each branch (with simplified events). The field `fd` *is initially some symbolic field* $\underline{fd}$ *and the parameters* `p` *is some symbolic value* $\underline{p}$*.*

$$\{\underline{fd} < 0\} \triangleright \left\langle \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto \underline{fd}\}\end{pmatrix}, \mathsf{invREv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto \underline{fd}\}\end{pmatrix}, \mathsf{noEv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto -1\}\end{pmatrix}, \mathsf{futEv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto -1\}\end{pmatrix} \right\rangle \tag{A}$$

$$\left\{\begin{array}{l}\underline{p} \geq 0,\\ \underline{fd} \geq 0\end{array}\right\} \triangleright \left\langle \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto \underline{fd}\}\end{pmatrix}, \mathsf{invREv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto \underline{fd}\}\end{pmatrix}, \mathsf{noEv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto \underline{fd} + \underline{p}\}\end{pmatrix}, \mathsf{futEv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto \underline{fd} + \underline{p}\}\end{pmatrix} \right\rangle \tag{B}$$

$$\left\{\begin{array}{l}\underline{p} < 0,\\ \underline{fd} \leq 0\end{array}\right\} \triangleright \left\langle \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto \underline{fd}\}\end{pmatrix}, \mathsf{invREv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto \underline{fd}\}\end{pmatrix}, \mathsf{noEv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto 10\}\end{pmatrix}, \mathsf{futEv}, \begin{pmatrix}\{\mathtt{p} \mapsto \underline{p}\}\\ \{\mathtt{fd} \mapsto 10\}\end{pmatrix} \right\rangle \tag{C}$$

If some context of the method is known, then we can exclude some concrete traces from being selected. If there is no further method in this class, then trace A is never selected: `this.fd` *is not negative in the initial state and so the trace is not selected in the first run of the method – but the field stays non-negative and so every further run also never selects this trace. The trace B is an abstraction for the following concrete trace:*

$$\{\mathbf{True}\} \triangleright \left\langle \begin{pmatrix}\{\mathtt{p} \mapsto 1\}\\ \{\mathtt{fd} \mapsto 5\}\end{pmatrix}, \mathsf{invREv}, \begin{pmatrix}\{\mathtt{p} \mapsto 1\}\\ \{\mathtt{fd} \mapsto 5\}\end{pmatrix}, \mathsf{noEv}, \begin{pmatrix}\{\mathtt{p} \mapsto 1\}\\ \{\mathtt{fd} \mapsto 6\}\end{pmatrix}, \mathsf{futEv}, \begin{pmatrix}\{\mathtt{p} \mapsto 1\}\\ \{\mathtt{fd} \mapsto 6\}\end{pmatrix} \right\rangle$$

This concrete trace cannot be selected: the symbolic value $\underline{fd}$ *is never substituted with* 5*, because the field* `fd` *starts at* 10 *and then only increases (or is reset to* 10*). Indeed, none of the traces substitutes* $\underline{fd}$ *with a value below* 10*. We stress that the above reasoning only requires the class to be known, not the whole program.*

Trace C is only selected if the parameter `p` *is strictly negative. If we know that the method is only called with positive parameters, then we may also say that trace C is never selected.*

For precision it is important to ensure that formal verification only considers traces that really correspond to possible runs, otherwise verification may fail with a false negative: verification may fail, but only for some behavior that does not correspond to a possible run. This is akin to the restriction on reachable states in, e.g., model checking, instead of checking all possible states of a system. While it is not possible to exclude all traces that do not correspond to real runs, one can still aim to be as precise as possible. Before we can formalize the reasoning about the above example, we require some technical tools to be more precise about substitution.

Concretizers.

The LAGC semantics substitute concrete values for the symbolic values and symbolic fields step-by-step. This means that a concrete trace is never produced in the semantics. We introduce *concretizers* to apply all substitutions during a run *at once* on a local trace.

Definition 3.22 (Concretizers)**.** *A concretizer* χ *is a function from symbolic fields and symbolic values to semantics value.*

Note that neither local states nor heaps are concretizers. A concretizer is a recording of substitutions during the concretization of local abstract traces.

Definition 3.23 (Used Concretizers)**.** *Let* $\underline{\sigma}$ *and* $\underline{\rho}$ *be fully symbolic and let* $\theta \in [\![\mathtt{m}]\!]_{\mathbf{X},\underline{dest},\mathtt{m},\left(\frac{\underline{\sigma}}{\underline{\rho}}\right)}$ *be a trace of* `m`*. A used concretizer* χ *for trace* θ *in a run of* Prgm *is constructed as follows.*

- *There is some application of* (**O-Add**) *with* $\theta' \in [\![\mathtt{m}]\!]_{\mathbf{X},dest,\mathtt{m},\left(\frac{\sigma}{\rho}\right)}$ *such that there are symbolic values* $\{\mathtt{v}_i \mapsto \underline{v}_i\}_{i\in I} \in \underline{\sigma}$ *and* $\{\mathtt{v}_i \mapsto v_i\}_{i\in I} \in \sigma$ *such that*

  $$\theta[\underline{v}_i \mapsto v_i]_{i\in I} = \theta'$$

  Then we set

  $$\chi(\underline{v}_i) = v_i \text{ for all } i \in I$$

- *for each agreement where θ' (or one of its continuations) agrees under some substitution $[\underline{v} \mapsto v]$:*

$$\chi(\underline{v}) = v$$

- *for each agreement where θ' (or one of its continuations) agrees under some heap which substitutes $\underline{\texttt{this.f}_i}$ by a concrete value v:*

$$\chi(\underline{\texttt{this.f}_i}) = v$$

We write the application of a concretizer χ to a trace θ (history, etc.) as $\theta\chi$

Example 3.9. *Ex. 3.8 shows fully symbolic traces. In Ex. 3.3, one fully symbolic trace is the following:*

$$\begin{aligned}\theta_{\text{symb}} = \{\underline{\texttt{this.f}_1} > \underline{i}, \underline{\texttt{this.f}_2} < \underline{\texttt{p}}\} \triangleright & \left\langle \begin{pmatrix}\texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never}\\ \texttt{f} \mapsto \underline{\texttt{this.f}_1}, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix}, \texttt{noEv}, \begin{pmatrix}\texttt{i} \mapsto 0, \texttt{fut} \mapsto \textbf{never}\\ \texttt{f} \mapsto \underline{\texttt{this.f}_1}, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix}, \texttt{invEv}(\texttt{x}, \underline{\texttt{x}'}, \underline{f}, \texttt{n}, \langle\rangle) \right\rangle \\ \circ & \left\langle \begin{pmatrix}\texttt{i} \mapsto 0, \texttt{fut} \mapsto \underline{f}\\ \texttt{f} \mapsto \underline{\texttt{this.f}_1} + 1, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix}, \texttt{futREv}(\texttt{x}, \underline{f}, \underline{\texttt{n}}, \underline{i}), \begin{pmatrix}\texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{f}\\ \texttt{f} \mapsto \underline{\texttt{this.f}_1} + 1, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix}, \texttt{condEv} \right\rangle \\ \circ & \left\langle \begin{pmatrix}\texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{f}\\ \texttt{f} \mapsto \underline{\texttt{this.f}_1} + 1, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix}, \diamond, \begin{pmatrix}\texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{f}\\ \texttt{f} \mapsto \underline{\texttt{this.f}_2}, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix}, \texttt{condREv} \right\rangle \\ \circ & \left\langle \begin{pmatrix}\texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{f}\\ \texttt{f} \mapsto \underline{\texttt{this.f}_2}, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix}, \texttt{noEv}, \begin{pmatrix}\texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{f}\\ \texttt{f} \mapsto \underline{0}, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix}, \texttt{futEv}(\texttt{x}, \mathit{dest}, \texttt{m}, 0), \begin{pmatrix}\texttt{i} \mapsto \underline{i}, \texttt{fut} \mapsto \underline{f}\\ \texttt{f} \mapsto \underline{0}, \texttt{o} \mapsto \underline{\texttt{x}'}\end{pmatrix} \right\rangle\end{aligned}$$

A used concretizer for θ_{symb}, as outlined by θ_1 and its continuations in examples 3.3 to 3.6 is the following (if during the suspension the heap does not change):

$$\chi_1 = \{\underline{\texttt{x}'} \mapsto \texttt{x}, \underline{\texttt{p}} \mapsto 5, \underline{\texttt{this.f}_1} \mapsto 2, \underline{f} \mapsto f, \underline{i} \mapsto 0, \underline{\texttt{this.f}_2} \mapsto 3, \underline{\texttt{n}} \mapsto \texttt{n}\}$$

Selected Traces.

Using concretizers, we are now able to define selected traces. A selected trace is the result of applying a concretizer on a fully symbolic trace of a method.

Definition 3.24. *Let* Prgm *be a program, $\underline{\sigma}$ and $\underline{\rho}$ be fully symbolic and let $\theta \in [\![\texttt{m}]\!]_{\underline{\texttt{x}}, \underline{\mathit{dest}}, \texttt{m}, (\frac{\underline{\sigma}}{\underline{\rho}})}$ be a trace of* m*. Let $X(\theta)$ be the set of used concretizers for θ.*

The set of selected concrete traces for θ is defined as follows:

$$\text{selected}(\theta) = \left\{\theta' \mid \exists \chi \in X(\theta).\ \theta\chi = \theta'\right\}$$

This is lifted to method as follows (where the evaluation parameters are as above):

$$\text{selected}(\texttt{m}) = \bigcup_{\theta \in [\![\texttt{m}]\!]_{\underline{\texttt{x}}, \underline{\mathit{dest}}, \texttt{m}, (\frac{\underline{\sigma}}{\underline{\rho}})}} \text{selected}(\theta)$$

Selected Traces of Statements.

The above definition does not allow us to reason about the traces from statements inside the method body. This prevents us from talking about the traces of single statements. To do so, we extend selected concrete traces to concretized subtraces. We say that a trace θ is a subtrace of a trace θ' if $\theta' = \theta_1 *** \theta *** \theta_2$. We write $\text{sub}(\theta, \theta')$. We are interested in the traces of a statement, which can be composed from subtraces of substatements.

Definition 3.25. *Let* s *be a statement. The set of substatements* ssub(s) *is defined as follows, where the first two clauses match only on* s *without loops, branchings or sequential compositions.*

$$\begin{aligned}\text{ssub}(\texttt{s;s'}) &= \{\texttt{s;s'}\} \cup \text{ssub}(\texttt{s}) \cup \text{ssub}(\texttt{s'})\\ \text{ssub}(\texttt{s}) &= \{\texttt{s}\} \qquad \textit{if s is not composed}\\ \text{ssub}(\textbf{while}\texttt{(e)}\{\texttt{s}\}\texttt{;s'}) &= \{\textbf{while}\texttt{(e)}\{\texttt{s}\}\texttt{;s'}\} \cup \text{ssub}(\texttt{s}) \cup \text{ssub}(\texttt{s'})\\ \text{ssub}(\textbf{if}\texttt{(e)}\{\texttt{s;}\textbf{skip}\texttt{;}\}\textbf{else}\{\texttt{s';}\textbf{skip}\texttt{;}\}\texttt{s''}) &= \{\textbf{if}\texttt{(e)}\{\texttt{s;}\textbf{skip}\texttt{;}\}\textbf{else}\{\texttt{s';}\textbf{skip}\texttt{;}\}\texttt{s''}\} \cup \text{ssub}(\texttt{s;s''}) \cup \text{ssub}(\texttt{s';s''})\end{aligned}$$

A statement may occur multiple times in a statement. We refrain from introducing program-point identifiers for all statements. During verification this is not a problem, because all our verification rules are given in prefix form, i.e., match on a statement and its suffix in the method. The semantics of all elements in $\mathsf{ssub}(\mathtt{s})$ is defined if the semantics of $\mathtt{s}$ is defined. Sub-selection generalizes selection of traces of methods to selection of subtraces of substatements of the method body.

Definition 3.26. *Let* Prgm *be a program,* $\underline{\sigma}$ *and* $\underline{\rho}$ *be fully symbolic and let* $\theta \in [\![\mathtt{m}]\!]_{\underline{\mathtt{X}},\underline{dest},\mathtt{m},(\underline{\sigma}\ \underline{\rho})}$ *be a trace of* $\mathtt{m}$*. Let* X *be the set of all concretizers,* $\mathtt{m}$ *a method and* $\mathtt{s}_\mathtt{m}$ *its method body. Let* $\mathtt{s}$ *be a substatement of the method,* $\mathtt{s} \in \mathsf{ssub}(\mathtt{s}_\mathtt{m})$*. A trace of* $\mathtt{s}$ *is sub-selected with respect to some concrete trace* θ*, if it can be concretized to some subtrace of* θ*.*

$$\mathsf{sub-selected}(\mathtt{s},\theta) = \left\{\theta' \mid \exists \chi \in X.\ \exists \theta'' \in [\![\mathtt{s}]\!]_{\underline{\mathtt{X}},\underline{dest},\mathtt{m},(\underline{\sigma}\ \underline{\rho})}.\theta''\chi = \theta' \wedge \mathsf{sub}(\theta',\theta)\right\}$$

A trace of a statement $\mathtt{s}$ *within* $\mathtt{m}$ *is sub-selected if it is sub-selected with respect to some used trace* θ *of* $\mathtt{m}$*.*

$$\mathsf{sub-selected}(\mathtt{s},\mathtt{m}) = \bigcup_{\theta \in \mathsf{selected}(\mathtt{m})} \mathsf{sub-selected}(\mathtt{s},\theta)$$

If $\mathtt{m}$ is understood, we simply write $\mathsf{sub-selected}(\mathtt{s})$. Note that the concretizer of the (symbolic) subtrace of the substatement $\mathtt{s}$ is not necessarily the one used to concretize $\mathtt{m}$, because the two fully symbolic object states may differ in their symbolic values.

Example 3.10. *Consider the method body, for some expression* `e` *for a class with some field* `f`*.*

```
await e?; while(i > 0){await e; i = i - 1;}return 0;
```

Consider the situation where both symbolic heaps map `f` *to* $\underline{\texttt{this.f}_1}$ *at the beginning and then to* $\underline{\texttt{this.f}_2}$*,* $\underline{\texttt{this.f}_3}$*,..., In the the LA semantics, the symbolic field after the first iteration of the loop is* $\underline{\texttt{this.f}_3}$*. However, if* `while(i > 0){await e; i = i - 1;}return 0;` *is evaluated isolated with the same symbolic index, the symbolic field after the first iteration of the loop is* $\underline{\texttt{this.f}_2}$*.*

As the name of the symbolic field has no importance, one may safely rename it to $\underline{\texttt{this.f}_2}$ in the whole trace. As the exact renaming carries no information either, we may as well use a different fully symbolic name, which is chosen such that the symbolic fields match the ones used by the method.

We can now also formalize our statements in Ex. 3.8.

Example 3.11. *If there is no further method in the class of* $\mathtt{m}$*, then there is no used concretizer* χ *for any trace in Ex. 3.8 that has* $\chi(\underline{fd}) = 5$*. The trace A is never selected.*

For the program logic defined in the next chapter it is crucial to discuss selected traces starting in some fixed state and we define selectability for statements within methods in a given state as follows:

$$\mathsf{selected}\left(\mathtt{s},\mathtt{m},\begin{pmatrix}\sigma\\ \rho\end{pmatrix}\right) = \left\{\theta \mid \theta \in \mathsf{sub-selected}(\mathtt{s},\mathtt{m}) \wedge \emptyset \triangleright \left\langle\begin{pmatrix}\sigma\\ \rho\end{pmatrix}\right\rangle ** \theta = \theta\right\}$$

3.5 Semantic Logics

We use several logics to express properties of states and traces. As the semantics consists of local states, local traces, global states and global traces, we have four logics. The state logics are standard first-order (FO) logics. The trace logics are monadic second-order (MSO) logics, that embed the state logics by using them similar to predicates on states. Similarly, they have event terms that allow to specify events.

The models are concrete traces, i.e., traces without symbolic values or fields. Thus, the model theory is not concerned with symbolic elements. The LA semantics given in the above sections, however, allows us to precisely define which concrete local traces must be considered to reason about a method.

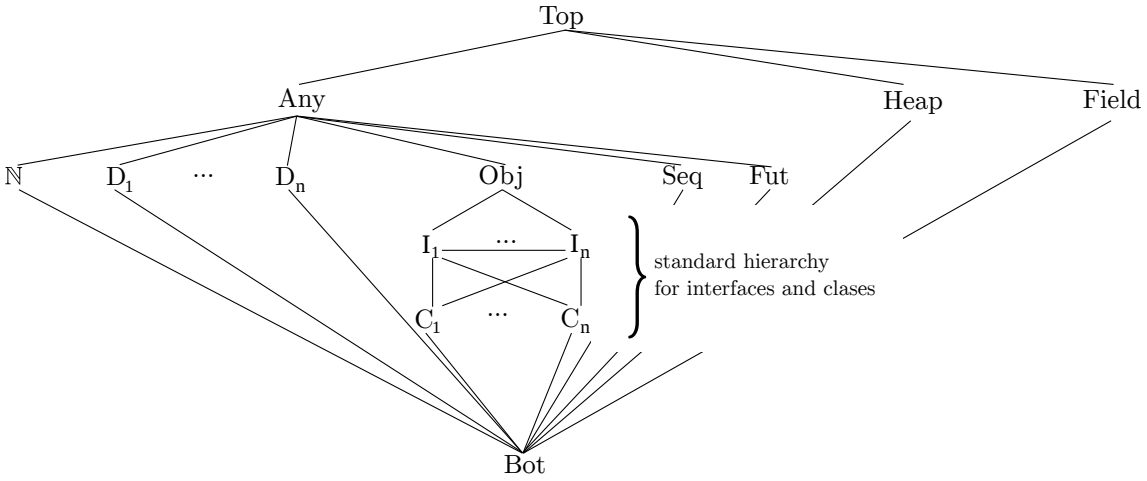


Figure 3.13: Type Hierarchy in lFOS. $\mathtt{D}_i$ range over the non-interface data types.

3.5.1 Local First-Order State Logic (lFOS)

Local First-Order State Logic (lFOS) describes a single, concrete object state. The logic is standard and we refer to Schmitt's [115] description of Java First-Order-Logic (JFOL) for omitted technical details.

Definition 3.27 (Syntax)**.** *Let p range over predicate symbols, f over function symbols, x over logical variable names and S over sorts. As sorts we take all data types* D*, all interfaces, all class names and additionally* Field*,* Fut*,* Any*,* Obj*,* Top*,* Bot*,* $\mathbb{N}$*,* Heap *and* Seq*. The logical heaps are functions from field names to semantic values. The type hierarchy is show in Fig. 3.13. Formulas φ and terms* t *are defined by the following grammar, where* v *ranges over program variables, consisting of local variables and the special variable* heap*, and* f *ranges over all field names. Let* e *range over expressions without fields, but with the extra program variables* heap*,* oldHeap *and* lastHeap*.*

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

We use the usual abbreviations ($\mathsf{True}, \forall, \rightarrow, \ldots$), the usual constants and that each operator defined in syntactic expressions e is a function symbol, so one can directly translate a syntactic expression into a lFOS term. We, thus, assume the existence of the following function symbols (where we write some binary function symbols in infix-form):

$$\begin{aligned}&\mathtt{Never} \mid \mathtt{Nil} \mid \mathtt{True} \mid \mathtt{False} \mid \mathtt{len}(\mathsf{t}) \mid \mathtt{hd}(\mathsf{t}) \mid \mathtt{tl}(\mathsf{t}) \mid \mathtt{Cons}(\mathsf{t},\mathsf{t}) \mid \mathsf{t} \sim \mathsf{t} \mid \mathtt{!t} \mid \mathtt{-t} \mid \mathsf{t}[\mathsf{t}]\\ &\quad \mid \mathsf{select}(\mathsf{t},\mathsf{t}) \mid \mathsf{store}(\mathsf{t},\mathsf{t},\mathsf{t}) \mid \mathsf{anon}(\mathsf{t})\end{aligned}$$

Term $\mathsf{t}[\mathsf{t}]$ is indexed list access, $\mathsf{select}(h, \mathtt{f})$ selects the value of field f from heap h[6], $\mathsf{store}(h, \mathtt{f}, \mathsf{t})$ stores the value of t in field f of heap h. Finally, $\mathsf{anon}(h)$ is a heap that may be different from heap h. This term is used to remove information when symbolically executing a **await** statement.

Convention 4. *We use expressions directly as terms and assume that field accesses are implicitly translated to terms over the* heap *variable. E.g.,* **this**.f *is translated into* $\mathsf{select}(\mathtt{heap}, \mathtt{f})$*. In examples we write terms of the form* $\mathsf{select}(\mathtt{heap}, \mathtt{f})$ *variable in their expression equivalent. Terms* $\mathsf{select}(\mathtt{lastHeap}, \mathtt{f})$ *are written* **last**.f *and terms* $\mathsf{select}(\mathtt{oldHeap}, \mathtt{f})$ *are written* **old**.f*.*

[6] Technically, there is a set of **select** functions, one per sort. We simplify the presentation here and assume a single selection function of Any sort and an appropriate casting mechanism

Definition 3.28 (Semantics of Terms)**.** *Let I be a map from function names to functions and from predicate names to predicates. Let β be a map from logical variables to semantic values. The evaluation of terms in an object state $\binom{\sigma}{\rho}$ is as follows*

$$[\![x]\!]_{\binom{\sigma}{\rho},I,\beta} ::= \beta(x)$$

$$[\![\mathsf{if}\ \varphi\ \mathsf{then\ t\ else\ t'}]\!]_{\binom{\sigma}{\rho},I,\beta} ::= \textit{if } \binom{\sigma}{\rho}, I, \beta \models \varphi \textit{ then } [\![\mathsf{t}]\!]_{\binom{\sigma}{\rho},I,\beta}\textit{, otherwise } [\![\mathsf{t'}]\!]_{\binom{\sigma}{\rho},I,\beta}$$

$$[\![\mathsf{v}]\!]_{\binom{\sigma}{\rho},I,\beta} ::= \begin{cases} \sigma(v) & \textit{if } \mathsf{v} \neq \mathsf{heap} \\ \rho & \textit{if } \mathsf{v} = \mathsf{heap} \end{cases}$$

$$[\![f(\mathsf{t}_1,\ldots,\mathsf{t}_n)]\!]_{\binom{\sigma}{\rho},I,\beta} ::= I(f)([\![\mathsf{t}_1]\!]_{\binom{\sigma}{\rho},I,\beta},\ldots,[\![\mathsf{t}_n]\!]_{\binom{\sigma}{\rho},I,\beta})$$

The semantics for the evaluation of the condition of the if term is the semantics of a formula, which we define next.

Definition 3.29 (Semantics of Formulas)**.** *Let $\binom{\sigma}{\rho}$ be a state without symbolic values. The satisfiability relation $\binom{\sigma}{\rho}, I, \beta \models \varphi$ is defined as follows*

$$\binom{\sigma}{\rho}, I, \beta \models p(\mathsf{t}_1,\ldots,\mathsf{t}_n) \iff I\big(p\big)\big([\![\mathsf{t}_1]\!]_{\binom{\sigma}{\rho},I,\beta},\ldots,[\![\mathsf{t}_n]\!]_{\binom{\sigma}{\rho},I,\beta}\big)$$

$$\binom{\sigma}{\rho}, I, \beta \models \varphi_1 \vee \varphi_2 \iff \binom{\sigma}{\rho}, I, \beta \models \varphi_1 \textit{ or } \binom{\sigma}{\rho}, I, \beta \models \varphi_2$$

$$\binom{\sigma}{\rho}, I, \beta \models \neg\varphi \iff \binom{\sigma}{\rho}, I, \beta \models \varphi \textit{ does not hold}$$

$$\binom{\sigma}{\rho}, I, \beta \models \exists x \in S.\ \varphi \iff \textit{there is some } y \textit{ in } S \textit{ such that} \binom{\sigma}{\rho}, I, \beta[x \mapsto y] \models \varphi$$

$$\binom{\sigma}{\rho}, I, \beta \models \mathsf{t} \doteq \mathsf{t'} \iff [\![\mathsf{t}]\!]_{\binom{\sigma}{\rho},I,\beta} = [\![\mathsf{t'}]\!]_{\binom{\sigma}{\rho},I,\beta}$$

We use a fixed interpretation I that maps each function symbol to its natural semantic counterpart and omit it from the evaluation. Concerning the heap functions we demand the following connection axiom, for all heaps h, all fields $\mathtt{f}$ and terms t.

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

For the full axiomatization we refer to Beckert et al. [8]. We furthermore assume that all logical variables are unique and that the type and number of parameters for functions and predicates is adhered too. We shorten comparisons for terms of `Bool` types by writing, e.g., $i > j$ instead of $i > j \doteq \mathbf{True}$.

Example 3.12. *The following expresses that in a given state, there is a positive entry in the list stored in* **this**`.f` *and this entry is equal to the sum of the variable* v *and its index.*

$$\exists i \in \mathbb{N}.\ \mathbf{this}.\mathtt{f}[i] > 0 \wedge \mathsf{v} + i \doteq \mathbf{this}.\mathtt{f}[i]$$

Note that `Int` *is semantically mapped to the integers and thus* $\mathsf{v} + i$ *is well-typed and well-defined.*

3.5.2 Local Monadic Second-Order Trace Logic (lMSOT)

Local Monadic Second-Order Trace Logic (lMSOT) expresses properties of local traces. Its models are local tracea and the whole semantic domain, to allow to quantify over method namesy etc. Additionally to standard MSO constructs, we use $[\mathsf{t}_{\mathsf{tr}}] \doteq \mathsf{evt}$ to say that the event at position t_{tr} of the trace is equal to the event term evt. Similarly, $[\mathsf{t}_{\mathsf{tr}}] \vdash \varphi$ expresses that the state at position t_{tr} is a model for the lFOS formula φ. Both these constructs evaluate to false, if the element at position t_{tr} is not an event (or state).

Definition 3.30 (Syntax)**.** *Let* op *range over effects (see Def. 3.2), p range over predicate symbols, f over function symbols, x over logical variables and S over sorts. As sorts we take all sorts of lFOS and additionally* Eff*,* P*,* **I***,* M*, where* Eff *is the set of all semantic effects,* P *is the set of program-point identifiers,* **I** *is the set of all traces indices and* M *the set of all method names. Formulas* ψ*, terms* t_{tr} *and event terms* evt *are defined by the following grammar.*

$$\begin{aligned}
\psi &::= p(\overrightarrow{\mathsf{t}_{\mathsf{tr}}}) \mid \psi \vee \psi \mid \neg\psi \mid \mathsf{t}_{\mathsf{tr}} \subseteq \mathsf{t}_{\mathsf{tr}} \mid \exists x \in S.\ \psi \mid \exists X \subseteq S.\ \psi \mid [\mathsf{t}_{\mathsf{tr}}] \doteq \mathsf{evt} \mid [\mathsf{t}_{\mathsf{tr}}] \vdash \varphi \\
\mathsf{t}_{\mathsf{tr}} &::= x \mid f(\overrightarrow{\mathsf{t}_{\mathsf{tr}}}) \mid \mathsf{singleton}(\mathsf{t}_{\mathsf{tr}}) \mid \mathsf{t}_{\mathsf{trop}} \\
\mathsf{evt} &::= \mathsf{invEv}(\mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}) \mid \mathsf{invREv}(\mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}) \mid \mathsf{futEv}(\mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}) \\
&\quad \mid \mathsf{futREv}(\mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}) \mid \mathsf{condEv}(\mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}) \mid \mathsf{condREv}(\mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}) \\
&\quad \mid \mathsf{suspEv}(\mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}) \mid \mathsf{suspREv}(\mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}, \mathsf{t}_{\mathsf{tr}}) \mid \mathsf{noEv}(\mathsf{t}_{\mathsf{tr}})
\end{aligned}$$

The term $\mathsf{t}_{\mathsf{trop}}$ is interpreted as a semantic effect with effect op. We write $\mathsf{singleton}(\mathsf{t}_{\mathsf{tr}}) \subseteq \mathsf{t}'_{\mathsf{tr}}$ as $\mathsf{t}_{\mathsf{tr}} \in \mathsf{t}'_{\mathsf{tr}}$ and use a number of predefined predicates and functions and abbreviations

- Two predicates $\mathsf{isEvent}(i)$ and $\mathsf{isState}(i)$ over indices with their obvious interpretation.
- A function symbol $\mathsf{singleton}(\mathsf{t}_{\mathsf{tr}})$ whose interpretation maps an element to a set containing only this element.
- A function symbol $\cup$ with the obvious semantics.
- For each type of event, we assume a predicate that holds iff the given position is an event of that kind, for some parameters, e.g.,

$$\mathsf{isfutREv}(i) \iff \exists f \in \mathtt{Fut}.\ \exists \mathtt{x} \in \mathsf{Obj}.\ \exists \mathtt{m} \in \mathsf{M}.\ \exists v \in \mathsf{Any}.\ [i] \doteq \mathsf{futREv}(\mathtt{x}, f, \mathtt{m}, v).$$

- A function last that denotes the last index of the trace. Its unrolling introduces quantifiers, e.g.,

$$[\mathsf{last}] \vdash \varphi \equiv \exists i \in \mathbf{I}.\ \forall j \in \mathbf{I}.\ j \leq i \wedge \varphi$$

  Similarly, we use $\mathsf{t}_{\mathsf{tr}} + 1$ and $\mathsf{t}_{\mathsf{tr}} - 1$ to denote the successor and predecessor of an index.
- A function $\mathsf{eff}(i)$ over indices that denotes the effect set of the event at the ith position. E.g.,

$$1_{\mathsf{R}} \in \mathsf{eff}(i) \equiv \exists t \in \mathsf{Any}.\ \mathsf{isEvent}(i) \wedge ((i \doteq \mathsf{noEv}(t) \rightarrow 1_{\mathsf{R}} \in t) \wedge (\mathsf{isfutEv}(i) \rightarrow \ldots)\ldots).$$

Definition 3.31 (Semantics of Terms)**.** *The semantics of terms and event terms is straightforward. Note that lMSOT-terms do not contain program variables or fields, thus the semantics of termss depend only on the variable assignment and the function symbol interpretation.*

$$\begin{aligned}
[\![x]\!]_{I,\beta} &= \beta(x) \qquad \left[\!\!\left[f(\overrightarrow{\mathsf{t}_{\mathsf{tr}}})\right]\!\!\right]_{I,\beta} = I(f)(\overrightarrow{[\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta}}) \qquad [\![\mathsf{singleton}(\mathsf{t}_{\mathsf{tr}})]\!]_{I,\beta} = \{[\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta}\} \\
[\![\mathsf{invEv}(\mathsf{t}_{\mathsf{tr}1}, \ldots, \mathsf{t}_{\mathsf{tr}6})]\!]_{I,\beta} &= \mathsf{invEv}([\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}, \ldots, [\![\mathsf{t}_{\mathsf{tr}6}]\!]_{I,\beta}) \\
[\![\mathsf{invREv}(\mathsf{t}_{\mathsf{tr}1}, \ldots, \mathsf{t}_{\mathsf{tr}5})]\!]_{I,\beta} &= \mathsf{invREv}([\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}, \ldots, [\![\mathsf{t}_{\mathsf{tr}5}]\!]_{I,\beta}) \\
[\![\mathsf{futEv}(\mathsf{t}_{\mathsf{tr}1}, \ldots, \mathsf{t}_{\mathsf{tr}5})]\!]_{I,\beta} &= \mathsf{futEv}([\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}, \ldots, [\![\mathsf{t}_{\mathsf{tr}5}]\!]_{I,\beta}) \\
[\![\mathsf{futREv}(\mathsf{t}_{\mathsf{tr}1}, \ldots \mathsf{t}_{\mathsf{tr}5})]\!]_{I,\beta} &= \mathsf{futREv}([\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}, \ldots, [\![\mathsf{t}_{\mathsf{tr}5}]\!]_{I,\beta}) \\
[\![\mathsf{condEv}(\mathsf{t}_{\mathsf{tr}1}, \ldots, \mathsf{t}_{\mathsf{tr}3})]\!]_{I,\beta} &= \mathsf{condEv}([\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}, \ldots, [\![\mathsf{t}_{\mathsf{tr}3}]\!]_{I,\beta}) \\
[\![\mathsf{condREv}(\mathsf{t}_{\mathsf{tr}1}, \ldots, \mathsf{t}_{\mathsf{tr}3})]\!]_{I,\beta} &= \mathsf{condREv}([\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}, \ldots, [\![\mathsf{t}_{\mathsf{tr}3}]\!]_{I,\beta}) \\
[\![\mathsf{suspEv}(\mathsf{t}_{\mathsf{tr}1}, \ldots, \mathsf{t}_{\mathsf{tr}4})]\!]_{I,\beta} &= \mathsf{suspEv}([\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}, \ldots, [\![\mathsf{t}_{\mathsf{tr}4}]\!]_{I,\beta}) \\
[\![\mathsf{suspREv}(\mathsf{t}_{\mathsf{tr}1}, \ldots, \mathsf{t}_{\mathsf{tr}4})]\!]_{I,\beta} &= \mathsf{suspREv}([\![\mathsf{t}_{\mathsf{tr}1}]\!]_{I,\beta}, \ldots, [\![\mathsf{t}_{\mathsf{tr}4}]\!]_{I,\beta}) \\
[\![\mathsf{noEv}(\mathsf{t}_{\mathsf{tr}})]\!]_{I,\beta} &= \mathsf{noEv}([\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta})
\end{aligned}$$

Definition 3.32 (Semantics of Formulas)**.** *The semantics of lMSOT-formulas is analogous to the one of lFOS, except that the model is a local trace and the constructs specific to MSO and our extension:*

$$\theta, I, \beta \models \exists X \subseteq S.\ \psi \iff \text{there is a subset } \mathbf{X} \text{ of } S \text{ such that } \theta, I, \beta[X \mapsto \mathbf{X}] \models \psi$$
$$\theta, I, \beta \models \mathsf{t}_{\mathsf{tr}} \subseteq \mathsf{t}'_{\mathsf{tr}} \iff [\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta} \subseteq [\![\mathsf{t}'_{\mathsf{tr}}]\!]_{I,\beta}$$
$$\theta, I, \beta \models [\mathsf{t}_{\mathsf{tr}}] \doteq \mathsf{evt} \iff 1 \leq [\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta} \leq |\theta| \wedge \theta\big[\, [\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta} \,\big] = [\![\mathsf{evt}]\!]_{I,\beta}$$
$$\theta, I, \beta \models [\mathsf{t}_{\mathsf{tr}}] \vdash \varphi \iff 1 \leq [\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta} \leq |\theta| \wedge \theta\big[\, [\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta} \,\big] \text{ is a state } \wedge\, \theta\big[\, [\![\mathsf{t}_{\mathsf{tr}}]\!]_{I,\beta} \,\big], I, \emptyset \models \varphi$$

Relativization [68] syntactically restricts a formula on a substructure of the original model. This substructure is defined by another formula. We later use relativization extensively to define the semantics of composed specifications, where the semantics of the composed specification is expressed in terms of the *relativized* semantics of its components.

Definition 3.33 (Relativization)**.** *Let φ, ψ be two formulas. Let y be a free variable of sort D in ψ. The relativization of φ on ψ with respect to y, written $\varphi[y \in D \setminus \psi(y)]$ is defined as follows:*

$$(\psi \wedge \psi')[y \in S \setminus \psi''(y)] = \psi[y \in S \setminus \psi''(y)] \wedge \psi'[y \in S \setminus \psi''(y)]$$
$$(\neg\psi)[y \in S \setminus \psi'(y)] = \neg(\psi[y \in S \setminus \psi'(y)])$$
$$\mathsf{t}_{\mathsf{tr}} \subseteq \mathsf{t}'_{\mathsf{tr}}[y \in S \setminus \psi(y)] = \mathsf{t}_{\mathsf{tr}} \subseteq \mathsf{t}'_{\mathsf{tr}}$$
$$p(\overrightarrow{\mathsf{t}_{\mathsf{tr}}})[y \in S \setminus \psi(y)] = p(\overrightarrow{\mathsf{t}_{\mathsf{tr}}})$$
$$([\mathsf{t}_{\mathsf{tr}}] \doteq \mathsf{evt})[y \in S \setminus \psi(y)] = [\mathsf{t}_{\mathsf{tr}}] \doteq \mathsf{evt}$$
$$([\mathsf{t}_{\mathsf{tr}}] \vdash \varphi)[y \in S \setminus \psi(y)] = [\mathsf{t}_{\mathsf{tr}}] \vdash \varphi$$
$$(\exists x \in S.\ \psi)[y \in S' \setminus \psi'(y)] = \begin{cases} \exists x \in S.\ (\psi'(x) \wedge \psi[y \in S' \setminus \psi'(y)]) & \text{if } S = S' \\ \exists x \in S.\ \psi[y \in S' \setminus \psi'(y)] & \text{otherwise} \end{cases}$$
$$(\exists X \subseteq S.\ \psi)[y \in S' \setminus \psi'(y)] = \begin{cases} \exists X \subseteq S.\ (\forall x \in X.\ \psi'(x) \wedge \psi[y \in S' \setminus \psi'(y)]) & \text{if } S = S' \\ \exists X \subseteq S.\ \psi[y \in S' \setminus \psi'(y)] & \text{otherwise} \end{cases}$$

Example 3.13. *The following expresses that a trace contains only* noEv *events.*

$$\psi = \forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \rightarrow [i] \doteq \mathsf{noEv}$$

The following expresses that the trace up to position j, a trace contains only noEv *events.*

$$\psi[k \in \mathbf{I} \setminus k < j] \equiv \forall i \in \mathbf{I}.\ \big(i < j \rightarrow \big(\mathsf{isEvent}(i) \rightarrow [i] \doteq \mathsf{noEv}\big)\big)$$

Validity can be restricted in several layers, with general validity being the classical one, that demands that every trace is a model for a formula.

Definition 3.34. *We use the standard notion of free variables. A trace θ is a model for a lMSOT formula ψ, if $\theta, I, \beta \models \psi$ holds for an empty variable assignment β. A lMSOT formula ψ without free variables is valid if every local trace is a model for ψ.*

This is not a useful notion of validity, given that most local traces cannot be generated by any method. However, we can restrict validity and only demand that the possible set of traces must be models.

Definition 3.35. *Let $\mathtt{m}$ be a method. A lMSOT formula ψ without free variables is $\mathtt{m}$-valid if every trace $\theta \in \mathsf{selected}(\mathtt{m})$ (from any program containing $\mathtt{m}$) is a model for ψ.*

For example, taking the code from Example 3.8, the following formula is $\mathtt{m}$-valid, because obviously it never reads from a future. It is not valid in general.

$$\forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \rightarrow \neg \mathsf{isfutREv}(i)$$

There are also formulas which are $\mathtt{m}$-valid for all possible methods, but not generally valid. E.g., that there is a terminating event.

$$\exists i \in \mathbf{I}.\ \mathsf{isfutEv}(i)$$

In example 3.8, we observed that the final value of `i` is always at least 11. Yet, the following formula is not $\mathtt{m}$-valid.

$$\exists i \in \mathbf{I}.\ \mathsf{last}(i) \wedge [i-1] \vdash \texttt{this.i} > 10$$

It is, however, $\mathtt{m}$-valid for every program that has no other method in its class. We formalize such restrictions with contextual validity.

Definition 3.36. *Let $\mathtt{m}$ be a method and P be a set of programs containing $\mathtt{m}$.*
A lMSOT formula ψ without free variables is $\mathtt{m}$-P-valid if every trace $\theta \in \mathsf{selected}(\mathtt{m})$ (from any program in P) is a model for ψ.

Finally, we have validity for statements within methods in some set of programs.

Definition 3.37. *A lMSOT formula ψ without free variables is $\mathtt{s}$-$\mathtt{m}$-P-valid if for every object state $\binom{\sigma}{\rho}$ and every trace $\theta \in \mathsf{selected}(\mathtt{s},\mathtt{m})$ (from any program in P) is a model for ψ.*

3.5.3 Global First-Order State (gFOS) and Monadic Second-Order Trace Logic (gMSOT)

Analogous to the previous definitions, we give logics gFOS and gMSOT to reason about global states and traces. The definitions are straightforward and we refrain from introducing all the formalism in full detail.

Definition 3.38 (Syntax)**.** *The syntax is analogous to the one of local logics, but we also include object names $\mathtt{x}$ as terms for gFOS. Formulas φ^{g} of gFOS, terms t^{g} of gFOS, formulas ψ^{g} of gMSOT, terms $\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}$ of gMSOT and event terms evt of gMSOT are defined by the following grammar.*

$$\varphi^{\mathsf{g}} ::= p(\overrightarrow{\mathsf{t}^{\mathsf{g}}}) \mid \varphi^{\mathsf{g}} \vee \varphi^{\mathsf{g}} \mid \neg\varphi^{\mathsf{g}} \mid \exists x \in S.\ \varphi^{\mathsf{g}} \qquad \mathsf{t}^{\mathsf{g}} ::= x \mid \mathtt{heap} \mid \mathtt{f} \mid f(\overrightarrow{\mathsf{t}^{\mathsf{g}}}) \mid \mathtt{x}$$

$$\begin{aligned}\psi^{\mathsf{g}} ::= {} & p(\overrightarrow{\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}}) \mid \psi^{\mathsf{g}} \vee \psi^{\mathsf{g}} \mid \neg\psi^{\mathsf{g}} \mid \exists x \in S.\ \psi^{\mathsf{g}} \mid \exists X \subseteq S.\ \psi^{\mathsf{g}} \\ & \mid [\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}] \doteq \mathsf{evt} \mid [\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}] \vdash \varphi^{\mathsf{g}} \mid \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}} \subseteq \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}} \mid \mathsf{singleton}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}})\end{aligned}$$

$$\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}} ::= x \mid f(\overrightarrow{\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}})$$

$$\begin{aligned}\mathsf{evt} ::= {} & \mathsf{invEv}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \overrightarrow{\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}}) \mid \mathsf{invREv}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \overrightarrow{\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}}) \mid \mathsf{futEv}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}) \mid \mathsf{futREv}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}) \\ & \mid \mathsf{condEv}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}) \mid \mathsf{condREv}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}) \mid \mathsf{suspEv}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}) \\ & \mid \mathsf{suspREv}(\mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}, \mathsf{t}^{\mathsf{g}}_{\mathsf{tr}}) \mid \mathsf{noEv}\end{aligned}$$

There is only one program variable. This variable `heap` is now a map from object names to a map from field names to semantics values. We call this sort global heaps. We demand that following function symbols for global heaps exist, as we may now refer to multiple objects.

$$\mathsf{select}(\mathsf{t}^{\mathsf{g}}, \mathsf{t}^{\mathsf{g}}, \mathsf{t}^{\mathsf{g}}) \mid \mathsf{store}(\mathsf{t}^{\mathsf{g}}, \mathsf{t}^{\mathsf{g}}, \mathsf{t}^{\mathsf{g}}, \mathsf{t}^{\mathsf{g}}) \mid \mathsf{anon}(\mathsf{t}^{\mathsf{g}})$$

Where $\mathsf{select}(h, \mathtt{x}, \mathtt{f})$ selects the value of field $\mathtt{f}$ of object $\mathtt{x}$ from heap h, $\mathsf{store}(h, \mathtt{x.f}, \mathsf{t})$ stores the value of t in field $\mathtt{f}$ of object $\mathtt{x}$ in heap h. Finally, $\mathsf{anon}(h)$ is a global heap that may be different from global heap h. We write $\mathtt{x.f}$ for $\mathsf{select}(\mathtt{heap}, \mathtt{x}, \mathtt{f})$.

Definition 3.39 (Semantics)**.** *The semantics of gFOS terms is defined as a function* $[\![\cdot]\!]_{\mathbf{g},I,\beta}$*, where* $\mathbf{g}$ *is a global state. The definition is analogous to the one of lFOS, except for program variables:*

$$[\![\mathtt{heap}]\!]_{\mathbf{g},I,\beta} = \mathbf{g}$$

The semantics of gFOS formulas is a relation $\mathbf{g}, I, \beta \models \varphi^{\mathrm{g}}$ *analogous to the one of lFOS. The semantics of gMSOT terms is a function* $[\![\cdot]\!]_{I,\beta}$*, and the semantics of gMSOT formulas is a relation* $\gamma, I, \beta \models \psi^{\mathrm{g}}$*, where* γ *is a global trace.*

We use a fixed interpretation I, that maps each function symbol to its natural semantic counterpart. Concerning the heap functions we demand only the following connection axiom, for all global heaps h, objects $\mathtt{x}$ all fields $\mathtt{f}$ and terms t.

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

Example 3.14. *The following formula expresses that whenever object* $\mathtt{x}$ *has a positive value stored in* $\mathtt{f}$*, then so does object* $\mathtt{x}'$ *in the next state.*

$$\forall i \in \mathbf{I}.\ \big([i] \vdash \mathtt{x.f} > 0 \land \neg \mathsf{last}(i)\big) \rightarrow [i+2] \vdash \mathtt{x}'\mathtt{.f} > 0\big)$$

Relativization and the notions of validity are analogous to the local logics, we only give validity.

Definition 3.40. *We use the standard definition of free variables. A global trace* γ *is a model for a gMSOT formula* ψ^{g}*, if* $\gamma, I, \beta \models \psi^{\mathrm{g}}$ *holds, for an empty variable assignment* β*. A gMSOT formula* ψ *without free variables is valid if every global trace is a model for* ψ*.*

Let Prgm *be a program. A gMSOT* ψ^{g} *without free variables is* Prgm*-valid if every trace* γ *that is realized by* Prgm *is a model for* ψ^{g}*. A gMSOT formula* ψ^{g} *without free variables is P-valid for some set P of program, if it is* Prgm*-valid for every* $\mathsf{Prgm} \in P$*.*

Example 3.15. *The following formula is not valid, but valid for all well-typed programs.*

$$\exists i \in \mathbf{I}.\ [i] \vdash \mathsf{True}$$

Sentences are defined analogously for all semantic logics:

Definition 3.41. *A lMSOT, gMSOT, lFOS or gFOS formula* φ *is a* sentence *if it contains no free variable.*

3.6 Discussion

In this section we discuss some of the design decisions made in the semantics.

On the Design and Limitations of CAO

Object Creation.

Object creation is restricted in CAO: the main block uses Rebeca-style initialization and one asynchronous method call and dynamic object creation is not allowed. The reason for this design decision requires a discussion of some concepts of the later chapter 7: Session Types for Active Objects so far lack the ability to assign roles to newly created objects, e.g., by role delegation [39], which is used in Session Types for channel-based concurrency models. Furthermore, channel-based concurrency models isolate a session by grouping a set of process by a shared channel. The session is thus *isolated* from the rest of the system, but processes may participate in multiple sessions [12]. It is harder to isolate sessions without channels, especially if the concurrency model allows interactions via the heap memory. As it is out of scope for this work to lift all possible extensions of Session Types from channel-based concurrency models to Active Objects, we regard the whole system as one session. Din et al. [45] isolate sessions in Active Objects, but introduce a new transaction mechanism to enforce isolation at runtime, while this work gives a fully static system.

Runtime Errors.

Our language does not include exceptions. In contrast, ABS has three implicitly thrown exceptions `NullPointerException`, `PatternMatchException`, `DivisionByZeroException`:

- `NullPointerException`. In CAO, all locations of class type are initialized at program start with non-null values. As there is no **null** constant, it is not possible to raise such an exception.
- `PatternMatchException`. This is thrown by the functional sublanguage of ABS when a deconstruction of an value with a abstract data type(ADT) is not possible.
- `DivisionByZeroException`. This exception is self-explaining.

CAO has only the `List` ADT, so runtime errors can only occur by evaluating `head(Nil)` and `e/0`. In this case the evaluation is not defined, no rule can be applied and the object blocks.

The reasoning system we present in the following chapters is only concerned about *partial* functional correctness: correctness in case of normal termination [8]. Thus, the ramifications of exception handling would offer no insights. Furthermore, their treatment in dynamic logic is well-explored. Excluding them vastly simplifies the assignment rules for symbolic execution, because there is no need for program transformation that ensure that the right-hand-side of assignments is a non-nested expression.

On the LAGC Semantics

Compared to the original formalization of Din et al. [43], we have made some adaptations.

The biggest difference is that we have no continuations. Din et al. use such special markers at points where input is required, instead of using fresh symbolic values and similarly, other markers to mark processes as starving or blocked. These markers are used to control interleavings in the external semantics, which have only two layers (method local, system global) instead of three (method local, object local, system global) and always add as much of a local trace as possible. We replaced this with our agreement mechanism that chops up a trace step by step.

With continuations, the semantics of a method is less clear and the resulting traces are harder to reason about (in the sense that they are not straightforward models for our semantic logics). For example, the semantics of `v = f.get; if(v>0) i=v else i=-v; return i` has two traces in our semantics, one for each branch of the **if** statement. Din et al. have three traces: two traces for the branches of the **if** statement, in case that the future in `f` has already been resolved, and one trace of the following form for the case that it is not[7]:

$$\emptyset \triangleright \left\langle \binom{\sigma}{\rho}, blkEv(\mathtt{X}, [\![f]\!]_{\mathtt{X},dest,\mathtt{m},\binom{\sigma}{\rho}}), blkCont(\mathtt{X}, dest, \texttt{v = f.get; if(v>0) i=v else i=-v; return i}) \right\rangle$$

The continuation marker *blkCont* is used to generate further local traces (here: two more, thus four traces in total, despite the statement having only two branches) by the global semantics. It is, thus, harder to talk about local traces in isolation, as one has to handle continuations in the traces. However, there is no need for the more complex handling of symbolic fields in our system: the fresh symbolic heap during generation can be seen as generating the continuations already at that point.

Without continuations, we can (1) have a standard notion of traces as sequences of all states and events occurring during execution, which are more suitable for our semantic logics and (2) can give a straightforward rule for sequential composition. The rule of the original LAGC system is the most complex rule in their system, as it has to deal with starvation and blocking continuations *already at local level*; (3) clearly separate local and global semantics: contrary to the original LAGC system our LA semantics is connected to the GC semantics only when creating new processes. Continuations require to invoke the LA semantics for every global step resolving a continuation.

[7] We refrain from using the original syntax for readability's sake.

Din et al. do not discuss selectability, which we propose is easier with our approach than with continuations. We also do not model starvation explicitly. Instead of adding a starvation marker to the local semantics, it is modeled explicit as a process that is never scheduled again. We propose that starvation is a property that is visible at object level and should, thus, not be modeled at method level. Our approach is also nearer to the definition of starvation used by static analyses [78].

Finally, we do not use a well-formedness predicate to ensure that basic consistency conditions are uphold, e.g., that a method starts only after it is called. The only places where explicit checks are needed are to ensure future freshness and that the correct value is read from a future.

Step Size.

Agreement always makes a single step, i.e., it adds a trace with one event. Instead, one can extended the candidate trace that is added until the next symbolic value (or marker). A method without future reads and suspension would be executed in just one step of the object semantics. This has the disadvantages that global interleavings are not visible in the global trace. Consider two methods without future reads and suspensions being called on two different objects. All its symbolic traces have length n. In the current semantics generates 2^n global traces, the semantics sketched here would generate 2. This would not allow us to express global properties such as *"the two statements run in parallel"* as properties of one global trace.

Effects.

In the given system, effects are parts of events. An alternative would be to consider events as effects themselves. The local traces would then alternate between object states and sets of effects (instead of alternating between object states and events). The reason we decided to design the semantics with effects in events is that (1) it allows us to build upon the established terminology of LAGC and (2) for all verification systems we give later, events are of crucial importance, while the semantics effects are only of relevance for the effect type systems in chapter 5.

On the Semantic Logics

We stress that both MSO logics are *not* MSO over finite words, because we include semantic values into the structure. As we give no calculus for the semantic logics, their choice is mainly a matter of preference and the use of full second order logic or a temporal logic is also possible in principle. One reason for MSO is that relativization for MSO is straightforward and quantifiers make it easier to express global properties (i.e., properties that relate potentially far away parts of the trace). Relativization has proven useful to describe complex, temporal properties of traces. One can use a more expressive logic instead, e.g., full second-order logic. We refrain from doing so as we do not need the expressive power. Similarly, one can use a temporal logic or any other semantic system instead of lMSOT and gMSOT, the framework we give in this work does not rely on the specific choice of semantic logic.

4 Behavioral Program Logic

This chapter introduces Behavioral Program Logic (BPL), a dynamic trace logic which extends lFOS to reason about lMSOT properties of programs. To do so, lFOS uses behavioral specifications τ, which are syntactic representations of lMSOT formulas, and behavioral modalities $[\mathsf{s} \Vdash \tau]$. The semantics of a behavioral modality is that every selected trace of s is a model for the lMSOT formula represented by τ.

BPL is, thus, a dynamic first-order logic and it is not necessary to give a second-order reasoning system. Second-order reasoning is only needed to prove soundness of the calculi we give for BPL and we do so by giving semantic arguments directly about the models.

The split between (semantic) trace property and syntactic representation enables the (simple) design of calculi to automatically verify these properties is a view on behavioral types [1, 74]. Behavioral types aim "*to describe properties associated with the behavior of programs and in this way also describe how a computation proceeds.*" [74] and are designed together with a type system.

The focus on the syntactic representation of trace properties is the first conceptual similarity to behavioral types in our program logic — in the sections below we emphasize further points where techniques from (behavioral) type systems influenced the design of our program logic. In the discussion section at the end of this chapter we give a more in-depth discussion about the notion of design and the relation between behavioral types and program logics.

It is also worth noting that BPL is used for local reasoning: it does not express global properties about the behavior of programs, only local properties about the behavior of processes. To compose the local results to global ones, we use well-established patterns: method contracts, object invariants, protocols. Again, the second-order reasoning has only to be done to establish soundness of composition (for a given pattern).

Behavioral specifications also allow us to have an expressive logic to model the properties established by (or required from) static analyses, as the semantics only needs to describe the results of the analysis, instead of their complex inner-workings. Reasoning about results can be used in verification in our logical framework, with the syntactical representation being the interface to external analysis.

4.1 Syntax and Semantics

A behavioral specification is a pair of a *behavioral language* and a semantics that maps elements of the language to lMSOT formulas.

Definition 4.1 (Behavioral Specification)**.** *A* behavioral specification $\mathbb{T}$ *is a pair* $(\tau_{\mathbb{T}}, \alpha_{\mathbb{T}})$*, where* $\alpha_{\mathbb{T}}$ *maps elements of* $\tau_{\mathbb{T}}$ *to lMSOT formulas.*

We call elements of the behavioral language also behavioral specifications. Intuitively, a behavioral language is a representation of a fragment of lMSOT. Such fragments are embedded into lFOS with behavioral modalities $[\mathsf{s} \Vdash^{\alpha_{\mathbb{T}}} \tau]$, where $\tau_{\mathbb{T}}$ is some set of specification syntax and $\tau \in \tau_{\mathbb{T}}$. A behavioral modality that describes all selected traces of s starting in the current state are models for $\alpha_{\mathbb{T}}(\tau)$.

Definition 4.2 (Syntax of BPL)**.** BPL*-formulas* φ*, terms* t *and updates* U *are defined by the following grammar, which extends Def. 3.27. The meta variables range as in Def. 3.27. Additionally let* s *range over statements,* $\mathbb{T}$ *over behavioral specifications and* τ *over* $\tau_{\mathbb{T}}$*.*

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

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

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

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

$$\binom{\sigma}{\rho}, I, \beta \models [\mathrm{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau] \Leftrightarrow \forall \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \alpha_{\mathbb{T}}(\tau)$$

Figure 4.1: Semantics of BPL. The satisfiability relation on the right of the semantics of behavioral modalities is the one of lMSOT.

To be more precise, we give a family of behavioral modalities (one per behavioral specification) and the signature of a BPL logic consists of the program variables, function symbols, predicate symbols and used behavioral specifications. As the signature is always clear from the context we do not give it explicitly.

Besides behavioral modalities, BPL extends lFOS with *updates* [5] to keep track of state changes, as decomposing, e.g., a sequential composition $\mathrm{s}_1;\mathrm{s}_2$ to reason about the sub-selected traces of s_2 requires to keep track of the last state of the sub-selected traces of s_1 (because at this state, the two traces are chopped together[1]). In the proof system we give, updates can be seen as representations of syntactic substitutions.

An elementary update $\mathrm{v} := \mathrm{t}$ updates the program variable to the value of term t, an empty update ϵ does not change the state. Parallel updates $U_1||U_2$ apply U_1 and U_2 in parallel, such that U_2 wins in case of conflicts because it can overwrite the change of U_1. An application of an update U, written $\{U\}\cdot$, to another formula, term or update expresses to first apply U to the state, and then evaluate the formula, term or update.

The semantics of updates is based on Beckert et al. [8] and realizes the intuition given above. As in the semantics of lFOS we require that the method in question is implicitly known.

Definition 4.3 (Semantics of BPL)**.** *The semantic extension of lFOS to* BPL *is given in Fig. 4.1.*

We stress that we replace lFOS by BPL everywhere in the semantics – in particular we replace it in the semantics of the $\vdash$ operator of lMSOT. There are no restrictions on the syntax of $\tau_{\mathbb{T}}$, in particular we allow program terms (expressions, statements, etc) and BPL terms and formulas. If the syntax of $\tau_{\mathbb{T}}$ contains logical first order variables they can be bound in the surrounding BPL formula.

The use of program elements is the second resemblance of behavioral types – if the behavioral specification contains program elements, one can syntactically match statement and type in the modality in the proof rules. This is not common in dynamic logics: previous dynamic trace logics [17, 8, 44] require to resolve the modality completely by symbolic execution, even if the very first statement already violates the specification. In contrast, fail-early is a common feature of type systems, even though type systems may be designed to uncover further type errors as well.

We give our first example for a behavioral specification: postconditions.

[1] The name of the chop is taken from Din et al. [43] and can be traced back to Chandra et al. [27] where it was used to disconnect traces. We realize it is named rather unfortunately to denote an operator that can be seen as *connecting* two traces, but we keep it for consistency.

Definition 4.4 (Postconditions)**.** *The behavioral specification* $\mathbb{T}_{\mathsf{pst}} = (\tau_{\mathsf{pst}}, \alpha_{\mathsf{pst}})$ *for postconditions is the pair of the set of all lFOS sentences and the function* α_{pst}*, defined below.* `D` *is the return type of the method in question. The underscores denote existentially bound logical variables, omitted for readability.*

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

The split in the semantics handles the special function symbol `result` if it occurs, and substitutes it with the returned value of the method. $\mathbb{T}_{\mathsf{pst}}$ allows us to immediately lift results from standard dynamic logic $[\mathtt{s}]\varphi$ into our system, as long as φ contains no modalities. We give an extended discussion on nested modalities in BPL in the next section of this chapter. We can encode a similar fragment (postcondition contains no modalities and no updates) of dynamic trace logic (DTL) [6]. In DTL $[\mathtt{s}]\varphi$ expresses that every trace follows the linear temporal logic (LTL) formula φ. We encode LTL as follows.

Definition 4.5 (LTL)**.** *The specification for LTL on traces is defined by the following pair:*

$$\mathbb{T}_{|\mathsf{t}|} = (\tau_{|\mathsf{t}|}, \alpha_{|\mathsf{t}|})$$

The syntax $\tau_{|\mathsf{t}|}$ *is the set of sentences of the logic defined by the following syntax, which again extends lFOS:*

$$\varphi ::= \ldots \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi \mid \Box\varphi$$

The semantics removes all events from the traces and restricts the model to the relevant subtrace:

$$\begin{aligned}
\alpha_{|\mathsf{t}|}(\varphi) &= \alpha'_{|\mathsf{t}|}(\varphi)[p \in \mathbf{I} \setminus \mathsf{isState}(p)] \\
\alpha'_{|\mathsf{t}|}(\varphi) &= \exists i \in \mathbf{I}.\ \forall j \in \mathbf{I}.\ i \geq j \wedge [i] \vdash \varphi \qquad \text{if } \varphi \text{ is an lFOS formula} \\
\alpha'_{|\mathsf{t}|}(\mathbf{X}\varphi) &= \exists i \in \mathbf{I}. \exists j \in \mathbf{I}.\ \forall k \in \mathbf{I}.\ \big((k \neq i \rightarrow k \geq j) \wedge \alpha'_{|\mathsf{t}|}(\varphi)[idx \in \mathbf{I} \setminus idx \geq j]\big) \\
\alpha'_{|\mathsf{t}|}(\Box\varphi) &= \forall i \in \mathbf{I}.\ \alpha'_{|\mathsf{t}|}(\varphi)[idx \in \mathbf{I} \setminus idx \geq i] \\
\alpha'_{|\mathsf{t}|}(\varphi_1 \mathbf{U} \varphi_2) &= \exists i \in \mathbf{I}.\ \big(\forall j \in \mathbf{I}.\ j < i \rightarrow \alpha'_{|\mathsf{t}|}(\varphi_2)[idx \in \mathbf{I} \setminus idx \geq j]\big) \wedge \\
&\qquad \alpha'_{|\mathsf{t}|}(\varphi_2)[idx \in \mathbf{I} \setminus idx \geq i]
\end{aligned}$$

Example 4.1. *For an example, we give the semantics for* $\mathbf{X}\Box(\mathtt{v == 1})$*. The formula expresses that* `x` *is 1 everywhere, but the first state.*

$$\begin{aligned}
\alpha_{|\mathsf{t}|}\big(\mathbf{X}\Box(\mathtt{v == 1})\big) = \exists i_1 \in \mathbf{I}.\ \mathsf{isState}(i_1) \wedge \exists j_1 \in \mathbf{I}.\ \mathsf{isState}(j_1) \wedge i_1 \leq j_1 \wedge \\
(\forall k \in \mathbf{I}.\ \mathsf{isState}(k) \wedge k \neq i_1 \rightarrow k \geq j_1) \wedge \\
\forall i_2 \in \mathbf{I}.\ \big((\mathsf{isState}(i_2) \wedge i_2 \geq j_1) \rightarrow [i_2] \vdash \mathtt{v == 1}\big)
\end{aligned}$$

BPL can express properties connecting multiple behavioral specifications. E.g., the following (valid) formula expresses, that if during the execution of `v = 1; while i >= 0{i = i - v;}return v;` we can show that if `v == 1` holds throughout the loop, then it also holds at its very end:

$$\begin{aligned}
\varphi_{|\mathsf{t}|} = \quad & \mathtt{i} \geq 1 \wedge [\mathtt{v=1;\ while(i >= 0)\{i = i - v;\}return\ v;} \overset{\alpha_{|\mathsf{t}|}}{\Vdash} \mathbf{X}\Box(\mathtt{v==1})] \\
\rightarrow & [\mathtt{v=1;\ while(i >= 0)\{i = i - v;\}return\ v;} \overset{\alpha_{\mathsf{pst}}}{\Vdash} (\mathtt{v==1})]
\end{aligned}$$

Behavioral modalities can be used to express any trace property expressible in lMSOT, independent of the form of its verification system. The following defines a points-to analysis (for the next statement), normally implemented in a data-flow framework, and not with a reasoning system such as the above logical systems.

Definition 4.6 (Points-To)**.** *The behavioral specification of a* points-to analysis *specifies that the next statement reads a future resolved by a method from set Mets.*

$$\mathbb{T}_{\text{p2}} = (\mathscr{P}(\mathsf{M}), \alpha_{\text{p2}}) \textit{ with}$$
$$\alpha_{\text{p2}}(\mathit{Mets}) = \exists \mathtt{x} \in \mathsf{Object}.\exists f \in \texttt{Fut}.\exists \mathtt{m} \in \mathsf{M}.\exists v \in \mathsf{Any}.\exists i \in \mathbb{N}.\ [1] \doteq \mathsf{futREv}(\mathtt{x}, f, \mathtt{m}, v, i) \wedge \bigvee_{\mathtt{m}' \in \mathit{Mets}} \mathtt{m} \doteq \mathtt{m}'$$

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

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

4.1.1 Excursus: Diamond and Nested Modalities for BPL

In this work we focus on verification of safety properties of Active Objects and use neither diamond modalities nor nested modalities. However, to justify our use of the notion of modality, we introduce both of them in this section.

The definition of the diamond modality is straightforward and if the behavioral specification is closed under negation, the relation between box and diamond is as expected.

Definition 4.7 (Diamonds)**.** *The syntax of a diamond behavioral modality is* $\langle \mathtt{s} \Vdash^{\alpha_{\mathbb{T}}} \tau \rangle$*. Its semantics is*

$$\begin{pmatrix}\sigma\\ \rho\end{pmatrix}, I, \beta \models \langle \mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau \rangle \Leftrightarrow \exists \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \begin{pmatrix}\sigma\\ \rho\end{pmatrix}\right).\ \theta, I, \beta \models \alpha_{\mathbb{T}}(\tau)$$

A behavioral specification $\mathbb{T}$ *is* closed under negation *if there is some operator* $\neg_{\mathbb{T}}$ *such that*

- *if* τ *is valid syntax, then so is* $\neg_{\mathbb{T}}\tau$ *and*
- *the operator* $\neg_{\mathbb{T}}$ *translates to negation:* $\alpha_{\mathbb{T}}(\neg_{\mathbb{T}}\tau) = \neg\alpha_{\mathbb{T}}(\tau)$

Every behavioral specification can be extended to a behavioral specification closed under negation. Diamonds in BPL behave as expected.

Lemma 4.1. *Let* $\mathbb{T}$ *be a behavioral specification closed under negation. The following equivalences hold:*

$$\neg\langle \mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \neg_{\mathbb{T}}\tau \rangle \equiv [\mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau]$$
$$\langle \mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau \rangle \equiv \neg[\mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \neg_{\mathbb{T}}\tau]$$

Furthermore, if a statement `s` *does not terminate, then* $\langle \mathtt{s} \Vdash^{\alpha_{\mathbb{T}}} \tau \rangle$ *is false for all behavioral specifications* $\mathbb{T}$*.*

Proof. See p. 167.

For nested modalities, consider an extension of $\mathbb{T}_{\text{pst}}$ that is defined not over lFOS sentences, but over BPL sentences. We denote this variant with $\mathbb{T}_{\text{bpst}}$.

Example 4.2. *A nested modality is a modality where the type contains a behavioral modality. E.g., the following formula.*

$$\left[\texttt{i = 0;} \overset{\alpha_{\text{bpst}}}{\Vdash} \left[\texttt{i = i + 1;} \overset{\alpha_{\text{bpst}}}{\Vdash} \texttt{i > 0}\right]\right]$$

Its semantics is as follows:

$$
\begin{aligned}
&\binom{\sigma}{\rho}, I, \beta \models \left[\texttt{i = 0;} \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \left[\texttt{i = i + 1;} \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \texttt{i > 0}\right]\right] \\
\iff &\forall \theta' \in \mathsf{selected}\left(\texttt{i = 0;}, \mathtt{m}, \binom{\sigma}{\rho}\right). \theta', I, \beta \models [\mathsf{last}] \vdash \left[\texttt{i = i + 1;} \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \texttt{i > 0}\right] \\
\iff &\forall \theta' \in \mathsf{selected}\left(\texttt{i = 0;}, \mathtt{m}, \binom{\sigma}{\rho}\right). \mathsf{last}(\theta'), I, \beta \models \left[\texttt{i = i + 1;} \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \texttt{i > 0}\right] \\
\iff &\forall \theta' \in \mathsf{selected}\left(\texttt{i = 0;}, \mathtt{m}, \binom{\sigma}{\rho}\right). \\
&\quad \forall \theta'' \in \mathsf{selected}\left(\texttt{i = i+1;}, \mathtt{m}, \mathsf{last}(\theta')\right). \mathsf{last}(\theta''), \models \texttt{i > 0}
\end{aligned}
$$

As we can see, nested modalities have the expected semantics, but satisfiability is now a recursive predicate. At least for $\mathbb{T}_{\mathsf{bpst}}$ we can, however, state that the semantics is always well-defined.

Lemma 4.2. *Let φ be a* BPL *formula containing only $\mathbb{T}_{\mathsf{bpst}}$ modalities. Its satisfiability relation is well-founded.*

Proof. See p. 168.

Lemma 4.3. *Nested modalities for postconditions adhere to the sequential composition axiom [66].*

$$
\left[\mathtt{s}_1 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \left[\mathtt{s}_2 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \varphi\right]\right] \equiv \left[\mathtt{s}_1; \mathtt{s}_2 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \varphi\right]
$$

Proof. See p. 168.

The above lemmas and the possibility to encode postconditions justifies our use of the term "modality". Contrary to standard modalities, behavioral modalities are not formulas that express modal statements about *formulas*, but formulas that express a modal statement about *more general specifications*. In the following we neither consider diamond behavioral modalities nor do we require that behavioral specifications are closed under negation or explicitly require the extension of lFOS to BPL in the semantics of the $\vdash$ operator for any of our behavioral specifications. A variant of $\mathbb{T}_{|\mathtt{t}|}$ which allows to nest behavioral modalities, say $\mathbb{T}_{\mathsf{b}|\mathtt{t}|}$, can be designed analogously to $\mathbb{T}_{\mathsf{bpst}}$.

4.1.2 Validity

As for lMSOT the obvious notion of validity is not sufficient – but $\mathtt{m}$-P-validity generalizes to BPL by considering every state in every trace of $\mathtt{m}$ in any program $\mathsf{Prg}\,\mathtt{m} \in P$.

