INSTITUT FÜR INFORMATIK UND PRAKTISCHE MATHEMATIK

LEHRSTUHL FÜR SOFTWARETECHNOLOGIE

Verification for Java's Reentrant Multithreading Concept: Soundness and Completeness

Erika Ábrahám-Mumm
Frank S. de Boer
Willem-Paul de Roever
Martin Steffen

Bericht Nr. TR-ST-02-01
15 März 2002

CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

Verification for Java's Reentrant Multithreading Concept: Soundness and Completeness

March 15, 2002

Erika Ábrahám-Mumm[1], Frank S. de Boer[2],
Willem-Paul de Roever[1], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] Utrecht University, The Netherlands

Abstract. Besides the features of a class-based object-oriented language, *Java* integrates concurrency via its thread-classes, allowing for a *multithreaded* flow of control. The concurrency model includes *shared-variable* concurrency via instance variables, *coordination* via reentrant synchronization monitors, *synchronous message passing,* and dynamic *thread creation.*
To reason about multithreaded programs, we introduce in this paper an *assertional proof method* for safety properties for *Java*$_{MT}$ ("*Multi-Threaded Java*"), a small concurrent sublanguage of *Java*, covering the mentioned concurrency issues as well as the object-based core of *Java*, i.e., object creation, side effects, and aliasing, but leaving aside inheritance and subtyping. We show soundness and relative completeness of the proof method.

Table of Contents

1 Introduction

The semantical foundations of *Java* [17] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [4, 31, 14]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sublanguages (see e.g. [22, 36, 29]). This paper presents a proof system for *multithreaded* *Java* programs. Concentrating on the issues of concurrency, we introduce an abstract programming language $Java_{MT}$, a subset of *Java*, featuring dynamic object creation, method invocation, object references with aliasing, and specifically concurrency. Threads are the units of concurrency. They are created as instances of specific thread-classes and share the instance variables of objects.

As a mechanism of concurrency control, methods can be declared as *synchronized,* where synchronized methods within a single object are executed by different threads mutually exclusive. A call chain corresponding to the execution of a single thread can contain several invocations of synchronized methods within the same object. This corresponds to the notion of re-entrant monitors and eliminates the possibility that a single thread deadlocks itself on an object's synchronization barrier.

The assertional proof system for verifying safety properties of $Java_{MT}$ is formulated in terms of *proof outlines* [26], i.e., of programs augmented by auxiliary variables and with Hoare-style assertions [16, 20] associated with every control point.

The behavior of a $Java_{MT}$ program results from the concurrent execution of method bodies, that can interact by

- shared-variable concurrency,
- synchronous message passing for method calls, and
- object creation.

In order to capture these features in a modular way, the assertional logic and the proof system are formulated in two levels, a local and a global one. The local assertion language describes the internal object behavior. The global behavior, including the communication topology of the objects, is expressed in the global language. As in the Object Constraint Language (OCL) [37], properties of object-structures are described in terms of a navigation or dereferencing operator.

The local level treats internal computations affecting a single object, but excluding communication. The execution of a single method body in isolation is captured by standard *local correctness* conditions that show the inductiveness of the annotated method bodies.

To support a clean interface between internal and external behavior, $Java_{MT}$ does not allow qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneously execution within a single object, but not across object boundaries, and can therefore be handled on the local level, as well. A further healthy effect of disallowing references to external instance variables is that it reduces the potential of interference considerably, which means much less proof obligations generated by the proof system. The *interference freedom test* [26, 24] formulates the corresponding verification conditions. It has especially to accommodate for reentrant code and the specific synchronization mechanism.

Affecting more than one instance, synchronous message passing and object creation can be established locally only relative to assumptions about the communicated values. These assumptions are verified in the *cooperation test* on the global level. The communication can take place within a single object or between different objects. As these two cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules used in [8] and in [24] for CSP.

Overview This paper is organized as follows. Section 2 defines the syntax of $Java_{MT}$, Section 2.3 its operational semantics. After introducing the assertion language in Section 3, the main Section 4 presents the proof system. Soundness and completeness of the proof system is shown in Section 5. In Section 6, we discuss related and future work. The proofs of the results are included in the appendix.

2 The programming language $Java_{MT}$

In this section we introduce the language $Java_{MT}$ (*"Multi-Threaded Java"*). We start with highlighting the features of $Java_{MT}$ and its relationship to full *Java*, before formally defining its abstract syntax and semantics.

