Locally Abstract, Globally Concrete Semantics of Concurrent Programming Languages

CRYSTAL CHANG DIN, University of Bergen, Norway
REINER HÄHNLE, Technische Universität Darmstadt, Germany
LUDOVIC HENRIO, Univ Lyon, EnsL, UCBL, CNRS, Inria, France
EINAR BROCH JOHNSEN, University of Oslo, Norway
VIOLET KA I PUN, Western Norway University of Applied Sciences, Norway
S. LIZETH TAPIA TARIFA, University of Oslo, Norway

Formal, mathematically rigorous programming language semantics are the essential prerequisite for the design of logics and calculi that permit automated reasoning about concurrent programs. We propose a novel modular semantics designed to align smoothly with program logics used in deductive verification and formal specification of concurrent programs. Our semantics separates local evaluation of expressions and statements performed in an abstract, symbolic environment from their composition into global computations, at which point they are concretised. This makes incremental addition of new language concepts possible, without the need to revise the framework. The basis is a generalisation of the notion of a program trace as a sequence of evolving states that we enrich with event descriptors and trailing continuation markers. This allows to postpone scheduling constraints from the level of local evaluation to the global composition stage, where well-formedness predicates over the event structure declaratively characterise a wide range of concurrency models. We also illustrate how a sound program logic and calculus can be defined for this semantics.

CCS Concepts: • **Theory of computation** → **Parallel computing models**; *Programming logic*; **Program semantics**; • **Software and its engineering** → **Concurrent programming structures**;

ACM Reference Format:
Crystal Chang Din, Reiner Hähnle, Ludovic Henrio, Einar Broch Johnsen, Violet Ka I Pun, and S. Lizeth Tapia Tarifa. 2024. Locally Abstract, Globally Concrete Semantics of Concurrent Programming Languages. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2024), 58 pages. https://doi.org/10.1145/3648439

1 INTRODUCTION

Denotational semantics describes the result of program execution by deriving a semantic entity, generally a mathematical structure, from the program. An example of such a mathematical structure is a state transformer from an initial to a final program state [10, 51]. The entities of a denotational semantics, such as state transformers, do not describe how a program executes. For parallel shared-variable programs, the result of execution may depend on the scheduling of the program's different atomic segments, leading to non-deterministic behavior. This makes denotational semantics in terms of state-transformers cumbersome, because the parallel composition of compound statements

Authors' addresses: Crystal Chang Din, Department of Informatics, University of Bergen, Norway; Reiner Hähnle, Technische Universität Darmstadt, Dept. of Computer Science, Germany; Ludovic Henrio, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, France; Einar Broch Johnsen, University of Oslo, Norway; Violet Ka I Pun, Western Norway University of Applied Sciences, Norway; S. Lizeth Tapia Tarifa, University of Oslo, Norway.

0164-0925/2024/1-ART1
https://doi.org/10.1145/3648439

cannot be determined from the state-transformers of the statements alone but give rise to additional requirements to avoid interference [11, 68, 76].

Trace semantics provides additional structure to the semantics of statements such that parallel statements can be described compositionally from the semantics of their components. To capture the result of program execution, trace semantics can be given in terms of sequences of states [16, 18, 46, 73] or sequences of communication events [25, 31, 52, 57]. Hybrid traces, that include both states and communication events, have previously been developed by Brookes [18] to define a denotational semantics of concurrent separation logic, using events to manage low-level resources.

In this paper, we show how denotational semantics that combine such hybrid traces with continuation markers can be used as a general framework to capture the execution of imperative parallel programs, including a range of concurrency constructs such as the dynamic spawning of threads, procedure calls, asynchronous and synchronous communication and cooperative concurrency. We develop a general denotational format to describe how parallel programs synchronise and interleave in terms of a hybrid notion of trace that combines states and communication events.

The proposed trace semantics scales flexibly to a range of concurrent, imperative programming paradigms, as found, for example, in C, Java, ProMeLa [53], Actors [49], or Active Objects [27]. Specifically, given a program P, running on a collection of communicating processors or cores, we want to obtain the set of all global system traces that P can produce starting from some initial state. Our overall goal is to provide a semantics that aligns well with contract-based deductive verification [42]. To this end, it is essential that the semantics cleanly separates local computations (one statement on one core) from global ones; i.e., there must be a suitable notion of *composition* that can generate the global traces from the local ones. It is known [17, 71] that compositionality in denotational semantics shifts the flavour of the semantics in the direction of operational semantics [78]. In our work, we isolate the operational aspects of composition into separate, global rules that operate over denotational, local rules.

The envisaged semantics should also be *modular* in the following sense: it must be possible to support a new language construct without the need to revise the whole framework. Ideally, there is a single evaluation rule for each construct of the target language that can be applied *independently* from all other rules: the recursive call to evaluate subsequent statements is not inside the semantic evaluation of each statement. This is not only a good match with deductive verification rules of program logics [42], but also with formal specification languages for concurrent programs that have a trace semantics [13, 32, 61, 81, 83].

A trace semantics for a given target language, satisfying the requirements sketched above, can be defined in three phases:

(1) Declare *local* evaluation rules for each syntax construct of the target language.
(2) Declare *composition* rules that combine local evaluation and process creation into global traces. Scheduling is expressed declaratively as well-formedness constraints over traces.
(3) Define the *generation* of all global traces with the help of the composition rules from a suitable initial configuration.

To achieve the desired degree of modularity, we generalise the standard notion of a program trace [54], i.e., a sequence of evolving system configurations, starting in some initial state. We make two generalisations. The first is that local states σ can be abstract. This means the value $\sigma(x)$ of a memory location x is permitted to be unspecified. One can think of an abstract value $\sigma(x)$ as a Skolem constant x_0 whose interpretation is determined by an external context. Alternatively, think of *symbolic execution* [21, 64], where the value of a memory location x may be a *symbolic* term x_0 representing an unknown value. States containing abstract values are called *symbolic*, otherwise they are called *concrete.*

Symbolic states allow us to evaluate local code independently of its call context. For example, when evaluating the semantics of a statement *s* that receives a message from another process, the value of that message cannot possibly be known independently of the call context. In this case, the local evaluation of *s* can be expressed in a trace with symbolic states. These symbolic states are *concretised* when the global context of the local computations (here, the sender) is resolved, i.e. in the composition rules during phase (2) above. Hence, the resulting global traces are concrete. Thus, the name of our semantic framework: *locally abstract, globally concrete* (LAGC).

The second generalisation of traces concerns *scheduling*: concurrency models differ in the exact locations at which a local computation can be interrupted (aka preempted) and how exactly the computation is continued afterwards, i.e. which scheduling decision is taken next. To achieve maximum modularity, we do not build scheduling into a fixed set of rules. Instead, we use *continuation markers* and *events* to specify when scheduling and preemption is possible.

As a result, local traces are not merely abstract, but as well contain *events* used in the composition rules as interaction points. With the event mechanism, various concurrency models can be defined easily in two parts: first, ensure that local evaluation rules generate suitable synchronisation events; second, define a *well-formedness* predicate on concrete traces restricting the global traces that can be generated to those reflecting the targeted concurrency model. Now it is sufficient to add the well-formedness predicate as a premise to each composition rule.

A standard approach to programming language semantics is SOS [58, 67], where each semantic rule either decomposes one complex or one atomic statement at a time. This results in a very local, also termed small-step, operational semantics. In consequence, SOS semantics must represent scheduling and the effects of interaction at the same time with other aspects of the semantics in a non-modular manner. Mosses [70] proposed modular semantics as a way to increase modularity of SOS by classifying interaction as read/write/read-write and providing a schematic approach to compose transition labels. LAGC places itself in a non-SOS setting. It postpones task interaction to composition rules that can express any form of interaction and so encapsulate complex communication or scheduling patterns. This is illustrated below with constraints on communication timing (in Section 6) and cooperative scheduling (in Section 7.3).

Local rules for sequential parts of a language, which combine with global composition rules synchronizing interactions between processes, are well-known from operational semantics of concurrent systems [47, 50, 58]. For example, synchronous communication between processes can be characterized by matching events on labelled transitions. In these operational semantics, asynchronous communication is treated differently from synchronous communication, for example, by spawning additional processes to encode the asynchronous nature of the communication [79]. In contrast, LAGC trace semantics permits a uniform treatment of synchronous *and* asynchronous communication purely in terms of well-formedness conditions on traces. In LAGC, this is achieved by defining well-formedness in terms of orderings of events in the traces: for example, sending and receiving a message is represented by separate events, such that sending happens before reception in well-formed traces. In fact, traces with events as used in LAGC turn out to be a flexible and intuitive mechanism, which can be used to express a range of dependencies between processes beyond synchronous communication (for example, object generation).

Each ingredient of the LAGC semantics—traces, events, abstraction, the separation of local and global computations—is not new on its own. What sets the LAGC framework apart from any other denotational semantics we are aware of, is its unique combination of these features: abstract traces with events are used to supply a strictly local semantics to sequential code snippets. These are then combined and instantiated to the sets of traces that can occur in a given concurrency model. Scheduling is completely separated from the generation of state updates and events. Different scheduling strategies are defined declaratively with well-formedness predicates. As a consequence,

$$\begin{aligned}
x \in \mathit{Var} &::= \mathit{identifier}\\
v \in \mathit{Val} &::= \mathtt{tt} \mid \mathtt{ff} \mid 0 \mid 1 \mid \ldots\\
op \in \mathit{Op} &::= \mathtt{==} \mid < \mid > \mid \leq \mid \geq \mid + \mid - \mid \star \mid / \mid \ldots\\
e \in \mathit{Exp} &::= x \mid v \mid e\ op\ e\\
se \in \mathit{Sexp} &::= e \mid *
\end{aligned}$$

Fig. 1. Syntax of expressions.

the LAGC semantic framework captures different forms of concurrency for imperative languages in a modular and uniform manner. This includes communication patterns that are hard to characterize in traditional small-step operational semantics [78] based on transitions between states, such as causally ordered communication chains.

A LAGC-style trace semantics was pioneered for the active object language ABS [32]. Here we show that, due to its modularity, LAGC semantics constitutes a general *semantic framework* for a wide range of concurrent programming models: it is easy to add new syntactic constructs and to accommodate different concurrency paradigms. We will demonstrate this as follows: starting with a simple WHILE-language, we define LAGC semantics for an increasingly complex series of languages by successively adding new features, sequential as well as parallel ones. In the end we cover a representative set of language features and concurrency models and instantiate our semantic framework to two quite different concurrent programming languages. Additional examples of applications of the LAGC semantics that make specific use of its design goals are found in Section 8.

The paper is organised as follows: In Section 2 we set up the formal framework that the LAGC semantics is based upon. As explained above, we need symbolic states and traces that may contain abstract values for memory locations. We also need a concretisation operator that instantiates a symbolic trace to match a concrete context. In addition, we equip traces with states and continuation markers. Section 3 introduces the LAGC framework along phases (1)–(3) for a WHILE-language. We substantiate the claim made above, that a LAGC semantics is a good match for deductive verification calculi, by defining in Section 4 a program logic for WHILE with a concise soundness proof. In Section 5, we gradually extend the semantics to local parallelism (with atomicity), local memory, and procedure calls. All definitions and theorems from Sections 2, 3, and 5 have been mechanized and proven [45] in Isabelle/HOL [74]. In Section 6, we take the step to multiple processors that send and receive messages among each other. We show that a wide range of communication patterns can be intuitively and declaratively characterised via well-formedness, including synchronous and asynchronous communication, bounded and unbounded FIFO, as well as causality. Section 7 instantiates the LAGC framework to the rather different concurrency models found in the languages ProMeLa and ABS, respectively. For the latter, we need to add objects and futures, as well as to change the interleaving semantics, which turns out to be easily possible. Related work is discussed in Section 9. Section 10 considers future directions for work and concludes.

2 BASICS

2.1 States

We assume given standard basic types, including integers, Booleans and process identifiers, with standard value domains and associated operators. In addition, we allow starred expressions, representing unknown, *symbolic* values.

Definition 2.1 (Variables, Values, (Starred) Expressions). Let *Var* be a set of program variables, *Val* a set of values, and *Op* a set of operators, with typical elements x, v, and op, respectively. The sets *Val* and *Op* include the values and operators of the basic types (see Figure 1). The set *Exp* contains expressions, with typical element e, obtained from variables, values, and by applying operators to expressions. The set *Sexp* contains starred expressions, with typical element *se*, obtained by extending expressions with an additional symbol $*$.

We assume that all expressions are well-typed; i.e., in expressions, operators are only applied to subexpressions which can be reduced to values of appropriate types. Let $\underline{op}$ denote an evaluation function for the operators *op* defined over the values of the basic types, such that $v\ \underline{op}\ v'$ is a value in the basic types. A Boolean expression is an expression that evaluates to a Boolean value when its arguments are values of the basic types, and similarly for expressions of other basic types. Overline notation is used for lists of different syntactic categories, e.g., $\overline{v}$ and $\overline{e}$ represent lists of values and expressions, respectively. Let $\mathrm{vars}(e)$ denote the set of variables in an expression e, which has a straightforward inductive definition.

We now define computation states. Usually, states are mappings from variables to concrete values. To permit *symbolic* expressions (i.e., expressions containing variables) occurring as *values* in states, the starred expression $*$ is used to represent a value for symbolic variables that cannot be further evaluated. The $*$ symbol does not occur in programs, it is part of the semantic domain.

Definition 2.2 (Symbolic State, State Update). A *symbolic state* σ is a partial mapping

$$\sigma : \mathit{Var} \rightarrow \mathit{Sexp}$$

from variables to starred expressions. The notation $\sigma[x \mapsto se]$ expresses the *update* of state σ at x with expression *se*:

$$\sigma[x \mapsto se](y) = \begin{cases} se & x = y \\ \sigma(y) & x \neq y\ . \end{cases}$$

In a symbolic state, a *symbolic variable* is defined as a variable bound to an unknown value, represented by the starred expression $*$. Symbolic variables play a different role than ordinary variables: they act as parameters, relative to which a local computation is evaluated. To distinguish them syntactically, we adopt the convention of using capital letters for symbolic variables as much as possible. Note that the set of symbolic and non-symbolic variables are not two distinct categories and a variable that is symbolic can become non-symbolic after substitution (see Example 2.7 below).

Definition 2.3 (Symbolic Variable). *Symbolic variables* in a state σ are variables mapped to $*$:

$$\mathrm{symb}(\sigma) = \{X \in \mathit{Var} \mid \sigma(X) = *\}\ .$$

We will also need the concept of the extension of a state:

Definition 2.4 (State Extension). We say that a state σ' *extends* a state σ if (i) $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$ and (ii) $\sigma(x) = \sigma'(x)$ for all $x \in \mathrm{dom}(\sigma)$. We overload the subset symbol and write $\sigma \subseteq \sigma'$.

Example 2.5. Consider a state $\sigma_0 = [x_0 \mapsto Y_0 + w_0,\ Y_0 \mapsto *,\ w_0 \mapsto 42,\ x_1 \mapsto Y_1]$. Observe that (1) the expressions in the range of σ_0 can be simplified and (2) there are dangling references, such as Y_1, not in the domain of σ_0.

The example shows that symbolic states are slightly too general for our purpose, motivating the following definition, which constrains the variables that may occur in value expressions of states to symbolic variables:

Definition 2.6 (Well-Formed State). A state σ is *well-formed* if it fulfils the following condition:

$$\{x \in \mathrm{vars}(\sigma(y)) \mid y \in \mathrm{dom}(\sigma)\} \subseteq \mathrm{symb}(\sigma)\ .$$

A well-formed state σ is *concrete* if $\mathrm{symb}(\sigma) = \{\}$. For a concrete, well-formed state σ and all $x \in \mathrm{dom}(\sigma)$, there is a value v such that $\sigma(x) = v$.

Example 2.7. The state σ_0 in Example 2.5 can be turned into a well-formed state σ_1 by simplifying the expression $Y_0 + w_0$, and binding Y_1 to a star, obtaining $\sigma_1 = [x_0 \mapsto Y_0 + 42,\ Y_0 \mapsto *,\ w_0 \mapsto 42,\ x_1 \mapsto Y_1,\ Y_1 \mapsto *]$. We have that $\sigma_1 = \sigma_0[x_0 \mapsto Y_0 + 42,\ Y_1 \mapsto *]$ and $\mathrm{symb}(\sigma_1) = \{Y_0, Y_1\}$. Let $\sigma_2 = \sigma_1[Y_0 \mapsto 3,\ Y_1 \mapsto 2]$. After simplification, σ_2 is a concrete, well-formed state that can be written as $\sigma_2 = [x_0 \mapsto 45,\ Y_0 \mapsto 3,\ w_0 \mapsto 42,\ x_1 \mapsto 2,\ Y_1 \mapsto 2]$.

Henceforth, all states are assumed to be well-formed. *We always assume states to be simplified by the propagation of concrete values*, similar to σ_2 in Example 2.7, otherwise non-symbolic variables might occur as values. Symbolic states, i.e. states with symbolic variables, are close to states in symbolic execution as used in path exploration [39], but simplified and with an additional restriction (Definition 2.16 below).

2.2 Evaluation

The evaluation of expressions in the context of a symbolic state reduces known variables to their values and keeps the symbolic variables inside the expression, reducing an expression as much as currently possible. The evaluation function is defined as follows:

Definition 2.8 (Evaluation Function). Let σ be a symbolic state. The evaluation function $\mathrm{val}_\sigma : Exp \longrightarrow Exp$ for expressions in the context of σ is defined inductively:

$$\begin{aligned}
\mathrm{val}_\sigma(x) &= \begin{cases} x & \text{if } \sigma(x) = * \\ \sigma(x) & \text{otherwise} \end{cases} \\
\mathrm{val}_\sigma(v) &= v \\
\mathrm{val}_\sigma(e_1\ op\ e_2) &= \begin{cases} \mathrm{val}_\sigma(e_1)\ \underline{op}\ \mathrm{val}_\sigma(e_2) & \text{if } \mathrm{val}_\sigma(e_1) \in Val \text{ and } \mathrm{val}_\sigma(e_2) \in Val \\ \mathrm{val}_\sigma(e_1)\ \overline{op}\ \mathrm{val}_\sigma(e_2) & \text{otherwise} \end{cases}
\end{aligned}$$

In the following we will ensure that $\mathrm{vars}(e) \subseteq \mathrm{dom}(\sigma)$ holds, such that val_σ is a total function.

Example 2.9. Using state σ_1 of Example 2.7, we evaluate $\mathrm{val}_{\sigma_1}(x_0 + Y_0 + Y_1) = (Y_0 + 42) + Y_0 + Y_1$.

Let $\mathrm{val}_\sigma(\overline{e})$ denote the element-wise application of the evaluation function to all expressions in the list $\overline{e}$, and likewise for sets.

2.3 Traces and Events

Traces are sequences over states and structured events. The presence of events makes it easy to ensure global properties of traces via well-formedness conditions over events. Since states may be symbolic, traces will be symbolic as well and it is necessary to constrain traces by symbolic path conditions. We start with a general definition of events. Their specific structure will be added later.

Definition 2.10 (Event Marker). Let $ev(\overline{e})$ be an *event marker* over expressions $\overline{e}$.

Definition 2.11 (Path Condition). A *path condition pc* is a finite set of Boolean expressions. If pc contains no variables, then we can assume it to be fully evaluated; i.e. it is either $\emptyset$, $\{\mathrm{ff}, \mathrm{tt}\}$, $\{\mathrm{ff}\}$, or $\{\mathrm{tt}\}$. A fully evaluated path condition is *consistent* if and only if it does not contain ff.

For any concrete state σ, $\mathrm{val}_\sigma(pc)$ is a path condition with no variables that can be fully evaluated.

Definition 2.12 (Conditioned Symbolic Trace). A *symbolic trace* τ is defined inductively by the following rules (ε denotes the empty trace):

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

A *conditioned symbolic trace* has the form $pc \triangleright \tau$, where pc is a path condition and τ is a symbolic trace. If pc is consistent, we simply write τ for $pc \triangleright \tau$.

By definition, traces are finite. For simplicity, let $\langle\sigma\rangle$ denote the *singleton trace* $\varepsilon \curvearrowright \sigma$. *Concatenation* of two traces τ_1, τ_2 is written as $\tau_1 \cdot \tau_2$. The final state of a non-empty trace τ is selected with $\mathrm{last}(\tau)$, the first state of a non-empty trace τ with $\mathrm{first}(\tau)$.

Example 2.13. Continuing Example 2.7, we define a conditioned symbolic trace $pc_0 \triangleright \tau_0 = \{Y_0 > Y_1\} \triangleright \langle\sigma_1\rangle \curvearrowright \sigma_1[x_0 \mapsto 17]$.

Sequential Composition. A central feature of traces is that they semantically model the sequential composition of program statements. Assume that τ_1 is a trace of a statement r and τ_2 a trace of another statement s. To obtain the trace corresponding to the sequential composition of r and s, the traces corresponding to r and s should be composed (joined) such that the last state of the first trace and the first state of the second trace are generally identical. We generalize such a composition in case the states are not exactly the same: to compose two traces τ_1 and τ_2 the first state of τ_2 should be an extension of the last state of τ_1, but the resulting trace should only contain the larger of the two states. This motivates the *semantic chop* $**$ on traces (following [73], who were inspired by interval temporal logic [43] and process logic [77]):

Definition 2.14 (Chop on Traces). Let pc_1, pc_2 be path conditions and τ_1, τ_2 be symbolic traces, and assume that τ_1 is a non-empty, finite trace. The semantic chop $(pc_1 \triangleright \tau_1) ** (pc_2 \triangleright \tau_2)$ is defined as follows:

$$(pc_1 \triangleright \tau_1) ** (pc_2 \triangleright \tau_2) = (pc_1 \cup pc_2) \triangleright \tau \cdot \tau_2 \text{ where } \tau_1 = \tau \curvearrowright \sigma,\ \tau_2 = \langle\sigma'\rangle \cdot \tau' \text{ and } \sigma \subseteq \sigma' \,.$$

We say that two non-empty conditioned traces $pc_1 \triangleright \tau_1$ and $pc_2 \triangleright \tau_2$ are *joinable* when $\mathrm{last}(\tau_1) \subseteq \mathrm{first}(\tau_2)$. Whenever the first trace is empty or the final state of the first trace cannot be extended to the first state of the second trace, the operator is undefined. The definition extends to traces without path conditions in the obvious way: $\tau_1 ** \tau_2$ is $(\emptyset \triangleright \tau_1) ** (\emptyset \triangleright \tau_2)$.

Traces with Events. Events will be uniquely associated with the state in a trace at which they occurred. The events do not update the values in a state, but they may extend a state with fresh symbolic variables. To this aim, an event $ev(\overline{e})$ is inserted into a trace after a state σ and the state is then augmented by a set $\overline{V}$ of symbolic variables. The notation we use for this operation is an *event trace* $ev_\sigma^{\overline{V}}(\overline{e})$ of length three:

$$ev_\sigma^{\overline{V}}(\overline{e}) = \langle\sigma\rangle \curvearrowright ev(\mathrm{val}_{\sigma'}(\overline{e})) \curvearrowright \sigma' \text{ where } \sigma' = \sigma[\overline{V} \mapsto *].$$

Given a trace τ of the form $\tau_1 \curvearrowright \sigma$ and event $ev(\overline{e})$, appending the event is achieved by the trace $\tau_1 \cdot ev_\sigma^{\overline{V}}(\overline{e})$. The preceding definition ensures that events in traces are joinable; $\tau ** ev_\sigma^{\overline{V}}(e)$ is well-defined whenever $\mathrm{last}(\tau) = \sigma$. If $\overline{V}$ is empty then the state is unchanged, in this case we omit the set of symbolic variables:

$$ev_\sigma(\overline{e}) = ev_\sigma^{\emptyset}(\overline{e})$$

Example 2.15. Event traces can be inserted at the middle of a trace. To insert an event $ev(Y_0)$ that does not introduce symbolic variables at σ_1 to trace τ_0 in Example 2.13, we use the event trace $ev_{\sigma_1}(Y_0) = \langle\sigma_1\rangle \curvearrowright ev(\mathrm{val}_{\sigma_1}(Y_0)) \curvearrowright \sigma_1 = \langle\sigma_1\rangle \curvearrowright ev(Y_0) \curvearrowright \sigma_1$. This results in the trace: $\tau_2 = \{Y_0 > Y_1\} \triangleright \langle\sigma_1\rangle \curvearrowright ev(Y_0) \curvearrowright \sigma_1 \curvearrowright \sigma_1[x_0 \mapsto 17]$. To insert an event $ev(Y_2)$ that introduces the symbolic variable Y_2, we use the event trace $ev_{\sigma_1}^{\{Y_2\}}(Y_2) = \langle\sigma_1\rangle \curvearrowright ev(Y_2) \curvearrowright \sigma_1[Y_2 \mapsto *]$. The trace in Example 2.13 results in the trace:

$$\tau_2' = \{Y_0 > Y_1\} \triangleright \langle\sigma_1\rangle \curvearrowright ev(Y_2) \curvearrowright \sigma_1[Y_2 \mapsto *] \curvearrowright \sigma_1[Y_2 \mapsto *, x_0 \mapsto 17].$$

Well-Formed Traces. Similar to the values of well-formed states, the expressions in events and path conditions of a well-formed trace should only contain symbolic variables. This requires all states in a trace to agree upon which variables are symbolic. We also impose an additional well-formedness condition on events: any event occurring in a trace which is preceded by state σ, must be followed by an extension of σ which at most includes the new symbolic variables introduced by the event; i.e., it can be obtained by inserting an event trace at σ. This implies that a trace always starts and ends with a state, never with an event. Well-Formed traces are formalised as follows:

Definition 2.16 (Well-Formed Trace). Let $pc \triangleright \tau$ be a conditioned symbolic trace and let $V = \bigcup_{\sigma\in\tau} \mathrm{symb}(\sigma)$. The trace $pc \triangleright \tau$ is well-formed if the following conditions hold:

$$\forall \sigma \in \tau.\, \sigma \text{ is well-formed} \tag{1}$$

$$\forall \sigma \in \tau.\, (\mathrm{dom}(\sigma) \setminus \mathrm{symb}(\sigma)) \cap V = \emptyset \tag{2}$$

$$\mathrm{vars}(pc) \subseteq V \tag{3}$$

$$\forall ev(\overline{e}) \in \tau.\ \mathrm{vars}(\overline{e}) \subseteq V \tag{4}$$

$$\forall ev(\overline{e}), \tau_1, \tau_2.\, \big(\tau = \tau_1 \curvearrowright ev(\overline{e}) \cdot \tau_2 \implies \exists \sigma, \sigma'.\, \mathrm{last}(\tau_1) = \sigma \land \mathrm{first}(\tau_2) = \sigma' \land \sigma \subseteq \sigma'\big) \tag{5}$$

Equation (2) states that a variable that is symbolic in a state cannot be non-symbolic in another state of the trace. Equations (3)–(4) ensure that any variable to occur in a path condition or event is a symbolic variable from some state. Equation (5) guarantees that any event in a trace preceded by σ is followed by an extension σ'.

Example 2.17. The traces τ_2 and τ_2' in Example 2.15 have well-formed states. Only symbolic variables occur in path conditions and events, and the events were added by inserting event traces. Hence, they are well-formed.

Definition 2.18 (Concrete Traces). A *concrete trace* is a well-formed trace containing only concrete states and events, as well as a fully evaluated, consistent path condition, which is reduced to $\emptyset$ or $\{\mathrm{tt}\}$ and, therefore, removed from the trace. We use *sh* for concrete traces, where the letters stand for "shining" trace.

Example 2.19. Let $\sigma_3 = [x_0 \mapsto 45,\ Y_0 \mapsto 3,\ w_0 \mapsto 42,\ x_1 \mapsto 2,\ Y_1 \mapsto 2]$, then a concrete trace of length two is $sh_0 = \langle\sigma_3\rangle \curvearrowright \sigma_3[x_0 \mapsto 17]$. The path condition $Y_0 > Y_1$ is evaluated to $3 > 2 = \mathrm{tt}$ in σ_3 and thus consistent.

Observe that sh_0 can be obtained from τ_0 in Example 2.13 by a suitable instantiation of its symbolic variables. The precise definition of this operation is the purpose of the following subsection.

2.4 Concretisation

A concretisation mapping is defined relative to a state. It associates a concrete value to each symbolic variable of the state.

Definition 2.20 (State Concretisation Mapping). A mapping $\rho : Var \to Val$ is a *concretisation mapping* for a state σ if $\mathrm{dom}(\rho) \cap \mathrm{dom}(\sigma) = \mathrm{symb}(\sigma)$.

A concretisation mapping ρ may additionally define the value of variables that are not in the domain of σ.

Example 2.21. Consider σ_1 of Example 2.7 where $\mathrm{symb}(\sigma_1) = \{Y_0,\ Y_1\}$. We define the concretisation mapping $\rho_1 = [Y_0 \mapsto 3,\ Y_1 \mapsto 2]$ for σ_1.

Let $\sigma_4 = [X \mapsto *,\ z \mapsto 3]$. Then a concretisation mapping must give a value to X, but *not* to z, for example, $\rho_2 = [X \mapsto 2,\ Y \mapsto 0]$ is a concretisation mapping for σ_4.

A symbolic state can be concretised using the values assigned to its symbolic variables in a concretisation mapping to evaluate the symbolic expressions.

Definition 2.22 (State Concretisation). Let σ be a state and ρ a concretisation mapping for σ. The concretisation of σ with ρ is defined as follows:

$$\rho \bullet \sigma = \rho \cup \{x \mapsto \mathrm{val}_\rho(\sigma(x)) \mid x \in \mathrm{dom}(\sigma) \setminus \mathrm{dom}(\rho)\}\ .$$

Example 2.23. Continuing Examples 2.19 and 2.21, we obtain $\rho_1 \bullet \sigma_1 = \sigma_3$.