Definition 4.8. *Let P be a set of programs and φ be a* BPL *sentence. Sentence φ is*

- valid *if every state is a model for φ, and*
- P-valid *if every state in every run of every program in P is a model for φ.*
- $\mathtt{m}$-P-valid *if every state in every run of $\mathtt{m}$ within any program in P is a model for φ.*

Fact 1. *Note that the universal quantifier in the semantics of the behavioral modality automatically filters out any state from which the statement is never executed. Thus, if we are interested in the validity of a formula $\varphi \to [\mathtt{s} \Vdash \tau]$, we only need to consider those states $\binom{\sigma}{\rho}$ where $\mathsf{selected}(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}) \neq \emptyset$.*

Example 4.3. *Consider the following program* P*:*

```
interface I(){ Unit m(); }
class C() implements I{
    Int i = 0;
    Unit m(){ this.i = this.i + 1; this!n(); }
    Unit n(){ this.i = this.i + 1; }
}
main{
    I c = new C();
    c!m();
}
```

The following formula is valid $i \geq 0 \rightarrow i > 0$.

The following formula is not valid, but {P}*-valid* $[\texttt{this.i = this.i + 1;} \overset{\alpha_{\text{pst}}}{\Vdash} \texttt{this.i > 0}]$

The following formula is not {P}*-valid, but* n-{P}*-valid* $[\texttt{this.i = this.i + 1;} \overset{\alpha_{\text{pst}}}{\Vdash} \texttt{this.i > 1}]$

To realize that the two behavioral modalities are valid, it is critical to know that the field is initialized with 0. If this information is not available, the formula must have a precondition. To model lost information, we consider more programs.

Example 4.4. *Let* P *be the set of programs which differ from* P *only by the initial value of* i*. The following formula is* P*-valid:*

$$\texttt{this.i >= 0} \rightarrow [\texttt{this.i = this.i + 1;} \overset{\alpha_{\text{pst}}}{\Vdash} \texttt{this.i > 0}]$$

4.2 Proof System

We use a sequent calculus to reason about behavioral modalities.

Definition 4.9 (Sequents)**.** *Let* Γ, Δ *be finite sets of* BPL*-formulas. A* sequent $\Gamma \Rightarrow \Delta$ *has the semantics of* $\bigwedge \Gamma \rightarrow \bigvee \Delta$*.* Γ *is called the antecedent and* Δ *the succedent.*

A sequent $\{\gamma_1, \ldots, \gamma_n\} \Rightarrow \{\delta_1, \ldots, \delta_m\}$ is written $\gamma_1, \ldots, \gamma_n \Rightarrow \delta_1, \ldots, \delta_m$.

Definition 4.10 (Rules)**.** *Let* C, P_i *be sequents. A rule has the form*

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

Where C *is called the conclusion and* P_i *the premise, while cond is a side-condition. Side-conditions are always decidable.*

Additionally we use rewrite rules, rules that replace a part of a sequent syntactically by a different one.

Definition 4.11 (Rewrite Rule)**.** *Let* $\mathtt{s}, \mathtt{s}'$ *be statements,* φ, φ' *formulas,* U, U' *updates,* t, t' *terms and* τ, τ' *two types of the same behavioral specification. A rewrite rule has the form* $\mathtt{s} \rightsquigarrow \mathtt{s}'$ *(and analogous for the other categories) and replaces in a sequent every occurrence of* s *by* s′*. If two rules* $\mathtt{s} \rightsquigarrow \mathtt{s}'$ *and* $\mathtt{s}' \rightsquigarrow \mathtt{s}$ *are given, then we write them together as* $\mathtt{s} \leftrightsquigarrow \mathtt{s}'$.

Rules may contain, in addition to expressions, schematic variables. Their handling is standard [8]. When reasoning about proof rules, schematic variables can be seen as meta-variables.

We require some technical notation to simplify presentation. The first is a concatenation of updates. It is necessary, because applying two updates U_1, U_2 consecutively, e.g., $\{U_1\}\{U_2\}\varphi$, cannot be represented syntactically without the target formula (or term) being known.

Definition 4.12 (Concatenation of Updates [119]). *Let φ be a formula and $U_1, \ldots, U_n$ be n updates. We represent the formula $\{U_1\} \cdots \{U_n\}\varphi$ as a pair $(U_1 \circ \cdots \circ U_n, \varphi)$ and say that $U_1 \circ \cdots \circ U_n$ is a concatenation.*

A single update is also a concatenation. Inside of rules, we are especially interested in rules which manipulate behavioral modalities. We use symbolic triples [114] for sequents that contain a single behavioral modality.

Definition 4.13 (Symbolic Triples). *A symbolic triple* st *is a triple* $\mathsf{st} = (\Phi, U, [\mathsf{s} \Vdash^{\alpha} \tau])$*, where* Φ *is a set of* BPL*-formulas and U a concatenation. All elements may contain schematic variables. A sequent*

$$\gamma_1, \ldots, \gamma_n \Rightarrow \{U_1\} \ldots \{U_n\}[\mathsf{s} \overset{\alpha}{\Vdash} \tau], \delta_1, \ldots, \delta_m$$

is represented as

$$\left(\{\gamma_1, \ldots, \gamma_n, \neg\delta_1, \ldots, \neg\delta_m\} \ , \ U_1 \circ \cdots \circ U_n \ , \ [\mathsf{s} \overset{\alpha}{\Vdash} \tau] \right)$$

Given two symbolic triples, we say that they share the behavioral type if their behavioral modalities contain the same behavioral type. Given a symbolic triple, we refer to its behavioral type by

$$\mathsf{t}\left(\left(\Phi, U, [\mathsf{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau]\right)\right) = \mathbb{T}$$

We say that the sequent has $\mathsf{t}(\mathsf{st})$ *as its type.*

A sequent can have multiple types. We can now distinguish two important classes of rules: those who implement symbolic execution on behavioral modalities and those who implement lFOS-theories.

Definition 4.14 (Symbolic Execution). *A sequent is called symbolic if it has a defined symbolic triple. A sequent is called pure if it contains no behavioral modality. For sequents in rules, schematic variables for formulas are* not *behavioral modalities.*

A rule is a symbolic execution rule for $\mathbb{T}$ if its conclusion is symbolic and has $\mathbb{T}$ as its type, all its premises are either symbolic or pure, at least one premise has the same type, the premisses follow the decomposition of the syntax in Def. 3.1 A rule is a basic rule if its conclusion and all its premises are pure.

Example 4.5. *The following rule for postcondition reasoning is a symbolic execution rule for* $\mathbb{T}_{\mathsf{pst}}$

$$\textit{(assign-to-update)} \ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \mathtt{e}\}[\mathtt{s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e}; \ \mathtt{s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}$$

A rewrite rule over terms is

$$\mathsf{select}(\mathsf{store}(h, \mathtt{f}, \mathtt{e}), \mathtt{f}) \rightsquigarrow \mathtt{e}$$

The following rule is a basic rule

$$\textit{(right-or)} \ \frac{\Gamma \Rightarrow \varphi, \psi, \Delta}{\Gamma \Rightarrow \varphi \vee \psi, \Delta}$$

The following rule connects LTL and postconditions and is neither a symbolic execution nor a basic rule

$$\textit{(end-to-always)} \ \frac{\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha_{|\mathsf{t}|}}{\Vdash} \Box\varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}$$

We can now define behavioral types in BPL: behavioral specifications extended with a symbolic execution *calculus* and rewrite rules and an *obligation schema*. The obligation schema maps every method within a program to a pair of a formula φ and a specification τ. This pair represents the formula

$$\varphi \to \{\texttt{oldHeap} := \texttt{heap} || \texttt{lastHeap} := \texttt{heap}\}[\mathtt{s}_{\mathtt{m}} \overset{\alpha}{\Vdash} \tau]$$

where $\mathtt{s}_{\mathtt{m}}$ is the method body of the method in question. The obligation schema is needed for soundness arguments: the proof that one method adheres to its specification may assume that each other method adheres to its specification, i.e., that the other proof obligation formulas hold.

Definition 4.15 (Behavioral Type)**.** *Let $(\tau_{\mathbb{T}}, \alpha_{\mathbb{T}})$ be a behavioral specification $\mathbb{T}$. A* behavioral type $\mathbb{T}$ *extending the behavioral specification is a quadruple $(\tau_{\mathbb{T}}, \alpha_{\mathbb{T}}, \iota_{\mathbb{T}}, \pi_{\mathbb{T}})$. The* calculus $\pi_{\mathbb{T}}$ *is a set of symbolic execution rules for $\mathbb{T}$ and rewrite rules over elements of $\tau_{\mathbb{T}}$. The* obligation schema $\iota_{\mathbb{T}}$ *is a map from method names to pairs $(\varphi^{\mathtt{m}}, \tau^{\mathtt{m}})$, where $\varphi^{\mathtt{m}}$ is a lFOS formula that may only contain field and local variables accessible by the method in question.*

We say that the following sequent corresponds to $\iota_{\mathbb{T}}(\mathtt{m}) = (\varphi^{\mathtt{m}}, \tau^{\mathtt{m}})$

$$\varphi^{\mathtt{m}} \to \{\texttt{oldHeap} := \texttt{heap} || \texttt{lastHeap} := \texttt{heap}\}[\mathtt{s}_{\mathtt{m}} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau^{\mathtt{m}}]$$

where $\mathtt{s}_{\mathtt{m}}$ is the method body of the method in question.

Additionally to the rules given by behavioral type, we also use basic rules and general rewrite rules. The rules for lFOS and the rules for the heap are all basic rules. We list the standard FO rules and some of the needed rules for update and the heap in Fig. 4.2. A full overview over the needed theory of heaps, sets and lists can be found in [8].

Definition 4.16 (Soundness)**.** *A rule is* sound *if validity of all premises implies validity of the conclusion.*
A rule is $\mathtt{m}$-P-sound *if $\mathtt{m}$-P-validity of all premises implies $\mathtt{m}$-P-validity of the conclusion.*
A rewrite rule $\tau_1 \rightsquigarrow \tau_2$ is sound if $\alpha(\tau_2)$ implies $\alpha(\tau_1)$ and analogous for the other categories.

Lemma 4.4. *All the rules in Fig. 4.2 are sound [56, 115].*

Obviously, soundness implies $\mathtt{m}$-P-soundness for any choice of $\mathtt{m}$ and P. Indeed, for basic rules, only standard validity is of interest, while for symbolic execution rules we consider $\mathtt{m}$-P-soundness. We can state some basic properties about soundness and operations on P.

Lemma 4.5. *Let P, P' be two sets of programs and $\mathtt{m}$ be a method.*

1. *If a formula is $\mathtt{m}$-P-valid and $\mathtt{m}$-P'-valid, then it is $\mathtt{m}$-$P \cup P'$-valid.*
2. *If $P \subseteq P'$ and a formula is $\mathtt{m}$-P'-valid, then it is $\mathtt{m}$-P-valid.*
3. *If $P \subseteq P'$ and a rule is $\mathtt{m}$-P'-sound, then it is $\mathtt{m}$-P-sound.*

Proof. See p. 169.

Before we introduce soundness of behavioral types, and to illustrate the above principles, we now give the first behavioral type: $\mathbb{T}_{\mathsf{sinv}}$, the simple[2] type for object invariants. The object invariant has to hold whenever no process is active. From the view of the object it holds whenever it assumes control over the object and has to be established whenever control is released.

Definition 4.17 (Behavioral Specification $\mathbb{T}_{\mathsf{sinv}}$)**.** *Let C be a class and φ a lFOS sentence containing only fields of C. The behavioral specification $\mathbb{T}_{\mathsf{sinv}} = (\tau_{\mathsf{sinv}}, \alpha_{\mathsf{sinv}})$ for C has as its syntax the set of all lFOS formulas which contain only fields of C. The semantics is defined as*

$$\alpha_{\mathsf{sinv}}(\varphi) = \forall i \in \mathbf{I}.\ \big(\mathsf{isEvent}(i) \wedge \neg(\mathsf{isFutREv}(i) \vee \mathsf{isNoEv}(i) \vee \mathsf{isInvEv}(i))\big) \to [i+1] \vdash \varphi$$

[2] In contrast to the led type for object invariants in the next sections.

$$\textbf{(left-and)}\ \frac{\Gamma, \varphi, \psi \Rightarrow \Delta}{\Gamma, \varphi \wedge \psi \Rightarrow \Delta} \qquad \textbf{(right-and)}\ \frac{\Gamma \Rightarrow \varphi, \Delta \qquad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \wedge \psi, \Delta}$$

$$\textbf{(left-or)}\ \frac{\Gamma, \varphi \Rightarrow \Delta \qquad \Gamma, \psi \Rightarrow \Delta}{\Gamma \Rightarrow \varphi \vee \psi, \Delta} \qquad \textbf{(right-or)}\ \frac{\Gamma \Rightarrow \varphi, \psi, \Delta}{\Gamma \Rightarrow \varphi \vee \psi, \Delta}$$

$$\textbf{(left-neg)}\ \frac{\Gamma, \neg\varphi \Rightarrow \Delta}{\Gamma \Rightarrow \varphi, \Delta} \qquad \textbf{(right-neg)}\ \frac{\Gamma, \varphi \Rightarrow \Delta}{\Gamma \Rightarrow \neg\varphi, \Delta}$$

$$\textbf{(left-close)}\ \frac{}{\Gamma, \mathsf{false} \Rightarrow \Delta} \qquad \textbf{(right-close)}\ \frac{}{\Gamma \Rightarrow \mathsf{true}, \Delta}$$

$$\textbf{(left-forall)}\ \frac{\Gamma, \forall v \in S.\ \varphi, \varphi[v \setminus t] \Rightarrow \Delta}{\Gamma, \forall v \in S.\ \varphi \Rightarrow \Delta}\ t \text{ is a term of } S \text{ sort}$$

$$\textbf{(left-exists)}\ \frac{\Gamma.\varphi[v \setminus c] \Rightarrow \Delta}{\Gamma, \exists v \in S.\ \varphi \Rightarrow \Delta}\ c \text{ is a fresh constant of } S \text{ sort}$$

$$\textbf{(right-forall)}\ \frac{\Gamma \Rightarrow \varphi[v \setminus c], \Delta}{\Gamma \Rightarrow \forall v \in S.\ \varphi, \Delta}\ c \text{ is a fresh constant of } S \text{ sort}$$

$$\textbf{(right-exists)}\ \frac{\Gamma \Rightarrow \exists v \in S.\ \varphi, \varphi[v \setminus t], \Delta}{\Gamma \Rightarrow \exists v \in S.\ \varphi, \Delta}\ t \text{ is a term of } S \text{ sort}$$

$$\textbf{(close)}\ \frac{}{\Gamma, \varphi \Rightarrow \varphi, \Delta}$$

$$\mathsf{select}(\mathsf{store}(h, f, x), g) \rightsquigarrow \mathsf{if}\ g \doteq f\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{select}(h, g)$$

$$\{\mathrm{v} := \mathrm{t}\}\mathrm{v} \rightsquigarrow \mathrm{t} \qquad \{\mathrm{v} := \mathrm{t}\}\mathrm{v}' \rightsquigarrow \mathrm{v}' \text{ if } \mathrm{v} \neq \mathrm{v}'$$

$$\{U\}p(\mathrm{t}_1, \ldots, \mathrm{t}_n) \rightsquigarrow p(\{U\}\mathrm{t}_1, \ldots, \{U\}\mathrm{t}_n) \qquad \{U\}f(\mathrm{t}_1, \ldots, \mathrm{t}_n) \rightsquigarrow f(\{U\}\mathrm{t}_1, \ldots, \{U\}\mathrm{t}_n)$$

Figure 4.2: Selected rules for first-order reasoning. The rules can be applied under updates.

The semantics expresses the above intuition in terms of events: After any visible event that releases or assumes control over the object, i.e., after any suspension or termination, as well as after any reactivation or method start the invariant holds.

Definition 4.18 (Behavioral Type $\mathbb{T}_{\mathsf{sinv}}$)**.** *The behavioral type $\mathbb{T}_{\mathsf{sinv}}$ for a formula φ extends the behavioral specification with $(\iota^{\varphi}_{\mathsf{sinv}}, \pi_{\mathsf{sinv}})$. The calculus is given in Fig. 4.3, the obligation schema is defined by $\iota^{\varphi}_{\mathsf{sinv}}(\mathtt{m}) = (\varphi, \varphi)$ for all method names.*

There is one type per invariant, but all types share their calculus. The proof rules are however, only sound if all method adhere to the same invariant — the obligation schema mirrors this restriction.

Rule ($\mathbb{T}_{\mathsf{sinv}}$-assignV) translates a variable assignment to an update. This is possible because the evaluation of the expression has no side-effects and expressions can be translated into terms. Rule ($\mathbb{T}_{\mathsf{sinv}}$-readV) reads from a future. To do so, a fresh function symbol v is assigned to the target variable in an update. There is no knowledge about the value. Rule ($\mathbb{T}_{\mathsf{sinv}}$-assignF) translates a field assignment to a field. This requires a separate rule, as this modifies the heap variable. Rule ($\mathbb{T}_{\mathsf{sinv}}$-awaitF) has two premises. First, the invariant has to be proven in the current state, i.e., before the suspension. The second premise anonymizes the heap: the only information about the heap $\mathsf{anon}(\mathsf{heap})$ that is available is that the invariant holds. Rule ($\mathbb{T}_{\mathsf{sinv}}$-awaitB) is analogous, except that additionally it is known that the guard expression holds. Rule ($\mathbb{T}_{\mathsf{sinv}}$-callV) is analogous to the rule for future reads: the fresh future is modeled by a fresh function symbol assigned to the target variable. Rule ($\mathbb{T}_{\mathsf{sinv}}$-if) proves that both branches preserve the invariant. Rule ($\mathbb{T}_{\mathsf{sinv}}$-while) proves that the loop body and the continuation both preserve the invariant. All information, except the guard, has to be removed, as the first premise abstract *all* states that may be at the start of an iteration and the second the first state afterwards. This can be seen as a special case of a loop invariant, using the loop invariant **true**. The other rules are straightforward.

Soundness is not enough for our purpose. Consider the rules (awaitF) and (awaitB): The validity of their premises only ensures that all traces of s that start with a state where the object invariant holds establish the semantics of their specification. But to establish that all traces of **await** e; s establish the semantics of their specification one must additionally show that this indeed covers all states where a reactivation (to execute s) is possible. m-P-soundness allows us to cover the case where other methods run during the suspension (as we may assume that they preserve the invariant). But this does *not* cover the case where multiple process of the method in question interleave.

We introduce the notion of an enabling rule: a rule is enabling if when it is used to close a proof of $\iota(\mathtt{m})$, then this implies validity of the sequent corresponding to $\iota(\mathtt{m})$. In practice this means that this rule may *only* be used for symbolic execution starting with a full method body.

Definition 4.19 (Enabling Rules)**.** *A rule is* $\mathbb{T}$ (-m-P-)enabling *if a closed proof for the sequent corresponding to $\iota_{\mathbb{T}}(\mathtt{m})$ that contains only $\mathbb{T}$-enabling rules implies (m-P-)validity of the sequent corresponding to $\iota_{\mathbb{T}}(\mathtt{m})$.*

Fact 2. *A sound rule is $\mathbb{T}$-enabling for every $\mathbb{T}$.*

Definition 4.20. *Let $\mathbb{T} = (\tau, \alpha, \iota, \pi)$ be a behavioral type. For the verification of one method, we assume that the others are verified and define the following set:*

$$P_{\mathtt{m}} = \left\{ \mathsf{Prgm} \mid \textit{for all methods } \mathtt{m}' \textit{ with } \mathtt{m} \neq \mathtt{m}' \textit{ the formula } \iota(\mathtt{m}') \textit{ is } \mathtt{m}'\textit{-}\{\mathsf{Prgm}\}\textit{-valid} \right\}$$

If for every $\mathtt{m}$ the type system π is m-$P_{\mathtt{m}}$-enabling, then we say that $\mathbb{T}$ is sound.

Intuitively, the set $P_{\mathtt{m}}$ is the set of programs where the specification of all methods besides $\mathtt{m}$ holds. It does not mean that the actual implementation of the methods adheres to the specification. However, we can prove m-$P_{\mathtt{m}}$ validity of $\mathtt{m}$ under the assumption that all other method are correct. This is *not* the composition step of results. To derive a global property an additional step in required. E.g., to prove object invariants, one can prove that one method preserves the invariant, if all other methods preserve it. The global step that the object invariant holds in all state of the object requires the additional argument

$$(\mathbb{T}_{\text{sinv}}\text{-assignV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \mathtt{e}\}[\mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e};\ \mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{sinv}}\text{-assignF})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{heap} := \mathsf{store}(\mathtt{heap}, \mathtt{f}, \mathtt{e})\}[\mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{this}.\mathtt{f} = \mathtt{e};\ \mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{sinv}}\text{-readV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \nu\}[\mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e}.\mathbf{get};\ \mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}\ \nu \text{ fresh}$$

$$(\mathbb{T}_{\text{sinv}}\text{-awaitF})\ \frac{\Gamma \Rightarrow \{U\}I, \Delta \qquad \Gamma, \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}I \Rightarrow \{U\}\{\mathbf{last} := \mathtt{heap} \| \mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}[\mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \mathtt{e?};\ \mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{sinv}}\text{-awaitB})\ \frac{\Gamma \Rightarrow \{U\}I, \Delta \qquad \Gamma, \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}(I \wedge \mathtt{e}) \Rightarrow \{U\}\{\mathbf{last} := \mathtt{heap} \| \mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}[\mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \mathtt{e};\ \mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{sinv}}\text{-callV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \nu\}[\mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{f!m}(\mathtt{e}_1, \ldots, \mathtt{e}_n);\ \mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}\ \nu \text{ fresh}$$

$$(\mathbb{T}_{\text{sinv}}\text{-if})\ \frac{\Gamma, \{U\}\mathtt{e} \Rightarrow \{U\}[\mathtt{s}_1; \mathtt{s}_3 \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta \qquad \Gamma, \{U\}\neg\mathtt{e} \Rightarrow \{U\}[\mathtt{s}_2; \mathtt{s}_3 \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{if}(\mathtt{e})\{\mathtt{s}_1\}\mathbf{else}\{\mathtt{s}_2\}; \mathtt{s}_3 \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{sinv}}\text{-return})\ \frac{\Gamma \Rightarrow \{U\}I, \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{return}\ \mathtt{e} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{sinv}}\text{-skipCont})\ \frac{\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{skip};\ \mathtt{s} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{sinv}}\text{-skip})\ \frac{}{\Gamma \Rightarrow \{U\}[\mathbf{skip} \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{sinv}}\text{-while})\ \frac{\Gamma, \{U\}\{U_{\mathscr{A}}\}\neg\mathtt{e} \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s}_2 \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta \qquad \Gamma, \{U\}\{U_{\mathscr{A}}\}\mathtt{e} \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s}_1 \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{while}(\mathtt{e})\{\mathtt{s}_1\}; \mathtt{s}_2 \overset{\alpha_{\text{sinv}}}{\Vdash} I], \Delta}$$

Figure 4.3: Rules for $\mathbb{T}_{\text{sinv}}$. Rules for variable assignments with declaration are analogous. $U_{\mathscr{A}}$ assigns heap the term anon(heap), and every other program variable except **old** a fresh function symbol.

that the object invariant is established (by the initial values) before the first run of any method and an induction over the following traces using the soundness of the type.

Such composition arguments may rest on additional analyses or requirements. E.g., the above requires the establishing of the object invariants by the initial value *which is not part of the type*.

Lemma 4.6. $\mathbb{T}_{\mathsf{sinv}}$ *is sound (in the sense of Def. 4.20).*

Proof. See p. 169.

The above lemma only expresses that, given a method in a class, if all other methods in a class preserve a loop invariant, then so does this one, given that its proof obligation can be proven. Yet, there remains a final step to express that an object always adheres to its invariant. In particular, one has to establish that the initial values for the fields establish the object invariant. The property that every object always preserves its object invariant is global and formalized in gMSOT.

Lemma 4.7. *Let* Prgm *be a program with objects* $\mathtt{x}_1, \ldots, \mathtt{x}_n$ *of class* $\mathtt{c}_1, \ldots, \mathtt{c}_n$*. Let* φ_{C} *be the class invariant of class a* C*. If (1) for every class* $\mathtt{c}_i$*, every proof obligation* $\iota^{\varphi}_{\mathsf{sinv}}(\mathtt{m})$ *for every method* m *within* $\mathtt{c}_i$ *can be proven and (2) the initial values of the fields establish* φ_{C}*. Then for every* $i \leq n$ *the invariant* φ_{C_i} *holds at every point* $\mathtt{x}_i$ *has no active process. I.e., every run of* Prgm *is a model for the following formula*

$$\bigwedge_{\mathsf{C}\ in\ \mathsf{Prgm}} \forall \mathtt{x} \in \mathsf{C}.\ \forall i \in \mathbf{I}.\ [i] \doteq \mathsf{release}(i, \mathtt{x}) \rightarrow [i+1] \vdash \varphi_{\mathsf{C}}[\mathbf{this} \setminus \mathtt{x}]$$

Proof. See p. 171.

Where $\mathsf{release}(i, \mathtt{x})$ holds iff the ith event is a suspension, reactivation, termination or invocation reaction event of x. The formal details behind initialization are given in the appendix.

4.3 Compositional Proof System

The object invariant system we give above did not use a (non-trivial) loop invariant. The reason is that $\mathbb{T}_{\mathsf{sinv}}$ cannot detect the end of a loop iteration and thus cannot detect the end of a loop iteration and thus cannot deal with non-trivial loop invariants. In contrast, a calculus for $\mathbb{T}_{\mathsf{pst}}$ can contain a standard loop invariant rule:

$$\textbf{(pst-inv)}\ \frac{\Gamma \Rightarrow \{U\}I, \Delta \qquad \Gamma, \{U_{\mathscr{A}}\}(I \wedge \mathtt{e}) \Rightarrow \{U_{\mathscr{A}}\}[\mathtt{s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} I], \Delta \qquad \Gamma, \{U_{\mathscr{A}}\}(I \wedge \neg \mathtt{e}) \Rightarrow \{U_{\mathscr{A}}\}[\mathtt{s'} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{while}(\mathtt{e})\{\mathtt{s}\};\mathtt{s'} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}$$

How to combine these systems? One solution would be to integrate postcondition reasoning into the object invariant system, with the two additional premises from the $\mathbb{T}_{\mathsf{pst}}$ type.

$$\textbf{(invpst-while)}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}I, \Delta \qquad \{U_{\mathscr{A}}\}(I \wedge \mathtt{e}) \Rightarrow \{U_{\mathscr{A}}\}[\mathtt{s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} I] \\ \{U_{\mathscr{A}}\}(I \wedge \mathtt{e}) \Rightarrow \{U_{\mathscr{A}}\}[\mathtt{s} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} \varphi] \qquad \{U_{\mathscr{A}}\}(I \wedge \neg \mathtt{e}) \Rightarrow \{U_{\mathscr{A}}\}[\mathtt{s'} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} \varphi]\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{while}(\mathtt{e})\{\mathtt{s}\};\mathtt{s'} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} \varphi], \Delta}$$

This has two downsides: (1) the loop body is symbolically executed twice and (2) this requires to have both calculi in the overall reasoning system: all statements need to have rules in each system *even if they model the same state changes*. E.g., besides the semantics function, the rule for synchronization for postcondition reasoning is identical to the one for object invariants.

$$\textbf{(pst-readV)}\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := v\}[\mathtt{s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e}.\mathbf{get};\ \mathtt{s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}\ v \text{ fresh}$$

This approach impedes the integration of new behavioral types. The rule **(invpst-while)** not only adds a premise to use postcondition reasoning, it also adds additional information to the premises from the object invariant rule. Every time a new type is added, the overall rule has to be manually reevaluated, its soundness has to be reproven and its complexity keeps growing. While the complexity is not necessarily a problem for soundness proofs (which can be mechanized), it raises the cognitive burden for human interaction and further extensions even further. Even in presence of mechanization, complex rules can be hard to handle due to subtle bugs: the loop invariant rule of JavaDL required soundness-critical bug fixes even after almost two decades of use [93] before being replaced by a variant [121] which reduced its (cognitive) complexity. We aim to keep calculi simple not by translating the program to minimal constructs, but by combining rules from multiple behavioral types to more complex rules. This allows an aspect-oriented design[3] of a calculus, where every aspect of verification is modeled with another behavioral type.

What is needed for the above situation is to express the modality that verifies a statement against *two* behavioral types, informally written as $[\mathsf{s}\,|\!\vdash^{\alpha_{\mathsf{pst}}\wedge\alpha_{\mathsf{inv}}}\varphi\wedge I]$. The rule for loop invariant would be

$$\textbf{(invpst-while2)}\ \frac{\Gamma,\{U_{\mathscr{A}}\}(I'\wedge\neg\mathsf{e})\Rightarrow\{U_{\mathscr{A}}\}[\mathsf{s}'\,|\!\vdash^{\alpha_{\mathsf{pst}}\wedge\alpha_{\mathsf{sinv}}}\varphi\wedge I],\Delta \qquad \Gamma\Rightarrow\{U\}I',\Delta \qquad \Gamma,\{U_{\mathscr{A}}\}(I'\wedge\mathsf{e})\Rightarrow\{U_{\mathscr{A}}\}[\mathsf{s}\,|\!\vdash^{\alpha_{\mathsf{pst}}\wedge\alpha_{\mathsf{sinv}}}\varphi\wedge I'],\Delta}{\Gamma\Rightarrow\{U\}[\textbf{while}(\mathsf{e})\{\mathsf{s}\};\mathsf{s}'\,|\!\vdash^{\alpha_{\mathsf{pst}}\wedge\alpha_{\mathsf{sinv}}}\varphi\wedge I],\Delta}$$

This rule allows us to use only *one* rule for synchronization and reduces the required human reasoning about the behavioral type: it suffices to reason about the soundness of the simple behavioral types and their composition, which is more simple to handle than reasoning about the composed system, especially if we add further behavioral types. In the rest of this section we describe how to combine two behavioral types. While the combination of syntax, semantics and obligation schema is straightforward, the main challenge is to combine two sound validity calculi into a new sound validity calculus by syntactic manipulation of the sequent calculus rules.

To do so, we use *leading composition*. Leading composition is not symmetric: one behavioral type is leading another one. The leading type provides the context which the led type may use. In the above sketch the postcondition type is leading the object invariant type by providing the information about the context by the invariant. An asymmetrical composition simplifies the design of led types.

4.3.1 Leading Composition

We define a behavioral specification that combines two behavioral specifications $\mathbb{T}_1, \mathbb{T}_2$ to express that both semantics hold.

Definition 4.21 (Composition of Behavioral Specifications)**.** *Let* $\mathbb{T}_1 = (\tau_{\mathbb{T}_1}, \alpha_{\mathbb{T}_1})$ *and* $\mathbb{T}_2 = (\tau_{\mathbb{T}_2}, \alpha_{\mathbb{T}_2})$ *be two behavioral specifications. The composed behavioral specification* $\mathbb{T}_1 \between \mathbb{T}_2 = (\tau_{\mathbb{T}_1} \between \tau_{\mathbb{T}_2}, \alpha_1 \between \alpha_2)$ *is defined as:*

$$\tau_{\mathbb{T}_1} \between \tau_{\mathbb{T}_2} = \{\tau_1 \between \tau_2 \mid \tau_1 \in \tau_{\mathbb{T}_1} \text{ and } \tau_2 \in \tau_{\mathbb{T}_2}\}$$
$$\alpha_{\mathbb{T}_1} \between \alpha_{\mathbb{T}_2}(\tau_1 \between \tau_2) = \alpha_{\mathbb{T}_1}(\tau_1) \wedge \alpha_{\mathbb{T}_2}(\tau_2)$$

For behavioral *specifications*, composition is symmetrical: for the syntax, $\between$ is a purely syntactic operator, for the semantics, a trace has to be a model for both composed types.

Lemma 4.8. *The following sequents are P-valid*

$$\Gamma\Rightarrow\{U\}[\mathsf{s}\overset{\alpha}{|\!\vdash}\tau],\Delta \qquad \Gamma\Rightarrow\{U\}[\mathsf{s}\overset{\alpha'}{|\!\vdash}\tau'],\Delta$$

iff the sequent $\Gamma\Rightarrow\{U\}[\mathsf{s}\,|\!\vdash^{\alpha\between\alpha'}\tau\between\tau'],\Delta$ *is P-valid.*

[3] The mechanism we give is less flexible than the ones used for aspect-oriented *programming* [94] and is not based on them. We do, however, acknowledge the conceptual similarities of integrating cross-cutting concerns.

$$\textbf{(split-⅋)}\ \frac{\Gamma \Rightarrow \{U\}[\mathsf{s} \Vdash^{\alpha} \tau], \Delta}{\Gamma \Rightarrow \{U\}[\mathsf{s} \Vdash^{\alpha \parr \alpha'} \tau \parr \tau'], \Delta} \qquad \begin{array}{c} [\mathsf{s} \overset{\alpha_1 \parr \alpha_2}{\Vdash} \tau_1 \parr \tau_2] \leftrightsquigarrow [\mathsf{s} \overset{\alpha_2 \parr \alpha_1}{\Vdash} \tau_2 \parr \tau_1] \\ [\mathsf{s} \overset{\alpha_1 \parr \alpha_1}{\Vdash} \tau_1 \parr \tau_1] \leftrightsquigarrow [\mathsf{s} \overset{\alpha_1}{\Vdash} \tau_1] \\ [\mathsf{s} \overset{\alpha_1 \parr (\alpha_2 \parr \alpha_3)}{\Vdash} \tau_1 \parr (\tau_2 \parr \tau_3)] \leftrightsquigarrow [\mathsf{s} \overset{(\alpha_1 \parr \alpha_2) \parr \alpha_3}{\Vdash} (\tau_1 \parr \tau_2) \parr \tau_3] \end{array}$$

Figure 4.4: General rules for leading compositions.

Proof. See p. 172.

From the above lemma and the definition, we immediately see that the rules in Fig. 4.4 are sound.

However, we are interested in composition, not decomposition. Composition simplifies the design of behavioral types, because it is not required to describe state changes in all calculi. Instead one can omit the context completely and design a sound, but imprecise rule that is supposed to get its precision from a leading type.

Definition 4.22. *Let (r1),(r2) be two rules of the form*

$$\textbf{(r1)}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\varphi_1, \Delta \quad \dots \quad \Gamma \Rightarrow \{U\}\varphi_n, \Delta \\ \Gamma, \{U_1\}\psi_1 \Rightarrow \{U_1\}[\mathsf{s}_1 \Vdash^{\alpha} \tau_1], \Delta \quad \dots \quad \Gamma, \{U_m\}\psi_m \Rightarrow \{U_m\}[\mathsf{s}_m \Vdash^{\alpha} \tau_m], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathsf{s} \Vdash^{\alpha} \tau], \Delta}\ cond_1$$

$$\textbf{(r2)}\ \frac{\begin{array}{c}\Rightarrow \psi_1 \quad \dots \quad \Rightarrow \psi_{n'} \\ \Rightarrow [\mathsf{s}_1 \Vdash^{\alpha'} \tau'_1] \quad \dots \quad \Rightarrow [\mathsf{s}_{m'} \Vdash^{\alpha'} \tau'_{m'}]\end{array}}{\Rightarrow [\mathsf{s} \Vdash^{\alpha'} \tau']}\ cond_2$$

Its composition (r1 ⅋ r2) is defined as follows.

- *The conclusion is* $\Gamma \Rightarrow \{U\}[\mathsf{s} \Vdash^{\alpha \parr \alpha'} \tau \parr \tau'], \Delta$
- *The side-condition is* $cond_1 \wedge cond_2$.
- *The premises are*
  1. *all basic sequents that are a premise from (r1) and*
  2. *for each premise of (r2), the sequent represented by* $\Gamma \Rightarrow \{U\}\psi_i, \Delta$
  3. *for any premise of (r1) represented by*

$$\left(\Phi, U_i, [\mathsf{s}_i \overset{\alpha}{\Vdash} \tau_i]\right)$$

and any premise of (r2) represented by

$$\left(\emptyset, \epsilon, [\mathsf{s}_j \overset{\alpha'}{\Vdash} \tau'_j]\right)$$

with $\mathsf{s}_i = \mathsf{s}_j$ *the sequent represented by*

$$\left(\Phi, U_i, [\mathsf{s} \overset{\alpha \parr \alpha'}{\Vdash} \tau_i \parr \tau'_j]\right)$$

4. *If some premise of (r1) or (r2) is not subsumed by the above construction, then it is also a premise of (r1 ◊ r2).*

We stress that the formulas $\{U_i\}\psi_i$ are added to the merged rule – they must now hold in less states than described in (r2). The following example shows how this can be used to merge multiple rules to increase precision if more information is known and how it simplifies the design of new types.

Example 4.6. *Consider the following rules for postcondition reasoning and object invariants.*

$$\textit{(pst-assign)}\ \frac{\Gamma \Rightarrow \{U\}\{\texttt{heap} := \mathsf{select}(\texttt{heap},\texttt{f},\texttt{e})\}[\texttt{s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\texttt{v = e;s} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta} \qquad \textit{(pst-return)}\ \frac{\Gamma \Rightarrow \{U\}\{\texttt{result} := \texttt{e}\}\varphi, \Delta}{\Gamma \Rightarrow \{U\}[\textbf{return}\ \texttt{e} \overset{\alpha_{\mathsf{pst}}}{\Vdash} \varphi], \Delta}$$

$$\textit{(inv-assign)}\ \frac{\Rightarrow [\texttt{s} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I]}{\Rightarrow [\texttt{v := e;s} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I]} \qquad \textit{(inv-return)}\ \frac{\Rightarrow I}{\Rightarrow [\textbf{return}\ \texttt{e} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I]}$$

The rules are all sound, albeit useless: the $\mathbb{T}_{\mathsf{sinv}}$ rules cannot match on sequents with non-empty antecedents. They are not supposed to be used for proofs, they are modeling the aspect of object invariants. Merging the rules according to Def. 4.22 results in the following rules.

$$\textit{(pstinv-assign)}\ \frac{\Gamma \Rightarrow \{U\}\{\texttt{heap} := \mathsf{select}(\texttt{heap},\texttt{f},\texttt{e})\}[\texttt{s} \overset{\alpha_{\mathsf{pst}} \lozenge \alpha_{\mathsf{sinv}}}{\Vdash} \varphi \lozenge I], \Delta}{\Gamma \Rightarrow \{U\}[\texttt{v = e;s} \overset{\alpha_{\mathsf{pst}} \lozenge \alpha_{\mathsf{sinv}}}{\Vdash} \varphi \lozenge I], \Delta}$$

$$\textit{(pstinv-return)}\ \frac{\Gamma \Rightarrow \{U\}\{\texttt{result} := \texttt{e}\}\varphi, \Delta \qquad \Gamma \Rightarrow \{U\}I, \Delta}{\Gamma \Rightarrow \{U\}[\textbf{return}\ \texttt{e} \overset{\alpha_{\mathsf{pst}} \lozenge \alpha_{\mathsf{sinv}}}{\Vdash} \varphi \lozenge I], \Delta}$$

This allows us now to show that the following method preserves the invariant `this.i > 0` *and has the postcondition* `old.i ≤ result − 1` *as shown in Fig.4.5a.*

```
1 Int m(){
2     this.i = this.i+this.i;
3     return this.i-1;
4 }
```

The benefit is that the calculus for loop invariants neither needs to take care of the return value, nor of the specifics of the heap. We want to similarly merge rules where both input rules are leading but before we do so, we observe that Def. 4.22 is not necessarily soundness-preserving: E.g., the original update and logical context of the leading rule are added to the basic premises of the led rule. This is only sound if the basic premise is modeling some obligation to be proven at the point where the statement in question is executed. We can formalize such requirements.

Definition 4.23. *Let (r) be a rule of the following form*

$$\textit{(r)}\ \frac{\Rightarrow \psi_1 \quad \dots \quad \Rightarrow \psi_n}{\Rightarrow [\mathtt{s}_1 \Vdash^{\alpha} \tau_1] \quad \dots \quad \Rightarrow [\mathtt{s}_m \Vdash^{\alpha} \tau_m]}\ \mathit{cond}$$

$$\Rightarrow [\mathtt{s} \Vdash^{\alpha} \tau]$$

We say that (r) is basic-local *if every statement $\mathtt{s}_i$ is a substatement of* s *and if for every state $\binom{\sigma}{\rho}$, every variable assignment β, every trace $\theta \in [\![\mathtt{s}]\!]_{\binom{\sigma}{\rho},I,\beta}$ the condition $\theta \models \alpha(\tau)$ holds whenever*

- $\binom{\sigma}{\rho} \models \bigwedge_{i \le n} \psi_i$

$$\dfrac{\dfrac{}{\texttt{this.i} > 0 \Rightarrow \texttt{this.i+this.i} > 0}}{\dfrac{\texttt{this.i} > 0 \Rightarrow \{\texttt{old} := \texttt{heap}\}\{\texttt{heap} := \mathsf{store}(\texttt{heap}, \texttt{i}, \texttt{this.i+this.i})\}\texttt{this.i} > 0}{*}}$$

$$\dfrac{\dfrac{\dfrac{}{\texttt{this.i} > 0 \Rightarrow 1 \leq \texttt{this.i}}}{\texttt{this.i} > 0 \Rightarrow \texttt{this.i} \leq ((\texttt{this.i+this.i}) - 1)}}{\dfrac{\texttt{this.i} > 0 \Rightarrow \{\texttt{old} := \texttt{heap}\}\{\texttt{heap} := \mathsf{store}(\texttt{heap}, \texttt{i}, \texttt{this.i+this.i})\}\{\texttt{result} := \texttt{this.i} - 1\}\texttt{old.i} \leq \texttt{result} - 1}{**}}$$

$$\dfrac{\dfrac{}{\texttt{this.i} > 0 \Rightarrow \{\texttt{old} := \texttt{heap}\}\{\texttt{heap} := \mathsf{store}(\texttt{heap}, \texttt{i}, \texttt{this.i+this.i})\}[\texttt{return this.i-1} \overset{\alpha_{\mathsf{pst}} \between \alpha_{\mathsf{sinv}}}{\Vdash} \texttt{old.i} \leq \texttt{result} - 1 \between \texttt{this.i} > 0]}{\texttt{this.i} > 0 \Rightarrow \{\texttt{old} := \texttt{heap}\}[\texttt{this.i = this.i+this.i; return this.i-1} \overset{\alpha_{\mathsf{pst}} \between \alpha_{\mathsf{sinv}}}{\Vdash} \texttt{old.i} \leq \texttt{result} - 1 \between \texttt{this.i} > 0]}} \quad * \quad **$$

(a) Proofs with a merged calculus.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{this.i>0} \rightarrow \{\ldots\}\mathsf{true}} \quad \dfrac{\dfrac{}{\texttt{this.i>0} \rightarrow \texttt{this.i+1>0}}}{\texttt{this.i>0} \rightarrow \{\texttt{old} := \texttt{heap}||\texttt{heap} := \mathsf{store}(\texttt{heap}, \texttt{i}, \texttt{this.i+1})\}\texttt{this.i>0}}}{\texttt{this.i>0} \rightarrow \{\texttt{old} := \texttt{heap}||\texttt{heap} := \mathsf{store}(\texttt{heap}, \texttt{i}, \texttt{this.i+1})\}[\texttt{skip}; \Vdash^{\alpha_{\mathsf{pst}} \between \alpha_{\mathsf{inv}}} \mathsf{true} \between \texttt{this.i>0}]}}{\texttt{this.i>0} \rightarrow \{\texttt{old} := \texttt{heap}\}[\texttt{this.i = this.i + 1; skip}; \Vdash^{\alpha_{\mathsf{pst}} \between \alpha_{\mathsf{inv}}} \mathsf{true} \between \texttt{this.i>0}]}}{\texttt{this.i>0} \rightarrow \{\texttt{old} := \texttt{heap}\}[\texttt{this.i = this.i + 1; skip}; \Vdash^{\alpha_{\mathsf{inv}}} \texttt{this.i>0}]}$$

(b) Rewriting into a composed type to gain access to rules that keep track of the context.

Figure 4.5: Example proofs using leading composition.

- *the side-condition holds and* $\mathtt{s}$ *matches* τ
- *there is an* $i \leq m$ *and an* $\binom{\sigma'}{\rho'}$ *such that for each* $\theta' \in [\![\mathtt{s}_i]\!]_{\binom{\sigma'}{\rho'},I,\beta}$ *that is a suffix of* θ, $\theta' \models \alpha(\tau_i)$ *holds.*

Basic-locality expresses that the rule is a schema to reduce reasoning about the question whether traces of $\mathtt{s}$ are models for τ to (1) some conditions of the first state (2) some syntactic side-conditions and (3) substatements of $\mathtt{s}$. It is defined over a *rule*, but it mirrors a composition property of the behavioral type semantics itself. These rules are sound, because in the led type they are must be shown for *all* states. To do so, we must ensure that the leading type indeed symbolically executes the statement.

Definition 4.24. *Let* (r) *be a symbolic execution rule of the following form with* $U_i = U \circ U_i'$ *for each* $i \leq m$.

$$(r1)\ \dfrac{\Gamma \Rightarrow \{U\}\varphi_1, \Delta \quad \ldots \quad \Gamma \Rightarrow \{U\}\varphi_n, \Delta \qquad \Gamma, \{U_1\}\psi_1 \Rightarrow \{U_1\}[\mathtt{s}_1 \Vdash^{\alpha} \tau_1], \Delta \quad \ldots \quad \Gamma, \{U_m\}\psi_m \Rightarrow \{U_m\}[\mathtt{s}_m \Vdash^{\alpha} \tau_m], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{s} \Vdash^{\alpha} \tau], \Delta}$$

We say that (r) *is* leading *if for every state* $\binom{\sigma}{\rho}$ *with* $\binom{\sigma}{\rho} \models \bigwedge_{i \leq n} \varphi_i$, *every variable assignment* β *and every trace* $\theta \in [\![\mathtt{s}]\!]_{\binom{\sigma}{\rho},I,\beta}$ *there is a* $\binom{\sigma'}{\rho'}$ *such that there is a* $i \leq m$ *and a trace* $\theta' \in [\![\mathtt{s}_i]\!]_{\binom{\sigma'}{\rho'},I,\beta}$ *that is (1) a suffix of* θ *and (2)* $\theta' \models \alpha(\tau_i)$, *and (3)* $\binom{\sigma'}{\rho'} = [\![U_i']\!]_{\binom{\sigma}{\rho},I,\beta}(\binom{\sigma}{\rho})$.

Leading rules describe that every trace of $\mathtt{s}$ has a suffix that is a trace of one of the statements in its premises and that the update updates the state correctly.

Theorem 2. *Let* $(r1)$ *and* $(r2)$ *be two rules according to Def. 4.22. Furthermore, let* $(r1)$ *be P-sound and leading and* $(r2)$ *be* P'*-sound and basic-local.* $(r1 \between r2)$ *is* $P \cap P'$*-sound.*

Proof. See p. 172.

Lemma 4.9. *Let (r1) and (r2) be two rules according to Def. 4.22. Furthermore, let (r1) be P-sound and leading, (r2) be P'-sound and basic-local. (r1 ◊ r2) is leading.*

Proof. See p. 173.

This lemma enables us to concatenate leading: if (r1 ◊ r2) is leading, then it can be used to lead in the composition (r1 ◊ r2 ◊ r3). As mentioned above, the led calculus itself may not be used in isolation. This is a problem, if the leading type is supposed to provide the context, but introduces new proof goals. In this work, we mostly use $\mathbb{T}_{\mathsf{pst}}$ as the (left-most) leading type, which allows us to introduce trivial proof obligations.

Lemma 4.10. *The following rewrite rule is sound:* $[\mathsf{s} \Vdash^{\alpha} \tau] \leftrightsquigarrow [\mathsf{s} \Vdash^{\alpha_{\mathsf{pst}} \lozenge \alpha} \mathsf{true} \lozenge \tau]$.

Proof. See p. 174.

If one needs to prove an invariant I (Using the variation of $\mathbb{T}_{\mathsf{sinv}}$ type sketched above), this can be used to rewrite the original proof obligation into one of the lead type to use the rules which keep track of the context. E.g., as in Fig. 4.5b. Again, it is not necessary to encode all aspects in one calculus — the $\mathbb{T}_{\mathsf{sinv}}$ calculus only encodes the aspect of object invariants, not keeping track of the heap.

Def. 4.22 cannot be applied to all rules. In particular, it misses two classes of rules: rules for loops and rules where both input rules add context to the premises. We first address loops.

In this work, iteration is handled by loop, not by recursive synchronous method calls. For loops, the standard approach to specification and verification are loop invariants. Originally [52], a loop invariant for a state-based verification is a formula that holds at the beginning and end of each loop iteration. During verification, it is proven that (1) the loop invariant holds before entering the loop and that (2) the loop body preserves the invariant. The loop invariant may then be assumed for the statement following the loop. Expressed as traces, the loop invariant is a formula that holds in the first and last state generated by the loop body. The split into two steps to establish a loop invariant is necessary, because the first state of the first iteration is not under control by the loop body, but is established by the previous statement.

The above notion of loop invariants is the one used for postcondition reasoning, but we generalize it to other behavioral types: A loop invariant is a specification of the traces generated by the loop body. To be useful, a loop invariant must be chosen such that the specification of the overall statement can be proven.

Definition 4.25. *A rule is a basic-local loop invariant rule if it has the following form.*

$$(\textit{while})\ \frac{\begin{array}{c}\Rightarrow [\mathsf{s} \Vdash^{\alpha} \tau_1] \\ \Rightarrow [\mathsf{s}' \Vdash^{\alpha} \tau_2]\end{array}}{\Rightarrow [\mathbf{while}(\mathsf{e})\{\mathsf{s}\}\mathsf{s}' \Vdash^{\alpha} \tau]}$$

As other basic-local rules, this rule is imprecise and only models the aspect of the loop invariant, here the type τ_1.

Definition 4.26. *A rule is a leading loop invariant rule if it has the following form.*

$$(\textit{while})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\psi_1, \Delta \qquad \ldots \qquad \Gamma \Rightarrow \{U\}\psi_n, \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(\mathsf{e} \wedge \varphi_1) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathsf{s} \Vdash^{\alpha} \tau_1], \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(\neg\mathsf{e} \wedge \varphi_2) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathsf{s}' \Vdash^{\alpha} \tau_2], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{while}(\mathsf{e})\{\mathsf{s}\}\mathsf{s}' \Vdash^{\alpha} \tau], \Delta}$$

Composing such rules is straightforward.

Definition 4.27. *The composition of a leading loop invariant rule* (loop1) *and a basic-local loop invariant rule* (loop2)*, both of the following form:*

$$\textit{(loop1)}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\psi_1, \Delta \qquad \dots \qquad \Gamma \Rightarrow \{U\}\psi_n, \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(\mathsf{e} \wedge \varphi_1) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathsf{s} \Vdash^{\alpha} \tau_1], \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(\neg\mathsf{e} \wedge \varphi_2) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathsf{s}' \Vdash^{\alpha} \tau_2], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{while}(\mathsf{e})\{\mathsf{s}\}\mathsf{s}' \Vdash^{\alpha} \tau], \Delta} \qquad \textit{(loop2)}\ \frac{\begin{array}{c}\Rightarrow [\mathsf{s} \Vdash^{\alpha'} \tau_1'] \\ \Rightarrow [\mathsf{s}' \Vdash^{\alpha'} \tau_2']\end{array}}{\Rightarrow [\textbf{while}(\mathsf{e})\{\mathsf{s}\}\mathsf{s}' \Vdash^{\alpha'} \tau']}$$

Is the following rule (loop3)*:*

$$\textit{(loop3)}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\psi_1, \Delta \qquad \dots \qquad \Gamma \Rightarrow \{U\}\psi_n, \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(\mathsf{e} \wedge \varphi_1) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathsf{s} \Vdash^{\alpha \lozenge \alpha'} \tau_1 \lozenge \tau_1'], \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(\neg\mathsf{e} \wedge \varphi_2) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathsf{s}' \Vdash^{\alpha \lozenge \alpha'} \tau_2 \lozenge \tau_2'], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{while}(\mathsf{e})\{\mathsf{s}\}\mathsf{s}' \Vdash^{\alpha \lozenge \alpha'} \tau \lozenge \tau'], \Delta}$$

Theorem 3. *Let* (r1) *and* (r2) *be two loop invariant rules according to Def. 4.27. Furthermore, let* (r1) *be P-sound and leading,* (r2) *be P'-sound and basic-local.* (r1 ◊ r2) *is $P \cap P'$-sound and leading.*

Proof. See p. 174.

Finally, we consider the case where both rules provide context. This is covered by the leading rules of Def. 4.24 – we compose two leading rules by considering both added contexts.

Definition 4.28. *Let* (r1),(r2) *be two leading rules according to Def. 4.24 of the following form.*

$$\textit{(r1)}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\varphi_1, \Delta \quad \dots \quad \Gamma \Rightarrow \{U\}\varphi_n, \Delta \\ \Gamma, \{U_1\}\psi_1 \Rightarrow \{U_1\}[\mathsf{s}_1 \Vdash^{\alpha} \tau_1], \Delta \quad \dots \quad \Gamma, \{U_m\}\psi_m \Rightarrow \{U_m\}[\mathsf{s}_m \Vdash^{\alpha} \tau_m], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathsf{s} \Vdash^{\alpha} \tau], \Delta}$$

$$\textit{(r2)}\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U'\}\varphi_1', \Delta \quad \dots \quad \Gamma \Rightarrow \{U'\}\varphi_n', \Delta \\ \Gamma, \{U_1'\}\psi_1' \Rightarrow \{U_1'\}[\mathsf{s}_1' \Vdash^{\alpha'} \tau_1'], \Delta \quad \dots \quad \Gamma, \{U_m'\}\psi_m' \Rightarrow \{U_m'\}[\mathsf{s}_m' \Vdash^{\alpha'} \tau_m'], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathsf{s} \Vdash^{\alpha'} \tau'], \Delta}$$

The composition (r1 ◊ r2) *is defined as follows:*

- *The conclusion is* $\Gamma \Rightarrow \{U\}[\mathsf{s} \Vdash^{\alpha \lozenge \alpha'} \tau \lozenge \tau'], \Delta$
- *Each basic premise of* (r1) *and* (r2) *is a premise of* (r1 ◊ r2)
- *for any premise of* (r1) *represented by*

  $$\left(\Phi \cup \{\{U_i\}\psi_i\}, U_i, [\mathsf{s}_i \overset{\alpha}{\Vdash} \tau_i]\right)$$

  and any premise of (r2) *represented by*

  $$\left(\Phi \cup \{\{U_j\}\psi_j\}, U_j, [\mathsf{s}_j \overset{\alpha'}{\Vdash} \tau_j']\right)$$

  with $\mathsf{s}_i = \mathsf{s}_j$ *and* $U_i = U_j$ *the sequent represented by*

  $$\left(\Phi \cup \{\{U_i\}\psi_i, \{U_i\}\psi_j\}, U_i, [\mathsf{s}_i \overset{\alpha \lozenge \alpha'}{\Vdash} \tau_i \lozenge \tau_j']\right)$$

- *If some premise of (r1) or (r2) are not subsumed by the above construction, then it is also a premise of (r1 ◊ r2).*

Theorem 4. *Let (r1) and (r2) be two rules according to Def. 4.28. Furthermore, let (r1) be P-sound and (r2) be P'-sound. (r1 ◊ r2) is $P \cap P'$-sound and leading.*

Proof. See p. 174.

Definition 4.29 (Composition of Behavioral Types)**.** *Let $\mathbb{T}_1 = (\tau_1, \alpha_1, \iota_1, \pi_1)$ and $\mathbb{T}_2 = (\tau_2, \alpha_2, \iota_2, \pi_»)$ be two behavioral types. The composed behavioral specification $\mathbb{T}_1 \lozenge \mathbb{T}_2 = (\tau_1 \lozenge \tau_2, \alpha_1 \lozenge \alpha_2, \iota_1 \lozenge \iota_2, \pi_1 \lozenge \pi_2)$ is defined as follows, extending Def. 4.21.*

- *The proof obligation schema is defined as $\iota_1 \lozenge \iota_2(\mathtt{m}) = (\varphi_1 \wedge \varphi_2, \tau_1 \lozenge \tau_2)$ where $\iota_1(\mathtt{m}) = (\varphi_1, \tau_1)$ and $\iota_2(\mathtt{m}) = (\varphi_2, \tau_2)$*
- *The calculus $\pi_1 \lozenge \pi_2$ is defined by applying Def. 4.22, Def. 4.27 and Def. 4.28 to all symbolic execution rules. If some of the definitions cannot be applied because the rules do not match or some prerequisite of Thm. 2, Thm 3 or Thm. 4 is not satisfied, then composition fails. All rewrite rules are also added to $\pi_1 \lozenge \pi_2$.*

So far we only considered sound rules — merging with (or of) enabling rules in general does not preserve soundness or enablement. The reason is that sound rules contain all information needed to establish their soundness are contained in the premises — reasoning about the result of merging them can be done by assuming the validity of the original premises. Enabling rule, however, have *external* arguments for their enablement, which may or may not be composable. Even with this restriction, merging of calculi saves time when (re)proving soundness of the merged calculus, as only enabling rules must be reproven. These are, in the example that we have, only the rules concerned with **await** and **get**.

Theorem 5. *If $\mathbb{T}_1$ and $\mathbb{T}_2$ are sound, all sound rules of $\mathbb{T}_1$ are leading and all sound rules of $\mathbb{T}_2$ are either leading or basic-local then all rules of $\mathbb{T}_1 \lozenge \mathbb{T}_2$, that are not composed of enabling but unsound rules, are sound.*

Proof. See p. 175.

4.3.2 Led Object-Invariants

We now present the behavioral type $\mathbb{T}_{\mathsf{pst}}$, which we use to model the aspects of heap memory and postcondition and $\mathbb{T}_{\mathsf{inv}}$, which is modeling the aspect of object invariants. $\mathbb{T}_{\mathsf{pst}}$ is used as the leading type in the composed type we give in later chapters, $\mathbb{T}_{\mathsf{inv}}$ illustrates the use of composition.

Definition 4.30. *The behavioral type $\mathbb{T}_{\mathsf{pst}}$ extends the behavioral specification of Def. 4.4 with $(\iota_{\mathsf{pst}}, \pi_{\mathsf{pst}})$. There are no restrictions on ι_{pst}. The calculus π_{pst} is given in Fig. 4.6*

Rules **(awaitF)** and **(awaitB)** anonymize the heap, but not the other variables, because the local state does not change during suspension. Additionally, **last** is set to the heap after suspension. Rule **(while)** uses an invariant I that has to hold at the beginning of each loop iteration and, thus, can be assumed after the loop. The other rules are analogous to $\mathbb{T}_{\mathsf{sinv}}$ in Fig. 4.3. There is no global composition for $\mathbb{T}_{\mathsf{pst}}$.

The led object invariant type is analogous to the one in Def. 4.18, but with a far simpler calculus and can be led by the postcondition type to include loop invariants.

Definition 4.31. *Let C be a class and I a lFOL sentence containing only fields of C. The behavioral type $\mathbb{T}^I_{\mathsf{inv}} = (\tau_{\mathsf{inv}}, \alpha_{\mathsf{inv}}, \iota^I_{\mathsf{inv}}, \pi_{\mathsf{inv}})$ for C has as its syntax the set of all lFOS formulas which contain only fields of C. The semantics is defined as*

$$\alpha_{\mathsf{inv}}(\varphi) = \forall i \in \mathbf{I}.\ \big(\mathsf{isEvent}(i) \wedge \neg(\mathsf{isFutREv}(i) \vee \mathsf{isNoEv}(i) \vee \mathsf{isInvEv}(i))\big) \rightarrow [i+1] \vdash \varphi$$

The calculus is given in Fig. 4.7, the obligation schema is defined by $\iota^I_{\mathsf{inv}}(\mathtt{m}) = (I, I)$ for all method names.

$$(\mathbb{T}_{\text{pst}}\text{-assignV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \mathtt{e}\}[\mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e};\ \mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta} \qquad (\mathbb{T}_{\text{pst}}\text{-readV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := v\}[\mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e}.\mathbf{get};\ \mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}\ v \text{ fresh}$$

$$(\mathbb{T}_{\text{pst}}\text{-awaitF})\ \frac{\Gamma, \Rightarrow \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\mathbf{last} := \mathtt{heap}\}[\mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \mathtt{e?};\ \mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}$$

$$(\mathbb{T}_{\text{pst}}\text{-awaitB})\ \frac{\begin{array}{c}\Gamma, \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\mathbf{last} := \mathtt{heap}\}\mathtt{e} \Rightarrow \\ \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\mathbf{last} := \mathtt{heap}\}[\mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \mathtt{e};\ \mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}$$

$$(\mathbb{T}_{\text{pst}}\text{-callV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := v\}[\mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{f!m}(\mathtt{e}_1, \ldots, \mathtt{e}_n);\ \mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}\ v \text{ fresh}$$

$$(\mathbb{T}_{\text{pst}}\text{-if})\ \frac{\begin{array}{c}\Gamma, \{U\}\neg\mathtt{e} \Rightarrow \{U\}[\mathtt{s'}; \mathtt{s''} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta \\ \Gamma, \{U\}\mathtt{e} \Rightarrow \{U\}[\mathtt{s}; \mathtt{s''} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{if}(\mathtt{e})\{\mathtt{s}\}\mathbf{else}\{\mathtt{s'}\}; \mathtt{s''} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta} \qquad (\mathbb{T}_{\text{pst}}\text{-return})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{result} := \mathtt{e}\}\varphi, \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{return}\ \mathtt{e} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}$$

$$(\mathbb{T}_{\text{pst}}\text{-skipCont})\ \frac{\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{skip};\ \mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta} \qquad (\mathbb{T}_{\text{pst}}\text{-skip})\ \frac{\Rightarrow \{U\}\varphi}{\Gamma \Rightarrow \{U\}[\mathbf{skip} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}$$

$$(\mathbb{T}_{\text{pst}}\text{-while})\ \frac{\begin{array}{c}\Gamma, \{U\}\{U_{\mathscr{A}}\}(I \wedge \mathtt{e}) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s} \overset{\alpha_{\text{pst}}}{\Vdash} I], \Delta \\ \Gamma \Rightarrow \{U\}I, \Delta \qquad \Gamma, \{U\}\{U_{\mathscr{A}}\}(I \wedge \neg\mathtt{e}) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s'} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{while}(\mathtt{e})\{\mathtt{s}\}; \mathtt{s'} \overset{\alpha_{\text{pst}}}{\Vdash} \varphi], \Delta}$$

Figure 4.6: Rules for π_{pst}. Rules for variable assignments with declaration are analogous. $U_{\mathscr{A}}$ assigns to heap the term anon(heap), and every other program variable except **old** a fresh function symbol.

$$(\mathbb{T}_{\text{inv}}\text{-assignV})\ \frac{\Rightarrow [\mathtt{s} \overset{\alpha_{\text{inv}}}{\Vdash} I]}{\Rightarrow [\mathtt{v = e;\ s} \overset{\alpha_{\text{inv}}}{\Vdash} I]} \qquad (\mathbb{T}_{\text{inv}}\text{-readV})\ \frac{\Rightarrow [\mathtt{s} \overset{\alpha_{\text{inv}}}{\Vdash} I]}{\Rightarrow [\mathtt{v = e.}\textbf{get}\mathtt{;\ s} \overset{\alpha_{\text{inv}}}{\Vdash} I]}$$

$$(\mathbb{T}_{\text{inv}}\text{-awaitF})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}I, \Delta \\ \Gamma, \{U\}\{\text{heap} := \text{anon}(\text{heap})\}\{\textbf{last} := \text{heap}\}I \Rightarrow \\ \{U\}\{\text{heap} := \text{anon}(\text{heap})\}\{\textbf{last} := \text{heap}\}[\mathtt{s} \overset{\alpha_{\text{inv}}}{\Vdash} I], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{await}\ \mathtt{e?;\ s} \overset{\alpha_{\text{inv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{inv}}\text{-awaitB})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}I, \Delta \\ \Gamma, \{U\}\{\text{heap} := \text{anon}(\text{heap})\}(I \wedge \mathtt{e}) \Rightarrow \\ \{U\}\{\text{heap} := \text{anon}(\text{heap})\}\{\textbf{last} := \text{heap}\}[\mathtt{s} \overset{\alpha_{\text{inv}}}{\Vdash} I], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{await}\ \mathtt{e;\ s} \overset{\alpha_{\text{inv}}}{\Vdash} I], \Delta}$$

$$(\mathbb{T}_{\text{inv}}\text{-callV})\ \frac{\Rightarrow [\mathtt{s} \overset{\alpha_{\text{inv}}}{\Vdash} I]}{\Rightarrow [\mathtt{v = f!m}(\mathtt{e}_1, \ldots, \mathtt{e}_n)\mathtt{;\ s} \overset{\alpha_{\text{inv}}}{\Vdash} I]} \qquad (\mathbb{T}_{\text{inv}}\text{-if})\ \frac{\begin{array}{c}\Rightarrow [\mathtt{s';s''} \overset{\alpha_{\text{inv}}}{\Vdash} I] \\ \Rightarrow [\mathtt{s;s''} \overset{\alpha_{\text{inv}}}{\Vdash} I]\end{array}}{\Rightarrow [\textbf{if}\mathtt{(e)\{s\}}\textbf{else}\mathtt{\{s'\};s''} \overset{\alpha_{\text{inv}}}{\Vdash} I]}$$

$$(\mathbb{T}_{\text{inv}}\text{-return})\ \frac{\Rightarrow I}{\Rightarrow [\textbf{return}\ \mathtt{e} \overset{\alpha_{\text{inv}}}{\Vdash} I]} \qquad (\mathbb{T}_{\text{inv}}\text{-skipCont})\ \frac{\Rightarrow [\mathtt{s} \overset{\alpha_{\text{inv}}}{\Vdash} I]}{\Rightarrow [\textbf{skip}\mathtt{;\ s} \overset{\alpha_{\text{inv}}}{\Vdash} I]} \qquad (\mathbb{T}_{\text{inv}}\text{-skip})\ \frac{}{\Rightarrow [\textbf{skip} \overset{\alpha_{\text{inv}}}{\Vdash} I]}$$

$$(\mathbb{T}_{\text{inv}}\text{-while})\ \frac{\Rightarrow [\mathtt{s'} \overset{\alpha_{\text{inv}}}{\Vdash} I] \qquad \Rightarrow [\mathtt{s} \overset{\alpha_{\text{inv}}}{\Vdash} I]}{\Rightarrow [\textbf{while}\mathtt{(e)\{s\};s'} \overset{\alpha_{\text{inv}}}{\Vdash} I]}$$

Figure 4.7: Rules for π_{inv}. Rules for variable assignments with declaration are analogous.

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{assignV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \mathtt{e}\}[\mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v}\ =\ \mathtt{e};\ \ \mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{readV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := v\}[\mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v}\ =\ \mathtt{e}.\mathbf{get};\ \ \mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}\ v \text{ fresh}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{awaitF})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}I, \Delta \\ \Gamma, \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\mathbf{last} := \mathtt{heap}\}I \Rightarrow \\ \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\mathbf{last} := \mathtt{heap}\}[\mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \ \mathtt{e?};\ \ \mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{awaitB})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}I, \Delta \\ \Gamma, \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\mathbf{last} := \mathtt{heap}\}(I \wedge \mathtt{e}) \Rightarrow \\ \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\mathbf{last} := \mathtt{heap}\}[\mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \ \mathtt{e};\ \ \mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{callV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := v\}[\mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v}\ =\ \mathtt{f!m}(\mathtt{e}_1, \ldots, \mathtt{e}_n);\ \ \mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}\ v \text{ fresh}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{if})\ \frac{\Gamma, \{U\}\neg\mathtt{e} \Rightarrow \{U\}[\mathtt{s'};\mathtt{s''}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta \qquad \Gamma, \{U\}\mathtt{e} \Rightarrow \{U\}[\mathtt{s};\mathtt{s''}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{if}(\mathtt{e})\{\mathtt{s}\}\mathbf{else}\{\mathtt{s'}\};\mathtt{s''}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{return})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{result} := \mathtt{e}\}\varphi, \Delta \qquad \Gamma \Rightarrow \{U\}\{\mathtt{result} := \mathtt{e}\}I, \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{return}\ \ \mathtt{e}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{skipCont})\ \frac{\Gamma \Rightarrow \{U\}[\mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{skip};\ \ \mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{skip})\ \frac{\Gamma \Rightarrow \{U\}\varphi, \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{skip}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}$$

$$(\mathbb{T}_{\mathbf{pst}}\,\between\,\mathbb{T}_{\mathbf{inv}}\text{-}\mathbf{while})\ \frac{\begin{array}{c}\Gamma, \{U\}\{U_{\mathscr{A}}\}(I \wedge \mathtt{e}) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ I' \between I], \Delta \\ \Gamma \Rightarrow \{U\}I', \Delta \qquad \Gamma, \{U\}\{U_{\mathscr{A}}\}(I' \wedge \neg\mathtt{e}) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s'}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathbf{while}(\mathtt{e})\{\mathtt{s}\};\mathtt{s'}\,|\!|\!-^{\alpha_{\mathsf{pst}}\between\alpha_{\mathsf{inv}}}\ \varphi \between I], \Delta}$$

Figure 4.8: Rules for $\pi_{\mathsf{pst}} \between \pi_{\mathsf{inv}}$. Rules for variable assignments with declaration are analogous.

Lemma 4.11. $\mathbb{T}_{\mathsf{pst}}$ *is sound and leading.* $\mathbb{T}_{\mathsf{inv}}$ *is sound.*

Proof. See p. 175.

The composed type is now able to handle object and loop invariants. The following directly follows.

Lemma 4.12. $\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}^I_{\mathsf{inv}}$ *is sound and leading. Fig. 4.8 gives the calculus* $\pi_{\mathsf{pst}} \between \pi_{\mathsf{inv}}$*. Lemma 4.7 holds for* $\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}^I_{\mathsf{inv}}$ *under the same assumptions, as long as the obligation schema* ι_{pst} *uses* true *as its precondition.*

Proof. See p. 175.