2.1 Introduction

$Java_{MT}$ is a multithreaded sublanguage of *Java*. Programs, as in *Java*, are given by a collection of classes containing instance variable and method declarations. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation,* i.e., synchronous message passing. As we focus on a proof system for the concurrency aspects of *Java*, all classes in $Java_{MT}$ are thread classes in the sense of *Java*: Each class contains a start-method that can be invoked only once for each object, resulting in a new thread of execution. The new thread starts to execute the start-method of the given object while the initiating thread continues its own execution.

As a mechanism of concurrency control, methods can be declared as *synchronized.* The execution of synchronized methods within a single object by different threads is mutually exclusive, whereas non-synchronized methods do not require such coordination. Note that in a single call chain recursive invocations of synchronized methods on the same object are allowed, as they are executed by the same thread. This corresponds to the notion of re-entrant monitors.

All programs are assumed to be well-typed, i.e., each method invoked on an object must be supported by the object, the types of the formal and actual parameters of the invocation must match, etc. As the static relationships between classes are orthogonal to multithreading aspects, we ignore in $Java_{MT}$ the issues of *inheritance*, and consequently subtyping, overriding, and late-binding. For simplicity, we neither allow method *overloading*, i.e., we require that each method name is assigned a unique list of formal parameter types and a return type. In short, being concerned with the verification of the run-time behavior, we assume a simple *monomorphic* type discipline for Java$_{MT}$.

2.2 Abstract syntax

Similar to *Java*, the language Java$_{MT}$ is strongly typed and supports class types and primitive, i.e., non-reference types. As built-in primitive types we restrict to integers and booleans, denoted by Int and Bool. Besides the built-in types, the set of user-definable types is given by a set of class names $\mathcal{C}$, with typical element c. Furthermore, the language allows pairs of type $t_1 \times t_2$ and sequences of type $\mathsf{list}\, t$. Side-effect expressions without a value, i.e., methods without a return value, will get the type Void. Thus the set of all types $\mathcal{T}$ with typical element t is given by the following abstract grammar:

$$t ::= \mathsf{Void} \mid \mathsf{Int} \mid \mathsf{Bool} \mid c \mid t \times t \mid \mathsf{list}\, t$$

For each type, the corresponding value domain is equipped with a standard set F of operators with typical element f. Each operator f has a unique type $t_1 \times \cdots \times t_n \to t$ and a fixed interpretation f, where constants are operators of zero arity. Apart from the standard repertoire of arithmetical and boolean operations, the set F of operators also contains operations on tuples and sequences like projection, concatenation, etc.

Since Java$_{MT}$ is strongly typed, all program constructs of the abstract syntax —variables, expressions, statements, methods, classes— are silently assumed to be well-typed. In other words, we work with a type-annotated abstract syntax where we omit the explicit mentioning of types when no confusion can arise.

For variables, we notationally distinguish between *instance* and *local* variables. Instance variables are always assumed to be private in Java$_{MT}$. They hold the state of an object and exist throughout the object's lifetime. Local variables are stack-allocated; they play the role of formal parameters and variables of method definitions and only exist during the execution of the method to which they belong.

The set of variables $Var = IVar \,\dot{\cup}\, TVar$ with typical element y is given as the disjoint union of the instance and the local variables. Var^t denotes the set of all variables of type t, and correspondingly for $IVar^t$ and $TVar^t$. As we assume a monomorphic type discipline, $Var^t \cap Var^{t'} = \emptyset$ for distinct types t and t'. We use $x, x', x_1, \ldots$ as typical elements from $IVar$, and $u, v, u', v_1, \ldots$ as typical elements from $TVar$.

Besides using instance and local variables, side-effect free *expressions* $e \in Exp$ are built from this, nil, and from subexpressions using the given operators. We use $Exp^t_{m,c}$ to denote the set of well-typed expressions of type t in method $m \in \mathcal{M}$ of class $c \in \mathcal{C}$, where $\mathcal{M}$ is an infinite set of method names containing main, start, and run. The expression this is used for self-reference within an object, and nil is a constant representing an empty reference. Expressions with *side-effects* $sexp \in SExp$ contain clauses for *object creation* and *method invocation*. By $SExp^t_{m,c}$ we denote the set of side-effect expressions of type t in method m of class c. The expression new^c stands for the reference to a new instance of class c. An invocation of a method with name m on object e_0 with actual parameters $e_1, \ldots, e_n$ is written as $e_0.m(e_1, \ldots, e_n)$.