To concretise a symbolic trace $pc \triangleright \tau$, we must apply a concretisation mapping ρ to all states and events of the trace. This means the domain of ρ must comprise all symbolic variables that occur in the trace. This is the case for well-formed traces, whose states must agree on which variables are symbolic (Equation (2) in Definition 2.16).

Definition 2.24 (Trace Concretisation Mapping). A mapping ρ is a *trace concretisation mapping* for τ if it is a concretisation mapping for all the states in τ. We say that ρ *concretises* τ.

Definition 2.25 (Symbolic Trace Concretisation). Let $pc \triangleright \tau$ be a well-formed trace and ρ a concretisation mapping for τ. The *concretisation* $\rho(pc \triangleright \tau)$ of $pc \triangleright \tau$ is obtained as $\rho(pc) \triangleright \rho(\tau)$, where $\rho(pc) = \mathrm{val}_\rho(pc)$ and $\rho(\tau)$ is defined as follows:

$$\rho(t_1 \cdots t_n \cdots) = \rho(t_1) \cdots \rho(t_n) \cdots$$
$$\rho(\sigma) = \rho \bullet \sigma$$
$$\rho(ev(\overline{e})) = ev(\mathrm{val}_\rho(\overline{e}))$$

Proposition 2.26. *The concretisation of a state is a concrete state. The concretisation of a well-formed trace is a concrete, well-formed trace. Any concrete state or trace is a concretisation of itself.*

Example 2.27. We continue to use the symbolic trace $pc_0 \triangleright \tau_0$ of Example 2.13 and concretise it with the mapping ρ_1 of Example 2.21. The resulting concrete trace is sh_0 from Example 2.19, where the path condition is reduced to $\{\mathtt{tt}\}$ and removed from the trace.

2.5 Continuations

To capture the local semantics of a language, below we define an evaluation function $\mathrm{val}_\sigma(s)$ that evaluates a single statement s in a—possibly symbolic—state σ to a set of conditioned, symbolic traces. *Compositional* local evaluation rules can then be defined for each statement by adding a *continuation marker* at the end. Continuation markers are needed, if s is a composite statement that requires separate evaluation of its constituent parts. In particular, s might not terminate, but this should not jeopardise parallel computations when computing a sequential trace for a global system. We avoid this issue by stopping the evaluation after a finite number of steps by means of a continuation marker, defined as follows:

Definition 2.28 (Continuation Marker). Let s be a program statement. The *continuation marker* $\mathrm{K}(s)$ expresses that a given trace is extended by the traces resulting from computing s. The empty continuation, denoted K(▯), expresses that nothing remains to be computed.[1] The statement ▯ occurs only in traces and not source programs.

The argument to the continuation marker is the code on which evaluation will continue. The evaluation of this code happens at a later stage, during the computation of the overall trace. But at this later stage, a proper trace extension must only produced if the continuation is not ▯. To this aim, ▯ must be a continuation (the only one) that is evaluated to the empty trace. In contrast,

[1]The mnemonics of the symbol is that from an empty bottle nothing can be consumed.

$$s \in \mathit{Stmt} ::= \texttt{skip} \mid x := e \mid \texttt{if}\ e\ \{\ s\ \} \mid s;s \mid \texttt{while}\ e\ \{\ s\ \}$$

Fig. 2. The syntax for statements in WHILE.

the **`skip`** statement is *not* evaluated to an empty trace, but to a singleton trace consisting of just the current state. When atomic statements are evaluated, there is no continuation code but only a return of control. In this case, we end the trace with an empty continuation (see Section 3.1). Local evaluation (corresponding to phase (1) in Section 1) is defined below such that for each statement s and symbolic state σ, the result of $\mathrm{val}_\sigma(s)$ is a set of conditioned, symbolic traces together with their continuation markers. Slightly abusing terminology, a conditioned symbolic trace with its continuation marker is called a *continuation trace* and written as $pc \triangleright \tau \cdot \mathrm{K}(s')$, where τ is a *finite* trace. The intuition is that a continuation trace extends into a trace; we denote by **CTr** the type of continuation traces.

3 LAGC SEMANTICS OF WHILE

We define a LAGC semantics for WHILE, a language with basic sequential constructs that can be seen as a kernel of imperative programming languages [10]. The statements of WHILE consist of **`skip`**, assignment, conditional, sequential composition, and **`while`**-loops. The syntax for statements is given in Figure 2, where we assume given standard expressions e. Assignment binds the value of an expression to a variable. Conditionals and **`while`**-loops depend on the value of a Boolean expression e. We assume the standard semantics of this language to be known, and use it to illustrate our trace semantics. *Local rules* (phase (1)) unfold the traces until the next possible scheduling point, marked by a continuation. *Composition rules* (phase (2)) select the next trace to be unfolded at the scheduling point; for the sequential language, there is only one trace so the latter selection is deterministic. Due to its simplicity, WHILE merely needs state-based traces, the use of events is covered in Section 5.4.

3.1 Local Evaluation

Local evaluation rules a single statement in the context of a symbolic state, and returns a set of finite continuation traces. Each evaluation rule represents the execution of a single statement, producing a set of continuation traces. The evaluation rules are reminiscent of small-step reduction rules, but work in a denotational setting and with a symbolic context. We overload the symbol val_σ and declare it with the type $\mathrm{val}_\sigma : \mathit{Stmt} \to 2^{\mathbf{CTr}}$.

The rule for **`skip`** generates an empty path condition, returns the state it was called in, and continues with the empty continuation. The result is a set containing one singleton trace:

$$\mathrm{val}_\sigma(\texttt{skip}) = \{\emptyset \triangleright \langle\sigma\rangle \cdot \mathrm{K}(\llbracket\rrbracket)\}\ . \tag{6}$$

The assignment rule generates an empty path condition and a trace from the current state σ to a state which updates σ at x, and continues with an empty continuation. The result is one trace of length two:

$$\mathrm{val}_\sigma(x := e) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto \mathrm{val}_\sigma(e)] \cdot \mathrm{K}(\llbracket\rrbracket)\}\ . \tag{7}$$

The rule for the conditional statement branches on the value of the condition, resulting in two traces with different path conditions. The first trace is obtained from the current state and the continuation with the statements in the **`if`**-branch, and the second trace has the empty continuation (corresponding to the empty else-branch):

$$\mathrm{val}_\sigma(\texttt{if}\ e\ \{\ s\ \}) = \{\{\,\mathrm{val}_\sigma(e) = \mathtt{tt}\} \triangleright \langle\sigma\rangle \cdot \mathrm{K}(s),\ \ \{\mathrm{val}_\sigma(e) = \mathtt{ff}\} \triangleright \langle\sigma\rangle \cdot \mathrm{K}(\llbracket\rrbracket)\,\}\ . \tag{8}$$

The semantics of **`while`** is similar to the semantics for the conditional statement. This gives us the opportunity to illustrate that the semantics of a statement can easily be expressed in terms of the semantics of another statement, without having to expose intermediate states:

$$\mathrm{val}_\sigma(\texttt{while } e \ \{ \ s \ \}) = \mathrm{val}_\sigma(\texttt{if } e \ \{ \ s; \ \texttt{while } e \ \{ \ s \ \}\}) \ . \tag{9}$$

Thus, evaluating $\mathrm{val}_\sigma(\texttt{while } e \ \{ \ s \ \})$ results in two traces. If the path condition e evaluates to true, then the continuation is $\mathrm{K}(s; \texttt{while } e \ \{ \ s \ \})$, otherwise it is the empty continuation.

The rule for sequential composition $r;s$ is obtained by first evaluating r to traces of the form $pc \triangleright \tau \cdot \mathrm{K}(r')$ with continuation r'. The statement s can simply be added to this continuation:

$$\mathrm{val}_\sigma(r;s) = \{pc \triangleright \tau \cdot \mathrm{K}(r';s) \mid pc \triangleright \tau \cdot \mathrm{K}(r') \in \mathrm{val}_\sigma(r)\} \ . \tag{10}$$

A subtle point concerns the propagation of empty continuations: r' might be the empty continuation $\lozenge$, which should be ignored. This behavior is captured in the following rewrite rule, which is eagerly applied to statements occurring inside continuations:

$$s;s' \rightsquigarrow s' \text{ if } s = \lozenge \ . \tag{11}$$

This rewrite rule reflects that the empty continuation is an identity element for sequential composition. Similar rewrite rules will be added to handle identity elements for other composition operators in the sequel.

Please observe that the evaluation rules for atomic statements always produce a set of continuation traces ending with empty continuation $\mathrm{K}(\lozenge)$ to indicate a return of control.

Example 3.1. Consider the sequential statement $s_{seq} = (x := 1; \ y := x + 1)$. To explain how its evaluation is performed, we start from an arbitrary symbolic state σ (to be instantiated later by a composition rule). The equation for sequential composition yields

$$\mathrm{val}_\sigma(s_{seq}) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 1] \cdot \mathrm{K}(y := x + 1)\} \ . \tag{12}$$

To perform this evaluation, we need the result of evaluating the first assignment in the context of σ:

$$\mathrm{val}_\sigma(x := 1) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 1] \cdot \mathrm{K}(\lozenge)\} \ .$$

In the composition rules it might be necessary to evaluate the empty continuation, so we must define it. As explained Sect. 2.5, in contrast to evaluation of statements, evaluating $\lozenge$ produces the empty set of traces:

$$\mathrm{val}_\sigma(\lozenge) = \{\} \ . \tag{13}$$

Proposition 3.2. *Given a concrete state σ and a program s such that $vars(s) \subseteq \mathrm{dom}(\sigma)$, then $\mathrm{val}_\sigma(s)$ is a set of concrete continuation traces of the form $pc \triangleright \tau \cdot \mathrm{K}(s')$. There is exactly one continuation trace with a consistent path condition.*

3.2 Trace Composition

Local traces are composed into concrete global ones. As WHILE is sequential and deterministic, we expect to obtain exactly one trace, provided that the execution starts in a concrete state that assigns values to all the variables of a program. Proposition 3.2 ensures that all local evaluation rules produce concrete traces in this case.

The task of the composition rule for WHILE-programs is to repeatedly evaluate one statement at a time in a concrete state until the next continuation, then stitch the resulting concrete traces together. Given a concrete trace sh with final state σ and a continuation $\mathrm{K}(s)$, we evaluate s starting in σ. The result is a set of conditioned traces from which one trace with a consistent path condition and a trailing continuation $\mathrm{K}(s')$ is chosen.[2] The chosen trace τ is joined with the given trace sh. Afterwards, the composition rule can be applied again to the extended concrete trace and $\mathrm{K}(s')$.

$$\frac{\sigma = \mathrm{last}(sh) \quad pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma(s) \quad pc \text{ consistent}}{sh, \mathrm{K}(s) \to sh ** \tau, \mathrm{K}(s')} \tag{14}$$

The rule assumes that s is evaluated to a concrete trace so that $sh ** \tau$ stays concrete. At this stage, symbolic traces do not yet figure. This works as long as Proposition 3.2 ensures that $pc \triangleright \tau$ is a concrete trace, however, in general the proviso $vars(s) \subseteq \mathrm{dom}(\sigma)$ does not hold. There are several approaches to address this. For example, it is sufficient to consider the *non-initialised* variables of s, instead of all of them. We avoid a lengthy definition to address this technicality by simply assuming that $\mathrm{last}(sh)$ defines all variables of s. This can be easily achieved by initialising all variables to default values at program start, as done below.

3.3 Global Trace Semantics

Let $sh, \mathrm{K}(s) \overset{*}{\to} sh', \mathrm{K}(s')$ denote the transitive closure of applying rule (14), expressing that $sh', \mathrm{K}(s')$ can be reached from $sh, \mathrm{K}(s)$ in zero or more steps. Let $\underline{sh}$ denote a possibly infinite trace. Such infinite traces never appear inside the rules, but only as a result from constructing a maximal sequence of successive *applications* of trace composition rules. Formally, an infinite trace is the limit of an increasing sequence of traces, in contrast to a trace that has no final element.

Definition 3.3 (Program Semantics). Given a program s and a state σ, let

$$sh_0,\ \mathrm{K}(s_0) \to sh_1,\ \mathrm{K}(s_1) \to \cdots$$

be a maximal sequence obtained by the repeated application of rule (14), starting from $\langle\sigma\rangle, \mathrm{K}(s)$. If the sequence is finite, then it must have the form[3]

$$\langle\sigma\rangle, \mathrm{K}(s) \overset{*}{\to} \underline{sh}, \mathrm{K}(\mathbb{0})\ .$$

If a sequence is infinite (and does not reach an empty continuation), we let $\underline{sh} = \lim_{i\to\infty} sh_i$ for this sequence. The set of all such possibly infinite traces $\underline{sh}$ for a program s starting from a state σ is denoted $\mathbf{Tr}(s, \sigma)$.

Please observe that the traces produced by a sequence of rule applications grow monotonically: In Definition 3.3 we have $sh_i \leq sh_{i+1}$ for all $i \geq 0$, where $\leq$ is the prefix order on traces. The traces and continuations may reach a fixpoint or grow indefinitely, in both cases, the limit in Definition 3.3 is well-defined.

For any statement s, let I_s be the state, where $[x \mapsto 0]$ for all $x \in vars(s)$ (for simplicity, assume all variables are of integer type—the generalisation is obvious). To help readability, we sometimes omit such default values from the states in the following examples.

Example 3.4. The only trace for **skip** starting from a state σ is the singleton trace $\langle\sigma\rangle$, thus $\mathbf{Tr}(\mathbf{skip}, \sigma) = \{\langle\sigma\rangle\}$. It follows that **skip** is the unit of the sequential composition operator: For any statement s and state σ, we have $\mathbf{Tr}(\mathbf{skip}; s, \sigma) = \mathbf{Tr}(s; \mathbf{skip}, \sigma) = \mathbf{Tr}(s, \sigma)$.

[2]For a deterministic language like WHILE, there is exactly one trace, but the rule is designed to work for the non-deterministic extension below as well.
[3]Observe that $sh, \mathrm{K}(\mathbb{0})$ is the end of the execution because the evaluation of $\mathrm{K}(\mathbb{0})$ returns $\{\}$ such that the composition rule (14) is no longer applicable. The empty continuation is obtained once the whole program has been evaluated.

Example 3.5. We apply rule (14) to $\langle I_{seq}\rangle, \mathrm{K}(s_{seq})$, the program from Example 3.1. To obtain the premise, we instantiate Equation (12) with $\sigma = I_{seq}$.

$$\frac{\begin{array}{c} I_{seq} = \mathrm{last}(\langle I_{seq}\rangle) \qquad \emptyset \text{ consistent} \\ \emptyset \triangleright \langle I_{seq}\rangle \curvearrowright I_{seq}[x \mapsto 1] \cdot \mathrm{K}(y := x+1) \in \mathrm{val}_{I_{seq}}(x := 1; y := x+1) \end{array}}{\langle I_{seq}\rangle, \mathrm{K}(x := 1; y := x+1) \to \langle I_{seq}\rangle \curvearrowright I_{seq}[x \mapsto 1], \mathrm{K}(y := x+1)} \tag{15}$$

For the subsequent rule application, it is necessary to evaluate the program $y := x + 1$ in the continuation. Again, we do this for a general state, while in the rule we use $\sigma = I_{seq}[x \mapsto 1]$ (from now on we omit I_{seq}):

$$\mathrm{val}_\sigma(y := x+1) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[y \mapsto \mathrm{val}_\sigma(x+1)] \cdot \mathrm{K}(\mathbb{0})\}$$

$$\frac{\begin{array}{c} [x \mapsto 1] = \mathrm{last}(\langle I_{seq}\rangle \curvearrowright [x \mapsto 1]) \qquad \emptyset \text{ consistent} \\ \emptyset \triangleright \langle [x \mapsto 1]\rangle \curvearrowright [x \mapsto 1][y \mapsto \mathrm{val}_{[x\mapsto 1]}(x+1)] \cdot \mathrm{K}(\mathbb{0}) \in \mathrm{val}_{[x\mapsto 1]}(y := x+1) \end{array}}{\langle I_{seq}\rangle \curvearrowright [x \mapsto 1], \mathrm{K}(y := x+1) \to \langle I_{seq}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2], \mathrm{K}(\mathbb{0})} \tag{16}$$

Hence, $\mathbf{Tr}(\mathsf{x{:=}1;\ y{:=}x{+}1}, I_{seq}) = \{\langle I_{seq}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2]\}$.

Example 3.6. We illustrate the semantics of a non-terminating loop without effect. Consider the program $s_{loop} = (\mathbf{\texttt{while}}\ \mathtt{tt}\ \{\mathbf{\texttt{skip}}\};\ x := 1)$, where we observe a fix-point in the local evaluation and in the global composition rule such that we actually never evaluate $x := 1$. To observe this effect, as with previous examples, we start with an arbitrary symbolic state σ. After applying the rules for **`while`** and **`if`**, we obtain

$$\begin{array}{rcl} \mathrm{val}_\sigma(s_{loop}) & = & \mathrm{val}_\sigma(\mathbf{\texttt{if}}\ \mathtt{tt}\ \{\mathbf{\texttt{skip}}; \mathbf{\texttt{while}}\ \mathtt{tt}\ \{\mathbf{\texttt{skip}}\}\}; x := 1) \\ & = & \{\{\mathtt{tt}\} \triangleright \langle\sigma\rangle \cdot \mathrm{K}(\mathbf{\texttt{skip}}; s_{loop}),\ \ \{\mathtt{ff}\} \triangleright \langle\sigma\rangle \cdot \mathrm{K}(x := 1)\ \} \end{array} \tag{17}$$

We can apply the rules for **`skip`** and sequential composition to $(\mathbf{\texttt{skip}}; s_{loop})$ and obtain:

$$\mathrm{val}_\sigma(\mathbf{\texttt{skip}};\ s_{loop}) = \{\emptyset \triangleright \langle\sigma\rangle \cdot \mathrm{K}(s_{loop})\}\ \ . \tag{18}$$

We now apply rule (14) to $\langle I_{loop}\rangle, \mathrm{K}(s_{loop})$, where we instantiate Equation (17) with $\sigma = I_{loop}$:

$$\frac{\begin{array}{c} I_{loop} = \mathrm{last}(\langle I_{loop}\rangle) \qquad \{\mathtt{tt}\} \text{ consistent} \\ \{\mathtt{tt}\} \triangleright \langle I_{loop}\rangle \cdot \mathrm{K}(\mathbf{\texttt{skip}}; s_{loop}) \in \mathrm{val}_{I_{loop}}(s_{loop}) \end{array}}{\langle I_{loop}\rangle, \mathrm{K}(s_{loop}) \to \langle I_{loop}\rangle, \mathrm{K}(\mathbf{\texttt{skip}};\ s_{loop})} \tag{19}$$

since $\langle I_{loop}\rangle ** \langle I_{loop}\rangle = \langle I_{loop}\rangle$. We apply rule (14) one more time, using Equation (18) and reach the initial configuration $\langle I_{loop}\rangle, \mathrm{K}(s_{loop})$ again:

$$\frac{\begin{array}{c} I_{loop} = \mathrm{last}(\langle I_{loop}\rangle) \qquad \emptyset \text{ consistent} \\ \emptyset \triangleright \langle I_{loop}\rangle \cdot \mathrm{K}(s_{loop}) \in \mathrm{val}_{I_{loop}}(\mathbf{\texttt{skip}};\ s_{loop}) \end{array}}{\langle I_{loop}\rangle, \mathrm{K}(\mathbf{\texttt{skip}}; s_{loop}) \to \langle I_{loop}\rangle, \mathrm{K}(s_{loop})} \tag{20}$$

Hence, the set of traces for s_{loop} starting from I_{loop} is the singleton trace $\mathbf{Tr}(s_{loop}, I_{loop}) = \{\langle I_{loop}\rangle\}$.

Example 3.6 shows that, in contrast to denotational or big step semantics, we only define trace semantics step by step. Thus, in our context, it is not a problem if the semantics of an infinite loop is a finite trace as the second part of the computation after an infinite loop (in the example, $x := 1$) is not evaluated.

Alternatively, we could have defined the result of evaluating **`while`** $\mathtt{tt}$ $\{$**`skip`**$\}$ to be an infinite trace by duplicating the state at the beginning of each loop body execution. However, in contrast to a compositional semantics, where this would matter, here it is merely a design choice of little consequence: Infinite traces are never composed in our case.

Example 3.7. A non-terminating loop with effects on the state has a different semantics compared to Example 3.6. Consider the program $s_w = (x := 0;\ \mathbf{while}\ \mathrm{tt}\ \{x := x + 1\})$. The LAGC semantics for this non-terminating program is defined as a limit of partial traces. It is an infinite trace with x indefinitely increasing:

$$\mathbf{Tr}(s_w, I_w) = \{\langle I_w\rangle \curvearrowright [x \mapsto 0] \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 2] \curvearrowright \ldots\}$$

3.4 Discussion

As mentioned, as long as we start semantic evaluation in a sufficiently initialised concrete state, Proposition 3.2 ensures that only concrete traces will be generated by local rules. Consequently, we have a semantics that can be aptly called modular and compositional (exactly one independent rule per language construct and a uniform composition rule), but the overhead introduced with *symbolic* traces is not yet justified.

The advantages offered by symbolic traces and states are realised in the following two sections. In Section 4 we define a program logic for WHILE that employs symbolic traces at the level of the *calculus*. This close correspondence between semantics and deduction system is the basis for an intuitive soundness proof for the deductive system.

In Sections 5–7 we extend WHILE with a number of complex instructions, in particular, for parallel programming. It will be seen that a wide range of concurrency paradigms fit naturally into the LAGC semantic framework.

4 A PROGRAM LOGIC AND SOUND CALCULUS FOR WHILE

We provide a dynamic logic (DL) [44] as well as a calculus for reasoning about the correctness of WHILE-programs that is sound relative to our semantics. In deductive verification dynamic logic [4, 12] offers technical advantages over Hoare logic [51]: it is syntactically closed with respect to first-order logic, more expressive, and cleanly separates first-order ("logical") variables from program variables [4, p. 50]. Below we define program formulas of the form $\psi \longrightarrow \tau\,[s]\,\phi$, where τ is a finite symbolic trace, s any WHILE-statement, ψ a first-order formula, and ϕ a formula that in turn may contain programs. The intuitive meaning is that any terminating execution of s continuing a trace that concretises τ and started in a state that satisfies ψ, must end in a state that satisfies ϕ. The modality $[s]$ corresponds to a *continuation* in the semantics of WHILE, represented symbolically.

The unusual aspect of this setup is the presence of a symbolic trace τ inside a formula. It aligns with our locally abstract semantics, but it is also justified, because—unlike a semantics—rule schemata in calculus rules *necessarily* deal with symbolic values: a verification calculus aims at proving a property that holds for *all* inputs of a program, not merely for a single run. Nevertheless, the presence of symbolic traces inside formulas may appear as insufficiently syntactic or as an inappropriate intrusion of the semantics into the calculus. However, efficient syntactic representations of symbolic assignments are fairly common and well understood: for example, the deductive verification system KeY uses symbolic *updates*[4] [4] and in the B-method explicit generalised substitutions play a comparable role [1]. To keep the calculus as general as possible, we do not commit to a particular implementation of symbolic traces.

The calculus rules given below will symbolically execute a program s in a sequent of the form $\psi \longrightarrow \tau\,[s]\,\phi$ and produce verification conditions of the form $\Gamma \Rightarrow \tau\,\phi$, where ϕ is a first-order formula and τ a symbolic trace.

[4]Updates can be viewed as a syntactic, efficient, lazy representation of symbolic traces: concrete values are not eagerly substituted and assignments are kept in single static (SSA) shape.

4.1 Dynamic Logic

Given a signature Σ with typed function and predicate symbols and a set V of logical variables which is disjoint from the symbols in Σ, let $\mathbf{Terms}(\Sigma, V)$ denote the well-formed terms over Σ and V (respecting type compatibility). Note that the logical variables in V are disjoint from program variables *Var*. Unlike the latter, the logical variables can be bound by quantifiers and do not change their value during program execution.

Definition 4.1 (DL Formula). Let Σ be a signature and V a set of logical variables disjoint from Σ. The language $\mathbf{DL}(\Sigma, V)$ of formulas in dynamic logic is defined inductively as follows:

(1) $B \in \mathbf{DL}(\Sigma, V)$ if $B \in \mathbf{Terms}(\Sigma, V)$ and the type of B is Boolean
(2) $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \rightarrow \phi_2$, $\phi_1 \leftrightarrow \phi_2 \in \mathbf{DL}(\Sigma, V)$ if ϕ_1, $\phi_2 \in \mathbf{DL}(\Sigma, V)$
(3) $\exists x \cdot \phi$, $\forall x \cdot \phi \in \mathbf{DL}(\Sigma, V)$ if $x \in V$ and $\phi \in \mathbf{DL}(\Sigma, V)$
(4) $[\mathrm{s}]\, \phi \in \mathbf{DL}(\Sigma, V)$ if s is a, possibly empty, WHILE-program and $\phi \in \mathbf{DL}(\Sigma, V)$

We often omit the signature and variable set of $\phi \in \mathbf{DL}(\Sigma, V)$ and simply write $\phi \in \mathbf{DL}$.

There is a subtle point about empty programs. As explained above, the program s in the modality can be viewed as the continuation of the current trace. The rules of the calculus defined below will symbolically execute s from left to right, until the program remaining to be executed is empty. For this reason, we allow empty programs, represented by **`skip`**, in clause (4). The empty program is the identity element for the composition operator (see Example 3.4). However, as pointed out in Sect. 2.5, the continuation with the empty program is not the same as the empty continuation. The former requires a semantics that amounts to a singleton trace, which is reflected in Definition 4.3 below.

Definition 4.2 (Substitution). Given a language $\mathbf{DL}(\Sigma, V)$ and a set of logical variables $V' = \{x_1, \ldots, x_n\}$ such that $V' \subseteq V$, a *substitution* $[x_1/t_1, \ldots, x_n/t_n]$ is a function $V' \longrightarrow \mathbf{Terms}(\Sigma, V)$ which associates with every $x_i \in V'$ a type-compatible term $t_i \in \mathbf{Terms}(\Sigma, V)$.

Denote by $\phi[x_1/t_1, \ldots, x_n/t_n]$ the application of a substitution $[x_1/t_1, \ldots, x_n/t_n]$ to a logical formula $\phi \in \mathbf{DL}(\Sigma, V)$. Observe that applying a substitution removes occurrences of *logical variables* in a formula and does not affect programs. The application of the substitution has a straightforward inductive definition over ϕ (omitted here).

We write $sh \models \phi$ and $\sigma \models \phi$ to denote[5] that a formula ϕ is valid for a *concrete* trace sh and in a *concrete* state σ, respectively (equivalently, sh and σ satisfy ϕ). Formally, satisfiability can be defined as follows:

Definition 4.3 (Satisfiability of DL Formulas). Let $\phi \in \mathbf{DL}(\Sigma, V)$ be a DL formula, $B \in \mathbf{Terms}(\Sigma, V)$ a Boolean term, σ a concrete state and sh a finite, concrete trace.

$\sigma \models B \iff \mathrm{val}_\sigma(B) = \mathtt{tt}$
$\sigma \models \neg\phi \iff \sigma \not\models \phi$
$\sigma \models \phi_1 \wedge \phi_2 \iff \sigma \models \phi_1$ and $\sigma \models \phi_2$ (analogous for the remaining propositional connectives)
$\sigma \models \exists x \cdot \phi \iff \sigma \models \phi[x/t]$ for some substitution $[x/t]$ where $t \in \mathbf{Terms}(\Sigma, V)$, variable-free
$\sigma \models \forall x \cdot \phi \iff \sigma \models \phi[x/t]$ for all substitutions $[x/t]$ where $t \in \mathbf{Terms}(\Sigma, V)$, variable-free
$\sigma \models [\mathrm{s}]\, \phi \iff sh \models \phi$ for all $sh \in \mathbf{Tr}(\mathrm{s}, \sigma)$
$sh \models \phi \iff sh$ non-empty, $\mathrm{last}(sh) = \sigma$, and $\sigma \models \phi$

Observe that the finiteness of sh in the clause "$\sigma \models [\mathrm{s}]\, \phi \iff sh \models \phi$ for all $sh \in \mathbf{Tr}(\mathrm{s}, \sigma)$" implies a partial correctness semantics.

[5]To simplify presentation, we assume a fixed standard interpretation of the symbols in Σ. It is straightforward to accommodate undefined symbols and relativise satisfiability to first-order models.

4.2 Calculus

As usual in deductive verification [42], we define a calculus operating on *sequents.* These have the form $\Gamma \Rightarrow \tau\ \phi$, where Γ is a set of formulas thought to be implicitly conjoined (called the *antecedent*), the symbol $\Rightarrow$ can be read as implication, τ is a non-empty, finite symbolic trace, and ϕ is a DL formula. Without loss of generality, we can assume that Γ consists of a single formula ψ.

Definition 4.4 (Valid Sequent). A sequent is *valid*, denoted $\models \psi \Rightarrow \tau\ \phi$, iff for *all* concretisation mappings ρ such that $\mathrm{first}(\rho(\tau)) \models \psi$, we have that

$$\mathrm{last}(\rho(\tau)) \models \phi \ .$$

The proof system is based on analysing statements in explicit contexts, inspired by the context-reduction semantics of Felleisen and Hieb [37]. Context-reduction semantics separates local evaluation from the context in which that evaluation occurs, for example, to decouple the evaluation of expressions from the evaluation of statements. Thus, in a context-reduction semantics, a context may be seen as the continuation of an evaluation. In our proof system, a context may be seen as an abstraction for a composition operator in the language syntax. In contrast to context-reduction semantics, which uses contexts to lift premises in transition rules to define a single execution path, our contexts reflect the composition operators of the considered language in the construction of the proof tree over symbolic execution paths. In the proof system, contexts will be passed around and manipulated in the tree, reminiscent of continuations in the LAGC semantics.