The rules ($\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}_{\mathsf{inv}}$-awaitF) and ($\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}_{\mathsf{inv}}$-awaitB) are enabling, the proof is identical to the one in Lemma 4.6 because we do not need to consider anything from the leading type. We note that $\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}_{\mathsf{inv}}$ strongly resembles ABSDL [44], but contrary to ABSDL, the object invariant in $\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}_{\mathsf{inv}}$ is explicit.

4.3.3 Skipping Postconditions

The type $\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}_{\mathsf{inv}}$ demonstrates the usage of leading composition, but other ways of composition are also possible and useful. For example, discharging proof obligations to remove dependency edges from deadlock graphs [78] requires that a condition has to hold at every point the method releases control *if it has regained control again*. I.e., it does not have to hold at the first suspension and not at the termination if there was no suspension. We can encode the system given in [78] by combining a new behavioral type with $\mathbb{T}_{\mathsf{pst}}$.

Example 4.7. *The behavioral type for* skipping postconditions *is* $\mathbb{T}_{\mathsf{skip}} = (\tau_{\mathsf{skip}}, \alpha_{\mathsf{skip}}, \iota_{\mathsf{skip}}, \pi_{\mathsf{skip}})$.

- *The syntax* τ_{skip} *is again the set of all lFOS sentences not containing* result.
- *The semantics is defined as*

$$\begin{aligned}\alpha_{\mathsf{skip}}(\varphi) = \exists i \in \mathbf{I}.\, \Big(&\big(\mathsf{isSuspREv}(i) \vee \mathsf{isCondREv}(i)\big) \wedge \\ &\big(\forall j \in \mathbf{I}.\, ((\mathsf{isSuspREv}(j) \vee \mathsf{isCondREv}(j)) \wedge j \geq i)\big) \wedge \\ &\alpha_{\mathsf{susp}}(\varphi)[k \in \mathbf{I} \setminus k \geq i+2]\Big)\end{aligned}$$

- *The obligation schema is provided by the user.*
- *The calculus is analogous to* π_{pst}*, except the rules given in Fig. 4.10 and that there is no loop invariant rule.*

The behavioral type for suspending postcondition *is* $\mathbb{T}_{\mathsf{susp}} = (\tau_{\mathsf{susp}}, \alpha_{\mathsf{susp}}, \iota_{\mathsf{susp}}, \pi_{\mathsf{susp}})$

- *The syntax* τ_{susp} *is again the set of all lFOS formulas (not containing* result*).*
- *The semantics is defined as*

$$\alpha_{\mathsf{susp}}(\varphi) = \alpha_{\mathsf{pst}}(\varphi) \wedge \forall i \in \mathbf{I}.\, \Big(\big(\mathsf{isSuspEv}(i) \vee \mathsf{isCondEv}(i)\big) \rightarrow [i+1] \vdash \varphi\Big)$$

- *The obligation schema is provided by the user.*
- *The calculus is analogous to* π_{pst}*, except the rules given in Fig. 4.11 and that there is no loop invariant rule.*

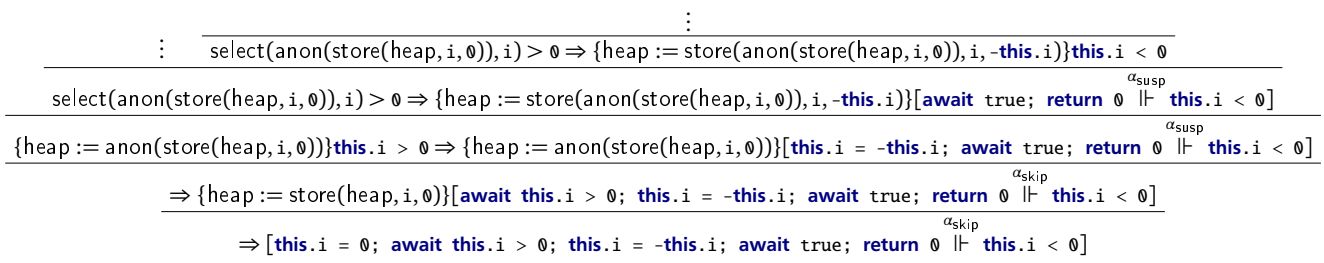


Figure 4.9: A proof with skipping postconditions.

$$(\mathbb{T}_{\mathbf{skip}}\text{-awaitF})\ \frac{\Gamma \Rightarrow \{U\}\{\mathrm{heap} := \mathrm{anon}(\mathrm{heap})\}\{\mathbf{last} := \mathrm{heap}\}[\mathtt{s}\ \overset{\alpha_{\mathrm{susp}}}{\Vdash}\ \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \mathtt{e?};\ \mathtt{s}\ \overset{\alpha_{\mathrm{skip}}}{\Vdash}\ \varphi], \Delta}$$

$$(\mathbb{T}_{\mathbf{skip}}\text{-awaitB})\ \frac{\Gamma, \{U\}\{\mathrm{heap} := \mathrm{anon}(\mathrm{heap})\}\{\mathbf{last} := \mathrm{heap}\}\mathtt{e} \Rightarrow \{U\}\{\mathrm{heap} := \mathrm{anon}(\mathrm{heap})\}\{\mathbf{last} := \mathrm{heap}\}[\mathtt{s}\ \overset{\alpha_{\mathrm{susp}}}{\Vdash}\ \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \mathtt{e};\ \mathtt{s}\ \overset{\alpha_{\mathrm{skip}}}{\Vdash}\ \varphi], \Delta}$$

$$(\mathbb{T}_{\mathbf{skip}}\text{-return})\ \frac{}{\Gamma \Rightarrow \{U\}[\mathbf{return}\ \mathtt{e}\ \overset{\alpha_{\mathrm{skip}}}{\Vdash}\ \varphi], \Delta} \qquad (\mathbb{T}_{\mathbf{skip}}\text{-skip})\ \frac{}{\Gamma \Rightarrow \{U\}[\mathbf{skip}\ \overset{\alpha_{\mathrm{skip}}}{\Vdash}\ \varphi], \Delta}$$

Figure 4.10: Selected Rules for π_{skip}.

$\mathbb{T}_{\mathrm{susp}}$ is integrated into $\mathbb{T}_{\mathrm{skip}}$ by falling back to it once a suspension point has been detected. Note that we give *no* loop invariant rule. The loop has to be explicitly unrolled once to detect the first suspension. One possible way to amend this is to restrict loop invariants to loop bodies without suspensions and use two premises using $\mathbb{T}_{\mathrm{pst}}$ as in the informal example in the beginning of this section. For example, one can prove that `this.i = 0; await this.i > 0; this.i = -this.i; await true; return 0` establishes `this.i < 0` at all release points, but the first one. This is shown in Fig. 4.9.

The update is slightly beautified for readability. Note the change from $\overset{\alpha_{\mathrm{skip}}}{\Vdash}$ to $\overset{\alpha_{\mathrm{susp}}}{\Vdash}$. The proof of Fig. 4.9 cannot be translated directly neither into object invariants (because the first suspension does not need to prove the invariant and no suspension may assume an invariant) nor into postconditions, because the condition has to be proven at some suspensions additionally to the very end.

4.4 Discussion

On Behavioral Types.

To the author's best knowledge, there is no formal definition of the term "behavioral type" in the literature. In the two books providing an overview over behavioral types, they are defined as descriptions of programs, "*in terms of the sequences of operations that allow for a correct interaction among the involved entities*" [1] or systems that aim "*to describe properties associated with the behavior of programs and in this way also describe how a computation proceeds.*" [74].

BPL behavioral types fit both definitions, even the simple object invariants type above: It describes a property associated with the behavior of the program (an invariant) and so describes how a computation proceeds (by preserving it). Similarly, the invariant allows for a correct interaction among involved entities, which are the processes of the object in question.

$$(\mathbb{T}_{\mathsf{susp}}\text{-awaitF})\ \frac{\Gamma \Rightarrow \{U\}\varphi, \Delta \qquad \Gamma \Rightarrow \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\textbf{last} := \mathtt{heap}\}[\mathtt{s} \overset{\alpha_{\mathsf{susp}}}{\Vdash} \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\textbf{await}\ \mathtt{e?};\ \mathtt{s} \overset{\alpha_{\mathsf{susp}}}{\Vdash} \varphi], \Delta}$$

$$(\mathbb{T}_{\mathsf{susp}}\text{-awaitB})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\varphi, \Delta \\ \Gamma, \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\textbf{last} := \mathtt{heap}\}\mathtt{e} \Rightarrow \\ \{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}\{\textbf{last} := \mathtt{heap}\}[\mathtt{s} \overset{\alpha_{\mathsf{susp}}}{\Vdash} \varphi], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{await}\ \mathtt{e};\ \mathtt{s} \overset{\alpha_{\mathsf{susp}}}{\Vdash} \varphi], \Delta}$$

Figure 4.11: Selected Rules for π_{susp}.

Classical dynamic logic systems for object invariants, e.g., ABSDL [44], also fit these definitions. For this reason, we deem the above definitions as not practical to distinguish between behavioral types and other static analyses for functional properties.

Behavioral types are sometimes (informally) distinguished from data types by having a subject reduction theorem where the typing relation is preserved, but not the type itself [41]. This fits the intuition that invariants are not behavioral types, but does not allow for a denotational semantics for behavioral types by requiring a subject reduction theorem as a specific form of soundness. It does, however, fit the intuition of the behavioral type better: it must include a type system, i.e., a calculus which matches type and statement in some context and it must describe behavior in the sense that a substatement may have a different type than the superstatement. The object invariant behavioral type only uses trivial matching of type and statement, but the system we give in the next sections show that BPL can handle more complex matching. In this sense, the behavioral modality can be seen as a generalization of a type judgment, where the context is encoded in the premise of the sequence and the update before the modality. The calculus can however connect multiple modalities. This generalization mirrors the generalization of Hoare Triples to standard dynamic logic modalities.

We can amend our definition and the one above by distinguishing between BPL behavioral types in the narrower sense and BPL behavioral types in the broader sense. For BPL behavioral types in the narrower sense, we demand that the type system must involve a rule where the type in the modality of a premise differs from the one in the conclusion. For BPL behavioral types in the broader sense, we do not impose this restriction. This fits the intuition that object invariants are not behavioral types (in the narrower sense), but Session Types (see chapter 7) are. In the rest of this work, we use behavioral type always in the broader sense.

We, thus, argue that BPL generalizes behavioral types (for Active Objects) as well as dynamic logic.

On Enablement.

Enablement is normally subsumed under the term “soundness” and occurs in the use of method contracts already for simple sequential programming languages. The difficulties we faced occur in the case of recursive synchronous method contracts. Enabling rules, just as method contracts of JavaDL [62], bind our calculus to a specific purpose. Sound rules reason about modalities without any context, while enabling rules require global information (i.e., not visible in the proof).

Enablement is not soundiness [101]. Soundiness describes that in many development contexts, users may accept unsound analyses. Enablement ensures that the overall analysis is sound, merely intermediate steps *which are not exposed to the user* are unsound.

On Completeness and Subtyping.

We do no discuss completeness. For $\mathbb{T}_{\mathsf{pst}}$, $\mathbb{T}_{\mathsf{inv}}$ and $\mathbb{T}_{|\mathsf{t}|}$ we conjecture that completeness results from [8, 44, 7] carry over. Similarly, we do not discuss subtyping. We conjecture that for two types τ_1, τ_2 of some behavioral type $\mathbb{T}$, one can formulate that τ_1 is a subtype of τ_2 iff $\alpha(\tau_1)$ implies $\alpha(\tau_2)$. This can be integrated into the calculus with a rewrite rule $\tau_1 \rightsquigarrow \tau_2$ with the side condition that τ_1 is a subtype of τ_2. For loop invariants this corresponds to proving a stronger loop invariant. We discuss behavioral subtyping for method contracts in chapter 6, but not general subtyping.

On the Difference of Behavioral Specifications and Logics.

One may wonder why a behavioral specification is not a logic, as it has a syntax, a class of structures it is evaluated upon and a satisfiability relation between a syntactic expression and a structure from this set. Some lines of work in abstract model theory require only such a triple to define a logic [54].

For the notion of a logic, we follow a more restricted view, that requires certain additional properties (e.g., closure under negation and relativization [28, 46]). We deem this variant more appropriate as it allows for characterization of logics, via, e.g., Lindström theorems [98, 123] and captures our intuition about logics better.

As we have no restrictions for the syntax of behavioral specifications, they are not a logic in this sense. Some of them, such as $\mathbb{T}_{\mathsf{pst}}$, however, are logics.

On Composition.

The composition we give requires structurally similar rules in two ways:

- The semantics of the behavioral type must be of a form that that allows to decompose the satisfiability of the semantics of a type for a given statement into the satisfiability for the first state and semantics of (different) types for the substatements. This does *not* impose the restriction that the first state is isolated, because if the first state influences the satisfiability of the rest of the trace, this may be encoded in the type for the substatement.
- Repetition is handled solely by loop invariant rules, it is an open question to define leading composition for loop scopes [121] or recursive synchronous method contracts or how to compose two rules which use different mechanisms for repetition.

On Histories.

Contrary to ABSDL [44], the behavioral type $\mathbb{T}_{\mathsf{inv}}$ does not allow to specify properties of the events in the trace as part of the invariants. To do so, a special program variable `history` keeps track of the events. E.g., ABSDL allows the following invariant (simplified for readability):

$$\mathbf{this}.\mathtt{i} > 0 \rightarrow \exists s \in \mathsf{Seq}.\ \mathtt{history} \doteq s \circ \langle \mathsf{futEv}(\mathtt{m}) \rangle$$

This expresses that if *after a suspension or method termination*, the field `i` is strictly positive, then the last event was a termination of method `m`. I.e., $\mathbf{this}.\mathtt{i} > 0$ holds only after `m` terminates.

We omitted this for two reasons:

- It is straightforward to add such a variable to $\mathbb{T}_{\mathsf{inv}}$ analogous to the work of Din et al. [44], yet it would dilute the presentation of BPL if included here.
- In the rest of this work we show how to verify trace properties *without explicitly* keeping track of the trace. Instead, traces are verified on-the-fly during symbolic execution of the statement in a modality.

5 Effect Types in BPL

Using the semantic effects, we give two behavioral type systems, similar to type-and-effect systems [108, 58] to verify properties concerning side-effects. The side-effects in question are memory read and write accesses, as well as synchronization. The first behavioral type expresses framing for methods, i.e., that a method only reads and writes fields it is specified to read or write. We use this *frame type* to specify and check dynamic frames. The second behavioral type, the *sequential effects type*, specifies for each field the allowed order of read and write accesses. This is a useful property for optimization, e.g., for parallelization [18]. For sequential effects, we distinguish between read accesses to copy the value and read access in suspension statements, which allows to express that a field may be read, but not in a guard. This may be not desirable if the number of read accesses is important, as the scheduler may evaluate a guard multiple times. Alternatively, one may verify that the scheduler never accesses a field at all.

5.1 Frames

We give a simple dynamic frame [92] system for CAO. The frame is a pair $(\mathfrak{R}, \mathfrak{W})$ of sets of field names. Intuitively, it specifies that the method may only change the fields in $\mathfrak{W}$ and only depends on the values in $\mathfrak{R}$. The $\mathfrak{W}$ set allows us, for example, simple checking of object invariants: if none of the fields used in an object invariant is written in a method, then it is obvious that the method preserves the invariant. The $\mathfrak{R}$ set allows us to reason about when two method calls result in the same execution: If the parameters and the values in $\mathfrak{R}$ are equal and no suspension occurs, then two method calls return the same value. In the framework of JavaDL [62], frames are only given for a sequential language, which makes the method-wide read location sets more useful. For CAO we generalize the JavaDL system and have a frame for *each point the method gains control*. I.e., for each suspension point and for the method start we give a frame which has to be observed until the next suspension point (or termination). This also makes the frames *dynamic*: the current frame changes during execution of the method. Instead of using model fields like JavaDL, we use program-point identifiers and an external specification.

The formalization in JavaDL verifies that all reads on fields not in $\mathfrak{R}$ do not influence the execution. This cannot be expressed just by checking whether the fields syntactically occur in the method and was encoded in the postcondition of a DL modality. In contrast, we forbid such reads syntactically, but not by analyzing the mere presence in the code — instead we check whether their occurrence in the code is executable given some specification, e.g., a method contract. This is not dead-code detection: A method may have a more restrictive frame when used in a protocol than as specified in its method contract or have method contracts with multiple frames. In practice, a method (or suspension point) can have *multiple* frames, depending on the context. Thus, the specification must be known to analyze whether a given statement is reached *in this situation*. We give the behavioral type for frames without any such specification. When used with a more precise specification, e.g., object invariants, the type must be led by the type verifying this invariant.

Definition 5.1. *Let* F *be the set of all fields. The behavioral type* $\mathbb{T}_{\mathsf{fr}}$ *for frames is defined as*

$$\mathbb{T}_{\mathsf{fr}} = \left(\mathscr{P}(\mathsf{F}) \times \mathscr{P}(\mathsf{F}) \times \left(\mathsf{P} \to \left(\mathscr{P}(\mathsf{F}) \times \mathscr{P}(\mathsf{F})\right)\right), \alpha_{\mathsf{fr}}, \iota_{\mathsf{fr}}, \pi_{\mathsf{fr}}\right)$$

The syntax of a frame type is a triple $(\mathfrak{R}, \mathfrak{W}, \mathfrak{S})$. *The first two elements are sets of field names, the last one is a map from program-point identifiers to pairs of sets of field names. We say that* $\mathfrak{R}$ *is the read set and* $\mathfrak{W}$ *is the write set and analogous for the pairs in the image of* $\mathfrak{S}$.

The obligation scheme ι_{fr} maps every method $\mathtt{m}$ to some frame

$$\iota_{\mathsf{fr}}(\mathtt{m}) = \big(\mathbf{true}, (\mathfrak{R}_{\mathtt{m}}, \mathfrak{W}_{\mathtt{m}}, \mathfrak{S}_{\mathtt{m}})\big)$$

The semantics of a type $(\mathfrak{R}, \mathfrak{W}, \mathfrak{S})$ is that only fields from $\mathfrak{R}$ are read and only fields from $\mathfrak{W}$ are written up to the next suspension and that $\mathfrak{S}(i)$ holds for all traces between two suspension points starting with the suspension at point i. Given a pair $\mathsf{rw} = (\mathfrak{R}, \mathfrak{W})$, we write $\mathfrak{R}(\mathsf{rw})$ to select the read set and $\mathfrak{W}(\mathsf{rw})$ to select the write set.

First, the semantics of a $(\mathfrak{R}, \mathfrak{W})$ pair, starting from a position i. We use a predicate $\mathsf{isLast}(i, j)$ that holds iff j is either the index of the first suspension event after i, or the last event in the trace if it contains no suspensions.

$$\mathsf{sem}\big(i, (\mathfrak{R}, \mathfrak{W})\big) =$$

$$\exists j \in \mathbf{I}.\ \Big(\mathsf{isLast}(i, j) \wedge \forall k \in \mathbf{I}.\ \Big(i \le k \le j \wedge \mathsf{isEvent}(k) \rightarrow \big(\forall f \in \mathsf{F}.\ \big((f_{\mathsf{R}} \in \mathsf{eff}(k) \rightarrow f \in \mathfrak{R}) \wedge (f_{\mathsf{W}} \in \mathsf{eff}(k) \rightarrow f \in \mathfrak{W})\big)\big)\Big)\Big)$$

The semantics expresses that the active pair holds up to the next suspension (or the end) and that after each suspension with PPI $\mathtt{p}$, the frame from $\mathfrak{S}(\mathtt{p})$ holds. The set $\mathfrak{P}$ of program-point identifiers is finite for a given method *and the formula is finite.*

$$\alpha_{\mathsf{fr}}\big((\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\big) = \exists i \in \mathbf{I}.\ \forall j \in \mathbf{I}.\ j \ge i \wedge \mathsf{sem}\big(i, (\mathfrak{R}, \mathfrak{W})\big) \wedge \bigwedge_{\mathtt{p} \in \mathbf{dom}(\mathfrak{S})} \forall i \in \mathbf{I}.\ \mathtt{p}_{\mathsf{E}} \in \mathsf{eff}(i) \rightarrow \mathsf{sem}\big(i, \mathfrak{S}(\mathtt{p})\big)$$

The calculus is given in Fig. 5.1 and the rules are straightforward, checking that every field that is evaluated is allowed to be read and every assigned field is allowed to be written. Note that we do not reason about local variables, only fields. The only more complex rule is the one for loops: Here, one must give the subsets of the read and write sets, because inside the loop there may be a suspension. The statement after the loop and the loop body up to the first suspension has to observe the read (and write) set of the previous suspension point before the loop *and the read (or write) set of the last suspension point(s) inside the loop body*. Thus, the loop body, the loop guard and the statement after the loop are all typed with the intersection of the type before entering the loop and the types of the last suspension points within the loop. We omit the context in all rules because we expect that when the frame is verified, a leading type is provided for precision.

Example 5.1. *Consider the method in Fig. 5.2a and the two frame types given there. The method can be typed with the first type, but it cannot be typed with the second one: It will read from* `f` *after executing the statement identified by* 1 *and this error is caught by the intersection in the loop rule. The two proof trees for the method are given in Fig. 5.2b.*

We can, however, show that the method does adhere to the second frame type if the behavioral type is led by the postcondition behavioral type and assume that the parameter `i` *is always negative and the loop is thus never executed. For readability, we write $(\mathfrak{R}, \mathfrak{W}, \mathfrak{S})$ for* $\mathsf{true} \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})$*. To integrate $\mathbb{T}_{\mathsf{pst}}$ we use the rule from Lem. 4.10. This is shown in Fig. 5.3. Loop unrolling can be realized with a (trivially sound) rewrite rule*

while`(e){s}s'` ⇝ **if**`(e){s;` **while**`(e){s}`**skip**`}`**else**`{`**skip**`}s'`

The calculus π_{fr} at no point uses logical operations beyond manipulations of the modality. The above example justifies that we nonetheless express it in BPL: by composing it with further calculi, we can analyze whether a method adheres to its frame *in a given context*. Similarly, we can analyze different frames in different contexts, e.g., a more restrictive frame can be used when a method is analyzed in the context of a protocol then in the context of a method contract. By expressing dynamic frames as behavioral types we do not need to encode them in the postcondition.

Lemma 5.1. *Type $\mathbb{T}_{\mathsf{fr}}$ is sound, all rules are basic-local and none of the rules is enabling but not sound.*

$$(\mathbb{T}_{\mathrm{fr}}\text{-assignV})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}{\Rightarrow \left[\mathtt{v = e;\ s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \mathrm{loc}(\mathtt{e}) \subseteq \mathfrak{R} \qquad (\mathbb{T}_{\mathrm{fr}}\text{-rdV})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}{\Rightarrow \left[\mathtt{v = e.}\textbf{get}\mathtt{;\ s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \mathrm{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\mathrm{fr}}\text{-assignF})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}{\Rightarrow \left[\textbf{this}\mathtt{.f = e;\ s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \mathrm{loc}(\mathtt{e}) \subseteq \mathfrak{R}, \mathtt{f} \in \mathfrak{W}$$

$$(\mathbb{T}_{\mathrm{fr}}\text{-awaitF})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}(\mathfrak{S}(\mathtt{p})), \mathfrak{W}(\mathfrak{S}(\mathtt{p})), \mathfrak{S})\right]}{\Rightarrow \left[\textbf{await}\ \mathtt{e?_p;\ s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \mathrm{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\mathrm{fr}}\text{-awaitB})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}(\mathfrak{S}(\mathtt{p})), \mathfrak{W}(\mathfrak{S}(\mathtt{p})), \mathfrak{S})\right]}{\Rightarrow \left[\textbf{await}\ \mathtt{e_p;\ s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \mathrm{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\mathrm{fr}}\text{-callV})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}{\Rightarrow \left[\mathtt{v = f!m(e1,...,en);\ s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \bigcup_{i \leq n} \mathrm{loc}(\mathtt{e}i) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\mathrm{fr}}\text{-if})\ \frac{\Rightarrow \left[\mathtt{s';s''} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right] \qquad \Rightarrow \left[\mathtt{s;s''} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}{\Rightarrow \left[\textbf{if}\mathtt{(e)\{s\}}\textbf{else}\mathtt{\{s'\}s''} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \mathrm{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\mathrm{fr}}\text{-skip})\ \frac{}{\Rightarrow \left[\textbf{skip} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]} \qquad (\mathbb{T}_{\mathrm{fr}}\text{-skipCont})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}{\Rightarrow \left[\textbf{skip}\mathtt{;\ s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}$$

$$(\mathbb{T}_{\mathrm{fr}}\text{-return})\ \frac{}{\Rightarrow \left[\textbf{return}\ \mathtt{e} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \mathrm{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\mathrm{fr}}\text{-while})\ \frac{\begin{array}{c}\Rightarrow \left[\mathtt{s} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R} \cap \bigcap_{i \in P'} \mathfrak{R}(\mathfrak{S}(i)), \mathfrak{W} \cap \bigcap_{i \in P'} \mathfrak{W}(\mathfrak{S}(i)), \mathfrak{S})\right] \\ \Rightarrow \left[\mathtt{s'} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R} \cap \bigcap_{i \in P'} \mathfrak{R}(\mathfrak{S}(i)), \mathfrak{W} \cap \bigcap_{i \in P'} \mathfrak{W}(\mathfrak{S}(i)), \mathfrak{S})\right]\end{array}}{\Rightarrow \left[\textbf{while}\mathtt{(e)\{s\};s'} \overset{\alpha_{\mathrm{fr}}}{\Vdash} (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right]}\ \mathrm{loc}(\mathtt{e}) \subseteq (\mathfrak{R} \cap \bigcap_{i \in P'} \mathfrak{R}(\mathfrak{S}(i)))$$

Figure 5.1: Calculus π_{fr} for the frame type $\mathbb{T}_{\mathrm{fr}}$. P' is the set of final program-point identifiers of suspensions inside the loop.

```
1 Int m(Int i){
2   this.f = i;
3   while(i > 0){
4       await this.f != i1;
5       this.c = this.c + 1;
6   }
7   return i;
8 }
```

$$\big(\{f\},\{f\},\{1 \mapsto (\{c,f\},\{c\})\big)$$
$$\big(\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\big)$$

(a) Code and Specification

$$\dfrac{\dfrac{\dfrac{}{\Rightarrow [\texttt{return i;} \Vdash^{\alpha_{fr}} (\emptyset,\{f\},\{1 \mapsto (\{c,f\},\{c\})\})]} \quad \dfrac{\dfrac{\dfrac{}{\Rightarrow [\texttt{skip;} \Vdash^{\alpha_{fr}} (\emptyset,\{c\},\{1 \mapsto (\{c,f\},\{c\})\})]}}{\Rightarrow [\texttt{this.c = this.c+1; skip;} \Vdash^{\alpha_{fr}} (\{c,f\},\{c\},\{1 \mapsto (\{c,f\},\{c\})\})]}}{\Rightarrow [\texttt{await this.f != i;}_1\texttt{this.c = this.c+1; skip;} \Vdash^{\alpha_{fr}} (\{f\},\emptyset,\{1 \mapsto (\{c,f\},\{c\})\})]}}{\Rightarrow [\texttt{while(i>0)\{await this.f != i;}_1\texttt{this.c = this.c+1; skip;\}return i;} \Vdash^{\alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c,f\},\{c\})\})]}}{\Rightarrow [\texttt{this.f = i; while(i>0)\{await this.f != i;}_1\texttt{this.c = this.c+1; skip;\}return i;} \Vdash^{\alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c,f\},\{c\})\})]}$$

$$\dfrac{\dfrac{\dfrac{}{\Rightarrow [\texttt{return i;} \Vdash^{\alpha_{fr}} (\emptyset,\emptyset,\{1 \mapsto (\{c\},\{c\})\})]} \quad \dfrac{\dfrac{}{\Rightarrow [\texttt{await this.f != i;}_1\texttt{this.c = this.c+1; skip;} \Vdash^{\alpha_{fr}} (\emptyset,\emptyset,\{1 \mapsto (\{c\},\{c\})\})]}}{\Rightarrow [\texttt{while(i>0)\{await this.f != i;}_1\texttt{this.c = this.c+1; skip;\}return i;} \Vdash^{\alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{\Rightarrow [\texttt{this.f = i; while(i>0)\{await this.f != i;}_1\texttt{this.c = this.c+1; skip;\}return i;} \Vdash^{\alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}$$

(b) Proof Tree

Figure 5.2: Example for dynamic frames.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{i} < 0 \Rightarrow \{\text{heap} := \text{store}(\text{heap},f,i)\}\text{true}}}{\texttt{i} < 0 \Rightarrow \{\text{heap} := \text{store}(\text{heap},f,i)\}[\texttt{return i;} \Vdash^{\alpha_{pst} \emptyset \alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{\texttt{i} < 0 \Rightarrow \{\text{heap} := \text{store}(\text{heap},f,i)\}[\texttt{skip;return i;} \Vdash^{\alpha_{pst} \emptyset \alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{\texttt{i} < 0, \{\text{heap} := \text{store}(\text{heap},f,i)\}\texttt{i} \leq 0 \Rightarrow \{\text{heap} := \text{store}(\text{heap},f,i)\}[\texttt{skip;return i;} \Vdash^{\alpha_{pst} \emptyset \alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{(*)}$$

$$\dfrac{\dfrac{}{\text{false} \Rightarrow \{\text{heap} := \text{store}(\text{heap},f,i)\}[\ldots \Vdash^{\alpha_{pst} \emptyset \alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{\texttt{i} < 0, \{\text{heap} := \text{store}(\text{heap},f,i)\}\texttt{i} > 0 \Rightarrow \{\text{heap} := \text{store}(\text{heap},f,i)\}[\ldots \Vdash^{\alpha_{pst} \emptyset \alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{(**)}$$

$$\dfrac{\dfrac{\dfrac{(*) \quad (**)}{\texttt{i} < 0 \Rightarrow \{\text{heap} := \text{store}(\text{heap},f,i)\}[\texttt{if(i>0)\{}\ldots\texttt{\}else skip;return i;} \Vdash^{\alpha_{pst} \emptyset \alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{\texttt{i} < 0 \Rightarrow \{\text{heap} := \text{store}(\text{heap},f,i)\}[\texttt{while(i>0)\{await this.f != i;}_1\texttt{this.c = this.c+1; skip;\}return i;} \Vdash^{\alpha_{pst} \emptyset \alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{\texttt{i} < 0 \Rightarrow [\texttt{this.f = i; while(i>0)\{await this.f != i;}_1\texttt{this.c = this.c+1; skip;\}return i;} \Vdash^{\alpha_{pst} \emptyset \alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}}{\texttt{i} < 0 \Rightarrow [\texttt{this.f = i; while(i>0)\{await this.f != i;}_1\texttt{this.c = this.c+1; skip;\}return i;} \Vdash^{\alpha_{fr}} (\{f\},\{f\},\{1 \mapsto (\{c\},\{c\})\})]}$$

Figure 5.3: Proving a frame in context.

5.2 Sequential Effect Types

The above system only reasons about reads and writes in general. We now give a more refined system that (1) reasons about the order of read and write accesses on a field, (2) differs between the read in a suspension statement and a read in other statements and (3) gives an effect type to each field separately. Properties we aim to specify are, e.g., that a certain field is not written after being used in in a guard, but may be read before, or that a field may only be read outside of guards.

The system we give generalizes frames, and we introduce it to demonstrate how types that are assigned to variables or fields (in contrast to processes, such as $\mathbb{T}_{|t|}$) are handled in BPL. First we define regular expressions over effects.

Definition 5.2 (Regular Expression over Effects)**.** *The set of regular expressions over sequences of effects is defined by the following grammar*

$$\mathsf{r} ::= \mathsf{RS} \mid \mathsf{RC} \mid \mathsf{RW} \mid \mathsf{W} \mid \mathsf{r}.\mathsf{r} \mid \mathsf{r} + \mathsf{r} \mid \mathsf{r}^* \mid \mathsf{no}$$

The effect RS (read-synchronization) expresses a read in a guard, RW (read-write) a read and a write at the same time, RC (read-copy) any other read and W a write. The semantics of the dot expresses sequential composition, + expresses alternative and the Kleene star expresses finite repetition.

Definition 5.3. *The behavioral specification $\mathbb{T}_{\mathsf{eff}}$ for sequential effects is*

$$\mathbb{T}_{\mathsf{eff}} = (\mathsf{F} \to \mathsf{r}, \alpha_{\mathsf{eff}})$$

We write sequential effect types SE *as* $\{\mathsf{r}_\mathtt{f}\}_{\mathtt{f}\in\mathsf{F}}$ *or* $\{\mathtt{f}_1 \mapsto \mathsf{r}_1, \dots\}$*. The semantics of a regular expression for a field* f *is as follows.*

$$\begin{aligned}
\mathsf{noeff}(j,\mathtt{f}) &= \{\mathtt{f}_\mathsf{R}, \mathtt{f}_\mathsf{W}\} \not\subseteq \mathsf{eff}(j)\\
\alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{RS}) &= \exists i \in \mathbf{I}.\ \big(\forall j \in \mathbf{I}.\ j \neq i \to \mathsf{noeff}(j,\mathtt{f})\big) \wedge \mathtt{f}_\mathsf{R} \in \mathsf{eff}(i) \wedge \exists p \in \mathsf{P}.\ \mathtt{p}_\mathsf{E} \in \mathsf{eff}(i)\\
\alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{RC}) &= \exists i \in \mathbf{I}.\ \big(\forall j \in \mathbf{I}.\ j \neq i \to \mathsf{noeff}(j,\mathtt{f})\big) \wedge \mathtt{f}_\mathsf{R} \in \mathsf{eff}(i) \wedge \mathtt{f}_\mathsf{W} \notin \mathsf{eff}(i) \wedge \neg\exists p \in \mathsf{P}.\ \mathtt{p}_\mathsf{E} \in \mathsf{eff}(i)\\
\alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{RW}) &= \exists i \in \mathbf{I}.\ \big(\forall j \in \mathbf{I}.\ j \neq i \to \mathsf{noeff}(j,\mathtt{f})\big) \wedge \mathtt{f}_\mathsf{R} \in \mathsf{eff}(i) \wedge \mathtt{f}_\mathsf{W} \in \mathsf{eff}(i)\\
\alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{W}) &= \exists i \in \mathbf{I}.\ \big(\forall j \in \mathbf{I}.\ j \neq i \to \mathsf{noeff}(j,\mathtt{f})\big) \wedge \mathtt{f}_\mathsf{W} \in \mathsf{eff}(i)\\
\alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r}_1.\mathsf{r}_2) &= \exists i \in \mathbf{I}.\ \alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r}_1)[j \in \mathbf{I} \setminus j \leq i] \wedge \alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r}_2)[j \in \mathbf{I} \setminus j > i]\\
\alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r}_1 + \mathsf{r}_2) &= \alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r}_1) \vee \alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r}_2)\\
\alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r}^*) &= \exists X \subseteq \mathbf{I}.\ \big(\exists i,j \in \mathbf{I}.\ i \in X \wedge j \in X \wedge \forall k \in \mathbf{I}.\ i \leq k \leq j\big) \wedge\\
&\qquad \forall i,j \in X.\ \big(i < j \wedge \neg\exists m \in X.\ i \leq m \leq j\big) \to \alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r})[l \in I \setminus i \leq l \leq j]\\
\alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{no}) &= \forall i \in \mathbf{I}.\ \mathsf{noeff}(i,\mathtt{f})
\end{aligned}$$

The semantics of a type $\{\mathsf{r}_\mathtt{f}\}_{\mathtt{f}\in\mathsf{F}}$ *is that every field* f *follows* $\mathsf{r}_\mathtt{r}$*.*

$$\alpha_{\mathsf{eff}}(\{\mathsf{r}_\mathtt{f}\}_{\mathtt{f}\in\mathsf{F}}) = \bigwedge_{\mathtt{f}\in\mathsf{F}} \alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{r}_\mathtt{f})[i \in \mathbf{I} \setminus \mathsf{isEvent}(i)]$$

The semantics $\alpha^{\mathtt{f}}_{\mathsf{eff}}$ works on a trace that only consists of events and follows the translation of regular expressions to MSO [20]. For the simple types (RS, RC, RW, W) the semantics expresses that only one event has an effect for the field ($\forall j \in \mathbf{I}.\ j \neq i \to \mathsf{noeff}(j,\mathtt{f})$) and that the effect set at this event has the correct effects (e.g., only a read for RC).

For the sequential composition $r_1.r_2$, the semantics picks one position i, such that the semantics of r_1 holds up to i and the semantics of r_2 after i. Relativization restricts all quantifiers in the semantics of the subexpression to the traces before (or after) i. Similarly, the semantics of the Kleene star picks a set of positions such that the translation of the subexpression holds between each pair of subsequent elements of this set. Additionally, the set contains the first and last element of the trace. Alternative is translated into a disjunction and no expresses that there are no effects of the given field. The semantics of the whole type relativizes the semantics of the regular expressions for a field to the subtrace of events which have an effects of this field. We stress that relativization is a syntactic operation and introduces no new constructs into the logic.

Example 5.2. *Consider* $\mathsf{SE}_{a,b} = \{a \mapsto \mathsf{RC.W} + \mathsf{RW}, b \mapsto (\mathsf{RS})^*.\mathsf{RC}\}$. *The type for* a *specifies that the field is either first read and then written, or read and written in one step. The type for* b *specifies finitely many synchronizations and then one read. We require that all operations are specified, the following formulas are valid:*[1]

$$[\texttt{this.a = this.a + this.b;} \overset{\alpha_{\mathsf{eff}}}{\Vdash} \mathsf{SE}_{a,b}]$$

$$[\texttt{Int v = this.a; await this.b > 0; v = v - this.b; this.a = v;} \overset{\alpha_{\mathsf{eff}}}{\Vdash} \mathsf{SE}_{a,b}]$$

The following formulas are not valid. The first example does not access a *despite being specified to do so and the second example does not read from* b.

$$[\texttt{this.c = this.b;} \overset{\alpha_{\mathsf{eff}}}{\Vdash} \mathsf{SE}_{a,b}] \quad \textit{does not follow the type of } \texttt{a}$$

$$[\texttt{Int v = this.a; await this.b > v; this.a = v+c;skip;} \overset{\alpha_{\mathsf{eff}}}{\Vdash} \mathsf{SE}_{a,b}] \quad \textit{does not follow the type of } \texttt{b}$$

As mentioned above, sequential effects generalize static frames. A static frame is a frame where each program point is mapped to the same pair of read and write sets. Such a frame $(\mathfrak{R}, \mathfrak{W}, \mathfrak{S})$, with $\mathfrak{S}(p) = (\mathfrak{R}, \mathfrak{W})$ for all p that occur in the method in question, can be encoded by allowing each field in $\mathfrak{R}$ to be read, but not written, each field in $\mathfrak{W}$ to be written and each field in both sets to be written and read. Only the fields in both sets are allowed to have the effect RW.

Lemma 5.2. *Let* $\mathfrak{R}$ *and* $\mathfrak{W}$ *be two sets of fields of some class and let* $\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}$ *be defined as follows:*

$$\begin{aligned}\mathsf{SE}_{\mathfrak{R},\mathfrak{W}} = \quad & \{f \mapsto (\mathsf{RS} + \mathsf{RC})^* \mid f \in \mathfrak{R}, f \notin \mathfrak{W}\} \cup \{f \mapsto (\mathsf{RS} + \mathsf{RC} + \mathsf{RW} + \mathsf{W})^* \mid f \in \mathfrak{R}, f \in \mathfrak{W}\} \\ & \cup \{f \mapsto (\mathsf{W})^* \mid f \notin \mathfrak{R}, f \in \mathfrak{W}\} \cup \{f \mapsto \mathsf{no} \mid f \notin \mathfrak{R}, f \notin \mathfrak{W}\}\end{aligned}$$

The sequential effect type $\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}$ *captures the semantics of the frame type* $(\mathfrak{R}, \mathfrak{W})$*:*

$$\alpha_{\mathsf{fr}}\big((\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\big) \equiv \alpha_{\mathsf{eff}}\big(\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}\big)$$

We conjecture that if the control-flow graph of the method is know, then a dynamic frame can also be synthesized into a sequential effect type, but refrain from proving this formally as it offers no insights into BPL.

Before we give the behavioral type itself, we require some auxiliary functions. The functions $\mathsf{app}(\mathsf{SE}, \mathsf{locs}, r)$ updates a type SE by removing the prefix r from the types of all fields within locs. If one of these types does not start with r, the update fails.

$$\mathsf{app}(\mathsf{SE}, \mathsf{locs}, r)(f) = \begin{cases} \mathsf{SE}(f) & \text{if } f \notin \mathsf{locs} \\ r' & \text{if } f \in \mathsf{locs} \wedge \mathsf{SE}(f) = r.r' \\ \mathsf{no} & \text{if } f \in \mathsf{locs} \wedge \mathsf{SE}(f) = r \\ \textit{undefined} & \text{otherwise} \end{cases}$$

[1] For appropriately defined sets of programs.

If the update fails, we write $\mathsf{app}(\mathsf{SE}, \mathsf{locs}, \mathsf{r})(\mathtt{f}) = \bot$. If $\mathsf{SE}(\mathtt{f}) = \mathsf{no}$ for all fields $\mathtt{f}$, then we write $\mathsf{SE} = \mathsf{no}$.

The function $\mathsf{inner}(\mathsf{SE})$ and $\mathsf{cont}(\mathsf{SE})$ take as input a type where each field has a type that allows us to execute a repetition first. If the input has the form $\mathsf{r}_1^*.\mathsf{r}_2$, then the inner function returns r_1 and cont returns r_2. This is used to check the method-body against the repeated type and the continuation against the part of type afterwards. In this pattern r_2 must also describe the evaluation of the guard after the last iteration.

$$\mathsf{inner}(\mathsf{SE})(\mathtt{f}) = \begin{cases} \mathsf{r}_1 & \text{if } \mathsf{SE}(\mathtt{f}) = \mathsf{r}_1^*.\mathsf{r}_2 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$$\mathsf{cont}(\mathsf{SE})(\mathtt{f}) = \begin{cases} \mathsf{r}_2 & \text{if } \mathsf{SE}(\mathtt{f}) = \mathsf{r}_1^*.\mathsf{r}_2 \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Again, these functions are partial and may be undefined. In this case a rule using them is not applicable.

Definition 5.4. *The behavioral type $\mathbb{T}_{\mathsf{eff}}$ extends the behavioral specification with any obligation schema and the calculus shown in Fig. 5.4.*

$$\mathbb{T}_{\mathsf{eff}} = (\mathsf{F} \to \mathsf{r}, \alpha_{\mathsf{eff}}, \iota_{\mathsf{eff}}, \pi_{\mathsf{eff}}) \textit{ with } \iota_{\mathsf{eff}}(\mathtt{m}) = \left(\mathbf{true}, \{\mathsf{r}_{\mathtt{f}}\}_{\mathtt{f} \in \mathsf{F}}\right)$$

The rules **(assignV)** and **(readV)** check that all read fields in the expression have RC as their first element in their regular expression. In the premise only the remainder of the regular expression has to be followed. Other fields have the same type. There are two rules for field assignment. In any case all read fields, except the written one, are checked to have a type that starts with RC. The rule **(assignF2)** is applied if the written field is also read. Its type must then start with RW. Otherwise, rule **(assignF1)** is applied and the type must start with W. Rule **(callV)** collects the fields of all expressions: caller and all parameters. Their type must start with RC. Rules **(awaitF)** and **(awaitB)** are analogous to **(assignV)** and **(readV)**, but check for RS.

The **(if)** rule checks that all fields in the guard have a type starting with RC and check both branches for the remainder. The loop rule **(while)** demands that the type of all fields has the form $\mathsf{r}_1^*.\mathsf{r}_2$ after evaluating the guards and checks the rule body against r_1 and the continuation against r_2. The other rules check that once the execution has finished, all fields have the type no, i.e., there is no access left over.

The rewrite rules allow one to append no to either side, choose a type when an alternative is available, unroll a repetition or wrap no into a repetition. The rewrite rules are needed to handle alternative and to ensure that the loop rule is always applicable, as any type can be rewritten into the correct form if the field is not accessed in the loop:

$$\mathsf{r} \rightsquigarrow \mathsf{no}.\mathsf{r} \rightsquigarrow \mathsf{no}^*.\mathsf{r}$$

Note that rewrite rules that are only applicable in one direction, as the ones for alternative, may lead to situations where backtracking is possible, because the wrong rewrite rule has been applied and cannot be undone by another rewrite rule.

Lemma 5.3. *Type $\mathbb{T}_{\mathsf{eff}}$ is sound, all rules are basic-local and none of the rules is enabling but not sound.*

Example 5.3. *Fig. 5.5 gives proof trees for some formulas in Ex. 5.2. In the first proof tree we apply one rule application per rewrite, in the others we summarize them. In all of them, the steps marked with $(*)$ are due to rewrites.*

The first proof cannot be closed, because **this**`.b` *is not read at the end. The type contains a repetition, but the program does not and the system can handle this via unrolling. Note that there is a second incomplete proof for the formula, as there is a choice when applying the rule for $+$. The other proof, however, also fails. The second proof can be closed. The two proofs in Fig. 5.6 show the matching of the Kleene star on loops. The second one illustrates the matching of the Kleene star on loops if the field is read in the guard.*

$$(\mathbb{T}_{\text{eff}}\text{-assignV})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RC})\right]}{\Rightarrow \left[\mathtt{v = e;\ s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]} \qquad (\mathbb{T}_{\text{eff}}\text{-readV})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RS})\right]}{\Rightarrow \left[\mathtt{v = e.}\textbf{get}\mathtt{;\ s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$(\mathbb{T}_{\text{eff}}\text{-assignF1})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RC}), \{\mathtt{f}\}, \mathsf{W})\right]}{\Rightarrow \left[\textbf{this}\mathtt{.f = e;\ s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$(\mathbb{T}_{\text{eff}}\text{-assignF2})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}) \setminus \{\mathtt{f}\}, \mathsf{RC}), \{\mathtt{f}\}, \mathsf{RW})\right]}{\Rightarrow \left[\textbf{this}\mathtt{.f = e;\ s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$(\mathbb{T}_{\text{eff}}\text{-awaitF})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RS})\right]}{\Rightarrow \left[\textbf{await}\ \mathtt{e?;\ s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]} \qquad (\mathbb{T}_{\text{eff}}\text{-awaitB})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RS})\right]}{\Rightarrow \left[\textbf{await}\ \mathtt{e;\ s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$(\mathbb{T}_{\text{eff}}\text{-callV})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{f}) \cup \bigcup_{i \le n} \mathsf{locs}(\mathtt{e}i), \mathsf{RC})\right]}{\Rightarrow \left[\mathtt{v = f!m(e1,...,en);\ s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$(\textbf{if})\ \frac{\Rightarrow \left[\mathtt{s;s''} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RC})\right] \qquad \Rightarrow \left[\mathtt{s';s''} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RC})\right]}{\Rightarrow \left[\textbf{if}\mathtt{(e)\{s\}}\textbf{else}\mathtt{\{s'\}s''} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$(\mathbb{T}_{\text{eff}}\text{-skip})\ \frac{}{\Rightarrow \left[\textbf{skip} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{no}\right]} \qquad (\mathbb{T}_{\text{eff}}\text{-skipCont})\ \frac{\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}{\Rightarrow \left[\textbf{skip}\mathtt{;\ s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$(\mathbb{T}_{\text{eff}}\text{-return})\ \frac{\mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RC}) = \mathsf{no}}{\Rightarrow \left[\textbf{return}\ \mathtt{e} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$(\mathbb{T}_{\text{eff}}\text{-while})\ \frac{\begin{array}{c}\Rightarrow \left[\mathtt{s} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{inner}(\mathsf{SE}), \mathsf{locs}(\mathtt{e}), \mathsf{RC})\right] \\ \Rightarrow \left[\mathtt{s'} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{app}(\mathsf{cont}(\mathsf{SE}), \mathsf{locs}(\mathtt{e}), \mathsf{RC})\right]\end{array}}{\Rightarrow \left[\textbf{while}\mathtt{(e)\{s\};s'} \overset{\alpha_{\text{eff}}}{\Vdash} \mathsf{SE}\right]}$$

$$\begin{array}{rl} \mathsf{r} & \rightsquigarrow \mathsf{r.no} \\ \mathsf{r} & \leftrightsquigarrow \mathsf{r} + \mathsf{r} \\ \mathsf{no.r} & \rightsquigarrow \mathsf{r} \\ \mathsf{r}_1 + \mathsf{r}_2 & \rightsquigarrow \mathsf{r}_1 \\ \mathsf{r}_1 + \mathsf{r}_2 & \rightsquigarrow \mathsf{r}_2 \\ \mathsf{r}^* & \rightsquigarrow \mathsf{r.r}^* + \mathsf{no} \\ \mathsf{no} & \rightsquigarrow \mathsf{no}^* \end{array}$$

Figure 5.4: Calculus π_{eff} for the sequential effect type $\mathbb{T}_{\text{eff}}$.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\Rightarrow [\textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto no, b \mapsto RC\}]}
}{(*)\ \Rightarrow [\textbf{this}.a\ =\ v{+}c; \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto RC\}]}
}{(*)\ \Rightarrow [\textbf{this}.a\ =\ v{+}c; \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto no.RC\}]}
}{(*)\ \Rightarrow [\textbf{this}.a\ =\ v{+}c; \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto (RS.(RS)^* + no).RC\}]}
}{\Rightarrow [\textbf{this}.a\ =\ v{+}c; \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto (RS)^*.RC\}]}
}{(*)\ \Rightarrow [\textbf{await this}.b\ >\ a;\ \textbf{this}.a\ =\ v{+}c;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto RS.(RS)^*.RC\}]}
}{(*)\ \Rightarrow [\textbf{await this}.b\ >\ a;\ \textbf{this}.a\ =\ v{+}c;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto (RS.(RS)^* + no).RC\}]}
}{\Rightarrow [\textbf{await this}.b\ >\ a;\ \textbf{this}.a\ =\ v{+}c;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto (RS)^*.RC\}]}
}{(*)\ \Rightarrow [\texttt{Int}\ v\ =\ \textbf{this}.a;\ \textbf{await this}.b\ >\ v;\ \textbf{this}.a\ =\ v{+}c;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto RC.W, b \mapsto (RS)^*.RC\}]}
}{\Rightarrow [\texttt{Int}\ v\ =\ \textbf{this}.a;\ \textbf{await this}.b\ >\ v;\ \textbf{this}.a\ =\ v{+}c;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto RC.W + RW, b \mapsto (RS)^*.RC\}]}
$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\Rightarrow [\textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto no, b \mapsto no\}]}
}{\Rightarrow [\textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto no\}]}
}{(*)\ \Rightarrow [v\ =\ v\ -\ \textbf{this}.b;\ \textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto RC\}]}
}{\Rightarrow [v\ =\ v\ -\ \textbf{this}.b;\ \textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto (RS)^*.RC\}]}
}{(*)\ \Rightarrow [\textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b;\ \textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto RS.(RS)^*.RC\}]}
}{\Rightarrow [\textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b;\ \textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto (RS)^*.RC\}]}
}{(*)\ \Rightarrow [\texttt{Int}\ v\ =\ \textbf{this}.a;\ \textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b;\ \textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto RC.W, b \mapsto (RS)^*.RC\}]}
}{\Rightarrow [\texttt{Int}\ v\ =\ \textbf{this}.a;\ \textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b;\ \textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto RC.W + RW, b \mapsto (RS)^*.RC\}]}
$$

Figure 5.5: Proof Trees for Sequential Effects.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\Rightarrow [\textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto no, b \mapsto no\}]}
}{\Rightarrow [v\ =\ v\ -\ \textbf{this}.b; \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto no, b \mapsto RC\}]}
}{\Rightarrow [\textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b; \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto no, b \mapsto RS.RC\}]}
\qquad
\dfrac{
\dfrac{}{\Rightarrow [\textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto no, b \mapsto no\}]}
}{\Rightarrow [\textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto no\}]}
}{(*)\ \Rightarrow [\textbf{while}(\ v\ >\ 0\ )\{\textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b;\ \textbf{skip};\}\textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto no^*.W, b \mapsto (RS.RC)^*.no\}]}
}{\Rightarrow [\textbf{while}(\ v\ >\ 0\ )\{\textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b;\ \textbf{skip};\}\textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto (RS.RC)^*.no\}]}
$$

$$
\dfrac{
\dfrac{
\vdots
}{\Rightarrow [\textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b; \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto no, b \mapsto RS.RC\}]}
\qquad
\dfrac{
\vdots
}{\Rightarrow [\textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto W, b \mapsto no\}]}
}{\Rightarrow [\textbf{while}(\ \textbf{this}.a\ >\ 0\ )\{\textbf{await this}.b\ >\ 0;\ v\ =\ v\ -\ \textbf{this}.b;\ \textbf{skip};\}\textbf{this}.a\ =\ v;\ \textbf{skip};\ \overset{\alpha_{\text{eff}}}{\Vdash} \{a \mapsto RC^*.RC.W, b \mapsto (RS.RC)^*.no\}]}
$$

Figure 5.6: Proof Trees for Sequential Effects with Loops.

Finally, Consider the following sequent:

$\Rightarrow [\texttt{while(this.a > 0){await this.b > 0; v = v - this.b; skip;}this.a = v; skip;}\overset{\alpha_{\text{eff}}}{\Vdash} \{\mathsf{a} \mapsto (\mathsf{RC}^*).\mathsf{W}, \mathsf{b} \mapsto (\mathsf{RS}.\mathsf{RC})^*.\mathsf{no}\}]$

The rule for loops is not applicable, because the below update is not defined for a.

$$\begin{aligned}&\mathsf{app}(\mathsf{cont}(\{\mathsf{a} \mapsto (\mathsf{RC}^*).\mathsf{W}, \mathsf{b} \mapsto (\mathsf{RS}.\mathsf{RC})^*.\mathsf{no}\}), \{\mathsf{a}\}, \mathsf{RC})\\ =&\mathsf{app}(\{\mathsf{a} \mapsto \mathsf{W}, \mathsf{b} \mapsto \mathsf{no}\}), \{\mathsf{a}\}, \mathsf{RC}) = \bot\end{aligned}$$

5.3 Discussion

We introduced effect types only as far as needed for our specifications and to demonstrate how types assigned to other elements than whole methods can be handled. In the next chapter, we use the execution effect on program points for method contracts to connect effects and state specification.

On Footprints and Location Sets.

JavaDL dynamic frames [62] are dynamic in another sense than the one of $\mathbb{T}_{\mathsf{fr}}$: the *assignable* clause is not a sets of fields (or locations), but an expression that may either be a location set (similar to $\mathbb{T}_{\mathsf{fr}}$) or a program variable of `LocSet` type. The value of this program variable can change during the execution of the method.

Both types we give are defined over sets of fields, not sets of locations in the sense of JavaDL. In JavaDL, a location is a pair of an object and a composed access. A composed access is either a field, a program variable or an array access expression, or a mixture of these (e.g., a field access in the index of an array program variable). We do not need to include the object of an accessed field, because an object may only access its own fields in Active Objects. Similarly, we do not include composed accesses as CAO does not include arrays. We conjuncture that the generalization to location sets is straightforward, but requires to move the side-conditions into their own premises in the rules and to include the theory of location sets in BPL.

On Flow and Scope.

Without introducing further events, one may use our framework to track the flow of values within a method. To do so, we use MSO to define a path on events that models a flow. I.e., an edge from i to j in this path models that a value written in i is read in j. Such an edge can be defined with the following predicate:

$$\mathsf{flow-edge}(\mathsf{v}, i, j) = \mathsf{v}_{\mathsf{W}} \in \mathsf{eff}(i) \wedge \mathsf{v}_{\mathsf{R}} \in \mathsf{eff}(j) \wedge \forall k \in \mathbf{I}.\ i < k < j \rightarrow \mathsf{v}_{\mathsf{W}} \notin \mathsf{eff}(k)$$

Using standard MSO encoding of paths we can now define a flow from `source` to `sink` over some set of intermediate variables using $\mathsf{flow-edge}$ as the edge relation. This allows to model analyses that require that from some set of fields or variables no value is written into some other set of fields or variables, e.g., tainted analysis [102] (where a special `untaint` function and effect could be introduced), location types [124] (to ensure that at a given point, the called object is "Far") or general dependencies [57]. Full behavioral types for such systems are left for future work.

A generic type-and-effect system presented by Marino and Millstein [102] requires effects over regions (object creation sites, which are program-points identifiers) and scopes (i.e., an effect that models entering a scope and leaving it) to cover a number of applications, including checked exceptions. In CAO, scopes issue no effects and all object creation sites are not tracked, we are thus not able to express regions without extending effects. However, we propose that the introduction of new effects is only a minor change to the LAGC semantics and can be done by extending the effect set in the LA semantics for each statement. This chapter shows that BPL can handle side-effects in general, we suggest that the exception handling of JavaDL can be reformulated to be a BPL-behavioral type.

On Repetition.

The sequential effect type uses the Kleene star to model repetition. In the later chapters of this work, we use similar constructs to model repetition. An alternative would be to use recursion. We consider this mainly a design choice: specifications using Kleene star-like constructs can be mapped to loops in the program, while constructs for recursion can be mapped to recursive method calls. We deem loops to be more elementary than synchronous method calls, as it allows us to reason about repetition locally, i.e., within one method, without introducing synchronous method calls or global reasoning (when using asynchronous recursion).

On Aliasing.

The strong encapsulation of Active Objects allows us to have rather simple frames — it suffices to specify the accessible fields of the object in question, because contrary to, e.g., Java, the fields of the objects referenced from the object in question itself cannot be accessed. In models where such accesses are possible, aliasing is a (bigger) problem. Consider that a class has a field `this.o` and one can access the field `f` of the referenced object. I.e., `this.o.f` is a valid expression. Now, even if one verifies that `this.o.f` can be read but not written, the reference to `this.o` can be shared and, thus, `this.o.f` can still change its value if accessed twice in a method without suspension. Even without the influence of other objects aliasing poses a problem: when updating `this.o2.f` one has to show that `this.o` and `this.o2` are referencing different objects.

In Active Objects, aliasing still occurs and needs to be handled. It does however not occur on the level of heap access, but at the level of communication: If an object x_1 follows a certain protocol (i.e., a certain order of method calls and future reads) with another object x_2, it may be the case that another object x_3 also attempts to communicate with x_2 and so interferes with the protocol between x_1 and x_2.

Memory Access vs. Footprint

JavaDL verifies the footprint by showing that a method has the same runs in all heaps which differ only in fields *outside* the footprint. I.e., a method may read from a field, but its result may not depend on it. We verify a stronger property: no read access occurs at all. This is not merely a question of taste: from the view of the memory system the JavaDL formalization does not give any additional guarantees, while our system ensures that parts of the memory need not be made available to this method.

6 Cooperative Method Contracts

In this chapter we introduce *Cooperative Method Contracts*, a generalization of method contracts to Active Objects. A method contract specifies the context which a method assumes as its precondition and the context it establishes as its postcondition. It is more fine-grained than object invariants, because the precondition only has to hold at the moment the method starts execution, not at any point.

It is not possible, or at least not useful, to adopt method contracts for synchronous method calls [103] to the Active Object setting.

Challenges.

Originally [103], method contracts were introduced as pairs of pre- and postcondition for sequential programs with synchronous method calls. The precondition specifies the prestate of the method and describes the method parameters and the heap. The postcondition specifies the poststate of the method and describes the result value and the heap. Their verification requires that at the point of the method call the precondition has to be proven by the caller and afterwards only the postcondition may be assumed.

For Active Objects, several additional challenges require a more elaborate specification:

Call Time Gap As the method calls are asynchronous, there is delay between the execution of the call statements and the start of the execution of the process. In this time, the callee may execute other processes and may modify its heap — even if the precondition holds at the moment of the call, it may not hold when the process starts.

Strong Encapsulation Due to the strong encapsulation, the caller object can neither write nor read the heap of the callee object. It can, thus, neither enforce nor check the precondition directly, if the precondition specifies a property of the heap of the callee object.

Return Time Gap The postcondition describes the heap of the terminating process and the result value. However, the result value is accessed through a future, which may be passed around before being accessed. When a future is accessed, only the result value is read — it may be the case that the resolving method is not known.

Interleaving Beyond the specification of the pre- and poststate, it is necessary to specify the state before and after suspension.

Approach.

Cooperative Method Contracts refine method contracts. The precondition is split into the *heap precondition*, which specifies only the heap of the prestate and the *parameter precondition*, which specifies only the parameters of the call. The caller only establishes the parameter precondition — the caller has access to the parameters and call parameters cannot be modified. The heap precondition must be established by the last active process in the *callee* object.

To specify the methods that may run before the start of a process, we use *context sets*. Each precondition has two context sets. The first is the *succeeding* context set, which is the set of methods which must have run before. The second it the *overlapping* context set, the set of methods which may run between start of execution, but after one method from the succeeding context sets has run. The methods in the succeeding context set must establish the heap precondition, the methods in the overlapping context set must preserve it. This does not prohibit these methods from *modifying* the heap, but when they modify it, they must ensure that the heap precondition in question still holds. No field is locked.

To ensure this property of context sets, it is required that all method contracts are *coherent*: If a method $\mathtt{m}_1$ is in the succeeding context set of a method $\mathtt{m}_2$, then the postcondition of $\mathtt{m}_1$ must imply the heap precondition of $\mathtt{m}_2$. Similarly, the preservation property for methods in the overlapping context set is required. If the set of method contracts is not coherent, the methods contracts are *enriched* such that the new method contracts are (1) coherent and (2) imply the old method contracts.

Each synchronization is specified by a *resolving contract*, a set of methods. The future read at this point must be resolved by one of these methods and we can assume that one of their postcondition holds.

The above three concepts, split precondition, context sets and resolving contracts, deal with the challenges of the call time gap, strong encapsulation and the return time gap. However, method start and method end are not the only points where the process loses or gains control over the method: this also happens at suspension points. Suspension points must also be specified. To do so, we make two changes to the framework.

First, we annotate each suspension point which a *suspension contract*. A suspension contracts consists of (1) a suspension condition, a specification of the heap that must hold before suspension (2) a suspension assumption, a specification of the heap that must hold after suspension and (3) two context sets analogous to method contracts, which specify the methods responsible for the suspension assumption. One can see the precondition of the method as the suspension assumption of the implicit first suspension point marked by the method start and the postcondition as the suspension condition of the **return** statement.

Secondly, all context sets are not defined over method names, but over method names *and program-point identifiers*. This allows to specify which suspension condition or postcondition exactly has to establish the suspension assumption or heap precondition.

6.1 Syntax, Extraction and Coherence

This section extends CAO with the syntax to specify cooperative method contracts and dynamic frames, as presented in section 5.1. We also describes how to extract the method contract from the syntax and establish the coherence of the method contracts.

The **old** and **last** program variables in are section 3.2 and are used to express that a property is preserved. E.g., to express that a method (without suspensions) preserves **this**.i > 0, we use the postcondition **last**.i > 0 → **this**.i > 0.

This is *not* possible by encoding **this**.i > 0 as the pre- and postcondition, because that expresses that φ *must* hold at method start. If the method would contain suspension points, then the formula **last**.i > 0 → **this**.i > 0 as the postcondition would express preservation since the last suspension, while **old**.i > 0 → **this**.i > 0 would express preservation since the method start. For the preservation needed for context sets, **last** suffices, but **old** increases the expressive power of our postconditions.

Definition 6.1 (Syntax with Method Contracts)**.** *Let* id *range over method names and program point identifiers,* φ *over lFOS formulas, and* m *over method names. The definition for the syntax with method contracts, ghost fields and assignments is given in Fig. 6.1. The grammar extends the one in Def. 3.1 by adding contracts* Spec *before method signatures and suspension points and resolving contracts* RSpec *before synchronization points.*

We say that requires and ensures are context clauses*, succeeds and overlaps are* context sets*, and assignable and accessible are* frame sets*.*

We demand the following restrictions:

- All formulas are sentences.
- The *assignable* and *accessible* clauses are only used when specifying a class. The *all—assignable* and *all—accessible* clauses are only used for the specification of method signatures in a class.

$$\text{MSig} ::= \text{Spec?}\ \texttt{D}\ \texttt{m}(\overrightarrow{\texttt{D}\ \texttt{v}}) \qquad \texttt{s} ::= \ldots \mid \text{Spec?}\ \textbf{await}\ \texttt{g}_\texttt{p} \mid \text{RSpec}\ \texttt{v} = \texttt{e}.\textbf{get}_\texttt{p}$$

$$\text{Spec} ::= /*@\ \text{Require?}\ \text{Ensure?}\ \text{Runs?}\ \text{Acc?}\ \text{Asg?}\ @*/ \quad \text{RSpec} ::= /*@\ \textit{resolvedBy}\ \{\overrightarrow{\texttt{m}}\}\ @*/$$
$$\text{Asg} ::= \textit{assignable}\ \{\overrightarrow{\texttt{f}}\};\ [\textit{all—assignable}\ \{\overrightarrow{\texttt{f}}\};] \qquad \text{Acc} ::= \textit{accessible}\ \{\overrightarrow{\texttt{f}}\};\ [\textit{all—accessible}\ \{\overrightarrow{\texttt{f}}\};]$$
$$\text{Runs} ::= \textit{succeeds}\ \{\overrightarrow{\texttt{id}}\};\ \ \textit{overlaps}\ \{\overrightarrow{\texttt{id}}\}; \qquad \text{Require} ::= \textit{requires}\ \varphi; \quad \text{Ensure} ::= \textit{ensures}\ \varphi;$$

Figure 6.1: Extended Syntax with Method Contracts

- All formulas within a class contain no fields from other classes.
- All formulas in the *requires* clauses of methods in interfaces may contain the method parameters, but no fields.
- All formulas in the *requires* clauses of methods in classes may contain fields, but no method parameters.
- The variable `result` is only used in *ensures* clauses of methods, not of suspension points.
- The formulas in the specification of the suspension points may contain field names, but no variables or method parameters.
- The heap variables **old** and **last** are only used in the *ensures* clauses of methods in classes.

The *all—assignable* and *all—accessible* clauses are only a short-hand notation: *all—assignable*$\{\overrightarrow{\texttt{f}}\}$ adds the fields $\{\overrightarrow{\texttt{f}}\}$ to the *assignable* clauses of all specifications (method contract and suspension contracts) of the specified method.

Inheritance.

We support inheritance of specifications, which is important in CAO because, as classes are no types, only the interface is known to the caller of method and not the class. We allow an interface to *redeclare* a method signature declared in a superinterfaces with additional specification. I.e., if a clause is given, it modifies (replaces or extends) the superspecification. If a clause is not given, the specification of the superinterfaces are used unchanged. We follow the behavioral subtyping principle [100]: the specification of the subinterface must *refine* the specification of the superinterface and all objects implementing the subinterface must thus adhere to the specification of the superinterface. This means that the precondition of the subinterface must imply the precondition of the superinterface and the postcondition of the superinterface must imply the postcondition of the subinterface.

For classes, the heap precondition is not relevant to the parameter precondition of the interfaces. The postcondition $\chi_\texttt{m}$ of a class cannot imply the postconditions from the interfaces, as it contains not only the `result` variable, but also the fields. Instead the following formula must imply all postconditions from the interfaces. Let $\texttt{f}_1, \ldots, \texttt{f}_n$ be all fields of the class in question, with types $T_1, \ldots, T_n$.

$$\exists v_1 \in \texttt{D}(\texttt{f}_1).\ \ldots \exists v_n \in \texttt{D}(\texttt{f}_n).\ \chi_\texttt{m}[\textbf{this}.\texttt{f}_1 \setminus v_1] \ldots [\textbf{this}.\texttt{f}_n \setminus v_n]$$

This means that for any values for the heap, the property of the result value specified by the class specification implies the property of the result value specified by the interface specification.

For readability, we refrain from writing down the full specifications in the class. Instead, the preconditions/postconditions of the interfaces are joined with a conjunction to the precondition of the class as the first step when handling the specifications.

For context sets, we use a different mechanism. Intuitively, the context sets of the subinterface should be subsets of the context sets of the superinterfaces – however, the subinterface contains more methods. Thus, the context set given in a subinterface may contain only method *freshly*[1] declared in this interface. Similarly, the class may specify additional, non-exposed methods and program-point identifiers in its context sets. These additional sets of the subinterfaces and class are *added* to the original context sets.

The handling of context sets breaks behavioral subtyping, as the subspecification is more strict than the superspecification. I.e., a caller may not be able to adhere to the specification of the callee, because he does not have access to all the methods needed to do so. This must be handled by the modeler by providing usable interfaces. Similarly for the addition of program-point identifiers in the class: it is the task of the class to ensure that its internal synchronization structure enforces its internal context sets if the callers adhere to the exposed context sets. It is, however, not possible to use behavioral subtyping for this *object-local* specification, as obviously an interface must be able to add methods to its superinterface. However, we do use behavioral subtyping for the *method-local* specification.

In case of multiple implemented or extended interfaces, a method may have multiple contracts. As dealing with multiple contracts is standard and straightforward [63], we restrict our presentation to one contract per method.

Default Contracts.

We do not require the user to annotate all clauses — if a specification clause is not given, we use default values:

- If a precondition or postcondition is not given in a specification, it is `true`.
- If a context set is not given in an interface, it is the set of all methods in this interface.
- If a context set is not given in a class, it is the set of all methods and PPIs in this class.
- The default *assignable* and *accessible* sets are the sets of all fields within the class in question, unless a *all—assignable* and *all—accessible* specification is given.
- Resolving contracts must always be given. The reason is that it is a program-wide specification, while the other clauses are interface-local or class-local. Local clauses can have a modular default value that is the most general value in every program where this class or interface is used. Program-wide clauses have different most general values in different programs. We discuss alternatives for this at the end of the chapter.