Besides the mentioned simplifications on the type system, we impose for technical reasons the following restrictions: We require that method invocation and object creation statements contain only local variables, i.e., that none of the expressions $e_0, \ldots, e_n$ contains instance variables, and that formal parameters do not occur on the left-hand side of assignments; this restriction implies that during the execution of a method the values of the actual and formal parameters are not changed. Finally, the result of an object creation or method invocation statement may not be assigned to instance variables. This restriction allows for a proof system with separated verification conditions for interference freedom and cooperation. It should be clear that it is possible to transform a program to adhere to this restrictions at the expense of additional local variables and thus new interleaving points.

Statements $stm \in Stm$ are built from side-effect expressions and assignments of the form $x := e$, $u := e$, and $u := sexp$ by using standard control constructs like sequential composition, conditional statements, and iteration, to form composite statements. Especially, we will use ϵ to denote the empty statement. We refer by $Stm_{m,c}$ to the set of statements in method m of class c.

A *method* definition $modif\ m(u_1, \ldots, u_n)\{\ stm; rexp\ \} \in Meth$ consists of a method name m, a list of formal parameters $u_1, \ldots, u_n$, and a method body $body_{m,c}$ of the form $stm; rexp$. The set $Meth_c$ contains the methods of class c. To simplify the proof system we require that method bodies are terminated by a single return statement, either giving back a value using return e, or not, written as return. Additionally, methods are decorated by a modifier *modif* distinguishing between *non-synchronized* and *synchronized* methods.[3] We use $sync(c, m)$ to state that method m in class c is synchronized. In the sequel we also refer to statements in the body of a synchronized method as being synchronized. A *class* $c\{meth_1 \ldots meth_n\, meth_{\mathsf{start}}\, meth_{\mathsf{run}}\}$ is defined by its name c and its methods, whose names are assumed to be distinct. As mentioned earlier, all classes in Java$_{MT}$ are thread classes; all classes contain a start-method $meth_{\mathsf{start}}$ and a run-method $meth_{\mathsf{run}}$ without return values. A *program* $\langle class_1 \ldots class_n\, class_{\mathsf{main}} \rangle$, finally, is a collection of class definitions having different class names, where $class_{\mathsf{main}}$ is the entry point of the program execution. This class specifically contains a main-method $meth_{\mathsf{main}}$ without return value. We call its body, written as $body_{\mathsf{main}}$, the main statement of the program.

The set of *instance* variables $IVar_c$ of a class c is implicitly given by the set of all instance variables occurring in that class. Correspondingly for methods,

[3] *Java* does not have the "non-synchronized" modifier: methods are non-synchronized by default.

the set of local variables $TVar_{m,c}$ of a method m in class c is given by the set of all local variables occurring in that method.

The syntax is summarized in Table 1.

exp	$::= x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp, \ldots, exp)$	$e \in Exp$	expressions
$sexp$	$::= \mathsf{new}^c \mid exp.m(exp, \ldots, exp)$	$sexp \in SExp$	side-effect exp
stm	$::= sexp \mid x := exp \mid u := exp \mid u := sexp$		
	$\mid\ \epsilon \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm$		
	$\mid\ \mathsf{while}\ exp\ \mathsf{do}\ stm \ldots$	$stm \in Stm$	statements
$modif$	$::= \mathsf{nsync} \mid \mathsf{sync}$		modifiers
$rexp$	$::= \mathsf{return} \mid \mathsf{return}\ exp$		
$meth$	$::= modif\ m(u, \ldots, u)\{\ stm; rexp\}$	$meth \in Meth$	methods
$meth_{\mathsf{run}}$	$::= modif\ \mathsf{run}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{run}} \in Meth$	run-meth.
$meth_{\mathsf{start}}$	$::= \mathsf{nsync}\ \mathsf{start}()\{\ \mathsf{this.run}(); \mathsf{return}\ \}$	$meth_{\mathsf{start}} \in Meth$	start-meth.
$meth_{\mathsf{main}}$	$::= \mathsf{nsync}\ \mathsf{main}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{main}} \in Meth$	main-meth.
$class$	$::= c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{\mathsf{start}}\}$	$class \in Class$	class defn's
$class_{\mathsf{main}}$	$::= c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{\mathsf{start}}\ meth_{\mathsf{main}}\}$	$class_{\mathsf{main}} \in Class$	main-class
$prog$	$::= \langle class \ldots class\ class_{\mathsf{main}}\rangle$		programs

Table 1. $Java_{MT}$ abstract syntax

2.3 Semantics

In this section, we define the *operational semantics* of $Java_{MT}$, especially, the mechanisms of multithreading, dynamic object creation, method invocation, and coordination via synchronization. After introducing the semantic domains, we describe states and configurations in the following section. The operational semantics is presented in Section 2.3.2 by transitions between program configurations.