Contexts for Sequential Composition. We introduce the concept of a "hole" and that of filling a hole. Let contexts be defined as follows for sequential composition:

$$C \in \mathit{Context} ::= \bullet \mid C; s \ .$$

A context has exactly one hole. We denote by $C[s]$ the statement in which the hole of the context C has been replaced by s and by $C_1[C_2]$ the context in which the hole in the context C_1 has been filled by the context C_2. Any statement is equivalent to itself in the empty context (i.e., $s = \bullet[s]$) and $C[s_1; s_2]$ is equivalent to $C[\bullet; s_2][s_1]$.

Contexts are used to express that we *focus* our analysis of a statement on a constituent statement; we denote by $s\ @\ C$ that we analyse s in a given context C. Observe that any statement s' can be expressed with such a focus $s\ @\ C$ if $s' = C[s]$. In a given state σ, we define the traces of statements that occur in a context in terms of the traces of their corresponding context-free continuations (Definition 3.3), as follows:

$$\mathbf{Tr}(s@C, \sigma) = \{sh \ast\ast \underline{sh'} \mid \langle\sigma\rangle, \mathrm{K}(s) \rightarrow sh,\ \mathrm{K}(s') \wedge \underline{sh'} \in \mathbf{Tr}(C[s'], \mathrm{last}(sh))\} \ .$$

The satisfiability of statements that occur in a context is then covered by Definition 4.3. In particular, for sequential contexts as defined above, we have $\mathbf{Tr}(s@C, \sigma) = \mathbf{Tr}(C[s], \sigma)$ for any σ, and consequently

$$\sigma \models [s\ @\ C]\phi \iff \sigma \models [C[s]]\phi \ .$$

It is further interesting to define *contextual trace equivalence*, i.e., the conditions under which two statements have the same trace sets in all contexts. There is a locality to local evaluation in the sense that local evaluation generates subtraces in the trace semantics. For sequential contexts, contextual trace equivalence reduces to trace equivalence, as expressed by the following lemma:

Lemma 4.5. *Let s_1, s_2 be programs, σ a state and C a context. If* $\mathbf{Tr}(s_1, \sigma) = \mathbf{Tr}(s_2, \sigma)$*, then* $\mathbf{Tr}(C[s_1], \sigma) = \mathbf{Tr}(C[s_2], \sigma)$.

Proof. The proof is by cases over contexts.

Case Empty Context. Then $C = \bullet$, so $\mathbf{Tr}(C[s_1], \sigma) = \mathbf{Tr}(s_1, \sigma) = \mathbf{Tr}(s_2, \sigma) = \mathbf{Tr}(C[s_2], \sigma)$.

Case Non-Empty Context. Then $C = \bullet; s$ for some statement s. We need to show that $\mathbf{Tr}(s_1; s, \sigma) = \mathbf{Tr}(s_2; s, \sigma)$. Consider an arbitrary trace $\underline{sh} \in \mathbf{Tr}(s_1; s, \sigma)$. There are two subcases, depending on whether the initial segment $\underline{sh}_1 \in \mathbf{Tr}(s_1, \sigma)$ is finite or not.

Subcase 1. $\underline{sh}_1$ is finite. Since $\underline{sh}_1 \in \mathbf{Tr}(s_1, \sigma)$, we know that $\langle\sigma\rangle, \mathrm{K}(s_1) \overset{*}{\rightarrow} \underline{sh}_1, \mathrm{K}(\square)$. It follows from Equations (10) and (14) that $\langle\sigma\rangle, \mathrm{K}(s_1; s) \overset{*}{\rightarrow} \underline{sh}_1, \mathrm{K}(s)$. Let $\underline{sh}' \in \mathbf{Tr}(s, \mathrm{last}(\underline{sh}))$ such that $\underline{sh} = \underline{sh}_1 ** \underline{sh}'$. Since $\underline{sh}_1 \in \mathbf{Tr}(s_2, \sigma)$, we similarly have $\langle\sigma\rangle, \mathrm{K}(s_2; s) \overset{*}{\rightarrow} \underline{sh}_1, \mathrm{K}(s)$, and consequently $\underline{sh} \in \mathbf{Tr}(s_2; s, \sigma)$

Subcase 2. $\underline{sh}_1$ is infinite. Then $\underline{sh}_1$ is the limit of the reduction sequence obtained by applying Equation (14) to $\langle\sigma\rangle, \mathrm{K}(s_1)$, which never reaches an empty continuation. It follows that $\underline{sh}_1$ is also a limit of the reduction sequence obtained from $\langle\sigma\rangle, \mathrm{K}(s_1; s)$, so $\underline{sh}_1 = \underline{sh}$. Consequently, $\underline{sh} \in \mathbf{Tr}(s_2, \sigma)$, and we similarly have that $\underline{sh} \in \mathbf{Tr}(s_2; s, \sigma)$. □

We denote by $\vdash \psi \Rightarrow \tau\, [\mathrm{s} \;@\; C]\, \phi$ that the DL sequent $\psi \Rightarrow \tau\, [\mathrm{s} \;@\; C]\, \phi$ can be derived in the proof system, and elide the topmost context $\bullet$ in sequents to simplify notation; i.e., we write $\psi \Rightarrow \tau\, [\mathrm{s}]\, \phi$ instead of $\psi \Rightarrow \tau\, [\mathrm{s} \;@\; \bullet]\, \phi$. To prove that a postcondition Post holds after termination of program s under a precondition Pre, it is sufficient to derive the sequent

$$\mathrm{Pre} \Rightarrow \langle\sigma_*\rangle\, [\mathrm{s}]\, \mathrm{Post}\ ,$$

s is analysed in the topmost context and $\sigma_*(x) \mapsto *$ for all $x \in \mathit{Var}$. By construction, the only symbolic variables that occur in a well-formed symbolic trace τ are introduced in the initial state of τ. Thus, all path conditions are expressed in terms of the initial symbolic state σ_*.

The proof rules for statements with contexts are given in Figure 3. To make the correspondence with the semantics of WHILE more immediate, we represent path conditions as a separate set in the antecedent of the sequents. Observe that this rule set is incomplete as it leaves open how a sequent of the form $\Gamma, pc \Rightarrow \tau\, \phi$ should be derived, where ϕ is a first-order formula. For generality, we do not commit to a specific representation of symbolic traces and how they are applied to first-order formulas. For example, if symbolic traces are represented with updates, then the rules in [4, Ch. 2] can be used. Another source of incompleteness is the lack of a loop invariant rule. A proof system with rules for updates as well as loop invariants based on a LAGC semantics for a *sequential* language with loops and recursive procedures is contained in [20]. It requires a non-trivial amount of technical machinery which is why we refrain from copying it here.

In the soundness proof below we assume that there is a sound calculus able to derive $\Gamma, pc \Rightarrow \tau\, \phi$ for first-order formulas ϕ.

Theorem 4.6 (Soundness). *Let τ be a well-formed symbolic trace and let ϕ, ψ be DL formulas. If $\vdash \psi \Rightarrow \tau\, \phi$, then $\models \psi \Rightarrow \tau\, \phi$.*

Proof. The proof is by induction over the derivation of $\vdash \psi \Rightarrow \tau\, \phi$, with one case for each proof rule, and makes use of Lemma 4.5 for finite traces. We assume the soundness of first-order (or "program-free") formulas; i.e., if $\vdash \phi$ then $\models \phi$ for any first-order formula ϕ.

Case Empty. Assume $\vdash \psi \Rightarrow \tau\, [\mathbf{skip}]\, \phi$. By rule Empty, we get $\vdash \psi \Rightarrow \tau\, \phi$. We proceed by cases over ϕ.

Subcase 1. ϕ is a first-order formula. Then, by assumption, $\models \psi \Rightarrow \tau\, \phi$, which means that for any concretisation mapping ρ, if $\mathrm{first}(\rho(\tau)) \models \psi$, then $\mathrm{last}(\rho(\tau)) \models \phi$. The latter is, by Definition 4.3, equivalent to $\mathrm{last}(\rho(\tau)) \models [\mathbf{skip}]\, \phi$. Since ρ was arbitrary, this yields $\models \psi \Rightarrow \tau\, [\mathbf{skip}]\, \phi$.

Subcase 2. ϕ is not first-order formula. In this case, ϕ can be any DL formula. Then, $\models \psi \Rightarrow \tau\, \phi$ follows from the induction hypothesis (hereafter, IH), and the rest of the proof is analogous to Subcase 1.

$$\textsc{(Empty)}\ \frac{\Gamma, pc \Rightarrow \tau\, \phi}{\Gamma, pc \Rightarrow \tau\, [\texttt{skip}]\, \phi} \qquad \textsc{(While)}\ \frac{\Gamma, pc \Rightarrow \tau\, [(\texttt{if}\ e\ \{s; \texttt{while}\ e\ \{s\}\})\ @\ C]\, \phi}{\Gamma, pc \Rightarrow \tau\, [\texttt{while}\ e\ \{s\}\ @\ C]\, \phi}$$

$$\textsc{(Assign)}\ \frac{\begin{array}{c}\mathrm{last}(\tau) = \sigma \\ \sigma' = \sigma[x \mapsto \mathrm{val}_\sigma(e)] \\ \Gamma, pc \Rightarrow \tau \curvearrowright \sigma'\, [C[\texttt{skip}]]\, \phi\end{array}}{\Gamma, pc \Rightarrow \tau\, [x := e\ @\ C]\, \phi} \qquad \textsc{(Cond)}\ \frac{\begin{array}{c}\mathrm{last}(\tau) = \sigma \\ \Gamma, pc \cup \{\mathrm{val}_\sigma(e) = \mathtt{tt}\} \Rightarrow \tau\, [C[s]]\, \phi \\ \Gamma, pc \cup \{\mathrm{val}_\sigma(e) = \mathtt{ff}\} \Rightarrow \tau\, [C[\texttt{skip}]]\, \phi\end{array}}{\Gamma, pc \Rightarrow \tau\, [\texttt{if}\ e\ \{s\}\ @\ C]\, \phi}$$

$$\textsc{(Skip)}\ \frac{\begin{array}{c}C \neq \bullet \\ \Gamma, pc \Rightarrow \tau\, [C\, [\texttt{skip}]]]\, \phi\end{array}}{\Gamma, pc \Rightarrow \tau\, [\texttt{skip}\ @\ C]\, \phi} \qquad \textsc{(Seq1)}\ \frac{\Gamma, pc \Rightarrow \tau\, [s\ @\ C]]\, \phi}{\Gamma, pc \Rightarrow \tau\, [\texttt{skip}; s\ @\ C]\, \phi} \qquad \textsc{(Seq2)}\ \frac{\begin{array}{c}s_1 \neq \texttt{skip} \\ \Gamma, pc \Rightarrow \tau\, [s_1\ @\ C[\bullet; s_2]]\, \phi\end{array}}{\Gamma, pc \Rightarrow \tau\, [s_1; s_2\ @\ C]\, \phi}$$

Fig. 3. Dynamic logic sequent calculus for WHILE.

Case While. This case follows from the case for Cond. Assume that $\vdash \psi \Rightarrow \tau\, [\texttt{while}\ e\ \{\ s\ \}\ @\ C]\, \phi$. By rule While, we get $\vdash \psi \Rightarrow \tau\, [\texttt{if}\ \mathrm{e}\ \{\ s; \texttt{while}\ e\ \{\ s\ \}\}\ @\ C]\, \phi$. Then, by IH we have $\models \psi \Rightarrow \tau\, [\texttt{if}\ \mathrm{e}\ \{\ s; \texttt{while}\ e\ \{\ s\ \}\}\ @\ C]\, \phi$. Let $\mathrm{last}(\tau) = \sigma$. For any concretisation mapping ρ and any $\underline{sh} \in \mathbf{Tr}(C\, [\texttt{if}\ \mathrm{e}\ \{\ s; \texttt{while}\ e\ \{\ s\ \}\}]\, ,\, \rho(\sigma))$, if $\mathrm{first}(\rho(\tau) ** \underline{sh}) \models \psi$ then $\mathrm{last}(\rho(\tau) ** \underline{sh}) \models \phi$. By Equation (9) and Lemma 4.5, we have $\mathbf{Tr}(C[\texttt{while}\ e\ \{\ s\ \}]\, , \rho(\sigma)) = \mathbf{Tr}(C\, [\texttt{if}\ \mathrm{e}\ \{\ s; \texttt{while}\ e\ \{\ s\ \}\}]\, , \rho(\sigma))$. Therefore $\rho(\sigma) \models [\texttt{while}\ e\ \{\ s\ \}\ @\ C]\, \phi$ and, since ρ was arbitrary, it follows that $\models \psi \Rightarrow \tau\, [\texttt{while}\ e\ \{\ s\ \}\ @\ C]\, \phi$.

Case Assign. Assume $\vdash \psi \Rightarrow \tau\, [x := e\ @\ C]\, \phi$. By rule Assign, we get $\vdash \psi \Rightarrow \tau \curvearrowright \sigma'\, [C\, [\texttt{skip}]]\, \phi$ where $\mathrm{last}(\tau) = \sigma$ and $\sigma' = \sigma[x \mapsto \mathrm{val}_\sigma(e)]$. By IH we have $\models \psi \Rightarrow \tau \curvearrowright \sigma'\, [C\, [\texttt{skip}]]\, \phi$. Hence, for any concretisation mapping ρ, if $\mathrm{first}(\rho(\tau \curvearrowright \sigma')) \models \psi$, then $\mathrm{last}(\rho(\tau \curvearrowright \sigma')) \models [C\, [\texttt{skip}]]\, \phi$, i.e. $\rho(\sigma') \models [C\, [\texttt{skip}]]\, \phi$. By Equations (7), (10) and (14), we have $\mathbf{Tr}(x := e\ @\ C, \rho(\sigma)) = \{\langle \rho(\sigma) \rangle \curvearrowright \rho(\sigma') ** \underline{sh} \mid \underline{sh} \in \mathbf{Tr}(C\, [\texttt{skip}]\, , \rho(\sigma'))\}$. Therefore, $\mathrm{last}(\rho(\tau)) \models [x := e\ @\ C]\, \phi$ and, since ρ was arbitrary and $\mathrm{first}(\rho(\tau \curvearrowright \sigma')) = \mathrm{first}(\rho(\tau))$, we have $\models \psi \Rightarrow \tau\, [x := e\ @\ C]\, \phi$.

Case Cond. Assume $\vdash \psi \Rightarrow \tau\, [\texttt{if}\ \mathrm{e}\ \{\mathrm{s}\}\ @\ C]\, \phi$. By rule Cond, we get $\vdash \psi, \mathrm{val}_\sigma(e) = \mathtt{ff} \Rightarrow \tau\, [C\, [\texttt{skip}]]\, \phi$ and $\vdash \psi, \mathrm{val}_\sigma(e) = \mathtt{tt} \Rightarrow \tau\, [C\, [s]]\, \phi$, where $\mathrm{last}(\tau) = \sigma$. Correspondingly, we have two induction hypotheses: $\models \psi, \mathrm{val}_\sigma(e) = \mathtt{ff} \Rightarrow \tau\, [C\, [\texttt{skip}]]\, \phi$ (hereafter, IH1) and $\models \psi, \mathrm{val}_\sigma(e) = \mathtt{tt} \Rightarrow \tau\, [C\, [s]]\, \phi$ (hereafter, IH2). Let ρ be any concretisation mapping and assume $\mathrm{first}(\rho(\tau)) \models \psi$. Depending on the value of $\mathrm{val}_{\rho(\sigma)}(e)$, exactly one of the subcases applies.

Subcase IH1. We can assume $\mathrm{val}_{\rho(\sigma)}(e) = \mathtt{ff}$ and, by IH1, $\models \psi, \mathrm{val}_\sigma(e) = \mathtt{ff} \Rightarrow \tau\, [C\, [\texttt{skip}]]\, \phi$, which together gives $\mathrm{last}(\rho(\tau)) \models [C\, [\texttt{skip}]]\, \phi$. By definition, $sh \models [C\, [\texttt{skip}]]\, \phi$ for all finite $sh \in \mathbf{Tr}(C\, [\texttt{skip}]\, , \mathrm{last}(\rho(\tau)))$. By Equations (8), (10) and (14) and Lemma 4.5, we have $\mathbf{Tr}(C\, [\texttt{skip}]\, , \rho(\sigma)) = \mathbf{Tr}(C\, [\texttt{if}\ \mathrm{e}\ \{\mathrm{s}\}]\, , \rho(\sigma))$, observing that the path condition is consistent. Therefore, $sh' \models [C\, [\texttt{if}\ \mathrm{e}\ \{\mathrm{s}\}]]\, \phi$ for all finite $sh' \in \mathbf{Tr}(C\, [\texttt{if}\ \mathrm{e}\ \{\mathrm{s}\}]\, , \mathrm{last}(\rho(\tau)))$ and so $\models \psi \Rightarrow \tau\, [C\, [\texttt{if}\ \mathrm{e}\ \{\mathrm{s}\}]]\, \phi$, which is equivalent to $\models \psi \Rightarrow \tau\, [\texttt{if}\ \mathrm{e}\ \{\mathrm{s}\}\ @\ C]\, \phi$.

Subcase IH2. We can assume $\mathrm{val}_{\rho(\sigma)}(e) = \mathtt{tt}$ and proceed by a completely analogous argument as in the other subcase.

Case Skip. By the definition of satisfiability for statements in contexts.

Case Seq1. Assume $\vdash \psi \Rightarrow \tau\, [\texttt{skip}; s\ @\ C]\, \phi$. By rule Seq1, we get $\vdash \psi \Rightarrow \tau\, [s\ @\ C]\, \phi$, and by IH we have $\models \psi \Rightarrow \tau\, [s\ @\ C]\, \phi$. Hence, for any concretisation mapping ρ, if $\mathrm{first}(\rho(\tau)) \models \psi$, then $\mathrm{last}(\rho(\tau)) \models [C[s]]\,\phi$. Then $sh \models [\mathtt{skip}]\,\phi$ for all finite $sh \in \mathbf{Tr}(C[s], \mathrm{last}(\rho(\tau)))$. By Equations (6), (10) and (14) and Lemma 4.5, we have $\mathbf{Tr}(C[s], \rho(\sigma)) = \mathbf{Tr}(C[\mathtt{skip}; s], \rho(\sigma))$ for any σ. Therefore, for all finite $sh' \in \mathbf{Tr}(C[\mathtt{skip}; s], \mathrm{last}(\rho(\tau)))$ we have $sh' \models [\mathtt{skip}]\,\phi$, and it follows that $\models \psi \Rightarrow \tau\,[\mathtt{skip}; s \text{ @ } C]\,\phi$.

Case Seq2. By the definition of satisfiability for statements in contexts. □

5 SEMANTICS FOR A SHARED-VARIABLE PARALLEL PROGRAMMING LANGUAGE

In this and the following section we show that the LAGC semantics naturally extends to cover advanced language constructs. We gradually extend WHILE with parallel programming constructs: parallel execution, procedure calls, distributed memory, dynamic process creation, and communication between procedures. To cater for interleaved execution of parallel instructions, and to allow for the compositional definition of semantics, the traces of the sequential language are gradually unfolded by means of continuations.

5.1 Local Parallelism

We extend WHILE with a statement for parallel execution, such that the syntax for statements from Figure 2 becomes:

$$s \in Stmt ::= \mathbf{co}\ s \mid\mid s\ \mathbf{oc} \ \mid \ \ldots\ .$$

The semantics of this statement consists in interleaving the evaluation of the two parallel branches at the granularity of atomic statements, as exemplified below.

Example 5.1. The execution of the program $s_{co} = \mathbf{co}\ x := 1;\ y := x + 1 \mid\mid x := 2\ \mathbf{oc}$ produces one of three possible traces that correspond to the traces of the following sequential programs: $x := 1;\ y := x + 1;\ x := 2$, or $x := 1;\ x := 2;\ y := x + 1$, or $x := 2;\ x := 1;\ y := x + 1$.

5.1.1 Local Evaluation. The evaluation rule for the parallel execution statement $\mathbf{co}\ r \mid\mid s\ \mathbf{oc}$ branches on the statement which first gets to execute, resulting in two sets of traces where the first set contains traces with the path condition of r and the continuation of r in parallel with s and the second set contains traces with the path condition of s and the continuation of s in parallel with r. The valuation rule is formalised as follows:

$$\begin{aligned} \mathrm{val}_\sigma(\mathbf{co}\ r \mid\mid s\ \mathbf{oc}) = & \{pc_r \triangleright \tau_r \cdot \mathrm{K}(\mathbf{co}\ r' \mid\mid s\ \mathbf{oc}) \mid pc_r \triangleright \tau_r \cdot \mathrm{K}(r') \in \mathrm{val}_\sigma(r)\} \\ & \cup \{pc_s \triangleright \tau_s \cdot \mathrm{K}(\mathbf{co}\ r \mid\mid s'\ \mathbf{oc}) \mid pc_s \triangleright \tau_s \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma(s)\}\ . \end{aligned} \tag{21}$$

As with sequential composition in rule (10), we need to ensure that empty continuations are not propagated. To this end we define the following rewrite rule for parallel composition inside continuations (observe that the case $s = s' = \emptyset$ cannot occur due to exhaustive applications of the rewrite rule (11)):

$$\mathbf{co}\ s \mid\mid s'\ \mathbf{oc} \rightsquigarrow \begin{cases} s' & \text{if } s = \emptyset,\ s' \neq \emptyset \\ s & \text{if } s \neq \emptyset,\ s' = \emptyset\ . \end{cases} \tag{22}$$

5.1.2 Trace Composition. The composition rule (14) can be kept unchanged (although the abstract variant in rule (28) below can be used as well). Its effect is that all combinations of execution sequences of atomic statements in the parallel branches can be generated. This behaviour corresponds to the classical interleaving semantics (e.g., [9]), as demonstrated below.

Example 5.2. Parallel execution has $\mathtt{skip}$ as a unit of composition; i.e., for any statement s and state σ, we have $\mathbf{Tr}(\mathbf{co}\ \mathtt{skip} \mid\mid s\ \mathbf{oc}, \sigma) = \mathbf{Tr}(\mathbf{co}\ s \mid\mid \mathtt{skip}\ \mathbf{oc}, \sigma) = \mathbf{Tr}(s, \sigma)$.

Example 5.3. Consider program s_{co} from Example 5.1, its evaluation is

$$\begin{aligned}
\mathrm{val}_\sigma(s_{co}) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 1] \cdot \mathrm{K}(\textbf{co}\ y := x+1\ \ ||\ \ x := 2\ \textbf{oc})\} \cup \\
&\quad\ \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 2] \cdot \mathrm{K}(\textbf{co}\ x := 1; y := x+1\ \ ||\ \ \square\ \textbf{oc})\} \\
&= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 1] \cdot \mathrm{K}(\textbf{co}\ y := x+1 \,||\, x := 2\ \textbf{oc})\} \cup \\
&\quad\ \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 2] \cdot \mathrm{K}(x := 1; y := x+1)\}\ ,
\end{aligned}$$

using the following sub-evaluations: $\mathrm{val}_\sigma(x := 1; y := x+1) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 1] \cdot \mathrm{K}(y := x+1)\}$ and $\mathrm{val}_\sigma(x := 2) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 2] \cdot \mathrm{K}(\square)\}$. For trace composition, we first need to evaluate the continuation $\textbf{co}\ y := x+1 \,||\, x := 2\ \textbf{oc}$:

$$\begin{aligned}
\mathrm{val}_\sigma(\textbf{co}\ y := x+1 \,||\, x := 2\ \textbf{oc}) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[y \mapsto \mathrm{val}_\sigma(x+1)] \cdot \mathrm{K}(\textbf{co}\ \square\ \ ||\ \ x := 2\ \textbf{oc})\} \cup \\
&\quad\ \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 2] \cdot \mathrm{K}(\textbf{co}\ y := x+1\ \ ||\ \ \square\ \textbf{oc})\} \\
&= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[y \mapsto \mathrm{val}_\sigma(x+1)] \cdot \mathrm{K}(x := 2)\} \cup \\
&\quad\ \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 2] \cdot \mathrm{K}(y := x+1)\}\ ,
\end{aligned}$$

using the following sub-evaluations: $\mathrm{val}_\sigma(y := x+1) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[y \mapsto \mathrm{val}_\sigma(x+1)] \cdot \mathrm{K}(\square)\}$ and $\mathrm{val}_\sigma(x := 2) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 2] \cdot \mathrm{K}(\square)\}$.

To illustrate trace composition, let us consider the trace where statement $x := 1$ is executed first and explore the remaining possible traces. We start from the state I_{co}. The first application of rule (14) results in the following concrete trace and continuation:

$$\langle I_{co}\rangle \curvearrowright [x \mapsto 1], \mathrm{K}(\textbf{co}\ y := x+1 \,||\, x := 2\ \textbf{oc})\ .$$

At this point, two different instances of the composition rule are applicable, corresponding to the two possible interleavings; i.e., there is a choice between the two continuations in $\mathrm{val}_{[x\mapsto 1]}(\textbf{co}\ y := x+1 \,||\, x := 2\ \textbf{oc})$ as shown above. The first possible instance of the composition rule is:

$$\frac{\begin{array}{c}[x \mapsto 1] = \mathrm{last}(\langle I_{co}\rangle \curvearrowright [x \mapsto 1]) \qquad \emptyset \text{ consistent} \\ \emptyset \triangleright \langle [x \mapsto 1]\rangle \curvearrowright [x \mapsto 1][y \mapsto \mathrm{val}_{[x\mapsto 1]}(x+1)] \cdot \mathrm{K}(x := 2) \\ \in \mathrm{val}_{[x\mapsto 1]}(\textbf{co}\ y := x+1 \,||\, x := 2\ \textbf{oc})\end{array}}{\begin{array}{c}\langle I_{co}\rangle \curvearrowright [x \mapsto 1], \mathrm{K}(\textbf{co}\ y := x+1 \,||\, x := 2\ \textbf{oc}) \to \\ \langle I_{co}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2], \mathrm{K}(x := 2)\end{array}} \tag{23}$$

One further rule application is possible, which results in the final step:

$$\begin{aligned}
&\langle I_{co}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2], \mathrm{K}(x := 2) \to \\
&\langle I_{co}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2] \curvearrowright [x \mapsto 2, y \mapsto 2], \mathrm{K}(\square)\ \ .
\end{aligned}$$

The second possible instance of the composition rule is similar and yields:

$$\begin{aligned}
&\langle I_{co}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 2], \mathrm{K}(y := x+1) \to \\
&\langle I_{co}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 2] \curvearrowright [x \mapsto 2, y \mapsto 3], \mathrm{K}(\square)\ \ .
\end{aligned}$$

The third possible trace is obtained analogously by starting with the second branch. After this, only one continuation is possible. Altogether, the following traces are obtained:

$$\begin{aligned}
\mathbf{Tr}(s_{co}, I_{co}) \quad = \{ \quad &\langle I_{co}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2] \curvearrowright [x \mapsto 2, y \mapsto 2], \\
&\langle I_{co}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 2] \curvearrowright [x \mapsto 2, y \mapsto 3], \\
&\langle I_{co}\rangle \curvearrowright [x \mapsto 2] \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2]\ \}\ .
\end{aligned}$$

$$\textsc{(Par)}\quad \frac{s_1 \neq \texttt{skip} \qquad s_2 \neq \texttt{skip} \qquad \Gamma, pc \Rightarrow \tau\,[s_1 \,@\, C[\texttt{co}\ \bullet\ \|\ s_2\ \texttt{oc}]]\,\phi \qquad \Gamma, pc \Rightarrow \tau\,[s_2 \,@\, C[\texttt{co}\ s_1\ \|\ \bullet\ \texttt{oc}]]\,\phi}{\Gamma, pc \Rightarrow \tau\,[\texttt{co}\ s_1\ \|\ s_2\ \texttt{oc} \,@\, C]\,\phi}$$

$$\textsc{(Par-Skip1)}\quad \frac{\Gamma, pc \Rightarrow \tau\,[C[s]]\,\phi}{\Gamma, pc \Rightarrow \tau\,[\texttt{co skip}\ \|\ s\ \texttt{oc} \,@\, C]\,\phi}$$

$$\textsc{(Par-Skip2)}\quad \frac{\Gamma, pc \Rightarrow \tau\,[C[s]]\,\phi}{\Gamma, pc \Rightarrow \tau\,[\texttt{co}\ s\ \|\ \texttt{skip oc} \,@\, C]\,\phi}$$

Fig. 4. Dynamic logic sequent calculus for local parallelism.

5.1.3 Calculus. We extend our syntax with parallel composition contexts for local parallelism:

$$C \in \mathit{Context} ::= \texttt{co}\ C\ \|\ s\ \texttt{oc} \mid \texttt{co}\ s\ \|\ C\ \texttt{oc} \mid \cdots$$

The corresponding proof rules, given in Figure 4, allow us to recursively unfold parallel compositions by exploiting the focus and context formulation of programs in the proof rules. For example, rule Cond (see Figure 3) pushes the **`if`**-statement s back into the hole, which makes **`if`**-statements non-atomic for parallel contexts and permits interleaving.

Example 5.4. We show how rule Par recursively unfolds parallel composition. Assume that the statements s_1, s_2 and s_3 are not **`skip`**. Applying Par to the sequent

$$\psi \Rightarrow \tau\,[\texttt{co}\ \texttt{co}\ s_1\ \|\ s_2\ \texttt{oc}\ \ \|\ s_3\ \texttt{oc}]\,\phi,$$

we obtain the premises

$$\psi \Rightarrow \tau\,[\texttt{co}\ s_1\ \|\ s_2\ \texttt{oc} \,@\, \texttt{co}\ \bullet\ \ \|\ s_3\ \texttt{oc}]\,\phi$$
$$\psi \Rightarrow \tau\,[s_3 \,@\, \texttt{co}\ \texttt{co}\ s_1\ \|\ s_2\ \texttt{oc}\ \ \|\ \bullet\ \texttt{oc}]\,\phi$$

By applying rule Par again to the first premise, we obtain the premises

$$\psi \Rightarrow \tau\,[s_1 \,@\, \texttt{co}\ \texttt{co}\ \bullet\ \|\ s_2\ \texttt{oc}\ \ \|\ s_3\ \texttt{oc}]\,\phi$$
$$\psi \Rightarrow \tau\,[s_2 \,@\, \texttt{co}\ \texttt{co}\ s_1\ \|\ \bullet\ \texttt{oc}\ \ \|\ s_3\ \texttt{oc}]\,\phi$$

When s_1 is analysed, the "rest" of the statement will be pushed back into the hole, so it can be interleaved with s_2 and s_3.