Example 6.1. *Consider the top left code in Fig. 6.2. It specifies an interface* `I` *with two methods* `m` *and* `n`*. Method* `m` *requires that it gets a future that can be possibly resolved as parameter* `f` *and a positive number as parameter* `g`*. It must run after* `n` *and returns a positive number. Before the start of* `m` *the last active process must have been executing* `n`*: if* `m` *is called twice,* `n` *must also be called twice and the second call must start* `n` *after the first process of* `m` *terminated.*

Interface `I2` *extends* `I`*. It adds that between the termination of* `n` *and the start of* `m` *the new method* `n2` *may also run. The interface redeclares* `m` *to specify this. The new method* `n2` *and the method* `I.n` *both take two parameters, where the first one is bigger than the second and return a negative value.*

The top right code in Fig. 6.2 specifies the class `C` *that implements* `I2`*. It additionally specifies that the return value of* `m` *is smaller than the value of* `this.i`*, and that* `m` *may read* `this.i`*, but write only in* `this.j`*. In the method body, the suspension contract expresses that on suspension* `this.j > 0` *holds, but upon reactivation the value of* `this.j` *is negative. The value of* `this.i` *stays positive (but may change). The suspension assumption must be established by* `n` *and preserved by* `n2`*. Finally, the read future must have been resolved by* `ext` *or* `n2`*. Method* `ext` *is not part of this class.*

The complete specification, i.e., containing all clauses, of `m` *is shown in the bottom left code in Fig, 6.2. Note that formally this is not valid input, because the class specification may not specify the input parameters.*

[1] I.e., not redeclared

```
interface I {
    /*@ requires f != never
                ∧ g > 0;
     ensures result > 0;
     succeeds {n};
     overlaps {}; @*/
    Int m(Fut<Int> f, Int g);

    /*@ requires a > b;
     ensures result < 0; @*/
    Int n(Int a, int b);
    }

    interface I2 extends I {
    /*@  overlaps {n2}; @*/
    Int m(Fut<Int> f, Int g);

    /*@ requires a > b;
     ensures result < 0; @*/
    Int n2(Int a, int b);

}
```

```
class C implements I2{
    Int i = 0;
    Int j = 0;
    Int n(Int a, int b){ ... }
    Int n2(Int a, int b){ ... }

    /*@ ensures result < this.i;
       assignable {j}; accessible {i};
     @*/
    Int m(Fut<Int> f, Int g){
        this.j = g;
        /*@ ensures this.j > 0 ∧ this.i > 0;
            requires this.j < 0 ∧ this.i > 0;
            succeeds {n}; overlaps {n2};
         @*/
        await f?1;
        /*@ resolvedBy {ext,n2} @*/
        Int k = f.get2;
        Fut<Int> ff = this!n(this.i, k);
        return k;
    }
}
```

```
class C implements I2{
Int i = 0;
Int j = 0;

Int n(Int a, int b){ ... }
Int n2(Int a, int b){ ... }

/*@ requires f != never ∧ g > 0;
   ensures result < this.i ∧ result > 0;
   succeeds {n}; overlaps {n2};
   assignable {j}; accessible {i};
@*/
Int m(Fut<Int> f, Int g){
    this.j = g;
/*@ ensures this.j > 0 ∧ this.i > 0;
   requires this.j < 0 ∧ this.i > 0;
   succeeds {n}; overlaps {n2};
@*/
    await f?1;
/*@ resolvedBy {ext,n2} @*/
    Int k = f.get2;
    Fut<Int> ff = this!n(this.i, k);
    return k;
    }
}
```

```
class C implements I2{
Int i = 0;
Int j = 0;

/*@ ensures this.j > 0 ∧ this.i > 0; @*/
Int n(Int a, int b){ ... }
/*@ ensures last.j > 0 ∧ last.i > 0
        → this.j > 0 ∧ this.i > 0@*/
Int n2(Int a, int b){ ... }

/*@ requires f != never ∧ g > 0;
   ensures result < this.i ∧ result > 0;
   succeeds {n}; overlaps {n2};
   assignable {j}; accessible {i};
@*/
Int m(Fut<Int> f, Int g){
    this.j = g;
/*@ ensures this.j > 0 ∧ this.i > 0;
   requires this.j < 0 ∧ this.i > 0;
   succeeds {n}; overlaps {n2};
@*/
    await f?1;
    ...
    }
}
```

Figure 6.2: Top left: specified interfaces. Top right: specified class. Bottom left: class with complete specification for `m`. Bottom right: generated consistent specification.

Extraction.

Cooperative Method Contracts describe *method-local* behavior (frames, postcondition, precondition, suspension assumptions, suspension guarantees, resolving contracts) and *object-local* behavior (context sets). Inside a method, the information needed to verify the context sets is not available, inside a class, the information is not complete because the object cannot control when it is called from the outside. Thus, Cooperative Method Contracts are handled by *multiple* different reasoning systems, some of which are global:

- Frames are verified by the frame type $\mathbb{T}_{\mathsf{fr}}$ (Def. 5.1).
- Postcondition, suspension assumptions and suspension guarantees by the contract type introduced in the next section. The type system connects this type directly to the points-to type $\mathbb{T}_{\mathsf{p2}}$ (Def. 4.6).
- Context sets are handled globally *outside* BPL. We specify them in gMSOT.

We expect that the frame type and the contract type are composed, nonetheless they are designed separately. The role of the gMSOT specification of context sets is to describe the global context needed to compose the method-local behavior needed to enable validity of the initial proof obligation: The method contract type introduced in the next section is sound and contains enabling but unsound rules. Thus, context sets are verified outside of the logic, instead of introducing concepts in the LA semantics to propagate object-local behavior into the abstract traces. This keeps (1) the logic simple, as it is purely method-local and (2) keeps the semantics clear, as it clearly separates LA and GC levels.

Extraction is the process of transforming a specified program into a formal method contract. Before extraction, behavioral subtyping is checked as described above. The result is a *method contract*. A method contract is a pair of a frame type τ_{fr} and a method specification $\mathscr{M}$.

Definition 6.2 (Method Frame)**.** *Given a specified* CAO *program, the frame type* $(\mathfrak{R}, \mathfrak{W}, \mathfrak{S})$ *for a method* $\mathtt{m}$ *is defined as follows:*

$$\begin{aligned}
\mathfrak{R} &= \{\mathtt{f} \mid \mathtt{f} \textit{ is in the accessible-clause or the all_accessible-clause of the method}\}\\
\mathfrak{W} &= \{\mathtt{f} \mid \mathtt{f} \textit{ is in the assignable-clause or the all_assignable-clause of the method}\}\\
\mathfrak{S}(\mathtt{p}) &= \left(\mathfrak{R}_{\mathtt{p}}, \mathfrak{W}_{\mathtt{p}}\right) \textit{ for each program-point identifier } \mathtt{p} \textit{ within } \mathtt{m}\\
\mathfrak{R}_{\mathtt{p}} &= \{\mathtt{f} \mid \mathtt{f} \textit{ is in the accessible-clause of } \mathtt{p} \textit{ or the all_accessible-clause of the method}\}\\
\mathfrak{W}_{\mathtt{p}} &= \{\mathtt{f} \mid \mathtt{f} \textit{ is in the assignable-clause of } \mathtt{p} \textit{ or the all_assignable-clause of the method}\}
\end{aligned}$$

Definition 6.3 (Method Specification)**.** *A program point specification* $\mathscr{S}$ *is a quadruple of a precondition, a postcondition and two context sets.*

A method specification is triple of program point specification, suspension contracts and resolving contracts. Suspension contracts are a function from program point identifiers to program point specifications and resolving contracts.

The method specification $\mathscr{M}_{\mathtt{m}}$ *is defined as follows:*

$$\begin{aligned}
&\mathscr{M}_{\mathtt{m}} = (\mathscr{S}_{\mathtt{m}}, \mathsf{Susp}_{\mathtt{m}}, \mathsf{Reso}_{\mathtt{m}}) \textit{ with } \mathscr{S}_{\mathtt{m}} = (\varphi_{\mathtt{m}}, \chi_{\mathtt{m}}, \mathsf{succ}_{\mathtt{m}}, \mathsf{over}_{\mathtt{m}})\\
&\quad \textit{where } \varphi_{\mathtt{m}} \textit{ is the conjunction of the parameter preconditions and the heap precondition,}\\
&\qquad \chi_{\mathtt{m}} \textit{ is the postcondition, } \mathsf{succ}_{\mathtt{m}} \textit{ is the succeeds set and } \mathsf{over}_{\mathtt{m}} \textit{ is the overlaps set}
\end{aligned}$$

Resolving and suspension contracts are defined as follows:

$$\begin{aligned}
&\mathsf{Reso}_{\mathtt{m}}(\mathtt{p}) = \{\mathtt{n} \mid \mathtt{n} \textit{ is in the resolvedBy set of } \mathtt{p}\}\\
&\mathsf{Susp}_{\mathtt{m}}(\mathtt{p}) = (\varphi_{\mathtt{p}}, \chi_{\mathtt{p}}, \mathsf{succ}_{\mathtt{p}}, \mathsf{over}_{\mathtt{p}})\\
&\qquad \textit{where } \varphi_{\mathtt{p}} \textit{ is the suspension assumption, } \chi_{\mathtt{p}} \textit{ is the suspension assertion,}\\
&\qquad\quad \mathsf{succ}_{\mathtt{p}} \textit{ is the succeeds set and } \mathsf{over}_{\mathtt{p}} \textit{ is the overlaps set}
\end{aligned}$$

The formula χ_{id} has a slightly different meaning, depending on whether it is in the point specification of the method or of a suspension point: For the method it specifies the condition that has to be guaranteed at the *end* of the method, for a suspension point it specifies the condition at the end of the execution *before* the suspension point.

Example 6.2. *The extracted method specification of the method* `C.m` *in Fig. 6.2 is as follows.*

$$\begin{aligned}\mathscr{S}_{\mathtt{m}} &= \big(\texttt{f!=never} \wedge \texttt{g>0},\ \texttt{result<this.i} \wedge \texttt{result>0},\ \{\texttt{n}\}, \{\texttt{n2}\}\big)\\ \mathsf{Susp}_{\mathtt{m}}(1) &= \big(\texttt{this.j>0} \wedge \texttt{this.i>0},\ \texttt{this.j<0} \wedge \texttt{this.i>0},\ \{\texttt{n}\}, \{\texttt{n2}\}\big)\\ \mathsf{Reso}_{\mathtt{m}}(2) &= \{\texttt{ext}, \texttt{n2}\}\end{aligned}$$

The frame type is $\big(\{\texttt{i}\}, \{\texttt{j}\}, \{1 \mapsto (\{\texttt{i},\texttt{j}\}, \{\texttt{i},\texttt{j}\})\}\big)$.

We express the context sets as gMSOT formulas using the following auxiliary predicates:

- $\mathsf{isInvocREv}(i, o)$ holds iff the ith element of the trace is an invocation reaction event on object o.
- $\mathsf{isFutEv}(i, o, \mathtt{m})$ holds iff the ith element of the trace is a future event of method $\mathtt{m}$ on object o.
- $\mathsf{isTermEv}(i, o, \mathtt{p})$ holds iff the ith element of the trace is a suspension event of PPI $\mathtt{p}$ on object o.

Definition 6.4 (Method Trace Context)**.** *Let* `C` *be a class and* `m` *be a method within* `C`*. Given the point specification* $\mathscr{S}_{\mathtt{m}} = (\varphi, \chi, \mathsf{succ}, \mathsf{over})$*, the method trace context is the following formula.*

$$\begin{aligned}&\forall \mathtt{X} \in \mathsf{C}.\ \forall i \in \mathbf{I}.\ \mathsf{isInvocREv}(i, \mathtt{X}) \rightarrow\\ &\quad \exists j \in \mathbf{I}.\ j < i \wedge \Big((\exists \mathtt{p} \in \mathsf{P}.\ \mathsf{isTerm}(j, \mathtt{X}, \mathtt{p}) \wedge \bigvee_{\substack{\mathtt{p}' \in \mathsf{succ}\\ \mathtt{p}' \text{ is a program-point}}} \mathtt{p} \doteq \mathtt{p}') \vee\\ &\qquad (\exists \mathtt{m} \in \mathsf{M}.\ \mathsf{isFutEv}(j, \mathtt{X}, \mathtt{m}) \wedge \bigvee_{\substack{\mathtt{m}' \in \mathsf{succ}\\ \mathtt{m}' \text{ is a method name}}} \mathtt{m} \doteq \mathtt{m}')\Big) \wedge\\ &\quad \forall k \in \mathbf{I}.\ j < k < i \rightarrow \Big((\exists \mathtt{p} \in \mathsf{P}.\ \mathsf{isTerm}(k, \mathtt{X}, \mathtt{p}) \rightarrow \bigvee_{\substack{\mathtt{p}' \in \mathsf{over}\\ \mathtt{p}' \text{ is a program-point}}} \mathtt{p} \doteq \mathtt{p}') \vee\\ &\qquad (\exists \mathtt{m} \in \mathsf{M}.\ \mathsf{isFutEv}(k, \mathtt{X}, \mathtt{m}) \rightarrow \bigvee_{\substack{\mathtt{m}' \in \mathsf{over}\\ \mathtt{m}' \text{ is a method name}}} \mathtt{m} \doteq \mathtt{m}')\Big)\end{aligned}$$

The method trace context for the point specification $\mathscr{S}_{\mathtt{p}}$ *of a program-point identifier* `p` *is analogous:*

$$\begin{aligned}&\forall \mathtt{X} \in \mathsf{C}.\ \forall i \in \mathbf{I}.\ \mathsf{isSusp}(i, \mathtt{X}) \rightarrow\\ &\quad \exists j \in \mathbf{I}.\ j < i \wedge \Big((\exists \mathtt{p} \in \mathsf{P}.\ \mathsf{isTerm}(j, \mathtt{X}, \mathtt{p}) \wedge \bigvee_{\substack{\mathtt{p}' \in \mathsf{succ}\\ \mathtt{p}' \text{ is a program-point}}} \mathtt{p} \doteq \mathtt{p}') \vee\\ &\qquad (\exists \mathtt{m} \in \mathsf{M}.\ \mathsf{isFutEv}(j, \mathtt{X}, \mathtt{m}) \wedge \bigvee_{\substack{\mathtt{m}' \in \mathsf{succ}\\ \mathtt{m}' \text{ is a method name}}} \mathtt{m} \doteq \mathtt{m}')\Big) \wedge\\ &\quad \forall k \in \mathbf{I}.\ j < k < i \rightarrow \Big((\exists \mathtt{p} \in \mathsf{P}.\ \mathsf{isTerm}(k, \mathtt{X}, \mathtt{p}) \rightarrow \bigvee_{\substack{\mathtt{p}' \in \mathsf{over}\\ \mathtt{p}' \text{ is a program-point}}} \mathtt{p} \doteq \mathtt{p}') \vee\\ &\qquad (\exists \mathtt{m} \in \mathsf{M}.\ \mathsf{isFutEv}(k, \mathtt{X}, \mathtt{m}) \rightarrow \bigvee_{\substack{\mathtt{m}' \in \mathsf{over}\\ \mathtt{m}' \text{ is a method name}}} \mathtt{m} \doteq \mathtt{m}')\Big)\end{aligned}$$

Coherence.

Our method contracts are not independent of their context – when proving that a method adheres to a method contract, it is necessary that the other methods adhere to theirs when interacting with this method. I.e., the parameter precondition is adhered to at the moment of the call and the last process established the heap precondition (and analogously for the suspension assumption). For the behavioral type of object invariants we demand that every method in one class has the same invariant. For cooperative contracts, we demand that the contracts are coherent.

- If a suspension point or method id occurs in the succ set of some program point specification $\mathscr{S}_{\mathsf{id}'}$, then the postcondition/suspension assertion (χ_{id}) of id implies the precondition/suspension assumption($\varphi_{\mathsf{id}'}$) of id'.
- If a suspension point or method id occurs in the over set of some program point specification $\mathscr{S}_{\mathsf{id}'}$, then the postcondition/suspension assertion (χ_{id}) of id implies that id preserves the precondition/-suspension assumption($\varphi_{\mathsf{id}'}$) of id'.

Note that program point specifications specify PPIs `p` as well as methods `m`.

Definition 6.5 (Coherence)**.** *Let $\{\mathscr{S}_{\mathsf{id}_i}\}_{i\in I}$ be the set of all program point specifications within a program* Prgm*. We say that* Prgm *is coherent if for every $\mathscr{S}_{\mathsf{id}_i} = (\varphi_{\mathsf{id}_i}, \chi_{\mathsf{id}_i}, \mathsf{succ}_{\mathsf{id}_i}, \mathsf{over}_{\mathsf{id}_i})$ the following holds:*

$$\forall \mathsf{id}' \in \mathsf{succ}_{\mathsf{id}}.\ \chi_{\mathsf{id}'} \rightarrow \varphi_{\mathsf{id}}$$
$$\forall \mathsf{id}' \in \mathsf{over}_{\mathsf{id}}.\ \chi_{\mathsf{id}'} \rightarrow (\varphi_{\mathsf{id}}[\mathbf{this} \setminus \mathbf{last}] \rightarrow \varphi_{\mathsf{id}})$$

Given an incoherent set of cooperative contracts, one can generate a coherent set of cooperative contracts (that still extend the original contracts) by adding additional conjuncts to the postcondition/-suspension assertion occurring in the context sets. We say that we *propagate* the precondition/suspension assumption.

Lemma 6.1 (Propagation for Method Contracts)**.** *Given an inconsistent set of program point specifications $\{\mathscr{S}_{\mathsf{id}_i}\}_{i\in I}$, a consistent set of program point specifications $\{\mathscr{S}'_{\mathsf{id}_i}\}_{i\in I}$ can be generated such that for all program point specifications within $\{\mathscr{S}'_{\mathsf{id}_i}\}_{i\in I}$.*

- *The context sets $\mathsf{succ}_{\mathsf{id}}$ and $\mathsf{over}_{\mathsf{id}}$ are unchanged.*
- *The precondition/suspension assumptions φ_{id} are unchanged.*
- *The postcondition/suspension assertions χ'_{id} of the generated method specification imply the postcondition/suspension assertions χ_{id} of the original method specification.*

Proof. See p. 179.

Example 6.3. *The bottom right code in Fig. 6.2 shows the specification after propagation. The specification of* `m` *and its suspension point is not changed. Method* `n` *now establishes the suspension assumption and* `n2` *preserves it.*

6.2 Behavioral Specification

Given a method specification, we now describe the *method contract behavioral type* that we use to verify the method-local behavior (except frames).

$$\alpha_{\mathsf{met}}\big((\mathfrak{P},\mathfrak{M},\varphi)\big) = \Big(\exists v \in \mathtt{D}.\ [\mathit{last}-1] \doteq \mathsf{futEv}(\_,\_,\_,v) \to [\mathit{last}] \vdash \varphi[\mathtt{result}\setminus v]\Big) \wedge \Big(\ \forall i \in \mathbf{I}. \tag{6.1}$$

$$\bigwedge_{\mathtt{p}\in\mathbf{dom}\mathfrak{P}} \Big(\Big(\big(\mathtt{p}_{\mathsf{E}} \in \mathsf{eff}(i) \wedge \big(\mathsf{isCondEv}(i) \vee \mathsf{isSuspEv}(i)\big)\big) \to [i+1] \vdash \mathsf{pre}(\mathfrak{P}(\mathtt{p}))\Big) \wedge \tag{6.2}$$

$$\Big(\big(\mathtt{p}_{\mathsf{E}} \in \mathsf{eff}(i) \wedge \big(\mathsf{isCondREv}(i) \vee \mathsf{isSuspREv}(i)\big)\big) \to [i+1] \vdash \mathsf{post}(\mathfrak{P}(\mathtt{p}))\Big) \wedge \tag{6.3}$$

$$\Big(\big(\mathtt{p}_{\mathsf{E}} \in \mathsf{eff}(i) \wedge \mathsf{isFutREv}(i)\big) \to$$

$$[i+1] \vdash \mathsf{post}(\mathfrak{P}(p))[\mathtt{result} \setminus \mathsf{ret}(i)] \wedge \bigvee_{\mathtt{m}\in\mathsf{read}(\mathfrak{P}(p))} \mathsf{met}(i) \doteq \mathtt{m}\Big)\Big) \wedge \tag{6.4}$$

$$\bigwedge_{\mathtt{m}\in\mathbf{dom}\mathfrak{M}} \Big(\big(\mathsf{isInvocEv}(i) \wedge \mathsf{met}(i) \doteq \mathtt{m}\big) \to [i+1] \vdash \widetilde{\mathfrak{M}(\mathtt{m})}\Big)\Big) \tag{6.5}$$

Figure 6.3: Semantics α_{met} of $\mathbb{T}_{\mathsf{met}}$. $\widetilde{\mathfrak{M}(\mathtt{m})}$ at index $i+1$ is the precondition $\mathfrak{M}(\mathtt{m})$ where the abstract method parameters are replaced by the parameters in the invocation event at i.

Definition 6.6. *The behavioral specification $\mathbb{T}_{\mathsf{met}}$ of method contracts is the pair $(\tau_{\mathsf{met}}, \alpha_{\mathsf{met}})$. The syntax τ_{met} is defined by the following triple. Its semantics is shown in Fig. 6.3, where* pre *selects the first element of a pair and* post *the second one.*

$\mathsf{P} \to \big((\mathsf{IFOS} \times \mathsf{IFOS}) \cup (\mathsf{IFOS} \times \mathscr{P}(\mathsf{M}))\big)$	*Program Point Specification*
$\times(\mathsf{M} \to \mathsf{IFOS})$	*Call Conditions*
$\times\mathsf{IFOS}$	*Method Postcondition*

We write method contract types as $(\mathfrak{P},\mathfrak{M},\varphi)$.

The method contract type maps contains three elements: The first one, the *program point specification* maps program-point identifiers of **await** statements to their suspension assumption and suspension assertions, and program-point identifiers of **get** statements to their resolving assumption. We demand that every program-point identifier p of an **await** statement is mapped to a pair of formulas that only contain class fields of the class containing the statement and that every program-point identifier p of a **get** statement is mapped to a formula containing no local variables and no field but the special symbol **result** and a set of method names. The second element, the *call conditions*, map method names to their parameter precondition. The last element is the method postcondition and may contain **result** and fields of the class of the method in question. The semantics describes the three elements separately:

- The method postcondition (6.1) models that if the last event is a resolving event, then the method postcondition holds. It does not use the semantics of the $\mathbb{T}_{\mathsf{pst}}$ type, because it does not specify the traces of loop bodies. As we only consider terminating systems, the last event of a *method* is always a resolving event.

- The program point specification is expressing the following: At suspension points the suspension assertion of the suspending point holds (6.2) and when the process continues, the suspension assumption does (6.3). This is something which cannot be enforced by the typed method, but must be in the specification to be used as additional information. In the soundness lemma we show that no selectable traces are lost by this condition. At each synchronization point, the futures are resolved by the correct method and their postcondition holds for the read value (6.4).

$$
\begin{aligned}
\mathfrak{P}_{\mathtt{m}} &= \{1 \mapsto (\textbf{this}.\mathtt{j} > 0 \wedge \textbf{this}.\mathtt{i} > 0, \textbf{this}.\mathtt{j} < 0 \wedge \textbf{this}.\mathtt{i} > 0),\\
&\qquad 2 \mapsto (\textbf{result}<0, \{\mathtt{ext}, \mathtt{n2}\})\}\\
\mathfrak{M}_{\mathtt{m}} &= \{\mathtt{ext} \mapsto \mathtt{a} > 0,\\
&\qquad \mathtt{n2} \mapsto \mathtt{a} > \mathtt{b},\\
&\qquad \mathtt{n} \mapsto \mathtt{a} > \mathtt{b},\\
&\qquad \mathtt{m} \mapsto \mathtt{f} \mathrel{!}= \mathtt{never} \wedge \mathtt{g} > 0\}\\
\varphi_{\mathtt{m}} &= \mathtt{result}<\textbf{this}.\mathtt{i} \wedge \mathtt{result}>0
\end{aligned}
$$

Figure 6.4: The method Contract of method `C.m`.

- The call conditions have the semantics that whenever a method is called, its parameter precondition holds (6.5).

Example 6.4. *We continue with the code in Fig. 6.2. Consider additionally the following interface to complete the specification. The method contract of method* `C.m` *as a* $\mathbb{T}_{\mathsf{met}}$ *specification is shown in Fig. 6.4*

```
1 interface J{
2     /*@ requires a > 0;
3      ensures result < 0; @*/
4     Int ext(Int a);
5 }
```

6.3 Behavioral Type

We next give the behavioral type for $\mathbb{T}_{\mathsf{met}}$.

Definition 6.7. *The method contract behavioral type* $\mathbb{T}_{\mathsf{met}}$ *extends the behavioral specification with* $(\iota_{\mathsf{met}}, \pi_{\mathsf{met}})$. *The calculus* π_{met} *is given in Fig. 6.5. Let* $\mathcal{S}_{\mathtt{m}} = (\varphi_{\mathtt{m}}, \chi_{\mathtt{m}}, \mathsf{succ}_{\mathtt{m}}, \mathsf{over}_{\mathtt{m}})$ *be the program point specification of method* $\mathtt{m}$. *Let* $\mathcal{S}_{\mathtt{p}} = (\varphi_{\mathtt{p}}, \chi_{\mathtt{p}}, \mathsf{succ}_{\mathtt{p}}, \mathsf{over}_{\mathtt{p}})$ *be the program point specification of method* $\mathtt{m}$. *The obligation schema of a* coherent *program is defined as*

$$
\begin{aligned}
\iota_{\mathsf{met}}(\mathtt{m}) &= \big(\varphi_{\mathtt{m}}, (\mathfrak{P}, \mathfrak{M}, \chi_{\mathtt{m}})\big)\\
\mathfrak{P}(\mathtt{p}) &= (\varphi_{\mathtt{p}}, \chi_{\mathtt{p}}) && \text{if } \mathtt{p} \text{ is a suspension point}\\
\mathfrak{P}(\mathtt{p}) &= (\mathsf{Reso}_{\mathtt{m}}(\mathtt{p}), \bigvee_{\mathtt{m}' \in \mathsf{Reso}_{\mathtt{m}}(\mathtt{p})} \widehat{\chi_{\mathtt{m}'}}) && \text{if } \mathtt{p} \text{ is a synchronization point}\\
\mathfrak{M}(\mathtt{m}') &= \psi_{\mathtt{m}'}
\end{aligned}
$$

where $\psi_{\mathtt{m}'}$ *is the parameter precondition of* $\mathtt{m}'$ *and* $\widehat{\chi_{\mathtt{m}'}}$ *replaces all heap accesses with logical variables and existentially quantifies over them. E.g. if* $\chi_{\mathtt{m}} = \textbf{this}.\mathtt{f} > \mathtt{result}$ *then* $\widehat{\chi_{\mathtt{m}}} = \exists f \in \mathtt{Int}.\ f > \mathtt{result}$.

Rule ($\mathbb{T}_{\mathsf{met}}$-**while**) is a again a trivial loop invariant rule. Rule ($\mathbb{T}_{\mathsf{met}}$-**return**) verifies the postcondition after the return statement. Note that rule ($\mathbb{T}_{\mathsf{met}}$-**skip**) does *not* do so, because φ is the method postcondition, not the statement postcondition. Rules ($\mathbb{T}_{\mathsf{met}}$-**awaitF**) and ($\mathbb{T}_{\mathsf{met}}$-**awaitB**) both verify the suspension assertion before the suspension, anonymize the heap and assume the suspension assumption after reactivation. Rule ($\mathbb{T}_{\mathsf{met}}$-**callV**) verifies the precondition of a method upon its call. The formula $\mathfrak{M}(\mathtt{m})(\mathtt{e}_1, ..., \mathtt{e}_n)$ replaces the method parameters of $\mathfrak{M}(\mathtt{m})$ by the expressions $\mathtt{e}_1, ..., \mathtt{e}_n$. Thus, this formula only needs to hold in the first state of the trace and the rule is composable. Rule ($\mathbb{T}_{\mathsf{met}}$-**readV**) verifies with a points-to type that the correct methods resolved the future and assumes the disjunction of their postconditions. The other rules are straightforward.

Lemma 6.2. *Type* $\mathbb{T}_{\mathsf{met}}$ *is sound.*

Proof. See p. 179.

As expected, the rules ($\mathbb{T}_{\mathsf{met}}$-awaitF), ($\mathbb{T}_{\mathsf{met}}$-awaitB) and($\mathbb{T}_{\mathsf{met}}$-readV) are enabling but not sound.

Lemma 6.3. *Let* Prgm *be a coherent program. If (1) all proof obligations can be closed, (2) the initial values of a class establish the heap precondition of the methods with empty succeeds sets and (3) the method trace condition holds than every global trace* γ *of* Prgm *is a model for the following gMSOT formula.*

$$\bigwedge_{\mathsf{C}\ in\ \mathsf{Prgm}} \forall \mathtt{x} \in \mathsf{C}.\ \forall m \in \mathsf{M}.\ \forall i \in \mathbf{I}.\ \mathsf{isFutEv}(i, \mathtt{x}, m) \rightarrow \bigwedge_{\mathtt{m}\ in\ \mathsf{C}} [i-1]m \doteq \mathtt{m} \rightarrow \widetilde{\chi_{\mathtt{m}}[\mathbf{this} \setminus \mathtt{x}]}$$

Proof. See p. 180.

I.e., if every method adheres to its contract and the context sets are adhered too, then every method establishes its postcondition. The $\tilde{\cdot}$ operation is analogous to the one in Fig. 6.3.

Lemma 6.4. *Type* $\mathbb{T}_{\mathsf{pst}}\ \mathbb{T}_{\mathsf{inv}}\ \mathbb{T}_{\mathsf{met}}\ \mathbb{T}_{\mathsf{fr}}$ *is sound and the global lemmas of the input types hold under their additional conditions.*

Proof. See p. 180.

This lemma follows almost directly from the soundness preservation of leading composition, only enablement has to be reproven. Fig. 6.6 shows $\mathbb{T}_{\mathsf{pst}}\ \mathbb{T}_{\mathsf{inv}}\ \mathbb{T}_{\mathsf{met}}\ \mathbb{T}_{\mathsf{fr}}$, where we shortened the semantic function $\alpha_{\mathsf{pst}}\ \alpha_{\mathsf{inv}}\ \alpha_{\mathsf{met}}\ \alpha_{\mathsf{fr}}$ to pimf and denote the composed type with $\mathbb{T}_{\mathsf{pimf}}$. The rules have been slightly beautified for presentation's sake.

6.4 Example

We give a larger example to illustrate method contracts in a more realistic scenario. As the case study, we use distributed computation of moving averages. This is a common task is data analysis, as long-term trends are clearer in smoothened data [104]. Given n data points $x_1, \ldots, x_n$, some forms of moving average $\mathsf{avg}(x_1, \ldots, x_n)$ can be expressed by some function f that takes the average of the first $n-1$ data points, the last data point and a parameter α.

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

E.g., an exponential moving average demands that α is between 0 and 1 and is expressed as

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

or with the above pattern and $f(acc, x, \alpha) = \alpha * x + (1-\alpha) * acc$.

Fig. 6.7 shows the principal class `Smoothing` of our system. A `Smoothing` instance gets a `Computation` instance `comp` passed, which is used for the actual computation and encapsulates f in its `cmp` method. The `Smoothing` instance itself gets passed a data row as a list to `smooth` and passes the data one-by-one to `comp` and collects the return values in the `inter` list of intermediate results. While doing so, it stays responsive: via `getCounter` one may inquire how many data points were already processed. Decoupling the list processing and the computation increases usability, as one `Smoothing` instance may be reused with different `Computation` instances.

It is necessary to specify the following properties for the usage of `smooth`:

1. No two executions of `smooth` overlap during the suspension.

$$\Gamma \Rightarrow \{U\}\big[\mathtt{v = e.}\mathbf{get}_{\mathtt{p}} \overset{\alpha_{\mathrm{p2}}}{\Vdash} \mathrm{reads}(\mathfrak{P}(\mathtt{p}))\big], \Delta$$

$$(\mathbb{T}_{\mathrm{met}}\text{-readV})\ \frac{\Gamma, \{U\}\{\mathtt{v} := \nu\}\mathrm{conds}(\mathfrak{P}(\mathtt{p})) \Rightarrow \{U\}\{\mathtt{v} := \nu\}\big[\mathtt{s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big], \Delta}{\Gamma \Rightarrow \{U\}\big[\mathtt{v = e.}\mathbf{get}_{\mathtt{p}}\mathtt{;\ s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big], \Delta}\ \nu \text{ fresh}$$

$$(\mathbb{T}_{\mathrm{met}}\text{-assignV})\ \frac{\Rightarrow \big[\mathtt{s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}{\Rightarrow \big[\mathtt{v = e;\ s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]} \qquad (\mathbb{T}_{\mathrm{met}}\text{-assignF})\ \frac{\Rightarrow \big[\mathtt{s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}{\Rightarrow \big[\mathbf{this}\mathtt{.f = e;\ s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}$$

$$(\mathbb{T}_{\mathrm{met}}\text{-awaitF})\ \frac{\Gamma \Rightarrow \{U\}\mathrm{pre}(\mathfrak{P}(\mathtt{p})), \Delta \qquad \Gamma, \{U\}\{U_{\mathscr{A}}\}\mathrm{post}(\mathfrak{P}(\mathtt{p})) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{await}\ \mathtt{e?}_{\mathtt{p}}\mathtt{;\ s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)], \Delta}$$

$$(\mathbb{T}_{\mathrm{met}}\text{-awaitB})\ \frac{\Gamma \Rightarrow \{U\}\mathrm{pre}(\mathfrak{P}(\mathtt{p})), \Delta \qquad \Gamma, \{U\}\{U_{\mathscr{A}}\}\mathtt{e}, \{U\}\{U_{\mathscr{A}}\}\mathrm{post}(\mathfrak{P}(\mathtt{p})) \Rightarrow \{U\}\{U_{\mathscr{A}}\}\big[\mathtt{s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big], \Delta}{\Gamma \Rightarrow \{U\}\big[\mathbf{await}\ \mathtt{e}_{\mathtt{p}}\mathtt{;\ s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big], \Delta}$$

$$(\mathbb{T}_{\mathrm{met}}\text{-callV})\ \frac{\Rightarrow \mathfrak{M}(\mathtt{m})(\mathtt{e}_1, \ldots, \mathtt{e}_n) \qquad \Rightarrow \big[\mathtt{s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}{\Rightarrow \big[\mathtt{v = f!m(e}_1, \ldots, \mathtt{e}_n\mathtt{);\ s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}$$

$$(\mathbb{T}_{\mathrm{met}}\text{-if})\ \frac{\Rightarrow \big[\mathtt{s';s''} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big] \qquad \Rightarrow \big[\mathtt{s;s''} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}{\Rightarrow \big[\mathbf{if}\mathtt{(e)\{s\}}\mathbf{else}\mathtt{\{s'\};s''} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}$$

$$(\mathbb{T}_{\mathrm{met}}\text{-skip})\ \frac{}{\Rightarrow \big[\mathbf{skip} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]} \qquad (\mathbb{T}_{\mathrm{met}}\text{-return})\ \frac{\Rightarrow \{\mathtt{result} := \mathtt{e}\}\varphi}{\Rightarrow \big[\mathbf{return}\ \mathtt{e} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}$$

$$(\mathbb{T}_{\mathrm{met}}\text{-while})\ \frac{\Rightarrow \big[\mathtt{s} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big] \qquad \Rightarrow \big[\mathtt{s'} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}{\Rightarrow \big[\mathbf{while}\mathtt{(e)\{s\};s'} \overset{\alpha_{\mathrm{met}}}{\Vdash} (\mathfrak{P}, \mathfrak{M}, \varphi)\big]}$$

Figure 6.5: Calculus π_{met} for the method contract type $\mathbb{T}_{\mathrm{met}}$.

$$(\mathbb{T}_{\text{pimf}}\text{-assignV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \mathtt{e}\}\left[\mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}{\Gamma \Rightarrow \{U\}\left[\mathtt{v} = \mathtt{e};\ \mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\text{pimf}}\text{-readV})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\left[\mathtt{v} = \mathtt{e}.\mathbf{get}_{\mathtt{p}} \overset{\alpha_{\text{p2}}}{\Vdash} \mathsf{reads}(\mathfrak{P}(\mathtt{p}))\right], \Delta \\ \Gamma, \{U\}\{\mathtt{v} := \nu\}\mathsf{conds}(\mathfrak{P}(\mathtt{p})) \Rightarrow \{U\}\{\mathtt{v} := \nu\}\left[\mathtt{s} \overset{\alpha_{\text{met}}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta\end{array}}{\Gamma \Rightarrow \{U\}\left[\mathtt{v} = \mathtt{e}.\mathbf{get}_{\mathtt{p}};\ \mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \nu \text{ fresh}, \mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\text{pimf}}\text{-assignF})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{heap} := \mathsf{store}(\mathtt{heap}, \mathtt{f}, \mathtt{e})\}\left[\mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}{\Gamma \Rightarrow \{U\}\left[\mathbf{this}.\mathtt{f} = \mathtt{e};\ \mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}, \mathtt{f} \in \mathfrak{W}$$

$$(\mathbb{T}_{\text{pimf}}\text{-awaitF})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}I, \Delta \qquad \Gamma \Rightarrow \{U\}\mathsf{pre}(\mathfrak{P}(\mathtt{p})), \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}\big(\mathsf{post}(\mathfrak{P}(\mathtt{p})) \wedge I\big) \Rightarrow \{U\}\{U_{\mathscr{A}}\}\left[\mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}(\mathfrak{S}(\mathtt{p})), \mathfrak{W}(\mathfrak{S}(\mathtt{p})), \mathfrak{S})\right], \Delta\end{array}}{\Gamma \Rightarrow \{U\}\left[\mathbf{await}\ \mathtt{e}?_{\mathtt{p}};\ \mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\text{pimf}}\text{-awaitB})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}I, \Delta \qquad \Gamma \Rightarrow \{U\}\mathsf{pre}(\mathfrak{P}(\mathtt{p})), \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}\big(\mathtt{e} \wedge \mathsf{post}(\mathfrak{P}(\mathtt{p})) \wedge I\big) \Rightarrow \{U\}\{U_{\mathscr{A}}\}\left[\mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}(\mathfrak{S}(\mathtt{p})), \mathfrak{W}(\mathfrak{S}(\mathtt{p})), \mathfrak{S})\right], \Delta\end{array}}{\Gamma \Rightarrow \{U\}\left[\mathbf{await}\ \mathtt{e}_{\mathtt{p}};\ \mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\text{pimf}}\text{-callV})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}\mathfrak{M}(\mathtt{m})(\mathtt{e}_1, \ldots, \mathtt{e}_n), \Delta \\ \Gamma \Rightarrow \{U\}\{\mathtt{v} := \nu\}\left[\mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta\end{array}}{\Gamma \Rightarrow \{U\}\left[\mathtt{v} = \mathtt{f!m}(\mathtt{e}_1, \ldots, \mathtt{e}_n);\ \mathtt{s} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \nu \text{ fresh}, \bigcup_{i \le n} \mathsf{loc}(\mathtt{e}_i) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\text{pimf}}\text{-if})\ \frac{\begin{array}{c}\Gamma, \{U\}\mathtt{e} \Rightarrow \{U\}\left[\mathtt{s};\mathtt{s}'' \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta \\ \Gamma, \{U\}\neg\mathtt{e} \Rightarrow \{U\}\left[\mathtt{s}';\mathtt{s}'' \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta\end{array}}{\Gamma \Rightarrow \{U\}\left[\mathbf{if}(\mathtt{e})\{\mathtt{s}\}\mathbf{else}\{\mathtt{s}'\};\mathtt{s}'' \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\text{pimf}}\text{-skip})\ \frac{\{U\}\chi}{\Gamma \Rightarrow \{U\}\left[\mathbf{skip} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}$$

$$(\mathbb{T}_{\text{pimf}}\text{-return})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{result} := \mathtt{e}\}\varphi, \Delta \qquad \Gamma \Rightarrow \{U\}\{\mathtt{result} := \mathtt{e}\}\chi, \Delta \qquad \Gamma \Rightarrow \{U\}I, \Delta}{\Gamma \Rightarrow \{U\}\left[\mathbf{return}\ \mathtt{e} \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}$$

$$(\mathbb{T}_{\text{pimf}}\text{-while})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}I', \Delta \qquad \Gamma, \{U\}\{U_{\mathscr{A}}\}(I' \wedge \neg\mathtt{e}) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s}' \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}', \mathfrak{W}', \mathfrak{S})\right], \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(I \wedge \mathtt{e}) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s} \overset{\text{pimf}}{\Vdash} I' \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}', \mathfrak{W}', \mathfrak{S})\right], \Delta\end{array}}{\Rightarrow \left[\mathbf{while}(\mathtt{e})\{\mathtt{s}\};\mathtt{s}' \overset{\text{pimf}}{\Vdash} \chi \between I \between (\mathfrak{P}, \mathfrak{M}, \varphi) \between (\mathfrak{R}, \mathfrak{W}, \mathfrak{S})\right], \Delta}\ \mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}'$$

with

$$\mathfrak{R}' = \mathfrak{R} \cap \bigcap_{i \in P'} \mathfrak{R}(\mathfrak{S}(i)), \mathfrak{W}' = \mathfrak{W} \cap \bigcap_{i \in P'} \mathfrak{W}(\mathfrak{S}(i))$$

Figure 6.6: Calculus for the composed method contract type $\mathbb{T}_{\text{pimf}}$.

```
interface ISmoothing extends IPositive{
 Unit setup(Computation comp);
 Int getCounter();
 List<Rat>
   smooth(List<Rat> input, Rat a);
}

class Smoothing(Computation c)
  implements ISmoothing {
  Int counter = 1;
  Bool lock = False;
  Unit setup(Computation comp) {
    this.c = comp;
  }
  Int getCounter() {
    return counter;
  }
```

```
 List<Rat> smooth(List<Rat> input, Rat a) {
   this.lock = True;
   this.counter = 1;
   List<Rat> work = tail(input);
   List<Rat> inter = list[input[0]];
   while (work != Nil) {
     Fut<Rat> f = c!cmp(last(inter), work[0], a);
     this.counter = this.counter + 1;
     await f?1;
     Rat res = f.get2;
     this.inter = concat(inter,list[res]);
     work = tail(work);
   }
   this.lock = False;
   return inter;
 }
}
```

Figure 6.7: ABS code of the controller part of the distributed moving average

2. The result is a smoothened version of the input.

The field `lock` is used to model whether an instance of smooth is already running or not[2].

The specified code is shown in Fig. 6.8. We use JML-style syntax for the formulas in the specification. The loop invariant is

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

Here, submin(`input`, i) returns the minimal element of `input` up to position i, submax(`input`, i) returns the maximal element of `input` up to position i and len(`input`) returns the length of `input`.

The specification of `Computation.cmp` requires that all parameters are positive and that the return value is between `iold` and `inew`, which is our specification for local smoothening [3].

The two interfaces for `ISmoothing` specify that the input list of `smooth` must contain only positive elements and be non-empty. The parameter a must be strictly between 0 and 1. The `smooth` method must be called after `setup` or after the termination of the last `smooth` process (but not interleaved after the suspension point). Method `getCounter` may be called anytime.

The class specifies that the `smooth` method requires an unlocked object and unlocks it upon termination. The suspension contract however requires that the object is locked before continuing. One may add this as a guard condition to the **await** statement, but this would *enforce* this behavior instead of *analyzing* whether it is established by the existing, weaker synchronization patterns.

Finally, the suspension point also requires that the suspension establishes the suspension assertion.

[2] In practice, and as presented in [84], this would be a ghost field. We discuss ghost fields in the next section.

[3] This is, however, not true for all algorithms.

Example 6.5. *Consider the three code fragments interacting with a* `Smoothing` *instance* `s` *given below. The left fragment fails to verify the context sets specified above: although called last, method* `smooth` *can be executed first due to reordering, failing its succeeds clause. The middle fragment also fails: The first* `smooth` *needs not terminate before the next* `smooth` *activation starts. They may interleave and violate the* `overlaps` *set of the suspension. The right fragment verifies. We use* **await** `o!m();` *as a shorthand notation for* `Fut<T> f = o!m();` **await** `f?;`.

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

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

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

Fig. 6.9 gives the specification after propagation. The specifications of `setup` and `getCounter` express different behavior patterns: `setup` ensures that it is unlocked when execution ends, `getCounter` only ensures that *if* it was locked it will stay locked. It does not specify the case what happens if it is unlocked.

6.5 Discussion

On the Verification of Resolving Contracts.

The $\mathbb{T}_{met}$-proofs of method contracts contain branches with $\mathbb{T}_{p2}$-behavioral modalities, for which we gave no calculus. To handle these branches, we propose two approaches:

1. Run a points-to analysis for futures [51] first and add the results as $\mathbb{T}_{p2}$-behavioral modalities to all preconditions. Then run the proof system.
2. Run the proof system, collect all open branches with $\mathbb{T}_{p2}$-behavioral modalities and then run the points-to analysis.

The second approach is more modular as the first approach, and allows one to postpone verification of a part of the program until the full program is available.

On the Specification of Resolving Contracts.

An alternative to resolving contracts would be to specify the expected postcondition. Resolving contracts emphasize that the **get** statement is a synchronization, while expected postconditions would emphasize that the **get** statement reads some data.

Annotating the postcondition, however, has the following downside: With resolving contracts, one can locally verify the method contract and only check the points-to specification once global information is available for composition. All heavy-weight deductive verification is done on modular specifications. If the method synchronizes wrongly, only light-weight verification fails.

If one would specify the expected postcondition, one would need heavy-weight deductive verification upon composition: One would use the points-to analysis to *infer* the set of methods a point synchronizes with and then one would need to verify that their postcondition implies the expected postcondition — which involves FOL reasoning and not just light-weight static analyses.

We did not give a default value for resolving contracts. One simple solution would be to introduce a special symbol that denotes all possible methods and give a special rule for it. We refrain from doing so here as it offers no insights but complicates the semantics.

On Coupling Parameter and Heap Precondition.

Our system is not able to specify a precondition that connects heap and parameters within one predicate symbol, e.g. that the input parameter is a valid index for a list stored in a field. We argue that such specifications are not natural in an Active Object setting when specifying *local* behavior: there is no entity that can control both heap and parameters for an asynchronous method call. Even a self-call allows the heap to change before the called process is scheduled. Thus, we do not include such preconditions for method contracts, but discuss them in our (global) Session Type system in chapter 7.

On Behavioral Subtyping and Propagation.

As already discussed, our system breaks behavioral subtyping when specifying context sets, but the reason is that in its original setting [52, 49] behavioral subtyping only considered properties established by one party — adherence to the context sets, however, is a global property established by multiple parties. We, thus, deem that in this case behavioral subtyping not only *could* be broken, but *must* be broken, because otherwise every party has to have full knowledge about the context sets of the called object — but this would break modularity and force that each class only implements one interface with context sets.

Similarly, propagation leads to the situation where the method is not checked against its specification, but against an enriched one. Thus, it may fail verification not because it fails to implement its contract, but because it fails to establish the precondition of another method, *which is not referenced from the verified method itself*. Synthesized specification is criticized in general [49], because it can introduce new mistakes or mask specification errors. We deem this acceptable in our case: If there would be no propagation, there would be an object-local check for coherence and if this check fails the user would have to do the propagation himself. Propagation is not distorting the original specification, but performs a modification that would have to be done by hand otherwise. In practice every piece of specification can be traced through the propagation and the user can be provided with feedback in terms of the original specification.

On Atomic Segments.

This work presents suspension contracts in terms of specifications of suspension points. An alternative view is that suspension contracts specify the *atomic segment* between two suspension points.[4]

An atomic segment is a sequence of statements starting at method start or a suspension point and ending at method end or a suspension point, such that there is no suspension point in between. The atomic segment has a purely sequential semantics, as during its execution no other process executes on the object.

Each atomic segment is specified by two suspension contracts: the suspension assumption of the suspension contract at its beginning is its precondition, the suspension assertion of the suspension contract at its end is its postcondition[5].

The atomic segment view emphasizes the generalization from method contracts without suspension, but is less clear about what is specified in more complex control flow patterns: In the presence of a conditional or loop between two suspension points, a suspension assertion of a suspension contract specifies the condition at the end of *multiple* atomic segments. The set of atomic segments of a method can be computed by generating all paths in the control flow graph that begin at the entry node or an await statement and end at an exit node or an await statement. We do, however, not require to compute the atomic segments explicitly. Fig. 6.10 shows a method and its atomic segments. The postcondition of `m` specifies the state at the end of *three* atomic segments (1,2 and 5). The precondition also specifies the state at the beginning of three atomic segments (1,2 and 3). Note that atomic segment 4 starts and ends at the same point.

When we, e.g., add point 0 to a succeeding context set, then we do not specify that one specific atomic segment must have run before, but that *any* of the atomic segments ending at suspension point 0 (i.e., 3 and 4) must have run before. All of them must guarantee the same suspension guarantee.

On Block Contracts and Atomic Segments.

The suspension contract specifies at least two atomic segments: the suspension guarantee is the postcondition of the atomic segments ending at this suspension points, the suspension assumption the precondition of the following atomic segments. An alternative would be to interpret the `ensures` clause as the postcondition of the *following* atomic segments. This way a suspension contract would resemble a

[4] This was also how it was presented in [84].

[5] Or by the outer method contract if it is one of the last or first atomic segments

block contract, with the block being the atomic segment(s). The reason that the system is not designed this way is that this would require that the postcondition of the whole method which specifies the result value would be annotated *inside the method body* and Furthermore, the chosen design ensures that the two specifications that are concerned with one suspension point (in the sense of rule ($\mathbb{T}_{\text{met}}$-awaitF) and ($\mathbb{T}_{\text{met}}$-awaitB)) are specified in the same place.

Ghost Specifications.

The field `lock` in the example in section 6.4 is only written, but never read, it is only present to specify whether a process of `smooth` is active or not. It is a natural part of the specification, not the implementation. We could use ghost specifications: ghost fields [73], which are declared by *//@ ghost D f = e;* and ghost assignments of the form *//@this.f = e;*. The semantics of a ghost field is the same as the one of a normal field and the semantics of the ghost assignment is the same as the one of a normal assignment. Ghost specifications allow us to introduce state into the object, that is *only* used for specifications. However, as their semantics is not distinguishable from the one of normal assignments, except that ghost fields cannot be read in normal assignments, we refrain from introducing them formally. We conjecture that their introduction is straightforward. Similarly, method redeclarations may be "ghost declarations" in special comments.

On Postconditions.

Contrary to preconditions, postconditions are not split for verification but projected upon usage. A weaker kind of split is introduced by inheritance: the `result` variable is specified in the interface(s), while the heap is specified in the class. However, the class postcondition may also include `result`. The reason for this design is that the postcondition has one process that must establish it, while the precondition has two (caller and last running process).

```
1  interface Computation{
2    /*@ requires inew > 0 && iold > 0 && param > 0;
3        ensures \result > 0 &&
4                (iold >= inew -> (\result <= iold && \result >= inew)) &&
5                (iold <= inew -> (\result >= iold && \result <= inew));
6    @*/
7    Rat cmp(Rat iold, Rat inew, Rat param);
8  }
9
10 interface IPositive{
11   /*@ requires \forall Int i; 0 <= i < len(input) ; input[i] > 0 @*/
12   List<Rat> smooth(List<Rat> input, Rat a):
13 }
14 interface ISmoothing extends IPositive{
15   Unit setup();
16   Int getCounter();
17   /*@ requires 1>a>0 && len(input)>0;
18        ensures len(result) == len(input) &&
19                \forall Int i; 0 <= i < len(result);
20                        result[i] > 0 && min(input) <= result[i] <= max(input);
21        succeeds {setup, smooth}; overlaps {getCounter};  @*/
22   List<Rat> smooth(List<Rat> input, Rat a);
23 }
24
25 class Smoothing(Computation c) implements ISmoothing {
26   Int counter = 1;
27   Bool lock = False;
28
29   Unit setup() { this.counter = 1;}
30   Int getCounter() { return counter; }
31
32 /*@ requires !this.lock && ensures !this.lock; @*/
33   List<Rat> smooth(List<Rat> input, Rat a) {
34     this.lock = True;
35     this.counter = 1;
36     List<Rat> work = tail(input);
37     List<Rat> inter = list[input[0]];
38     while (work != Nil) {
39       Fut<Rat> f = c!cmp(last(inter), work[0], a);
40       this.counter = this.counter + 1;
41       /*@ requires this.lock; succeeds {1}; overlaps {getCounter}; @*/
42       await f?1;
43       /*@ resolvedBy {Computation.cmp} @*/
44       Rat res = f.get2;
45       inter = concat(inter,list[res]);
46       work = tail(work);
47     }
48     this.lock = False;
49     return inter;
50   }
51 }
```

Figure 6.8: Specified Distributed Moving Average, using JML-style formulas.

```
class Smoothing (Computation c) implements ISmoothing {
  Int counter = 1;
  Bool lock = False;

  /*@ requires true; ensures !this.lock: @*/
  Unit setup(Computation comp) { c = comp; }

  /*@ ensures old.lock ==> this.lock @*/
  Int getCounter() { return counter; }

  /*@ requires !this.lock @*/
  /*@ requires \forall Int i; 0 <= i < len(input) ; input[i] > 0 @*/
  /*@ requires 1>a>0 && len(input)>0 @*/
  /*@ ensures len(\result) == len(input) && \forall Int i; 0 <= i < len(\result);
              \result[i] > 0 && min(input) <= \result[i] <= max(input); @*/
  /*@ ensures !this.lock@*/
  List<Rat> smooth(List<Rat> input, Rat a) {
    this.lock = True;
    counter = 1;
    List<Rat> work = tail(input);
    List<Rat> inter = list[input[0]];

    while (work != Nil) {
      Fut<Rat> f = c!cmp(last(inter), work[0], a);
      counter = counter + 1;
      /*@ requires this.lock; ensures this.lock; @*/
      await f?1;
      /*@ resolvedBy {Computation.cmp} @*/
      Rat res = f.get2;
      inter = concat(inter,list[res]);
      work = tail(work);
    }
    this.lock = False;
    return inter;
  }
}
```

Figure 6.9: Specified Distributed Moving Average, after propagation.

```
Unit m() {
  s1
  if (...) {
    s2
    while(...){
      s3 await f?0; s4
    }
    s5
  } else { s6 }
  s7
}
```

m has five atomic segments:

1. s_1, s_6, s_7
2. s_1, s_2, s_5, s_7
3. $s_1, s_2, s_3,$ **await** $f?_0$
4. $s_4, s_3,$ **await** $f?_0$
5. s_4, s_5, s_7

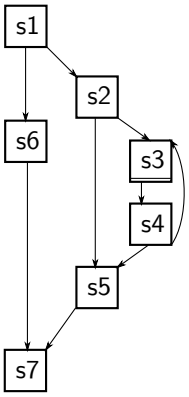


Figure 6.10: Structure of atomic segments. The statements s_i contain no **if**, **while**, or **await** statements.

7 Stateful Session Types for Active Objects

The systems discussed so far are *bottom-up* approaches: they specify only parts of the system, such as methods or classes — the specification of the whole system is given only implicitly. This chapter describes a *top-down* approach: Stateful Session Types for Active Objects. Session Types [71] globally specify a part of the system that may be composed itself. The subsystem is called *session* and is required to be a *"natural unit of communication"* [72]. For Active Objects, a natural unit is a set of objects that do not communicate to objects outside of this set [83] — here we consider the case where the whole system is a single session, but describe how multiple sessions in a system can be handled.

In the original formulation [71, 72], Session Types describe the communication over channels in channel-based concurrency by (1) specifying a global type that describes all the communication over a given channel, (2) projecting the global type on the endpoints and generating a local type for each endpoint and (3) statically checking each endpoint against its local type. Projection and type system are designed such that the local static checks guarantee generic communication correctness: Globally, deadlock freedom and locally, that there are no unexpected or orphaned messages on a given channel. This approach was later adopted to object-oriented languages with channels [42] and other concurrency models, such as actors [30, 107], boxed ambients [55] and Active Objects [83].

Session Types, in the sketch above, check two properties: deadlock freedom/communication correctness and protocol adherence. Protocol adherence describes that the session follows the protocol specified by the global type. The system we present here checks *only* for protocol adherence and builds on earlier work to express richer protocols with data and assertions [83, 13] — they specify the state changes within a protocol, hence they are called Stateful Session Types.

Using Session Types for Active Objects poses multiple challenges. For one, the use of multiple communication possibilities: the protocol describes not only message sending, but method calls, future synchronization and communication via the heap. For another, the notion of endpoints is layered: method calls target an *object*, while future synchronization targets a *process* inside an object.

Previous systems for Session Types for Active Objects, thus, had two projections [83, 82]. First, the global type is projected on the objects participating in the session. This generates object-local types, the local view of the object on the protocol. Then each object-local type is projected on the participating method. This generates method-local types, the local view of all processes of the method on the protocol. Here, we are able to simplify the projection mechanism in multiple ways: First, only one projection is needed, second, we use two analyses to discard invalid global types *before* projection and, third, we are able to merge propagation and causality analysis into one step, which is less complex than previous propagation and causality analyses. We also added the ability to track values passed over multiple parties, a feature available in previous systems for channels [12], but not for Active Objects.

We give the behavioral type for Session Types as a leading type. For one, this specification touches upon most aspects in a way that makes decomposition into multiple types which are composed by the leading operator difficult. For another, it allows us to demonstrate further composition patterns: the type still makes use of $\mathbb{T}_{\mathsf{pst}}$ for loop invariants and can encode object invariants and a class of method contracts.

7.1 Global Types

Global types describe the global view of the communication in a session/system, i.e., a protocol. To do so, it specifies interactions between *roles*. A role is an endpoint in the protocol – for verification, a mapping from roles to objects has to be given by the user. The interactions themselves describe the kind

of interaction, the involved roles, conditions on communication data and *tracked values*. A tracked value is used to track a semantic value through the execution of the protocol *on specification level*.

Definition 7.1 (Tracked Values and Roles)**.** *We denote the set of all roles with $\mathscr{R}$ and the set of tracked values with $\mathscr{T}$. These sets do not intersect, roles does not intersect with any syntactic category or the set of symbolic values, $\mathscr{T}$ is a subset of the function symbols (in* BPL*).*

A tracking constraint C is a map from $\mathscr{T}$ to sets of expressions. It models that a tracked value t is available in the expressions $\mathsf{C}(\mathsf{t})$.

This means that all expressions in $\mathsf{C}(\mathsf{t})$ evaluate to the same semantic value.

Convention 5. *We write tracking constraints where every tracked value is mapped onto a single expression, i.e., of the form $\{\mathsf{t}_1 \mapsto \{\mathtt{e}_1\}, \mathsf{t}_2 \mapsto \{\mathtt{e}_2\}, \dots\}$, as follows in examples*

$$\mathsf{t}_1 \textbf{ as } \mathtt{e}_1, \mathsf{t}_2 \textbf{ as } \mathtt{e}_2, \dots$$

Each tracked value is associated with a logical variable in gMSOT, a logical variable in lMSOT and a function symbol in BPL (all of the same name and fitting type), so formulas may refer to them.

Syntax

For the syntax of global types, we distinguish between global protocols and global types. A global protocol is a global type, prefixed by an initialization action.

Definition 7.2 (Global Types and Protocols)**.** *Let $\mathbf{p},\mathbf{q}$ range over $\mathscr{R}$, t over $\mathscr{T}$, φ over lFOS formulas, ψ over gFOS formulas and C over tracking constrains.*

$\mathbf{GP} ::= \mathbf{0} \xrightarrow{\mathsf{t}} \mathbf{p} : \mathtt{m}(\!\langle \varphi, \mathsf{C} \rangle\!) \,.\, \mathbf{G}$	*Global Protocol*
$\mathbf{G} ::= \mathbf{p} \xrightarrow{\mathsf{t}} \mathbf{q} : \mathtt{m}(\!\langle \varphi, \mathsf{C} \rangle\!)$	*Call Action*
$\mid \mathbf{p}{\downarrow}\mathsf{t}(\!\langle \varphi \rangle\!)$	*Termination Action*
$\mid \mathbf{p}{\uparrow_{\mathsf{t}}}\mathsf{t}$	*Synchronization Action*
$\mid \mathsf{Rel}(\!\langle \mathbf{p}, \mathsf{t}, \varphi, \varphi \rangle\!)$	*Suspension Action*
$\mid \mathbf{G} \,.\, \mathbf{G}$	*Sequential Composition*
$\mid \mathbf{p}\{\varphi_i : \mathbf{G}_i\}_{i \in I}$	*Branching*
$\mid (\mathbf{G})^*_\psi$	*Repetition*
$\mid \mathsf{skip}$	*Empty Action*
$\mid \mathbf{end}$	*Protocol End*

We call non-composed types actions *and tracked values that are tracking the generated future in initial and call actions* tracked futures*.*

A call action $\mathbf{p} \xrightarrow{\mathsf{t}} \mathbf{q} : \mathtt{m}(\!\langle \varphi, \mathsf{C} \rangle\!)$ specifies a method call from role $\mathbf{p}$ to $\mathbf{q}$ on method $\mathtt{m}$. The tracked value t is used to track the future generated on $\mathbf{p}$-side. Formula φ specifies the state of $\mathbf{p}$ and the parameters of the call. The track constraint C defines a set of equalities between parameters and tracked values. This specifies that either a passed value in a parameter is known as a tracked value in the rest of the protocol (i.e., name generation) or that a parameter has a tracked value as its passed value. Consider the following protocol:

$$\mathbf{0} \xrightarrow{\mathsf{t}_f} \mathbf{p} : \mathtt{m}(\!\langle \mathsf{true}, \mathsf{t} \textbf{ as } \mathtt{x} \rangle\!) \,.\, \mathbf{p} \xrightarrow{\mathsf{t}_{f_2}} \mathbf{q} : \mathtt{m}_2(\!\langle \mathtt{y} > 0 \wedge \mathsf{t} \doteq \mathtt{y}, \mathsf{t}_2 \textbf{ as } \mathtt{y}{*}\mathtt{y} \rangle\!) \,.\, \mathbf{G}$$

It specifies that **p** tracks the value it got sent as parameter `x` as `t`. It then sends this value to **q** as parameter `y`. In the type **G** the square of the value of `y` is known as t_2 and `y` is guaranteed to be greater than 0. The initial call of the global protocol is analogous to the call action with a special role **0**.

A termination action $\mathbf{p}\downarrow \mathtt{t}(\!|\varphi|\!)$ specifies the termination of the process with the tracked future `t` in **p**, i.e., `t` has been used in the call action for this process. The specification φ specifies the state of **p** upon termination (of the currently active process).

A synchronization action $\mathbf{p}\uparrow_{t_2} t_1$ specifies that the role **p** reads from the future tracked by the value t_1 (i.e., t_1 has been used in the call action for some process). The tracked value t_2 introduces a name for the read value.

A suspension action $\mathsf{Rel}(\!|\mathbf{p}, \mathtt{t}, \varphi_1, \varphi_2|\!)$ specifies that the role **p** suspends its currently active process until the future tracked by `t` is resolved. The value `t` is *not* the future of the suspended process. The first formula φ_1 specifies the state of **p** at the moment of suspension, the second formula φ_2 specifies the state of **p** at the moment of reactivation.

Sequential composition is straightforward. Branching $\mathbf{p}\{\varphi_i : \mathbf{G}_i\}_{i\in I}$ gives role **p** the choice how to continue the protocol. Each protocol branch $\mathbf{G}_i$ is guarded by a condition φ_i. These conditions may overlap[1]. Repetition $(\mathbf{G})^*_\psi$ specifies that **G** is executed zero times or finitely often, with ψ as a *global* invariant. I.e., it describes the state of the whole system. A ending action **end** denotes the end of the protocol. Syntactically we allow actions after **end**, but they are treated as dead specification, analogous to dead code after **return** statements in, e.g., Java.

We demand that the two roles in the call action differ. Regarding the formulas, the following restrictions have to hold:

- The formula in the call action only contains field symbols of the caller and method parameters of `m` as program variables (except `heap`). The connection of role and caller is established below.
- The formula in the termination action only contains field symbols of the class of the terminating process and only `result` as its sole program variable (except `heap`).
- The formula in the repetition only contains `heap` as a program variable.
- The formulas in the guards contain only tracked values.

During projection we also reject protocols that cannot be established cooperatively, e.g., if an endpoint verifies a tracking constraint that contains tracked values that this endpoint has no access to.

Convention 6. *If the condition φ in an action is* `true` *or a track constraint is $\emptyset$, then we omit it in examples. E.g., we write* $\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m}(\!|\mathtt{true},\emptyset|\!)$ *as* $\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m}$ *and* $\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m}(\!|\mathtt{i}<0,\emptyset|\!)$ *as* $\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m}(\!|\mathtt{i}<0|\!)$. *We also omit the tracked value identifying a future in a call if it is not used in an example.*

Tracking constraints either introduce new tracked values or describe propagation of the tracked value.

Example 7.1. *The following protocol describes that* **p** *sends some value to* **q** *in parameter* `v` *which is tracked by* `t`. **q** *sends the same value to* **r** *in parameters* `w`.

$$0\rightarrow\mathbf{p}:\mathtt{m}.\mathbf{p}\rightarrow\mathbf{q}:\mathtt{m}_2(\!|\mathtt{t}>0,\mathtt{t}\ \mathbf{as}\ \mathtt{v}|\!).\mathbf{q}\rightarrow\mathbf{r}:\mathtt{m}_3(\!|\mathtt{t}\ \mathbf{as}\ \mathtt{w}|\!)\ldots.$$

The following protocol describes that the parameter `w` *is tracked by* **r** *by a fresh tracked value* t_2.

$$0\rightarrow\mathbf{p}:\mathtt{m}.\mathbf{p}\rightarrow\mathbf{q}:\mathtt{m}_2(\!|\mathtt{t}>0,\mathtt{t}\ \mathbf{as}\ \mathtt{v}|\!).\mathbf{q}\rightarrow\mathbf{r}:\mathtt{m}_3(\!|\mathtt{t}_2\ \mathbf{as}\ \mathtt{w}|\!)\ldots.$$

Another aspect of the difference is that in the first protocol, **r** *knows that* $\mathtt{w}>0$ *holds.*

[1] Two formulas overlap, if there is a structure which is a model for both. E.g., $i>0$ and $i<0$ do not overlap, but $i\geq 0$ and $i\leq 0$ do.

The semantics of a global protocol are sequences of events and states that are not distinguishable by any participant. It is given as a translation into gMSOT. For readability, we (1) shorten parameter lists to single parameters (2) omit semantics effects (which are not specified by Session Types anyway) and (3) replace some unspecified elements with _. This symbol denotes a logical variable of the fitting sort, existentially bound with the smallest possible scope. (4) use abbreviations like 0 or $i+1$ instead of defining the first index or the successor of i explicitly. We stress, however, that for the purpose of relativization we consider the unrolled formula. E.g.

$$[i] \doteq \mathsf{condEv}(\_,\_,\emptyset) \equiv \exists \mathtt{x} \in \mathsf{Obj}.\ \exists f \in \mathsf{Fut}.\ [i] \doteq \mathsf{condEv}(\mathtt{x}, f, \emptyset)$$

Definition 7.3 (Global Type Semantics)**.** *The semantics of global protocols and global types are given in Fig. 7.1. We assume that equalities between the logical variables (in gMSOT) and function symbols (in lFOS) of the tracked values are implicitly added. We require the following auxiliary operations.*

Semantics of Tracking Constraints. *A tracking constraint* C *is translated into a sequence of quantifiers binding every tracked value* $\mathtt{t}$ *as a logical variables of fitting type* $\mathtt{D}(\mathtt{t})$ *to the given expression. Note that we assume that every expression is implicitly translated into a term. Translation is relative to a formula* φ*, where the tracked values may be used (if they are free within* φ*), and a set of known tracked values* T*. The set* T *describes those tracked values, which are already bound — the translation needs only to express their propagation.*

$$Q(\mathsf{C}, \varphi, T) = \exists_{\mathtt{t} \in \mathbf{dom}(\mathsf{C}) \setminus T} \mathtt{t} \in \mathtt{D}(\mathtt{t}).\ \left(\left(\bigwedge_{\substack{e \in \mathsf{C}(\mathtt{t}) \\ \mathtt{t} \in \mathbf{dom}(\mathsf{C})}} \mathtt{t} \doteq e \right) \wedge \varphi \right)$$

the notation $\exists_{\mathtt{t} \in \mathbf{dom}(\mathsf{C}) \setminus T}$ *stands for a sequence of quantifiers, one for each variables in the set* $\mathbf{dom}(\mathsf{C}) \setminus T$*. For example*

$$Q\big(\mathtt{t}\ \mathbf{as}\ \mathtt{v}, \mathtt{t}_2\ \mathbf{as}\ \mathtt{w{+}2}, \varphi, \{\mathtt{t}\}\big) = \exists \mathtt{t}_2 \in \mathtt{Int}.\ \big(\mathtt{t}_2 \doteq \mathtt{w{+}2} \wedge \mathtt{t} \doteq \mathtt{v} \wedge \varphi\big)$$

Actively Communicating Objects. *We say that an object* $\mathtt{x}$ *is actively communicating at position* i*, written* $\mathsf{Act}(i, \mathtt{x})$ *if* i *is an event issued by* $\mathtt{x}$*. An event is issued by* $\mathtt{x}$ *if it is not* noEv *and its first parameter is* $\mathtt{x}$*, e.g.,* $[i] \doteq \mathsf{invREv}(\mathtt{x}, \_, \_, \_)$*.*

The call action models that there are invocation and invocation reaction events with correct caller, callee and method. The tracking constraint is rewritten with a quantifier as described above – under this quantifiers, the call condition has to hold in the state when calling, i.e., when instantiating the method parameters with the actual call parameters. Additionally, the following global type $\mathbf{G}'$ has to hold, but is relativized as follows: if any specified event is issued by the same object as the call, then it has to be at an index *idx* after the invocation event in this call action. This enforces that events may be reordered only in a way that is invisible to any object. The initial call is analogous, but with fixed constants for the first event.