2.3.1 States and configurations To give semantics to the *expressions,* we first fix the domains Val^t of the various types t. Thus Val^{Int} and Val^{Bool} denote the set of integers and booleans, $Val^{\mathsf{list}\,t}$ are finite sequences over values from Val^t, and $Val^{t_1 \times t_2}$ stands for the product $Val^{t_1} \times Val^{t_2}$. For class names $c \in \mathcal{C}$, the set Val^c with typical elements $\alpha, \beta, \ldots$ denotes an infinite set of *object identifiers,* where the domains for different class names are assumed to be disjoint. For each class name c, $nil^c \notin Val^c$ represents the value of nil in the corresponding type. In general we will just write nil, when c is clear from the context. We define Val^c_{nil} as $Val^c \dot{\cup} \{nil^c\}$, and correspondingly for compound types. The set of all possible non-nil values $\bigcup_t Val^t$ is written as Val, and Val_{nil} denotes $\bigcup_t Val^t_{nil}$.

The configuration of a program is characterized by the configurations of all currently executing threads together with the set of existing objects and the values of their instance variables. Before formalizing the global configurations of a program, we define local states and local configurations. In the sequel we in general identify the occurrence of a statement in a program with the statement itself.

A *local state* $\tau \in \Sigma_{\mathsf{loc}}$ of a thread holds the values of its local variables and is modeled as a partial function of type $TVar \,\dot{\cup}\, \{\mathsf{this}\} \rightharpoonup Val_{nil}$. We will maintain as invariant, that the local state contains a reference to the object in which the corresponding thread is currently executing, i.e., $\mathsf{this} \in dom(\tau)$ with $\tau(\mathsf{this}) \neq nil$. For a class c and a method m of c we use the notation $\tau^{m,c}$ for local states with domain $TVar_{m,c} \,\dot{\cup}\, \{\mathsf{this}\}$ such that $\tau^{m,c}(\mathsf{this}) \in Val^c$, i.e., $\tau^{m,c}$ describes the local state of a thread executing method m of an instance of class c. We denote by τ_{init} or by $\tau_{init}^{m,c}$ local states which assign to each class-typed local variable of type c' from $dom(\tau)\backslash\{\mathsf{this}\}$ the value of $nil^{c'}$, to each boolean variable the value *false*, and to each integer variable the value 0. Pairs are initialized correspondingly; sequences are initially empty.

A *local configuration* (τ, stm) of a thread executing within an object $\tau(\mathsf{this})$ specifies, in addition to its local state, its point of execution represented by the statement stm. A *thread configuration* ξ is a stack of local configurations $(\tau_0, stm_0)(\tau_1, stm_1)\ldots(\tau_n, stm_n)$, representing the chain of method invocations of the given thread. We write $\xi \circ (\tau, stm)$ for pushing a new local configuration onto the top of the stack.

The state of an object is characterized by its *instance state* $\sigma_{inst} \in \Sigma_{inst}$ of type $IVar \rightharpoonup Val_{nil}$ which assigns values to its instance variables. For a class c we write σ_{inst}^c to denote instance states assigning values to the instance variables of class c, i.e., σ_{inst}^c is of type $IVar_c \rightarrow Val_{nil}$. The initial instance state σ_{inst}^{init} or $\sigma_{inst}^{c,init}$ assigns to each of its instance variables of type c' the value $nil^{c'}$, to each of its boolean instance variable the value *false*, and to each integer variable the value 0. Pairs are initialized correspondingly; sequences are initially empty. A *global state* $\sigma \in \Sigma$ stores for each currently *existing* object its instance state and is modeled as a partial function of type $(\bigcup_{c\in\mathcal{C}} Val^c) \rightharpoonup \Sigma_{inst}$. The set of existing objects of type c in a state σ is given by $dom^c(\sigma)$, and $dom_{nil}^c(\sigma)$ is defined by $dom^c(\sigma) \cup \{nil^c\}$. For the built-in types Int and Bool we define dom^t and dom_{nil}^t, independently of σ, as the set of pre-existing values Val^{Int} and Val^{Bool}, respectively. For compound types, dom^t and dom_{nil}^t are defined correspondingly. We refer to the set $\bigcup_t dom^t$ by $dom(\sigma)$; $dom_{nil}(\sigma)$ denotes $\bigcup_t dom_{nil}^t$. The instance state of an object $\alpha \in dom(\sigma)$ is given by $\sigma(\alpha)$. We call an object $\alpha \in dom(\sigma)$ existing in σ, and we throughout require that, given a global state, no instance variable in any of the existing objects refers to a non-existing object, i.e., $\sigma(\alpha)(x) \in dom_{nil}(\sigma)$ for all $\alpha \in dom^c(\sigma)$. This will be an invariant of the operational semantics of the next section.