In contrast to sequential contexts, where we have $\mathbf{Tr}(s \,@\, C, \sigma) = \mathbf{Tr}(C\,[s]\,, \sigma)$ for any state σ, parallel contexts merely ensure $\mathbf{Tr}(s \,@\, C, \sigma) \subseteq \mathbf{Tr}(C\,[s]\,, \sigma)$. However, the premises of Par, Par-Skip1 and Par-Skip2 cover all the traces; e.g., $\mathbf{Tr}(s_1 \,@\, \texttt{co} \bullet \|\ s_2\ \texttt{co}, \sigma) \cup \mathbf{Tr}(s_2 \,@\, \texttt{co}\ s_1\ \|\ \bullet\ \texttt{co}, \sigma) = \mathbf{Tr}(\texttt{co}\ s_1\ \|\ s_2\ \texttt{co}, \sigma)$ for any σ. We need to adapt the notion of contextual trace equivalence and Lemma 4.5 to statements that have the same trace sets modulo interleaving. The soundness argument for the proof rules for local parallelism, relative to the LAGC semantics, is then similar to the proof of Theorem 4.6, but the rules are impractical because they quickly lead to path explosion. Several approaches to mitigate this effect have been proposed in the literature (e.g., [11, 38, 75, 76]).

5.2 Atomic

We extend WHILE with atomic blocks to control interleaving in the execution of parallel statements, such that the syntax for statements from Figure 2 becomes

$$s \in \mathit{Stmt} ::= \texttt{co}\ s\ \|\ s\ \texttt{oc} \mid \texttt{atomic}(st) \mid \ldots\ ,$$

where st is a statement without **`while`** loops (this design choice is discussed below).

5.2.1 Local Evaluation. The **atomic** statement protects its argument against interleaving computations from other branches of a parallel execution operator. A (loop-free) statement st can be made to execute atomically, i.e. without preemption, written $\texttt{atomic}(st)$. Atomic execution requires interleaving points to be *removed* during trace composition.

$$\begin{aligned}\mathrm{val}_\sigma(\texttt{atomic}(st)) = \{&pc_1 \cup pc_2 \triangleright \tau_1 ** \tau_2 \cdot \mathrm{K}(\mathbb{0}) \mid pc_1 \triangleright \tau_1 \cdot \mathrm{K}(st') \in \mathrm{val}_\sigma(st) \wedge st' \neq \mathbb{0} \wedge \\ &pc_2 \triangleright \tau_2 \cdot \mathrm{K}(\mathbb{0}) \in \mathrm{val}_{\sigma'}(\texttt{atomic}(st')) \wedge \sigma' = \mathrm{last}(\tau_1)\} \cup \\ \{&pc \triangleright \tau \cdot \mathrm{K}(\mathbb{0}) \mid pc \triangleright \tau \cdot \mathrm{K}(\mathbb{0}) \in \mathrm{val}_\sigma(st)\}\end{aligned} \tag{24}$$

The main idea behind rule (24) is to recursively unfold the execution of its atomic argument st, while removing continuation markers. During trace composition, this will prevent the atomic code from being interleaved. The rule has two cases, depending on whether the evaluation of st contains a non-empty continuation or not: if the continuation is not empty, we need to evaluate it immediately, as the execution cannot be interrupted before the end of the atomic statement. Note the structural similarity in the definition of the first trace set above and the composition rule (14). The difference is that the consistency check and concretisation are deferred until the actual trace composition.

Discussion. In this execution model, the semantics val_σ of non-terminating atomic statements is *undefined*, which is the reason for excluding loops. This is a design choice and not a principal limitation: It is possible to define the semantics of non-terminating atomic programs with suitable scheduling events that allow to process non-terminating code piece-wise [32]. However, that approach is more complex than the adopted solution and breaks with the modularity we aim at with only one rule per syntactic construct. A different solution is to equip **atomic** with a path condition argument (using pc_1 in the recursive call) and only keep consistent traces. When σ is concrete, this would suffice for terminating loops. Finally, one could define traces co-inductively and use them to specify non-terminating loops. This is attempted in [19], but a highly complex loop invariant rule was obtained already for the sequential WHILE language. The extension of the present setting to a co-inductive semantics and calculus is a major undertaking.

Another solution, presented in Section 7.3 below, is to define a trace composition rule, where atomic execution of *any* statement is the default and can be interrupted only at explicit suspension points.

Observe that the semantics of non-terminating *non-atomic* programs is well-defined: infinite traces are produced by an infinite number of applications of the composition rule.

5.2.2 Trace Composition. The trace composition rule (14) is unchanged, but it is worthwhile to observe how it works in the presence of **atomic**. When rule (14) processes a continuation of the form $\texttt{atomic}(st)$, it needs to evaluate $\mathrm{val}_\sigma(\texttt{atomic}(st))$ for the concrete state $\sigma = \mathrm{last}(sh)$. Rule (24) evaluates s until it reaches a continuation s', then recursively evaluates s', until s is completely executed. No interleaving can occur during the evaluation, which reflects the intended semantics of atomic statements.

Example 5.5. To illustrate the evaluation of interleaved execution with **atomic**, we modify Example 5.3 as follows: let $s_{at} = \texttt{co}\ \texttt{atomic}(x := 1; y := x + 1) \mid\mid x := 2\ \texttt{oc}$. The evaluation branches into either $\texttt{atomic}(x := 1; y := x + 1)$ and then $x := 2$, or $x := 2$ and then $\texttt{atomic}(x := 1; y := x + 1)$. In either case, the final value of y is 2, as shown in the following evaluation:

$$\begin{aligned}\mathrm{val}_\sigma(s_{at}) =&\{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 1][y \mapsto \mathrm{val}_{\sigma[x\mapsto 1]}(x+1)] \cdot \mathrm{K}(x := 2)\} \cup \\ &\{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto 2] \cdot \mathrm{K}(\texttt{atomic}(x := 1; y := x + 1))\}\ .\end{aligned}$$

Now the trace in which y has the final value 3, is no longer produced; starting from state $I_{at} = I_{co}$ we have $\mathbf{Tr}(s_{at}, I_{at}) = \{\langle I_{at}\rangle \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2] \curvearrowright [x \mapsto 2, y \mapsto 2],\ \langle I_{at}\rangle \curvearrowright [x \mapsto 2] \curvearrowright [x \mapsto 1] \curvearrowright [x \mapsto 1, y \mapsto 2]\}$.

5.2.3 Calculus. The rule for `atomic` is remarkably simple, because of the explicit contexts. Rule Atomic keeps the entire atomic statement s in focus, but changes from an outermost context which may be parallel to a sequential context, thereby enforcing sequential execution of the atomic statement.

$$\text{(Atomic)}\quad \frac{\Gamma, pc \Rightarrow \tau\,[s \mathbin{@} \bullet; C[\mathtt{skip}]]\,\phi}{\Gamma, pc \Rightarrow \tau\,[\mathtt{atomic}(s) \mathbin{@} C]\,\phi}$$

5.3 Local Memory

We extend WHILE with local variable declarations by introducing a syntactic category of variable declarations *VarDecl* and add blocks to the syntax for statements from Figure 2:

$$\begin{aligned} s \in \mathit{Stmt} &::= \{\ d\ s\ \} \mid \ldots \\ d \in \mathit{VarDecl} &::= \varepsilon \mid x;\ d \end{aligned}$$

For simplicity, we omit an empty list of variable declarations in a scope $\{\ \varepsilon\ s\ \}$ and just write $\{\ s\ \}$.

5.3.1 Local Evaluation. A block $\{\ d\ s\ \}$ introduces a local variable scope such that the program variables d should only be accessible for the statement s. To avoid interference among variable declarations in different scopes, possibly introducing the same variable name, the evaluation rule renames the variables in d to unused names and adds the renamed program variables to the state. The local program variables are correspondingly renamed in s. The local evaluation rules cover a non-empty and empty list of local variable declarations, respectively.

$$\mathrm{val}_\sigma(\{\ x; d\ s\ \}) = \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x' \mapsto 0] \cdot \mathrm{K}(\{\ d\ s[x \leftarrow x']\ \}) \mid x' \notin \mathrm{dom}(\sigma)\} \tag{25}$$

$$\mathrm{val}_\sigma(\{\ s\ \}) = \mathrm{val}_\sigma(s) \tag{26}$$

Here, $s[x \leftarrow x']$ denotes the textual substitution of program variable x by x' in the statement s.

5.3.2 Trace Composition. Trace composition in rule (14) is unchanged. It will gradually extend the state with fresh variables, bound to the default value 0, until the local variable declarations of the scope have been reduced to an empty list, at which point the scope is removed and execution can continue as normal.

5.3.3 Calculus. The rules are straightforward. We model initialisation by an assignment to the default value, for which a rule already exists. To avoid name clashes for multiple local variables with the same variable names, these are replaced by fresh names, mimicking the LAGC semantics. The occurrences of local variables in the post-condition are similarly renamed (which suggests how the calculus could be extended with method contracts by adding an additional conjunct to the postcondition ϕ' in the premise).

$$\text{(local}_1\text{)}\quad \frac{x' \text{ fresh} \qquad s' = s[x \leftarrow x'] \qquad \phi' = \phi[x/x'] \qquad \Gamma, pc \Rightarrow \tau\,[x' := 0 \mathbin{@} C\,[\bullet; \{d\ s'\}]]\,\phi'}{\Gamma, pc \Rightarrow \tau\,[\{x; d\ s\} \mathbin{@} C]\,\phi}$$

$$\text{(local}_2\text{)}\quad \frac{\Gamma, pc \Rightarrow \tau\,[C\,[s]]\,\phi}{\Gamma, pc \Rightarrow \tau\,[\{s\} \mathbin{@} C]\,\phi}$$

$$\begin{aligned} P \in \mathit{Prog} &::= \overline{M}\,\{s\} \\ M \in \mathit{Meth} &::= m(x)\{s\} \\ s \in \mathit{Stmt} &::= \mathbf{skip} \mid x := e \mid \mathbf{if}\ e\ \{\ s\ \} \mid s;s \mid \mathbf{while}\ e\ \{\ s\ \} \\ &\quad\ \mid \mathbf{co}\ s \mid\mid s\ \mathbf{oc} \mid \mathbf{atomic}(s) \mid \mathbf{call}(m,e) \end{aligned}$$

Fig. 5. Program Syntax with Procedure Calls.

5.4 Introducing Events and Symbolic Traces

Symbolic traces can contain states in which variables may be bound to unknown values. A simple case where this become necessary is when WHILE is extended with a statement to express user input, such that the syntax for statements from Figure 2 becomes

$$s \in \mathit{Stmt} ::= \mathbf{input}(x) \mid \ \ldots\ .$$

For the **input**-statement, Proposition 3.2 does not hold, because we do not know the value of the user input x when locally evaluating the statement. To represent these unknown values in the local semantics, we use symbolic variables introduced in Definition 2.3. Events can be used to introduce a point of interaction; we define an event type $\mathit{inpEv}(Y)$ that captures the introduction of a symbolic variable Y by an **input**-statement. The local evaluation rule for $\mathbf{input}(x)$ can be formalised as follows:

$$\mathrm{val}_\sigma(\mathbf{input}(x)) = \{\emptyset \triangleright \mathit{inpEv}_\sigma^{\{Y\}}(Y) \curvearrowright \sigma[Y \mapsto *][\,x \mapsto Y] \cdot \mathrm{K}(\lozenge) \mid Y \notin \mathrm{dom}(\sigma)\}\ . \quad (27)$$

The trace is well-formed, because the variable Y inside the event is symbolic.

The composition rule accommodates Proposition 3.2 by using a concretisation mapping to concretise the emitted trace. Concretisation captures the actual user input when the statement is executed in a concrete trace.

$$\frac{\sigma = \mathrm{last}(sh) \quad pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma(s) \quad \rho \text{ concretises } \tau \quad \rho(pc) \text{ consistent}}{sh, \mathrm{K}(s) \to sh \ast\ast \rho(\tau), \mathrm{K}(s')} \quad (28)$$

The concretisation mapping ρ ensures that the resulting trace $sh \ast\ast \rho(\tau)$ is concrete.

The previous composition rule (14) is a special case of this one, when all the states are concrete (this follows from Proposition 2.26).

Example 5.6. $\mathbf{Tr}(\mathbf{input}(x), I_{seq}) = \{\langle I_{seq}\rangle \ast\ast \mathit{inpEv}_\sigma^{\{Y\}}(Y) \curvearrowright \sigma[Y \mapsto v][\,x \mapsto v] \mid v \in \mathit{Val}\} = \{\langle I_{seq}\rangle \curvearrowright \mathit{inpEv}(v) \curvearrowright I_{seq}[Y \mapsto v] \curvearrowright I_{seq}[Y \mapsto v,\ x \mapsto v] \mid v \in \mathit{Val}\}$. All possible values v may occur as an input. Variable Y has a concrete value now; x, Y are needed to propagate the input value correctly, for example, in subsequent code of the form "**if** $(x > 0)$ **then** ...".

Calculus. The calculus rule directly follows the local evaluation rule (27) by introducing a fresh variable Y to model the unknown user input. By definition we have $\sigma_*(Y) = *$, so we do not need this assignment.

$$\textsc{(Input)}\quad \frac{\mathrm{last}(\tau) = \sigma \qquad Y \text{ fresh} \qquad \Gamma,\ pc \Rightarrow \tau \ast\ast \mathit{inpEv}_\sigma^{\{Y\}}(Y)\ [x := Y\ @\ C]\ \phi}{\Gamma,\ pc \Rightarrow \tau\ [\mathbf{input}(x)\ @\ C]\ \phi}$$

Interestingly, the abstract composition rule (28) has no effect on the calculus, which is already symbolic, but the soundness proof needs to be adapted.

5.5 Procedure Calls

Figure 5 introduces procedure[6] calls with the statement $\texttt{call}(m, e)$: A program is a set of method declarations $\overline{M}$ with main block $\{s\}$. We consider $\overline{M}$ to be a *method table* that is implicitly provided with a program. Each method declaration associates a unique name m with the statement in its body. To reduce technicalities, we assume without loss of generality that a method call has one argument and no return value on which the caller needs to synchronise.[7]

In contrast to parallel $\texttt{co}\ s \mid\mid s\ \texttt{oc}$ statements, where parallelism is explicit in the program syntax, procedure calls introduce implicit parallel execution: there is a context switch between the syntactic call site and the processor, whereupon the call is executed in parallel. This decoupling has an important consequence: according to the local evaluation principle, method bodies are evaluated independently of the call context. This necessitates a new composition rule that starts the execution of a called method. We also need to ensure that a method is not executed before it is called, requiring the introduction of suitable *events*.

5.5.1 Local Evaluation. The new events are $invEv(m, \mathrm{val}_\sigma(e))$ and $invREv(m, \mathrm{val}_\sigma(e))$ for a given state σ. These events denote the *invocation* and the *activation* (also called the *invocation reaction*) of a method m with argument e, respectively. Recall from Section 2.3 that events inserted into a trace at a state σ are preceded and succeeded by that state. The local evaluation rule (29) for a call to m with argument e has an empty path condition, inserts an *invocation* event $invEv(m, \mathrm{val}_\sigma(e))$ into the trace, and continues with the empty continuation. This means that the call is *non-blocking*: the code following the call could be executed immediately, if scheduled. The rule is formalised as follows:

$$\mathrm{val}_\sigma(\texttt{call}(m, e)) = \{\emptyset \triangleright invEv_\sigma(m, \mathrm{val}_\sigma(e)) \cdot \mathrm{K}(\llbracket\rrbracket)\}\ . \tag{29}$$

5.5.2 Trace Composition: Concrete Variant. The concurrency model so far is still very simple: method invocations and processors are anonymous, there is no way to distinguish between two calls to the same method: these execute identical code and all executions can be interleaved. For this reason, it is sufficient to represent continuation candidates for the next execution step as a multiset q. We denote with "+" the *disjoint* multiset union. We add a rewrite rule to simplify empty continuations in the multisets of tasks; this rule is applied exhaustively when adding tasks in multisets.

$$q+\{\mathrm{K}(\llbracket\rrbracket)\} \rightsquigarrow q.$$

The judgment representing trace extension by one step has the form: $sh,\ q \longrightarrow sh',\ q'$ where q, q' are multisets of continuations of the form $\mathrm{K}(s)$.

$$\frac{\sigma = \mathrm{last}(sh) \quad pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma(s) \quad pc \text{ consistent}}{sh, q+\{\mathrm{K}(s)\} \longrightarrow sh ** \tau, q+\{\mathrm{K}(s')\}} \tag{30}$$

The difference between rule (14) and rule (30) is that we no longer commit to one possible continuation, because the continuation code is now located in a method. Therefore, the rule selects and removes one matching continuation from the pool q, executes it to the next continuation marker, exactly like rule (14), and then puts the code remaining to be executed back into q. We also need a new composition rule that adds method bodies to the pool q.

$$\frac{m(x)\{s\} \in \overline{M} \quad \sigma = \mathrm{last}(sh) \quad wf(sh ** invREv_\sigma(m, v)) \quad y \notin \mathrm{dom}(\sigma)}{sh, q \longrightarrow sh ** invREv_\sigma(m, v) \curvearrowright \sigma[y \mapsto v], q+\{\mathrm{K}(s[x \leftarrow y])\}} \tag{31}$$

[6]Historically, it is more common to use the term *procedure*, whereas in object-oriented programming the term *method* is usual, which is where the LAGC concept originated from. We use both terms interchangeably.
[7]Synchronisation on return values is discussed in Section 7.3.

The rule handles the creation of a new execution thread. We select a method with body s from table $\overline{M}$ and create a new continuation $\mathrm{K}(s[x \leftarrow y])$, where the call parameter x is substituted with a fresh variable y for disambiguation. The new continuation is added to the pool q. Next, we need to record that a new method with argument v has started to execute, as a consequence of a previous method call. The existence of this call is ensured by the premise expressing the well-formedness of the extended trace, which requires an invocation event of the form $invEv(m, v)$ to be present in sh. We extend the current trace with the *invocation reaction* event[8] $invREv_\sigma(m, v)$ to mark the start of method execution and record the extension of the current state in which the parameter y has the value v. It remains to formalise well-formedness.

5.5.3 Well-Formedness.

A trace is well-formed if its events obey certain ordering restrictions. Whenever a trace sh is extended with an invocation reaction event of the form $invREv(m, v)$, there must be a corresponding invocation event $invEv(m, v)$ in sh. This ordering restriction can be captured by counting the number of occurrences of both event forms in sh using the comprehension expression $\#_{sh}(ev(\overline{e}))$. In all other cases, the trace stays well-formed when it is extended with a new event. We formalise well-formedness as a predicate on traces.

Definition 5.7 (Well-Formedness). The *well-formedness* of a concrete trace sh is formalised by a predicate $wf(sh)$, defined inductively over the length of sh:

$$\begin{aligned}
wf(\varepsilon) &= true \\
wf(sh \curvearrowright \sigma) &= wf(sh) \\
wf(sh \curvearrowright invEv(m, v)) &= wf(sh) \\
wf(sh \curvearrowright invREv(m, v)) &= wf(sh) \wedge \#_{sh}(invEv(m, v)) > \#_{sh}(invREv(m, v))
\end{aligned}$$

The well-formedness predicate is used in the global composition rules to ensure that only a valid concrete trace for a given program can be generated during the trace composition. Any trace emitted by the trace composition rules is well-formed, because it is an invariant established each time a trace is extended. It is sufficient to define well-formedness on concrete traces, because in the end it has to hold for them. It follows that well-formedness is always a simple, decidable property.

5.5.4 Global Trace Semantics.

Definition 5.8 (Program Semantics with Procedure Calls). Given a finite program P with a method table $\overline{M}$ and a main block s_{main}. Let

$$sh_0,\ q_0 \to sh_1,\ q_1 \to \cdots$$

be a maximal sequence obtained by the repeated application of the composition rules (30)–(31), starting with $\langle I_P \rangle,\ \{\mathrm{K}(s_{main})\}$. If the sequence is finite, then it must have the form

$$\langle I_P \rangle, \{\mathrm{K}(s_{main})\} \to \cdots \to \underline{sh}, \emptyset \ .$$

If the sequence is infinite[9], let $\underline{sh} = \lim_{i \to \infty} sh_i$. The set of all such potentially infinite traces $\underline{sh}$ is denoted by $\mathbf{Tr}(P, I_P)$.

Example 5.9. Consider the following program P:

$$\begin{gathered}
m(x)\ \{y := x;\ x := x + 1\} \\
\{\mathtt{call}(m, 1);\ z := 2\}
\end{gathered}$$

[8]Recall that $invREv_\sigma(m, v)$ is a triple with two copies of σ around $invREv(m, v)$.

[9]Like in the previous cases, the limit is well-defined as traces are always growing along the sequence of concrete traces.

The method table for this program is $\overline{M} = \{m(x)\ \{y := x;\ x := x + 1\}\}$. The evaluation of the main method is as follows:

$$\begin{aligned} \mathrm{val}_\sigma(\mathbf{call}(m,1)) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright invEv(m,1) \curvearrowright \sigma \cdot \mathrm{K}(\lozenge)\} \\ \mathrm{val}_\sigma(z := 2) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[z \mapsto 2] \cdot \mathrm{K}(\lozenge)\} \\ \mathrm{val}_\sigma(\mathbf{call}(m,1); z := 2) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright invEv(m,1) \curvearrowright \sigma \cdot \mathrm{K}(z := 2)\} \end{aligned}$$

To prepare the evaluation of the method body of $m(x)$ (where x is any integer):

$$\begin{aligned} \mathrm{val}_\sigma(y := x) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[y \mapsto \mathrm{val}_\sigma(x)] \cdot \mathrm{K}(\lozenge)\} \\ \mathrm{val}_\sigma(x := x+1) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[x \mapsto \mathrm{val}_\sigma(x+1)] \cdot \mathrm{K}(\lozenge)\} \\ \mathrm{val}_\sigma(y := x; x := x+1) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[y \mapsto \mathrm{val}_\sigma(x)] \cdot \mathrm{K}(x := x+1)\} \end{aligned} \tag{32}$$

Let I_P be the initial state of P. We consider the state where the statement $\mathbf{call}(m, 1)$ in main has already been executed, and explore the possible traces. The concrete trace and the continuation pool at this point are $sh = \langle I_P\rangle \curvearrowright invEv(m,1) \curvearrowright I_P$ and $q = \{\mathrm{K}(z := 2)\}$, respectively. Let s_m be the method body of m. Both composition rules (30) and (31) are applicable and yield different interleavings: the execution continues either with the main method or with the creation of a new execution thread. In the following, we assume that the execution continues with thread creation, i.e. we apply rule (31):

$$\frac{m(x)\{s_m\} \in \overline{M} \quad I_P = \mathrm{last}(sh) \quad wf(sh ** invREv_{I_P}(m,1)) \quad w \notin \mathrm{dom}(I_P)}{sh, q \longrightarrow sh ** invREv_{I_P}(m,1) \curvearrowright [w \mapsto 1], q + \{\mathrm{K}(s_m[x \leftarrow w])\}} \tag{33}$$

At this point, both composition rules are again applicable, but only rule (30) is useful: since only one invocation event is present in sh, well-formedness will not allow more than one instance of m to execute. We use some abbreviations:

$$\begin{aligned} sh' &= sh ** invREv_{I_P}(m,1) \curvearrowright [w \mapsto 1] \\ &= \langle I_P\rangle \curvearrowright invEv(m,1) \curvearrowright I_P \curvearrowright invREv(m,1) \curvearrowright I_P \curvearrowright [w \mapsto 1] \\ s'_m &= \{s_m[x \leftarrow w]\} = \{y := w;\ w := w+1\} \\ q' &= q + \{\mathrm{K}(s_m[x \leftarrow w])\} = \{\mathrm{K}(z := 2), \mathrm{K}(y := w;\ w := w+1)\} \end{aligned}$$

The two continuations in q' indicate two possible interleavings. We assume execution continues with the body of m:

$$\frac{\begin{array}{c}[w \mapsto 1] = \mathrm{last}(sh') \qquad \emptyset \text{ consistent} \\ \emptyset \triangleright \langle[w \mapsto 1]\rangle \curvearrowright [w \mapsto 1][y \mapsto \mathrm{val}_{[w\mapsto 1]}(w)] \cdot \mathrm{K}(w := w+1) \in \mathrm{val}_{[w\mapsto 1]}(s'_m)\end{array}}{\begin{array}{c}sh', \{\mathrm{K}(z := 2)\} + \{\mathrm{K}(y := w; w := w+1)\} \longrightarrow \\ sh' ** \langle[w \mapsto 1]\rangle \curvearrowright [w \mapsto 1, y \mapsto 1], \{\mathrm{K}(z := 2)\} + \{\mathrm{K}(w := w+1)\}\end{array}} \tag{34}$$

where $\mathrm{val}_{[w\mapsto 1]}(s'_m)$ is taken from Equation (32). We introduce the abbreviations:

$$\begin{aligned} sh'' &= \langle I_P\rangle \curvearrowright invEv(m,1) \curvearrowright I_P \curvearrowright invREv(m,1) \curvearrowright I_P \curvearrowright [w \mapsto 1] \curvearrowright [w \mapsto 1, y \mapsto 1] \\ q'' &= \{\mathrm{K}(z := 2)\} + \{\mathrm{K}(w := w+1)\} \end{aligned}$$

At this point, rule (30) is again applicable and the two continuations in q'' indicate two possible interleavings. We assume the execution continues with the main method body.

$$\frac{\begin{array}{c}[w \mapsto 1, y \mapsto 1] = \mathrm{last}(sh'') \qquad \emptyset \text{ consistent} \\ \{\emptyset \triangleright \langle[w \mapsto 1, y \mapsto 1]\rangle \curvearrowright [w \mapsto 1, y \mapsto 1, z \mapsto 2] \cdot \mathrm{K}(\lozenge)\} \in \mathrm{val}_{[w\mapsto 1, y\mapsto 1]}(z := 2)\end{array}}{sh'', q'' \longrightarrow sh'' ** \langle[w \mapsto 1, y \mapsto 1]\rangle \curvearrowright [w \mapsto 1, y \mapsto 1, z \mapsto 2], \{\mathrm{K}(w := w+1)\}} \tag{35}$$

At this point only one continuation is possible, which results in the concrete trace:

$$\langle I_P\rangle \curvearrowright \mathit{invEv}(m,1) \curvearrowright I_P \curvearrowright \mathit{invREv}(m,1) \curvearrowright I_P \curvearrowright [w \mapsto 1] \\ \curvearrowright [w \mapsto 1, y \mapsto 1] \curvearrowright [w \mapsto 1, y \mapsto 1, z \mapsto 2] \curvearrowright [w \mapsto 2, y \mapsto 1, z \mapsto 2]\ .$$

There are two other possible interleavings that all result in the same final state. If, for example, the assignment in the main method of P were changed to $y := 2$, then traces with different final states would be generated. □

5.5.5 Trace Composition: Symbolic Variant. The trace composition in rule (31) works "eagerly" or "on demand" in the sense that a concrete trace extending σ and a continuation containing the local parameter y, whose value is fixed in $\sigma[y \mapsto v]$, are generated on the spot. In the presence of multiple calls to the same method, this leads to multiple evaluation of the same code, i.e. the method body. A deductive verification calculus would avoid this and evaluate each method only once, but symbolically [4]. We demonstrate that abstract traces allow semantic evaluation that works similarly by defining a local evaluation function for *methods* which combines the use of symbolic variables from the **input**-statement in Section 5.4 with the variable renaming used for local memory in Section 5.3. We first introduce a symbolic variable Y to act as a placeholder for the value of the substituted call parameter y and then substitute the call parameter x with a fresh variable y for disambiguation, as done in rule (31):

$$\mathrm{val}_\sigma(m(x)\{s\}) = \\ \{\emptyset \triangleright \mathit{invREv}_\sigma^{\{Y\}}(m,Y) \curvearrowright \sigma[Y \mapsto *,\ y \mapsto Y] \cdot \mathrm{K}(s[x \leftarrow y]) \mid y, Y \notin \mathrm{dom}(\sigma)\}\ . \tag{36}$$

We conventionally write $y, Y \notin \mathrm{dom}(\sigma)$ to express that no element in a list of variables is in the domain of σ.

The concrete composition rule (31) needs to be adapted to a composition rule which takes the evaluation of a method declaration and concretises the resulting symbolic trace in a well-formed and consistent way:

$$\frac{\begin{array}{ccc} m(x)\{s\} \in \overline{M} & \rho \text{ concretises } \tau & \rho(pc) \text{ consistent} \\ \sigma = \mathrm{last}(sh) & pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma(m(x)\{s\}) & \mathit{wf}(sh ** \rho(\tau)) \end{array}}{sh,\ q \longrightarrow sh ** \rho(\tau),\ q{+}\{\mathrm{K}(s')\}} \tag{37}$$

The definition of well-formedness stays the same. Definition 5.8 of trace semantics of P stays the same, except that rule (37) is used instead of rule (31). It is interesting to note that, while we need one more local rule (to evaluate method declarations), the composition rule becomes simpler, more uniform with rule (30), and more modular (not referring to events) as a consequence.