Example 7.2. *Consider the semantics of the following global type:*

$$\mathbf{p} \xrightarrow{\mathtt{t}} \mathbf{q} : \mathtt{m}.\mathbf{p} \xrightarrow{\mathtt{t}_2} \mathbf{r} : \mathtt{m}_2.\mathbf{G}$$

It must express that the calls happen in the correct order, but the methods may start in any order. Applying the semantics above results in (slightly simplified)

$$\begin{aligned}
&\exists i \in \mathbf{I}.\ \exists j \in \mathbf{I}.\ \Big([i] \doteq \mathsf{invEv}(\mathbf{p}, \mathbf{q}, \mathtt{t}, \mathtt{m}, _) \wedge [j] \doteq \mathsf{invREv}(\mathbf{q}, \mathtt{t}, \mathtt{m}, _) \wedge \\
&\quad \exists i' \in \mathbf{I}.\ \exists j' \in \mathbf{I}.\ \Big(\\
&\quad (\mathsf{Act}(i', \mathbf{p}) \to i' > i) \wedge (\mathsf{Act}(i', \mathbf{q}) \to i' > j) \wedge (\mathsf{Act}(j', \mathbf{p}) \to j' > i) \wedge (\mathsf{Act}(j', \mathbf{q}) \to j' > j) \wedge \\
&\quad [i'] \doteq \mathsf{invEv}(\mathbf{p}, \mathbf{r}, \mathtt{t}_2, \mathtt{m}_2, _) \wedge [j'] \doteq \mathsf{invREv}(\mathbf{r}, \mathtt{t}_2, \mathtt{m}_2, _) \wedge \\
&\quad \mathit{gtsem}(\mathbf{G})[\mathit{idx} \in \mathbf{I} \setminus \dots] \Big) \Big)
\end{aligned}$$

$$gtsem\big(\mathbf{0}\xrightarrow{\mathsf{t}}\mathbf{p}{:}\mathtt{m}(\!|\varphi, \mathsf{C}|\!)\ .\ \mathbf{G}\big) = \exists \mathsf{t} \in \mathsf{Fut}.\ \exists \mathtt{e} \in \mathtt{D}^{\mathtt{p}}(\mathtt{m}).\ \Big($$
$$[2] \doteq \mathsf{invREv}(\mathbf{p}, \mathsf{t}, \mathtt{m}, \mathtt{e})$$
$$\wedge Q\big(\mathsf{C}, [3] \vdash \varphi(\mathtt{e})[\mathbf{this} \setminus \mathbf{p}] \wedge gtsem\big(\mathbf{G}, \mathbf{dom}(\mathsf{C}) \cup \{\mathsf{t}\}\big), \{\mathsf{t}\}\big)\big[idx \in \mathbf{I} \setminus (\mathsf{Act}(idx, \mathbf{p}) \rightarrow idx > 1)\big]\Big)$$

$$gtsem\big(\mathbf{p}\xrightarrow{\mathsf{t}}\mathbf{q}{:}\mathtt{m}(\!|\varphi, \mathsf{C}|\!)\ .\ \mathbf{G}, T\big) = \exists \mathsf{t} \in \mathsf{Fut}.\ \exists i, j \in \mathbf{I}.\ \exists \mathtt{e} \in \mathtt{D}^{\mathtt{p}}(\mathtt{m}).\ \Big($$
$$[i] \doteq \mathsf{invEv}(\mathbf{p}, \mathbf{q}, \mathsf{t}, \mathtt{m}, \mathtt{e}) \wedge [j] \doteq \mathsf{invREv}(\mathbf{q}, \mathsf{t}, \mathtt{m}, \mathtt{e})$$
$$\wedge Q\big(\mathsf{C}, [i+1] \vdash \varphi(\mathtt{e})[\mathbf{this} \setminus \mathbf{p}] \wedge gtsem\big(\mathbf{G}, T \cup \mathbf{dom}(\mathsf{C}) \cup \{\mathsf{t}\}\big), T \cup \{\mathsf{t}\}\big)$$
$$\big[idx \in \mathbf{I} \setminus (\mathsf{Act}(idx, \mathbf{p}) \rightarrow idx > i) \wedge (\mathsf{Act}(idx, \mathbf{q}) \rightarrow idx > j)\big]\Big)$$

$$gtsem\big(\mathbf{p}{\downarrow}\mathsf{t}(\!|\varphi|\!)\ .\ \mathbf{G}, T\big) = \exists i \in \mathbf{I}.\ \exists \mathtt{e} \in \mathtt{D}(\mathtt{m}).\ \Big($$
$$[i] \doteq \mathsf{futEv}(\mathbf{p}, \mathsf{t}, \_, \mathtt{e}) \wedge [i-1] \vdash \varphi[\mathtt{result} \setminus \mathtt{e}]$$
$$\wedge Q\big(\mathsf{C}, gtsem\big(\mathbf{G}, T\big), T\big)\big[idx \in \mathbf{I} \setminus \mathsf{Act}(idx, \mathbf{p}) \rightarrow idx > i\big]\Big)$$

$$gtsem\big(\mathbf{p}{\uparrow_{\mathsf{t}_2}}\mathsf{t}_1\ .\ \mathbf{G}, T\big) = \exists i \in \mathbf{I}.\ \exists \mathsf{t}_2 \in \mathtt{D}(\mathsf{t}_2).\ \Big($$
$$[i] \doteq \mathsf{futREv}(\mathbf{p}, \mathsf{t}_1, \_, \mathsf{t}_2)$$
$$\wedge Q\big(\mathsf{C}, gtsem(\mathbf{G}, T \cup \{\mathsf{t}_2\}), T \cup \{\mathsf{t}_2\}\big)\big[idx \in \mathbf{I} \setminus (\mathsf{Act}(idx, \mathbf{p}) \rightarrow idx > i)\big]\Big)$$

$$gtsem\big(\mathsf{Rel}(\!|\mathbf{p}, \mathsf{t}, \varphi, \varphi'|\!)\ .\ \mathbf{G}, T\big) = \exists i, j \in \mathbf{I}, \exists f \in \mathsf{Fut}.\ \Big($$
$$[i] \doteq \mathsf{suspEv}(\mathbf{p}, f, \mathsf{t}) \wedge [i+1] \vdash \varphi[\mathbf{this} \setminus \mathbf{p}]$$
$$\wedge [j] \doteq \mathsf{suspREv}(\mathbf{p}, f, \mathsf{t}) \wedge [j+1] \vdash \varphi'[\mathbf{this} \setminus \mathbf{p}]$$
$$\wedge \forall k \in \mathbf{I}.\ (i < k < j) \rightarrow [k] \not\doteq \mathsf{suspREv}(\mathbf{p}, f, \mathsf{t})$$
$$\wedge gtsem(\mathbf{G}, T)\big[idx \in \mathbf{I} \setminus (\mathsf{Act}(idx, \mathbf{p}) \rightarrow idx > i)\big]\Big)$$

$$gtsem\big(\mathbf{p}\{\varphi_i : \mathbf{G}_i\}_{i \in I}\ .\ \mathbf{G}, T\big) = \bigvee_{i \in I}\big(\exists i_0 \in \mathbf{I}.\ \wedge [i_0] \vdash \varphi_i[\mathbf{this} \setminus \mathbf{p}] \wedge gtsem(\mathbf{G}_i.\mathbf{G}, T)[idx \in \mathbf{I} \setminus idx \geq i_0]\big)$$

$$gtsem\big((\mathbf{G})^*_{\psi}\ .\ \mathbf{G}', T\big) = gtsem\big(\mathbf{G}', T\big) \vee \exists I \subseteq \mathbf{I}.\ \Big($$
$$\forall i, j \in I.\ \big(i < j \wedge \nexists k \in I.\ i < k < j\big) \rightarrow gtsem(\mathbf{G}, T)[idx \in \mathbf{I} \setminus i \leq idx \leq j]$$
$$\wedge \forall i \in I.\ [i] \vdash \psi$$
$$\wedge \exists min \in I.\ \forall i \in I.\ i \geq min \wedge \forall i \in \mathbf{I}.(i < min \wedge isEvent(i)) \rightarrow [i] \doteq \mathsf{noEv}$$
$$\wedge \exists max \in I.\ \forall i \in I.\ i \leq max \wedge gtsem\big(\mathbf{G}', T\big)[idx \in \mathbf{I} \setminus idx \geq max]\Big)$$

$$gtsem\big(\mathbf{end}, T\big) = gtsem\big(\mathbf{skip}, T\big) = \mathsf{true} \qquad gtsem\big(\mathbf{skip}.\mathbf{G}, T\big) = gtsem\big(\mathbf{G}, T\big)$$

Figure 7.1: Semantics of Global Protocols and Types. $\mathtt{D}^{\mathtt{p}}(\mathtt{m})$ is the type of the parameter of $\mathtt{m}$, $\mathtt{D}(\mathtt{m})$ is the return type of $\mathtt{m}$. For types $\mathbf{G}$ without a sequential composition, we set $gtsem(\mathbf{G}, T) = gtsem(\mathbf{G}\ .\ \mathbf{skip}, T)$.

Obviously, the guards introduced by the relativization can be simplified: On one hand, $\mathsf{Act}(j', \mathbf{r})$ *holds and thus both* $\mathsf{Act}(j', \mathbf{p})$ *and* $\mathsf{Act}(j', \mathbf{q})$ *do not. On the other hand,* $\mathsf{Act}(i', \mathbf{p})$ *does hold. Simplified, the semantics becomes:*

$$\begin{aligned}\exists i \in \mathbf{I}.\, \exists j \in \mathbf{I}.\, \exists i' \in \mathbf{I}.\, \exists j' \in \mathbf{I}.\, \Big(& i' > i \wedge gtsem(\mathbf{G})[idx \in \mathbf{I} \setminus \dots] \wedge \\ & [i] \doteq \mathsf{invEv}(\mathbf{p}, \mathbf{q}, \mathrm{t}, \mathtt{m}, \_) \wedge [j] \doteq \mathsf{invREv}(\mathbf{q}, \mathrm{t}, \mathtt{m}, \_) \wedge \\ & [i'] \doteq \mathsf{invEv}(\mathbf{p}, \mathbf{r}, \mathrm{t}_2, \mathtt{m}_2, \_) \wedge [j'] \doteq \mathsf{invREv}(\mathbf{r}, \mathrm{t}_2, \mathtt{m}_2, \_) \Big)\end{aligned}$$

Note that the order on $\mathbf{p}$ *side is specified, but not the order of the actions of* $\mathbf{q}$ *and* $\mathbf{r}$*.*

The semantics of termination, synchronization and suspension are analogous, but specify different events. Strictly speaking, the return type of the method when specifying the termination action is unknown, as the terminating method is not specified. However, we only consider *well-formed* global types that allow us to derive the method and its return type for each termination action. We introduce formally well-formed types in Def. 7.23.

The semantics of branching is a guarded disjunction. Each of the disjuncts corresponds to one branch: the guard of the branch holds in the first state and the trace is a model for the semantics of the branch type. The guards do not depend on the state, but only on input parameters of methods and tracked values and we may safely put the evaluation of the guard at the first state. Repetition is the standard translation of the Kleene star into MSO [19] and guesses a set of indices which mark the start and end of single iterations. The translation of the inner type has to hold between any two neighboring indices. The first disjunct covers the case that the Kleene star is unrolled 0-times. The other actions specify no communication and are needed for technical reasons. In particular, for types $\mathbf{G}$ without a sequential composition, we set $gtsem(\mathbf{G}, T) = gtsem(\mathbf{G}\ .\ \mathbf{skip}, T)$.

Closed Semantics and Witnesses

The semantics *gtsem* specifies a protocol, but does *not* specify that no further (non-specified) communication occurs. Such a closed semantics *gcsem* is constructed as follows:

1. Bring $gtsem(\mathbf{GP})$ in prefix normal form, denoted $Q.\varphi$ where Q are all quantifiers and φ the formula.
2. Let $i_1, \dots, i_n$ be those logical variables explicitly specifying the events (e.g., i, j in the translation of the call action).
3. Set
$$gcsem(\mathbf{GP}) = Q.\Big(\varphi \wedge \forall k \in I.\, \big((isEvent(k) \wedge \bigwedge_{j \le n} k \neq i_j) \rightarrow [k] \doteq \mathsf{noEv}\big)\Big)$$
This specifies that every event is either explicitly specified or noEv.

Example 7.3. *Consider the protocol in Ex. 7.2 for* $\mathbf{G} = \mathbf{skip}$*. Its closed semantics is the following:*

$$\begin{aligned}\exists i \in \mathbf{I}.\, \exists j \in \mathbf{I}.\, \exists i' \in \mathbf{I}.\, \exists j' \in \mathbf{I}.\, \Big(& \\ & \forall k \in I.\, \big((isEvent(k) \wedge k \neq i \wedge k \neq j \wedge k \neq i' \wedge k \neq j') \rightarrow [k] \doteq \mathsf{noEv}\big) \wedge \\ & [i] \doteq \mathsf{invEv}(\mathbf{p}, \mathbf{q}, \mathrm{t}, \mathtt{m}, \_) \wedge [j] \doteq \mathsf{invREv}(\mathbf{q}, \mathrm{t}, \mathtt{m}, \_) \wedge \\ & i' > i \wedge [i'] \doteq \mathsf{invEv}(\mathbf{p}, \mathbf{r}, \mathrm{t}_2, \mathtt{m}_2, \_) \wedge [j'] \doteq \mathsf{invREv}(\mathbf{r}, \mathrm{t}_2, \mathtt{m}_2, \_) \wedge \dots \Big)\end{aligned}$$

We observe that in a model for $gcsem(\mathbf{GP})$ each event, which is not noEv, can be traced back to the global action that specifies its existence. This allows us to analyze all possible models by reasoning about the global protocol itself.

Fact 3. *Let* **GP** *be a global protocol and* γ *a model for* $gcsem(\mathbf{GP})$*. For each event* $\gamma[i]$ *at position* i *with* $\gamma[i] \neq \mathsf{noEv}$ *there is a global action* **G** *within* **GP** *such that the following holds*

- *The translation of* **G** *introduced a variable* j *of* **I** *type.*
- *During the successful evaluation of* $\gamma, I, \emptyset \models gcsem(\mathbf{GP})$*,* i *is assigned to to* j*.*

Definition 7.4 (Witnesses)**.** *Given the above situation, we say that* $\gamma[i]$ *is a witness for* **G***. Analogously we define states as witnesses for a global action* **G***, if the state is at a position that is assigned to the variable stemming from the translation of* **G***.*

Role Assignments

Global types are abstract descriptions of protocols. They are disconnected from the implementation by roles: each role must be assigned to an object in the implementation. We do so by providing *role assignments*. A global role assignment specifies which object is having a role in the protocol, while a local role assignment specifies which parameter of a class is having which role. Global assignments are specified by the user, local assignments are derived.

Definition 7.5 (Role Assignment)**.** *A role assignment is a partial function* $r : \mathscr{R} \mapsto \mathsf{Obj}$ *that maps roles to object names. A role assignment* r *satisfies a global protocol* **GP** *for a program* Prgm *if* $\mathbf{dom}(r) =$ {*all roles in* **GP**} *and*

$$\forall \gamma. \Big(\mathsf{Prgm} \Downarrow \gamma \to \gamma, I, \beta \models \big(\exists_{\mathbf{p} \in \mathbf{dom}(r)} \mathbf{p} \in \mathtt{D}(\mathbf{p}).\ \mathbf{p} \doteq r(\mathbf{p}) \wedge gcsem(\mathbf{GP}) \big) \Big)$$

hold. I.e., if all roles are assigned an object and every trace with the assignment is a model for the global protocol.

A local role assignment $l : \mathscr{R} \mapsto \mathsf{F}$ *is a partial function from roles to field names. A local role assignment* l *for a class* C *matches a role assignment* r *if for every role* $\mathbf{p}$*, such that* $r(\mathbf{p})$ *is not of class* C*,* $r(\mathbf{p})$ *is assigned to the parameter* $l(\mathbf{p})$ *in every object creation of* C*.*

The reader should recall that we identify the ith object creation with a variable name $\mathtt{v}_i$ and the object name $\mathtt{x}_i$. A role assignment, thus, may be written as a function to variable names: $\{\mathbf{p}_1 \mapsto \mathtt{v}_1, \dots\}$.

A local role assignment models for an object, whose parameter refers to an object implementing a role and may be represented in lFOS as

$$\mathsf{log}(l) = \bigwedge_{\mathbf{p} \in \mathbf{dom}(l)} \big(\mathsf{select}(\mathtt{heap}, l(\mathbf{p})) \doteq \mathbf{p} \big)$$

where $\mathbf{p}$ is a (free) logical variable.

Example 7.4. *Consider the following two class signatures and main block. We use Rebeca-style main blocks, so* p *may indeed reference* q*.*

```
1 class C(J a) implements I{...}
2 class D(I b) implements J{...}
3 {
4     I p = new C(q);
5     J q = new D(p);
6     p!m();
7 }
```

A role assignment r *and matching local assignments* $l_{\mathtt{C}}$ *for* C *and* $l_{\mathtt{D}}$ *for* D *are, for example the following:*

$$r = [\mathbf{p} \to \mathtt{p}, \mathbf{q} \to \mathtt{q}]$$
$$l_{\mathtt{C}} = [\mathbf{q} \mapsto \mathtt{a}]$$
$$l_{\mathtt{D}} = [\mathbf{p} \mapsto \mathtt{b}]$$

A global type is not a standalone specification: it must still refer to parameter names, field names and method names of the implementation. We could use abstract message names instead of method names and connect message names to method names as part of the specification, but refrain from doing so because (a) it offers no further insights, as we explore the needed mechanism already for roles (b) it obfuscates technical details and (c) this is standard for all specifications. E.g., the original Session Types [72] use the data types declared by the implementation.

Definition 7.6. *A program* Prgm*, a role assignment* r *and a global protocol* **GP** *match if*

- $r(\mathbf{p})$ *refers to an object creation such that the class created at* $r(\mathbf{p})$ *has the correct fields and methods according to* **GP***.*
- *There is a matching local assignment* l_C *for each class.*
- *If* $\mathbf{p}$ *sends a message to* $\mathbf{q}$ *in* **GP***, then* $r(\mathbf{q})$ *is passed as a parameter to the object implementing* $\mathbf{p}$*.*
- *The call in the main block corresponds to the one in the call actions of* $\mathbf{0}$*.*

Example 7.5. *The following classes, together with the main block and role assignments from Ex. 7.4 and the following global protocol match.*

$$\mathbf{0} \xrightarrow{t_0} \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{p} \xrightarrow{t_1} \mathbf{q}{:}\mathtt{m}_1(\!|\mathtt{y} > 0 \wedge \mathtt{t} \doteq \mathtt{y}, t_2 \textbf{ as } \mathtt{y}^*\mathtt{y}|\!) \,.\, \mathbf{p}{\downarrow}t_0(\!|\textbf{this}.\mathtt{f} > 0|\!) \,.\, \mathbf{q}{\downarrow}t_1 \,.\, \textbf{end}$$

```
1  interface I{ Unit m(); }
2  interface J{ Unit m1(Int y); }
3  class C(J a) implements I{
4     Int f = 0;
5     Unit m(){
6         a!m1(10);
7         this.f = 1;
8     }
9  }
10 class D(I b) implements J{
11     Unit m1(Int y){ skip; }
12 }
```

7.2 Well-Formedness

Some global types and protocols describe impossible protocols, which are not possible in the Active Object concurrency models. Similarly they may describe protocols that may not be statically enforced. Thus, we apply a semantic analysis to discard malformed protocols. Such a preprocessing step is common for Session Types, e.g. the linearity check in the original asynchronous session type system [72] or the well-assertedness check for choreographies [14].

Example 7.6. *Consider the following two types.*

$$\mathbf{G}_1 = \mathbf{0} \xrightarrow{t} \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{q}{\downarrow}\mathtt{t} \,.\, \textbf{end}$$
$$\mathbf{G}_2 = \mathbf{0} \to \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{p} \to \mathbf{q}{:}\mathtt{m}_2 \,.\, \mathbf{p} \to \mathbf{r}{:}\mathtt{m}_3 \,.\, \mathbf{q} \to \mathbf{r}{:}\mathtt{m}_4 \,.\, \textbf{end}$$

The first type $\mathbf{G}_1$ *does not specify possible behavior, as* $\mathtt{t}$ *identifies a process running on* $\mathbf{p}$*, not* $\mathbf{q}$*. Formally: no trace of any program is a model for gcsem*$(\mathbf{G}_1)$*. The second type* $\mathbf{G}_2$ *is not enforceable: without synchronization it is not possible to ensure that* $\mathbf{r}$ *will execute* $\mathtt{m}_3$ *before* $\mathtt{m}_4$*, even if every party behaves as specified locally.*

To detect both kinds of modeling errors, we use a three-fold analysis:

Preanalysis. We give three simple analyses to discard certain classes of global protocols, which are either impossible or not enforceable: protocols not adhering to scoping, protocols not adhering to cooperative scheduling and protocols not delegating tracked values correctly.

Projection. During projection, the generation of local from global types, we additionally discard some non-enforceable protocols, e.g., those with non-cooperative global invariants.

Postanalysis. After projection, we discard further non-enforceable protocols: Protocols which do not specify enough synchronization to ensure that methods are executed in the specified order. E.g., the pattern in $\mathbf{G}_2$ is discarded at this step.

The use of denotational semantics allows us to reason about global types *without a type system*. To do so, we connect actions with their witnesses. Before that, we introduce some short-hand notation.

Convention 7. *Let* $\mathbf{G}$ *be a global type that occurs at least twice in a global type* $\mathbf{G}'$*. We assume that each copy of* $\mathbf{G}$ *is implicitly distinguishable from the other by its position in* $\mathbf{G}'$*.*

For uniformity, we consider the initial action of global protocols as a special global type, $\mathbf{0}$ *a special role and global protocols as global types, in the context of our analyses.*

Definition 7.7 (Witness Order $<_{\mathbf{G}}$)**.** *Let* $\mathbf{G}$ *be a global type and* $\mathbf{G}_1$*,* $\mathbf{G}_2$ *two global types within. We define the witness order* $<_{\mathbf{G}}$ *in two steps.* $\mathbf{G}_1 <^*_{\mathbf{G}} \mathbf{G}_2$ *holds if one of the following conditions holds.*

- $\mathbf{G}$ *contains* $\mathbf{G}_1.\mathbf{G}_2$ *or* $\mathbf{G}_1.\mathbf{G}'.\mathbf{G}_2$ *for some* $\mathbf{G}'$
- $\mathbf{G}$ *contains* $\mathbf{G}'.\mathbf{p}\{\varphi_i : \mathbf{G}_i\}_{i\in I}.\mathbf{G}''$ *and* $\mathbf{G}_1 <^*_{\mathbf{G}'.\mathbf{G}_i.\mathbf{G}''} \mathbf{G}_2$ *for some* $i \in I$
- $\mathbf{G}$ *contains* $\mathbf{p}\{\varphi_i : \mathbf{G}_i\}_{i\in I}.\mathbf{G}'$ *and* $\mathbf{G}_1 <^*_{\mathbf{G}_i.\mathbf{G}'} \mathbf{G}_2$ *for some* $i \in I$
- $\mathbf{G}$ *contains* $\mathbf{G}'.\mathbf{p}\{\varphi_i : \mathbf{G}_i\}_{i\in I}$ *and* $\mathbf{G}_1 <^*_{\mathbf{G}'.\mathbf{G}_i} \mathbf{G}_2$ *for some* $i \in I$
- $\mathbf{G}$ *contains* $\mathbf{p}\{\varphi_i : \mathbf{G}_i\}_{i\in I}$ *and* $\mathbf{G}_1 <^*_{\mathbf{G}_i} \mathbf{G}_2$ *for some* $i \in I$

$\mathbf{G}_1 <_{\mathbf{G}} \mathbf{G}_2$ *holds if one of the following conditions holds.*

- $\mathbf{G}_1 <^*_{\mathbf{G}} \mathbf{G}_2$
- $\mathbf{G}$ *contains* $\mathbf{G}'.(\mathbf{G}'')^*_\psi.\mathbf{G}'''$ *and* $\mathbf{G}_1 <_{\mathbf{G}'.\mathbf{G}''.\mathbf{G}'''} \mathbf{G}_2$
- $\mathbf{G}$ *contains* $(\mathbf{G}'')^*_\psi.\mathbf{G}'''$ *and* $\mathbf{G}_1 <_{\mathbf{G}''.\mathbf{G}'''} \mathbf{G}_2$
- $\mathbf{G}$ *contains* $\mathbf{G}'.(\mathbf{G}'')^*_\psi$ *and* $\mathbf{G}_1 <_{\mathbf{G}'.\mathbf{G}''} \mathbf{G}_2$
- $\mathbf{G}$ *contains* $(\mathbf{G}'')^*_\psi$ *and* $\mathbf{G}_1 <_{\mathbf{G}''} \mathbf{G}_2$

With $\mathbf{G}_1 \leq_{\mathbf{G}} \mathbf{G}_2$ *and* $\mathbf{G}_1 \leq^*_{\mathbf{G}} \mathbf{G}_2$ *we denote the corresponding reflexive closure.*

We use $<^*_{\mathbf{G}}$ to reason about orders ignoring repetition. This is necessary because scoping ensures that a repetition does not introduce, for example, a tracked value that is available outside of it. Additionally to smaller examples, we use the following scenario for illustration of our analyses.

Example 7.7. *Consider a system with two components, a* GUI *and a* computation server*. The GUI gets as user input a list of data values, which need to be processed. To stay responsive, the GUI delegates the list to the computation server, and only receives updates on the intermediate results, which it may show the user. When the list of data values is empty, the computation server may reject the request. In this case the GUI must display an error. Alternatively, the computation server may proceed as in the normal case. This is*

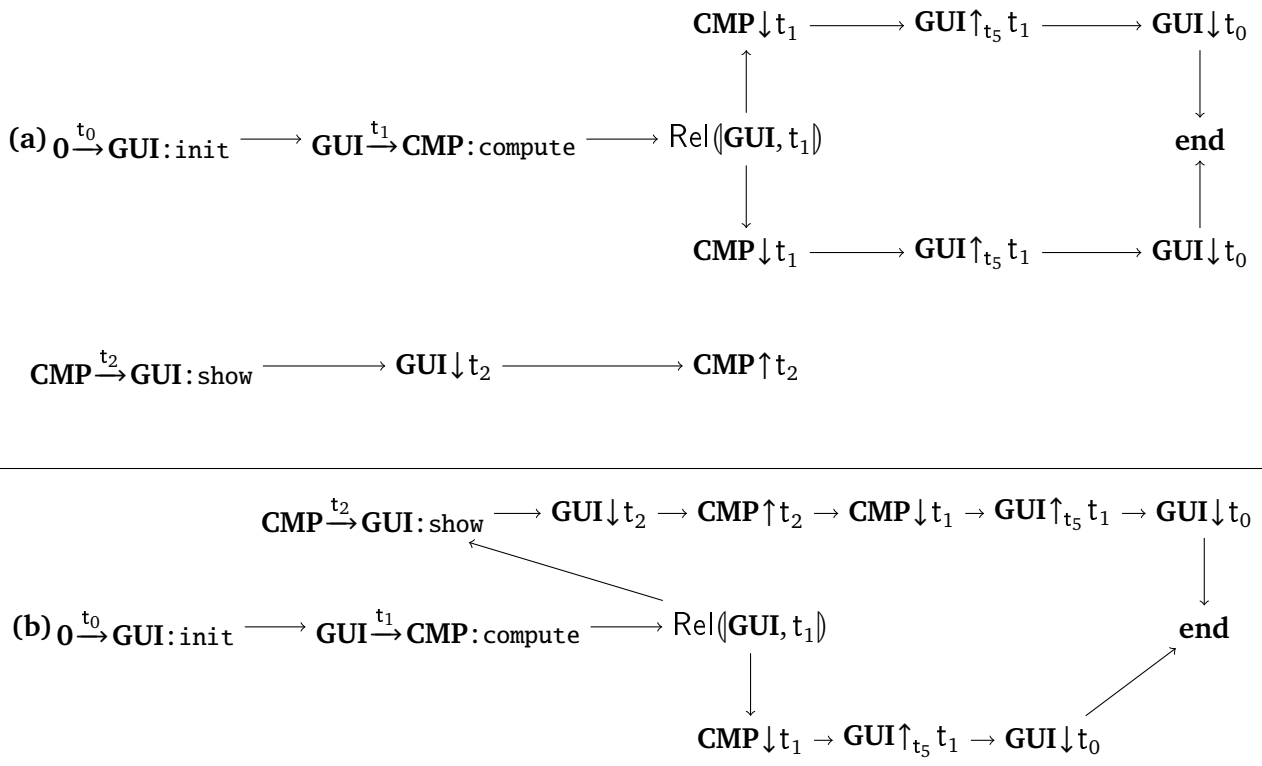


Figure 7.2: Witness orders.

modeled by the following global protocol **S**. *A return value of* `-1` *is the error code return value of the server and* `0` *the one of the GUI. Let* $\psi_{\mathbf{S}} = \mathbf{CMP}.\texttt{count} >= \texttt{0} \wedge \mathbf{GUI}.\texttt{shown} >= \texttt{0}$

$$\mathbf{S} = \mathbf{0} \xrightarrow{t_0} \mathbf{GUI}:\texttt{init} \,.\, \mathbf{GUI} \xrightarrow{t_1} \mathbf{CMP}:\texttt{compute}(\!|t_v \textbf{ as } \texttt{len(v)}|\!) \,.\, \mathsf{Rel}(\!|\mathbf{GUI}, t_1|\!) \,.\, \mathbf{CMP} \left\{ \begin{array}{ll} t_v \geq 0: & \left(\mathbf{CMP} \xrightarrow{t_2} \mathbf{GUI}:\texttt{show} \,.\, \mathbf{GUI}\downarrow t_2 \,.\, \mathbf{CMP}\uparrow t_2\right)^*_{\psi_{\mathbf{S}}} \,.\, \mathbf{CMP}\downarrow t_1(\!|\texttt{result} \doteq 0|\!) \,.\, \mathbf{GUI}\uparrow_{t_5} t_1 \,.\, \mathbf{GUI}\downarrow t_0(\!|\texttt{result} \doteq -1|\!) \\ t_v \doteq 0: & \mathbf{CMP}\downarrow t_1(\!|\texttt{result} \doteq -1|\!) \,.\, \mathbf{GUI}\uparrow_{t_5} t_1 \,.\, \mathbf{GUI}\downarrow t_0(\!|\texttt{result} \doteq 0|\!) \end{array} \right\} \,.\, \textbf{end}$$

Example 7.8. *Consider the protocol* **S** *from Ex. 7.7. The spine of* $<^*_{\mathbf{G}}$ *is the upper graph (a) in Fig. 7.2 The spine of* $<_{\mathbf{G}}$ *is the lower graph (b) in Fig. 7.2*

The following lemma justifies to reason about models by analyzing the witness order of global types:

Lemma 7.1 (Soundness of Witness Order $<_{\mathbf{G}}$)**.** *Let* $\mathbf{G}$ *be a global type and* $\mathbf{G}_1, \mathbf{G}_2$ *be two actions with* $\mathbf{G}_1 \leq_{\mathbf{G}} \mathbf{G}_2$. *If the witness(es) of* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *are issued by the same object, then the witness(es) of* $\mathbf{G}_1$ *occur before the witness(es) of* $\mathbf{G}_2$ *in any model of* $\mathbf{G}$.

Proof. See p. 181.

Scope Analysis

First, we analyze whether the global type adheres to scoping. Under scoping we understand two related properties, which are similar to the standard notion of variable scopes in programming languages.

Activity Scoping. That tracked futures are terminated after being called, every terminated future was called before[2]. Synchronization and suspension happens after the call. Protocols not adhering to activity scoping are impossible to realize.

[2] With "before" and "after" we always refer to the witness order $<_{\mathbf{G}}$ for a suitable $\mathbf{G}$.

Repetition Scoping. A tracked future introduced inside a repetition (or at the outermost level) is terminated and suspended inside the same repetition, but not inside a nested repetition within. Protocols not adhering to repetition scoping are either modeling errors (e.g., process that repeatedly terminate), or not enforceable.

Example 7.9. *Consider the following global protocols, which all contain modeling errors:*

- $\mathbf{GP}_0$ *models that a tracked future is used by two different processes.*

$$\mathbf{GP}_0 = \mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{p} \xrightarrow{\mathtt{t}_0} \mathbf{q}{:}\mathtt{m}_2 \,.\, \mathbf{q}{\downarrow}\mathtt{t}_0 \,.\, \mathbf{p}{\downarrow}\mathtt{t}_0 \,.\, \mathbf{end}$$

- $\mathbf{GP}_1$ *models that a process is started but never terminates.*

$$\mathbf{GP}_1 = \mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{end}$$

- $\mathbf{GP}_2$ *models that a process is started but never terminates, but a process that is never started is terminated.*

$$\mathbf{GP}_2 = \mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{p}{\downarrow}\mathtt{t}_1 \,.\, \mathbf{end}$$

- $\mathbf{GP}_3$ *models that* $\mathtt{m}_2$ *is called repeatedly, but terminated only once.*

$$\mathbf{GP}_3 = \mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{p}{:}\mathtt{m} \,.\, (\mathbf{p} \xrightarrow{\mathtt{t}} \mathbf{q}{:}\mathtt{m}_2)^* \,.\, \mathbf{q}{\downarrow}\mathtt{t} \,.\, \mathbf{p}{\downarrow}\mathtt{t}_0 \,.\, \mathbf{end}$$

- $\mathbf{GP}_4$ *models that* $\mathtt{m}_2$ *is called once, but terminated repeatedly.*

$$\mathbf{GP}_4 = \mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{p} \xrightarrow{\mathtt{t}} \mathbf{q}{:}\mathtt{m}_2 \,.\, (\mathbf{q}{\downarrow}\mathtt{t})^* \,.\, \mathbf{p}{\downarrow}\mathtt{t}_0 \,.\, \mathbf{end}$$

- $\mathbf{GP}_5$ *models that a future is read that does not identify a call.*

$$\mathbf{GP}_5 = \mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{p}{\uparrow}\mathtt{t} \,.\, \mathbf{p}{\downarrow}\mathtt{t}_0 \,.\, \mathbf{end}$$

We define scoping in terms of the witness order: e.g., a method must be called before being able to terminate and a future can only be read after having been resolved. Additionally, we have constraints that express that a method must *always* terminate after being called.

Definition 7.8 (Correct Scoping)**.** *A global protocol* $\mathbf{GP}$ *(or a global type), is correctly scoped if the following conditions hold:*

1. *Each tracked value is used at most once as the future in a global call action. This means that every tracked future identifies one call action.*
2. *For each* $\mathbf{p} \xrightarrow{\mathtt{t}} \mathbf{q}:\mathtt{m}$ *there exists at least one* $\mathbf{q}{\downarrow}\mathtt{t}$*. All such termination actions are after the call action and inside the same repetition:*

$$\mathbf{p} \xrightarrow{\mathtt{t}} \mathbf{q}:\mathtt{m} <^*_{\mathbf{G}} \mathbf{q}{\downarrow}\mathtt{t}$$

Additionally, we demand that there is no $\mathbf{G}'$ *with* $\mathbf{p} \xrightarrow{\mathtt{t}} \mathbf{q}:\mathtt{m} <^*_{\mathbf{G}} \mathbf{G}'$ *that is either before or after* all $\mathbf{q}{\downarrow}\mathtt{t}$*. I.e., for no* $\mathbf{G}'$*:*

$$\mathbf{q}{\downarrow}\mathtt{t} \not<^*_{\mathbf{G}} \mathbf{G}' \not<^*_{\mathbf{G}} \mathbf{q}{\downarrow}\mathtt{t} \qquad (*)$$

3. *For each* $\mathbf{q}{\downarrow}\mathtt{t}$ *there is exactly one* $\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m}$ *before in the same repetition:*

$$\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m} <^*_{\mathbf{G}} \mathbf{q}{\downarrow}\mathtt{t}$$

Additionally, we demand that there is no $\mathbf{G}'$ *with* $\mathbf{G}' <^*_{\mathbf{G}} \mathbf{q}{\downarrow}\mathtt{t}$ *that is either before or after the call*

$$\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m} \not<^*_{\mathbf{G}} \mathbf{G}' \not<^*_{\mathbf{G}} \mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m} \qquad (**)$$

4. *For each* $\mathsf{Rel}(\!|\mathbf{r},\mathtt{t}|\!)$ *there is exactly one* $\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m}$ *before and at least one* $\mathbf{q}{\downarrow}\mathtt{t}$ *in the same scope:*

$$\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m} <^*_{\mathbf{G}} \mathsf{Rel}(\!|\mathbf{r},\mathtt{t}|\!) <^*_{\mathbf{G}} \mathbf{q}{\downarrow}\mathtt{t}$$

5. *For each* $\mathbf{r}{\uparrow}\mathtt{t}$ *there is exactly one* $\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m}$ *before:*

$$\mathbf{p}\xrightarrow{\mathtt{t}}\mathbf{q}:\mathtt{m} <_{\mathbf{G}} \mathbf{r}{\uparrow}\mathtt{t}$$

In Ex. 7.9 $\mathbf{GP}_0$ is not scoped correctly because of condition 1, $\mathbf{GP}_1$ is not scoped correctly because of condition 2, $\mathbf{GP}_2, \mathbf{GP}_3, \mathbf{GP}_4$ because of the conditions 2 and 3 and $\mathbf{GP}_5$ because of the condition 5. We do not give a soundness lemma – we use Lemma 7.1 to lift scoping directly to the model level and use it in the composition theorem. The restriction on Rel is motivated by the possibility to track the currently active process in an object/role, as we discuss in the next section.

The restrictions $(*)$ and $(**)$ catch the cases that a call is only active in some branches of a choice.

Example 7.10. *Restriction* $(**)$ *models that there must be no action before a termination action catches the case where a call is executed only sometimes.*

$$\mathbf{0}\xrightarrow{\mathtt{t}_0}\mathbf{p}:\mathtt{m}\,.\,\mathbf{p}\left\{\begin{array}{c}\mathbf{p}\xrightarrow{\mathtt{t}_1}\mathbf{q}:\mathtt{m}_1\\ \mathbf{p}\xrightarrow{\mathtt{t}_2}\mathbf{q}:\mathtt{m}_2\end{array}\right\}\,.\,\mathbf{q}{\downarrow}\mathtt{t}_1\,.\,\mathbf{q}{\downarrow}\mathtt{t}_2\,.\,\mathbf{p}{\downarrow}\mathtt{t}_0\,.\,\mathbf{end}$$

Here only one of $\mathtt{m}_1$ *or* $\mathtt{m}_2$ *are specified as being called, but both are specified as being terminated in every case. In terms of the restriction, the call to* $\mathtt{m}_2$ *is before the resolving action of* $\mathtt{t}_1$ *and can, thus, be possible executed before it:*

$$\mathbf{p}\xrightarrow{\mathtt{t}_2}\mathbf{q}:\mathtt{m}_2 <_{\mathbf{G}} \mathbf{q}{\downarrow}\mathtt{t}_1$$

On the other hand, it may be executed instead of the call to $\mathtt{m}_1$*. Thus, it is not guaranteed that* $\mathtt{t}_1$ *is used for a call before being terminated:*

$$\mathbf{p}\xrightarrow{\mathtt{t}_1}\mathbf{q}:\mathtt{m}_1 \not<_{\mathbf{G}} \mathbf{p}\xrightarrow{\mathtt{t}_2}\mathbf{q}:\mathtt{m}_2 \not<_{\mathbf{G}} \mathbf{p}\xrightarrow{\mathtt{t}_1}\mathbf{q}:\mathtt{m}_1$$

The restriction for $(*)$ *is analogous, consider:*

$$\mathbf{0}\xrightarrow{\mathtt{t}_0}\mathbf{p}:\mathtt{m}\,.\,\mathbf{p}\left\{\begin{array}{c}\mathbf{p}{\downarrow}\mathtt{t}_0\\ \mathbf{skip}\end{array}\right\}\,.\,\mathbf{end}$$

Next, we discard protocols that do not adhere to the Active Object concurrency model. This simplifies further analyses, as they can assume the absence of (some) impossible patterns. This analysis is performed after checking scoping.

Example 7.11. *Consider the following global protocol:*

$$\mathbf{0} \xrightarrow{\mathsf{t}_0} \mathbf{p}{:}\mathtt{m} \,.\, \mathbf{p} \xrightarrow{\mathsf{t}} \mathbf{q}{:}\mathtt{m}_2(\!|\mathsf{t}_f \textbf{ as } \mathsf{v}|\!) \,.\, \mathbf{q}{\downarrow}\mathsf{t} \,.\, \mathbf{q}{\uparrow}\mathsf{t}_f \,.\, \mathbf{p} \xrightarrow{\mathsf{t}'} \mathbf{q}{:}\mathtt{m}_2 \,.\, \mathbf{q}{\downarrow}\mathsf{t}' \,.\, \mathbf{p}{\downarrow}\mathsf{t}_0 \,.\, \textbf{end}$$

It does not specify what process will execute the action $\mathbf{q} \uparrow \mathsf{t}_f$*. The first process on* $\mathbf{q}$ *(identified by* t*) is already terminated, the second process on* $\mathbf{q}$ *(identified by* t'*) has not started yet. No other process on* $\mathbf{q}$ *is specified. The global protocol is, thus, a modeling error and cannot be realized. It fails to specify according to the cooperative scheduling in the Active Object concurrency model.*

These modeling errors are errors in modeling *activity*. It does not adhere to the Active Object model, where at every point in time, at most one process is active in an object and an object cannot perform any actions if no process is active. We call such global types *uncooperative*. Formally, we require the notion of an activity path to detect such modeling errors. An activity path tracks where the process identified by a tracked future is active.

Definition 7.9 (Activity Paths)**.** *Given a global type* $\mathbf{G}$*, the activity path of a tracked future* t *on* $\mathbf{p}$ *is a sequence of actions within* $\mathbf{G}$*, written* $\langle \mathbf{G}_1, \ldots, \mathbf{G}_n \rangle$*, with even length, such that the following holds.*

- *For each* i *with* $0 < i < n$ *the sequence is ordered with the witness order:* $\mathbf{G}_i <_{\mathbf{G}} \mathbf{G}_{i+1}$
- $\mathbf{G}_1 = \mathbf{q} \xrightarrow{\mathsf{t}} \mathbf{p}{:}\mathtt{m}(\!|\varphi, \mathsf{C}|\!)$ *for some role* $\mathbf{q}$*, method* $\mathtt{m}$ *and specification* $(\!|\varphi, \mathsf{C}|\!)$*.*
- $\mathbf{G}_n = \mathbf{p}{\downarrow}\mathsf{t}(\!|\varphi|\!)$ *for some specification* $(\!|\varphi|\!)$*.*
- *For all even* i *with* $1 < i < n$

  $$\mathbf{G}_i = \mathsf{Rel}(\!|\mathbf{p}, \mathsf{t}', \varphi_1, \varphi_2|\!) \qquad \mathbf{G}_{i+1} = \mathbf{r}{\downarrow}\mathsf{t}'(\!|\varphi_3|\!)$$

  for some tracked future t'*, role* $\mathbf{r}$ *and specifications* $\varphi_1, \varphi_2, \varphi_3$*.*
- *For all even* i *with* $1 < i \leq n$*, there is no* $\mathsf{Rel}(\!|\mathbf{p}, \mathsf{t}', \varphi_1, \varphi_2|\!)$ *with* $\mathbf{G}_{i-1} <_{\mathbf{G}} \mathsf{Rel}(\!|\mathbf{p}, \mathsf{t}', \varphi_1, \varphi_2|\!) <_{\mathbf{G}} \mathbf{G}_i$

A tracked future may have multiple activity paths.

We call actions $\mathbf{G}'$ *with* $\mathbf{G}_{i-1} \leq_{\mathbf{G}} \mathbf{G}' <_{\mathbf{G}} \mathbf{G}_i$ *for some even* i covered by t*. We call the actions* $\mathbf{G}_i$ *for some even* i terminally covered by t*.*

Example 7.12. *Fig. 7.3 shows a protocol, the three activity paths of the contained tracked futures and the covered actions. Grayed out actions are covered, but not part of the path. Note that some covered actions are covered by tracked futures from a different object and are not relevant for checking cooperativeness.*

The elements of the activity path with an odd index are those where the process identified by t is scheduled and the ones with an even index are those where it is descheduled. Intuitively, every action between (w.r.t. $<_{\mathbf{G}}$) an odd and an even index can assume that it is executed by the process identified by t. We check that every action is covered exactly once per object — this models that each action can be assigned to exactly one responsible process[3]. No action is skipped and no action is assigned twice.

[3] Or to a set of processes, if the action is inside a repetition, which are all identified by another future in the semantics, but only one tracked future in the type. This is ensured by scoping, which checks that such processes are started and finished inside their repetition of the protocol.

$$\text{Protocol:}\quad \mathbf{0}\xrightarrow{\mathtt{t}_0}\mathbf{p}{:}\mathtt{m}_0\,.\,\mathbf{p}\xrightarrow{\mathtt{t}_1}\mathbf{q}{:}\mathtt{m}_1\,.\,\mathsf{Rel}(\!|\mathbf{p},\mathtt{t}_1|\!)\,.\,\mathbf{q}\xrightarrow{\mathtt{t}_2}\mathbf{r}{:}\mathtt{m}_2\,.\,\mathsf{Rel}(\!|\mathbf{q},\mathtt{t}_2|\!)\,.\,\mathbf{r}{\downarrow}\mathtt{t}_2\,.\,\mathbf{q}{\downarrow}\mathtt{t}_1\,.\,\mathbf{p}{\downarrow}\mathtt{t}_0\,.\,\mathbf{end}$$

$$\text{Activity path of } \mathtt{t}_0\text{:}\quad \left\langle \mathbf{0}\xrightarrow{\mathtt{t}_0}\mathbf{p}{:}\mathtt{m}_0,\ \mathbf{p}\xrightarrow{\mathtt{t}_1}\mathbf{q}{:}\mathtt{m}_1,\ \mathsf{Rel}(\!|\mathbf{p},\mathtt{t}_1|\!)\ ,\ \mathbf{q}{\downarrow}\mathtt{t}_1,\ \mathbf{p}{\downarrow}\mathtt{t}_0 \right\rangle$$

$$\text{Activity path of } \mathtt{t}_1\text{:}\quad \left\langle \mathbf{p}\xrightarrow{\mathtt{t}_1}\mathbf{q}{:}\mathtt{m}_1,\ \mathsf{Rel}(\!|\mathbf{p},\mathtt{t}_1|\!),\ \mathbf{q}\xrightarrow{\mathtt{t}_2}\mathbf{r}{:}\mathtt{m}_2,\ \mathsf{Rel}(\!|\mathbf{q},\mathtt{t}_2|\!),\ \mathbf{r}{\downarrow}\mathtt{t}_2,\ \mathbf{q}{\downarrow}\mathtt{t}_1 \right\rangle$$

$$\text{Activity path of } \mathtt{t}_2\text{:}\quad \left\langle \mathbf{q}\xrightarrow{\mathtt{t}_2}\mathbf{r}{:}\mathtt{m}_2,\ \mathsf{Rel}(\!|\mathbf{q},\mathtt{t}_2|\!),\ \mathbf{r}{\downarrow}\mathtt{t}_2 \right\rangle$$

Figure 7.3: Activity Paths. Grayed out actions are covered, but not part of the path.

Definition 7.10 (Cooperation)**.** *A global protocol (or type) is* cooperative *if the following holds*

- *Every synchronization action* $\mathbf{p}\uparrow\mathtt{t}'$ *and every call action* $\mathbf{p}\to\mathbf{q}:\mathtt{m}$ *is covered by exactly one tracked future on* $\mathbf{p}$*.*
- *No call action* $\mathbf{p}\to\mathbf{q}:\mathtt{m}$ *is covered by any tracked future on* $\mathbf{q}$*.*
- *Every branching by* $\mathbf{p}$ *is non-terminally covered by exactly one tracked future on* $\mathbf{p}$*. By this we understand that the action with an edge in the witness graph caused by this branching is non-terminally covered by exactly one tracked future on* $\mathbf{p}$*.*

A tracked future may have multiple activity paths. Indeed, if its termination is after a choice, it must have multiple paths to be correctly scoped. The reader should recall that by convention we can distinguish syntactically equal actions at different positions in a global type.

Example 7.13. *We illustrate cooperativeness by the following examples:*

- *Ex. 7.11 is not cooperative:* $\mathbf{q}\uparrow\mathtt{t}_f$ *is not covered on* $\mathbf{q}$*.*
- *The following type is not cooperative, because (1)* $\mathbf{q}\uparrow\mathtt{t}_f$ *is covered twice and (2)* $\mathbf{p}\xrightarrow{\mathtt{t}_2}\mathbf{q}:\mathtt{m}_2(\!|\mathtt{t}_f\ \mathbf{as}\ \mathtt{v}|\!)$ *is covered on* $\mathbf{q}$*.*

  $$\mathbf{0}\xrightarrow{\mathtt{t}_0}\mathbf{p}:\mathtt{m}\,.\,\mathbf{p}\xrightarrow{\mathtt{t}_1}\mathbf{q}:\mathtt{m}_2(\!|\mathtt{t}_f\ \mathbf{as}\ \mathtt{v}|\!)\,.\,\mathbf{p}\xrightarrow{\mathtt{t}_2}\mathbf{q}:\mathtt{m}_2(\!|\mathtt{t}_f\ \mathbf{as}\ \mathtt{v}|\!)\,.\,\mathbf{q}\uparrow\mathtt{t}_f\,.\,\mathbf{q}{\downarrow}\mathtt{t}_2\,.\,\mathbf{q}{\downarrow}\mathtt{t}_1\,.\,\mathbf{p}{\downarrow}\mathtt{t}_0\,.\,\mathbf{end}$$

  The first error models that there are two process on $\mathbf{q}$ *which can be responsible to read from* $\mathtt{t}_f$*. The second error models that* $\mathbf{q}$ *starts a new process (for* $\mathtt{t}_2$*), despite already having an active process.*
- *The following type is not cooperative, because the branching is not covered by* $\mathbf{p}$ *(which is computed by checking that the termination action before is not covered by* $\mathbf{p}$*).*

  $$\mathbf{0}\xrightarrow{\mathtt{t}_0}\mathbf{p}:\mathtt{m}\,.\,\mathbf{p}\xrightarrow{\mathtt{t}_1}\mathbf{q}:\mathtt{m}_2\,.\,\mathbf{p}{\downarrow}\mathtt{t}_0\,.\,\mathbf{p}\left\{\begin{array}{l}\mathbf{q}{\downarrow}\mathtt{t}_1\\ \mathbf{q}{\downarrow}\mathtt{t}_1\end{array}\right\}\,.\,\mathbf{end}$$

  It is, however, correctly scoped and $\mathtt{t}_1$ *has* two *activity paths.*

We use Lemma 7.1 directly in the later composition proof to lift the activity path to events.

History-Sensitivity

Finally, we check that if an action has the obligation to guarantee some property over a set of tracked values, then it has access to these futures.

Example 7.14. *Consider the following global protocol.*

$$\mathbf{0} \xrightarrow{\mathtt{t_p}} \mathbf{p}{:}\mathtt{m}_1(\!|\mathtt{t}\ \mathbf{as}\ \mathtt{v}|\!)\ .\ \mathbf{p} \xrightarrow{\mathtt{t_q}} \mathbf{q}{:}\mathtt{m}_2\ .\ \mathbf{p}{\downarrow}\mathtt{t_p}\ .\ \mathbf{q}{\downarrow}\mathtt{t_q}(\!|\mathtt{result} > \mathtt{t}|\!)\ .\ \mathbf{end}$$

The process executing $\mathtt{m}_2$ *on* $\mathbf{q}$ *must guarantee that its result is bigger than* $\mathtt{t}$ *— but it is not specified that* $\mathtt{t}$ *was ever send to this process. The global protocol is thus, not* history-sensitive*. A fix would be to specify that* $\mathtt{t}$ *is sent to* $\mathbf{q}$ *(for some suitable parameter of* $\mathtt{m}_2$*):*

$$\mathbf{0} \xrightarrow{\mathtt{t_p}} \mathbf{p}{:}\mathtt{m}_1(\!|\mathtt{t}\ \mathbf{as}\ \mathtt{v}|\!)\ .\ \mathbf{p} \xrightarrow{\mathtt{t_q}} \mathbf{q}{:}\mathtt{m}_2(\!|\mathtt{t}\ \mathbf{as}\ \mathtt{w}|\!)\ .\ \mathbf{p}{\downarrow}\mathtt{t_p}\ .\ \mathbf{q}{\downarrow}\mathtt{t_q}(\!|\mathtt{result} > \mathtt{t}|\!)\ .\ \mathbf{end}$$

We use activity paths to keep track of known tracked values of a process. To handle tracked values communicated via futures, we characterize which postconditions communicate tracked values.

Definition 7.11. *A terminating action* $\mathbf{p}{\downarrow}\mathtt{t}(\!|\varphi|\!)$ *communicates* $\mathtt{t}'$ *if* $\varphi \rightarrow \mathtt{result} \doteq \mathtt{t}'$ *is valid.*

Example 7.15. *Consider the following type. It models that* $\mathbf{p}$ *sends some value* $\mathtt{t}_f$ *as parameter* $\mathtt{v}$ *to* $\mathbf{q}$ *and* $\mathbf{q}$ *returns* $\mathtt{t}_f$*. Then,* $\mathbf{p}$ *reads* $\mathtt{t}_1$ *and tracks the return value of* $\mathbf{q}$ *in* $\mathtt{t}_2$ *and must prove that the return value of* $\mathbf{q}$ *is indeed* $\mathtt{t}_f$*.*

$$\mathbf{GP_{ret}} = \mathbf{0} \xrightarrow{\mathtt{t_0}} \mathbf{p}{:}\mathtt{m}\ .\ \mathbf{p} \xrightarrow{\mathtt{t_1}} \mathbf{q}{:}\mathtt{m}_2(\!|\mathtt{t}_f\ \mathbf{as}\ \mathtt{v}|\!)\ .\ \mathbf{q}{\downarrow}\mathtt{t}_1(\!|\mathtt{result} \doteq \mathtt{t}_f|\!)\ .\ \mathbf{p}{\uparrow}_{\mathtt{t}_2}\mathtt{t}_1\ .\ \mathbf{p}{\downarrow}\mathtt{t}_0(\!|\mathtt{t}_2 \doteq \mathtt{t}_f|\!)\ .\ \mathbf{end}$$

The terminating action $\mathbf{q}{\downarrow}\mathtt{t}_1(\!|\mathtt{result} \doteq \mathtt{t}_f|\!)$ *communicates* $\mathtt{t}_f$*.*

The extended activity path of a tracked value is the activity path including the covered actions on the object resolving the future in question. The inclusion of covered actions is necessary because they add known tracked values.

Definition 7.12 (Extended Activity Path)**.** *Let* $\mathbf{G}$ *be a global type. Given an activity path* $\langle \mathbf{G}_1, \ldots, \mathbf{G}_n \rangle$ *for a tracked future* $\mathtt{t}$ *on role* $\mathbf{p}$ *inside* $\mathbf{G}$*, the extended activity path* $\langle \mathbf{G}'_1, \ldots, \mathbf{G}'_m \rangle$ *is obtained by ordering all elements of the activity paths and all covered actions* of $\mathbf{p}$ *according to* $<_{\mathbf{G}}$*.*

For any global action (and type) $\mathbf{G}$ *we write* $act(\mathbf{G}, \mathbf{p})$ *for the tracked future containing* $\mathbf{G}$ *on an extended activity path on* $\mathbf{p}$*.*

We now define history-sensitivity by examining all used tracked values and distinguishing three cases: (1) the action using the tracked value is introducing it (2) it is introduced by an action on the same extended activity path or (3) there is a future read before, that reads from a tracked future, whose resolving action communicates the tracked value in question.

Definition 7.13 (History-Sensitivity)**.** *Let* $\mathtt{t}$ *be a tracked future on* $\mathbf{p}$*, and* $\langle \mathbf{G}_i \rangle_{i \in I}$ *an extended activity path within some global type* $\mathbf{G}$*. The activity path is history-sensitive if one of the following conditions hold for any tracked value* $\mathtt{t}_v$ *within this path with* $\mathtt{t}_v \neq \mathtt{t}$*. Let* $\mathbf{G}_v$ *be the action where* $\mathtt{t}_v$ *is used for the first time (w.r.t.* $<_{\mathbf{G}}$*).*

- $\mathtt{t}_v$ *is globally introduced by* $\mathbf{G}_v$*. I.e.,* $\mathbf{G}_v \leq_{\mathbf{G}} \mathbf{G}'$ *for any action* $\mathbf{G}'$ *that uses* $\mathtt{t}_v$ *even outside the activity path in question.*
- $\mathtt{t}_v$ *is globally introduced by some* $\mathbf{G}_j$ *in the activity path with* $\mathbf{G}_j <_{\mathbf{G}} \mathbf{G}_v$
- *There is a receiving action* $\mathbf{p}{\uparrow}_{\mathtt{t}''}\mathtt{t}' <_{\mathbf{G}} \mathbf{G}_v$ *in the activity path such that there is a termination action* $\mathbf{q}{\downarrow}\mathtt{t}'(\!|\varphi|\!) <_{\mathbf{G}} \mathbf{p}{\uparrow}_{\mathtt{t}''}\mathtt{t}'$ *outside the activity path that communicates* $\mathtt{t}_v$

A global type, (or global protocol), is history-sensitive *if all its extended activity paths are history-sensitive.*

Example 7.16. *Protocol* $\mathbf{GP_{ret}}$ *in Ex. 7.15 is history-sensitive. The following are the extended activity paths:*

$$\textit{Extended activity path of } \mathtt{t}_0\textit{: } \mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{p}{:}\mathtt{m} \;.\; \mathbf{p} \xrightarrow{\mathtt{t}_1} \mathbf{q}{:}\mathtt{m}_2(\!|\mathtt{t}_f \textbf{ as } \mathtt{v}|\!) \;.\; \mathbf{p}{\uparrow_{\mathtt{t}_2}}\mathtt{t}_1 \;.\; \mathbf{p}{\downarrow}\mathtt{t}_0(\!|\mathtt{t}_2 \doteq \mathtt{t}_f|\!)$$

$$\textit{Extended activity path of } \mathtt{t}_1\textit{: } \mathbf{p} \xrightarrow{\mathtt{t}_1} \mathbf{q}{:}\mathtt{m}_2(\!|\mathtt{t}_f \textbf{ as } \mathtt{v}|\!) \;.\; \mathbf{q}{\downarrow}\mathtt{t}_1(\!|\mathtt{result} \doteq \mathtt{t}_f|\!)$$

- *The use of* $\mathtt{t}_f$ *and* $\mathtt{t}_1$ *in the call action* $\mathbf{p} \xrightarrow{\mathtt{t}_1} \mathbf{q}{:}\mathtt{m}_2(\!|\mathtt{t}_f \textbf{ as } \mathtt{v}|\!)$ *in the activity path of* $\mathtt{t}_0$ *conforms to Def. 7.13 because it introduces* $\mathtt{t}_f$ *and* $\mathtt{t}_1$.
- *The use of* $\mathtt{t}_2$ *in the termination action of the activity path of* $\mathtt{t}_0$ *conforms to Def. 7.13 because it was introduced by the reading action before.*
- *The use of* $\mathtt{t}_f$ *in the termination action of the activity path of* $\mathtt{t}_0$ *conforms to Def. 7.13 because it was made available by the reading action before, that reads from the tracked future* $\mathtt{t}_2$ *that communicates* $\mathtt{t}_f$ and *because it was introduced by the call action (to* $\mathbf{q}$*) before.*
- *The use of* $\mathtt{t}_f$ *in the path of* $\mathtt{t}_1$ *conforms to Def. 7.13 because it was introduced by the call action at the beginning of the path.*

Definition 7.14. *A global protocol is a* projection candidate *if it is scoped, cooperative and history-sensitive.*

The analyses are performed in the given order: scoping, cooperativeness and history-sensitivity. Scenario **S** from Ex. 7.7 is a projection candidate.

Temporal Satisfiability.

History-Sensitivity was first proposed by Bocchi et al. [14] for channel-based systems. Bocchi et al. also propose *Temporal Satisfiability* that expresses that tracked values may not be specified to have no possible value.

Example 7.17. *Consider the following global protocol:*

$$\mathbf{0} \xrightarrow{\mathtt{t_p}} \mathbf{p}{:}\mathtt{m}_1(\!|\mathtt{t} > 0, \mathtt{t} \textbf{ as } \mathtt{v}|\!) \;.\; \mathbf{p} \xrightarrow{\mathtt{t_q}} \mathbf{q}{:}\mathtt{m}_2(\!|\mathtt{t} < 0, \mathtt{t} \textbf{ as } \mathtt{w}|\!) \;.\; \mathbf{p}{\downarrow}\mathtt{t_p} \;.\; \mathbf{q}{\downarrow}\mathtt{t_q} \;.\; \textbf{end}$$

This extends the specification above with predicates that demand that $\mathtt{t}$ *is strictly positive and strictly negative. Obviously, this protocol is not realizable, as its semantics can have no model generated by any program.*

We do not check temporal satisfiability at the global type. Instead, it results in local types that cannot be proven. In the example above, the proof obligation of $\mathtt{m}_2$ will not be provable as it has to prove at one point a formula of the form

$$\mathtt{t} > 0 \rightarrow \mathtt{t} < 0$$

7.3 Local Types

Local types are the local view of an endpoint on a global type. In our case, the notion of endpoints is two-fold: for one, objects, targeted by roles and processes, targeted through their tracked value. The process is identified by a tracked future, the object by a role. Contrary to prior work [83, 82], there is no special local type to specify on object-level: Each local type is *method-local*. In this section we introduce the syntax and semantics of local types, as well as the fragment that we use as the syntax of the behavioral type. This is *not* the behavioral specification — the behavioral specification in section 7.5 uses an extension of local types with tracking information.

Definition 7.15 (Local Types)**.** *Let* φ *range over lFOS formulas,* C *over tracking constraints,* $\mathtt{t}$ *over tracked values,* $\mathbf{p}$ *over roles and* $\mathtt{m}$ *over method names. The syntax of local types and local protocols is defined by the grammar in Fig. 7.4.*

A receiving action $?\mathtt{m}(\!|\varphi, \mathsf{C}|\!)$ models that a process executing `m` starts in a state where φ holds (φ may also specify the parameters) with tracking constraints C. A calling action $\mathbf{p}!_{\mathsf{t}}\mathtt{m}(\!|\varphi, \mathsf{C}|\!)$ is a call to role $\mathbf{p}$ on method `m` with t tracking the generated future. φ specifies the state in the moment of the call and the call parameters. C may keep track of sent values. E.g., for a method with signature `Unit m(Int i)`, a call `o!m(this.f+10)` and a tracking constraint t **as** `i`, the tracking value t tracks the value of **this**`.f+10` in the moment of the call.

The terminating action $\mathsf{Put}\ \varphi$ models the termination of the currently active process in a state where φ holds, where φ specifies state and the return value. The suspension action $\mathsf{Susp}(\!|\mathsf{t}, \varphi, \varphi|\!)$ is analogous to its global counterpart. The empty action **skip** denotes no visible action and is only added for technical reasons in an intermediate state.

The analogue to global choice from the point of view of the choosing role is active choice $\oplus$. Passive choice $\&^{\mathsf{t}_1}_{\mathsf{t}_2}\{\varphi_i : \mathbf{L}_i\}_{i\in I}$ models that the process reads from the future tracked by t_1 and depending on which condition φ_i holds for the read value (tracked by t_2) it follows the local type $\mathbf{L}_i$. The conditions do not overlap. They are *not* the conditions from the global type, but the termination conditions of the process making the choice. The other actions are straightforward.

Convention 8. *Analogously to Conv. 6 we omit empty specifications and the tracked future of a calling action if it is not used anywhere, and conditions in the active and passive choices if all of them are* true.

Example 7.18. *The local protocols in Fig. 7.5 are the local protocols for the views of the three methods on the global protocol* **S** *in Ex. 7.7.*

Additionally to active choice ($\mathbf{S}_{\mathsf{t}_1}$, a process makes the choice how to continue) and passive choice (in $\mathbf{S}_{\mathsf{t}_0}$, a process reacts to a choice made by another process by reading the choice from a future), $\mathbf{S}_{\mathsf{t}_2}$ shows another kind of choice: If `show` is called, then the first branch has been chosen, but this choice is not communicated via the future of the choosing process, but by being called. Thus, $\mathbf{S}_{\mathsf{t}_2}$ is aware of the chosen branch. Similarly, we allow methods to be called in different branches, as long as the preconditions enable them to deduce in which branch they are. This mechanism, and that the above local protocols are indeed the local views on **S** is derived by projection, is introduced in the next section. Before, we give the formal semantics of local types.

Definition 7.16. *The semantics of local types and protocols is shown in Fig. 7.6.*

The semantics of local types follows the general ideas of the semantics of global types, but is more simple because it does not specify that events must be ordered for each role separately. The semantics of local protocols takes as a parameter a role, the semantics of local types is parametric in role and

$\mathbf{LP} ::= ?\mathtt{m}(\!|\varphi, \mathsf{C}|\!)\ .\ \mathbf{L}$	Receiving Action
$\mathbf{L} ::= \mathbf{p}!_{\mathsf{t}}\mathtt{m}(\!|\varphi, \mathsf{C}|\!)$	Calling Action
$\mid \mathsf{Put}\ \varphi$	Terminating Action
$\mid \mathsf{Susp}(\!|\mathsf{t}, \varphi, \varphi|\!)$	Suspension Action
$\mid \mathbf{skip}$	Empty Action
$\mid \mathbf{L}\ .\ \mathbf{L}$	Sequential Composition
$\mid \oplus\{\varphi_i : \mathbf{L}_i\}_{i\in I}$	Active Choice
$\mid \&^{\mathsf{t}}_{\mathsf{t}}\{\varphi_i : \mathbf{L}_i\}_{i\in I}$	Passive Choice
$\mid (\mathbf{L})^*_{\varphi}$	Local Repetition

Figure 7.4: Syntax of Local Protocols and Local Types.

$$\mathbf{S}_{t_0} = ?_{t_0}\texttt{init} \,.\, \mathbf{CMP}!_{t_1}\texttt{compute}(\!|t_v \textbf{ as } \texttt{len(v)}|\!).\mathsf{Susp}(\!|t_1|\!).\&^{t_1}_{t_5}\left\{\begin{array}{l} t_5 \doteq\ \ 0 : \mathsf{Put}\ \texttt{result} \doteq -1 \\ t_5 \doteq -1 : \mathsf{Put}\ \texttt{result} \doteq 0 \end{array}\right\} . \textbf{skip}$$

$$\mathbf{S}_{t_1} = ?_{t_1}\texttt{compute}(\!|t_v \textbf{ as } \texttt{len(v)}|\!). \oplus \left\{\begin{array}{ll} t_v \geq 0 : & \left(\mathbf{GUI}!_{t_2}\texttt{show}.\&^{t_2}\{\textbf{skip}\}\right)^*_{\textbf{this}.\texttt{count}\geq 0}.\mathsf{Put}\ \texttt{result} \doteq 0 \\ t_v \doteq 0 : & \mathsf{Put}\ \texttt{result} \doteq -1 \end{array}\right\} .\textbf{skip}$$

$$\mathbf{S}_{t_2} = ?_{t_2}\texttt{show} \,.\, \mathsf{Put}$$

Figure 7.5: Local Protocols for **S** in Ex. 7.7.

tracked future. This is only needed later to simplify composition arguments. As an abbreviation we use $\mathsf{evBound}(i, j)$ that expresses that every event before j that is not at index i is not a communication event, i.e., it is noEv.

- The semantics of the receiving action is that the first event (at index 2, as the first element of any trace is a state) is an invocation reaction event on the correct method and that the parameter satisfies the precondition. The local type **L** has to be adhered to on the rest of the trace — thus, its semantics is relativized to the trace that is following the invocation reaction event.
- The semantics of the calling action is that there are positions i, j, such that i is the index of a invocation event as specified. Note that without a role assignment, **q** is a free variable. Variable j is a position chosen such that is after i and the rest of the trace models the following type. It must be after i and it is ensured that all further events are part of the trace considered by the relativization. The tracking constraint is handled analogously to global types with the auxiliary Q function.
- The semantics of the suspending action follows the above pattern of specifying a position i that describes the specified event and a position j used for relativization, such that $i < j$ and j does not cut off any event described neither by i (or a constant offset from i) nor the relativized continuation.
- The semantics of the empty action describes traces without any visible events. If **skip** is continued, the semantics of the continuation has to hold.
- Semantics of active choice is a disjunction over all possibilities, such that there is always a position j before any event where the guards are evaluated.
- Semantics of passive choice is mostly analogous, but (1) at j must be a resolving reaction event and (2) the branches are joined by a *conjunction*, mirroring the intuition that all possibilities must be considered.
- Semantics of termination cuts off possibly trailing actions and semantics of repetition is analogous to the global case.

7.4 Projection

Projection generates local protocols for methods participating in a global protocol. After the projection itself, we perform two further steps:

Causality Check We derive the causality graph to check whether the order of methods described in a global protocol is always enforced by the protocol in the Active Object concurrency model. This rules out that they do not contain "enough" synchronization.