Example 5.10. Consider the program P from Example 5.9. Let s_m be the method body of m. The evaluation of the method declaration with rule (36) is as follows:

$$\begin{aligned} &\mathrm{val}_\sigma(m(x)\{s_m\}) \\ &\quad = \{\emptyset \triangleright \mathit{invREv}_\sigma^{\{W\}}(m,W) \curvearrowright \sigma[W \mapsto *, w \mapsto W] \cdot \mathrm{K}(s_m[x \leftarrow w]) \mid w, W \notin \mathrm{dom}(\sigma)\} \\ &\quad = \{\emptyset \triangleright \mathit{invREv}_\sigma^{\{W\}}(m,W) \curvearrowright \sigma[W \mapsto *, w \mapsto W] \cdot \mathrm{K}(y := w;\ w := w+1)\ \mid w, W \notin \mathrm{dom}(\sigma)\}\ . \end{aligned}$$

5.5.6 Calculus. We need to design proof rules modeling the semantic rules (29) and (36)–(37). In contrast to the semantic rules, the calculus must ensure that all possible interleavings are analysed; i.e., the new procedure must be able to start executing immediately. This effect can be captured in a single proof rule MtdCall, letting the body of the procedure run in parallel with the continuation of the calling statement. The new procedure m is introduced as a branch in a parallel statement by wrapping the body of m into a block and declaring its formal parameter x as a local variable. This allows us to re-use the previously declared rules; in particular, we use the parallel operator to explore all interleavings of the execution of the new procedure activation with the existing ones. Note that the extended trace τ' is obviously well-formed (assuming that τ was well-formed), because the invocation reaction event occurs after the corresponding invocation.

$$\frac{\text{last}(\tau) = \sigma \qquad m(x)\{s'\} \in \overline{M} \qquad \tau' = \tau ** \mathit{invEv}_\sigma(m, \text{val}_\sigma(e)) ** \mathit{invREv}_\sigma(m, \text{val}_\sigma(e)) \\ \Gamma, pc \Rightarrow \tau' \; [\textbf{co}\; C\, [\textbf{skip}] \;||\; \{x;\; x := e;\; s'\}\; \textbf{oc}]\; \phi}{\Gamma, pc \Rightarrow \tau \; [\textbf{call}(m, e) \;@\; C]\; \phi}\;(\textsc{MtdCall})$$

For simplicity we do not introduce separate postconditions for different procedure calls. This would require variable renaming in postconditions corresponding to the renaming in rule $\textsc{Local}_1$ of our calculus. Such postconditions would be more natural in a setting with trace predicates [20] rather than the state predicates considered in the present paper.

5.6 Guarded Statements

Guarded statements are a common synchronisation mechanism in concurrent languages [29, 53, 58]. A statement is preceded by a Boolean guard expression that blocks execution of the current process, until the guard is evaluated to true. This can be used, for example, to ensure that the result of a computation is ready before it is used, a message has arrived, etc. Our syntax for guarded commands is inspired by ProMeLa; the syntax for statements from Figure 2 becomes (where g is a Boolean expression):

$$s \in \mathit{Stmt} ::= ::\, g; s \mid \;\ldots\;.$$

5.6.1 Local Evaluation. A local evaluation rule for a guarded statement is straightforward to design. If the Boolean guard g evaluates to true, the execution continues normally. If the guard g evaluates to false, the execution is blocked until g evaluates to true. Blocking is modeled by re-scheduling the entire guarded statement in the continuation.

$$\text{val}_\sigma(::\, g; s) = \{\{\text{val}_\sigma(g) = \text{tt}\} \triangleright \langle\sigma\rangle \cdot \text{K}(s)\},\quad \{\text{val}_\sigma(g) = \text{ff}\} \triangleright \langle\sigma\rangle \cdot \text{K}(::\, g; s)\}\} \tag{38}$$

5.6.2 Trace Composition. The trace composition rule (14) is unchanged, but it is interesting to observe how it works in the presence of $::\, g; s$. When rule (14) processes a continuation with a guarded statement, it may result in a *deadlocked* trace due to the second alternative, where from some point onward no progress is made on some non-empty continuations in the process pool. How to *detect* and, possibly, to *avoid* deadlocks is a question outside the scope of this paper.

5.6.3 Calculus. Structurally, the local rule (38) for guarded statements is very similar to the rule (8) for the conditional (in fact, the case when the guard is true, is actually identical). Indeed, we can design a calculus rule modeled after Cond (cf. Figure 3):

$$\frac{\text{last}(\tau) = \sigma \\ \Gamma, pc \cup \{\text{val}_\sigma(g) = \text{tt}\} \Rightarrow \tau\, [C\, [s]]\, \phi \qquad \Gamma, pc \cup \{\text{val}_\sigma(g) = \text{ff}\} \Rightarrow \tau\, [C\, [::\, g; s]]\, \phi}{\Gamma, pc \Rightarrow \tau\, [::\, g; s \;@\; C]\, \phi}\;(\textsc{GrdStmt})$$

While this rule is certainly sound, it is clearly incomplete, because it admits infinitely many possible interleavings. To achieve completeness, one would need to establish a suitable invariant that holds whenever the guard is evaluated and that is strong enough to render the proof finite (for example, by excluding a branch when the guard is false). For details on such techniques, we refer the reader to the literature [35, 63].

6 SEMANTICS FOR A SHARED-MEMORY MULTIPROCESSOR LANGUAGE

We consider a multiprocessor extension to the programming language of Section 5. This goes some way to realize a concurrent, object-oriented setting: We will now spawn processes, each with its own identity and its own task queue to execute, but memory is still shared among the processes. In addition, processes may exchange values by sending and receiving messages, which have their own identity and thus provide the capability to impose ordering constraints among them. Tasks are method invocations similar to the ones defined in the previous section. In particular, we use the second variant of the semantics, defined in Section 5.5.5 with the rules (36)–(37).

Expressions on the right-hand side of assignments are extended with $\mathbf{spawn}(m, e)$, which creates a new (virtual) process where task m is called with an actual value e. It returns the identifier of the newly created process. Furthermore, two statements $\mathbf{send}(e_1, e_2)$, to send a value e_1 to a process with identifier e_2, and $\mathbf{receive}(x, e)$, to bind a value received from e to the variable x, are introduced such that processes may exchange values. The syntax is shown in Figure 6.

$$
\begin{aligned}
P \in \mathit{prog} &::= \overline{M}\{s\}\\
M \in \mathit{Meth} &::= m(x)\{s\}\\
s \in \mathit{Stmt} &::= \mathbf{skip} \mid x := \mathit{rhs} \mid \mathbf{if}\ e\ \{\ s\ \} \mid s; s \mid \mathbf{while}\ e\ \{\ s\ \} \mid \mathbf{co}\ s \mid\mid s\ \mathbf{oc}\\
&\quad\ \mid \mathbf{atomic}(st) \mid \mathbf{call}(m, e) \mid \mathbf{send}(e, e) \mid \mathbf{receive}(x, e)\\
\mathit{rhs} \in \mathit{Rhs} &::= e \mid \mathbf{spawn}(m, e)
\end{aligned}
$$

Fig. 6. Syntax for the Shared-Memory Multiprocessor Language.

6.1 LAGC Semantics

In a multiprocessor setting it is necessary to keep track of what each process does, in particular, about the origins and destinations of messages. This is achieved by introducing process identifiers and tagging each event in a trace with the identifier of the process that produced it. Remark that this tagging is orthogonal to the local semantic evaluation.

Definition 6.1 (Tagged Trace, Projection). Let $ev(\overline{e})$ be an event, τ, τ_1, τ_2 traces, and $p \in \mathit{PId}$ a process identifier. A *tagged trace* τ^p is defined inductively as follows:

$$
\begin{aligned}
&(ev(\overline{e}))^p = ev^p(\overline{e})\\
&\sigma^p = \sigma\\
&(\tau \curvearrowright t)^p = \tau^p \curvearrowright t^p
\end{aligned}
$$

6.1.1 Local Evaluation.

In the following, let *PId* be a set of process identifiers with typical element p and *MId* a set of message identifiers with typical element i. We start with the evaluation rule for **send**. The rule evaluates the arguments to **send** and creates a trace from the current state σ with an event $\mathit{sendEv}(v, p, i)$ expressing that a value v is sent to the process p by a message with the identifier i. The message can have any possible identifier, so the rule provides traces for all of them:

$$\mathrm{val}_\sigma(\mathbf{send}(e, e')) = \{\emptyset \triangleright \mathit{sendEv}_\sigma(\mathrm{val}_\sigma(e), \mathrm{val}_\sigma(e'), i) \cdot \mathrm{K}(\llbracket\rrbracket)\} \mid i \in \mathit{MId}\} \ . \tag{39}$$

The local semantics of **receive** and **spawn** is more complex, because the values received in these statements are not known in the local evaluation. We employ the technique using symbolic variables introduced in Section 5.4. In contrast to the case for method calls (Section 5.5.2), we do not have the option to compose concrete continuations “on demand”, because the return values for **receive** and **spawn** are going to be resolved later.[10] The rules evaluate to a trace with the

[10]It would be possible to introduce symbolic values for message identifiers as well in the rule for **send** and **receive**. However, since programs do not manipulate message identifiers, there is no reason to do so.

event *receiveEv*(v, p, i), expressing that the value v is received from a process p by a message with identifier i, and a trace with the event *spawnEv*(m, v, p), expressing that a new process with identifier p is created to execute the task m with the parameter value v. In each rule, the symbolic variable Y represents the (as yet) unknown received value.

$$\begin{aligned}\mathrm{val}_\sigma(\mathbf{receive}(x, e)) = \{&\emptyset \triangleright \mathit{receiveEv}_\sigma^{\{Y\}}(Y, \mathrm{val}_\sigma(e), i) \curvearrowright \sigma[x \mapsto Y,\ Y \mapsto *] \cdot \mathrm{K}(\mathbb{0}) \\ &\mid Y \notin \mathrm{dom}(\sigma),\ i \in \mathit{MId}\}\end{aligned} \tag{40}$$

$$\begin{aligned}\mathrm{val}_\sigma(x := \mathbf{spawn}(m, e)) = \{&\emptyset \triangleright \mathit{spawnEv}_\sigma^{\{Y\}}(m, \mathrm{val}_\sigma(e), Y) \curvearrowright \sigma[x \mapsto Y,\ Y \mapsto *] \cdot \mathrm{K}(\mathbb{0}) \\ &\mid Y \notin \mathrm{dom}(\sigma)\}\end{aligned} \tag{41}$$

All previous local rules are unchanged.

6.1.2 Trace Composition. The trace composition rules generalize rules (30) and (37). There are three main differences: First, *both* composition rules now work on symbolic traces, simply because rules (40)–(41) are symbolic. Second, to account for having several processes, we introduce a mapping Ω from process identifiers p to multisets of continuations q, i.e. $\Omega(p)$ gives the current task list for a given process. As before, we denote with + the *disjoint* multiset union on task lists. The judgment representing trace extension by one step has the form: $sh,\ \Omega \to sh',\ \Omega'$. Third, we tag the generated concrete trace with the process identifier that created it.

The following rule selects a processor p with non-empty task list and from there a continuation s, evaluated in $\mathrm{last}(sh)$. The resulting symbolic trace τ must be concretisable such that, after tagging, it extends sh in a well-formed manner. Observe that sh is tagged already. The remaining continuation $\mathrm{K}(s')$ is added back to p's task list.

$$\frac{\begin{array}{c}p \in \mathrm{dom}(\Omega) \qquad \Omega(p) = q+\{\mathrm{K}(s)\} \qquad \sigma = \mathrm{last}(sh) \\ pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma(s) \qquad \rho \text{ concretizes } \tau \qquad \rho(pc) \text{ consistent} \qquad wf(sh ** \rho(\tau)^p)\end{array}}{sh, \Omega \to sh ** \rho(\tau)^p, \Omega[p \mapsto q+\{\mathrm{K}(s')\}]} \tag{42}$$

The following rule is similar and follows the pattern established in rule (37). It starts to evaluate a method body on a processor p and adds the remaining continuation $\mathrm{K}(s')$ to its task list. If $p \notin \mathrm{dom}(\Omega)$, we use the notational convention $\Omega(p) = \emptyset$ and avoid a special operation to create a process. The following rule thus either creates a task inside an existing process or spawns a new process if necessary, similar to rule (37). Well-formedness ensures that the correct number of tasks is created, relying on the fact that $\mathrm{val}_\sigma(m(x)\{s\})$ starts with an invocation reaction event.

$$\frac{\begin{array}{c}m(x)\{s\} \in \overline{M} \qquad \sigma = \mathrm{last}(sh) \qquad pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma(m(x)\{s\}) \\ \rho \text{ concretizes } \tau \qquad \rho(pc) \text{ consistent} \qquad wf(sh ** \rho(\tau)^p)\end{array}}{sh, \Omega \to sh ** \rho(\tau)^p, \Omega[p \mapsto \Omega(p)+\{\mathrm{K}(s')\}]} \tag{43}$$

6.1.3 Well-formedness. We need to establish the basic properties of sending and receiving as well as to ensure that each process has the correct number of tasks. We extend the well-formedness predicate, introduced in Definition 5.7, except for the last rule that handles invocation reaction events, which is replaced by the last rule in the following definition.

Definition 6.2 (Well-formedness). The first three rules come from Definition 5.7. The fourth rule ensures no two messages will be sent with the same message identifier, the fifth one states that each message is received only once. Similarly, the sixth rule guarantees that each spawn allocates a new process identifier. The final rule reflects the fact that the code executed on a processor p can be the reaction to either a call or a spawn event, i.e. $spawnEv^p$ acts as a particular invocation event.

$$\begin{aligned}
wf(\epsilon) &= true \\
wf(sh \curvearrowright \sigma) &= wf(sh) \\
wf(sh \curvearrowright invEv^p(m,v)) &= wf(sh) \\
wf(sh \curvearrowright sendEv^p(e,p',i)) &= wf(sh) \wedge \nexists p'', e', p'''.\, sendEv^{p''}(e',p''',i) \in sh \\
wf(sh \curvearrowright receiveEv^p(e,p',i)) &= wf(sh) \wedge \nexists p'', e', p'''.\, receiveEv^{p''}(e',p''',i) \in sh \\
wf(sh \curvearrowright spawnEv^p(m,v,p')) &= wf(sh) \wedge \nexists p'', m', v'.\, spawnEv^{p''}(m',v',p') \in sh \\
wf(sh \curvearrowright invREv^p(m,v)) &= wf(sh) \wedge \\
&\quad \#_{sh}(invEv^p(m,v)) + \sum_{p' \in PId} \#_{sh}(spawnEv^{p'}(m,v,p)) > \#_{sh}(invREv^p(m,v))
\end{aligned}$$

Message identifiers are used to avoid sending or receiving the same message twice (i.e. two messages with the same identifier). In Section 6.2 we develop the well-formedness predicate in an incremental manner. For example, up to now we do not specify when a message is received. This is one of the aspects discussed in Section 6.2, where for example, the basic correctness criterion stating that "all received messages have been sent" is captured by the well-formedness condition in Definition 6.4, which should be used together with the *wf* predicate in Definition 6.2.

6.1.4 *Global Trace Semantics.*

Definition 6.3 (Program Semantics with Multiprocessors). Given a program P with a method table $\overline{M}$ and a main block s_{main}, let

$$sh_0,\ \Omega_0 \to sh_1,\ \Omega_1 \to \cdots$$

be a maximal sequence obtained by the repeated application of the composition rules (42)–(43), starting with

$$\langle I_P \rangle \curvearrowright spawnEv^0(main,0,0),\ \Omega[0 \mapsto \{\mathrm{K}(s_{main})\}] \ . \tag{44}$$

If the sequence is finite, then it must have the form

$$\langle I_P \rangle \curvearrowright spawnEv^0(main,0,0),\ \Omega[0 \mapsto \{\mathrm{K}(s_{main})\}] \to \cdots \to \underline{sh}, \Omega^{\emptyset} \ ,$$

where $\Omega^{\emptyset}(p) = \emptyset$ for all $p \in \mathrm{dom}(\Omega)$. If the sequence is infinite, let $\underline{sh} = \lim_{i \to \infty} sh_i$. The set of all such potentially infinite traces $\underline{sh}$ is denoted with $\mathbf{Tr}(P, I_P)$.

The spawn event at the start of a trace represents the creation of an initial process that runs the main method. The well-formedness of traces ensures that subsequently spawned processes will not erroneously be assigned index 0, which is reserved for the code executed in the main method.

6.2 Communication Patterns

We now unleash the power of traces with events and show that several well-known communication patterns can be defined simply by adding well-formedness constraints to Definition 6.2. Imposing such patterns in the LAGC semantics is modular in the sense that all other rules are unaffected by the enforced patterns. The well-formedness constraints defined in this section are to be added conjunctively to the well-formedness predicate *wf* over concrete traces *sh* (Definition 6.2) to achieve the desired semantic properties. We recall for the definitions below that e is an expression, $p, p' \in PId$ process identifiers, and $i \in MId$ a message identifier.

We first consider the basic tenet of *asynchronous communication*, stipulating that received messages must have been sent.

Definition 6.4 (Asynchronous Communication Constraint). Well-formed asynchronous communication is captured by the constraint wf_{ac}, defined as follows :

$$wf_{\text{ac}}(sh \curvearrowright receiveEv^{p}(e, p', i)) = wf(sh) \land sendEv^{p'}(e, p, i) \in sh$$

Next, we consider the *FIFO* (first-in, first-out) ordering principle, which stipulates that messages between the same processors must be received in the same order as they were sent; i.e. messages may not overtake each other. Let the notation $ev^{P}(\overline{v}) <_{sh} ev'^{P'}(\overline{v'})$ express that $ev^{P}(\overline{v})$ appears before $ev'^{P'}(\overline{v'})$ in a trace sh.

Definition 6.5 (FIFO Communication Constraint). Well-formed FIFO communication is captured by the constraint wf_{fifo}, defined as follows :

$$\begin{aligned} wf_{\text{fifo}}(sh \curvearrowright receiveEv^{p}(e, p', i)) = wf_{\text{ac}}(sh \curvearrowright receiveEv^{p}(e, p', i)) \land \\ \big(\forall sendEv^{p'}(e', p, i') \in sh.\ sendEv^{p'}(e', p, i') <_{sh} sendEv^{p'}(e, p, i) \\ \implies receiveEv^{p}(e', p', i') \in sh\big) . \end{aligned}$$

A *channel* is an ordered pair (p, p') of processor identifiers (not necessarily distinct) such that p sends messages to p' and p' receives messages sent by p. The messages in a channel after a trace sh can be given as a set of message identifiers, defined by the following function:

Definition 6.6 (Channel). Given a concrete trace sh and a channel $ch = (p, p')$, we define by $inChannel(sh, ch)$ the messages in ch that have been sent, but not yet received within sh:

$$inChannel(sh,\ (p, p')) \ = \ \{i \mid \exists e.sendEv^{p}(e, p', i) \in sh \land receiveEv^{p'}(e, p, i) \notin sh\} .$$

We can detect that a fixed bound $N > 0$, the *capacity* of a FIFO channel, has been reached by the following condition, which blocks a send event if the channel is full, i.e. the channel already contains $N-1$ messages.

Definition 6.7 (Bounded FIFO Communication Constraint). Let N be an integer. Well-formed N-bounded FIFO, ensured by the predicate $wf_{\text{bd}(N)}$, is captured by extending the constraint wf_{fifo} for standard (unbounded) FIFO by a check on the bound of the channel as follows. Well-formedness of a trace ending by a reception event is given by wf_{fifo}, and an additional rule checks well-formedness of message sending:

$$wf_{\text{bd}(N)}(sh \curvearrowright sendEv^{p}(e, p', i)) = wf(sh) \ \land \ |\ inChannel(sh,\ (p, p'))\ | < N .$$

Causally ordered (CO) messages are harder to characterize than the previous communication patterns, because in the most simple definition of CO found in [23], the receiving events must occur in the right order provided that sending events are causally ordered. Our event structure does not record causal ordering. This could certainly be realized, but it would add considerable complexity to the evaluation rules. Instead, we use the information that is already provided by the events in a given trace, based on an alternative characterization of CO [23]: "a computation is CO if and only if no message is bypassed by a chain of other messages". We first define a chain predicate such that $Chain(e,\ e',\ sh)$ is true if there is a chain of messages inside sh that asserts the causal ordering of e and e' induced by messages.

Definition 6.8 (Communication Chain). The predicate *Chain* holds for two events e and e' in a given trace sh, if there is a chain of messages asserting the causal ordering between e and e':

$$Chain(e,\ e',\ sh) \ = \ \begin{array}{l} \exists n, e_1, \dots, e_n, i_1, \dots, i_n, p_1, \dots, p_{n+1}. \\ \big(\forall 1 \le k \le n.\ sendEv^{p_k}(e_k, p_{k+1}, i_k) \in sh \land receiveEv^{p_{k+1}}(e_k, p_k, i_k) \in sh\big) \land \\ \big(\forall 1 < k \le n.\ receiveEv^{p_k}(e_{k-1}, p_{k-1}, i_{k-1}) <_{sh} sendEv^{p_k}(e_k, p_{k+1}, i_k)\big) \\ \text{such that } e <_{sh} sendEv^{p_1}(e_1, p_2, i_1) \land receiveEv^{p_{n+1}}(e_n, p_n, i_n) <_{sh} e' . \end{array}$$

We now define a well-formedness constraint that checks the absence of a message sent in the past (according to the definition of *Chain*) but not yet received. The following definition generalises Definition 6.5 to messages causally ordered (according to the *Chain* predicate) instead of messages originating from the same process:

Definition 6.9 (CO Communication Constraint). Well-formed CO communication is captured by the constraint wf_{co}, defined as follows :

$$\begin{aligned} wf_{\text{co}}(sh \curvearrowright receiveEv^{p}(e,p',i)) = wf(sh) \wedge sendEv^{p'}(e,p,i) \in sh \wedge \\ \forall e'',p'',i''.\, Chain(sendEv^{p''}(e'',p,i''), sendEv^{p'}(e,p,i), sh) \\ \implies receiveEv^{p}(e'',p'',i'') \in sh\,. \end{aligned}$$

Finally, we have a look at *synchronous call* patterns. Several definitions can be found in [23]. We adopt a definition that constrains well-formed traces rather strongly: any send event is immediately followed by the corresponding receive event.

Definition 6.10 (Strict Synchronous Communication Constraint). Well-formed strict synchronous communication is captured by the constraint wf_{sync}, defined as follows :

$$wf_{\text{sync}}(sh \curvearrowright receiveEv^{p}(e,p',i)) = \big(wf(sh) \wedge sh = sh' ** sendEv_{\sigma}^{p'}(e,p,i)\big)$$

The above constraint is realized, for example, by rendez-vous channels in ProMeLa [53], see Section 7.1.

One can define more liberal notions of synchronous communication that accept the presence of independent events between the sending and the reception of the message. The crown criterion [23], for example, can be used for this purpose. Its definition is global over traces and requires to define *wf* in a non-inductive manner (as an *invariant* holding for the trace at any time). We omit a detailed elaboration, because this would distract from the main point of this paper.

7 CASE STUDIES

We apply the LAGC framework to define the semantics of a channel-based language with processes—ProMeLa [53], of an actor-based language, and, finally, of an active object language [27] such as ABS [2, 40, 58].

In channel-based communication, where channels have an identifier, the processes send and receive messages via channels: messages are received in the order they are sent and they are received only once (consumed) by any process that has access to the channel identifier. In contrast, in actor-based and active object languages (1) each object has an active process that executes one message at a time, (2) messages are asynchronously sent to objects, so there is no fixed order on receiving messages, and (3) in active object languages replies from messages are received in a mail box (called *future* [28]) and might be read multiple times by anyone with access to the mail box's identifier. These case studies not only show how versatile the LAGC framework is for capturing diverse semantics and concurrent language constructs, but also that it is applicable to practically used languages.

7.1 ProMeLa

ProMeLa [53] is a concurrent modeling language that has been used to model and analyse numerous industrial systems. It comes with an industry-strength model checker [14]. We do not give a full LAGC-style semantics of ProMeLa, but we discuss its main features and illustrate that they can be formalised using minor variations of the concepts discussed above. We also do not give the DL calculus for ProMeLa, which mostly follows the calculus developed in Sections 4–5,

```
1  chan request = [0] of {byte};
2
3  active proctype Client0() {
4    request!0;
5  }
6  active proctype Client1() {
7    request!1;
8  }
```

```
 9  active proctype Server() {
10    byte n;
11
12    do
13    :: request?n;
14       printf("client %d\n", n)
15    od
16  }
```

Fig. 7. A simple ProMeLa program.

since ProMeLa is designed for model checking. We assume the reader is familiar with ProMeLa. A simple example of a ProMeLa program is shown in Figure 7.

Types, Variables, Expressions. All ProMeLa datatypes are mapped into finite integer types, including Booleans, process and channel identifiers, enumerations, arrays, etc. Strings occur only as literals in print statements. Variables declared outside a process are global and can be accessed by any process, such as the request channel in Figure 7, line 1. Like in WHILE, ProMeLa expressions are side effect-free and their evaluation is standard.

Processes. Methods and method calls do not exist in ProMeLa, so **call** is not present. All ProMeLa processes are declared and started in the beginning by a—possibly implicit—initial process, so **spawn** does not occur in the ProMeLa syntax either. In Figure 7, three processes are declared and started upfront. It is easy to create an initial judgment similar to rule (44) from the process declarations of a ProMeLa program P:

$$\langle I_P \rangle,\ \Omega_P\ ,$$

where $\Omega_P(p) = \{\mathrm{K}(\{s_p\})\}$ for each process $p \in P$ with program code s_p. The code is wrapped in a scope to handle local variable declarations as in Section 5.3.

There is no need for a method table, because there are no method calls. For the same reason, the range of Ω is a singleton. ProMeLa processes execute in parallel and interleave on global variables. The granularity of interleaving in ProMeLa is the same as in Section 5.1, so the continuations in the local rules are unchanged. Composition rule (42) is adequate for the ProMeLa semantics, while rule (43) is not needed, because there are no method calls and the program code of all processes is part of the initial judgment. In consequence, neither spawn, nor invocation, nor invocation reaction events occur in ProMeLa traces.

Statements. There is a print statement with no effect on states, we decide not to observe the effect on the trace, we can thus give it the semantics of **skip** in rule (6). Assignments are exactly as in WHILE, discussed in Section 3. Instead of **if**- and **while**-statements, ProMeLa has selection and repetition statements over guarded commands, which may occur only there. The semantics of guarded statements is as in Section 5.6. The semantics of a selection **if** $::\ \mathrm{g}_1; \mathrm{s}_1 \cdots :: \mathrm{g}_n; \mathrm{s}_n$ **fi** is a straightforward generalisation of rules (8) and (38) with $n + 1$ premises; one for each guarded statement and one premise if no guard is true. The path condition of the i-th guarded statement is g_i. When more than one guard is true, any one of them can be taken.

More generally, in ProMeLa, any statement s can serve as a guard, so a guard can have a side effect. This can be modeled simply by putting s into the continuation of its premise. All statements except **send** and **receive** (see below) are executable, their path condition simply becomes $\mathtt{tt}$. This can be assumed even for **send** and **receive**, because their execution is modeled by well-formedness constraints (see below), not by guards.

A repetition loops infinitely over its body and executes any of the statements whose guard is true; in the loop on the lines 12–15 of Figure 7, line 13 has an empty guard, which defaults to true. A repetition can only be exited by **break** or **goto** statements (see below). The local evaluation of repetitions can be reduced to selection in a similar manner as **while** is reduced to **if** in rule (9).

Jumps. ProMeLa features a **goto** statement, whose argument is a label within the same process. Such unconditional jumps without a (process) context switch are easy to characterise semantically via continuations:

$$\mathrm{val}_\sigma(\texttt{goto}\ l) = \{\emptyset \triangleright \langle\sigma\rangle \cdot \mathrm{K}(\mathrm{from}(l,p))\} \qquad \text{where } p \text{ is the evaluated process} \ . \tag{45}$$

The code in the continuation is computed from l and s_p via a code transformation $\mathrm{from}(l,p)$, defined as follows: If l appears in a top-level statement, then $\mathrm{from}(l,p)$ is the code syntactically following l. If l is inside a clause s_i of a selection statement s, then $\mathrm{from}(l,p)$ is the remaining clause of s_i after l, followed by the code after s. If l occurs inside a clause s_i of a repetition statement r, then $\mathrm{from}(l,p)$ is the remaining clause of s_i after l, followed by r and then the code that follows r. For nested statements this transformation is repeatedly applied inside out.

The **break** statement is equivalent to a **goto** with an implicit label that points to the code after the repetition it is contained in.

Atomic. ProMeLa has an **atomic** statement[11], which works as in Section 5.2, except it may contain a guard that can cause blocking. In this case, interleaving *is* possible. In case the guard g evaluates to true, statement s is simply atomically executed according to rule (24).[12] Otherwise, it puts the whole guarded atomic block into a continuation, so that the guard can be re-evaluated later.

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(\texttt{atomic}\{::g;s\}) = \{&\{\mathrm{val}_\sigma(g) = \mathrm{ff}\} \triangleright \langle\sigma\rangle \cdot \mathrm{K}(\texttt{atomic}\{::g;s\})\}\\ &\cup\{\{\mathrm{val}_\sigma(g) = \mathrm{tt}\} \cup pc \triangleright \tau \cdot \mathrm{K}(\llbracket\rrbracket) \mid pc \triangleright \tau \cdot \mathrm{K}(\llbracket\rrbracket) \in \mathrm{val}_\sigma^{O,F}(\texttt{atomic}(s))\}\end{aligned} \tag{46}$$

Channels. In ProMeLa, the **send** and **receive** commands have the syntax seen in Figure 7: if c is a channel identifier and $\overline{e}$ a list of expressions, then the current process uses $c\,!\,\overline{e}$ to send the value of $\overline{e}$ to c. Dually, with $c\,?\,\overline{e}$ it receives $\overline{e}$ from c.[13] In contrast to Definition 6.6 (Section 6), ProMeLa channels are explicitly declared, and the address of a send (the origin of a receive) statement is not the recipient (sender) process, but a channel. A message can be read by anyone who has access to the channel identifier it was sent to. In the case of globally declared channels (line 1), this is any process. For locally declared channels the receiving process must have received or declared the channel identifier.

The semantics of ProMeLa's send and receive can be modeled in analogy to Section 6.1, but one uses send and receive events of the form $sendEv^p(\overline{e}, c, i)$ and $receiveEv^p(\overline{e}, c, i)$, respectively, where the address is a channel.[14]

Well-formedness. Only the first two rules of Definition 6.2 (with obvious adaptations) are needed, because there are no method calls or dynamically created processes. On the other hand, the option

[11]Although ProMeLa permits loops to occur inside **atomic**, these are actually discouraged; if loops occur, they must always terminate.

[12]Rule (24) for unguarded atomic blocks ensures that the derived trace always ends with an empty continuation.

[13]ProMeLa allows either a variable, which is then set to the received value or an expression that is matched against it.

[14]In case one of the e is not a variable, but a match expression, the assignment in the local evaluation rule (39) has the variables *occurring* in e in its domain.

of local channels requires to model the visibility of channels. This can be done by well-formedness constraints of the following form:

$$wf_{\mathrm{ch}}(sh \curvearrowright receiveEv^{p}(e,c,i)) = wf(sh) \wedge \big(\mathrm{isLocal}(p,c) \vee \mathrm{isGlobal}(c) \vee receiveEv^{p}(c,c',i) \in sh\big) \ .$$

The predicate $\mathrm{isLocal}(p,c)$ holds if the process p declares a local channel named c, $\mathrm{isGlobal}(c)$ holds if channel c is declared globally. These predicates can be easily checked by inspecting the code of a ProMeLa program. The constraint expresses that each channel used in a receive statement in process p must be visible at that point. This is the case if it is either global, or locally declared in p, or it was received earlier. There is a similar constraint for send statements.

There are two channel variants in ProMeLa: *rendez-vous channels* impose strictly synchronous communication. For those channels the well-formedness constraint in Definition 6.10 is used. The other channel type are *buffered channels* with a capacity $N > 0$. These are characterised by the bounded FIFO pattern in Definition 6.7.

The standard ProMeLa semantics stipulates that received messages are consumed, i.e. they can be read only once, which is ensured by the following constraint:

$$wf(sh \curvearrowright receiveEv^{p}(e,c,i)) \ = \ wf(sh) \wedge \nexists e', p'.\, receiveEv^{p'}(e',c,i) \in sh \ .$$

It is possible to change certain features of ProMeLa channels, for example, reception can be made non-consumptive, out-of-order reception can be permitted, send and receive can be made non-blocking. All of these can be characterised by suitable well-formedness constraints, but we refrain from spelling out details.

Synchronisation. Channels, like guarded commands, are often used for synchronisation purposes in ProMeLa. The receive statement on a buffered channel is only executable if it contains a message; one can only send to a channel that is not full. Non-executable statements block a process until they become executable. In contrast to guarded commands, no specific evaluation rule is required to model blocking senders and receivers, because the correct communication order is already guaranteed by the well-formedness constraints.

This consideration suggests that blocking guards might as well be modeled by suitable "waiting" events. Indeed, this is possible, but less natural in that case.

7.2 Actors

We define a pure, object-based actor language based on the language in Section 6; Figure 8 shows the syntax. There is a one-to-one mapping between objects and actors. Each actor has its own process with local memory and executes calls to its methods with run-to-completion semantics (in consequence, non-terminating method calls render an actor unresponsive). Each actor can only access its own local memory. The state of another actor can only be modified via a method call.

$$\begin{aligned}
P \in Prog &::= \overline{CD}\ sc \\
CD \in ClassDecl &::= \mathtt{class}\ C\ \{\overline{fd}\ \overline{M}\} \\
M \in MethDecl &::= m(\overline{x})\ sc \\
d \in VarDecl &::= \varepsilon \mid x;\ d \\
sc \in Scope &::= \{d\ \mathtt{atomic}(s)\} \\
s \in Stmt &::= \ \mathtt{skip} \mid x := rhs \mid \mathtt{if}\ e\ \{\ s\ \} \mid x!m(\overline{e}) \mid s; s \\
rhs \in Rhs &::= e \mid \mathtt{new}\ C\ (\overline{e})
\end{aligned}$$

Fig. 8. Syntax of the Actor Language in Section 7.2

In this actor language, a program P consists of a set of class declarations CD with a main block sc. Each class has a name C, which also becomes the type of the class, with a sequence of field declarations fd and method declarations M. Field declarations have the same syntax as local variable declarations d. For simplicity, we let each method declaration associate a *unique* name m to a list of arguments $\overline{x}$ and a method body sc.

A method body starts with local variable declarations followed by an **`atomic`** statement wrapping a sequence of statements s. Recall from Section 5.2 that we excluded loops inside **`atomic`**, and thus exclude the **`while`** statement from the syntax. This is no fundamental restriction, because actors may call each other and permit unbounded recursion.

The effect of **`atomic`** is to enforce a run-to-completion semantics. Rather than changing the semantics of the programs discussed in Sections 3–6, we here make atomicity explicit. In most actor languages [27], the **`atomic`** keyword is not used because actors are single threaded, and thus "locally atomic" by default.

We extend the statements of WHILE, given in Figure 2, with asynchronous method calls $x!m(\overline{e})$ on a caller object x. The semantics of asynchronous calls combines the semantics of sending a message and a method call. These calls are not blocking, so in the absence of futures or promises, there is no direct way to return a value to the caller (return values via futures are modeled in Section 7.3 below).[15] The right-hand-side of assignments includes expressions and **`new`** $C\ (\overline{e})$. The latter creates a new actor object of class C with constructor arguments $\overline{e}$.

In the following, let OId be a set of object identifiers with typical element o. We represent a program P in terms of a global lookup table $\mathcal{G}$: A set of triples each consisting of a class name with its field and method declarations:

$$\mathcal{G} = \{\langle C, \overline{fd}, \overline{m(\overline{x})\ sc}\rangle \mid fd \in \text{fields}(C),\ m \in \text{methods}(C),\ C \in P\}\ .$$

In the above expression we use selector functions with obvious definitions: $\text{fields}(C)$ returns the field names of a class C, $\text{methods}(C)$ returns the methods declared in a class C. We also define $\text{class}(X)$, where X is either an object identifier or a variable of type OId, that returns the class of X, as well as $\text{lookup}(m, \mathcal{G})$ that returns the declaration of a method with name m in $\mathcal{G}$.

7.2.1 Local Evaluation. When following the principle of local evaluation, we face the problem that a statement s, which is locally evaluated, cannot possibly know the object it is running on. We address this issue in the same manner as we dealt with unknown values before, i.e. by introducing a parameter O that is instantiated during trace composition. Since this parameter must be instantiated consistently for all statements an object O executes, it becomes a parameter of the semantic evaluation function, denoted $\text{val}_\sigma^O(s)$, where O is a symbolic variable. The evaluation function $\text{val}_\sigma^O(s)$ produces the traces when s is running on object O; in particular, we let $\text{val}_\sigma^O(\textbf{this}) = O$.

The evaluation rule for **`new`** below introduces an event $newEv_\sigma(o, \overline{v})$ to capture that a new object o with arguments $\overline{v}$ is created. Similar to Rule (41) that spawns a task, the local evaluation Rule (47) for object creation creates a fresh symbolic variable X to represent the unknown object identity that is returned. Hence, Rule (47) has an empty path condition, extends the trace by the *new object creation* event followed by an updated state, where x is mapped to symbolic variable X and object fields are mapped to the constructor arguments. We suppose that $\mathcal{G}$ is pre-populated with enough variables of each class. Consequently, Rule (47) can pick a symbolic variable X that is fresh in σ and such that $\text{class}(X) = C$.

[15]It is possible to pass the caller's identity as a call argument, which the callee could then use to return a value to the caller's state via a separate callback. But this does not give the caller a handle to access the result.

$$\begin{aligned}&\mathrm{val}^{O}_{\sigma}(x := \textbf{new}\ C\ (\overline{e})) = \\ &\qquad \{\emptyset \triangleright \mathit{newEv}^{\{X\}}_{\sigma}(X, \mathrm{val}^{O}_{\sigma}(\overline{e})) \curvearrowright \sigma[x \mapsto X, X \mapsto *, \overline{X.fd \mapsto \mathrm{val}^{O}_{\sigma}(e)}] \cdot \mathrm{K}(\lozenge) \\ &\qquad \mid X \notin \mathrm{dom}(\sigma),\ \mathrm{class}(X) = C,\ fd \in \mathrm{fields}(C)\}\end{aligned} \tag{47}$$

Rule (48) for non-blocking asynchronous method calls is similar to Rule (29), but the invocation event $\mathit{invEv}_{\sigma}(\overline{v}, x, m, i)$ also includes the callee object x and a message identifier $i \in \mathit{MId}$. Thus, asynchronous method calls can be viewed as a combination of procedure calls and message sending.

$$\mathrm{val}^{O}_{\sigma}(x!m(\overline{e})) = \{\emptyset \triangleright \mathit{invEv}_{\sigma}(\mathrm{val}^{O}_{\sigma}(\overline{e}), \mathrm{val}^{O}_{\sigma}(x), m, i) \cdot \mathrm{K}(\lozenge)\} \mid i \in \mathit{MId}\} \tag{48}$$

The following rule evaluates the body of a method and corresponds to Rule (36) but adapted to a list of arguments instead of a single one. The invocation reaction event inserted by the rule includes symbolic variables which represent the method's actual parameters, which are unknown in the local context. Since there are no return values, the caller needs not to be known. Consequently, the event does not include the caller identity.[16]

$$\begin{aligned}\mathrm{val}^{O}_{\sigma}(m(\overline{x})\ sc) = \{&\emptyset \triangleright \mathit{invREv}^{\overline{Z}}_{\sigma}(\overline{Z}, m, i) \curvearrowright \sigma[\overline{z} \mapsto \overline{Z}, \overline{Z} \mapsto *] \cdot \mathrm{K}(sc[\overline{x} \leftarrow \overline{z}]) \\ &\mid \overline{z}, \overline{Z} \notin \mathrm{dom}(\sigma),\ i \in \mathit{MId}\}\end{aligned} \tag{49}$$

7.2.2 Trace Composition. As explained above, the local evaluation is parameterised with the executing object O. During trace composition, this parameter is instantiated by a concrete object identifier o. To associate events with the object they originate from, traces are tagged with that object. This works exactly like the process identifier tags in Definition 6.1 of Section 6.

We use a mapping Ω into multisets of possible continuations like in Section 6, with the difference that its domain is object identifiers instead of process identifiers. We use the same notation for multiset operations as before.

Trace composition in Rule (50) follows Rule (42) closely , with two small modifications: instead of process identifiers, objects identifiers are used and the evaluation of the continuation s is performed on the object o, where it is scheduled. The first two premises of the rule capture the scheduling decision to continue the trace by executing task s on object o.

$$\frac{\begin{array}{c} o \in \mathrm{dom}(\Omega) \qquad \Omega[o] = q + \{\mathrm{K}(s)\} \qquad \sigma = \mathrm{last}(sh) \\ pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}^{o}_{\sigma}(s) \qquad \rho \text{ concretizes } \tau \qquad \rho(pc) \text{ consistent} \qquad \mathit{wf}(sh \ast\ast \rho(\tau)^{o}) \end{array}}{sh, \Omega \rightarrow sh \ast\ast \rho(\tau)^{o}, \Omega[o \mapsto q + \{\mathrm{K}(s')\}]} \tag{50}$$

Likewise, Rule (51) *exactly* follows the pattern established with Rule (43). The rule picks a method m and an object o, on which it is supposed to be executed. If that object is not yet in the domain of Ω, then we use the notational convention $\Omega(o) = \emptyset$. This corresponds to object creation. Either way, the method declaration is evaluated on o. Observe that Rule (49) issues an *invocation reaction* event. Well-formedness ensures that an *invocation* event of the form $\mathit{invEv}(\overline{v}, o, m, i)$ is present in sh that matches the message identifier, method parameters, and callee.

$$\frac{\begin{array}{c} \mathrm{lookup}(m, \mathcal{G}) = m(\overline{x})\ sc \qquad \sigma = \mathrm{last}(sh) \qquad pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}^{o}_{\sigma}(m(\overline{x})\ sc) \\ \rho \text{ concretizes } \tau \qquad \rho(pc) \text{ consistent} \qquad \mathit{wf}(sh \ast\ast \rho(\tau)^{o}) \end{array}}{sh, \Omega \rightarrow sh \ast\ast \rho(\tau)^{o},\ \Omega[o \mapsto \Omega[o] + \{\mathrm{K}(s')\}]} \tag{51}$$

[16] See Rule (55) in Section 7.3 that handles returning values via futures.

7.2.3 Well-formedness. Well-formedness must essentially ensure two aspects: (1) the uniqueness of events, for example, there is at most one $newEv(o, \_)$ per object, i.e., an object cannot be created twice; (2) the event sequence related to a call to a method m on object o with parameters $\overline{v}$ has the following form:

$$\cdots \quad newEv^{o''}(o, \overline{v}) \quad \cdots \quad invEv^{o'}(\overline{v'}, o, m, i) \quad \cdots \quad invREv^{o}(\overline{v'}, m, i) \quad \cdots$$

where o' and o'', which are the object identifiers specified in the superscripts of the new event and the invocation event, can be the same object. This is achieved with the equations in Figure 9 (we do not repeat the first two lines of Definition 5.7, which are always assumed to be part of well-formedness). Please observe that the well-formedness of the event sequence corresponding to a method m invoked on an object o is ensured by the identifiers of the callee and the message in the invocation event and the invocation reaction event.

$$\begin{aligned}
wf(sh \curvearrowright newEv^{o'}(o, \overline{v})) &= wf(sh) \wedge \nexists o'', \overline{v'}.\ newEv^{o''}(o, \overline{v'}) \in sh \\
wf(sh \curvearrowright invEv^{o'}(\overline{v}, o, m, i)) &= wf(sh) \wedge\ \exists o'', \overline{v}'.newEv^{o''}(o, \overline{v}') \in sh \wedge \\
&\quad\ \nexists o''', \overline{v}'', o'''', m'.\ invEv^{o'''}(\overline{v}'', o'''', m', i) \in sh \\
wf(sh \curvearrowright invREv^{o'}(\overline{v}, m, i)) &= wf(sh) \wedge\ \exists o.invEv^{o}(\overline{v}, o', m, i) \in sh \wedge \\
&\quad\ \nexists o'', \overline{v}', m'.\ invREv^{o''}(\overline{v'}, m', i) \in sh
\end{aligned}$$

Fig. 9. Well-formedness for Actor language.

7.2.4 Global Trace Semantics.

Definition 7.1 (Program Semantics for Actors). Given a finite program P with a lookup table $\mathcal{G}$ and a main block *main*, i.e., $\Omega_{init}(o_{main}) = \{\mathrm{K}(main)\}$. Let σ_ε denote the empty state, i.e. $\mathrm{dom}(\sigma_\varepsilon) = \emptyset$. Let

$$sh_0,\ \Omega_0 \longrightarrow\ sh_1,\ \Omega_1 \longrightarrow \cdots$$

be a maximal sequence obtained by the repeated application of the composition rules (Rules (50)–(51)), starting with[17]

$$\langle \sigma_\varepsilon \rangle \curvearrowright newEv^{o_{main}}(o_{main}, \varepsilon),\ \Omega_{init}\ .$$

If the sequence is finite, then it must have the form

$$\langle \sigma_\varepsilon \rangle \curvearrowright newEv^{o_{main}}(o_{main}, \varepsilon),\ \Omega_{init} \longrightarrow \cdots \longrightarrow \underline{sh}, \Omega^{\emptyset}\ ,$$

where $\Omega^{\emptyset}(o) = \emptyset$ for all $o \in \mathrm{dom}(\Omega)$. If the sequence is infinite, let $\underline{sh} = \lim_{i\to\infty} sh_i$. The set of all such potentially infinite traces $\underline{sh}$ is denoted with $\mathbf{Tr}(P, \sigma_\varepsilon)$.

The new event at the start of a trace represents creation, by the system, of an initial object that runs the main method. The well-formedness of events ensures that subsequently created objects will not erroneously be the initial object, which is reserved for the code executed in the main method.

[17]Note that we denote by ε an empty list of method arguments.

7.3 Active Objects

Active object languages [27] have a mechanism like futures or promises that provides a reference to the value computed by an asynchronous method call. This makes it possible for a task to free its processor resource while waiting for a result to be finished (so-called cooperative multi-tasking). We modify the syntax from Section 7.2 and introduce futures, a **`return`** statement, and **`get`**-expressions to retrieve the value stored in a future; the resulting syntax is given in Figure 10. Note that the latter can block if this value is not yet available.

$$
\begin{array}{rl}
P \in \mathit{Prog} ::= & \overline{CD}\ sc \\
CD \in \mathit{ClassDecl} ::= & \texttt{class}\ C\ \{\overline{fd}\ \overline{M}\} \\
M \in \mathit{MethDecl} ::= & m(\overline{x})\ sc \\
d \in \mathit{VarDecl} ::= & \varepsilon \mid x;\ d \\
sc \in \mathit{Scope} ::= & \{d\ s;\ \texttt{return}\ e\} \\
g \in \mathit{guard} ::= & e? \mid e \\
s \in \mathit{Stmt} ::= & \texttt{skip} \mid x := rhs \mid \texttt{if}\ e\ \{\ s\ \} \mid \texttt{while}\ e\ \{\ s\ \} \mid \texttt{this}.m(\overline{e}) \mid \texttt{await}\ g \mid s; s \\
rhs \in \mathit{Rhs} ::= & e \mid \texttt{new}\ C\ (\overline{e}) \mid x!m(\overline{e}) \mid e.\texttt{get}
\end{array}
$$

Fig. 10. Syntax of Active Objects with Futures.

We need to capture cooperative multi-tasking and the blocking of future access into the semantics. This cannot be done with **`atomic`**, as in the previous section, because the suspending statements might occur nested inside loops and recursive calls. Instead we ensure that a task runs uninterrupted by imposing structural requirements that identify the currently running task. As a consequence we specify a semantics with finer-grained, explicitly controlled interleaving between tasks. For this reason, we now include **`while`**-loops as well as synchronous self-calls **`this`**.$m(\overline{e})$ in the syntax. The fine-grained semantics can handle non-terminating loops and recursion (with or without suspending statements inside).

Task suspension takes place in **`await`** g statements, where the guard g suspends the execution of a task; it works similarly to the guarded statements of Section 5.6. An **`await`** releases its object's process, such that other tasks may be executed. The execution of the statements following the guard can resume when the guard evaluates to true. Even when the guard evaluates to true, the execution may be suspended. Guard expressions either have the form x?, which synchronize with the future referenced by x receiving a value (the future thereby gets *resolved*), or they are Boolean expressions e.

The second difference to the Actor language of Section 7.2 is that asynchronous method calls return values to their associated future.[18] The **`return`** statement terminates the execution of an asynchronous method and returns its argument. The syntax enforces that the **`return`** statement is at the end of a method body. Asynchronous method calls now appear on the right hand side of an assignment, the variable on the left is a *future* associated with the call. The future's value can be retrieved with **`get`**, whose argument expression must evaluate to a future. To avoid blocking, one can precede e.**`get`** with **`await`** e?. Futures are first-class values and may be passed as method arguments to other objects.

7.3.1 Local Evaluation. For similar reasons as for objects in Section 7.2, we need to provide the future towards whose value a local statement is contributing, as a parameter of the evaluation function. This future, sometimes called the *destiny* of the executing code [28], cannot possibly be

[18]Obviously, synchronous calls can also be equipped with return values in a straightforward manner.

known locally. Hence, the form of the semantic evaluation function becomes $\mathrm{val}_{\sigma}^{O,F}(s)$, where F is a symbolic variable of type future with values in *FId*; the evaluation function captures that s is running on object O and has destiny F. To keep track of the destiny, we store it together with the continuations; in the configurations, continuations now take the form $\mathrm{K}^{f}(s)$.

The rule for assignment with an asynchronous call on the right-hand-side first emits an invocation event $invEv(\overline{e}, o, m, f)$, similar to Rule (48). The difference is that now the future F', that is the destiny of the call, must be recorded both in the left-hand-side variable on the left and in the invocation event. That future can also be used to identify the call, thus replacing the message identifier. Since the value of F' cannot be known locally, it is modeled as a fresh symbolic variable, similarly as in Rules (40)–(41).

$$\begin{aligned}&\mathrm{val}_{\sigma}^{O,F}(x := e!m(\overline{e'})) =\\ &\quad\{\emptyset \triangleright invEv_{\sigma}^{\{F'\}}(\mathrm{val}_{\sigma}^{O,F}(\overline{e'}), \mathrm{val}_{\sigma}^{O,F}(e), m, F') \curvearrowright \sigma[x \mapsto F', F' \mapsto *] \cdot \mathrm{K}(\llbracket\rrbracket)\\ &\qquad \mid F' \notin \mathrm{dom}(\sigma)\}\end{aligned} \tag{52}$$

To ensure that only futures with an available value are retrieved, we introduce new events $compEv_{\sigma}(f, v)$ and $compREv_{\sigma}(f, v)$. These events denote the *completion* (the value of future f is available) and *completion reaction* (the value of f is retrieved) of an asynchronous method. The evaluation rule for returning the completed result simply inserts a *completion* event for the current future.

$$\mathrm{val}_{\sigma}^{O,F}(\mathbf{return}\ e) = \{\emptyset \triangleright compEv_{\sigma}(F, \mathrm{val}_{\sigma}^{O,F}(e)) \cdot \mathrm{K}(\llbracket\rrbracket)\} \tag{53}$$

To retrieve the returned value stored in a future, Rule (54) inserts a completion reaction event and extends the current state with a symbolic variable V that holds the as yet unknown return value.

$$\begin{aligned}\mathrm{val}_{\sigma}^{O,F}(x := e.\mathbf{get}) = \{&\emptyset \triangleright compREv_{\sigma}^{\{V\}}(\mathrm{val}_{\sigma}^{O,F}(e), V) \curvearrowright \sigma[x \mapsto V, V \mapsto *] \cdot \mathrm{K}(\llbracket\rrbracket)\\ &\mid V \notin \mathrm{dom}(\sigma)\}\end{aligned} \tag{54}$$

The rule for evaluating a method body is similar to Rule (49), except that the future variable f of the freshly running process is unified with the variable f of the invocation reaction event. It also must contain the (as yet unknown) caller Y. Well-formedness of the trace containing $invREv(\overline{X}, Y, m, f)$ will ensure that the f of the invocation reaction is matched with the same f for a matching invocation event (see the definition of $wf(sh)$ below).

$$\begin{aligned}&\mathrm{val}_{\sigma}^{O,F}(m(\overline{x})\ sc) =\\ &\quad\{\emptyset \triangleright invREv_{\sigma}^{\overline{X}\cup\{Y\}}(\overline{X}, Y, m, F) \curvearrowright \sigma[\overline{x}' \mapsto \overline{X}, \overline{X} \mapsto *, Y \mapsto *] \cdot \mathrm{K}(sc[\overline{x} \leftarrow \overline{x}'])\\ &\qquad \mid \overline{x}', \overline{X}, Y \notin \mathrm{dom}(\sigma)\}\end{aligned} \tag{55}$$

For **while**-loops, we can use Rule (9). Rule (56) handles synchronous self-calls (no other synchronous calls are considered here) by inlining the method body. It obtains the method declaration of m with body sc using the auxiliary function lookup() and turns the formal parameters $\overline{x}$ into local variable declarations bound to the argument values $\overline{e}$. The entire statement sequence is wrapped into a scope for name disambiguation.

$$\mathrm{val}_{\sigma}^{O,F}(\mathbf{this}.m(\overline{e})) = \mathrm{val}_{\sigma}^{O,F}(\{\overline{x};\ \overline{x} := \mathrm{val}_{\sigma}^{O,F}(\overline{e});\ sc\}),\ \text{where}\ \mathrm{lookup}(m, \mathcal{G}) = m(\overline{x})\ sc \tag{56}$$

Suspended tasks are introduced by **await** statements. We only need to specify how to progress after an **await**—the trace composition semantics will deal with actual task suspension by checking for the presence of an **await** guard. There are two syntactically distinct cases. The first corresponds to a guarded command in Rule (38), the second requires that the guarded future is resolved and thus introduces a completion reaction event. This event must match a previous completion event involving the same future, which will be ensured by well-formedness in the global rules.

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(\mathtt{await}\ e) &= \{\{\mathrm{val}_\sigma(e) = \mathtt{tt}\} \triangleright \langle\sigma\rangle \cdot \mathrm{K}(\emptyset)\}\\ \mathrm{val}_\sigma^{O,F}(\mathtt{await}\ e?) &= \{\emptyset \triangleright \mathit{compREv}_\sigma^{\{V\}}(\mathrm{val}_\sigma^{O,F}(e), V) \cdot \mathrm{K}(\emptyset) \mid V \notin \mathrm{dom}(\sigma)\}\end{aligned} \tag{57}$$

7.3.2 Trace Composition. We extend the continuations in the configuration with a future identifier that corresponds to the future to be resolved by the considered task (thus, continuations take the form $\mathrm{K}^f(s)$). With active objects, there is no data-race among concurrent tasks, because each data item belongs to one single active object and there is at most a single active task for each active object. One way to realize this in the semantics is to assign a single process to each active object and make sure that among all the tasks of this process, only one is *not* waiting for its turn. In other words, for each object identifier o, all continuations, except possibly one, are of the form $\mathrm{K}^f(\mathtt{await}\ g; s)$.

We use the symbol Q to denote a multiset of *suspended* or empty continuations of the form $\mathrm{K}^f(\mathtt{await}\ g; s)$ or $\mathrm{K}^f(\emptyset)$. The current global state of a program trace is represented by a mapping Σ from object identifiers to multisets of continuations. For all object identifiers o, $\Sigma[o]$ is either empty or of the form $Q+\{\mathrm{K}^f(s)\}$, which reflects that there is at most one active task. If s is of the form $\mathtt{await}\ g; s'$, then no task is currently executing on object o and any task can be activated; otherwise, s must be the only continuation that can be executed in object o. It is called the *active* task. The rewrite rule for empty trace simplification is extended to the new continuations: $Q + \mathrm{K}^f(\emptyset) \rightsquigarrow Q$. The trace composition rule closely follows Rule (50):

$$\frac{\begin{array}{c}\Sigma[o] = Q+\{\mathrm{K}^f(s)\} \qquad \sigma = \mathrm{last}(sh)\\ pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma^{o,f}(s) \qquad \rho \text{ concretizes } \tau \qquad \rho(pc) \text{ consistent} \qquad wf(sh ** \rho(\tau)^o)\end{array}}{sh, \Sigma \longrightarrow sh ** \rho(\tau)^o, \Sigma[o \mapsto Q+\{\mathrm{K}^f(s')\}]} \tag{58}$$

The main difference is that the global state has a different signature and the nature of Q enforces the selection of the current task. Observe that this rule either evaluates the current active task or activates a new task in o when there is no active task.

Rule (59) follows Rule (51) except that the evaluation function is now tagged with a concrete future identity f. The correctness of the concretisation is ensured by the well-formedness premise that requires an invocation event of the form $invEv(\overline{v}, o, m, f)$ to be present in sh. Note that we only run a new method if there is no active task ($\Sigma[o] = Q$). Here, the destiny of the created thread is the future that was created during the evaluation of the invocation event $invEv(\overline{v}, o, m, f)$. As no other task is active, the method can start running immediately.[19]

$$\frac{\begin{array}{c}\Sigma[o] = Q \qquad \mathrm{lookup}(m, \mathcal{G}) = m(\overline{x})\ sc\\ \sigma = \mathrm{last}(sh) \qquad f \in \mathit{FId} \qquad o \in \mathit{OId} \qquad pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma^{o,f}(m(\overline{x})\ sc)\\ \rho \text{ concretizes } \tau \qquad \rho(pc) \text{ consistent} \qquad wf(sh ** \rho(\tau)^o)\end{array}}{sh, \Sigma \longrightarrow sh ** \rho(\tau)^o,\ \Sigma[o \mapsto Q+\{\mathrm{K}^f(s')\}]} \tag{59}$$

This last rule can be triggered when Q is empty, i.e. when a newly created object handles its first invocation or when an object has no task currently running.

PROPOSITION 7.2. *The property that $\Sigma[o]$ contains at most one active task for any $o \in \mathit{OId}$ is an invariant preserved by applications of Rules* (58)–(59).

[19]Running the method immediately allows us, for example, to encode a FIFO service policy by composing with the communication ordering rules of Section 6.2

$$
\begin{aligned}
wf(sh \curvearrowright newEv^{o'}(o,\overline{v})) &= wf(sh) \wedge \nexists o'',\overline{v'}.\, newEv^{o''}(o,\overline{v'}) \in sh\\
wf(sh \curvearrowright invEv^{o'}(\overline{v},o,m,f)) &= wf(sh) \wedge\ \exists o'',\overline{v}'.newEv^{o''}(o,\overline{v}') \in sh \wedge\\
&\quad \nexists o''',\overline{v}'',o'''',m'.\, invEv^{o'''}(\overline{v}'',o'''',m',f) \in sh\\
wf(sh \curvearrowright invREv^{o'}(\overline{v},o,m,f)) &= wf(sh) \wedge\ invEv^{o}(\overline{v},o',m,f) \in sh \wedge\\
&\quad \nexists o'',\overline{v'},o''',m'.\, invREv^{o''}(\overline{v'},o''',m',f) \in sh\\
wf(sh \curvearrowright compEv^{o}(f,v)) &= wf(sh)\\
wf(sh \curvearrowright compREv^{o}(f,v)) &= wf(sh) \wedge \exists o', compEv^{o'}(f,v) \in sh
\end{aligned}
$$

Fig. 11. Well-Formedness for Actors with Futures.