Knowledge Propagation The loop invariant of global types may be not visible to a local type, because the whole type is inside the repetition. However, these types must still be aware of the invariant — in particular, they may be required to establish it upon termination.

$$evBound(i,j) = \forall k \in \mathbf{I}.\ \Big(\big(k < j \wedge i \neq k \wedge \mathsf{isEvent}(k)\big) \rightarrow [k] \doteq \mathsf{noEv}\Big)$$

$$lsem\big(?\mathtt{m}(\!|\varphi, \mathsf{C}|\!)\ .\ \mathbf{L}, \mathbf{p}\big) =$$
$$\exists \mathsf{t} \in \mathsf{Fut}.\ \exists \mathtt{e} \in \mathtt{D}^{\mathtt{p}}(\mathtt{m}).\ Q\Big(\mathsf{C}, [2] \doteq \mathsf{invREv}(\mathbf{p}, \mathsf{t}, \mathtt{m}, \mathtt{e}) \wedge [3] \vdash \varphi(\mathtt{e}) \wedge lsem^{\mathsf{t}}_{\mathbf{p}}\big(\mathbf{L}, \mathbf{dom}(\mathsf{C}) \cup \{\mathsf{t}\}\big)\big[idx \in \mathbf{I} \setminus idx \geq 3\big], \emptyset\Big)$$

$$lsem^{f}_{\mathbf{p}}\big(\mathbf{q}!_{\mathsf{t}}\mathtt{m}(\!|\varphi, \mathsf{C}|\!)\ .\ \mathbf{L}, T\big) =$$
$$\exists i \in \mathbf{I}.\ \exists j \in \mathbf{I}.\ \exists \mathtt{e} \in \mathtt{D}^{\mathtt{p}}(\mathtt{m}).\ \exists \mathsf{t} \in \mathsf{Fut}.\ \Big(i < j \wedge Q\Big([i] \doteq \mathsf{invEv}(\mathbf{p}, \mathbf{q}, \mathsf{t}, \mathtt{m}, \mathtt{e}) \wedge [i+1] \vdash \varphi(\mathtt{e}) \wedge evBound(i,j) \wedge lsem^{f}_{\mathbf{p}}\big(\mathbf{L}, T \cup \mathbf{dom}(\mathsf{C}) \cup \{\mathsf{t}\}\big)\big[idx \in \mathbf{I} \setminus idx \geq j\big]\Big)\Big)$$

$$lsem^{f}_{\mathbf{p}}\big(\mathsf{Susp}(\!|\mathsf{t}, \varphi_1, \varphi_2|\!).\mathbf{L}, T\big) =$$
$$\exists i \in \mathbf{I}.\ \exists j \in \mathbf{I}.\ j - i > 4 \wedge evBound(i+4, j) \wedge [i] \doteq \mathsf{suspEv}(\mathbf{p}, f, \mathsf{t}) \wedge [i+4] \doteq \mathsf{suspREv}(\mathbf{p}, f, \mathsf{t}) \wedge [i+1] \vdash \varphi_1 \wedge [i+3] \vdash \varphi_2 \wedge lsem^{f}_{\mathbf{p}}(\mathbf{L}, T)[idx \in \mathbf{I} \setminus idx \geq j]$$

$$lsem^{f}_{\mathbf{p}}(\mathbf{skip}, T) = \forall i \in \mathbf{I}.\mathsf{isEvent}(i) \rightarrow [i] \doteq \mathsf{noEv} \qquad lsem^{f}_{\mathbf{p}}(\mathbf{skip}.\mathbf{L}, T) = lsem^{f}_{\mathbf{p}}(\mathbf{L}, T)$$

$$lsem^{f}_{\mathbf{p}}\big(\oplus\{\varphi_i : \mathbf{L}_i\}_{i \in I}\ .\ \mathbf{L}, T\big) =$$
$$\exists j \in \mathbf{I}.\ \Big(\forall k \in \mathbf{I}.\ evBound(j,k) \wedge \bigvee_{i \in I}\Big(\big([j] \vdash \varphi_i \wedge lsem^{f}_{\mathbf{p}}(\mathbf{L}_i\ .\ \mathbf{L}, T)\big)\big[idx \in \mathbf{I} \setminus idx \geq j\big]\Big)\Big)$$

$$lsem^{f}_{\mathbf{p}}\big(\&^{\mathsf{t}_1}_{\mathsf{t}_2}\{\varphi_i : \mathbf{L}_i\}_{i \in I}\ .\ \mathbf{L}, T\big) =$$
$$\exists j \in \mathbf{I}.\ \Big(\forall k \in \mathbf{I}.\ evBound(j,k) \wedge \exists \mathsf{t}_2 \in \mathtt{D}(\mathsf{t}_1).\ [j] \doteq \mathsf{futREv}(\mathbf{p}, \mathsf{t}_1, \_, \mathsf{t}_2) \wedge \bigwedge_{i \in I}\Big(\big([j+1] \vdash \varphi_i(\mathsf{t}_2) \rightarrow lsem^{f}_{\mathbf{p}}(\mathbf{L}_i\ .\ \mathbf{L}, T \cup \{\mathsf{t}_2\})\big)\big[idx \in \mathbf{I} \setminus idx \geq j\big]\Big)\Big)$$

$$lsem^{f}_{\mathbf{p}}\big(\mathsf{Put}\ \varphi\ .\ \mathbf{L}, T\big) = lsem^{f}_{\mathbf{p}}(\mathsf{Put}\ \varphi, T) =$$
$$\exists i \in \mathbf{I}.\ \forall j \in \mathbf{I}.\ \exists \mathtt{e} \in \mathtt{D}(\mathtt{m}).\ \Big(\big(\mathsf{isEvent}(j) \wedge i \neq j\big) \rightarrow \big(j = \mathsf{noEv} \wedge [i] \doteq \mathsf{futEv}(\mathbf{p}, f, \_, \mathtt{e}) \wedge [i+1] \vdash \varphi(\mathtt{e})\big)\Big)$$

$$lsem^{f}_{\mathbf{p}}\big((\mathbf{L}_1)^{*}_{\varphi}\ .\ \mathbf{L}_2, T\big) =$$
$$lsem^{f}_{\mathbf{p}}(\mathbf{L}_2, T) \vee \exists I \subset \mathbf{I}.\ \Big($$
$$\forall i, j \in I.\ \big(i < j \rightarrow \nexists k \in I.\ i < k < j\big) \rightarrow lsem^{f}_{\mathbf{p}}(\mathbf{L}_1, T)[idx \in \mathbf{I} \setminus i \leq idx \leq j]$$
$$\wedge \forall i \in I.\ [i] \models \varphi$$
$$\wedge \exists min \in I.\ \forall i \in I.\ i \geq min \wedge \forall i \in \mathbf{I}.(i < min \wedge isEvent(i)) \rightarrow [i] \doteq \mathsf{noEv}$$
$$\wedge \exists max \in I.\ \forall i \in I.\ i \leq max \wedge lsem^{f}_{\mathbf{p}}\big(\mathbf{L}_2, T\big)[idx \in \mathbf{I} \setminus idx \geq max]\Big)$$

Figure 7.6: Semantics of Local Protocols and Types. $\mathtt{D}(\mathsf{t})$ denotes the data type of t.

We perform both operations with the help of the causality graph, which we connect to the witness order to reason about its adequacy. This is an advancement over prior approaches [83, 82], which performed propagation on an additional auxiliary representation.

In the following, we only consider projection candidates.

Definition 7.17. *The projection of a projection candidate* $\mathbf{G}$ *on a tracked future* t *on* $\mathbf{x}$ *is a local type* $\mathbf{L}$ *defined by the judgment*

$$\mathbf{G} \leadsto^{\mathbf{x}}_{\mathsf{t}} \mathbf{L}$$

in Fig. 7.7. Role $\mathbf{x}$ *is the role of* t*, i.e., the receiver in the call action that makes* t *a tracked future.* $\mathbf{L}$ *is the resulting local type.*

We use an auxiliary partial function $\mathsf{divide}(\psi, \mathsf{t})$ that maps global lFOS formulas to local lFOS formulas as follows: if ψ can be written as a conjunction $\bigwedge_{\mathbf{p}} \psi_{\mathbf{p}}$, such that each $\psi_{\mathbf{p}}$ *only* references $\mathbf{p}$ in heap selections, then $\mathsf{divide}(\psi, \mathsf{t}) = \psi_{\mathbf{p}}$, for the role $\mathbf{p}$ that computes t[4]. Otherwise, the partial function is not defined and propagation fails. The rules work as follows:

- Rules (skip) and (end) end projection with a **skip** action. We discuss such final actions below in Conv. 9.
- Rule (put-other) covers the case that the termination action in question is the one of another tracked future and ignores the global termination action.
- Rule (put-self) covers the case that the termination action in question is the one of the target tracked future and produces a local termination action. The trailing global type is ignored, as in projection candidates we have the guarantee that t is never active again (i.e., its activity path ends here)
- Rule (call-caller) checks that the currently active tracked future on $\mathbf{p}$ is the target tracked future and produces a calling action.
- Rule (call-callee) covers the case that the target tracked future is generated here and produces a receiving action. The precondition $\hat{\varphi}$ is generated by replacing all field accesses with fresh logical variables and existentially quantifying over them. E.g., $\mathtt{v} \geq \mathbf{this}.\mathtt{f}$ becomes $\exists f \in \mathtt{D}(\mathtt{f}).\ \mathtt{v} \geq f$.
- Rule (call-other) is for the other cases: the global action is not relevant for the target tracked future.
- Rule (rel-self) and (rel-other) are an analogous pair to (put-self) and (put-other).
- There are three rules for repetition. (loop-inner) covers the case where the tracked future is active only within the repetition. This *drops* the repetition for the local type. (loop-next) covers the case where the tracked future is not active within the repetition. Finally, (loop-outer) covers the cases where the target tracked value is active within and outside the repetition. It also checks the cooperativeness of global loop invariants as discussed in the section above. If divide is not defined, projection fails.
- (get-self) projects a synchronization action on a passive choice with trivial guard. The sole branch is the projection of the trailing global type. This trivial passive choice may be resolved into a non-trivial passive choice, if it is projected in the context of a global choice. The mechanism is explained in the description of the (choice-passive) rule.
- (get-other) is analogous to (put-other).
- (choice-equal) drops the choice if the target tracked future behaves the same in all branches.

[4] I.e., for the receiving role in the call action that introduces t. This can be computed in advance to projection and is omitted for brevity's sake.

- (choice-active) produces an active choice if the target tracked future is the currently active one on the choosing role.
- (choice-within) drops the choice if the target tracked future is only active in one the the branches.
- Finally, (choice-passive) generates non-trivial passive choice as follows: It is checked that the choosing process is in another role that **x** or has another active tracked future. I.e., the target tracked future reacts to a choice.

  The choosing tracked future is projected in each branch and must end in a termination action[5]. The termination conditions are the guard of the passive choice. It is checked explicitly that these guard do not overlap.

  The target tracked future is projected in each branch and must start with a trivial passive choice *that reads from the choosing tracked future*. The branches of the produced passive choice are the branches of the trivial passive choices for each branch.

Convention 9. *We remove* **skip** *actions from sequential composition whenever possible after projection. This does not change the semantics of any local type.*

Example 7.19. *The global protocol* **S** *from Ex. 7.7 is indeed projected on the local types in Ex. 7.18. The derivation of the projection of* $\mathtt{t}_0$ *is shown in Fig. 7.9 and Fig. 7.10.*

We observe the following effects:

- *The conditions of the active choice do overlap, but the conditions of the passive choice do not. As discussed, this is intentional:* **CMP** *is making the choice, thus, it has the freedom of non-determinism.*[6] **GUI** *must react in its passive choice and must know the* externally *chosen branch exactly.*
- *The local type of* $\mathtt{t}_2$ *is not aware of being repeated — as every process is wrapped in one repetition, no single process can observe the fact that the method is called repeatedly. This information is visible, however, to the* object*.*
- *The suspension action is necessary in* **S***, but it would not be needed if* `show` *would be called on a third role* **r***. Consider the global protocols in Fig. 7.8.*

  While $\mathbf{GP}_{\mathbf{ok}}$ *is a projection candidate,* $\mathbf{GP}_{\mathbf{fail}}$ *is* not*, because it is not cooperative:* **GUI** *cannot start execution of* `show`*, because the process of* `init` *is still specified as being active.*

Causality Graph

We now check that the projected types ensure that there is only one possible order of visible events[7] To do so, we use a *causality graph*, which is also used for propagation.

The causality graph contains all local action of all projected local types and the causality relation: a node has a directed edge to another node, if its witness must precede the witness of the second node. Additionally, the graph contains nodes to handle repetition and branching. These nodes have no witnesses, but have multiple outgoing or in-going edges — they can be interpreted as control-flow nodes, the following analysis is indeed inspired by control-flow graph based analysis.

Definition 7.18. *Let* **GP** *be a projection candidate, such that all projections are defined. Its causality graph* $\mathfrak{C}(\mathbf{GP}) = (V, E)$ *is defined as in Fig. 7.11. An action (or composed type) can occur multiple times in a local type, but each such copy has its own node. The edges mirror the structure of* **G** *in terms of the local types resulting from projection.*

[5] The extension where it may end in several termination actions is trivial.

[6] In the specification, the implementation is deterministic.

[7] Considering repetitions, there is one possible order per number of iterations, we formalize this later.

$$\text{(skip)}\ \frac{}{\mathbf{skip} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{skip}} \qquad \text{(end)}\ \frac{}{\mathbf{end} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{skip}}$$

$$\text{(put-other)}\ \frac{\mathsf{t} \neq \mathsf{t}' \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\mathbf{p}{\downarrow}\mathsf{t}'(\!|\varphi|\!).\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}} \qquad \text{(put-self)}\ \frac{\mathsf{t}' = \mathsf{t}}{\mathbf{p}{\downarrow}\mathsf{t}'(\!|\varphi|\!).\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathsf{Put}\ \varphi}$$

$$\text{(call-caller)}\ \frac{act\big(\mathbf{p} \xrightarrow{\mathsf{t}'} \mathbf{q}{:}\mathtt{m}(\!|\varphi, \mathsf{C}|\!), \mathbf{p}\big) = \mathsf{t} \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\mathbf{p} \xrightarrow{\mathsf{t}'} \mathbf{q}{:}\mathtt{m}(\!|\varphi, \mathsf{C}|\!).\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{q}!_{\mathsf{t}'}\mathtt{m}(\!|\varphi, \mathsf{C}|\!).\mathbf{L}} \qquad \text{(call-callee)}\ \frac{\mathsf{t}' = \mathsf{t} \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\mathbf{p} \xrightarrow{\mathsf{t}'} \mathbf{q}{:}\mathtt{m}(\!|\varphi, \mathsf{C}|\!).\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} ?\mathtt{m}(\!|\hat{\varphi}, \mathsf{C}|\!).\mathbf{L}}$$

$$\text{(call-other)}\ \frac{\mathsf{t}' \neq \mathsf{t} \text{ and } \mathbf{x} \neq \mathbf{p} \text{ or } act\big(\mathbf{p} \xrightarrow{\mathsf{t}'} \mathbf{q}{:}\mathtt{m}(\!|\varphi, \mathsf{C}|\!), \mathbf{p}\big) \neq \mathsf{t} \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\mathbf{p} \xrightarrow{\mathsf{t}'} \mathbf{q}{:}\mathtt{m}(\!|\varphi, \mathsf{C}|\!).\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}$$

$$\text{(rel-self)}\ \frac{act\big(\mathsf{Rel}(\!|\mathbf{p}, \mathsf{t}', \varphi, \varphi'|\!), \mathbf{p}\big) = \mathsf{t} \qquad \mathbf{p} = \mathbf{x} \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\mathsf{Rel}(\!|\mathbf{p}, \mathsf{t}', \varphi, \varphi'|\!).\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathsf{Susp}(\!|\mathsf{t}', \varphi, \varphi'|\!).\mathbf{L}}$$

$$\text{(rel-other)}\ \frac{act\big(\mathsf{Rel}(\!|\mathbf{p}, \mathsf{t}', \varphi, \varphi'|\!), \mathbf{p}\big) \neq \mathsf{t} \text{ or } \mathbf{p} \neq \mathbf{x} \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\mathsf{Rel}(\!|\mathbf{p}, \mathsf{t}', \varphi, \varphi'|\!).\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}$$

$$\text{(loop-outer)}\ \frac{\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L} \qquad \mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}' \qquad \mathbf{L} \neq \mathbf{skip} \qquad \mathbf{L}' \neq \mathbf{skip} \qquad \mathsf{divide}(\psi, \mathsf{t}) = \varphi}{\big(\mathbf{G}\big)^*_{\psi}.\mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \big(\mathbf{L}\big)^*_{\varphi}.\mathbf{L}'}$$

$$\text{(loop-inner)}\ \frac{\mathsf{t} \text{ is introduced within } \mathbf{G} \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\big(\mathbf{G}\big)^*_{\psi}.\mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}} \qquad \text{(loop-next)}\ \frac{\mathbf{G}.\mathbf{skip} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{skip} \qquad \mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\big(\mathbf{G}\big)^*_{\psi}.\mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}$$

$$\text{(get-self)}\ \frac{\mathbf{p} = \mathbf{x} \qquad act\big(\mathbf{p}{\uparrow}_{\mathsf{t}''}\mathsf{t}', \mathbf{x}\big) = \mathsf{t} \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\mathbf{p}{\uparrow}_{\mathsf{t}''}\mathsf{t}'.\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \&_{\mathsf{t}''}^{\mathsf{t}'}\{\mathsf{true}{:}\mathbf{L}\}}$$

$$\text{(get-other)}\ \frac{\mathbf{p} \neq \mathbf{x} \text{ or } act\big(\mathbf{p}{\uparrow}_{\mathsf{t}''}\mathsf{t}', \mathbf{x}\big) \neq \mathsf{t} \qquad \mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}{\mathbf{p}{\uparrow}_{\mathsf{t}''}\mathsf{t}'.\mathbf{G} \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}}$$

$$\text{(choice-equal)}\ \frac{\text{for all } i \in I\colon \mathbf{G}_i \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L} \qquad \mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}' \qquad \mathbf{p} \neq \mathbf{x} \text{ or } act\big(\mathbf{p}\{\varphi_i{:}\mathbf{G}_i\}_{i\in I}, \mathbf{p}\big) \neq \mathsf{t}}{\mathbf{p}\{\varphi_i{:}\mathbf{G}_i\}_{i\in I}.\mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}.\mathbf{L}'}$$

$$\text{(choice-active)}\ \frac{\text{for all } i \in I\colon \mathbf{G}_i \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}_i \qquad \mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}' \qquad \mathbf{p} = \mathbf{x} \qquad act\big(\mathbf{p}\{\varphi_i{:}\mathbf{G}_i\}_{i\in I}, \mathbf{p}\big) = \mathsf{t}}{\mathbf{p}\{\varphi_i{:}\mathbf{G}_i\}_{i\in I}.\mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \oplus\{\varphi_i{:}\mathbf{L}_i\}_{i\in I}.\mathbf{L}'}$$

$$\text{(choice-within)}\ \frac{\mathbf{G}_j \text{ is introduced for some } j \in I \qquad \mathbf{G}_j \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L} \qquad \mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}'}{\mathbf{p}\{\varphi_i{:}\mathbf{G}_i\}_{i\in I}.\mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}.\mathbf{L}'}$$

$$\text{(choice-passive)}\ \frac{\begin{array}{c}\text{for all } i \in I\colon \mathbf{G}_i \rightsquigarrow_{\mathsf{t}'}^{\mathbf{p}} \mathbf{L}'_i.\mathsf{Put}\ \varphi'_i \qquad \text{all } \varphi_i \text{ do not pairwise overlap} \\ \text{for all } i \in I\colon \mathbf{G}_i \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \&_{\mathsf{t}'}^{\mathsf{t}''}\{\mathsf{true}{:}\mathbf{L}_i\} \qquad \mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \mathbf{L}' \qquad \mathbf{p} \neq \mathbf{x} \text{ or } act\big(\mathbf{p}\{\varphi_i{:}\mathbf{G}_i\}_{i\in I}, \mathbf{p}\big) \neq \mathsf{t}\end{array}}{\mathbf{p}\{\varphi_i{:}\mathbf{G}_i\}_{i\in I}.\mathbf{G}' \rightsquigarrow_{\mathsf{t}}^{\mathbf{x}} \&_{\mathsf{t}'}^{\mathsf{t}''}\{\varphi'_i{:}\mathbf{L}_i\}_{i\in I}.\mathbf{L}'}$$

Figure 7.7: Projection Rules. *act* (**G**,**x**) denotes the currently active tracked future according to Def. 7.12 with the current argument of the projection as **G**.

$$\mathbf{GP_{ok}} = \mathbf{GP} = \mathbf{0} \xrightarrow{t_0} \mathbf{GUI}\colon \texttt{init} \;.\; \mathbf{GUI} \xrightarrow{t_1} \mathbf{CMP}\colon \texttt{compute}(\!|\mathsf{t_v}\ \textbf{as}\ \texttt{len(v)}|\!)$$
$$. \mathbf{CMP} \left\{ \begin{array}{ll} \mathsf{t_v} \geq 0 \colon & \left(\mathbf{CMP} \xrightarrow{t_2} \mathbf{CMP}\colon \texttt{show} \;.\; \mathbf{CMP}{\downarrow}\mathsf{t}_2 \;.\; \mathbf{CMP}{\uparrow}\mathsf{t}_2\right)^* \;.\; \mathbf{CMP}{\downarrow}\mathsf{t}_1(\!|\texttt{result} \doteq 0|\!) \;.\; \mathbf{GUI}{\uparrow_{t_5}}\mathsf{t}_1 \;.\; \mathbf{GUI}{\downarrow}\mathsf{t}_0(\!|\texttt{result} \doteq -1|\!) \\ \mathsf{t_v} \doteq 0 \colon & \mathbf{CMP}{\downarrow}\mathsf{t}_1(\!|\texttt{result} \doteq -1|\!) \;.\; \mathbf{GUI}{\uparrow_{t_5}}\mathsf{t}_1 \;.\; \mathbf{GUI}{\downarrow}\mathsf{t}_0(\!|\texttt{result} \doteq 0|\!) \end{array} \right\} . \mathbf{end}$$

$$\mathbf{GP_{fail}} = \mathbf{0} \xrightarrow{t_0} \mathbf{GUI}\colon \texttt{init} \;.\; \mathbf{GUI} \xrightarrow{t_1} \mathbf{CMP}\colon \texttt{compute}(\!|\mathsf{t_v}\ \textbf{as}\ \texttt{len(v)}|\!)$$
$$. \mathbf{CMP} \left\{ \begin{array}{ll} \mathsf{t_v} \geq 0 \colon & \left(\mathbf{CMP} \xrightarrow{t_2} \mathbf{GUI}\colon \texttt{show} \;.\; \mathbf{GUI}{\downarrow}\mathsf{t}_2 \;.\; \mathbf{CMP}{\uparrow}\mathsf{t}_2\right)^* \;.\; \mathbf{CMP}{\downarrow}\mathsf{t}_1(\!|\texttt{result} \doteq 0|\!) \;.\; \mathbf{GUI}{\uparrow_{t_5}}\mathsf{t}_1 \;.\; \mathbf{GUI}{\downarrow}\mathsf{t}_0(\!|\texttt{result} \doteq -1|\!) \\ \mathsf{t_v} \doteq 0 \colon & \mathbf{CMP}{\downarrow}\mathsf{t}_1(\!|\texttt{result} \doteq -1|\!) \;.\; \mathbf{GUI}{\uparrow_{t_5}}\mathsf{t}_1 \;.\; \mathbf{GUI}{\downarrow}\mathsf{t}_0(\!|\texttt{result} \doteq 0|\!) \end{array} \right\} . \mathbf{end}$$

Figure 7.8: Variations of Global Protocols.

Example 7.20. *Consider again* $\mathbf{S}$ *in Ex. 7.7. Fig. 7.12 shows* $\mathfrak{C}(\mathbf{S})$*, with state specifications abbreviated. The dashed lines are the edges introduced by the last three clauses in the definition above.*

Causal-Coherence

We require some technical definitions to define a coherent causality graph.

Definition 7.19 (Cover)**.** *A node* v *is choice-covered by* v' *if there is a path from* v' *to* v *and* v' *has the form* $(\mathbf{p}, \mathsf{t}, \&(\mathsf{t}'))$*,* $(\mathbf{p}, \mathsf{t}, *)$ *or* $(\mathbf{p}, \mathsf{t}, \oplus)$*. A path is choice-covered by* v' *is one of its nodes its choice-covered by* v'*. A set of paths* S *is fully choice-covered by* v *if for every branch going out from* v*, there is a path in* S *that is choice covered by* v *through this branch.*

Definition 7.20 (Causal-Coherent)**.** *Let* $\mathcal{V}(\mathbf{GP}, \mathbf{p}) = \langle v_1, \ldots, v_n \rangle$ *be all nodes of the form* $(\mathbf{p}, \mathsf{t}, ?_{\mathsf{t}}\mathtt{m})$ *and* $(\mathbf{p}, \mathsf{t}', \mathsf{Susp}^{\mathsf{in}}_{\mathsf{t}''})$ *for a given* $\mathbf{p}$ *in* $\mathfrak{C}(\mathbf{GP})$ *and some* $\mathsf{t}, \mathsf{t}', \mathsf{t}'', \mathtt{m}$*. With each* $v \in \mathcal{V}(\mathbf{GP}, \mathbf{p})$ *we associate the global type* $\mathbf{G}_v$*, whose projection generated* v*.*

A projection candidate $\mathbf{GP}$ *is causal-coherent if the following holds:*

- *If* $v, v' \in \mathcal{V}(\mathbf{GP}, \mathbf{p})$ *are two nodes, such that* $\mathbf{G}_v <_{\mathbf{G}} \mathbf{G}_{v'}$ *and there is no* $v'' \in \mathcal{V}(\mathbf{GP}, \mathbf{p})$ *with* $\mathbf{G}_v <_{\mathbf{G}} \mathbf{G}_{v''} <_{\mathbf{G}} \mathbf{G}_{v'}$*, then there is a non-empty set of paths in* $\mathfrak{C}(\mathbf{GP})$ *from* v *to* v'*. If there is any* $w \neq *$ *that is a choice-cover for any element of any path, but not for* v*, then the set must be fully choice-covered by* w*. If there are multiple choice covers, then this condition has to hold for any of them.*
- *If* $v \in \mathcal{V}(\mathbf{GP}, \mathbf{p})$ *is choice-covered by a repetition node* $(\mathbf{p}, \mathsf{t}, *)$*, but is not within the repetition*[8] *, then removing all nodes within the repetition must result in a causal-coherent graph for this* v*.*
- *If* $v \in \mathcal{V}(\mathbf{GP}, \mathbf{p})$ *is within a repetition, then there is a set of paths from* v *to itself going through this repetition node, for which the above conditions hold.*

We give two examples where the above conditions are not adhered to.

Example 7.21. *Consider the four projection candidates and their causality graphs in Fig. 7.13.* $\mathbf{GP}^{\mathbf{branch}}_{\mathbf{fail}}$ *and* $\mathbf{GP}^{\mathbf{rep}}_{\mathbf{fail}}$ *are both not causal-coherent.*

- $\mathbf{GP}^{\mathbf{branch}}_{\mathbf{fail}}$ *does not read from the tracked future* t_2 *in one of the branches — in this case the order of* $\mathtt{m}_2$ *and* $\mathtt{m}_3$ *on* $\mathbf{q}$ *is not fixed. The projection candidate* $\mathbf{GP}^{\mathbf{branch}}_{\mathbf{ok}}$ *shows how to fix the protocol. The blue path is the path in* $\mathbf{GP}^{\mathbf{branch}}_{\mathbf{fail}}$ *— the node for reading from* t_1 *is choice covered by a node not part of any path. The red path is the one added by* $\mathbf{GP}^{\mathbf{branch}}_{\mathbf{ok}}$*.*
- $\mathbf{GP}^{\mathbf{rep}}_{\mathbf{fail}}$ *does not read from* t_1 *if the repetition is not repeated at all (remember that we allow zero executions of the repetition). The fixed projection candidates ensures that the tracked future is read at least once.*

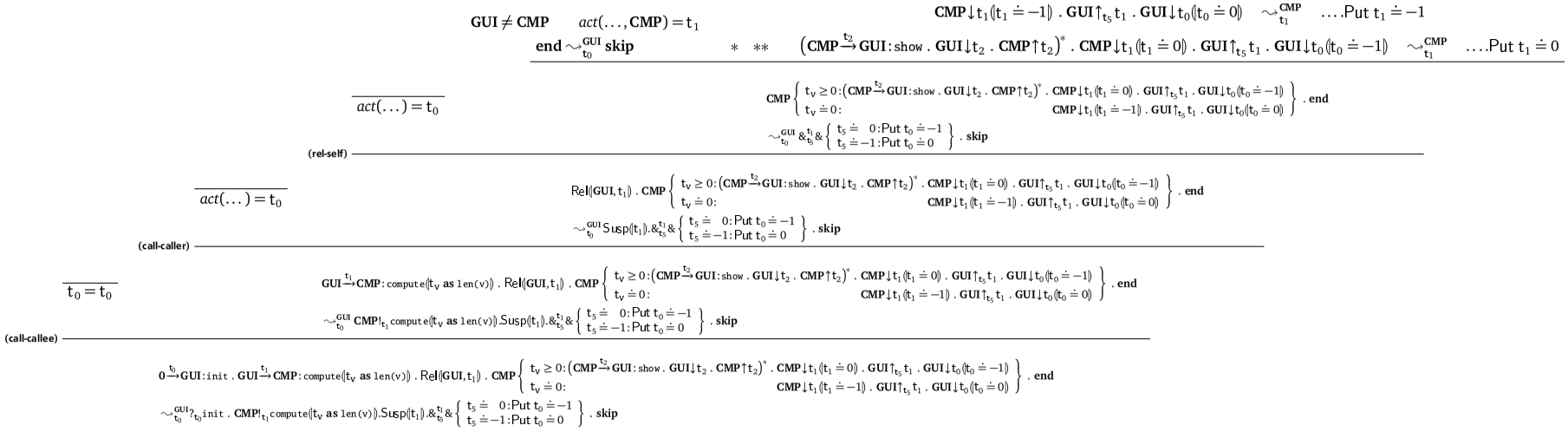


Figure 7.9: A sample derivation for projection

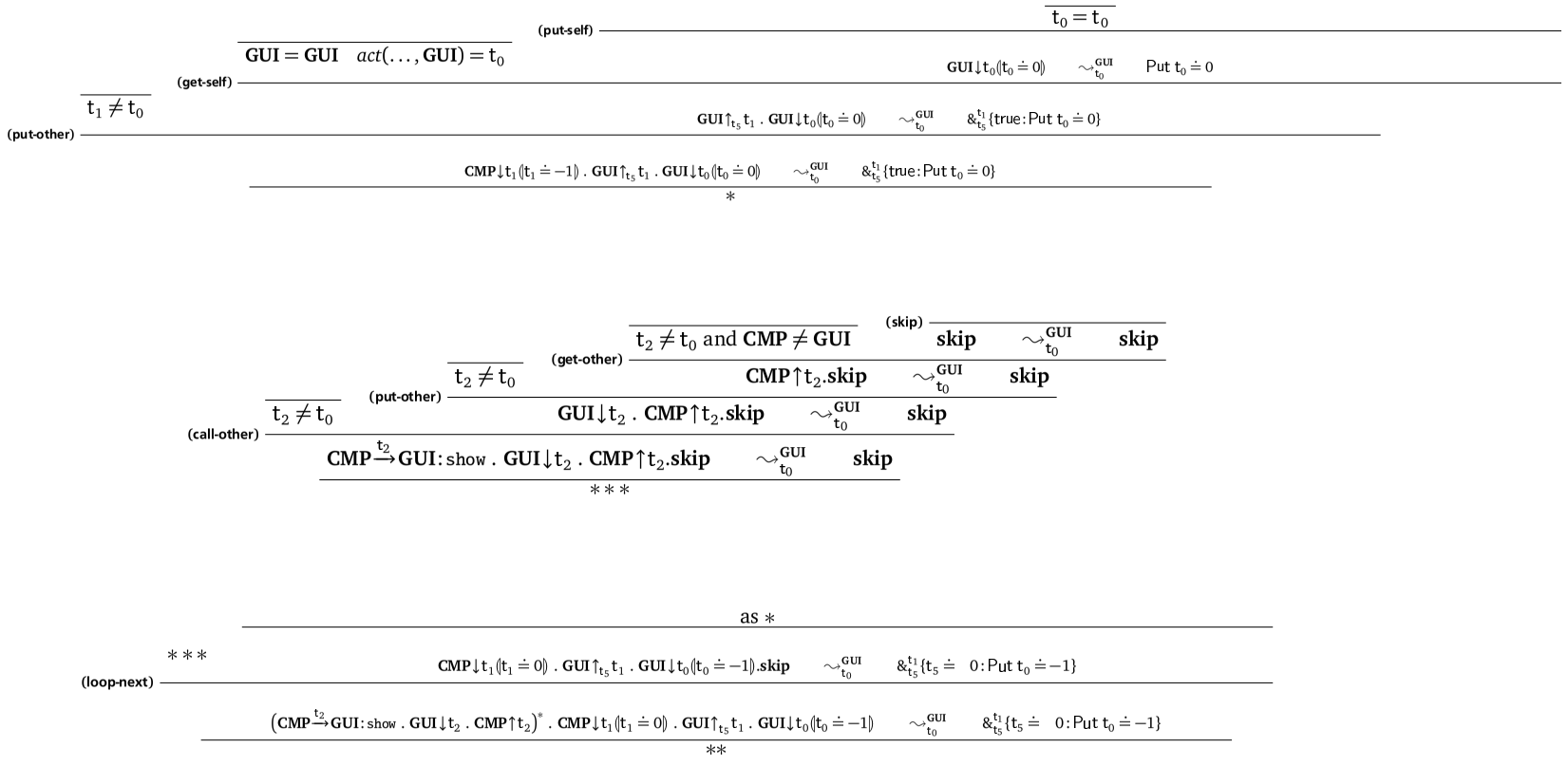


Figure 7.10: A sample derivation for projection (cont.)

$$
\begin{aligned}
V = \ & \{(\mathbf{p}, \mathrm{t}, \mathbf{L}) \mid \mathbf{L} \text{ is an action, but not a suspension, } \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}' \text{ is defined and } \mathbf{L}' \text{ contains } \mathbf{L}\} \\
& \cup \{(\mathbf{p}, \mathrm{t}, \mathsf{Susp}^{\mathsf{in}}_{\mathrm{t}'}) \mid \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}' \text{ contains } \mathsf{Susp}(\!|\mathrm{t}', \varphi, \varphi'|\!)\} \\
& \cup \{(\mathbf{p}, \mathrm{t}, \mathsf{Susp}^{\mathsf{out}}_{\mathrm{t}'}) \mid \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}' \text{ contains } \mathsf{Susp}(\!|\mathrm{t}', \varphi, \varphi'|\!)\} \\
& \cup \{(\mathbf{p}, \mathrm{t}, \&(\mathrm{t}')) \mid \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}' \text{ contains } \&^{\mathrm{t}'}_{\mathrm{t}''}\{\varphi_i : \mathbf{L}_i\}_{i \in I} \text{ for some } \varphi_i, \mathbf{L}_i\} \\
& \cup \{(\mathbf{p}, \mathrm{t}, \oplus) \mid \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}' \text{ contains } \oplus\{\varphi_i : \mathbf{L}_i\}_{i \in I} \text{ for some } \varphi_i, \mathbf{L}_i\} \\
& \cup \{(\mathbf{p}, \mathrm{t}, *) \mid \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}' \text{ contains } (\mathbf{L})^*_\varphi \text{ for some } \varphi, \mathbf{L}\}
\end{aligned}
$$

$$
\begin{aligned}
\big((\mathbf{p}, \mathrm{t}, \mathbf{L}), (\mathbf{p}, \mathrm{t}, \mathbf{L}')\big) \in E &\iff \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}'' \text{ contains } \mathbf{L}.\mathbf{L} \\
\big((\mathbf{p}, \mathrm{t}, \mathsf{Susp}^{\mathsf{in}}_{\mathrm{t}'}), (\mathbf{p}, \mathrm{t}, \mathbf{L})\big) \in E &\iff \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}' \text{ contains } \mathsf{Susp}(\!|\mathrm{t}', \varphi, \varphi'|\!).\mathbf{L} \\
&\qquad \text{where } \mathsf{Susp}(\!|\mathrm{t}', \varphi, \varphi'|\!) \text{ is the action generating } \mathsf{Susp}^{\mathsf{in}}_{\mathrm{t}'} \\
\big((\mathbf{p}, \mathrm{t}, \mathbf{L}), (\mathbf{p}, \mathrm{t}, \mathsf{Susp}^{\mathsf{out}}_{\mathrm{t}'})\big) \in E &\iff \mathbf{GP} \leadsto^{\mathrm{x}}_{\mathrm{t}} \mathbf{L}' \text{ contains } \mathbf{L}.\mathsf{Susp}(\!|\mathrm{t}', \varphi, \varphi'|\!) \\
&\qquad \text{where } \mathsf{Susp}(\!|\mathrm{t}', \varphi, \varphi'|\!) \text{ is the action generating } \mathsf{Susp}^{\mathsf{out}}_{\mathrm{t}'} \\
\big((\mathbf{p}, \mathrm{t}, *), v\big) \in E &\iff v \text{ is the first action in the repetition that caused } (\mathbf{p}, \mathrm{t}, *) \\
\big(v, (\mathbf{p}, \mathrm{t}, *)\big) \in E &\iff v \text{ is the last action in the repetition that caused } (\mathbf{p}, \mathrm{t}, *) \\
\big(v, (\mathbf{p}, \mathrm{t}, *)\big) \in E &\iff v \text{ is the first action after the repetition that caused } (\mathbf{p}, \mathrm{t}, *) \\
\big((\mathbf{p}, \mathrm{t}, \mathbf{L}), (\mathbf{p}, \mathrm{t}, \mathbf{L}')\big) \in E &\iff \mathbf{L} \text{ is } \oplus \text{ or } \& \text{ and } \mathbf{L}' \text{ is the first action in one of the branches} \\
\big((\mathbf{p}, \mathrm{t}, \mathbf{q}!_{\mathrm{t}'}\mathtt{m}), (\mathbf{q}, \mathrm{t}', ?_{\mathrm{t}'}\mathtt{m})\big) \in E & \\
\big((\mathbf{p}, \mathrm{t}, \mathsf{Put}\ \varphi), (\mathbf{q}, \mathrm{t}', \&(\mathrm{t}))\big) \in E & \\
\big((\mathbf{p}, \mathrm{t}, \mathsf{Put}\ \varphi), (\mathbf{q}, \mathrm{t}', \mathsf{Susp}^{\mathsf{in}}_{\mathrm{t}})\big) \in E &
\end{aligned}
$$

Figure 7.11: Definition of a causality graph.

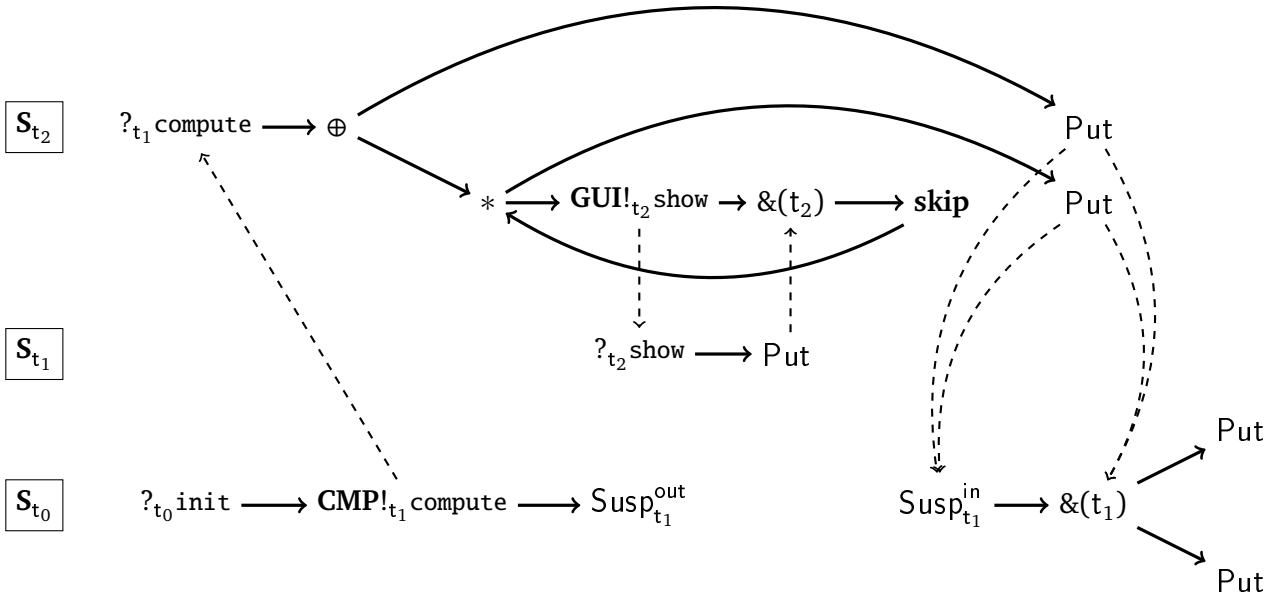
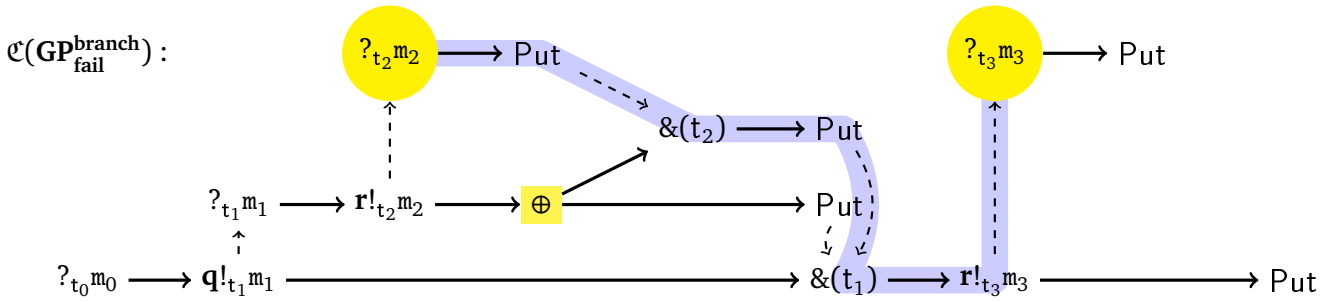


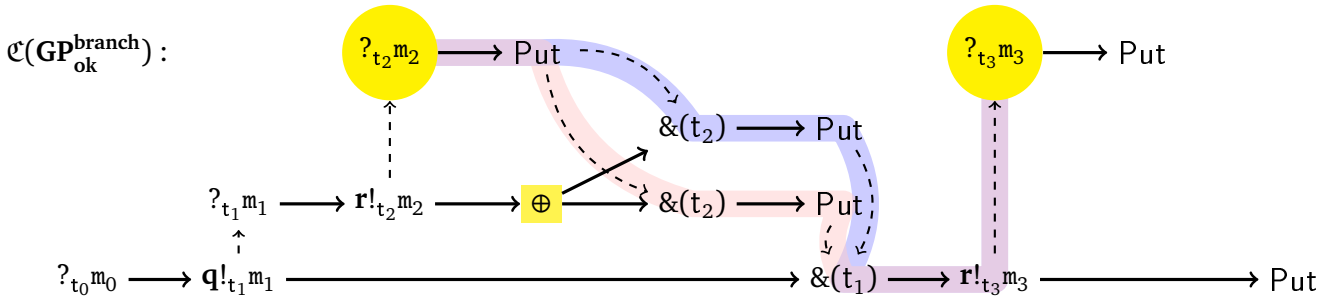
Figure 7.12: Causality Graph $\mathfrak{C}(\mathbf{GP}_{\mathbf{ok}})$.

$$\mathbf{GP}^{\mathbf{branch}}_{\mathbf{fail}} = \mathbf{0} \xrightarrow{t_0} \mathbf{p}{:}m_0.\mathbf{p} \xrightarrow{t_1} \mathbf{q}{:}m_1.\mathbf{q} \xrightarrow{t_2} \mathbf{r}{:}m_2.\mathbf{r}{\downarrow}t_2.\mathbf{q} \left\{ \begin{array}{l} \mathbf{q}{\uparrow}t_2.\mathbf{q}{\downarrow}t_1.\mathbf{p}{\downarrow}t_1 \\ \mathbf{q}{\downarrow}t_1.\mathbf{p}{\downarrow}t_1 \end{array} \right\}.\mathbf{p} \xrightarrow{t_3} \mathbf{r}{:}m_3.\mathbf{r}{\downarrow}t_3.\mathbf{p}{\downarrow}t_0.\mathbf{end}$$



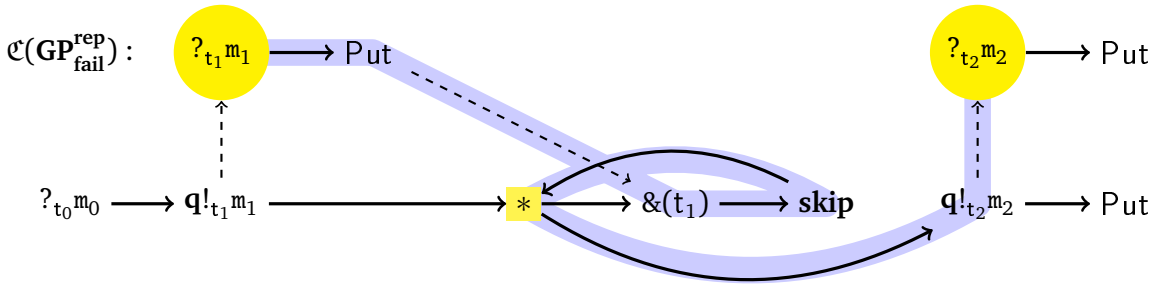
(a) A non-causal projection candidate and its causality graph.

$$\mathbf{GP}^{\mathbf{branch}}_{\mathbf{ok}} = \mathbf{0} \xrightarrow{t_0} \mathbf{p}{:}m_0.\mathbf{p} \xrightarrow{t_1} \mathbf{q}{:}m_1.\mathbf{q} \xrightarrow{t_2} \mathbf{r}{:}m_2.\mathbf{r}{\downarrow}t_2.\mathbf{q} \left\{ \begin{array}{l} \mathbf{q}{\uparrow}t_2.\mathbf{q}{\downarrow}t_1.\mathbf{p}{\downarrow}t_1 \\ \mathbf{q}{\uparrow}t_2.\mathbf{q}{\downarrow}t_1.\mathbf{p}{\downarrow}t_1 \end{array} \right\}.\mathbf{p} \xrightarrow{t_3} \mathbf{r}{:}m_3.\mathbf{r}{\downarrow}t_3.\mathbf{p}{\downarrow}t_0.\mathbf{end}$$



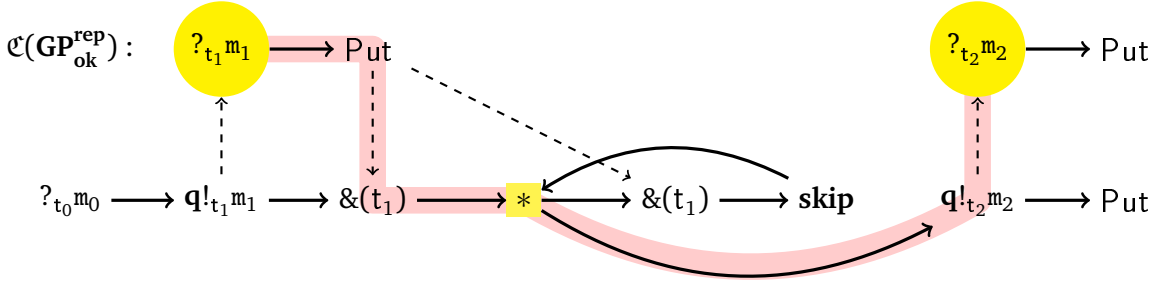
(b) A causal projection candidate and its causality graph.

$$\mathbf{GP}^{\mathbf{rep}}_{\mathbf{fail}} = \mathbf{0} \xrightarrow{t_0} \mathbf{p}{:}m_0.\mathbf{p} \xrightarrow{t_1} \mathbf{q}{:}m_1.\mathbf{q}{\downarrow}t_1.\left(\mathbf{p}{\uparrow}t_1\right)^*\mathbf{p} \xrightarrow{t_2} \mathbf{q}{:}m_2.\mathbf{q}{\downarrow}t_2.\mathbf{p}{\downarrow}t_0.\mathbf{end}$$



(c) A non-causal projection candidate and its causality graph.

$$\mathbf{GP}^{\mathbf{rep}}_{\mathbf{ok}} = \mathbf{0} \xrightarrow{t_0} \mathbf{p}{:}m_0.\mathbf{p} \xrightarrow{t_1} \mathbf{q}{:}m_1.\mathbf{q}{\downarrow}t_1.\mathbf{p}{\uparrow}t_1.\left(\mathbf{p}{\uparrow}t_1\right)^*\mathbf{p} \xrightarrow{t_2} \mathbf{q}{:}m_2.\mathbf{q}{\downarrow}t_2.\mathbf{p}{\downarrow}t_0.\mathbf{end}$$



(d) A causal projection candidate and its causality graph.

Figure 7.13: Causality Graphs.

Lemma 7.2. *Let* **GP** *be a projection candidate without repetitions with a coherent causality graph. Let* Prgm *be a program and* γ_1 *a global trace with* Prgm $\Downarrow \gamma_1$*, such that* γ_1 *is a model for gcsem*(**GP**)*. Let* γ_2 *be another global trace with* Prgm $\Downarrow \gamma_2$*, such that all events of* γ_2 *are witnesses in* γ_1*. The order of these events for every object is the same.*

Intuitively, the lemma expresses that if **GP** is causal-coherent, then in all the models of its semantics, the order of all its invocation reaction events and all its suspension reaction events is the same. The requirement on the witnessed actions expresses that the order only has to be preserved if the same branches have been taken.

To generalize the above lemma to global protocols with repetitions, we require some technical auxiliary structure. The main challenge is that repeated actions have multiple witnesses. Thus, we introduce, for the correctness proof alone, bounded repetitions.

Definition 7.21 (Bounded Repetition)**.** *Let* **GP** *be global protocol with at least one repetition. Let* $(\mathbf{G}'_i)^*$ *be the repeated types for* $i \leq m$*.*

- *The n-*$\mathbf{G}'_i$*-unrolling of* **GP** *is the global protocol that results from replacing* $(\mathbf{G}'_i)^*$ *with n copies of* $\mathbf{G}'_i$*.*[9] *The unrolling of* **GP** *is the set containing all n-*$\mathbf{G}'_i$*-unrollings of* **GP***.*

Kleene-Repetition is equivalent to an infinite choice over all bounded repetitions.

Lemma 7.3. *If a trace is a model for* **GP***, then it is a model for some protocol in its unrolling and vice versa.*

Lemma 7.4. *Let* **GP** *be a projection candidate with a coherent causality graph. Let* $\mathbf{GP}'$ *be any of the protocols in its unrolling. Let* Prgm *be a program and* γ_1 *a global trace with* Prgm $\Downarrow \gamma_1$*, such that* γ_1 *is a model for gcsem*($\mathbf{GP}'$)*. Let* γ_2 *be another global trace of* $\mathbf{GP}'$ *with* Prgm $\Downarrow \gamma_2$*, such that all events of* γ_2 *are witnesses in* γ_1*. The order of these events for every object is the same.*

Propagation

We require one last step to generate local types. Right now, the loop invariant of global repetition is ignored by the methods called within and the formula describing the reactivation in Rel is not ensured to hold.

Example 7.22. *We continue the leading example* **S** *from Ex. 7.7, the invariant* $\psi_{\mathbf{S}} = \mathbf{CMP}.\texttt{count} >= 0 \wedge \mathbf{GUI}.\texttt{shown} >= 0$ *of its repetition and its projected local protocols:*

$$\mathbf{S}_{\mathsf{t}_0} = ?_{\mathsf{t}_0}\texttt{init}\,.\,\mathbf{CMP}!_{\mathsf{t}_1}\texttt{compute}(\!|\mathsf{t}_\mathsf{v}\ \mathbf{as}\ \texttt{len(v)}|\!).\mathsf{Susp}(\!|\mathsf{t}_1, \mathsf{true}, \mathsf{true}|\!).\&^{\mathsf{t}_1}_{\mathsf{t}_5}\left\{\begin{array}{l}\mathsf{t}_5 \doteq\ \ 0 : \mathsf{Put}\ \texttt{result} \doteq -1\\ \mathsf{t}_5 \doteq -1 : \mathsf{Put}\ \texttt{result} \doteq 0\end{array}\right\}.\ \mathbf{skip}$$

$$\mathbf{S}_{\mathsf{t}_1} = ?_{\mathsf{t}_1}\texttt{compute}(\!|\mathsf{t}_\mathsf{v}\ \mathbf{as}\ \texttt{len(v)}|\!).\oplus\left\{\begin{array}{ll}\mathsf{t}_\mathsf{v} \geq 0 : & \left(\mathbf{GUI}!_{\mathsf{t}_2}\texttt{show}.\&^{\mathsf{t}_2}\{\mathbf{skip}\}\right)^*_{\mathbf{this}.\texttt{count}\geq 0}.\mathsf{Put}\ (\!|\texttt{result} \doteq 0|\!)\\ \mathsf{t}_\mathsf{v} \doteq 0 : & \mathsf{Put}\ (\!|\texttt{result} \doteq -1|\!)\end{array}\right\}.\mathbf{skip}$$

$$\mathbf{S}_{\mathsf{t}_2} = ?_{\mathsf{t}_2}\texttt{show}\,.\,\mathsf{Put}$$

The type of t_1*, establishes its part of the invariant, but the types of* **GUI** *do not.*

We call the final step propagation: it adds the loop invariant at the places where it has to be proven (and can be assumed). The causality graph is used for that, as it allows us to determine (1) what the last specified action of an object before the repetition is — this action has to establish the invariant, (2) what the first action within the repetition is — this action may assume the invariant and (3) the last action within the repetition — this action has to establish the invariant.

This is only needed for local types that are repeatedly called, thus (2) is always the node of a receiving action and (3) the node of a termination action. (1) is either a termination action or an out-going node of a suspension action.

[8] I.e., there are paths to and from v to this repetition node.

[9] With appropriate renaming of introduced tracked futures.

Definition 7.22 (Propagation)**.** *Let* **GP** *be a projection candidate with a coherent causality graph* $\mathfrak{C}(\mathbf{GP})$*. The propagated global protocol* $\mathbf{GP}'$ *is generated as follows.*

- *For every repetition* $*$ *with invariant* ψ*, for each tracked future* t *that is active within this repetition:*
  - *For the last suspension* Susp *with an* out *node or termination action* Put *before the repetition (i.e., there is a path from this node to the repetition that does not contain any node from the same projection),* $\mathsf{divide}(\psi, \mathsf{t})$ *is added to the suspension assertion or postcondition.*
  - *for the last suspension* Susp *with an* out *node or termination action* Put *inside the repetition (i.e., there is a path from this node to the repetition that does not contain any node from the same projection),* $\mathsf{divide}(\psi, \mathsf{t})$ *is added to the suspension assertion or postcondition.*
  - *for the first suspension* Susp *with an* in *node or receiving action* $?\mathtt{m}$ *before the repetition (i.e., there is a path from the repetition to this node that does not contain any node from the same projection),* $\mathsf{divide}(\psi, \mathsf{t})$ *is added to the suspension assumption or precondition.*
  - *for the first suspension* Susp *with an* in *node or receiving action* $?\mathtt{m}$ *inside the repetition (i.e., there is a path from the repetition to this node that does not contain any node from the same projection),* $\mathsf{divide}(\psi, \mathsf{t})$ *is added to the suspension assumption or precondition.*
  - *If a repetition has a non-trivial invariant, but there is neither a last suspension or termination before, then propagation fails.*
- *For every branching, we add the branch condition to the precondition of every call within the branch in question.*
- *For the last* Susp *with an* out *node or termination action* Put *before an* Susp *with an* in *node and a reactivation condition* φ*, we add* φ *to the suspension condition or termination condition.*

There may be multiple last or first nodes, e.g., if there is a branching inside the repetition.

Example 7.23. *After propagation,* $\mathbf{S}_{\mathsf{t}_0}$ *and* $\mathbf{S}_{\mathsf{t}_2}$ *are modified:*

$$\mathbf{S}^{\mathsf{prop}}_{\mathsf{t}_0} = ?_{\mathsf{t}_0}\mathtt{init}\,.\,\mathbf{CMP}!_{\mathsf{t}_1}\mathtt{compute}(\!|\mathsf{t}_\mathtt{v}\ \mathbf{as}\ \mathtt{len(v)}|\!).\mathsf{Susp}(\!|\mathsf{t}_1, \mathbf{this}.\mathtt{shown} \geq 0, \mathbf{this}.\mathtt{shown} \geq 0|\!)$$
$$.\,\&^{\mathsf{t}_1}_{\mathsf{t}_5}\left\{\begin{array}{l}\mathsf{t}_5 \doteq\ \ 0 : \mathsf{Put}\ \mathtt{result} \doteq -1\\ \mathsf{t}_5 \doteq -1 : \mathsf{Put}\ \mathtt{result} \doteq 0\end{array}\right\}\,.\,\mathbf{skip}$$
$$\mathbf{S}^{\mathsf{prop}}_{\mathsf{t}_2} = ?_{\mathsf{t}_2}\mathtt{show}(\!|\mathsf{t}_\mathtt{v} \geq 0 \wedge \mathbf{this}.\mathtt{shown} \geq 0|\!)\ .\ \mathsf{Put}\ \mathbf{this}.\mathtt{shown} \geq 0$$

The type of `show` has now knowledge about $\mathsf{t}_\mathtt{v}$, because it is called in the branch guarded by $\mathsf{t}_\mathtt{v} \geq 0$. `show` may not have access to $\mathsf{t}_\mathtt{v}$, but this addition is performed after the history-sensitivity analysis and is not critical, because if `show` does not have access to it, then it cannot use this information anywhere (if it would use it, it must be specified and if it is specified it will be caught by the history-sensitivity analysis before.)

Definition 7.23. *Let* **GP** *be a causal-coherent projection candidate, such that propagation succeeds. We define the types of a method as*

$$\mathscr{L}(\mathtt{m}) = \left\{?_{\mathsf{t}}\mathtt{m}(\!|\varphi|\!).\mathbf{L} \mid \mathbf{GP} \rightsquigarrow^{\mathbf{p}}_{\mathsf{t}} ?_{\mathsf{t}}\mathtt{m}(\!|\varphi'|\!).\mathbf{L}' \text{ for some } \mathsf{t}, \mathbf{p} \text{ s.t. } ?_{\mathsf{t}}\mathtt{m}(\!|\varphi|\!).\mathbf{L} \text{ is the propagation of } ?_{\mathsf{t}}\mathtt{m}(\!|\varphi'|\!).\mathbf{L}'\right\}$$

GP *is well-formed if for any two* $?_{\mathsf{t}_1}\mathtt{m}(\!|\varphi_1|\!).\mathbf{L}_1, ?_{\mathsf{t}_2}\mathtt{m}(\!|\varphi_2|\!).\mathbf{L}_2 \in \mathscr{L}(\mathtt{m})$*, the receiving conditions do not overlap.*

This models the method can always distinguish which type it has to follow by its initial conditions.

7.5 Behavioral Type and Soundness

In this section we define the behavioral type $\mathbb{T}_{|oc} = (\tau_{|oc}, \alpha_{|oc}, \iota_{|oc}, \pi_{|oc})$. For the behavioral specification, we can reuse semantics and syntax from the previous sections.

Definition 7.24 (Behavioral Specification $\mathbb{T}_{|oc}$)**.** *The syntax $\tau_{|oc}$ of the behavioral specification $\mathbb{T}_{|oc}$ is defined by*

$$\tau_{|oc} ::= \mathbf{L} \triangleright \mathsf{C}$$

I.e., the statement has to follow $\mathbf{L}$ *and may assume that so far every tracked value* t *is available in all expressions in* $\mathsf{C}(\mathsf{t})$*. The semantics $\alpha_{|oc}$ expresses the semantics of the local type* $\boldsymbol{L}$ *under tracking constraint* C*. Contrary to the semantics of local type, however, we use the tracking constraint as an assumption.*

$$\alpha_{|oc}(\mathbf{L} \triangleright \mathsf{C}) = \exists_{\mathsf{t} \in \mathbf{dom}(\mathsf{C})} \mathsf{t} \in \mathtt{D}(\mathsf{t}).\ \left(\left(\bigwedge_{\substack{e \in \mathsf{C}(\mathsf{t}) \\ \mathsf{t} \in \mathbf{dom}(\mathsf{C})}} \mathsf{t} \doteq e \right) \rightarrow lsem^{f}_{\mathbf{p}}(\mathbf{L}, \mathbf{dom}(\mathsf{C})) \right)$$

Where $\mathbf{p}$ *is the role of the class in question and* $f \notin \mathbf{dom}(\mathsf{C})$ *some constant symbol for the resolved future.*

Before we give the type system, we require some auxiliary operations on tracking constraints.

Definition 7.25 (Auxiliary Operations)**.** *Given an expression* e*, the removal operation* rem *removes all expressions containing* e *from the tracking set of all tracked values.*

$$\mathsf{rem}(\mathsf{C}, \mathsf{e})(\mathsf{t}) = \mathsf{C}(\mathsf{t}) \setminus \{\mathsf{e}_1, \ldots, \mathsf{e}_n\} \text{ where } \forall i \leq n.\ \mathsf{e}_i \text{ contains } \mathsf{e} \text{ and } \mathsf{e}_i \in \mathsf{C}(\mathsf{t})$$

For example

$$\mathsf{rem}\Big(\big\{\mathsf{t}_1 \mapsto \{\mathtt{a}, \mathtt{b+1}\}, \mathsf{t}_2 \mapsto \{\mathtt{b+2}\}, \mathsf{t}_3 \mapsto \{\mathtt{x}\}\big\}, \mathtt{b}\Big) = \big\{\mathsf{t}_1 \mapsto \{\mathtt{a}\}, \mathsf{t}_2 \mapsto \emptyset, \mathsf{t}_3 \mapsto \{\mathtt{x}\}\big\}$$

We extend union and intersection of sets of expressions to tracking constraints.

$$(\mathsf{C} \Cup \mathsf{C}')(\mathsf{t}) = \mathsf{C}(\mathsf{t}) \cup \mathsf{C}'(\mathsf{t}) \qquad (\mathsf{C} \Cap \mathsf{C}')(\mathsf{t}) = \mathsf{C}(\mathsf{t}) \cap \mathsf{C}'(\mathsf{t})$$

If a tracking value t *does not occur in* $\mathbf{dom}(\mathsf{C})$ *we set* $\mathsf{C}(\mathsf{t}) = \emptyset$*. For example*

$$\Big(\big\{\mathsf{t}_1 \mapsto \{\mathtt{a}\}, \mathsf{t}_2 \mapsto \{\mathtt{b}\}\big\} \Cup \big[\mathsf{t}_2 \mapsto \{\mathtt{c}\}, \mathsf{t}_2 \mapsto \{\mathtt{d}\}\big\}\Big) = \big\{\mathsf{t}_1 \mapsto \{\mathtt{a}\}, \mathsf{t}_2 \mapsto \{\mathtt{b}, \mathtt{c}\}, \mathsf{t}_3 \mapsto \{\mathtt{d}\}\big\}$$

The bind operation binds an expression e *as tracking for some tracked value* t'*.*

$$\mathsf{bind}(\mathsf{C}, \mathsf{t}', \mathsf{e})(\mathsf{t}) = \begin{cases} \mathsf{C}(\mathsf{t}) & \text{if } \mathsf{t} \neq \mathsf{t}' \\ \mathsf{C}(\mathsf{t}) \cup \{\mathsf{e}\} & \text{if } \mathsf{t} = \mathsf{t}' \end{cases}$$

For example

$$\mathsf{bind}\Big(\big\{\mathsf{t}_1 \mapsto \{\mathtt{a}\}, \mathsf{t}_2 \mapsto \{\mathtt{b}\}\big\}, \mathsf{t}_2, \mathtt{c}\Big) = \big\{\mathsf{t}_1 \mapsto \{\mathtt{a}\}, \mathsf{t}_2 \mapsto \{\mathtt{b}, \mathtt{c}\}\big\}$$

The copy operation mirrors a copy from e' *to* e*: any tracked value available in* e' *is also available in* e*.*

$$\mathsf{copy}\big(\mathsf{C}, \mathsf{e}, \mathsf{e}'\big)(\mathsf{t}) = \begin{cases} \mathsf{C}(\mathsf{t}) \cup \{\mathsf{e}\} & \text{if } \mathsf{e}' \in \mathsf{C}(\mathsf{t}) \\ \mathsf{C}(\mathsf{t}) & \text{otherwise} \end{cases}$$

For example

$$\mathsf{copy}\Big(\big\{\mathsf{t}_1 \mapsto \{\mathtt{a}\}, \mathsf{t}_2 \mapsto \{\mathtt{b}\}\big\}, \mathtt{c}, \mathtt{a}\Big) = \big\{\mathsf{t}_1 \mapsto \{\mathtt{a}, \mathtt{c}\}, \mathsf{t}_2 \mapsto \{\mathtt{b}\}\big\}$$

Finally, we give a logical characterization for tracking constraints. Let f_{t} *be a unique function symbol assigned to* t*.*

$$\mathsf{log}(\mathsf{C}) = \bigwedge_{\mathsf{t} \in \mathbf{dom}(\mathsf{C})} \bigwedge_{\mathsf{e} \in \mathsf{C}(\mathsf{t})} f_{\mathsf{t}} \doteq \mathsf{e}$$

Domain and image of each tracking constraint must be finite. Given a formula φ *containing tracked values, we denote with* $\dot{\varphi}$ *the formula which substitutes each tracked value with the function symbol assigned to it.*

For simplicity, we assume that $|\mathscr{L}(\mathtt{m})| = 1$. This can be easily achieved by copying and renaming methods that are called at multiple call sites in a program and accordingly renaming in the global type.

Definition 7.26 (Behavioral Type $\mathbb{T}_{|\mathsf{oc}}$)**.** *This extends Def. 7.24. The type system $\pi_{|\mathsf{oc}}$ is given in Fig. 7.14. Given a well-formed global protocol* **GP** *we with $\mathscr{L}(\mathtt{m}) = \{?_{\mathtt{t}}\mathtt{m}(\!|\varphi_{\mathtt{m}}, \mathsf{C}_{\mathtt{m}}|\!) \,.\, \mathbf{L}_{\mathtt{m}}\}$ and a local role assignment l for the role in question, we set*

$$\iota_{|\mathsf{oc}}(\mathtt{m}) = (\varphi_{\mathtt{m}} \wedge \mathsf{log}(l), \mathbf{L}_{\mathtt{m}} \triangleright \mathsf{C}_{\mathtt{m}})$$

Rule ($\mathbb{T}_{|\mathsf{oc}}$-assignF) and ($\mathbb{T}_{|\mathsf{oc}}$-assignV) both work the same way: the local type is kept unchanged, the update keeps track of the assignment and the tracking constraint first removes the overwritten location from all tracking sets and then tracks the assigned expression if it is already a tracked value.

Rule ($\mathbb{T}_{|\mathsf{oc}}$-return) matches the terminating action and verifies its condition, but uses the logical characterization of the tracking constraint as additional information.

Example 7.24. *Consider a method that keeps track of some tracked value* t *that is available upon termination in two variables* a *and* b*. Tracking constraint and their logical characterization allows to specify that a method keeps track of the value correctly (here: return* t*) without referring to program elements.*

$$\dfrac{\dfrac{\dfrac{}{\mathtt{t} \doteq \mathtt{a} \wedge \mathtt{t} \doteq \mathtt{b{+}1} \Rightarrow \mathtt{a} \doteq \mathtt{t}}}{\Rightarrow \{\mathtt{result} := \mathtt{a}\}((\mathtt{t} \doteq \mathtt{a} \wedge \mathtt{t} \doteq \mathtt{b{+}1}) \rightarrow (\mathtt{result} \doteq \mathtt{t}))}}{\Rightarrow \left[\textbf{return}\ \ \mathtt{a} \overset{\alpha_{|\mathsf{oc}}}{\Vdash} \mathsf{Put}\ \mathtt{result} \doteq \mathtt{t} \triangleright [\mathtt{t} \mapsto \{\mathtt{a}, \mathtt{b{+}1}\}]\right]}$$

This specific example can be done without tracking constraints.

Rule ($\mathbb{T}_{|\mathsf{oc}}$-awaitF) matches the suspending action. The first premise verifies the suspending condition, again using the logical characterization of the tracking constraint and that the process synchronizes on the correct future, which is given as a tracked value. The second premise verifies the continuation after reactivation. The anonymizing heap is analogous to the one of π_{pst}. The behavioral specification is the continuation of the local type and the new tracking constraint. This constraint is constructed by removing all tracked values which are tracked by expressions containing the heap and adding the new binding to keep track of value available in the new heap.

There is no rule for boolean awaits, because we cannot derive causality graphs for them.

Rule ($\mathbb{T}_{|\mathsf{oc}}$-if) matches active choice against branching. It does *not* reduce the number of choices. Instead, the rewrite rules allow to drop certain branches at any point and another rewrite rule to remove the active choice once it becomes a guarded specification of a single choice. Only at this point, the choice condition is verified and the choice guard may only reason about tracked values and input parameters — postponing their verification is possible because they do *not* rely on the concrete branching structure and computation order.