Proof. When an object o is created, $\Sigma[o]$ is empty. Every time one of the Rules (58)–(59) is applied to o, a continuation is added to $\Sigma[o]$. As long as $\Sigma[o]$ contains only suspended continuations, either rule can be applied to o. Consequently, a suspended task on o can be activated or a new method invocation on o can start to be executed. If there is one active task $K^{f}(s)$ in $\Sigma[o]$, only Rule (58) can be applied to o. In this case, execution on o is forced by Rule (58) to continue with this specific $K(s)$, i.e., the active task on o. □

The proposition guarantees that each object has at most one active task at any time. This captures the semantics of sequential execution on an object between two suspension points.

7.3.3 Well-Formedness. Compared to Section 7.2.3, well-formedness must reflect the life cycle of asynchronous method calls with completion and completion reaction events to ensure that return values are not delivered and retrieved too early:

$$
\begin{gathered}
\cdots \quad newEv^{o''}(o,\overline{v}) \quad \cdots \quad invEv^{o'}(\overline{v'},o,m,f) \quad \cdots \quad invREv^{o}(\overline{v'},o',m,f) \quad \cdots\\
\cdots \quad compEv^{o}(f,v) \quad \cdots \quad compREv^{o'''}(f,v) \quad \cdots
\end{gathered}
$$

where o' and o'' can be the same object, and o''' can be any object except o. Up to tagging, the first four events are unique in each trace, but not so the final completion reaction event. This is, because any object can retrieve the value stored in a future as long as it possesses the future's identifier.

The fourth equation in Figure 11 is straightforward, because (i) a **return** can only be encountered after an invocation reaction event, which is guaranteed to be unique by the third equation, and (ii) Rule (58) makes sure that it is evaluated on the matching future and object.

The final equation ensures the value of a future can only be fetched after the future is resolved.

7.3.4 Global Trace Semantics.

Definition 7.3 (Program Semantics for Active Objects). Given a program P with a method table $\mathcal{G}$ and a main block *main*, i.e., $\Sigma_{init}(o_{main}) = \{K^{f_{init}}(main)\}$. Let σ_ε denote the empty state, i.e. $\mathrm{dom}(\sigma_\varepsilon) = \emptyset$. Let

$$sh_0,\, \Sigma_0 \longrightarrow\ sh_1,\, \Sigma_1 \longrightarrow \cdots$$

be a maximal sequence obtained by the repeated application of the composition rules (Rules (58)–(59)), starting with

$$\langle \sigma_\varepsilon \rangle \curvearrowright newEv^{o_{main}}(o_{main}, \varepsilon),\ \Sigma_{init}\ .$$

If the sequence is finite, then it must have the form

$$\langle \sigma_\varepsilon \rangle \curvearrowright newEv^{o_{main}}(o_{main}, \varepsilon),\ \Sigma_{init} \longrightarrow \cdots \longrightarrow \underline{sh}, \Sigma^{\emptyset}\ ,$$

where $\Sigma^{\emptyset}(o) = \emptyset$ for all $o \in \mathrm{dom}(\Sigma)$. If the sequence is infinite, let $\underline{sh} = \lim_{i \to \infty} sh_i$. The set of all such potentially infinite traces $\underline{sh}$ is denoted with $\mathbf{Tr}(P, \sigma_\varepsilon)$.

The new event at the start of a trace represents the creation, by the system, of an initial object that runs the main method. The well-formedness of events ensures that subsequently created objects will not erroneously be the initial object, which is reserved for code executed in the main method.

Example 7.4. Consider the following program P:

```
1  class C {
2    m(n) {
3      n := n + 1;
4      return n;
5    }
6  }
```

```
7  { // main block
8      a; x; f; y;
9      a := 1;
10     x := new C();
11     f := x!m(a);
12     await f?;
13     y := f.get;
14 }
```

Let the global lookup table $\mathcal{G}$ for this program be $\{\langle C, \varepsilon, m(n)\{n := n + 1; \texttt{return}\ n; \}\rangle\}$. In the following, we use s_i to indicate the sequence of statements of a method body starting from line i.

We first show the abstract local traces. The evaluation of the *main* method is as follows, where s_{main} represents the body in *main*, and s_9 corresponds to $s_{main}[a \leftarrow a', x \leftarrow x', f \leftarrow f', y \leftarrow y']$.

$$\begin{aligned}
\mathrm{val}_\sigma^{O,F}(\{\ a; x; f; y; s_{main}\ \}) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[a' \mapsto 0, x' \mapsto 0, f' \mapsto 0, y' \mapsto 0] \cdot \mathrm{K}(\{\ s_9\ \}) \\
&\qquad \mid a', x', f', y' \notin \mathrm{dom}(\sigma)\} \\
\mathrm{val}_\sigma^{O,F}(\{\ s_9\ \}) &= \mathrm{val}_\sigma^{O,F}(s_9) \\
\mathrm{val}_\sigma^{O,F}(a' := 1) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[a' \mapsto 1] \cdot \mathrm{K}(\text{🍶})\} \\
\mathrm{val}_\sigma^{O,F}(x' := \texttt{new}\ C()) &= \{\emptyset \triangleright \mathit{newEv}_\sigma^{\{X\}}(X, \varepsilon) \curvearrowright \sigma[x' \mapsto X, X \mapsto *] \cdot \mathrm{K}(\text{🍶}) \\
&\qquad \mid X \notin \mathrm{dom}(\sigma),\ \mathrm{class}(X) = C\} \\
\mathrm{val}_\sigma^{O,F}(f' := x'!m(a')) &= \{\emptyset \triangleright \mathit{invEv}_\sigma^{\{F'\}}(\mathrm{val}_\sigma^{O,F}(a'), \mathrm{val}_\sigma^{O,F}(x'), m, F') \curvearrowright \sigma[f' \mapsto F', F' \mapsto *] \cdot \mathrm{K}(\text{🍶}) \\
&\qquad \mid F' \notin \mathrm{dom}(\sigma)\} \\
\mathrm{val}_\sigma^{O,F}(\texttt{await}\ f'?) &= \{\emptyset \triangleright \mathit{compREv}_\sigma^{\{V\}}(\mathrm{val}_\sigma^{O,F}(f'), V) \curvearrowright \sigma[V \mapsto *] \cdot \mathrm{K}(\text{🍶}) \mid V \notin \mathrm{dom}(\sigma)\} \\
\mathrm{val}_\sigma^{O,F}(y' := f'.\texttt{get}) &= \{\emptyset \triangleright \mathit{compREv}_\sigma^{\{V\}}(\mathrm{val}_\sigma^{O,F}(f'), V) \curvearrowright \sigma[y' \mapsto V, V \mapsto *] \cdot \mathrm{K}(\text{🍶}) \\
&\qquad \mid V \notin \mathrm{dom}(\sigma)\}
\end{aligned}$$

The evaluation of the body of method m is shown below, where sc represents $\{n := n+1; \texttt{return}\ n; \}$ and s_3 the content of the block $sc[n \leftarrow n']$.

$$\begin{aligned}
\mathrm{val}_\sigma^{O,F}(m(n)\ sc) &= \{\emptyset \triangleright \mathit{invREv}_\sigma^{\{N,X,F'\}}(N, X, m, F') \\
&\qquad \curvearrowright \sigma[n' \mapsto N, N \mapsto *, X \mapsto *, F' \mapsto *] \cdot \mathrm{K}(sc[n \leftarrow n']) \mid n', N, X, F' \notin \mathrm{dom}(\sigma)\} \\
\mathrm{val}_\sigma^{O,F}(\{\ s_3\ \}) &= \mathrm{val}_\sigma^{O,F}(s_3) \\
\mathrm{val}_\sigma^{O,F}(n' := n' + 1) &= \{\emptyset \triangleright \langle\sigma\rangle \curvearrowright \sigma[n' \mapsto \mathrm{val}_\sigma^{O,F}(n' + 1)] \cdot \mathrm{K}(\text{🍶})\} \\
\mathrm{val}_\sigma^{O,F}(\texttt{return}\ n') &= \{\emptyset \triangleright \mathit{compEv}_\sigma(F, \mathrm{val}_\sigma^{O,F}(n')) \cdot \mathrm{K}(\text{🍶})\}
\end{aligned}$$

We now continue with the computation of a global concrete trace. Let I_P be the initial state of P. We consider the state where the statements in lines 8–9 have already been executed. At this point, the concrete trace is $sh = \langle I_P\rangle \curvearrowright [a' \mapsto 0, x' \mapsto 0, f' \mapsto 0, y' \mapsto 0] \curvearrowright \sigma$ with $\sigma = [a' \mapsto 1, x' \mapsto 0, f' \mapsto 0, y' \mapsto 0]$, and the mapping Σ is $[o_{main} \mapsto \{\mathrm{K}^{f_{main}}(s_{10})\}]$. Only Rule (58) is applicable:

$$\frac{\begin{array}{c}\Sigma[o_{main}] = \{\mathrm{K}^{f_{main}}(s_{10})\} \quad \sigma = \mathrm{last}(sh)\\ \emptyset \triangleright newEv_{\sigma}^{\{X\}}(X,\_) \curvearrowright \sigma[x' \mapsto X, X \mapsto *] \cdot \mathrm{K}(s_{11}) \in \mathrm{val}_{\sigma}^{o_{main},f_{main}}(s_{10})\\ \rho = [X \mapsto o_m] \quad \emptyset \text{ consistent} \quad wf(sh')\end{array}}{sh, \Sigma \rightarrow sh', \Sigma[o_{main} \mapsto \{\mathrm{K}^{f_{main}}(s_{11})\}]} \tag{60}$$

where sh' is the concrete trace

$$\begin{array}{rl} & sh ** \rho(newEv_{\sigma}^{\{X\}}(X,\varepsilon) \curvearrowright \sigma[x' \mapsto X, X \mapsto *])^{o_{main}}\\ = & sh ** \rho(\langle\sigma\rangle \curvearrowright newEv_{\sigma}^{\{X\}}(X,\varepsilon) \curvearrowright \sigma[X \mapsto *] \curvearrowright \sigma[x' \mapsto X, X \mapsto *])^{o_{main}}\\ = & \langle I_P\rangle \curvearrowright \ldots \curvearrowright \sigma \curvearrowright newEv^{o_{main}}(o_m,\varepsilon) \curvearrowright \sigma \curvearrowright \sigma' \end{array}$$

with $\sigma' = \sigma[x' \mapsto o_m] = [a' \mapsto 1, x' \mapsto o_m, f' \mapsto 0, y' \mapsto 0]$.[20] We must continue with s_{11} by applying Rule (58):

$$\frac{\begin{array}{c}\Sigma[o_{main}] = \{\mathrm{K}^{f_{main}}(s_{11})\} \quad \sigma' = \mathrm{last}(sh')\\ \rho = [F' \mapsto f_m] \quad \emptyset \text{ consistent} \quad wf(sh'')\\ \emptyset \triangleright invEv_{\sigma'}^{\{F'\}}(1,o_m,m,F') \curvearrowright \sigma'[f' \mapsto F', F' \mapsto *] \cdot \mathrm{K}(\mathtt{await}\ f'?; s_{13})\\ \in \mathrm{val}_{\sigma'}^{o_{main},f_{main}}(s_{11})\end{array}}{sh', \Sigma \rightarrow sh'', \Sigma[o_{main} \mapsto \{\mathrm{K}^{f_{main}}(\mathtt{await}\ f'?; s_{13})\}]} \tag{61}$$

where sh'' is the concrete trace

$$\begin{array}{rl} & sh' ** \rho(invEv_{\sigma'}^{\{F'\}}(1,o_m,m,F') \curvearrowright \sigma'[f' \mapsto F', F' \mapsto *])^{o_{main}}\\ = & \langle I_P\rangle \curvearrowright \ldots \curvearrowright \sigma' \curvearrowright invEv^{o_{main}}(1,o_m,m,f_m) \curvearrowright \sigma' \curvearrowright \sigma'' \end{array}$$

with $\sigma'' = \sigma'[f' \mapsto f_m] = [a' \mapsto 1, x' \mapsto o_m, f' \mapsto f_m, y' \mapsto 0]$.

At this point, one could in principle choose to continue with the **await** statement in *main* with Rule (58) or to start executing method m on object o_m with Rule (59). However, Rule (58) is not applicable, because it does not result in a well-formed trace before method m returns. Therefore, we proceed with the second option:

$$\frac{\begin{array}{c}\Sigma[o_m] = \emptyset \quad \mathrm{lookup}(m,\mathcal{G}) = m(n)\ sc \quad \sigma'' = \mathrm{last}(sh'') \quad f_m \in FId\\ \emptyset \triangleright invREv_{\sigma''}^{\{N,X',F''\}}(N,X',m,F'')\\ \curvearrowright \sigma''[n' \mapsto N, N \mapsto *, X' \mapsto *, F'' \mapsto *] \cdot \mathrm{K}(\{\ s_3\ \}) \in \mathrm{val}_{\sigma''}^{o_m,f_m}(m(n)\ sc)\\ \rho = [N \mapsto 1, X' \mapsto o_{main}, F'' \mapsto f_m] \quad \emptyset \text{ consistent} \quad wf(sh_m)\end{array}}{sh'', \Sigma \rightarrow sh_m, \Sigma[o_m \mapsto \{\mathrm{K}^{f_m}(\{\ s_3\ \})\}]} \tag{62}$$

where the mapping Σ is updated to $[o_{main} \mapsto \{\mathrm{K}^{f_{main}}(\mathtt{await}\ f?; s_{13})\}, o_m \mapsto \{\mathrm{K}^{f_m}(\{\ s_3\ \})\}]$, and sh_m is the concrete trace

$$\begin{array}{rl} & sh'' ** \rho(invREv_{\sigma''}^{\{N,X',m,F''\}}(N,X',m,F'') \curvearrowright \sigma''[n' \mapsto N, N \mapsto *, X' \mapsto *, F'' \mapsto *])^{o_m}\\ = & \langle I_P\rangle \curvearrowright \ldots \curvearrowright \sigma'' \curvearrowright invREv^{o_m}(1,o_{main},m,f_m) \curvearrowright \sigma'' \curvearrowright \sigma_m \end{array}$$

with $\sigma_m = \sigma''[n' \mapsto 1]$.

We have to continue with $\{\ s_3\ \}$ from the method m. Executing $n' := n' + 1$ in s_3 updates σ_m to $\sigma'_m = \sigma_m[n' \mapsto \mathrm{val}_{\sigma_m}^{o_m,f_m}(n'+1)] = [a' \mapsto 1, x' \mapsto o_m, f' \mapsto f_m, y' \mapsto 0, n' \mapsto 2]$, which extends the concrete trace to sh'_m as follows:

$$\langle I_P\rangle \curvearrowright \ldots \curvearrowright \sigma_m \curvearrowright \sigma'_m\ .$$

We proceed with the **return** statement in method m by applying Rule (58) once again:

[20]We simplify the states in the concrete trace by removing the symbolic variables once they are instantiated.

$$\frac{\begin{array}{c}\Sigma[o_m] = \{\mathrm{K}^{f_m}(\mathtt{return}\ n')\} \quad \sigma'_m = \mathrm{last}(sh'_m) \\ \emptyset \triangleright compEv_{\sigma'_m}(f_m, 2) \cdot \mathrm{K}(\Box) \in \mathrm{val}^{o_m,f_m}_{\sigma'_m}(\mathtt{return}\ n') \quad \rho = [] \quad \emptyset \text{ consistent} \quad wf(sh''_m)\end{array}}{sh'_m, \Sigma \longrightarrow sh''_m, \Sigma[o_m \mapsto \emptyset]} \tag{63}$$

where $\Sigma[o_m \mapsto \emptyset]$ is the result of simplifying $\Sigma[o_m \mapsto \emptyset + \mathrm{K}^{f_m}(\Box)]$. The resulting concrete trace sh''_m is:

$$\begin{array}{rl} & sh'_m ** \rho(compEv_{\sigma'_m}(f_m, 2))^{o_m} \\ = & \langle I_P \rangle \curvearrowright \ldots \sigma'_m \curvearrowright compEv^{o_m}(f_m, 2) \curvearrowright \sigma'_m \ . \end{array}$$

The mapping Σ is now updated to $[o_{main} \mapsto \{\mathrm{K}^{f_{main}}(\mathtt{await}\ f'?; s_{13})\}, o_m \mapsto \emptyset]$. Executing the **await** statement in *main* at this point produces a well-formed trace by Rule (58):

$$\frac{\begin{array}{c}\Sigma[o_{main}] = \{\mathrm{K}^{f_{main}}(\mathtt{await}\ f'?; s_{13})\} \quad \sigma'_m = \mathrm{last}(sh''_m) \\ \emptyset \triangleright compREv^{\{V\}}_{\sigma'_m}(f_m, V) \curvearrowright \sigma'_m[V \mapsto *] \cdot \mathrm{K}(s_{13}) \in \mathrm{val}^{o_{main},f_{main}}_{\sigma'_m}(\mathtt{await}\ f'?; s_{13}) \\ \rho = [V \mapsto 2] \quad \emptyset \text{ consistent} \quad wf(sh''')\end{array}}{sh''_m, \Sigma \longrightarrow sh''', \Sigma[o_{main} \mapsto \{\mathrm{K}^{f_{main}}(s_{13})\}]} \tag{64}$$

with the concrete trace sh''':

$$\begin{array}{rl} & sh''_m ** \rho(compREv^{\{V\}}_{\sigma'_m}(f_m, V) \curvearrowright \sigma'_m[V \mapsto *])^{o_{main}} \\ = & \langle I_P \rangle \curvearrowright \ldots \curvearrowright \sigma'_m \curvearrowright compREv^{o_{main}}(f_m, 2) \curvearrowright \sigma'_m \curvearrowright \sigma'_m \ . \end{array}$$

The final state σ'_m is actually a simplification of $\sigma'_m[V \mapsto 2]$ by removing the concretised symbolic variable V and thus identical to σ'_m.

Finally, we execute the **get** statement, again by Rule (58):

$$\frac{\begin{array}{c}\Sigma[o_{main}] = \{\mathrm{K}^{f_{main}}(s_{13})\} \quad \sigma'_m = \mathrm{last}(sh''') \\ \emptyset \triangleright compREv^{\{V'\}}_{\sigma'_m}(\mathrm{val}^{o_{main},f_{main}}_{\sigma'_m}(f), V') \curvearrowright \sigma'_m[y' \mapsto V', V' \mapsto *] \cdot \mathrm{K}(\Box) \in \mathrm{val}^{o_{main},f_{main}}_{\sigma'_m}(s_{13}) \\ \rho = [V' \mapsto 2] \quad \emptyset \text{ consistent} \quad wf(sh_P)\end{array}}{sh''', \Sigma \longrightarrow sh_P, \Sigma[o_{main} \mapsto \emptyset]} \tag{65}$$

which results the mapping $\Sigma = [o_{main} \mapsto \emptyset, o_m \mapsto \emptyset]$, and produces the concrete trace sh_P:

$$\begin{array}{rl} & sh''' ** \rho(compREv^{\{V'\}}_{\sigma'_m}(\mathrm{val}^{o_{main},f_{main}}_{\sigma'_m}(f)\}, V') \curvearrowright \sigma'_m[y' \mapsto V', V' \mapsto *])^{o_{main}} \\ = & \langle I_P \rangle \curvearrowright [a' \mapsto 0, x' \mapsto 0, f' \mapsto 0, y' \mapsto 0] \curvearrowright \sigma \curvearrowright newEv^{o_{main}}(o_m, \_) \curvearrowright \sigma \curvearrowright \sigma' \\ & \curvearrowright invEv^{o_{main}}(1, o_m, m, f_m) \curvearrowright \sigma' \curvearrowright \sigma'' \curvearrowright invREv^{o_m}(1, o_{main}, m, f_m) \curvearrowright \sigma'' \curvearrowright \sigma_m \curvearrowright \sigma'_m \\ & \curvearrowright compEv^{o_m}(f_m, 2) \curvearrowright \sigma'_m \curvearrowright compREv^{o_{main}}(f_m, 2) \curvearrowright \sigma'_m \curvearrowright \sigma'_m \\ & \curvearrowright compREv^{o_{main}}(f_m, 2) \curvearrowright \sigma'_m \curvearrowright \sigma_P \end{array}$$

with $\sigma_P = [a' \mapsto 1, x' \mapsto o_m, f' \mapsto f_m, y' \mapsto 2, n' \mapsto 2]$.

7.3.5 ABS. ABS is an actor-based executable modeling language [2, 40, 58] that falls in the class of Active Object languages [27]. ABS is closely related to the language in Figure 10. Before we make this relation precise, we mention the main features of ABS that have *not* been discussed in this paper:

Functional Expressions ABS lets the user declare algebraic datatypes and provides a pure, functional language with pattern matching over these. An evaluation semantics for such a language is standard [69]. It is easily incorporated into states and their evaluation.

Interfaces ABS supports multiple implementations of interfaces, but no inheritance or dynamic dispatch. It is sufficient to equip the lookup table $\mathcal{G}$ with suitable selectors.

Modules, Traits ABS has a simple module system. Modules do not have an operational semantics, but manage the name space. One can remove them by replacing relative with absolute names. ABS supports code reuse via traits: sets of method declarations that can be added to a class via a *uses* clause. Like modules, traits can be assumed to have been resolved.

Error Handling ABS can throw and catch exceptions. The semantics of the corresponding statements is a combination of the case distinction rule (8) and the local jump rule (45). The design of the corresponding local evaluation rules is left as an exercise.

Otherwise, ABS is identical to the Active Object language discussed in the present section. Specifically, ABS tasks are *atomic by default*. Their execution is only suspended explicitly, either at the end of a method, or with a **suspend** statement (equivalent to "**await** true"), or by **await** g.

7.4 A Unifying Perspective on Trace Composition Rules

As mentioned above, our semantics is *modular* in the sense that the *local* evaluation rules are identical (up to addition of parameters) for all statements common to the considered languages. What—necessarily—must change are the trace composition rules, because these implement different parallel and concurrent composition patterns. But it is possible to provide a *unifying* set of meta composition rules that can be instantiated to each previous incarnation and that sheds light on the commonalities. We refrained to work with these general rules from the start, because that would have been a major obstacle to readability.

We begin with a second look at the simplest composition rule (14) in the light of the various extensions presented in the preceding sections. We repeat the rule here for convenience:

$$\frac{\sigma = \mathrm{last}(sh) \quad pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma(s) \quad pc \text{ consistent}}{sh, \mathrm{K}(s) \rightarrow sh ** \tau, \mathrm{K}(s')}$$

The following aspects of this rule were generalized: (i) the result of $\mathrm{val}_\sigma(s)$ in general is an *abstract* trace that needs to be concretized; (ii) the extension of sh with the concretized τ must be *well-formed*; (iii) in a multi-processor setting, the events and states in traces are *tagged* with the processor they relate to; (iv) evaluation of s is *parameterized* with the current processor and the destiny; (v) instead of a single continuation $\mathrm{K}(s)$, there is a queue or process pool of pending tasks which must be updated in the target configuration. Unifying meta rules must handle all of these aspects simultaneously.

We unify task queues and pools with a function pool : proc $\rightarrow$ queue $\times$ queue that maps each processor p to a pair $\langle w \mid a \rangle$ of waiting tasks w and active tasks a with the conventions of Sect. 7.3.2. Waiting tasks are merely needed for the semantics of active objects, otherwise the argument is ignored. For both waiting and active task queues we define an *abstract* union operation + that is instantiated to sets or multisets as required. Each element of a task queue is a continuation with an optional destiny parameter.

The main challenge in the design of uniform composition rules is to unify the two scheduling modes used in the languages under consideration: interleaving versus run-to-yield. In particular, the composition rule that is responsible to execute method calls, must be able either to fire at any time or else only when no active task is executed, respectively. To achieve this, active tasks a may contain a special token "enabled" that serves as a flag, controlling when a rule can be fired.

Processors p are instantiated to either processors or objects as needed, thus unifying the Ω's and Σ's in rule (42) and after. With these conventions in place, the following two rules unify the progress and the method execution rules, respectively:

$$\frac{\begin{array}{c}\mathrm{pool}(p) = \langle w \mid a + \{\mathrm{K}^f(s)\}\rangle \qquad \sigma = \mathrm{last}(sh) \\ pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma^{p,f}(s) \qquad \rho \text{ concretizes } \tau \qquad \rho(pc) \text{ consistent} \qquad wf(sh \ast\ast \rho(\tau)^p)\end{array}}{sh, \mathrm{pool} \to sh \ast\ast \rho(\tau)^p,\ \mathrm{pool}[p \mapsto \langle w \mid a + \{\mathrm{K}^f(s')\}\rangle]} \tag{66}$$

$$\frac{\begin{array}{c}\mathrm{pool}(p) = \langle w \mid a\rangle \quad a = \{\} \text{ or enabled} \in a \quad \mathrm{lookup}(m, \mathcal{G}) = m(\overline{x})\ sc \quad \sigma = \mathrm{last}(sh) \\ pc \triangleright \tau \cdot \mathrm{K}(s') \in \mathrm{val}_\sigma^{p,f}(m(\overline{x})\ sc) \quad \rho \text{ concretizes } \tau \quad \rho(pc) \text{ consistent} \quad wf(sh \ast\ast \rho(\tau)^p)\end{array}}{sh, \mathrm{pool} \to sh \ast\ast \rho(\tau)^p,\ \mathrm{pool}[p \mapsto \langle w \mid a + \{\mathrm{K}^f(s')\}\rangle]} \tag{67}$$

For languages with suspension (Sect. 7.3) we need two more rules that move waiting tasks back and forth between the waiting and active task queues. These rules are not required for languages without suspension statements and cannot fire in the latter case.

$$\frac{\mathrm{pool}(p) = \langle w \mid a + \{\mathrm{K}^f(\mathbf{await}\, g;\ s)\}\rangle}{sh, \mathrm{pool} \to sh, \mathrm{pool}[p \mapsto \langle w + \{\mathrm{K}^f(\mathbf{await}\, g;\ s)\} \mid a\rangle]} \tag{68}$$

$$\frac{\mathrm{pool}(p) = \langle w + \{\mathrm{K}^f(\mathbf{await}\, g;\ s)\} \mid \{\}\rangle}{sh, \mathrm{pool} \to sh, \mathrm{pool}[p \mapsto \langle w \mid \{\mathrm{K}^f(\mathbf{await}\, g;\ s)\}\rangle]} \tag{69}$$

Together, the four rules above simulate rules (58)–(59), provided that f ranges over futures, p over objects, where non-existing objects, i.e. those not occurring in sh, are initialized $\mathrm{pool}(p) = \langle\{\} \mid \{\}\rangle$. Observe that the constant "enabled" does not occur anywhere, so the method execution rule can only fire when $a = \{\}$.

To obtain rules (50)–(51), fix f to an arbitrary concrete value which is ignored during local evaluation. From now on, there is no **await** statement in the language, so w is always empty and can be ignored in the following. Since sc has **atomic** shape, method bodies run to completion when the condition of the method execution rule is satisfied.

To obtain rules (42)–(43), fix f to an arbitrary concrete value, while both f and p are ignored during local evaluation. The method lookup function now yields $sc = \{s\}$. To instantiate the meta rules correctly, it is sufficient to initialize $\mathrm{pool}(p) = \langle\{\} \mid \{\mathrm{enabled}\}\rangle$.

Rule (37) is obtained by additionally fixing p to an arbitrary concrete value p_0. Task queue a (minus "enabled") corresponds to q in rule (37). Languages, whose local evaluation rules do not produce symbolic variables, yield unconditioned concrete traces. In this case, the ρ in rule (37) is empty, the second and third premise are always true and can be omitted, which gives an alternative concrete rule to (31).

Finally, when no calls are present in the language, then the call rule is inapplicable. Also $\mathrm{pool}(p_0) = \langle\{\} \mid \{\mathrm{K}^f(s)\}\rangle$, which specializes the abstract progress rule to rule (28) and, for concrete local traces to rule (14).

8 APPLICATIONS

We review application scenarios, where the LAGC semantics has *already* been shown to be useful, while in Section 10.1 we mention further areas, where we expect it to become so. All of the applications of the LAGC semantic framework discussed below rely crucially on its properties and could not easily have been realized with one of the existing semantics.

The program logic introduced in Section 4 is a proof of concept and kept at a deliberately abstract level, because it serves as an illustration for the LAGC concept and it is not the main point of the present paper. In [20] a similar program logic is instantiated to a directly implementable system and the specification language of Section 4, which is based on first-order pre-/postconditions, is generalized to a *trace logic*. This makes it possible to specify the structure of recursive calls and, in general, any kind of event occurring *during* execution of a program. The resulting deductive verification approach generalizes big-step procedure contracts [66] to trace properties. The LAGC semantics aligns perfectly with the trace-based specification logic and permits a concise soundness proof.

The distinction between local and composition rules permits to clearly identify and to understand global scheduling and global synchronization points in the semantics of languages that model time and resources (e.g., [59, 86]). The LAGC framework provides a fully modular semantics in these cases [84].

The separation of concerns in the LAGC semantics between rules that generate computation states on one hand and the scheduling rules on the other, makes it possible to characterize fairness *constructively* at a semantic level. Even though all traces in LAGC are finite, semantics of non-terminating programs are obtained by taking the limit over a set of finite (increasing) traces. This make it possible to reason on fair scheduling in the context of LAGC [41]. There it is shows that rules (58)–(59) can be refined so they become deterministic and constitute a provably fair scheduler for a version of the active object language in Section 7.3. This is in contrast to most other approaches to characterize fairness, where fairness is either defined abstractly in terms of transition systems (for example, [80]) or via an external scheduler at system level [26]. Those approaches cannot express fairness directly within the semantics. In contrast, in the LAGC semantics we are able to formulate fair scheduling rules as an extension of the standard semantics. In addition, the LAGC semantics permits to define a provably fair and directly implementable scheduler. In this setting, the LAGC semantics revealed its strength compared to the alternatives, where reasoning about fairness is notoriously difficult: Either because the fair semantics is significantly modified and hard to compare with the original one—this happens with a naive implementation of a scheduler in SOS; or, because reasoning on possible traces and fairness becomes difficult—this happens with denotational semantics and other abstract approaches.