Example 7.25. *Consider the proof fragment below. Only the proof branch for the first branch is shown. The whole active choice is moved to the first branch, the second rule then selects which of the branches is taken (note that it is sound to drop active choice branches) and the third rule then checks whether the branch has been taken under the correct condition.*

We stress that the nested binary branching of the code may not structurally match the multi-branching of the type.

$$(\mathbb{T}_{\text{loc}}\text{-assignF})\ \frac{\Gamma \Rightarrow \{U\}\{\text{heap} := \text{store}(\text{heap}, \mathtt{f}, \mathtt{e})\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \text{copy}(\text{rem}(\mathsf{C}, \mathtt{f}), \textbf{this}.\mathtt{f}, \mathtt{e})], \Delta}{\Gamma \Rightarrow \{U\}[\textbf{this}.\mathtt{f} = \mathtt{e};\ \mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \mathsf{C}], \Delta}$$

$$(\mathbb{T}_{\text{loc}}\text{-return})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{result} := \mathtt{e}\}(\log(\mathsf{C}) \rightarrow \dot{\varphi}), \Delta}{\Gamma \Rightarrow \{U\}[\textbf{return}\ \mathtt{e};\ \overset{\alpha_{|\text{oc}}}{\Vdash} \text{Put}\ \varphi \triangleright \mathsf{C}], \Delta} \qquad (\mathbb{T}_{\text{loc}}\text{-assignV})\ \frac{\Gamma \Rightarrow \{U\}\{\mathtt{v} := \mathtt{e}\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \text{copy}(\text{rem}(\mathsf{C}, \mathtt{v}), \mathtt{v}, \mathtt{e})], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e};\ \mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \mathsf{C}], \Delta}$$

$$(\mathbb{T}_{\text{loc}}\text{-awaitF})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}(\log(\mathsf{C}) \rightarrow \dot{\varphi} \wedge f_{\mathtt{t}} \doteq \mathtt{e}), \Delta \\ \Gamma, \{U\}\{\text{heap} := \text{anon}(\text{heap})\}\dot{\varphi}' \Rightarrow \{U\}\{\text{heap} := \text{anon}(\text{heap})\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \text{rem}(\mathsf{C}, \textbf{this})], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{await}\ \mathtt{e?};\ \mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \text{Susp}(\mathtt{t}, \varphi, \varphi') \,.\, \mathbf{L} \triangleright \mathsf{C}], \Delta}$$

$$(\mathbb{T}_{\text{loc}}\text{-if})\ \frac{\begin{array}{c}\Gamma, \{U\}\neg\mathtt{e} \Rightarrow \{U\}[\mathtt{s}';\mathtt{s}'' \overset{\alpha_{|\text{oc}}}{\Vdash} \oplus\{\varphi_i : \mathbf{L}_i\}_{i\in I} \,.\, \mathbf{L}' \triangleright \mathsf{C}], \Delta \\ \Gamma, \{U\}\mathtt{e} \Rightarrow \{U\}[\mathtt{s};\mathtt{s}'' \overset{\alpha_{|\text{oc}}}{\Vdash} \oplus\{\varphi_i : \mathbf{L}_i\}_{i\in I} \,.\, \mathbf{L}' \triangleright \mathsf{C}], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{if}(\mathtt{e})\{\mathtt{s};\textbf{skip}\}\textbf{else}\{\mathtt{s}';\textbf{skip}\};\mathtt{s}'' \overset{\alpha_{|\text{oc}}}{\Vdash} \oplus\{\varphi_i : \mathbf{L}_i\}_{i\in I} \,.\, \mathbf{L}' \triangleright \mathsf{C}], \Delta}$$

$$(\mathbb{T}_{\text{loc}}\text{-}\oplus\text{-remove})\ \frac{\Gamma \Rightarrow \{U\}\varphi, \Delta \qquad \Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \mathsf{C}], \Delta}{\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \oplus\{\varphi : \mathbf{L}\} \triangleright \mathsf{C}], \Delta} \qquad \begin{array}{c}\mathbf{skip}.\mathbf{L} \triangleright \mathsf{C} \leftrightsquigarrow \mathbf{L} \triangleright \mathsf{C} \leftrightsquigarrow \mathbf{L}.\mathbf{skip} \triangleright \mathsf{C} \\ \mathbf{L} \triangleright \mathsf{C} \leftrightsquigarrow \oplus\{\text{true} : \mathbf{L}\} \triangleright \mathsf{C} \\ \oplus\{\varphi_i : \mathbf{L}_i\}_{i\in I} \rightsquigarrow \oplus\{\varphi_i : \mathbf{L}_i\}_{i\in I'} \text{ with } I' \subseteq I\end{array}$$

$$(\mathbb{T}_{\text{loc}}\text{-callV})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}(\log(\mathsf{C}) \rightarrow \dot{\varphi}(\mathtt{e}') \wedge \mathbf{p} \doteq \mathtt{e}), \Delta \\ \Gamma \Rightarrow \{U\}\{\mathtt{v} := \nu\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \text{bind}(\text{rem}(\mathsf{C}, \mathtt{v}), \mathtt{t}, \nu) \uplus \mathsf{C}'(\mathtt{m}, \mathtt{e}')], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e}!\mathtt{m}(\mathtt{e}');\ \mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{p}!_{\mathtt{t}}\mathtt{m}(\varphi, \mathsf{C}') \,.\, \mathbf{L} \triangleright \mathsf{C}], \Delta}\ \nu \text{ fresh}$$

$$(\mathbb{T}_{\text{loc}}\text{-readV-}n)\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}(\log(\mathsf{C}) \rightarrow f_{\mathtt{t}'} \doteq \mathtt{e}), \Delta \\ \Gamma, \{U\}\{\mathtt{v} := \nu\}\{\mathtt{t} := \nu\}\varphi_1 \Rightarrow \{U\}\{\mathtt{v} := \nu\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L}_1 \,.\, \mathbf{L} \triangleright \text{bind}(\text{rem}(\mathsf{C}, \mathtt{v}), \mathtt{t}, \nu)], \Delta \\ \cdots \\ \Gamma, \{U\}\{\mathtt{v} := \nu\}\{\mathtt{t} := \nu\}\varphi_n \Rightarrow \{U\}\{\mathtt{v} := \nu\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L}_n \,.\, \mathbf{L} \triangleright \text{bind}(\text{rem}(\mathsf{C}, \mathtt{v}), \mathtt{t}, \nu)], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\mathtt{v} = \mathtt{e}.\textbf{get};\ \mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \&_{\mathtt{t}}^{\mathtt{t}'}\{\varphi_i : \mathbf{L}_i\}_{i\in\{1..n\}} \,.\, \mathbf{L} \triangleright \mathsf{C}], \Delta}\ \nu \text{ fresh}$$

$$(\mathbb{T}_{\text{loc}}\text{-while})\ \frac{\begin{array}{c}\Gamma \Rightarrow \{U\}(\log(\mathsf{C}) \rightarrow (\varphi \wedge I \wedge \log(\mathsf{C}'))), \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(\neg\mathtt{e} \wedge I \wedge \varphi) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s}' \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \mathsf{C}'], \Delta \\ \Gamma, \{U\}\{U_{\mathscr{A}}\}(\mathtt{e} \wedge I \wedge \varphi) \Rightarrow \{U\}\{U_{\mathscr{A}}\}[\mathtt{s} \overset{\alpha_{\text{pst}} \wr \alpha_{|\text{oc}}}{\Vdash} (I \wedge \varphi \wedge \log(\mathsf{C}')) \wr \mathbf{L} \triangleright \mathsf{C}'], \Delta\end{array}}{\Gamma \Rightarrow \{U\}[\textbf{while}(\mathtt{e})\{\mathtt{s}\}\mathtt{s}' \overset{\alpha_{|\text{oc}}}{\Vdash} (\mathbf{L})^*_{\varphi}.\mathbf{L}' \triangleright \mathsf{C}], \Delta}$$

$$(\mathbb{T}_{\text{loc}}\text{-skip})\ \frac{}{\Gamma \Rightarrow \{U\}[\textbf{skip} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{skip} \triangleright \mathsf{C}], \Delta} \qquad (\mathbb{T}_{\text{loc}}\text{-skipCont})\ \frac{\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \mathsf{C}], \Delta}{\Gamma \Rightarrow \{U\}[\textbf{skip};\mathtt{s} \overset{\alpha_{|\text{oc}}}{\Vdash} \mathbf{L} \triangleright \mathsf{C}], \Delta}$$

Figure 7.14: Rules for $\pi_{|\text{oc}}$. $\mathsf{C}\,(\mathtt{m},\mathtt{e})$ is the tracking constraint C with all method parameters of $\mathtt{m}$ set according to the call parameters $\mathtt{e}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{a} \doteq 0 \Rightarrow 10 \doteq 10}}{\mathtt{a} \doteq 0 \Rightarrow \{\mathtt{v} := 10 || \mathtt{result} := 10\}\mathtt{result} \doteq 10}}{\vdots}}{\mathtt{a} \doteq 0 \Rightarrow \left[\mathtt{v = 10;\ return\ v;} \overset{\alpha_{|\mathsf{oc}}}{\Vdash} \mathsf{Put}\ \mathtt{result} \doteq 10\right]} \quad \dfrac{}{\mathtt{a} \doteq 0 \Rightarrow \mathtt{a} \doteq 0}}{\mathtt{a} \doteq 0 \Rightarrow \left[\mathtt{v = 10;\ return\ v;} \overset{\alpha_{|\mathsf{oc}}}{\Vdash} \oplus \left\{\ \mathtt{a} \doteq 0 : \mathsf{Put}\ \mathtt{result} \doteq 10\ \right\}\right]}}{\mathtt{a} \doteq 0 \Rightarrow \left[\mathtt{v = 10;\ return\ v;} \overset{\alpha_{|\mathsf{oc}}}{\Vdash} \oplus \left\{\begin{array}{rl} \mathtt{a} \doteq 0 & : \mathsf{Put}\ \mathtt{result} \doteq 10 \\ \mathtt{a} \doteq 1 & : \mathsf{Put}\ \mathtt{result} \doteq 20 \\ \mathtt{a} < 0 \vee \mathtt{a} > 1 & : \mathsf{Put}\ \mathtt{result} \doteq 30 \end{array}\right\}\right] \quad \vdots}{\Rightarrow \left[\mathtt{if(a == 0)\{v = 10;\}else\ if\ (a == 1)\{v = 20;\}else\ v = 30;\ return\ v;} \overset{\alpha_{|\mathsf{oc}}}{\Vdash} \oplus \left\{\begin{array}{rl} \mathtt{a} \doteq 0 & : \mathsf{Put}\ \mathtt{result} \doteq 10 \\ \mathtt{a} \doteq 1 & : \mathsf{Put}\ \mathtt{result} \doteq 20 \\ \mathtt{a} < 0 \vee \mathtt{a} > 1 & : \mathsf{Put}\ \mathtt{result} \doteq 30 \end{array}\right\}\right]}$$

Rule ($\mathbb{T}_{|\mathsf{oc}}$-callV) matches a call statement with a calling action. The first premise checks that the call condition is adhered too and that the correct object is called. Note that roles are mapped to function symbols in the logic. The second premise then continues symbolic execution with the designated tracking value for the generated future and the new bindings over the parameter.

The **get** statement is handled by a family of rules ($\mathbb{T}_{|\mathsf{oc}}$-readV-$n$) for $n \in \mathbb{N}, n \geq 1$. Each rule is indexed by the number of branches in the passive choice — each branch has its own premise that checks that under a given condition the correct protocol is adhered to. The first premise in every rule checks that the correct future is synchronized on.

The loop invariant rule has three premises and must observe not only the given invariant φ from the type, but may use an additional invariant I and has to ensure that an invariant tracking constraint C' can be found. The reason for I is that φ results from the type and may be not strong enough as a postcondition for the loop, because it is not aware of local variables, etc. The rule itself is, thus, verifying that $\varphi \wedge I \wedge \mathsf{log}(\mathsf{C}')$ is a loop invariant for the given loop. Additionally, the loop body is checked against the inner part of the repeating type. The first premise checks that under the current tracking constraint, the invariant is established before the loop. The second premise uses the invariant afterwards for the continuation. Note that here, C' moves back into the behavioral modality. The third premise checks that the invariant is preserved by the method body. This premise uses the leading composition with $\mathbb{T}_{\mathsf{pst}}$— $\mathbb{T}_{|\mathsf{oc}}$ has no mechanism to detect the end of a method body and reuses the one from postcondition reasoning.

Finally, the **skip** statement is matched against the **skip** action if it is the sole statement. Note that we may introduce such an action as a prefix or suffix of any type. A **skip** statement followed by another statement has no influence on type or update.

Lemma 7.5. *Type $\mathbb{T}_{|\mathsf{oc}}$ is sound.*

Proof. See p. 182.

Theorem 6. *Let* **GP** *be a well-formed global protocol and* Prgm *a fitting program with fitting global and local role assignments.*

i If every proof obligation of the generated proof obligation schema can be proven, then every local trace of any method m *is a model for some* $\mathbf{LP} \in \mathscr{L}(\mathtt{m})$.

ii If every local trace of any method m *is a model for some* $\mathbf{LP} \in \mathscr{L}(\mathtt{m})$ *and the initial block sets up* **GP** *correctly, then every global trace of* Prgm *is a model for gcsem*(**GP**).

Proof. See p. 184.

I.e., if every method follows its local protocol, then the system follows the global protocol.

7.6 Discussion

On Tracking Values.

Tracking values can only be introduced at future calls and future reads. One can easily add an operation $\nu^{\mathbf{p}}\mathsf{t}.\ \mathbf{G}$ that binds t in $\mathbf{G}$ on $\mathbf{p}$. We refrained from introducing an action that models neither control flow nor communication: It is not bound to a syntactic construct of the language, so it is not clear when exactly a value is bound.

During symbolic execution, the tracking constraint is part of the type in the modality, *not* of the logical context. The reason is that it is easy to remove information from the type, while removing information from the logical context requires an anonymization.

On Behavioral Types and Behavioral Contracts.

Our Local Session Types are descriptions of the behavior of a *process*, not a *channel*. They are, in this sense, more similar to behavioral contracts than to behavioral types. Nonetheless, we call them Session Types: (1) to avoid confusion with method contracts (2) because this is standard for actor-based concurrency [107, 30] (3) other type systems for processes are similarly named types, e.g., conversation types [22] and (4) because the projection mechanism is specific to Session Types.

The BPL-type $\mathbb{T}_{\mathsf{eff}}$ shows that BPL can handle types assigned to data instead of processes as well.

Note that futures are already typed by the data type and the type of the parameters of the called method. A future realizes always the same "protocol": (1) The caller sends data (according to the method signature) to the callee (2) the callee reads this data (3) the callee sends its return value (according to the return type) repeatedly and (4) anybody can read this value. The only failure to realize that protocol is a deadlock, but this generic property is handled by multiple tools for Active Objects [78, 51, 69, 59][10] and we refrain from making the program logic more complicated by checking for deadlock freedom as well. But, as discussed in Ex. 4.7, BPL can be used in the context of a deadlock analysis.

On Session Encapsulation.

We presented here a way to handle the whole system as a single session. In [83] we give a less restrictive variant, where the system contains multiple sessions. Each session is a set of objects, initialized by another asynchronous call from the main block.

The next step would be to have session which are initialized during the run of the system, instead of being initialized at the very beginning. To do so, one would require that the session is encapsulated: either the objects participating in the session are not communicating *during the session* with other objects, or they communicate in a way that does not interfere with the session. Either option is an open research question.

Din et al. [45] use an explicit session mechanism in an extension of Active Objects. For channel-based systems, where the session is defined by a channel rather than a set of processes, this issue does not directly occur.

On Further Session Type Features.

There is a multitude of extensions for Session Types which are out of scope of this work. One such feature are *parametric* Session Types [40], where the number of participants is not fixed but parametric. The topology of the participants remains fixed.

An example for a feature where we expect that carrying it over to Active Objects is more challenging is *session delegation* [72]. Session delegation describes that an endpoint delegates its role to another endpoint. In channel-based systems, this is realized by passing the typed channel to another process, which continues the communication. This is possible because the typed channel is the only medium the endpoints communicate over. In Active Objects, delegating would include updating the pointers of the other endpoints *as they communicate via addresses*.

[10] For different fragments of ABS, only [78] handles full coreABS.

On the Semantics.

Contrary to the operational semantics given for Session Types for channel-based systems, we use a denotational semantics. Our well-formedness arguments make use of traces by using the structure of the semantic formulas, instead of runs of Session Types (in their operational semantics). This simplifies the semantics of our system, as it is easier to integrate it with other behavioral types on the model-level, then on run-level. Furthermore, by mapping Session Types to MSO, we are able to use model theory to compare it with other behavioral types. E.g., $\mathbb{T}_{\mathsf{sinv}}$ is expressible in FO, while $\mathbb{T}_{\mathsf{loc}}$ requires MSO, so $\mathbb{T}_{\mathsf{loc}}$ is more complex. Similarly, given two Session Types, we can compare their quantifier rank to argue which of them expresses a more complex protocol.

8 Discussion

In chapter 2 we identified problems with the reasoning systems for Active Objects. The presented system has overcome these limitations.

Problem 1: Homogeneous Specification. We have given specifications for memory access order (Ch. 5), introduced method contracts for asynchronous method calls and cooperative scheduling (Ch. 6), and generalized Session Types far beyond prior limitations (chapter 7), Furthermore, we are still able to formulate object invariants (Def. 4.31), LTL properties (Def. 4.5) and postconditions (Def. 4.4) — indeed $\mathbb{T}_{\mathsf{pst}}$ plays a central role in BPL as a prototypical leading type. Finally, we gave specifications for interactions with a static analysis as $\mathbb{T}_{\mathsf{p2}}$.

Problem 2: Homogeneous Verification. We have introduced BPL (chapter 4), a novel logic that combines principles from dynamic logics and behavioral types. BPL allows us to use different behavioral modalities for different specifications and we combine them with multiple techniques:

- either with rules that connect behavioral specifications of different behavioral types, such as the rule $(\mathbb{T}_{\mathsf{met}}\text{-get})$,
- or by leading composition, a completely novel way to combine a large class of behavioral specification calculi on rule level. (Def. 4.21).

BPL enables us to engineer deductive verification systems by introducing the core principles of reuse and modularity into calculi.

Problem 3: Lack of Projection when Handling State. Our extension of Session Types is the first automatic decomposition formalism for Active Objects that decomposes synchronization patterns with state specifications and tracked values.

Problem 4: Lack of Integration of Static Analyses and Deduction We give examples how BPL enables integration in both directions:

- Method contracts utilize the pointer analysis.
- Deadlock checkers utilize a specialized program logic (Ex. 4.7) .

Additionally, the separation of syntax and semantics in BPL allows us to analyze the specification independently of the logic, as the manifold analyses on global protocols exemplified.

It remains to show that (1) the additional features of Session Types for Active Objects and (2) the specification with object invariants, method contracts and Session Types are indeed practical, i.e., there are realistic scenarios where the specification in layers is natural.

To address the first point, we specify and verify the *Zugmeldegespräch* from Def. 2.2 with Session Types using tracking values for the train number. This was not possible before. To address the second point, we specify and verify a variant of the distributed sum, the distributed product, in all three languages.

Zugmeldegespräch

The *Zugmeldegespräch* between two Zmst A and B is defined as the global protocol **Zug** below and tracks the communicated train id from the `offer` (where it is bound) through the protocol. It ensures that both parties talk about the same train. Additionally, it is specified that the train id is saved by A as waiting

```
interface Zmst{
    Unit start();
    Unit offer(Int trainId);
    Unit accept(Int trainId);
    Unit announce(Int trainId);
}

class Zmst(Zmst other) implements Zmst{
    List<Int> wait = Nil;
    List<Int> arriving = Nil;

    Unit start(){ other!offer(10); wait = Cons(10,wait); }
    Unit offer(Int trainId){
        //here are computations that ensure that there is place when the train arrives
        other!accept(trainId);
    }
    Unit accept(Int trainId){
        other!announce(trainId);
        wait = without(wait, trainId);
    }
    Unit announce(Int trainId){ announce = Cons(trainId,announce); }
}

{
   Zmst A = new Zmst(B);
   Zmst B = new Zmst(A);
   a!start();
}
```

Figure 8.1: Implementation of one communication realizing the *Zugmeldegespräch*.

(for acceptance) after being offered and is removed from this list after being accepted. Similarly, B saves the train id as arriving soon.

$$
\begin{aligned}
&\mathbf{0} \xrightarrow{t_0} A : \texttt{start} \\
&. A \xrightarrow{t_1} B : \texttt{offer}(\texttt{trainId}\ \textbf{as}\ t_{train}) \ . A \downarrow t_0 (\texttt{contains}(\textbf{this}.\texttt{wait}, t_{train})) \\
&. B \xrightarrow{t_2} A : \texttt{accept}(\texttt{trainId}\ \textbf{as}\ t_{train}) \ . B \downarrow t_1 \\
&. A \xrightarrow{t_3} B : \texttt{announce}(\texttt{trainId}\ \textbf{as}\ t_{train}) \ . A \downarrow t_2 (\neg\texttt{contains}(\textbf{this}.\texttt{wait}, t_{train})) \\
&. B \downarrow t_3 (\texttt{contains}(\textbf{this}.\texttt{arriving}, t_{train})) \ . \ \textbf{end}
\end{aligned}
$$

Fig. 8.1 shows an implementation that adheres to this type. It is extracted from the FormbaR model to accommodate our restriction that sessions are initiated from the main block. We do not need to specify that `wait` contains the received id, because this is verified by the protocol. This kind of specification, automatic projection and verification was not possible during FormbaR.

Distributed Product

The specified distributed product, a variant of the distributed sum in Ex. 3.1, is shown in Fig. 8.2. The specification is kept simple to stress the interactions between the specification layers. We specify only

the object invariant of `Server`, which expresses that the field `val` is always non-negative. This expresses that the product is multiplied by some constant set at the beginning. The session type (and the role assignment) is annotated at the main block. It describes the expected behavior of the system, the main feature is that the repetition demands that after `reset` is called, the value of `val` is larger then 1. This is only the case if the offset after the reset is also strictly positive, which is only demanded for the protocol, not in general. This expresses that the product is amplified, or not modified. The method contracts specify that only positive values are send to `cmp` (and to `compute` in the beginning). The heap postcondition of `cmp` specifies that the server actually mutliplies the inputs.

The session type specifies a repetition invariant that is effectively an object invariant *after* `reset` *has been called*. This cannot be expressed by object invariants or method contracts. It is worth noting that the specifications depend on each other: to prove that `cmp` preserves the object invariant (and the repetition invariant), the contract precondition is necessary. To prove that the parameter postcondition of `cmp` holds, the object invariant (or repetition invariant) is necessary.

Additionally, the layered specification allows the user to specify information where it is intuitively useful: The heap postcondition cannot be expressed in a session type, due to the use of **old**. Method `cmp` gets (through propagation in the projection of local types) a postcondition $\texttt{val} \geq 1$, which is only needed in context of the protocol (as the object invariant suffices otherwise) — it would not be a good specification to have it as part of the method contract.

Despite this, it is not necessary to encode one specification language in another or have a calculus to deal with all specifications, instead leading composition can be used to generate the $\mathbb{T}_{\mathsf{loc}} \between \mathbb{T}_{\mathsf{inv}} \between \mathbb{T}_{\mathsf{met}}$ behavioral type. The sequent of the proof obligation is as follows[1].

$$\underbrace{\textbf{this}.\texttt{val} \geq 1}_{\text{Propagated invariant from } \mathbb{T}_{\mathsf{loc}}} \land \underbrace{\textbf{this}.\texttt{val} \geq 0}_{\text{Object invariant from } \mathbb{T}_{\mathsf{inv}}} \land \underbrace{\texttt{next} > 0}_{\text{Parameter precondition from } \mathbb{T}_{\mathsf{met}}}$$

$$\Longrightarrow$$

$$\{\texttt{oldHeap} := \texttt{heap}\}\{\texttt{lastHeap} := \texttt{heap}\}$$

$$\left[\textbf{this}.\texttt{val} = \textbf{this}.\texttt{val} + 1;\ \textbf{return}\ \textbf{this}.\texttt{val} \overset{\alpha_{\mathsf{loc}} \between \alpha_{\mathsf{inv}} \between \alpha_{\mathsf{met}}}{\Vdash}\right.$$

$$\left.\underbrace{\mathsf{Put}\ \textbf{this}.\texttt{val} \geq 1}_{\text{Propagated invariant from } \mathbb{T}_{\mathsf{loc}}} \between \underbrace{\textbf{this}.\texttt{val} \geq 0}_{\text{Object invariant from } \mathbb{T}_{\mathsf{inv}}} \between \underbrace{\texttt{result} > 0 \land \textbf{this}.\texttt{val} \doteq \textbf{old}.\texttt{val} + \texttt{next}}_{\text{Postconditions from } \mathbb{T}_{\mathsf{met}}}\right]$$

Limitations

The presented reasoning system for Active Objects here is modular, but was designed from scratch: a new semantics, a new program logic, several multiple specification languages and several, modular, verification systems. It was, thus, out of scope to cover all features of mature Active Object languages like ABS and all advanced features of method contracts. In the following we discuss the limitations of our approach and sketch how they may be overcome. Many limitations are already discussed in the previous chapters.

Explicit History.

Our system does not keep track of the history in the symbolic execution, every specification of the trace must, thus, be modeled as a new behavioral type. This may be too costly for simple properties. An explicit history can be easily added by adding a new program variable `history` analogous to ABSDL and record past events. The LAGC semantics then must also include the history of past events in its local state. We have shown, however, that an explicit history in not needed for the most common specification patterns.

[1] We omitted the trivial pre- and postcondition of $\mathbb{T}_{\mathsf{pst}}$ for brevity's sake.

```
interface IClient {
    /*@ requires ∀ Int i. 0 <= i < len(values) =>  values[i] > 0; @*/
  Unit compute(List<Int> values);
}
class Client implements IClient(IServer server){
  Unit compute(List<Int> values){
    List<Int> val = values;
    Fut<Unit> fut = server!reset(1);
    Unit u = fut.get;
    while(val != Nil){
      Int v = hd(val);
      Fut<Unit> f = server!cmp(v);
      f.get;
      val = tl(val);
    }
  }
}
interface IServer {
    /*@ succeeds {}; overlaps {}; @*/
    Unit reset(Int i);
    /*@ requires next > 0; ensures result > 0; @*/
    /*@ succeeds {Server.reset}; overlaps {Server.cmp}; @*/
    Unit cmp(Int next);
}
class Server implements IServer{
  /*@ invariant this.val > 0 @*/
  Int val = 0;
  Unit reset(Int i){ val = i;}
  /*@ ensures this.val == \old(this.val) * next; @*/
  Unit cmp(Int next){
    this.val = this.val * next;
    return this.val;
  }
}
/*@ role assignment [prod -> S, c -> C]; protocol Prod; @*/
main{
  IServer sum = new Server();
  IClient c = new Client(sum);
  c!compute(Cons(1,Cons(2,Cons(3,Nil))));
}
```

$$\mathbf{Prod} = \mathbf{0} \xrightarrow{t_0} C : \mathtt{compute} \,.\, C \xrightarrow{t_1} S : \mathtt{reset} \,.\, S\!\downarrow\! t_1 \,.\, C\!\uparrow\! t_1 \,.\, \left(C \xrightarrow{t_2} S : \mathtt{cmp} \,.\, S\!\downarrow\! t_2 \,.\, C\!\uparrow\! t_2 \,.\, \right)^{*}_{S.\mathtt{val} \geq 1} \,.\, C\!\downarrow\! t_0 \,.\, \mathbf{end}$$

Figure 8.2: Specified Distributed Product.

Object Creation.
We do not allow objects to be dynamically created. The sole reason is that this would require a more complex notion of session for Session types, where roles can enter and leave a session. Such work is available for channel-based systems [12] and its adaption to Active Objects is an open research question. We stress that method contracts can be formulated for object creation [85], we refrained from introducing it here to keep the semantics uniform through the thesis.

Other Decomposition Patterns.
Session types are only one decomposition pattern, other patterns may be more suited in other settings, depending on the restrictions. Other patterns are an open research question.

Implementation.
BPL is partially implemented in the `Crowbar` tool[2]. The tool supports the calculus of $\mathbb{T}_{\mathsf{pst}}$ $\Diamond\mathbb{T}_{\mathsf{inv}}$ $\Diamond\mathbb{T}_{\mathsf{met}}$.

[2] `github.com/Edkamb/crowbar-tool`

9 Conclusion, Related and Future Work

We have presented a system that consequently applies modularity to all levels of a reasoning system for object-oriented models of distributed systems:

- The semantics of the Active Object language CAO is a novel LAGC semantics that separates the local semantics of methods from the semantics of objects and systems. The semantics given here is more consequently modular than previous LAGC semantics, by restricting the interactions between LA and GC semantics to a single point: process creation.
- The specification of CAO uses a modular structure: each aspect of specification, e.g., method contracts, object invariants, postconditions, protocols, effects or frames, are specified in their own language independently of the other aspects. Previous approaches required to, e.g., encode method contracts in object invariants.
- The program logic BPL is modular in two ways: first, each specification aspects has its own behavioral type and type system. The type systems are composed on-demand, depending on the used specifications. Second, behavioral specification can be used to interface with external analyses. To our best knowledge, no other program logic has these features.

The modularity of BPL builds upon the modularity of the semantics and the specifications: LAGC semantics simplifies the isolation of methods from their context and modular specification enables the design of simple proof calculi that reuse context established by other specifications.

Conceptually, BPL bridges between program logics and behavioral types in a novel way: behavioral types are formalized as special program logics, behavioral modalities use the denotational semantics of behavioral types to connect them to the program logic. By generalizing typing judgments to modalities, we are able to deductively reason about multiple types in the same formal expression – generalizing the step from Hoare triple to Dynamic Logic. We show that BPL can express systems classically considered as program logics by giving behavioral types for postcondition reasoning, LTL and object invariants, encoding previous work on Dynamic Logic over traces and that BPL can express systems classically considered as behavioral types by encoding effect types and Session Types.

Furthermore, we introduce a generalization of method contracts to cooperative method contracts and Session Types for Active Objects.

9.1 Related Work

We discussed the original formulation of LAGC by Din et al. [43] in section 3.6 and Active Object languages at the beginning of chapter 3. In this chapter we discuss the related work for BPL: Related behavioral types, other connections between behavioral types and logic, dynamic logics over traces and combinations of logics.

Behavioral Types

Behavioral Types, including Session Types, are a broad research field and we refrain from attempting to survey the whole field and focus on (1) Session Types and (2) Behavioral Types for Actors and related concurrency models. We give some recent work on consolidation of the theory below, general surveys were done by Hüttel et al. [74] and Ancona et al. [1].

Session types stem from a line of work derived from data types and in their original formulation [71] aim to avoid error states (states where the program is not finished but execution may not proceed, i.e., deadlocks) in the π-calculus, or channel-based concurrency in general. For a historical overview, we refer to the survey of Hüttel et al. [74].

However, Session Types are not limited to channel-based concurrency, some systems consider the pure Actor concurrency model. Contrary to the Active Object model, pure Actors have no interface — messages may be lost, if the receiver is in a state where no handler for a given message is available. Thus, pure Actors are not a strict subset of Active Objects, but have additional drawbacks that may lead to programming errors, which are not possible in Active Objects *by design*. Neykova [107] uses Multi-Party Session Types to generate *runtime monitors* for Actors in Python and Erlang.

Mostrous and Vasconcelos [105] use Binary Session Types to statically check Actors in Erlang, Crafa [30] uses Multi-Party Session Types for simplified Erlang. Both works do not consider state. Neither of them considers futures (or any other synchronization mechanism) or cooperative scheduling.

Recently, two lines of work revised the numerous extensions of Session Types to consolidate the general theory.

- Bejleri et al. [10] summarize a number of extensions of the original formulation of Multi-Party Session Types (MPST) [72] by removing redundant language features and unifying the extensions in a system with a minimal set of operators.

  Their work considers only channel-based concurrency and operational semantics. Passed data is handled by the assertion mechanism of Bocchi et al. [13] which checks that sent data satisfies a boolean constraint.

- Scalas and Yoshida [113] similarly give a more simple system, motivated by the complex and error-prone original theory of MPST. However, their work sees Session Types as a technique to establish deadlock freedom and one part of their consolidated theory is that global types and projection are removed.

  This thesis and our prior work [83] explore the opposite direction: instead of removing protocol adherence to achieve simpler checking for deadlock freedom, we remove deadlock freedom to achieve simpler checking for protocol adherence: BPL-Session Types do not (necessarily) establish deadlock freedom, but there are a number of specialized tools for Active Objects [78, 51, 69, 59] that do so and we can focus on partial correctness.

Other behavioral types for actor-like concurrency models are *Mailbox Types* and *LAMs*, see below. Both lack the projection mechanism typical for Session Types.

Mailbox Types.

De'Liguoro and Padovani [38] present Mailbox Types, which are regular expressions over actions in a mailbox. A mailbox is a message storage, which one or more entities can use to receive messages. Mailbox Types can be seen as local types, one main difference to our work is that their concurrency models allow one to model concurrent objects, futures and actors, but does not consider state and has no global types, i.e., cannot describe (global) protocols. Similarly, they have no choice operation, only alternative. Nonetheless, we consider Mailbox Types as the system most close to BPL-local types. For one, because of the related concurrency model and focus on protocol adherence (but Mailbox Types do ensure deadlock freedom, too), for another, because contrary to other systems they use the Kleene Star for repetition, instead of a recursion operator. We conjecture that the object-local types in [83] are the direct correspondents of Mailbox Types for Active Objects.

LAMs.

deadLock Analysis Models (LAMs), first introduced by Giachino et al. [57] for value-passing CCS and then used for ABS [69] are behavioral types that abstract methods to analyze dependencies between

processes. Contrary to Mailbox types and BPL-local types, LAMs are not specified by the user, but inferred automatically during a deadlock or resource analysis.

Further Behavioral Types.

There are a number of behavioral types which were not applied to Actors or Active Objects. Behavioral Contracts [26] are types for processes which are shown to be dual to Session Types [16, 11] in the sense that a system can be session-type-checked iff it can be type checked with behavioral contracts (under some restrictions on the behavioral contract). Conversation types [22] type a variant of the π-calculus with limited message passing and allow to mix local and global levels.

Behavioral Types and their Relation to Logics

To our best knowledge, this work is the first to use a denotational semantics for behavioral types, but Session Types as formulas have been examined by Caires et al. [21] and Carbone et al. [24] 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 [95] also characterize Session Types as formulas, but their approach characterizes the *subtyping* relation, not the execution traces as in our work.

Peters et al. [111] use Session Types to generate a sequential variant of a (well-typed) program to simplify model checking of LTL properties.

Trace Program Logics

In the following we discuss three dynamic program logics for traces, which are the closest works to BPL.

DTL.

Beckert and Bruns [6] use LTL formulas in dynamic logic modalities in their Dynamic Trace Logic (DTL) for Java. Given an LTL formula φ, the DTL-formula $[\mathtt{s}]\varphi$ expresses that φ describes all traces of `s`. DTL uses a restricted form of pattern matching: its three loop invariant rules depend on the outermost operator of φ and other rules may consume a "next" operator. However, the rules do not match on predicate and function symbols, only on the outermost operator.

DTL does not use events and specifies patterns of state changes, not of interactions, because it is designed to analyze information flow properties in Java programs.

ABSDL.

The Abstract Behavior Specification Dynamic Logic (ABSDL) of Din and Owe [44] is for ABS. In ABSDL, a formula $[\mathtt{s}]\varphi$, where φ is a first-order formula over the program state, has the standard meaning that φ holds after `s` is executed. Contrary to DTL, ABSDL uses a special program variable to keep track of the visible events, `history`, which contains only events. The history is not the full trace of the statement, but contains only the visible events (synchronization, suspension, etc.). These events mark the interactions between multiple processes.

As specification, only object invariants are available, and these are inbuilt into the rules – the object invariant is *not* visible in the modality. Its rules are tightly coupled with object-invariant reasoning. This makes it impossible to specify the state at arbitrary interactions. It is, e.g., not possible to specify *"A method is only called when some condition on the heap holds"* as an object invariant. This is due to the subtle point that the object invariant only holds at points where the process is descheduled, not at the intermediate states. A specification of the form *"If the last event in the history is a call, then some property holds"* fully expanded reads as *"If* after termination or suspension*, the last event in the history is a call, then some property holds"*. The last event after termination, however, is always futEv, suspEv or condEv.

Finally, the pattern of interactions has to be specified into a FO formula over the history, which blows up the size of any specification besides object invariants [77].

DLCT.

Bubel et al. [17] define Dynamic Logic with Coinductive Traces (DLCT). In DLCT, a formula $[\mathtt{s}]\varphi$, where φ is a trace modality formula, containing symbolic trace formulas, has the meaning that every trace of `s` is a model for φ. Contrary to ABSDL, DLCT keeps track of the whole trace, not just the events. It does so, however, explicitly in the calculus. DLCT refines ABSDL by introducing symbolic trace formulas. These formulas may contain wild card terms. E.g., **finite** is a wild card term that matches any finite sequence. The following symbolic trace formula θ specifies all finite traces starting with a state where φ holds and end with a state where ψ holds:

$$\theta = \lceil \varphi \rceil ** \textbf{finite} ** \lceil \psi \rceil$$

DLCT is not able to specify the property that between two states, some form of event does *not* occur, as symbolic trace formulas are not closed under negation.

Contrary to DTL and BPL, the loop invariant rule of DLCT does not match statement and postcondition, leading to six complex and subtle premises. Furthermore, it always symbolic executes the program in its modality before evaluating the symbolic trace formula. Even if, e.g., the very first call targets the wrong method, the statement is executed to its end, while the BPL-type we give for Session Types would fail-early at this point.

Other logics

For completeness' sake, we note that JavaDL can be seen as a logic over traces once `oldHeap` is used, as it connects the first state of the trace with the last state, by explicitly keeping track of the first state. Similarly, differential dynamic logic [112] is a dynamic logic over traces: a modality $[\mathtt{s}]\varphi$ expresses that φ holds at every hybrid point in time during the execution of the hybrid program `s`. This is realized by allowing hybrid programs to non-deterministically terminate and evaluate the postcondition at every point in time, when following the differential dynamics of the hybrid program. Technically, thus, both logics are not trace logics, but model trace properties. The trace modality of Steinhöfel and Hähnle [120] shares some similarities with BPL. However, Steinhöfel and Hähnle only sketch possible use cases and give no full example of any application. Neither is a validity calculus for the trace modality given, nor are the restrictions of its automata-based validity mechanism explored.

Combining Logics

The semantics logic lMSOT and gMSOT are nested: they consists of a MSO layer that refers to lFOS (or gFOS) formulas for single state. This is a special case of fibring [53] logics, namely *temporalizing* [50].

Given two logics L_1, L_2, with classes of Kripke models M_1, M_2, fibring combines L_1, L_2 to a logic $L_1(L_2)$ whose models are constructed as follows. Given a model m_1, a fibring function F is given that maps every world $w \in m_1$ to a model of M_2. A formulas of $L_1(L_2)$ is then evaluated on m_1, as far as only operators of L_1 are used. Once a L_2 operator is the top-most operator for evaluation in a world $w \in m_1$, evaluation is continued with the evaluation function of L_2 on $F(w)$.

In the case of lMSOT, the temporalization is MSOT(lFOS). The logic MSOT is a standard MSO logic with a theory of futures etc. and an operator $[i] \vdash f(v_1, \ldots, v_n)$ (and analogously for events). Syntactically, instead of using any function symbol f, lFOS formulas are used. Semantically, each world of the Kripke model of MSOT is replaced by a lFOS model, i.e., a state.

The proof system and the leading operator, however, cannot be described in terms of fibring (or weaving, the operation on proof systems of fibred logics). The reason is that the leading operator combines two behavioral specifications that share their models — there is no need to fiber the logics. To the

author's best knowledge, merging of proof systems for different parts of the same *program* logic is not explored (with the aim of reusing fragments of proof systems) beyond this work.

For a detailed discussion of fibring and further combinations of logics beyond temporalization we refer to the work of Gabbay [53] and Caleiro et al. [23].

9.2 Future Work

Besides completing the implementation of `Crowbar`, we see multiple possible lines of research continuing this work.

Soundness Preserving composition of Calculi.

The composition for rules we give in this work is general enough to deal with the calculi required for object-oriented languages with loops but without synchronous recursion. It is an important question how to generalize Defs. 4.22, 4.27 and 4.28 to handle other forms of input rules and characterize the conditions on behavioral type and rules that enables soundness-preserving composition, as well as more general composition.

Further Behavioral Types, Specifications and Concurrency Models.

We formalized only some Behavioral Types in this dissertation and we propose to investigate the expressive power of BPL by formalizing more Behavioral Types, in particular the discussed Ownership Types, LAMs and Flow-Types. Further extension points are the application to other concurrency models, such as channel-based systems based on the π-calculus. We propose that one can use a Separation Logic logic[109] instead of lFOS to handle concurrency models with shared memory. As discussed, recursion, total correctness and advanced features of Session Types, remain open questions.

The connection of aspect-oriented programming (AOP) to aspect-oriented composition of program logics (AOC) is not explored in depth.

Finally, our composition and soundness lemmas use arguments to discard certain traces, because they are impossible to realize in any program. We do not give a logical characterization of such traces, but recent work on characterizing traces of concurrent objects [36] may provide a starting point to do so.

Bibliography

[1] Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S. J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E. B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V. T., and Yoshida, N. Behavioral types in programming languages. *Foundations and Trends in Programming Languages 3*, 2-3 (2016), 95–230.

[2] Armstrong, J. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] Baker, H. C., and Hewitt, C. The incremental garbage collection of processes. *SIGPLAN Not. 12*, 8 (Aug. 1977), 5559.

[4] Baumann, C., Beckert, B., Blasum, H., and Bormer, T. Lessons learned from microkernel verification – specification is the new bottleneck. In *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012.* (2012), F. Cassez, R. Huuck, G. Klein, and B. Schlich, Eds., vol. 102 of *EPTCS*, pp. 18–32.

[5] Beckert, B. A dynamic logic for the formal verification of Java Card programs. In *Java Card Workshop* (2000), vol. 2041 of *Lecture Notes in Computer Science*, Springer, pp. 6–24.

[6] Beckert, B., and Bruns, D. Dynamic logic with trace semantics. In *Automated Deduction - CADE 2013. Proceedings* (2013), M. P. Bonacina, Ed., vol. 7898 of *Lecture Notes in Computer Science*, Springer, pp. 315–329.

[7] Beckert, B., and Hähnle, R. Reasoning and verification: State of the art and current trends. *IEEE Intelligent Systems 29*, 1 (2014), 20–29.

[8] Beckert, B., Klebanov, V., and Weiss, B. Dynamic logic for java. In *Deductive Software Verification - The KeY Book - From Theory to Practice*, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., vol. 10001 of *Lecture Notes in Computer Science*. Springer, 2016, pp. 49–106.

[9] Béjar, R., Hähnle, R., and Manyà, F. A modular reduction of regular logic to classical logic. In *31st IEEE International Symposium on Multiple-Valued Logic, ISMVL 2001, Warsaw, Poland, May 22-24, 2001, Proceedings* (2001), IEEE Computer Society, pp. 221–226.

[10] Bejleri, A., Domnori, E., Viering, M., Eugster, P., and Mezini, M. Comprehensive multiparty session types. *Programming Journal 3*, 3 (2019), 6.

[11] Bernardi, G. T., and Hennessy, M. Modelling session types using contracts. *Mathematical Structures in Computer Science 26*, 3 (2016), 510–560.

[12] Bocchi, L., Demangeon, R., and Yoshida, N. A multiparty multi-session logic. In *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers* (2012), C. Palamidessi and M. D. Ryan, Eds., vol. 8191 of *Lecture Notes in Computer Science*, Springer, pp. 97–111.

[13] Bocchi, L., Honda, K., Tuosto, E., and Yoshida, N. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings* (2010), P. Gastin and F. Laroussinie, Eds., vol. 6269 of *Lecture Notes in Computer Science*, Springer, pp. 162–176.

[14] Bocchi, L., Lange, J., and Tuosto, E. Three algorithms and a methodology for amending contracts for choreographies. *Sci. Ann. Comp. Sci. 22*, 1 (2012), 61–104.

[15] Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E. B., Pun, K. I., Tapia Tarifa, S. L., Wrigstad, T., and Yang, A. M. Parallel objects for multicores: A glimpse at the parallel language encore. In *SFM* (2015), vol. 9104 of *Lecture Notes in Computer Science*, Springer, pp. 1–56.

[16] Bravetti, M., and Zavattaro, G. Relating session types and behavioural contracts: The asynchronous case. In *SEFM* (2019), vol. 11724 of *Lecture Notes in Computer Science*, Springer, pp. 29–47.

[17] Bubel, R., Din, C. C., Hähnle, R., and Nakata, K. A dynamic logic with traces and coinduction. In *Automated Reasoning with Analytic Tableaux and Related Methods - 24th International Conference, TABLEAUX 2015, Wrocław, Poland, September 21-24, 2015. Proceedings* (2015), H. de Nivelle, Ed., vol. 9323 of *Lecture Notes in Computer Science*, Springer, pp. 307–322.

[18] Bubel, R., Hähnle, R., and Tabar, A. H. A program logic for dependence analysis. In *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings* (2019), W. Ahrendt and S. L. T. Tarifa, Eds., vol. 11918 of *Lecture Notes in Computer Science*, Springer, pp. 83–100.

[19] Büchi, J. R., and Landweber, L. H. Definability in the monadic second-order theory of successor. *J. Symb. Log. 34*, 2 (1969), 166–170.

[20] Büchi, J. R. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly 6*, 16 (1960), 66–92.

[21] Caires, L., and Pfenning, F. Session types as intuitionistic linear propositions. In *CONCUR* (2010), vol. 6269 of *Lecture Notes in Computer Science*, Springer, pp. 222–236.

[22] Caires, L., and Vieira, H. T. Conversation types. *Theor. Comput. Sci. 411*, 51-52 (2010), 4399–4440.

[23] Caleiro, C., Sernadas, A., and Sernadas, C. Fibring logics: Past, present and future. In *We Will Show Them! Essays in Honour of Dov Gabbay, Volume One* (2005), S. N. Artëmov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, Eds., College Publications, pp. 363–388.

[24] Carbone, M., Lindley, S., Montesi, F., Schürmann, C., and Wadler, P. Coherence generalises duality: A logical explanation of multiparty session types. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada* (2016), J. Desharnais and R. Jagadeesan, Eds., vol. 59 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 33:1–33:15.

[25] Caromel, D., and Henrio, L. *A theory of distributed objects - asynchrony, mobility, groups, components*. Springer, 2005.

[26] Castagna, G., and Padovani, L. Contracts for mobile processes. In *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings* (2009), M. Bravetti and G. Zavattaro, Eds., vol. 5710 of *Lecture Notes in Computer Science*, Springer, pp. 211–228.

[27] Chandra, A., Halpern, J., Meyer, A., and Parikh, R. Equations between regular terms and an application to process logic. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1981), STOC '81, ACM, pp. 384–390.

[28] Chang, C., and Keisler, H. Models constructed from constants. In *Model Theory*, vol. 73 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1990, pp. 61 – 135.

[29] Cousot, P., Giacobazzi, R., and Ranzato, F. Program analysis is harder than verification: A computability perspective. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II* (2018), H. Chockler and G. Weissenbacher, Eds., vol. 10982 of *Lecture Notes in Computer Science*, Springer, pp. 75–95.

[30] Crafa, S. Behavioural types for actor systems. *CoRR abs/1206.1687* (2012).

[31] Damiani, F., Hähnle, R., Kamburjan, E., and Lienhardt, M. A unified and formal programming model for deltas and traits. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings* (2017), M. Huisman and J. Rubin, Eds., vol. 10202 of *Lecture Notes in Computer Science*, Springer, pp. 424–441.

[32] Damiani, F., Hähnle, R., Kamburjan, E., and Lienhardt, M. Interoperability of software product line variants. In *Proceeedings of the 22nd International Systems and Software Product Line Conference, SPLC 2018* (2018), T. Berger, P. Borba, G. Botterweck, T. Männistö, D. Benavides, S. Nadi, T. Kehrer, R. Rabiser, C. Elsner, and M. Mukelabai, Eds., ACM, pp. 264–268.

[33] Damiani, F., Hähnle, R., Kamburjan, E., and Lienhardt, M. Same same but different: Interoperability of software product line variants. In *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday* (2018), P. Müller and I. Schaefer, Eds., Springer, pp. 99–117.

[34] DB Netz AG. Richtlinie 408, Fahrdienstvorschrift, 2017.

[35] DB Netz AG. Richtlinie 819, LST-Anlagen planen, 2017.

[36] de Boer, F. S., and Hiep, H. A. Axiomatic characterization of trace reachability for concurrent objects. In *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings* (2019), W. Ahrendt and S. L. T. Tarifa, Eds., vol. 11918 of *Lecture Notes in Computer Science*, Springer, pp. 157–174.

[37] de Boer, F. S., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C. C., Johnsen, E. B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., and Yang, A. M. A survey of active object languages. *ACM Comput. Surv. 50*, 5 (Oct. 2017), 76:1–76:39.

[38] de'Liguoro, U., and Padovani, L. Mailbox types for unordered interactions. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands* (2018), T. D. Millstein, Ed., vol. 109 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 15:1–15:28.

[39] Deniélou, P., and Yoshida, N. Dynamic multirole session types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 435–446.

[40] Deniélou, P., Yoshida, N., Bejleri, A., and Hu, R. Parameterised multiparty session types. *Logical Methods in Computer Science 8*, 4 (2012).

[41] Dezani-Ciancaglini, M. personal communication, 19.10.2018.

[42] Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., and Drossopoulou, S. Session types for object-oriented languages. In *ECOOP 2006 – Object-Oriented Programming* (Berlin, Heidelberg, 2006), D. Thomas, Ed., Springer Berlin Heidelberg, pp. 328–352.

[43] Din, C. C., Hähnle, R., Johnsen, E. B., Pun, K. I., and Tapia Tarifa, S. L. Locally abstract, globally concrete semantics of concurrent programming languages. In *Automated Reasoning with Analytic Tableaux and Related Methods - 26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25-28, 2017, Proceedings* (2017), R. A. Schmidt and C. Nalon, Eds., vol. 10501 of *Lecture Notes in Computer Science*, Springer, pp. 22–43.

[44] Din, C. C., and Owe, O. A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebr. Meth. Program. 83*, 5-6 (2014), 360–383.

[45] Din, C. C., Schlatte, R., and Chen, T. Program verification for exception handling on active objects using futures. In *Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings* (2018), E. B. Johnsen and I. Schaefer, Eds., vol. 10886 of *Lecture Notes in Computer Science*, Springer, pp. 73–88.

[46] Ebbinghaus, H.-D. *Chapter II: Extended Logics: The General Framework*, vol. Volume 8 of *Perspectives in Mathematical Logic*. Springer-Verlag, New York, 1985, pp. 25–76.

[47] Eckhardt, J., Vogelsang, A., and Fernández, D. M. Are "non-functional" requirements really non-functional?: An investigation of non-functional requirements in practice. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 832–842.

[48] Eisenbahnbundesamt (Federal Railway Authority). Eisenbahn-Signalordnung, 2017.

[49] Findler, R. B., Latendresse, M., and Felleisen, M. Behavioral contracts and behavioral subtyping. *SIGSOFT Softw. Eng. Notes 26*, 5 (Sept. 2001), 229–236.

[50] Finger, M., and Gabbay, D. M. Adding a temporal dimension to a logic system. *Journal of Logic, Language and Information 1*, 3 (1992), 203–233.

[51] Flores-Montoya, A., Albert, E., and Genaim, S. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings* (2013), D. Beyer and M. Boreale, Eds., vol. 7892 of *Lecture Notes in Computer Science*, Springer, pp. 273–288.

[52] Floyd, R. W. Assigning meaning to programs. In *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia* (Providence RI, 1967), J. T. Schwartz, Ed., vol. 19, American Mathematical Society, pp. 19–31.

[53] Gabbay, D. M. Fibred semantics and the weaving of logics, part 1: Modal and intuitionistic logics. *J. Symb. Log. 61*, 4 (1996), 1057–1120.

[54] García-Matos, M., and Väänänen, J. Abstract model theory as a framework for universal logic. In *Logica Universalis* (Basel, 2005), J.-Y. Beziau, Ed., Birkhäuser Basel, pp. 19–33.

[55] Garralda, P., Compagnoni, A. B., and Dezani-Ciancaglini, M. BASS: boxed ambients with safe sessions. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy* (2006), A. Bossi and M. J. Maher, Eds., ACM, pp. 61–72.

[56] Gentzen, G. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift 39*, 1 (1935), 176–210.

[57] Giachino, E., Kobayashi, N., and Laneve, C. Deadlock analysis of unbounded process networks. In *CONCUR 2014 – Concurrency Theory* (Berlin, Heidelberg, 2014), P. Baldan and D. Gorla, Eds., Springer Berlin Heidelberg, pp. 63–77.

[58] Gifford, D. K., and Lucassen, J. M. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (New York, NY, USA, 1986), LFP '86, ACM, pp. 28–38.

[59] Gkolfi, A., Din, C. C., Johnsen, E. B., Kristensen, L. M., Steffen, M., and Yu, I. C. Translating active objects into colored Petri nets for communication analysis. *Sci. Comput. Program. 181* (2019), 1–26.

[60] Glinz, M. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India* (2007), IEEE Computer Society, pp. 21–26.

[61] Grahl, D. *Deductive Verification of Concurrent Programs and its Application to Secure Information Flow for Java*. PhD thesis, 2015.

[62] Grahl, D., Bubel, R., Mostowski, W., Schmitt, P. H., Ulbrich, M., and Weiss, B. *Modular Specification and Verification*. Springer International Publishing, Cham, 2016, pp. 289–351.

[63] Grahl, D., and Ulbrich, M. From specification to proof obligations. In *Deductive Software Verification - The KeY Book - From Theory to Practice*, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., vol. 10001 of *Lecture Notes in Computer Science*. Springer, 2016, pp. 243–287.

[64] Hähnle, R., and Huisman, M. Deductive software verification: From pen-and-paper proofs to industrial tools. In *Computing and Software Science - State of the Art and Perspectives*, B. Steffen and G. J. Woeginger, Eds., vol. 10000 of *Lecture Notes in Computer Science*. Springer, 2019, pp. 345–373.

[65] Halstead Jr., R. H. Multilisp: a language for concurrent symbolic computation. *ACM TOPLAS 7*, 4 (1985), 501–538.

[66] Harel, D., Tiuryn, J., and Kozen, D. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.

[67] Hedden, B., and Zhao, X. A comprehensive study on bugs in actor systems. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018* (2018), ACM, pp. 56:1–56:9.

[68] Henkin, L. Relativization with respect to formulas and its use in proofs of independence. *Compositio Mathematica 20* (1968), 88–106.

[69] Henrio, L., Laneve, C., and Mastandrea, V. Analysis of synchronisations in stateful active objects. In *Integrated Formal Methods* (Cham, 2017), N. Polikarpova and S. Schneider, Eds., Springer International Publishing, pp. 195–210.

[70] Hewitt, C., Bishop, P., and Steiger, R. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (1973), IJCAI'73, Morgan Kaufmann Publishers Inc., pp. 235–245.

[71] Honda, K. Types for dyadic interaction. In *CONCUR'93* (Berlin, Heidelberg, 1993), E. Best, Ed., Springer Berlin Heidelberg, pp. 509–523.

[72] Honda, K., Yoshida, N., and Carbone, M. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008* (2008), pp. 273–284.

[73] Huisman, M., Ahrendt, W., Grahl, D., and Hentschel, M. Formal specification with the Java Modeling Language. In *Deductive Software Verification*, vol. 10001 of *Lecture Notes in Computer Science*. Springer, 2016, pp. 193–241.

[74] Hüttel, H., Lanese, I., Vasconcelos, V. T., Caires, L., Carbone, M., Deniélou, P.-M., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H. T., and Zavattaro, G. Foundations of session types and behavioural contracts. *ACM Comput. Surv. 49*, 1 (Apr. 2016), 3:1–3:36.

[75] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects, 9th Intl. Symp., FMCO* (2010), B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds., pp. 142–164.

[76] Johnsen, E. B., Owe, O., and Yu, I. C. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci. 365*, 1-2 (2006), 23–66.

[77] Kamburjan, E. Session Types for ABS. Tech. rep., TU Darmstadt, 2016.

[78] Kamburjan, E. Detecting deadlocks in formal system models with condition synchronization. *ECEASST 76* (2018).

[79] Kamburjan, E. Behavioral program logic. In *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings* (2019), S. Cerrito and A. Popescu, Eds., vol. 11714 of *Lecture Notes in Computer Science*, Springer, pp. 391–408.

[80] Kamburjan, E. Behavioral program logic and LAGC semantics without continuations (technical report). *CoRR abs/1904.13338* (2019).

[81] Kamburjan, E., and Chen, T. Stateful behavioral types for ABS. *CoRR abs/1802.08492* (2018).

[82] Kamburjan, E., and Chen, T. Stateful behavioral types for active objects. In *Integrated Formal Methods - 14th International Conference, IFM 2018, Proceedings* (2018), C. A. Furia and K. Winter, Eds., vol. 11023 of *Lecture Notes in Computer Science*, Springer, pp. 214–235.

[83] Kamburjan, E., Din, C. C., and Chen, T. Session-based compositional analysis for actor-based languages using futures. In *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Proceedings* (2016), K. Ogata, M. Lawford, and S. Liu, Eds., vol. 10009 of *Lecture Notes in Computer Science*, pp. 296–312.

[84] Kamburjan, E., Din, C. C., Hähnle, R., and Johnsen, E. B. Asynchronous cooperative contracts for cooperative scheduling. In *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings* (2019), P. C. Ölveczky and G. Salaün, Eds., vol. 11724 of *Lecture Notes in Computer Science*, Springer, pp. 48–66.

[85] Kamburjan, E., Din, C. C., Hähnle, R., and Johnsen, E. B. Behavioral contracts for cooperative scheduling. In *Deductive Verification: The Next 70 Years* (2020). To appear in LNCS.

[86] Kamburjan, E., and Hähnle, R. Uniform modeling of railway operations. In *Formal Techniques for Safety-Critical Systems - 5th International Workshop, FTSCS 2016, Revised Selected Papers* (2016), C. Artho and P. C. Ölveczky, Eds., vol. 694 of *Communications in Computer and Information Science*, pp. 55–71.

[87] Kamburjan, E., and Hähnle, R. Deductive verification of railway operations. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification, RSSRail 2017, Proceedings* (2017), A. Fantechi, T. Lecomte, and A. B. Romanovsky, Eds., vol. 10598 of *Lecture Notes in Computer Science*, Springer, pp. 131–147.

[88] Kamburjan, E., and Hähnle, R. Formalisierung von betrieblichen und anderen Regelwerken - Das FormbaR Projekt. In *Scientific Railway Signalling Symposium* (2017).

[89] Kamburjan, E., and Hähnle, R. Prototyping formal system models with active objects. In *Proceedings 11th Interaction and Concurrency Experience, ICE 2018* (2018), M. Bartoletti and S. Knight, Eds., vol. 279 of *EPTCS*, pp. 52–67.

[90] Kamburjan, E., Hähnle, R., and Schön, S. Formal modeling and analysis of railway operations with active objects. *Sci. Comput. Program. 166* (2018), 167–193.

[91] Kamburjan, E., and Stromberg, J. Tool support for validation of formal system models: Interactive visualization and requirements traceability. In *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE 2019* (2019), R. Monahan, V. Prevosto, and J. Proenca, Eds., vol. 310 of *EPTCS*.

[92] Kassios, I. T. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods* (Berlin, Heidelberg, 2006), J. Misra, T. Nipkow, and E. Sekerinski, Eds., Springer Berlin Heidelberg, pp. 268–283.

[93] KeY Development Group. Internal bug tracker of the KeY project, issues #1495, #1236, `git.key-project.org`.

[94] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming* (Berlin, Heidelberg, 1997), M. Akşit and S. Matsuoka, Eds., Springer Berlin Heidelberg, pp. 220–242.

[95] Lange, J., and Yoshida, N. Characteristic formulae for session types. In *TACAS* (2016), vol. 9636 of *Lecture Notes in Computer Science*, Springer, pp. 833–850.

[96] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D. M., and Dietl, W. *JML Reference Manual*, May 2013. Draft revision 2344.

[97] Lightbend Inc. https://doc.akka.io/docs/akka/current/actors.html, retrieved 21.08.2019.

[98] Lindström, P. On extensions of elementary logic. *Theoria 35*, 1 (1969), 1–11.

[99] Liskov, B. H., and Shrira, L. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)* (June 1988), D. S. Wise, Ed., ACM Press, pp. 260–267.

[100] Liskov, B. H., and Wing, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (Nov. 1994), 1811–1841.

[101] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. In defense of soundiness: a manifesto. *Commun. ACM 58*, 2 (2015), 44–46.

[102] Marino, D., and Millstein, T. A generic type-and-effect system. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation* (New York, NY, USA, 2009), TLDI '09, ACM, pp. 39–50.

[103] Meyer, B. Applying "design by contract". *Computer 25*, 10 (Oct. 1992), 40–51.

[104] Montgomery, D. C. *Introduction to Statistical Quality Control, 8th Edition*. John Wiley & Sons, Inc., 2019.

[105] Mostrous, D., and Vasconcelos, V. T. Session typing for a featherweight erlang. In *Coordination Models and Languages* (Berlin, Heidelberg, 2011), W. De Meuter and G.-C. Roman, Eds., Springer Berlin Heidelberg, pp. 95–109.

[106] Nakata, K., and Uustalu, T. A Hoare logic for the coinductive trace-based big-step semantics of while. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (2010), A. D. Gordon, Ed., vol. 6012 of *Lecture Notes in Computer Science*, Springer, pp. 488–506.

[107] Neykova, R. *Multiparty session types for dynamic verification of distributed systems*. PhD thesis, Imperial College London, UK, 2016.

[108] Nielson, F., Nielson, H. R., and Hankin, C. *Type and Effect Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 283–363.

[109] O'Hearn, P. W., Reynolds, J. C., and Yang, H. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings* (2001), L. Fribourg, Ed., vol. 2142 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

[110] Pachl, J. *Systemtechnik des Schienenverkehrs: Bahnbetrieb Planen, Steuern und Sichern*. Springer Vieweg, 2008.

[111] Peters, K., Wagner, C., and Nestmann, U. Taming concurrency for verification using multiparty session types. In *Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings* (2019), R. M. Hierons and M. Mosbah, Eds., vol. 11884 of *Lecture Notes in Computer Science*, Springer, pp. 196–215.

[112] Platzer, A. Differential dynamic logic for hybrid systems. *J. Autom. Reasoning 41*, 2 (2008), 143–189.

[113] Scalas, A., and Yoshida, N. Less is more: multiparty session types revisited. *PACMPL 3*, POPL (2019), 30:1–30:29.

[114] Scheurer, D., Hähnle, R., and Bubel, R. A general lattice model for merging symbolic execution branches. In *ICFEM* (2016), vol. 10009 of *Lecture Notes in Computer Science*, pp. 57–73.

[115] Schmitt, P. H. First-order logic. In *Deductive Software Verification - The KeY Book - From Theory to Practice*, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., vol. 10001 of *Lecture Notes in Computer Science*. Springer, 2016, pp. 23–47.

[116] Schön, S., and Kamburjan, E. The future use cases of formal methods in railways. In *Scientific Railway Signalling Symposium* (2018).

[117] Sirjani, M., Movaghar, A., Shali, A., and de Boer, F. S. Modeling and verification of reactive systems using Rebeca. *Fundam. Inform. 63*, 4 (2004), 385–410.

[118] Sommerville, I., and Sawyer, P. *Requirements Engineering: A Good Practice Guide*, 1st ed. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[119] Steinhöfel, D., and Hähnle, R. Abstract execution. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings* (2019), M. H. ter Beek, A. McIver, and J. N. Oliveira, Eds., vol. 11800 of *Lecture Notes in Computer Science*, Springer, pp. 319–336.

[120] Steinhöfel, D., and Hähnle, R. The trace modality. In *Dynamic Logic. New Trends and Applications - Second International Workshop, DaLí 2019, Porto, Portugal, October 7-11, 2019, Proceedings* (2019), L. S. Barbosa and A. Baltag, Eds., vol. 12005 of *Lecture Notes in Computer Science*, Springer, pp. 124–140.

[121] Steinhöfel, D., and Wasser, N. A new invariant rule for the analysis of loops with non-standard control flows. In *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings* (2017), N. Polikarpova and S. Schneider, Eds., vol. 10510 of *Lecture Notes in Computer Science*, Springer, pp. 279–294.

[122] Tasharofi, S., Dinges, P., and Johnson, R. E. Why do Scala developers mix the actor model with other concurrency models? In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings* (2013), G. Castagna, Ed., vol. 7920 of *Lecture Notes in Computer Science*, Springer, pp. 302–326.

[123] van Benthem, J. A new modal Lindström theorem. *Logica Universalis 1*, 1 (Jan 2007), 125–138.

[124] Welsch, Y., and Schäfer, J. Location types for safe distributed object-oriented programming. In *Objects, Models, Components, Patterns* (Berlin, Heidelberg, 2011), J. Bishop and A. Vallecillo, Eds., Springer Berlin Heidelberg, pp. 194–210.

[125] Yonezawa, A., Briot, J.-P., and Shibayama, E. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*. *Sigplan Notices 21*, 11 (Nov. 1986), 258–268.

Proofs

Proofs Chapter 3

Lemma 3.1 (p. 31)

Let e *be an expression, σ a local state and ρ a heap, such that $\underline{\mathtt{e}} = [\![\mathtt{e}]\!]_{(\rho}^{\sigma)}$ is defined. If σ and ρ contain no symbolic values, then* $\underline{\mathtt{e}}$ *is a semantic value.*

Proof. Induction over e.

Base Cases In the case of constants (n, `True`, `False`, `Never`, `unit`, `Nil`) the rule assigned a fixed semantic value. In the case of accesses (v, **this**`.f`) the state is read, which contains no symbolic values by assumption.

Induction Cases This cases are all analogous: if the subexpression is not symbolic, which is checked explicitly and ensured by the induction hypothesis, then it is naturally evaluated further with additional operators.

□

Theorem 1 (p. 44)

Let Prgm *be a program and γ a global trace with* Prgm $\Downarrow \gamma$. *γ contains no symbolic values.*

Proof. We use an induction on the length of the run generating γ to show that $\gamma[1..i]$ contains no symbolic values.

Base case $i = 1$ The first global state is the initial state defined in Def. 3.19 with initial objects defined in Def. 3.14.

- For the non-called objects it suffices to check the initial heaps. The fields are mapped to evaluated expressions that contain only constants. Thus, the evaluation is independent of any symbolic value and by Lemma 3.1 the result of the evaluation is a semantic value. The parameters are mapped to non-symbolic object names.
- For the called object we additionally check the local state. It maps variables to their (non-symbolic) default value and parameters to the evaluation of an expression that is well-typed in the main block. I.e., it may only contain constants and by Lemma 3.1 the result of the evaluation is a semantic value.

Induction case $i > 1$ By induction hypothesis the trace $\gamma[1..i-1]$ contains no symbolic value. As every step adds exactly one event, we must show that if a state without symbolic values $\mathbf{S}_j$ is reached during a run, then a step of the GC semantics of a system of the form

$$\mathbf{S}_j \xrightarrow{\mathsf{ev}} \mathbf{S}_{j+1}$$

preserves this property. I.e., neither ev nor $\mathbf{S}_{j+1}$ contain symbolic values. We make a case distinction on the applied rule in Def. 3.20.

(S-Internal) We make another case distinction on the rule executed by $\mathbf{O}$. In any case, we must show that $\mathsf{procpool}\ \rho$-agrees on some θ that contains no symbolic values under some event ev that contains no symbolic values. We observe that θ always has the form $\left\langle \binom{\sigma}{\rho}, \mathsf{ev}, \binom{\sigma'}{\rho'} \right\rangle$ where $\binom{\sigma}{\rho}$ contains no symbolic value (as it is chopped to the object trace. We need only show that the local semantics of statements, when evaluated in a state $\binom{\sigma}{\rho}$ without symbolic values, generates traces with a prefix $\left\langle \binom{\sigma}{\rho}, \mathsf{ev}, \binom{\sigma'}{\rho'} \right\rangle$ where ev and $\binom{\sigma'}{\rho'}$ only contain those symbolic values ev introduces. These values are replaced by values from $\gamma[1..i-1]$, so they cannot be symbolic.

Assignments The event contains only semantic effects, which are not symbolic. The object state $\binom{\sigma}{\rho}$ coincides with the concrete object state $\binom{\sigma}{\rho}$ everywhere except the left-hand-side. It is, however, assigned a value that is the result of an evaluation of an expression in a state without symbolic values.

Termination The state does not change, we must only consider the event itself:

- Effects are always concrete and the return value its again the result of an evaluation of an expression in a state without symbolic values.
- For method, object and destiny we observe that the process must be added by a previous application of **(O-Add)** which evaluates with a concrete object, destiny and method name. The process must also be scheduled before and by induction hypothesis we know that **(O-Schedule)** concretizes the heap.

Branching The added state is identical to the first one and, thus, contains no symbolic values. The added event only has semantic effects which are never symbolic.

Synchronization Here, $\underline{f}$ and $\underline{\mathtt{n}}$ are introduced. Everything else is concrete, because it is evaluated in the old state.

Method Call Here, $\underline{f}$ is introduced. Everything else is concrete, because it is evaluated in the old state.