9 RELATED WORK

We position our work within denotational semantics. The discussion focuses on three different strands of work on trace semantics: Semantics based on execution traces, on communication traces, and on hybrid traces combining execution states and communication events. To first motivate the use of trace semantics, we start by drawing some lines from state-transformer semantics. We then consider related work on execution traces and communication traces. We finally discuss related work on hybrid traces, combining these directions.

State-transformer semantics. State-transformer semantics explain a program statement as a transition from one program state to another [10, 51]. This style of semantics elegantly hides intermediate states in the transition, thereby providing an abstraction from the inner, intermediate states visited in the actual execution of statements. State-transformer semantics is fully abstract for sequential while-programs, resulting in a compositional semantics with respect to, for example, partial correctness behavior. However, for parallel shared-variable languages the final outcome of a program may depend on the scheduling of the different atomic sections, leading to non-deterministic behavior. For a parallel statement, the state transformer that results from the parallel composition of component statements cannot be determined from the transformer for these statements alone [17, 71]. Hence, state-transformer semantics for parallel languages are not compositional; it is necessary to capture the intermediate states at which interleaving may occur. Mosses observes [71] that by capturing these intermediate states as so-called *resumptions* [46], the resulting denotational semantics corresponds much more closely to operational semantics [78].

Early work on verification of concurrent systems extended state-transformer semantics with additional side-conditions, and is as such non-compositional; for example, interference freedom tests were used for shared variable concurrency [76] and cooperation tests for synchronous message passing [11]. Compositional approaches were introduced for shared variables in the form of rely-guarantee [60] and for synchronous message passing in the form of assumption-commitment [68]. Extending these principles for compositional verification, object invariants can be used to achieve modularity (for example, [55]).

Execution traces. The use of execution traces to describe program behavior can be motivated by the need for additional structure in the semantics of statements, such that parallel statements can be described compositionally from the semantics of their components. Brookes developed a trace semantics for a shared-variable parallel language based on *transition traces* [17]. In his work, a command is described by a sequence of state transformers reflecting the different atomic blocks in the execution of the command, but abstracting from the inner states of these blocks. Transition traces are thus more abstract than resumptions that are discussed above. The semantics of parallel commands then corresponds to the union of these segments, obtained by stitching together the state transitions corresponding to the different atomic sections in different ways. Brookes shows that this semantics, which can be realised at different levels of granularity, is fully abstract. In fact, we considered this solution in our work, but decided against it because it constructs infinitely large mathematical objects in order to include all possible states in which the next atomic block can start. Instead, we opted for continuation markers, to be resolved in the composition, as well as symbolic traces that are concretised on demand during the composition phase. This ensures that all semantic elements are finite, which makes the LAGC semantics easy and effective to compute.

Communication traces. Communication traces first appeared in the object-oriented setting [25] and then for CSP [52]. Soundararajan developed an axiomatic proof system for CSP using histories and projections [82], which was compositional and removed the need for cooperation tests. Zwiers developed the first sound and complete proof system using communication traces [89]. Jeffrey and Rathke introduced a semantics for object-oriented programs based on communication traces, and showed full abstraction for testing equivalence [56, 57]. Reasoning about asynchronous method calls and cooperative scheduling using histories was first done for Creol [36] and later adapted to Dynamic Logic [5]. Din introduced a proof system based on four communication events, significantly simplifying the proof rules [31], and extended the approach to futures [33, 34]. This four-event proof system, which forms the basis for KeY-ABS [30], was the starting point for the communication events used in our paper. This way, communication events can always be local to one semantic object, and their ordering is captured by well-formedness predicates. Compared to this line of work, we introduce continuation markers and locally symbolic traces. This allows us to constructively derive global traces, in contrast to, for example, Din's work which, building on Soundararajan's work, simply eliminates global traces which do not project correctly to the local trace sets of the components.

Hybrid traces. Brookes' *action traces* [18] bear some similarity to our work. He aims at denotational semantics using collecting semantics, explicitly represents divergence, synchronises communication using events, and captures parallel execution of two components by means of a so-called *mutex fairmerge* which ensures that both components get the chance to be executed. Action traces were used to develop a semantics for concurrent separation logic [16], where scheduling is based on access to shared resources with associated invariants and data races are exposed. Brookes' work elegantly develops a trace semantics for low-level programming mechanisms with lock resources. However, it does not cover the dynamic spawning of processes, procedure calls, method invocations, and similar topics covered in our work, which led to the locally abstract, globally concrete formulation of hybrid trace semantics. In previous work [32] we used *scheduling events* and well-formedness properties over scheduling events to capture all legal interleavings among concurrent objects at a granularity similar to that of action traces. In contrast, parallel execution in the present paper is based on a more fine-grained interleaving of processes, exploiting continuations, by means of different composition rules (for example, (58) and (59)) to capture global and local interleaving of processes. The approach of Brookes [18, 32] is easily expressible within our framework. We obtain the equivalent of the fairmerge operator by collecting all traces that can be constructed by the composition rules from a given concrete initial state.

In trace semantics, the composition of traces from parallel executions can be formalised in different ways; e.g., as continuation semantics, erasure semantics or collecting semantics. We have opted for the first approach and captured the interleaved execution by means of local continuations, such that the global trace is obtained by gradually unfolding traces that correspond to local execution by means of continuations. Kamburjan has explored erasure semantics in a hybrid trace semantics developed as a foundation for behavioral program logic [61, 62]; this work uses an explicit representation of the heap which is exploited for composition by introducing a fresh symbolic heap as a local placeholder for execution in other objects at the scheduling points. The introduction of fresh symbolic states for merging local behaviors can equivalently be replaced by a set collecting all possible concrete states after the scheduling point, such that the composition of local behaviors can be done by selecting the appropriate, compatible concrete traces rather than by instantiating a symbolic trace. This approach has been studied in the context of hybrid trace semantics in [63].

The interplay between local and global views, which promotes a separation of concern between local artefacts and their synchronisation, lies at the heart of choreography [8, 24, 50, 85]: Choreographies describe global protocols, which can be seen as scheduling patterns for interactions between processes. A choreography captures a global view of an interaction protocol which, if well-formed, can be decomposed into local processes (or types). The semantics of choreography languages typically define a so-called endpoint projection, which generates local programs that behave according to the global protocol. In contrast to the local projections of choreographies, which start from global traces and decompose these into local programs, LAGC semantics starts from local symbolic trace denotations of local programs, which are composed into global traces using global rules and concretisation.

Parameterised labelled transition systems with queues have been used in ASP/ProActive [6] to model communication structures for interaction with futures in a fine-grained, operational manner. In contrast, our work with traces allows futures to be abstracted into communication events and well-formedness conditions. We are unaware of previous work on programming language semantics that captures different communication patterns as well-formedness constraints over trace semantics; these constraints are in fact *orthogonal* to the building blocks of the trace semantics and allow to study language behavior ranging over architectures that support different forms of communication within a single semantic framework.

Interaction trees [87] have been used to encode the interaction with the environment in denotational semantics for imperative programs. An interaction tree constitutes a form of trace that expresses the effects of the program but abstract away from states. The authors provide co-inductive principles for reasoning about diverging programs, which is more precise than our current handling of infinite traces, but their approach is restricted to sequential programs. Choice trees [22] extend interaction trees to address concurrency, focusing on a Coq formalisation. The concrete semantics in choice trees is given by successive interpretations whereas LAGC semantics fully concretises symbolic trace segments in the global rules. We believe that our approach is complementary to choice trees and better adapted to logical reasoning about program properties.

The pure trace-based proof system of ABS [33, 34] requires strong hiding of local state: the state of other objects can only be accessed through method calls, so shared state is internal and controlled by cooperative scheduling. Consequently, specifications can be purely local. More expressive specifications require significantly more complex proof systems, for example modifies clauses in Boogie [55], fractional permissions [48] in Chalice [65], or dynamic frames in KeY [72]. To specify fully abstract interface behavior these systems need to simulate histories in an ad hoc manner, see [55, Figure 1]. A combination of permission-based separation logic [7] and histories has recently been proposed for modular reasoning about multi-threaded concurrency [88]. The motivation for our work stems from our aim to devise compositional proof systems to verify protocol-like behavior for object-oriented languages [32] by combining communication histories for ABS with the trace modality formulas of Nakata et al. [19, 73].

10 FUTURE WORK AND CONCLUSION

10.1 Future Work

The original motivation for the present work was to create a modular semantics that aligns well with the kind of program logics used in deductive verification [42]. In Sections 4 and 5 we reached this goal by providing a program logic and calculus for the shared variable language and by giving a compact soundness proof (Thm. 4.6) for the sequential fragment. This result can and should be extended to a state-of-art calculus for the language in Section 7.3 [63].

An obvious direction for future work is to fully mechanize the LAGC semantics in a proof assistant [15, 74]. All definitions and theorems from Sections 2, 3, and 5 have been mechanized and proven[21] in Isabelle/HOL. The mechanization is *executable* in the sense that all traces in the examples of the mentioned sections can be generated automatically from the HOL theories. We plan to extend the Isabelle/HOL mechanization to cover also Sections 6, 7.2, and 7.3.

In Sections 5–7 we showed that a wide variety of parallel programming concepts can be formalised in LAGC style with relative ease. It would be interesting to explore how far this can be carried. On the one hand, one could try to formalise the semantics of a major programming language, such as Java or C. On the other hand, one could try to apply the LAGC framework to weakly consistent memory models [3] or to the target the language of concurrent separation logic [16]. A further possible extension concerns programs with non-terminating, atomic segments, see Section 5.2. A mechanized, executable LAGC semantics opens the possibility of *early prototyping* new semantics of concurrent and distributed languages with relative ease.

We rely heavily on well-formedness of traces, but we did not discuss how to define properties on possibly infinite sets of infinite traces with events, for example, notions related to fairness. This is a complementary problem to trace generation, and not the focus of the present paper.

Our approach makes composition of local rules easy, and extension of the language straightforward. Global trace composition on the contrary relies on well-formedness criteria for the whole trace; even if this criteria is defined modularly, it is not compositional in the sense that a trace can become invalid by extension of the criteria or parallel composition. This is often unavoidable because the semantics of many concurrent models like CCS and π-calculus are by nature very much sensitive to the execution context. However, depending on the language and the form of concurrency, different compromises could be found, defining a greater part of the concurrent semantics in a symbolic and compositional way. For example our semantics of futures is similar to message passing but one could probably specify new composition tools that better take into account the single-assignment property ensured by futures.

[21] See https://gitlab.com/Niklas_Heidler/mechanization-of-lagc-semantics-in-isabelle and [45].

10.2 Conclusion

The semantics of concurrent programming languages is inherently a technically demanding subject. In the best case, a formal semantics can illuminate the design space created by the choice of different concurrency concepts, it shows the consequences of these choices, and it makes different version comparable. This is only possible in a semantic framework that enforces uniform and modular definitions, and it is what we strove to achieve with LAGC: the central design decision is to strictly separate local, sequential computations and their parallel composition. Also, we decided to render local evaluations abstract. In this way, one and the same *schematic* semantic evaluation rule can be re-used for any initial state, executing processor, and destiny of the result. While abstract local rules are not a theoretical necessity, they drastically simplify the complexity of definitions.

A central technical problem to address in a locally-globally divided setup is to ensure that enough context information is available when composing concurrent behavior. Instead of computing *all* possible states, in which an atomic segment can continue, we pass the remaining code to be executed as a continuation. Again, this constitutes a considerable technical simplification compared to the former option. For example, in Section 7.3 we characterised the behavior of active objects concisely with a suitable definition of a continuation pool.

But mere continuation is not sufficient: one needs to orchestrate different local evaluations within a global trace, for example, to ensure a method is called before it is being executed. This is achieved by suitable events emitted during local evaluation. Orchestration of local computations by events leads to a further separation of concerns: many concurrency models can be characterised in a declarative manner by well-formedness of events inside traces, as shown in Section 6.2.

We believe the achieved separation of concerns, locally abstract evaluation—trace composition with a continuation pool—orchestration of computations by well-formedness, provides a flexible and usable semantic framework concurrent languages can be formalises and compares. It is close to modern deductive verification calculi and could even be fully mechanised.

Acknowledgment

We would like to thank the following people for carefully reading drafts of this paper and for detecting several omissions, errors, typos, and inaccuracies: Lukas Grätz, Dilian Gurov, Niklas Heidler, Eduard Kamburjan, and Marco Scaletta. We also would like to thank the anonymous reviewers for their valuable and constructive feedback that led to numerous improvements of the text.

REFERENCES

[1] Jean-Raymond Abrial. 1996. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, Cambridge, UK.

[2] ABS Development Team 2021. *ABS Documentation* (1.9.3 ed.). ABS Development Team. https://abs-models.org/manual.

[3] Sarita V. Adve and Mark D. Hill. 1993. A Unified Formalization of Four Shared-Memory Models. *IEEE Trans. Parallel Distributed Syst.* 4, 6 (1993), 613–624. https://doi.org/10.1109/71.242161

[4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice.* LNCS, Vol. 10001. Springer, Cham, Switzerland.

[5] Wolfgang Ahrendt and Maximilian Dylla. 2012. A system for compositional verification of asynchronous objects. *Science of Computer Programming* 77, 12 (2012), 1289–1309.

[6] Rabéa Ameur-Boulifa, Ludovic Henrio, Oleksandra Kulankhina, Eric Madelaine, and A. Savu. 2017. Behavioural semantics for asynchronous components. *J. Logical and Algebraic Methods in Programming* 89 (June 2017), 1–40.

[7] Afshin Amighi, Christian Haack, Marike Huisman, and Clément Hurlin. 2015. Permission-Based Separation Logic for Multithreaded Java Programs. *LMCS* 11 (2015), 1–66. Issue 1.

[8] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi,

Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

[9] Gregory Andrews. 1999. *Multithreading, parallel, and concurrent programming* (2nd ed.). Addison-Wesley, One Jacob Way, Reading, MA 01867-3999.

[10] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs.* Springer, Heidelberg. https://doi.org/10.1007/978-1-84882-745-5

[11] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. 1980. A Proof System for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems* 2, 3 (1980), 359–385.

[12] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. 2000. Formal System Development with KIV. In *Fundamental Approaches to Software Engineering (LNCS, Vol. 1783)*, Tom Maibaum (Ed.). Springer, Heidelberg, 363–366.

[13] Bernhard Beckert and Daniel Bruns. 2013. Dynamic Logic with Trace Semantics. In *Automated Deduction, 24th International Conference on Automated Deduction, Lake Placid, USA (LNCS, Vol. 7898)*, Maria Paola Bonacina (Ed.). Springer, Heidelberg, 315–329.

[14] Mordechai Ben-Ari. 2008. *Principles of the Spin Model Checker.* Springer, Heidelberg.

[15] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions.* Springer, Berlin Heidelberg.

[16] Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 227–270.

[17] Stephen D. Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Inf. Comput.* 127, 2 (1996), 145–163.

[18] Stephen D. Brookes. 2002. Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. In *Proc. 13th Intl. Conf. on Concurrency Theory (CONCUR 2002) (LNCS, Vol. 2421)*, Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera (Eds.). Springer, Berlin Heidelberg, 466–482.

[19] Richard Bubel, Crystal Chang Din, Reiner Hähnle, and Keiko Nakata. 2015. A Dynamic Logic with Traces and Coinduction. In *Intl. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, Wroclaw, Poland (LNCS, Vol. 9323)*, Hans De Nivelle (Ed.). Springer, Cham, Switzerland, 303–318.

[20] Richard Bubel, Dilian Gurov, Reiner Hähnle, and Marco Scaletta. 2022. Trace-based Deductive Verification. https://doi.org/10.48550/ARXIV.2211.09487

[21] Rod M. Burstall. 1974. Program proving as hand simulation with a little induction. In *Information Processing '74.* Elsevier/North-Holland, Amsterdam, 308–312.

[22] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proc. of the ACM on Programming Languages* 7, POPL (Jan. 2023), 1770–1800. https://doi.org/10.1145/3571254

[23] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. 1996. Synchronous, Asynchronous, and Causally Ordered Communication. *Distributed Computing* 9, 4 (1996), 173–191.

[24] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2022. A Formal Theory of Choreographic Programming. https://doi.org/10.48550/arXiv.2209.01886

[25] Ole-Johan Dahl. 1977. Can Program Proving be made Practical? In *Les Fondements de la Programmation*, M. Amirchahy and D. Néel (Eds.). Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, 57–114.

[26] Matthias Daum, Jan Dörrenbächer, and B. Wolff. 2009. Proving Fairness and Implementation Correctness of a Microkernel Scheduler. *Journal of Automated Reasoning* 42 (2009), 349–388.

[27] Frank de Boer, Crystal Chang Din, Kiko Fernandez-Reyes, Reiner Hähnle, Ludovic Henrio, Einar Broch Johnsen, Ehsan Khamespanah, Justine Rochas, Vlad Serbanescu, Marjan Sirjani, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *Comput. Surveys* 50, 5 (Oct. 2017), 76:1–76:39. Article 76.

[28] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. 2007. A Complete Guide to the Future. In *Proceedings of the 16th European Symposium on Programming, (ESOP 2007) (LNCS, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, Berlin, Heidelberg, 316–330. https://doi.org/10.1007/978-3-540-71316-6_22

[29] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

[30] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. 2015. KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS. In *Proc. 25th Intl. Conf. on Automated Deduction (CADE), Berlin, Germany (LNCS, Vol. 9195)*, Amy Felty and Aart Middeldorp (Eds.). Springer, Cham, 517–526.

[31] Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. 2012. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming* 81, 3 (2012), 227–256.

[32] Crystal Chang Din, Reiner Hähnle, Einar Broch Johnsen, Violet Ka I Pun, and Silvia Lizeth Tapia Tarifa. 2017. Locally Abstract, Globally Concrete Semantics of Concurrent Programming Languages. In *Proc. 26th Intl. Conf. on Automated Reasoning with Tableaux and Related Methods (LNCS, Vol. 10501)*, Cláudia Nalon and Renate Schmidt (Eds.). Springer,

Cham, Switzerland, 22–43.

[33] Crystal Chang Din and Olaf Owe. 2014. A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebraic Methods Program.* 83, 5–6 (2014), 360–383. https://doi.org/10.1016/j.jlamp.2014.03.003

[34] Crystal Chang Din and Olaf Owe. 2015. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing* 27, 3 (2015), 551–572.

[35] Crystal Chang Din, Silvia Lizeth Tapia Tarifa, Reiner Hähnle, and Einar Broch Johnsen. 2015. History-Based Specification and Verification of Scalable Concurrent and Distributed Systems. In *Proc. 17th International Conference on Formal Engineering Methods, ICFEM, Paris (LNCS, Vol. 9407)*, Michael Butler, Sylvain Cochon, and Fatiha Zaïdi (Eds.). Springer, Cham, Switzerland, 217–233.

[36] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. 2005. Verification of Concurrent Objects with Asynchronous Method Calls. In *Proc. IEEE Intl. Conference on Software Science, Technology & Engineering(SwSTE'05)*. IEEE Computer Society Press, Los Alamitos, CA, 141–150.

[37] Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

[38] Patrice Godefroid. 1997. Model Checking for Programming Languages using Verisoft. In *Proc. 24th Symp. on Principles of Programming Languages (POPL)*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM, New York, NY, 174–186. https://doi.org/10.1145/263699.263717

[39] Patrice Godefroid. 2012. Test Generation Using Symbolic Execution. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS, Hyderabad, India (LIPIcs, Vol. 18)*, Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan (Eds.). Leibniz-Zentrum fuer Informatik, Schloss Dagstuhl, 24–33.

[40] Reiner Hähnle. 2013. The Abstract Behavioral Specification Language: A Tutorial Introduction. In *International School on Formal Models for Components and Objects: Post Proceedings (LNCS, Vol. 7866)*, Marcello Bonsangue, Frank de Boer, Elena Giachino, and Reiner Hähnle (Eds.). Springer, Cham, Switzerland, 1–37.

[41] Reiner Hähnle and Ludovic Henrio. 2023. Provably Fair Cooperative Scheduling. *The Art, Science, and Engineering of Programming* 8, 2 (2023).

[42] Reiner Hähnle and Marieke Huisman. 2019. Deductive Verification: from Pen-and-Paper Proofs to Industrial Tools. In *Computing and Software Science: State of the Art and Perspectives*, Bernhard Steffen and Gerhard Woeginger (Eds.). LNCS, Vol. 10000. Springer, Cham, Switzerland, 345–373.

[43] Joseph Y. Halpern, Zohar Manna, and Ben C. Moszkowski. 1983. A Hardware Semantics Based on Temporal Intervals. In *Automata, Languages and Programming, 10th Colloquium, Barcelona, Spain (LNCS, Vol. 154)*, Josep DÃaz (Ed.). Springer, Berlin, Heidelberg, 278–291. https://doi.org/10.1007/BFb0036915

[44] David Harel, Dexter Kozen, and Jerzy Tiuryn. 2000. *Dynamic Logic.* MIT Press, Boston, MA, USA.

[45] Niklas Heidler. 2021. *Mechanization of LAGC Semantics in Isabelle.* Bachelor Thesis. Technical University of Darmstadt, Department of Computer Science. https://arxiv.org/abs/2202.08017

[46] Matthew Hennessy and Gordon D. Plotkin. 1979. Full Abstraction for a Simple Parallel Programming Language. In *Proc. 8th Symposium on the Mathematical Foundations of Computer Science (LNCS, Vol. 74)*, Jirí Becvár (Ed.). Springer, Heidelberg, 108–120.

[47] Ludovic Henrio, Florian Kammüller, and Muhammad Uzair Khan. 2010. A Framework for Reasoning on Component Composition. In *Proceedings of the 8th International Symposium on Formal Methods for Components and Objects (FMCO 2009) (Lecture Notes in Computer Science)*, Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel (Eds.). Springer, Heidelberg, 1–20. https://doi.org/10.1007/978-3-642-17071-3_1

[48] Stefan Heule, K. Rustan M. Leino, Peter Müller, and AlexanderJ. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). LNCS, Vol. 7737. Springer, Berlin Heidelberg, 315–334. https://doi.org/10.1007/978-3-642-35873-9_20

[49] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. 3rd Intl. Joint Conf. on Artificial Intelligence. Standford, CA, USA*, Nils J. Nilsson (Ed.). William Kaufmann, San Francisco, CA, USA, 235–245. http://ijcai.org/Proceedings/73/Papers/027B.pdf

[50] Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: Higher-Order Typed Functional Choreographies. *Proc. ACM Program. Lang.* 6, POPL, Article 23 (jan 2022), 27 pages. https://doi.org/10.1145/3498684

[51] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Comm. of the ACM* 12, 10 (Oct. 1969), 576–580, 583.

[52] C. A. R. Hoare. 1985. *Communicating Sequential Processes.* Prentice Hall, Upper Saddle River, NJ.

[53] Gerard J. Holzmann. 2003. *The SPIN Model Checker.* Pearson Education, Boston, MA, USA.

[54] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and Reasoning about Systems* (2nd ed.). Cambridge University Press, Cambridge, UK.

[55] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. 2005. Safe Concurrency for Aggregate Objects with Invariants. In *Third IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM 2005)*, Bernhard K. Aichernig and Bernhard Beckert (Eds.). IEEE Computer Society, Los Alamitos, CA, 137–147.

[56] Alan Jeffrey and Julian Rathke. 2005. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.* 338, 1-3 (2005), 17–63.

[57] Alan Jeffrey and Julian Rathke. 2005. Java Jr: Fully Abstract Trace Semantics for a Core Java Language. In *Proc. 14th European Symposium on Programming (ESOP 2005) (LNCS, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, Berlin Heidelberg, 423–438.

[58] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2011. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010) (LNCS, Vol. 6957)*, Bernhard K. Aichernig, Frank de Boer, and Marcello M. Bonsangue (Eds.). Springer, Heidelberg, 142–164.

[59] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. 2015. Integrating deployment architectures and resource consumption in timed object-oriented models. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 67–91. https://doi.org/10.1016/j.jlamp.2014.07.001

[60] Cliff B. Jones. l981. *Development Methods for Computer Programmes Including a Notion of Interference.* Ph. D. Dissertation. Oxford University, UK.

[61] Eduard Kamburjan. 2019. Behavioral Program Logic. In *Proc. 28th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2019) (LNCS, Vol. 11714)*, Serenella Cerrito and Andrei Popescu (Eds.). Springer, Cham, Switzerland, 391–408. https://doi.org/10.1007/978-3-030-29026-9_22

[62] Eduard Kamburjan. 2020. *Modular Verification of a Modular Specification: Behavioral Types as Program Logics.* Ph. D. Dissertation. Darmstadt University of Technology, Germany. http://tuprints.ulb.tu-darmstadt.de/11664/

[63] Eduard Kamburjan, Crystal Chang Din, Reiner Hähnle, and Einar Broch Johnsen. 2020. Behavioral Contracts for Cooperative Scheduling. In *Deductive Software Verification: Future Perspectives*, Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich (Eds.). LNCS, Vol. 12345. Springer, Cham, 85–121.

[64] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

[65] K. Rustan M. Leino, Peter Müller, and Jan Smans. 2009. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V*, Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri (Eds.). LNCS, Vol. 5705. Springer, Berlin Heidelberg, 195–222. https://doi.org/10.1007/978-3-642-03829-7_7

[66] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (Oct. 1992), 40–51.

[67] Robin Milner. 1989. *Communication and Concurrency.* Prentice-Hall, Inc., USA.

[68] Jayadev Misra and K. Mani Chandy. 1981. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering* 7, 4 (1981), 417–426.

[69] John C. Mitchell. 1996. *Foundations for programming languages.* MIT Press, Boston, MA, USA.

[70] Peter D. Mosses. 2004. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming* 60–61 (2004), 195–228. https://doi.org/10.1016/j.jlap.2004.03.008 Structural Operational Semantics.

[71] Peter D. Mosses. 2006. Formal Semantics of Programming Languages: — An Overview —. *Electron. Notes Theor. Comput. Sci.* 148, 1 (2006), 41–73. https://doi.org/10.1016/j.entcs.2005.12.012

[72] Wojciech Mostowski. 2020. From Explicit to Implicit Dynamic Frames in Concurrent Reasoning for Java. In *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich (Eds.). LNCS, Vol. 12345. Springer, Cham, 177–203. https://doi.org/10.1007/978-3-030-64354-6_7

[73] Keiko Nakata and Tarmo Uustalu. 2015. A Hoare logic for the coinductive trace-based big-step semantics of While. *Log. Methods Comput. Sci.* 11, 1 (2015), 1–32. https://doi.org/10.2168/LMCS-11(1:1)2015

[74] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS, Vol. 2283. Springer, Berlin Heidelberg.

[75] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science* 375, 1-3 (2007), 271–307.

[76] Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319–340.

[77] Rohit Parikh, Ashok K. Chandra, Joseph Y. Halpern, and Albert R. Meyer. 1985. Equations Between Regular Terms and an Application to Process Logic. *SIAM J. Comput.* 14, 4 (1985), 935–942. https://doi.org/10.1137/0214066

[78] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60–61 (2004), 17–139.

[79] Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus: A theory of mobile processes.* Cambridge University Press, Cambridge, UK.

[80] Roberto Segala. 1997. Quiescence, Fairness, Testing, and the Notion of Implementation. *Information and Computation* 138, 2 (1997), 194–210. https://doi.org/10.1006/inco.1997.2652

[81] A. Prasad Sistla and Edmund M. Clarke. 1985. The Complexity of Propositional Linear Temporal Logics. *J. ACM* 32, 3 (1985), 733–749.

[82] Neelam Soundararajan. 1984. Axiomatic Semantics of Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems* 6, 4 (1984), 647–662.

[83] Dominic Steinhöfel and Reiner Hähnle. 2020. The Trace Modality. In *2nd Intl. Workshop on Dynamic Logic: New Trends and Applications* (Porto, Portugal) *(LNCS, Vol. 12005)*, Alexandru Baltag and Luis S. Barbosa (Eds.). Springer, Cham, 124–140.

[84] Silvia Lizeth Tapia Tarifa. 2022. Locally Abstract Globally Concrete Semantics of Time and Resource Aware Active Objects. In *The Logic of Software. A Tasting Menu of Formal Methods (LNCS, Vol. 13360)*. Springer, Cham, 481–499. https://doi.org/10.1007/978-3-031-08166-8_23

[85] Emilio Tuosto and Roberto Guanciale. 2018. Semantics of global view of choreographies. *J. Log. Algebraic Methods Program.* 95 (2018), 17–40. https://doi.org/10.1016/j.jlamp.2017.11.002

[86] Gianluca Turin, Andrea Borgarelli, Simone Donetti, Ferruccio Damiani, Einar Broch Johnsen, and Silvia Lizeth Tapia Tarifa. 2023. Predicting resource consumption of Kubernetes container systems using resource models. *J. Syst. Softw.* 203 (2023), 111750. https://doi.org/10.1016/j.jss.2023.111750

[87] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan 2020), 1–32. https://doi.org/10.1145/3371119

[88] Marina Zaharieva-Stojanovski, Marieke Huisman, and Stefan Blom. 2014. Verifying Functional Behaviour of Concurrent Programs. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs* (Uppsala, Sweden) *(FTfJP'14)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2635631.2635849

[89] Job Zwiers. 1989. *Compositionality, Concurrency and Partial Correctness: Proof Theories for Networks of Processes, and Their Relationship*. LNCS, Vol. 321. Springer, Heidelberg.