Suspensions Here, we must consider the three elements before $\diamond$ and the three elements after $\diamond$. However in both cases the third state is identical to the first one, so if the resulting triple is added in a state where the first one is concrete, then the third state is also concrete.

Branching are just unions of trace ans for loops no case is needed as its semantics is defined in terms of recursive branchings.

(S-Get) This is analogous to **(S-Internal)** for the state. For the event, we observe that we access the previous events to generate f and $\mathtt{m}$ and both are, thus, not symbolic.

(S-Invoc) Object $\mathbf{O}_1$ is analogous to **(S-Internal)**, object $\mathbf{O}_2$ is analogous to the initial case, where we know that all parameters evaluate to semantic values because they are evaluated in $\mathbf{S}_j$, which contains no symbolic value itself. □

Proofs Chapter 4

Lemma 4.1 (p. 59)

Let $\mathbb{T}$ be a behavioral specification closed under negation. The following equivalences hold:

$$\neg\langle \mathsf{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \neg_{\mathbb{T}} \tau \rangle \equiv [\mathsf{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau]$$

$$\langle \mathsf{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau \rangle \equiv \neg[\mathsf{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \neg_{\mathbb{T}} \tau]$$

Furthermore, if a statement $\mathtt{s}$ *does not terminate, then* $\langle \mathtt{s} \Vdash^{\alpha_{\mathbb{T}}} \tau \rangle$ *is false for all behavioral specifications* $\mathbb{T}$ *and all* $\tau \in \tau_{\mathbb{T}}$.

Proof. We derive the first equivalence as follows. The negation in the first line is the one of BPL, the one in the second line the one of lMSOT, in the third line the one of the descriptive meta-logic.

$$\begin{aligned}
\neg\langle \mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \neg_{\mathbb{T}}\tau \rangle &\iff \neg\exists\theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \alpha_{\mathbb{T}}(\neg_{\mathbb{T}}\tau_{\mathbb{T}}) \\
&\iff \neg\exists\theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \neg\alpha_{\mathbb{T}}(\tau_{\mathbb{T}}) \\
&\iff \neg\exists\theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \neg(\theta, I, \beta \models \alpha_{\mathbb{T}}(\tau_{\mathbb{T}})) \\
&\iff \forall\theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \alpha_{\mathbb{T}}(\tau_{\mathbb{T}}) \iff [\mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau]
\end{aligned}$$

The second one is analogous.

$$\begin{aligned}
\neg[\mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \neg_{\mathbb{T}}\tau] &\iff \neg\forall\theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \alpha_{\mathbb{T}}(\neg_{\mathbb{T}}\tau_{\mathbb{T}}) \\
&\iff \neg\forall\theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \neg\alpha_{\mathbb{T}}(\tau_{\mathbb{T}}) \\
&\iff \neg\forall\theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \neg(\theta, I, \beta \models \alpha_{\mathbb{T}}(\tau_{\mathbb{T}})) \\
&\iff \exists\theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \alpha_{\mathbb{T}}(\tau_{\mathbb{T}}) \iff \langle \mathtt{s} \overset{\alpha_{\mathbb{T}}}{\Vdash} \tau \rangle
\end{aligned}$$

For the last point we observe that the set $\mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right)$ is empty if $\mathtt{s}$ does not terminate. □

Lemma 4.2 (p. 60)

Let φ *be a* BPL *formula containing only* $\mathbb{T}_{\mathsf{bpst}}$ *modalities. Its satisfiability relation is well-founded.*

Proof. Any formula has a finite nesting depth. The proof is an induction on this depth. A modality with nesting depth 0 is simply a $\mathbb{T}_{\mathsf{pst}}$ modality and for the induction step it suffices that the satisfiability of a formula with nesting depth n only requires the satisfiability of subformulas with nesting depth $n-1$. □

Lemma 4.3 (p. 60)

Nested modalities for postconditions adhere to the sequential composition axiom [66].

$$\left[\mathtt{s}_1 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} [\mathtt{s}_2 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \varphi]\right] \equiv \left[\mathtt{s}_1; \mathtt{s}_2 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \varphi\right]$$

Proof.

$$
\begin{aligned}
&\binom{\sigma}{\rho}, I, \beta \models \left[\mathtt{s}_1 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} [\mathtt{s}_2 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \varphi]\right] \\
\iff &\forall \theta' \in \mathsf{selected}\left(\mathtt{s}_1, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta', I, \beta \models [\mathsf{last}] \vdash [\mathtt{s}_2 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \varphi] \\
\iff &\forall \theta' \in \mathsf{selected}\left(\mathtt{s}_1, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \mathsf{last}(\theta'), I, \beta \models [\mathtt{s}_2 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \varphi] \\
\iff &\forall \theta' \in \mathsf{selected}\left(\mathtt{s}_1, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \forall \theta'' \in \mathsf{selected}\left(\mathtt{s}_2, \mathtt{m}, \mathsf{last}(\theta')\right).\ \mathsf{last}(\theta''), I, \beta \models \varphi \\
\iff &\forall \theta' \in \mathsf{selected}\left(\mathtt{s}_1\,;\mathtt{s}_2, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \mathsf{last}(\theta''), I, \beta \models \varphi \\
\iff &\binom{\sigma}{\rho}, I, \beta \models \left[\mathtt{s}_1\,;\mathtt{s}_2 \overset{\alpha_{\mathsf{bpst}}}{\Vdash} \varphi\right]
\end{aligned}
$$

□

Lemma 4.5 (p. 63)

Let P, P' be two sets of programs and $\mathtt{m}$ be a method.

1. *If a formula is $\mathtt{m}$-P-valid and $\mathtt{m}$-P'-valid, then it is $\mathtt{m}$-$P \cup P'$-valid.*
2. *If $P \subseteq P'$ and a formula is $\mathtt{m}$-P'-valid, then it is $\mathtt{m}$-P-valid.*
3. *If $P \subseteq P'$ and a rule is $\mathtt{m}$-P'-sound, then it is $\mathtt{m}$-P-sound.*

Proof. We stress that we use method names to identify whole methods, i.e., $\mathtt{m}$ is fully qualified and has the same implementation in every program.

1. Every trace generated by $\mathtt{m}$ in a program in $P \cup P'$ is generated either by a program in P or P'.
2. Every trace generated by $\mathtt{m}$ in a program in P is also generated by a program in P'.
3. This follows directly from 2.

□

Lemma 4.6 (p. 67)

$\mathbb{T}_{\mathsf{sinv}}$ *is sound.*

Proof. In our system every rule application is inside a proof with the full method body at its root. In this proof, we omit the interpretation from the satisfiability relation, as it does never change.

(assignV) We may assume that for every $\binom{\sigma}{\rho}$ in every trace θ of $\mathtt{m}$ the premise holds:

$$\binom{\sigma}{\rho}, \beta \models \bigwedge \Gamma \to \{U\}\{\mathtt{v} := \mathtt{e}\}[\mathtt{s} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I] \vee \bigvee \Delta$$

We must show that for every $\binom{\sigma^c}{\rho^c}$ in every trace θ of $\mathtt{m}$ the conclusion holds:

$$\binom{\sigma^c}{\rho^c}, \beta_c \models \bigwedge \Gamma \to \{U\}[\mathtt{v = e;\ s}] \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I] \vee \bigvee \Delta$$

We distinguish two cases: either the premises holds independently of the highlighted formula or not. In the first case the conclusion obviously holds in $\binom{\sigma^c}{\rho^c}$ as the same context Γ, Δ is available. In the second case we must show that

$$\binom{\sigma}{\rho}, \beta \models \{U\}\{\mathtt{v} := \mathtt{e}\}[\mathtt{s} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I]$$

implies

$$\binom{\sigma^c}{\rho^c}, \beta_c \models \{U\}[\mathtt{v = e;\ s} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I]$$

For this we observe that all traces of `v = e; s` after U have the form

$$\left\langle U\left(\binom{\sigma_c}{\rho_C}\right), \mathsf{noEv}, U\left(\binom{\sigma_c[\mathtt{v} \mapsto \mathtt{e}]}{\rho_c}\right)\right\rangle ** \theta$$

On premise-side, all traces of `s` after $\{U\}\{\mathtt{v} := \mathtt{e}\}$ have the form

$$\left\langle U\left(\binom{\sigma^c}{\rho^c[\mathtt{v} \mapsto \mathtt{e}]}\right)\right\rangle ** \theta$$

By validity of the premise, we may for choose for every trace of `v = e; s` after U a fitting trace of `s` *as they are in the same method* and, thus, every trace of `s` starts in a state after `v = e`. We may ignore the variable assignment, as all invariants are sentences.

(return) We must show that

$$\binom{\sigma}{\rho}, \beta \models \{U\}[\textbf{return}\ \mathtt{e} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I]$$

holds for every state $\binom{\sigma}{\rho}$. Every trace of **return** `e` has the form $\left\langle U(\binom{\sigma}{\rho}), \mathsf{futEv}, U(\binom{\sigma}{\rho})\right\rangle$, we must show that

$$U\left(\binom{\sigma}{\rho}\right), \beta \models \bigwedge I \iff \binom{\sigma}{\rho}, \beta \models \bigwedge \{U\} I$$

which is shown by the premise.

(awaitB) For this rule our notion of soundness deviates from the standard one. We have to show that for every $\binom{\sigma^c}{\rho^c}$ in every trace θ of `m` the conclusion holds:

$$\binom{\sigma^c}{\rho^c}, \beta_c \models \bigwedge \Gamma \to \{U\}[\textbf{await}\ \mathtt{e_p};\ \mathtt{s} \overset{\alpha_{\mathsf{sinv}}}{\Vdash} I] \vee \bigvee \Delta$$

By the semantics this requires two parts: (1) the next state is a model for I and (2) the trace of the continuation is a model for $\alpha_{\mathsf{sinv}}(I)$. The first part directly corresponds to the first premise, analogously to **(return)**. The second one corresponds to the second premise, but we must justify that adding $\{U\}\{\mathtt{heap} := \mathsf{anon}(\mathtt{heap})\}(I \wedge \mathtt{e})$ to the antecedent is sound, i.e., that every trace of `s` is starting in a state where this condition holds. The guard `e` holds by the semantics of the language, the local variables must not change as the local state is not modified by other processes. It remains to show that I may be assumed.

We observe that the trace starts in an object state with a heap that is the heap of either a previous suspension or termination. If the suspension or termination was executed by another method, then I holds as all other methods preserve the object invariant (due to our notion of validity).

If the suspension was issued by the method we are analyzing, then we require a more complex approach. The reason is that two processes of this method may interleave (i.e., the first suspends and the second reactivates). We first narrow down the situation.

- The suspension was issued by the same process as the reactivation. But then the first premise suffices to guarantee that I holds.
- The suspension was issued by another process, but at the same statement. Again the first premise suffices, as the partial proof so far describes both local traces.
- The termination was issued by another process and the **return** statement is part of $\mathtt{s}$. Again, this is covered by the premise.

The final case is the one where the termination or suspension was issued by another process of the same method that is not contained in $\mathtt{s}$. By a symmetrical argument, we may also assume that the program point $\mathtt{p}$ is not contained in the continuation of the other suspension.

This is the case that requires the move from soundness to enablement. We make an induction on n, the number of suspensions and termination from processes of the method in question before the reactivation in question to prove that all these suspensions establish the object invariant.

Base case $n = 1$ We must show that the first suspension or termination establishes the invariant. This only requires that other methods establish the object invariant at their suspension. By assumption this holds and so the first suspension or termination depends only on the soundness of the non-suspending rules.

Base case $n > 1$ We must show that the n-th suspension of termination established the invariant. By induction hypothesis the first $n-1$ suspensions and terminations did so. I.e., all previous application of the rule were sound. This means that at this point we can assume the object invariant.

Now, if every suspension and termination establishes the object invariant, then the second premise indeed describes all possible continuations.

Rules (while) is standard, rule (awaitF) is analogous to (awaitB)[1], the others are analogous to (assignV). □

Lemma 4.7 (p. 67)

Let Prgm *be a program with objects* $\mathtt{x}_1, \ldots, \mathtt{x}_n$ *of class* $\mathtt{C}_1, \ldots, \mathtt{C}_n$. *Let* φ_{C} *be the class invariant of class a* C. *If (1) for every class* $\mathtt{C}_i$, *every proof obligation* $\iota^{\varphi}_{\mathsf{sinv}}(\mathtt{m})$ *for every method* $\mathtt{m}$ *within* $\mathtt{C}_i$ *can be proven and (2) the initial values of the fields establish* φ_{C}. *Then for every* $i \leq n$ *the invariant* $\varphi_{\mathtt{C}_i}$ *holds at every point* $\mathtt{x}_i$ *has no active process. I.e., every run of* Prgm *is a model for the following formula*

$$\bigwedge_{\mathsf{C}\ in\ \mathsf{Prgm}} \forall \mathtt{x} \in \mathsf{C}.\ \forall i \in \mathbf{I}.\ [i] \doteq \mathsf{release}(i, \mathtt{x}) \rightarrow [i+1] \vdash \varphi_{\mathsf{C}}[\mathbf{this} \setminus \mathtt{x}]$$

Proof. The first state of each class is defined by its field initializations, every field $\mathtt{f}$ is initialized with a constant $\mathtt{e}_{\mathtt{f}}$. Thus, it can be checked whether the first state satisfies the object invariant with proving validity of

$$\{\mathtt{f}_1 := \mathtt{e}_{\mathtt{f}_1}\} \ldots \{\mathtt{f}_n := \mathtt{e}_{\mathtt{f}_n}\} \varphi_{\mathsf{C}}$$

This establishes that the first state satisfies the invariant. For the other states we observe that as the rule is sound, each method preserves the method invariant, if we can assume it — i.e., we do not require that every other trace is a model for its specification (as is expressed in $\mathtt{m}$-P-soundness), but only that every trace *so far* is a model for its specification. The rest of the proof is a simple induction over the number i of run processes to establish that the i-th process adheres to its specification. For $i = 1$ we use that the initialization established the invariant, for $i > 1$ we use the induction hypothesis that the first $i-1$ processes realize the specification and thus the i-th may assume the object invariant. □

[1] To be precise: the proofs must be done in parallel for the induction.

Lemma 4.8 (p. 68)

The following sequents are P-valid

$$\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha}{\Vdash} \tau], \Delta \qquad \Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha'}{\Vdash} \tau'], \Delta$$

iff the following sequent is P-valid

$$\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha \between \alpha'}{\Vdash} \tau \between \tau'], \Delta$$

Proof.

$$\begin{aligned}
&\forall \binom{\sigma}{\rho}.\ \forall \beta.\ \binom{\sigma}{\rho}, I, \beta \models [\mathtt{s} \overset{\alpha}{\Vdash} \tau] \wedge \forall \binom{\sigma}{\rho}.\ \forall \beta.\ \binom{\sigma}{\rho}, I, \beta \models [\mathtt{s} \overset{\alpha'}{\Vdash} \tau'] \\
\Longleftrightarrow\ &\forall \binom{\sigma}{\rho}.\ \forall \beta.\ \forall \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \alpha(\tau) \wedge \\
&\forall \binom{\sigma}{\rho}.\ \forall \beta.\ \forall \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \alpha'(\tau') \\
\Longleftrightarrow\ &\forall \binom{\sigma}{\rho}.\ \forall \beta.\ \forall \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models (\alpha(\tau) \wedge \alpha'(\tau')) \\
\Longleftrightarrow\ &\forall \binom{\sigma}{\rho}.\ \forall \beta.\ \forall \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \binom{\sigma}{\rho}\right).\ \theta, I, \beta \models \alpha \between \alpha'(\tau \between \tau') \\
\Longleftrightarrow\ &\forall \binom{\sigma}{\rho}.\ \forall \beta.\ \binom{\sigma}{\rho}, I, \beta \models [\mathtt{s} \overset{\alpha \between \alpha'}{\Vdash} \tau \between \tau']
\end{aligned}$$

□

Theorem 2 (p. 71)

Let (r1) and (r2) be two rules according to Def. 4.22. Furthermore, let (r1) be P-sound and leading and (r2) be P′-sound and basic-local. (r1 ◊ r2) is P ∩ P′-sound.

Proof. We may assume that both rules have only one basic premise, since we can merge them easily. We must show that

$$\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha \between \alpha'}{\Vdash} \tau \between \tau'], \Delta$$

is $P \cap P'$-valid if all premises are $P \cap P'$-valid.

By Lemma 4.8 the conclusion is $P \cap P'$-valid if the following two sequents are $P \cap P'$-valid:

$$\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha}{\Vdash} \tau], \Delta \tag{A.1}$$

$$\Gamma \Rightarrow \{U\}[\mathtt{s} \overset{\alpha'}{\Vdash} \tau'], \Delta \tag{A.2}$$

Sequent A.1 This shows that the combined rule is valid with respect to the leading type.

We use the following implication chain for the sequent (A.1):

$$\begin{aligned}
&\text{all premises of (r1 ◊ r2) are } P \cap P'\text{-valid} \\
\Rightarrow &\text{all premises of (r1) are } P \cap P'\text{-valid} && \text{(A.3)} \\
\Rightarrow &\text{conclusion of (r1) is } P \cap P'\text{-valid} && \text{(A.4)}
\end{aligned}$$

The conclusion of (r1) is exactly sequent (A.1). By assumption rule (r1) is P-sound, thus, implication (A.4) holds, since $P \cap P' \subseteq P$ and Lemma 4.5. We must, thus, show implication (A.3), i.e., that $P \cap P'$-validity of all premises of (r1 ◊ r2) implies $P \cap P'$-validity of all premises of (r1) .

- The basic premise of (r1) is P-valid, and as $P \cap P' \subseteq P$ and by Lemma 4.5, it is also $P \cap P'$-valid.
- The same argument applies for the premise added by step 4 in the construction.
- For all other premises in (r1 ◊r2), represented by $\mathfrak{s}' = \left(\Phi, U, [\mathsf{s} \Vdash^{\alpha_1 \Diamond \alpha_2} \tau_1 \Diamond \tau_2]\right)$, there is a premise represented by $\mathfrak{s} = (\Phi, U, [\mathsf{s} \Vdash^{\alpha_1} \tau_1])$ in (r1). Validity of $\mathfrak{s}'$ implies validity of $\mathfrak{s}$, by Lemma 4.8.
- The validity of the premise added by step 2 is not used in this case.

Sequent A.2 This shows that the combined rule is valid with respect to the led type.

We are required to show the validity of sequent (A.2) in the restricted context. I.e.

$$\forall \binom{\sigma}{\rho}.\ \forall \beta.\ \left([\![U]\!]_{\binom{\sigma}{\rho},I,\beta}(\binom{\sigma}{\rho}), I, \beta \models \Gamma \wedge \neg\Delta \right) \rightarrow [\![U]\!]_{\binom{\sigma}{\rho},I,\beta}(\binom{\sigma}{\rho}), I, \beta \models [\mathsf{s} \overset{\alpha'}{\Vdash} \tau']$$

As (r2) is basic-local and sound, the three conditions of basic locality must hold for the restricted context to establish the validity of sequent (A.2): We observe that the side-conditions still hold and that the modified premise added by step 2 establishes that ψ holds in any of the states in the restricted context, because it adds the same context to it as to the conclusion

$$\forall \binom{\sigma}{\rho}.\ \forall \beta.\ \left([\![U]\!]_{\binom{\sigma}{\rho},I,\beta}\left(\binom{\sigma}{\rho}\right), I, \beta \models \Gamma \wedge \neg\Delta \right) \rightarrow [\![U]\!]_{\binom{\sigma}{\rho},I,\beta}\left(\binom{\sigma}{\rho}\right), I, \beta \models \psi$$

It remains to show the third condition: that there is an $i \leq m$ and an $\binom{\sigma'}{\rho'}$ such that for each $\theta' \in [\![\mathsf{s}_i]\!]_{\binom{\sigma'}{\rho'},I,\beta}$ that is a suffix of θ, $\theta' \models \alpha(\tau_i)$ holds. Each of the original premises is either still available directly (step 4) without context (and so established the property for any context) or added in the new context in the composed modality. The new context comes from a leading rule. So each state $\binom{\sigma'}{\rho'}$ starting a prefix of some s_i is described by the added update. Thus, validity of the premises of (r1 ◊r2) establishes validity of the sequent (A.2).

As the premises of (r1 ◊r2) establish the $P \cap P'$-validity of the sequents (A.1) and (A.2), they also establish the $P \cap P'$-validity of the conclusion of (r1 ◊r2). □

Lemma 4.9 (p. 72)

Let (r1) and (r2) be two rules according to Def. 4.22. Furthermore, let (r1) be P-sound and leading, (r2) be P'-sound and basic-local. (r1 ◊r2) is leading.

Proof. As (r1) is leading for every state $\binom{\sigma}{\rho}$ with $\binom{\sigma}{\rho} \models \bigwedge_{i \leq n} \varphi_i \wedge \bigwedge_{i \leq n'} \psi_i$, every variable assignment β and every trace $\theta \in [\![\mathsf{s}]\!]_{\binom{\sigma}{\rho},I,\beta}$ there is a $\binom{\sigma'}{\rho'}$ such that there is a $i \leq m$ and a trace $\theta' \in [\![\mathsf{s}_i]\!]_{\binom{\sigma'}{\rho'},I,\beta}$ that is (1) a suffix of θ and (2) $\theta' \models \alpha(\tau_i)$, and (3) $\binom{\sigma'}{\rho'} = [\![U_i']\!]_{\binom{\sigma}{\rho},I,\beta}(\binom{\sigma}{\rho})$.

We have to show that for every $\binom{\sigma}{\rho}$ with $\binom{\sigma}{\rho} \bigwedge_{i \leq n} \varphi_i \wedge \bigwedge_{i \leq n'} \psi_i$ and every trace $\theta \in [\![\mathsf{s}]\!]_{\binom{\sigma}{\rho},I,\beta}$ there is a $\binom{\sigma'}{\rho'}$ such that there is a s_k that is the statement of one of the new premises and a trace $\theta' \in [\![\mathsf{s}_k]\!]_{\binom{\sigma'}{\rho'},I,\beta}$ that is (1) a suffix of θ and (2) $\theta' \models \alpha(\tau_i \Diamond \tau_j')$, and (3) $\binom{\sigma'}{\rho'} = [\![U_k']\!]_{\binom{\sigma}{\rho},I,\beta}(\binom{\sigma}{\rho})$.

This is easily done by choosing s_k mirroring the choice made for (r1): (1) Its trace is still a suffix for every trace, because the statement in the conclusion does not change and the new premises do not add new starting states. It is a model for some $\tau_i \Diamond \tau_j'$ (note that there may be multiple choices for j, but for at least one). And finally, it is reachable by the same update (which is the same for all choices of j). □

Lemma 4.10 (p. 72)

The following rewrite rule is sound: $[\mathtt{s} \Vdash^{\alpha} \tau] \leftrightsquigarrow [\mathtt{s} \Vdash^{\alpha_{\mathsf{pst}} \between \alpha} \mathsf{true} \between \tau]$.

Proof.

$$
\begin{aligned}
[\mathtt{s} \overset{\alpha_{\mathsf{pst}} \between \alpha}{\Vdash} \mathsf{true} \between \tau] &\iff \forall \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \begin{pmatrix}\sigma \\ \rho\end{pmatrix}\right). \theta, I, \beta \models \alpha_{\mathsf{pst}} \between \alpha(\mathsf{true} \between \tau) \\
&\iff \forall \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \begin{pmatrix}\sigma \\ \rho\end{pmatrix}\right). \theta, I, \beta \models \alpha_{\mathsf{pst}}(\mathsf{true}) \wedge \alpha(\tau) \\
&\iff \forall \theta \in \mathsf{selected}\left(\mathtt{s}, \mathtt{m}, \begin{pmatrix}\sigma \\ \rho\end{pmatrix}\right). \theta, I, \beta \models \alpha(\tau) \iff [\mathtt{s} \overset{\alpha}{\Vdash} \tau]
\end{aligned}
$$

The step which drops $\alpha_{\mathsf{pst}}(\mathsf{true})$ is sound because true does not contain `result`. □

Theorem 3 (p. 73)

Let (r1) *and* (r2) *be two loop invariant rules according to Def. 4.27. Furthermore, let* (r1) *be P-sound and leading,* (r2) *be P'-sound and basic-local.* (r1 ◊ r2) *is $P \cap P'$-sound and leading.*

Proof. We have to show that every trace of

while`(e){s}s'`

is a model for

$$\alpha \between \alpha'(\tau \between \tau') \equiv \alpha(\tau) \wedge \alpha'(\tau')$$

if all premises are valid.

- For $\alpha(\tau)$ we observe that the original premises are all present from the original rule (r1), which is sound by assumption.
- For $\alpha'(\tau')$ we observe that the original premises are all present in a different context. However, as (r1) is providing the context we can state that every trace generated by **while**`(e){s}s'` can be split up into n segments where the first $n-1$ segments are traces generated by the method body. The first $n-1$ segments are all models for τ_1, the last is a model for τ_2, at the beginning of each the first $n-1$ segment φ_1 holds and at the beginning of the last one φ_2 holds. Thus, when restricting the context of the premises of (r2), we can assume the same — the traces which are covered by (r2) but not (r1 ◊ r2) are not generated by any run of any program in $P \cap P'$, as otherwise (r1) would not be P-sound. □

Theorem 4 (p. 74)

Let (r1) *and* (r2) *be two rules according to Def. 4.28. Furthermore, let* (r1) *be P-sound and* (r2) *be P'-sound.* (r1 ◊ r2) *is $P \cap P'$-sound and leading.*

Proof. The construction we give is symmetrical up to the use ◊ for semantics and syntax, for which symmetry follows from the symmetry of conjunction. For soundness, the proof is identical to the cases concerned with (r1) of theorem 2 and applying symmetry for (r2). For leadership, the proof is analogous to lemma 4.9 — if it were not leading, there would be a trace starting that is not covered by any premise, but all premises are still present. While new knowledge may be added (because both φ_i and ψ_j are now guarding the modality), we note that leadership is defined relative to the formulas in the antecedent of premises. □

Theorem 5 (p. 74)

If $\mathbb{T}_1$ and $\mathbb{T}_2$ are sound, all sound rules of $\mathbb{T}_1$ are leading and all sound rules of $\mathbb{T}_2$ are either leading or basic-local then all rules of $\mathbb{T}_1 \between \mathbb{T}_2$, that are not composed of enabling but unsound rules, are sound.

Proof. This follows directly from theorems 2, 3 and 4. □

Lemma 4.11 (p. 78)

$\mathbb{T}_{\mathsf{pst}}$ is sound and leading. $\mathbb{T}_{\mathsf{inv}}$ is sound.

Proof. $\mathbb{T}_{\mathsf{inv}}$ is a less precise variant of $\mathbb{T}_{\mathsf{sinv}}$, $\mathbb{T}_{\mathsf{pst}}$ is the standard postcondition calculus and we refer to [8]. □

Lemma 4.12 (p. 78)

$\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}^I_{\mathsf{inv}}$ is sound and leading. Lemma 4.7 holds for $\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}^I_{\mathsf{inv}}$ under the same assumptions, as long as the obligation scheme ι_{pst} uses `true` *as its precondition.*

Proof. Soundness follows directly from the composition theorem for all rules, but ($\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}^I_{\mathsf{inv}}$-awaitB) and ($\mathbb{T}_{\mathsf{pst}} \between \mathbb{T}^I_{\mathsf{inv}}$-awaitB), which are identical to $\mathbb{T}_{\mathsf{sinv}}$, as is the proof of Lemma 4.7. □

Proofs Chapter 5

Lemma 5.1 (p. 83)

Type $\mathbb{T}_{\mathsf{fr}}$ is sound, all rules are basic-local and none of the rules is enabling but not sound.

Proof. ($\mathbb{T}_{\mathsf{fr}}$-assignV) Every trace of `v = e; s` has the form

$$\theta = \left\langle \begin{pmatrix}\sigma\\ \rho\end{pmatrix}, \mathsf{noEv}\big(\{\mathsf{v}_\mathsf{W}\} \cup \{\mathtt{f}_\mathsf{R} \mid \mathsf{loc}_\mathsf{R}(\mathtt{e})\}\big), \begin{pmatrix}\sigma'\\ \rho'\end{pmatrix} \right\rangle * \theta'$$

We must show that

$$\theta, \beta, I \models \alpha_{\mathsf{fr}}\big(\mathfrak{R}, \mathfrak{W}, \mathfrak{S}\big)$$

By validity of the premise we know that $\theta', \beta, I \models \alpha_{\mathsf{fr}}\big(\mathfrak{R}, \mathfrak{W}, \mathfrak{S}\big)$, by the side-condition we know that $\mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}$ and that no field is written, thus

$$\{\mathtt{f} \mid \mathtt{f}_\mathsf{R} \in \mathsf{loc}_\mathsf{R}(\mathtt{e})\} \subseteq \mathfrak{R} \qquad \emptyset \subseteq \mathfrak{W}$$

($\mathbb{T}_{\mathsf{fr}}$-assignF) Every trace of **this**`.f = e; s` has the form

$$\theta = \left\langle \begin{pmatrix}\sigma\\ \rho\end{pmatrix}, \mathsf{noEv}\big(\{\mathtt{f}_\mathsf{W}\} \cup \{\mathtt{f}_\mathsf{R} \mid \mathsf{loc}_\mathsf{R}(\mathtt{e})\}\big), \begin{pmatrix}\sigma'\\ \rho'\end{pmatrix} \right\rangle * \theta'$$

We must show that

$$\theta, \beta, I \models \alpha_{\mathsf{fr}}\big(\mathfrak{R}, \mathfrak{W}, \mathfrak{S}\big)$$

By validity of the premise we know that $\theta', \beta, I \models \alpha_{\mathsf{fr}}\big(\mathfrak{R}, \mathfrak{W}, \mathfrak{S}\big)$, by the side-condition we know that $\mathsf{loc}(\mathtt{e}) \subseteq \mathfrak{R}$ and $\mathtt{f} \in \mathfrak{W}$, thus

$$\{\mathtt{f} \mid \mathtt{f}_\mathsf{R} \in \mathsf{loc}_\mathsf{R}(\mathtt{e})\} \subseteq \mathfrak{R} \qquad \{\mathtt{f}\} \subseteq \mathfrak{W}$$

($\mathbb{T}_{fr}$-readV) Analogous to ($\mathbb{T}_{fr}$-assignV).

($\mathbb{T}_{fr}$-awaitB) Every trace of **await** $\mathtt{e_p}$; s has the form

$$\theta = \left\langle \begin{pmatrix}\sigma\\ \rho\end{pmatrix}, \mathsf{condEv}\left(\mathtt{x}, \mathit{dest}, \{\mathtt{p_E}\} \cup \mathsf{loc_R}(\mathtt{e})\right), \begin{pmatrix}\sigma\\ \rho\end{pmatrix}, \diamond . \begin{pmatrix}\sigma'\\ \rho'\end{pmatrix}, \mathsf{condREv}\left(\mathtt{x}, \mathit{dest}, \emptyset\right), \begin{pmatrix}\sigma'\\ \rho'\end{pmatrix} \right\rangle ** \theta'$$

We must show that $\{\mathtt{f} \mid \mathtt{f_R} \in \mathsf{loc_R}(\mathtt{e})\} \subseteq \mathfrak{R}$, which is again covered by the side-condition. After $\diamond$ the pair of $\mathfrak{S}(\mathtt{p})$ describes the read and write accesses before the next termination or suspension. This is established by the premise for θ'. For the condREv event it suffices to note that it has no effects at all.

($\mathbb{T}_{fr}$-awaitF) Analogous to ($\mathbb{T}_{fr}$-awaitB).

($\mathbb{T}_{fr}$-callV) Analogous to ($\mathbb{T}_{fr}$-assignV).

($\mathbb{T}_{fr}$-if) The traces of **if**(e){s}**else**{s'}s'' are exactly the traces of s;s'' and s';s'', whose type adherence is established by the premises, prefixed with a single noEv, whose effect set is checked by the side-condition.

($\mathbb{T}_{fr}$-skip) The traces of **skip** contain no event and are, thus, trivially described by the semantics.

($\mathbb{T}_{fr}$-skipCont) The traces of **skip**; s are the traces of s, whose adherence of the type is established by the premise.

($\mathbb{T}_{fr}$-return) Analogous to ($\mathbb{T}_{fr}$-assignV).

($\mathbb{T}_{fr}$-while) We distinguish between the cases whether the loop body contains an **await** or not. If it does not contain an **await**, the proof is analogous to ($\mathbb{T}_{fr}$-if) If it does contain at least one await, we observe that all traces θ of **while**(e){s}s' have the form

$$\theta_1 ** \ldots ** \theta_n ** \theta'$$

where θ' is generated by s' and each θ_i is generated by s. Similarly, the trace can also be separated at the markers. Each θ also has the form

$$\hat{\theta}_1 *** \ldots *** \hat{\theta}_n *** \hat{\theta}'$$

Such that each $\hat{\theta}_i$ contains no $\diamond$. For the last trace we can say that θ' is a suffix of $\hat{\theta}'$. Thus, the frame pair that holds there is the one specified by the last suspension point p. This suspension statement is within s, so the intersection over all suspension points (and the current pair) is a sound approximation for all traces of s'. This is established by the second premise.

For the first premise, we split each θ_i into three parts[2]:

$$\theta_i = \theta_i^{\mathsf{pre}} *** \theta_i^{\mathsf{inner}} *** \theta_i^{\mathsf{post}}$$

such that θ_i^{pre} and θ_i^{post} contain no $\diamond$. The active frame pair for θ_i^{pre} is the one of the p that splits off $\theta_{i-1}^{\mathsf{post}}$ (or is the currently active frame pair if $i = 0$). p occurs in s, so the first premise again soundly approximates the method body, except for the effects caused by the guard, which are explicitly checked by the side-condition. Note that $\theta_i^{\mathsf{inner}}$ and θ_i^{post} also fall under the method body.

All rules are sound, no rule is enabling but not sound. Basic-locality is trivial to check. □

[2] We omit $\theta_i^{\mathsf{inner}}$ if only one suspension occurs within θ_i.

Lemma 5.2 (p. 87)

Let $\mathfrak{R}$ and $\mathfrak{W}$ be two sets of fields of some class and let $\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}$ be defined as follows:

$$\begin{aligned}\mathsf{SE}_{\mathfrak{R},\mathfrak{W}} = \quad & \{\mathtt{f} \mapsto (\mathsf{RS}+\mathsf{RC})^* \mid \mathtt{f} \in \mathfrak{R}, \mathtt{f} \notin \mathfrak{W}\} \cup \{\mathtt{f} \mapsto (\mathsf{RS}+\mathsf{RC}+\mathsf{RW}+\mathsf{W})^* \mid \mathtt{f} \in \mathfrak{R}, \mathtt{f} \in \mathfrak{W}\} \\ & \cup \{\mathtt{f} \mapsto (\mathsf{W})^* \mid \mathtt{f} \notin \mathfrak{R}, \mathtt{f} \in \mathfrak{W}\} \cup \{\mathtt{f} \mapsto \mathsf{no} \mid \mathtt{f} \notin \mathfrak{R}, \mathtt{f} \notin \mathfrak{W}\}\end{aligned}$$

The sequential effect type $\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}$ captures the semantics of the frame type (R,W):

$$\alpha_{\mathsf{fr}}\big((\mathfrak{R},\mathfrak{W},\mathfrak{S})\big) \equiv \alpha_{\mathsf{eff}}\big(\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}\big)$$

Proof. For static frames we can simplify the semantics:

$$\begin{aligned}&\alpha_{\mathsf{fr}}\big((\mathfrak{R},\mathfrak{W},\mathfrak{S})\big) \\ \equiv &\forall i \in \mathbf{I}.\ \Big(\mathsf{isEvent}(i) \to \forall \mathtt{f} \in \mathsf{F}.\ \Big(\big(\mathtt{f}_{\mathsf{R}} \in \mathsf{eff}(i) \to \mathtt{f} \in \mathfrak{R}\big) \wedge \big(\mathtt{f}_{\mathsf{W}} \in \mathsf{eff}(i) \to \mathtt{f} \in \mathfrak{W}\big)\Big)\Big) \\ \equiv &\forall \mathtt{f} \in \mathsf{F}.\ \forall i \in \mathbf{I}.\ \Big(\mathsf{isEvent}(i) \to \Big(\big(\mathtt{f}_{\mathsf{R}} \in \mathsf{eff}(i) \to \mathtt{f} \in \mathfrak{R}\big) \wedge \big(\mathtt{f}_{\mathsf{W}} \in \mathsf{eff}(i) \to \mathtt{f} \in \mathfrak{W}\big)\Big)\Big) \\ \equiv &\forall i \in \mathbf{I}.\ \bigwedge_{\mathtt{f} \in \mathsf{F}} \Big(\mathsf{isEvent}(i) \to \Big(\big(\mathtt{f}_{\mathsf{R}} \in \mathsf{eff}(i) \to \mathtt{f} \in \mathfrak{R}\big) \wedge \big(\mathtt{f}_{\mathsf{W}} \in \mathsf{eff}(i) \to \mathtt{f} \in \mathfrak{W}\big)\Big)\Big) \qquad \text{(L)}\end{aligned}$$

The step L is sound because in a given program there are only finitely many fields. To simplify the semantics, we make a case distinction of the semantics of single fields.

- $\mathtt{f} \notin \mathfrak{R}, \mathtt{f} \notin \mathfrak{W}$. In this case the type of this field is no and

$$\alpha^{\mathtt{f}}_{\mathsf{eff}}\big(\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}(\mathtt{f})\big) \equiv \forall i \in \mathbf{I}.\ \mathsf{noeff}(i,\mathtt{f})$$

- $\mathtt{f} \in \mathfrak{R}, \mathtt{f} \notin \mathfrak{W}$. In this case the type of this field is $(\mathsf{RS}+\mathsf{RC})^*$ and

$$\alpha^{\mathtt{f}}_{\mathsf{eff}}\big(\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}(\mathtt{f})\big) \equiv \forall i \in \mathbf{I}.\ \mathsf{noeff}(i,\mathtt{f}) \vee \neg \mathtt{f}_{\mathsf{W}} \in \mathsf{eff}(i)$$

- $\mathtt{f} \in \mathfrak{R}, \mathtt{f} \in \mathfrak{W}$. In this case the type of this field is $(\mathsf{RS}+\mathsf{RC}+\mathsf{RW}+\mathsf{W})^*$ and

$$\alpha^{\mathtt{f}}_{\mathsf{eff}}\big(\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}(\mathtt{f})\big) \equiv \mathsf{true}$$

- $\mathtt{f} \notin \mathfrak{R}, \mathtt{f} \in \mathfrak{W}$. In this case the type of this field is $(\mathsf{W})^*$ and

$$\alpha^{\mathtt{f}}_{\mathsf{eff}}\big(\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}(\mathtt{f})\big) \equiv \forall i \in \mathbf{I}.\ \mathsf{noeff}(i,\mathtt{f}) \vee \neg \mathtt{f}_{\mathsf{R}} \in \mathsf{eff}(i)$$

Now, we can simplify the semantics of the sequential effects:

$$
\begin{aligned}
&\alpha_{\mathsf{eff}}(\mathsf{SE}_{\mathfrak{R},\mathfrak{W}})\\
\equiv&\bigwedge_{\mathtt{f}\in\mathsf{F}} \alpha^{\mathtt{f}}_{\mathsf{eff}}(\mathsf{SE}_{\mathfrak{R},\mathfrak{W}}(\mathtt{f}))[i \in \mathbf{I} \setminus \mathsf{isEvent}(i)\\
\equiv&\bigwedge_{\mathtt{f}\notin\mathfrak{R},\mathtt{f}\notin\mathfrak{W}} \forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \to \mathsf{noeff}(i,\mathtt{f}) \wedge\\
&\bigwedge_{\mathtt{f}\in\mathfrak{R},\mathtt{f}\notin\mathfrak{W}} \forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \to \big(\mathsf{noeff}(i,\mathtt{f}) \vee \neg \mathtt{f}_{\mathsf{W}} \in \mathsf{eff}(i)\big) \wedge\\
&\bigwedge_{\mathtt{f}\in\mathfrak{R},\mathtt{f}\in\mathfrak{W}} \forall i \in \mathbf{I}.\ \mathsf{true} \wedge\\
&\bigwedge_{\mathtt{f}\notin\mathfrak{R},\mathtt{f}\in\mathfrak{W}} \forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \to \big(\mathsf{noeff}(i,\mathtt{f}) \vee \neg \mathtt{f}_{\mathsf{R}} \in \mathsf{eff}(i)\big)\\
\equiv&\forall i \in \mathbf{I}.\ \mathsf{isEvent}(i) \to\\
&\bigwedge_{\mathtt{f}\notin\mathfrak{R},\mathtt{f}\notin\mathfrak{W}} \mathsf{noeff}(i,\mathtt{f}) \wedge \bigwedge_{\mathtt{f}\in\mathfrak{R},\mathtt{f}\notin\mathfrak{W}} \big(\mathsf{noeff}(i,\mathtt{f}) \vee \neg \mathtt{f}_{\mathsf{W}} \in \mathsf{eff}(i)\big) \wedge\\
&\bigwedge_{\mathtt{f}\in\mathfrak{R},\mathtt{f}\in\mathfrak{W}} \mathsf{true} \wedge \bigwedge_{\mathtt{f}\notin\mathfrak{R},\mathtt{f}\in\mathfrak{W}} \big(\mathsf{noeff}(i,\mathtt{f}) \vee \neg \mathtt{f}_{\mathsf{R}} \in \mathsf{eff}(i)\big)\\
\equiv&\forall i \in \mathbf{I}.\ \bigwedge_{\mathtt{f}\in\mathsf{F}} \Big(\mathsf{isEvent}(i) \to \big((\mathtt{f}_{\mathsf{R}} \in \mathsf{eff}(i) \to \mathtt{f} \in \mathfrak{R}) \wedge (\mathtt{f}_{\mathsf{W}} \in \mathsf{eff}(i) \to \mathtt{f} \in \mathfrak{W})\big)\Big) \qquad \square
\end{aligned}
$$

Lemma 5.3 (p. 88)

Type $\mathbb{T}_{\mathsf{eff}}$ is sound, all rules are basic-local and none of the rules is enabling but not sound.

Proof. **($\mathbb{T}_{\mathsf{eff}}$-assignV)** Every trace of `v = e; s` has the form

$$\theta = \left\langle \begin{pmatrix}\sigma\\ \rho\end{pmatrix}, \mathsf{noEv}\big(\{\mathtt{v}_{\mathsf{W}}\} \cup \{\mathtt{f}_{\mathsf{R}} \mid \mathsf{loc}_{\mathsf{R}}(\mathtt{e})\}\big), \begin{pmatrix}\sigma'\\ \rho'\end{pmatrix} \right\rangle ** \theta'$$

We must show that

$$\theta, \beta, I \models \alpha_{\mathsf{eff}}(\mathsf{SE})$$

By validity of the premise we know that

$$\theta', \beta, I \models \alpha_{\mathsf{eff}}\big(\mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RC})\big)$$

By definition of app we know that θ is a model for SE if it coincides with $\mathsf{app}(\mathsf{SE}, \mathsf{locs}(\mathtt{e}), \mathsf{RC})$ on all fields that do not occur in $\{\mathtt{f}_{\mathsf{R}} \mid \mathsf{loc}_{\mathsf{R}}(\mathtt{e})\}$ and that all fields that occur in this set have a type that starts with RC. by the same argument as for Lemma 5.1, $\mathsf{locs}(\mathtt{e})$ is the correct set to pass.

($\mathbb{T}_{\mathsf{eff}}$-assignF1), ($\mathbb{T}_{\mathsf{eff}}$-assignF2), ($\mathbb{T}_{\mathsf{eff}}$-readV), ($\mathbb{T}_{\mathsf{eff}}$-awaitF), ($\mathbb{T}_{\mathsf{eff}}$-awaitB) and **($\mathbb{T}_{\mathsf{eff}}$-callV)** These rules are all analogous to ($\mathbb{T}_{\mathsf{eff}}$-assignV), except that ($\mathbb{T}_{\mathsf{eff}}$-assignF1) and ($\mathbb{T}_{\mathsf{eff}}$-assignF2) check the written field and that the rule ($\mathbb{T}_{\mathsf{eff}}$-readV), ($\mathbb{T}_{\mathsf{eff}}$-awaitF) and ($\mathbb{T}_{\mathsf{eff}}$-awaitB) check against RS and not RC because their statement adds an execution effect $\mathtt{p}_{\mathsf{E}}$.

($\mathbb{T}_{\mathsf{eff}}$-if) The traces of `if(e){s}else{s'}s''` are exactly the traces of `s;s''` and `s',s''`, whose type adherence is established by the premises, prefixed with a single noEv, whose effect set is checked by the side-condition.

($\mathbb{T}_{\text{eff}}$-skip) The traces of **skip** contain no event and are, thus, trivially described by the semantics.

($\mathbb{T}_{\text{eff}}$-skipCont) The traces of **skip**; s are the traces of s, whose adherence of the type is established by the premise.

($\mathbb{T}_{\text{eff}}$-return) Again analogous to ($\mathbb{T}_{\text{eff}}$-assignV).

($\mathbb{T}_{\text{eff}}$-while) We observe that all traces θ of **while**(e){s}s' have the form

$$\theta_1 ** \ldots ** \theta_n ** \theta'$$

where θ' is generated by s' and each θ_i is generated by s. Each subtrace starts with a noEv with the reads from the guards as effects. The premises establish exactly that the type has the form $r^*.r'$ for every field and that each type of a field syntactically present in the guard starts with a read inside the Kleene star and continues with a read after the Kleene star. Furthermore they establish that the loop body follows the repeated type and the continuation of the program follows the continuation of the type.

Rewrite Rules
- $r \rightsquigarrow r.no$. If a trace is a model for $\alpha_{\text{eff}}(r.no)$, then it is also a model for $\alpha_{\text{eff}}(r)$, as we can pick the empty suffix for the semantics of the sequential composition in the rewritten type, as the trace consisting only of a state is a model for no.
- $r \leftrightsquigarrow r + r$. Follows from the idempotency of $\vee$ in lMSOT.
- $r_1 + r_2 \rightsquigarrow r_1$ and $r_1 + r_2 \rightsquigarrow r_2$. If a trace is a model for $\alpha_{\text{eff}}(r_1)$ or $\alpha_{\text{eff}}(r_2)$ then it is a model for $\alpha_{\text{eff}}(r_1 + r_2) = \alpha_{\text{eff}}(r_1) \vee \alpha_{\text{eff}}(r_2)$.
- $no.r \rightsquigarrow r$, $r^* \rightsquigarrow r.r^* + no$ and $no \rightsquigarrow no^*$. Follows again form the fact that empty traces are models for no.

All rules are sound, no rule is enabling but not sound. Basic-locality is trivial to check. □

Proofs Chapter 6

Lemma 6.1 (p. 100)

Given an inconsistent set of program point specifications $\{\mathcal{S}_{\text{id}_i}\}_{i\in I}$, a consistent set of program point specifications $\{\mathcal{S}'_{\text{id}_i}\}_{i\in I}$ can be generated such that for all program point specifications within $\{\mathcal{S}'_{\text{id}_i}\}_{i\in I}$.

- *The context sets succ_{id} and over_{id} are unchanged.*
- *The precondition/suspension assumptions φ_{id} are unchanged.*
- *The postcondition/suspension assertions χ'_{id} of the generated method specification imply the postcondition/suspension assertions χ_{id} of the original method specification.*

Proof. For all id, set for $\text{id}' \in \text{over}_{\text{id}}$ the new specification to $\chi'_{\text{id}} = \chi_{\text{id}'} \wedge \varphi_{\text{id}}$. For all $\text{id}' \in \text{succ}_{\text{id}}$ set $\chi'_{\text{id}} = \chi_{\text{id}'} \wedge (\varphi_{\text{id}}[\textbf{this} \setminus \textbf{last}] \rightarrow \varphi_{\text{id}})$. □

Lemma 6.2 (p. 103)

Type $\mathbb{T}_{\text{met}}$ is sound.

Proof. First, note that $\alpha_{\mathsf{met}}\big((\mathfrak{P},\mathfrak{M},\varphi)\big)$ only specifies states after invocation, resolving, suspending and reactivating events. Thus, ($\mathbb{T}_{\mathsf{met}}$-**assignV**), ($\mathbb{T}_{\mathsf{met}}$-**assignF**) and ($\mathbb{T}_{\mathsf{met}}$-**skip**) are sound, because they just throw away a prefix that does not issue any of these events. Rule ($\mathbb{T}_{\mathsf{met}}$-**if**) is sound because the two premises cover all traces of the conclusion. Rule ($\mathbb{T}_{\mathsf{met}}$-**while**) is sound because of the same argument as ($\mathbb{T}_{\mathsf{inv}}$-**while**).

In rule ($\mathbb{T}_{\mathsf{met}}$-**return**), the premise establishes that φ holds for every value of $\mathtt{e}$ in every state. In particular, it holds in the final state of **return** $\mathtt{e}$ in the conclusion. Analogously for ($\mathbb{T}_{\mathsf{met}}$-**callV**): the first premise establishes the call condition for the given parameters in every state, the second premise establishes that every trace of $\mathtt{s}$ follows the type. The rules ($\mathbb{T}_{\mathsf{met}}$-**awaitB**) and ($\mathbb{T}_{\mathsf{met}}$-**awaitF**) are enabling. The proof is analogous to ($\mathbb{T}_{\mathsf{inv}}$-**awaitB**) and ($\mathbb{T}_{\mathsf{inv}}$-**awaitB**): the main point is that given a closed proof that uses one of these rules, one can extract a proof that prunes the second premise and establishes that this specific **await** statement establishes the suspension assertion. The only difference to $\mathbb{T}_{\mathsf{inv}}$ is that instead of establishing the same object invariant everywhere, we can assume that the global constraint holds and thus the correct suspension assertion holds when reactivating (for the full proof). We omit the induction on reactivations and technical details, as this part is completely analogous to $\mathbb{T}_{\mathsf{inv}}$.

The last rule that remains is ($\mathbb{T}_{\mathsf{met}}$-**readV**). This rule is enabling. The first premise establishes that $\mathtt{p}$ reads from the specified set of methods. Every other method can be assumed to adhere to their contract, so the assumption that one of their postconditions holds for the read value is sound. The only critical part is the case where the **get** statement reads a future issued by another process of the same methods, which is analogous to the object invariants, but requires an induction on how often the **get** in question is executed before a given read. We again omit this tedious but straightforward technical detail. □

Lemma 6.3 (p. 103)

Let Prgm *be a coherent program. If (1) all proof obligations can be closed, (2) the initial values of a class establish the heap precondition of the methods with empty succeeds sets and (3) the method trace condition holds then every global trace γ of* Prgm *is a model for the following gMSOT formula.*

$$\bigwedge_{\mathsf{C}\ in\ \mathsf{Prgm}} \forall \mathtt{x} \in \mathsf{C}.\ \forall m \in \mathsf{M}.\ \forall i \in \mathbf{I}.\ \mathsf{isFutEv}(i,\mathtt{x},m) \rightarrow \bigwedge_{\mathtt{m}\ in\ \mathsf{C}} [i-1] \vdash m \doteq \mathtt{m} \rightarrow \widetilde{\chi_{\mathtt{m}}[\mathbf{this} \setminus \mathtt{x}]}$$

Proof. This is analogous to Lemma 4.7 □

Lemma 6.4 (p. 103)

Type $\mathbb{T}_{\mathsf{pst}}$ $\mathbb{T}_{\mathsf{inv}}$ $\mathbb{T}_{\mathsf{met}}$ $\mathbb{T}_{\mathsf{fr}}$ is sound and the global lemmas of the input types hold under their additional conditions.

Proof. Rule ($\mathbb{T}_{\mathsf{pimf}}$-**readV**) For the frame type, the soundness argument follows from ($\mathbb{T}_{\mathsf{fr}}$-**readV**). It is only necessary to show that it is enabling w.r.t $\mathbb{T}_{\mathsf{met}}$ and, indeed, it is enabling for the same reason that ($\mathbb{T}_{\mathsf{met}}$-**readV**) is enabling.

Rules ($\mathbb{T}_{\mathsf{pimf}}$-**awaitF**) and ($\mathbb{T}_{\mathsf{pimf}}$-**awaitB**) For the frame type, the soundness argument follows from ($\mathbb{T}_{\mathsf{fr}}$-**readV**). Concerning the other types, we can again extract partial proofs for the premises establishing the suspension assertion and object invariant, and make the same induction on the number of executions of a fixed **await** statement.

For all other rules soundness follows from the composition theorem. The global lemmas hold because they only rely on validity of formulas and thus only on the semantics of composition, which is a simple conjunction. □

Proofs Chapter 7

Lemma 7.1 (p. 121)

Let $\mathbf{G}$ *be a global type and* $\mathbf{G}_1, \mathbf{G}_2$ *be two actions with* $\mathbf{G}_1 \leq_{\mathbf{G}} \mathbf{G}_2$*. If the witness(es) of* $\mathbf{G}_1$ *and* $\mathbf{G}_2$ *are issued by the same object, then the witness(es) of* $\mathbf{G}_1$ *occur before the witness(es) of* $\mathbf{G}_2$ *in any model of* $\mathbf{G}$*.*

Proof. If $\mathbf{G}_1$ and $\mathbf{G}_2$ are issued by the same object and $\mathbf{G}_1 \leq_{\mathbf{G}} \mathbf{G}_2$, then by definition of $\leq_{\mathbf{G}}$ there is some sequential composition connecting $\mathbf{G}_1$ and $\mathbf{G}_2$. By definition of the semantics, sequential composition is always translated into a relativization of the indices of a given object. This means that if i is the index variable of the witness of $\mathbf{G}_1$ and j the index variable of the witness of $\mathbf{G}_2$, then $\mathsf{gcsem}(\mathbf{G})$ contains the conjunct $i < j$[3] and every model must in particular be a model for this conjunct, i.e., the witness of $\mathbf{G}_1$ is before the witness of $\mathbf{G}_2$. □

Lemma 7.2 (p. 139)

Let **GP** *be a projection candidate without repetitions with a coherent causality graph. Let* Prgm *be a program and* γ_1 *a global trace with* $\mathsf{Prgm} \Downarrow \gamma_1$*, such that* γ_1 *is a model for gcsem*(**GP**)*. Let* γ_2 *be another global trace with* $\mathsf{Prgm} \Downarrow \gamma_2$*, such that all events of* γ_2 *are witnesses in* γ_1*. The order of these events for every object is the same.*

Proof. First, as γ_1 can obviously realize only one branch, we may assume that **GP** contains no branchings. Second, we observe that as γ_2 contains the same events, it executed the same statements and if the lemma statement would not hold, then only because some invocation reaction or reactivation events are reordered (and the events issued by the atomic segment following the method start or suspension in question).

Now, assume that the lemma would not hold and there would be two events at indices $i_1 < j_1$ in γ_1 and indices $i_2 > j_2$ in γ_2 that are ordered differently. Without loss of generality we assume they are method starts, because we handle suspension analogously in causal graphs. As **GP** has a coherent causality graph, then there are causality paths from the projected local type of the action witnessed by i_1 (and i_2) and the projected local type of the action witnessed by j_1 (and j_2) according to Def. 7.20. As the **GP** contains no repetition, we can ignore the second constraint. We fix the path in the causality graph that corresponds to the chosen branches in the run of the protocol when generating γ_1. Causal-coherence ensures that the choice of branches is not relevant and such a path always exists. If there is such a path, then the order of i_1 and j_1 cannot be reversed, as every edge (v_1, v_2) connects two nodes where the witness for the action of v_1 causes the witness for the action of v_2:

- The first three cases in the definition of the causality graph model that v_1 is before v_2 in some local type. I.e., it is part of the same method, which executes its statements always in the same order.
- The cases for repetition cannot occur.
- The case for branching is obviously not reversible.
- The case for method call and method start, the events cannot be reversed, because by the semantics a method cannot start before being called.
- Analogously for `Put`: a future cannot be read or synchronized on, before being resolved.

If the graph is causal-coherent, the order of the witnessed events cannot be reverted, thus $i_2 > j_2$ cannot hold. □

[3] More precise: a chain of such conjuncts.

Lemma 7.3 (p. 139)

If a trace is a model for **GP***, then it is a model for some protocol in its unrolling and vice versa.*

Proof. For this lemma it suffices to note that we only consider finite traces — every model thus must repeat the unrolling a finite number of times. I.e., the set of indices of the last/first index in the semantics of the Kleene star is finite. □

Lemma 7.4 (p. 139)

Let **GP** *be a projection candidate with a coherent causality graph. Let* $\mathbf{GP}'$ *be any of the protocols in its unrolling. Let* Prgm *be a program and* γ_1 *a global trace with* $\mathsf{Prgm} \Downarrow \gamma_1$*, such that* γ_1 *is a model for gcsem*$(\mathbf{GP}')$*. Let* γ_2 *be another global trace of* $\mathbf{GP}'$ *with* $\mathsf{Prgm} \Downarrow \gamma_2$*, such that all events of* γ_2 *are witnesses in* γ_1*. The order of these events for every object is the same.*

Proof. We need only show that repetition is handled correctly. Fixing the number of unrollings requires to show that the unrolling cannot overlap, this is ensured by the condition of causal-coherence that every suspension or method start in a repetition must have a path to itself in the causality graph. The causality graph of the unrolling thus contains paths between each copy *even if the edges from/to the Kleene star are removed*. As the unrolling contain no repetitions anymore, the correctness of the path is due to Lemma 7.2. □

Lemma 7.5 (p. 144)

Type $\mathbb{T}_{\mathsf{loc}}$ *is sound.*

Proof. It suffices to show that every rule is enabling. All rules but ($\mathbb{T}_{\mathsf{loc}}$-**awaitF**) and ($\mathbb{T}_{\mathsf{loc}}$-**readV-***n*) are sound.

($\mathbb{T}_{\mathsf{loc}}$-**assignF**) As the premise is valid, we can assume that every trace of `s` that starts in a state where **this**`.f` is set to `e` is a model for

$$\alpha_{\mathsf{loc}}(\mathbf{L} \triangleright \mathsf{copy}(\mathsf{rem}(\mathsf{C}, \mathtt{f}), \mathbf{this}.\mathtt{f}, \mathtt{e}))$$

We have to show that every trace of **this**`.f = e; s` is a model for

$$\alpha_{\mathsf{loc}}(\mathbf{L} \triangleright \mathsf{C})$$

We observe that every trace of **this**`.f = e; s` has the form

$$\left\langle \binom{\sigma}{\rho}, \mathsf{noEv}, \binom{\sigma}{\rho[\mathtt{f} \mapsto \mathtt{e}]} \right\rangle ** \theta$$

where θ is generated by `s` under the conditions of the premise.

Next, note that local types always specify that some none-noEv event is happening next – so the first noEv can safely be chopped off and the model remains a model for $\mathbf{L}$, if the constraint does not change. It remains to show that the tracking constraint manipulation is correct.

By the premise we know that in $\binom{\sigma}{\rho[\mathtt{f} \mapsto \mathtt{e}]}$ everything that is traced by `e` is also traced by `f` and that nothing containing `f` is traced by anything, Now in $\binom{\sigma}{\rho}$, we have no knowledge about `f`. Thus, it is safe to undo the copy operation, because it removes some information that is available to us from the premise. For rem, we know by the premise that no expression containing `f` is used for tracing.

I.e., in $\alpha_{|\mathsf{oc}}(\mathbf{L} \triangleright \mathsf{rem}(\mathsf{C}, \mathtt{f}))$ no expression containing `f` is used to bind any free variable from C used by the Q function. This means that the truth value of the semantics *does not depend* on the evaluation of such expressions in tracking constraints at all. So we can safely drop this expression in $\alpha_{|\mathsf{oc}}(\mathbf{L} \triangleright \mathsf{C})$ as well and the truth value of this semantics does not change at all, because the prefix before θ is not relevant for the semantics and θ is covered by the premise. Thus, every expression may be safely dropped.

($\mathbb{T}_{\mathsf{loc}}$-assignV) This is analogous to **($\mathbb{T}_{\mathsf{loc}}$-assignF)**.

($\mathbb{T}_{\mathsf{loc}}$-return) By validity of the premise we know that in every state $\binom{\sigma}{\rho}$ where $\mathsf{log}(\mathsf{C})$ holds, $\varphi[\mathtt{result} \setminus \mathtt{e}]$ also holds. We must show that every trace of **return** `e` is a model for

$$\alpha_{|\mathsf{oc}}(\mathsf{Put}\ \varphi \triangleright \mathsf{C})$$

We observe that every trace of **return** `e` has the form

$$\left\langle \binom{\sigma}{\rho}, \mathsf{futEv}(\_,\_,\_,\mathtt{e}), \binom{\sigma}{\rho} \right\rangle$$

and the semantics expresses that if C holds (which is equivalent to $\mathsf{log}(\mathsf{C})$), then the next non-noEv event need be a futEv and $\varphi[\mathtt{result} \setminus \mathtt{e}]$ must, where `e` is the return value of the event. This is exactly the condition established by the premise.

($\mathbb{T}_{\mathsf{loc}}$-if) This follows directly from the fact that the semantics of the **if** is the union of each branch, which are covered with the very same type in the two premises.

($\mathbb{T}_{\mathsf{loc}}$-$\oplus$-remove) As a disjunction over a single formula is equivalent to the formula. It remains to check the two parts of the semantics which is the guard (which is ensured by the first premise) and the inner type itself (which is ensured to be followed by the second premise).

($\mathbb{T}_{\mathsf{loc}}$-callV) We ignore $\mathbf{p} \doteq \mathtt{e}$ for soundness, it is only needed for compositions. For the local type itself, we observe that every trace of `v = e!m(e'); s` has the following form (for fitting event parameters):

$$\left\langle \binom{\sigma}{\rho}, \mathsf{invEv}, \binom{\sigma[\mathtt{v} \mapsto f\,]}{\rho} \right\rangle ** \theta$$

By the second premise we know that θ is always a model for $\mathbf{L}$ under the manipulated constraint. By the first premise we know that the call condition is checked for the parameters and by pattern matching we know that the correct method has been called. It remains to show that the constraint has been manipulated correctly, which is analogous to **($\mathbb{T}_{\mathsf{loc}}$-assignF)**.

($\mathbb{T}_{\mathsf{loc}}$-while) This is a standard loop invariant rule, the only detail worth pointing out is that we also have to choose an invariant tracking constraint C', that must be shown to hold at the beginning of the first iteration, which is done explicitly in the first premise.

($\mathbb{T}_{\mathsf{loc}}$-skip) and **($\mathbb{T}_{\mathsf{loc}}$-skipCont)** These rules are trivial.

($\mathbb{T}_{\mathsf{loc}}$-readV-$n$) This is a family of rules, one for each $n \geq 1$. We fix n in the following. The first premise ensures that in the given state we read from the correct tracked value. By validity of the premises we know that for any i, if the guard φ_i holds, then the suffix of the trace of the statement in the conclusion follows $\mathbf{L}_i$ (under the usual correctness of the tracking constraint). The rule is enabling but not sound: to ensure that every trace of the statement in the conclusion is covered by some φ_i, we need the knowledge that the read future is indeed a model for some guard. Each guard is the termination condition of some method. If reading from another method, this is covered by our notion of validity that assumes that all other methods adhere to their type. If reading from the same method, but another process, we apply the same argument as for **($\mathbb{T}_{\mathsf{met}}$-readV)**.

($\mathbb{T}_{loc}$-awaitF) The proof is analogous to ($\mathbb{T}_{inv}$-awaitF), but the reason why φ' holds is that (1) we propagated it to the suspension or termination before (last case in Def. 7.22). (2) all other method adhere to their type and (3) as the obligation scheme stems from a causal-coherent global protocol, this action is indeed issuing the last event before reactivation by Lemmas 7.2 and 7.4. In case it is propagated to an action of the same method, we apply the same argument as for ($\mathbb{T}_{inv}$-awaitF).

Rewrite Rules. The first rewrite rule is sound because the semantics of **skip** is true. The second rewrite rule is sound because for the same reason as ($\mathbb{T}_{loc}$-if). The third rewrite rule is sound because adding disjuncts is sound. □

Theorem 6 (p. 144)

Let **GP** *be a well-formed global protocol and* Prgm *a fitting program with fitting global and local role assignments.*

i If every proof obligation of the generated proof obligation schema can be proven, then every local trace of any method m *is a model for some* $\mathbf{LP} \in \mathscr{L}(\mathtt{m})$.

ii If every local trace of any method m *is a model for some* $\mathbf{LP} \in \mathscr{L}(\mathtt{m})$ *and the initial block sets up* **GP** *correctly, then every global trace of* Prgm *is a model for gcsem*(**GP**).

Proof. We first prove the second part. Suppose this would not hold, i.e., there is a global trace that is not a model. We distinguish five cases:

1. It contains an event that is not specified.
2. It does not contain an event that is specified.
3. It contains the witnesses in the wrong order.
4. A state does not adhere to its specified condition.
5. Wrong role interactions.

This is exhaustive because the reason why a trace stops following the protocol is either an event or a state (we can always pick the first index of such events and states). Either there is an event too much (1), an event too few (2), an ordering error (3) or an error in the content. The error in the content is either the object (5) or the message load. The message load is also specified in the state before the event. Case 4 handles states.

Case 1. We distinguish two further subcases.

The event is not a witness for any action in **GP**. If it contains an event that is not specified and this event is not a witness for any action in **GP**, then it must nonetheless have been issued by some process. This process executes some method for some role. By assumption, each local trace is a model for the local type. This local type results from projection and a simple check shows that each local action corresponds to a global action. Thus, any method only has local traces with witnesses for globally specified events.

The event is a witness for some action in **GP**. I.e., there is some action with two witnesses for one specified event. If this were the case, then the extra event would have been issued by some process – but as **GP** is cooperative, every action is assigned to exactly one tracked future and projection thus generates exactly one local type responsible for the event.

Thus, it can not be the case that an event occurs in the global trace that is not specified.

Case 2. As in the case above we observe that each global action has corresponding local actions generated in the projection. If a global trace does not contain an event that is specified, there is some local type with a local action which is corresponding to the witness of the event in the global type, because **GP** is correctly scoped. By assumption, however, every local trace of every method is a model for its local type, so it can not be the case that an event that is specified does not occur in the global trace.

Case 3. We distinguish two cases.

The witnesses are a permutation of a model. This is covered by the Lemmas 7.2 and 7.4, which can be applied because **GP** is causal-coherent.

The witnesses stem from different branches of **GP**. I.e., the local models realize the correct behavior, but under the wrong condition. However, this cannot be the role that gets the active choice, because this is the first one with a difference between its models and thus the error is in relation to this choice. The wrong choice can also not be realized by any role with a passive choice, because projection checks that the conditions do not overlap for passive choice. Finally, it can also not be the error of a method with an external choice, because we **GP** is well-formed and the preconditions also do not overlap. Thus, every role follows the choice of the choosing role.

Case 4. We distinguish several cases.

Invariants. Some repetition does not observe its invariant. Projection ensures that the global invariant is a conjunction of essentially local invariants, one for each role active within the repetition. Obviously each role with a local type that has a local repetition for the global repetition in question has witnesses for its conjunct. For the other roles, propagation ensures that the repeated part assumes and reestablishes its invariant, and that it is established before the repeated part. It remains to show that the local invariants hold at the same time, i.e., in the same global state. For this it suffices to note that by causal-coherence the witnesses of single iterations do not overlap.

Preconditions and Reactivation Conditions. That preconditions hold is established by the fact that they contain only parameters and that projection ensures that each method is called such that its parameter precondition is implied by the call condition. Reactivation is handled by propagation, the argument is analogous to the global composition of method contracts.

Other Conditions. Any other condition is directly handed down to local types and is tied to some event — as we have established that no event is missing and each local trace is a model for a local type, this follows directly. That conditions can be established if using tracked values follows directly from history-sensitivity.

Case 5. This cannot be the case because we assume fitting local and global role assignments.

It remains to show the first part of the theorem, which follows directly from the semantics of the proof obligation, Lemma 7.5 and the above argument why the precondition can be assumed so that the proof obligation indeed covers all possible traces. □

Acknowledgments

This dissertation would not have been possible without the support, guidance and trust of my advisor Prof. Dr. Reiner Hähnle. He gave me the freedom to explore many areas of computer science in my research, but was always there to help me out when I got lost. I am also very grateful for the conversations we had about topics beyond our work and that he always joined for lunches when nobody else would.

I thank Prof. Dr. Frank de Boer for agreeing to act as the second reviewer for this thesis.

This work was done in the context of the FormbaR project, in cooperation with the Institute of Railway Engineering. FormbaR was financed by DB Netz AG and I thank DB for this opportunity, and Max Schubert, Dr. Michael Leining and Prof. Dr. Andreas Oetting for their support while supervising FormbaR.

I thank Dr. Tzu-Chun Chen for introducing me to behavioral types and her patience with me when we wrote our papers together. I would have never pursued this topic if not for her. Sebastian Schön introduced me into the world of railway operations. I am thankful for his collaboration and the very pleasant time we spent during FormbaR.

This work benefited from feedback on early drafts on some chapters by Dr. Richard Bubel, Dr. Crystal Chang Din, Prof. Dr. Einar Broch Johnsen, Dr. Michael Lienhardt and Dr. Nathan Wasser.

I had the pleasure to do my PhD working together with the ABS and KeY developing groups and the AG Signalling, all of which I thank for their support. I want to thank my office mates and colleagues for making work much more enjoyable — in particular, I thank Gudrun Harris for the cake, Dominic Steinhöfel for the coffee and all of them for the fun and interesting time we had.

Ich danke meiner Frau Nina für ihre schier endlose Unterstützung, Geduld und Liebe in all den Jahren, die ich das Glück hatte mit ihr zu teilen, vor und während meiner Promotion.