A *global configuration* $\langle T, \sigma\rangle$ consists of a set T of thread configurations of the currently executing threads, together with a global state σ describing the currently existing objects. Analogously to the restriction on global states, we require that local configurations (τ, stm) in $\langle T, \sigma\rangle$ do not refer to non-existing object identities, i.e., $\tau(u) \in dom_{nil}(\sigma)$ for all variables u from the domain of τ, and again this will be an invariant of the operational semantics. In the following we write $(\tau, stm) \in T$ if there exists a local configuration (τ, stm) within one of the execution stacks of T.

Expressions $e \in Exp^t_{m,c}$ are evaluated with respect to an *instance local* state $(\sigma^c_{inst}, \tau^{m,c}) \in \Sigma_{inst} \times \Sigma_{\mathsf{loc}}$, where, as mentioned, the local state defines the object $\tau^{m,c}(\mathsf{this})$ in which the thread is currently executing and the values of the current local variables of the thread, and σ^c_{inst} defines the values of the instance variables of $\tau^{m,c}(\mathsf{this})$. This means, the semantic function $[\![\_]\!]_{\mathcal{E}} : (\Sigma_{inst} \times \Sigma_{\mathsf{loc}}) \rightarrow (Exp \rightharpoonup Val_{nil})$ shown in Table 2 evaluates in the context of an instance local state (σ_{inst}, τ) expressions containing only variables from $dom(\sigma_{inst}) \cup dom(\tau)$: Instance variables x and local variables u are evaluated to $\sigma_{inst}(x)$ and $\tau(u)$, respectively. The value of this refers to the object in which the expression is evaluated, nil has the undefined value *nil*. Finally, the evaluation of compound expressions are defined by homomorphic lifting.

$$
\begin{aligned}
[\![x]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= \sigma_{inst}(x) \\
[\![u]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= \tau(u) \\
[\![\mathsf{this}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= \tau(\mathsf{this}) \\
[\![\mathsf{nil}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= nil \\
[\![\mathsf{f}(e_1, \ldots, e_n)]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} &= f([\![e_1]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}, \ldots, [\![e_n]\!]_{\mathcal{E}}^{\sigma_{inst},\tau})
\end{aligned}
$$

Table 2. Expression evaluation

For a local state τ, a local variable u of type t, and $val \in Val^t_{nil}$, we denote by $\tau[u \mapsto val]$ the local state which assigns val to u and agrees with τ on the values of all other variables. The semantic update $\sigma_{inst}[x \mapsto val]$ of instance states is defined analogously. Correspondingly for global states, $\sigma[\alpha.x \mapsto val]$ denotes the global state resulting from σ by assigning val to the instance variable x of object α. We use these operators analogously for simultaneously setting the values of a vector of variables. We use $\tau[\vec{y} \mapsto \vec{v}]$ also for arbitrary variable sequences, where instance variables are untouched, i.e., $\tau[\vec{y} \mapsto \vec{v}]$ is defined by $\tau[\vec{u} \mapsto \vec{v}_u]$, where $\vec{u}$ is the sequence of the local variables in $\vec{y}$ and $\vec{v}_u$ the corresponding value sequence. Similarly, for instance states, $\sigma_{inst}[\vec{y} \mapsto \vec{v}]$ is defined by $\sigma_{inst}[\vec{x} \mapsto \vec{v}_x]$ where $\vec{x}$ is the sequence of the instance variables in $\vec{y}$ and $\vec{v}_x$ the corresponding value sequence. The semantics of $\sigma[\alpha.\vec{y} \mapsto \vec{v}]$ is analogous. Finally for global states, $\sigma[\alpha \mapsto \sigma^c_{inst}]$ equals σ except on α; note that in case $\alpha \notin dom^c(\sigma)$, the operation extends the set of existing objects by α, that has its instance state initialized to σ^c_{inst}.

2.3.2 Operational semantics Computation steps of a program are represented by transitions between global configurations. The operational semantics of *Java*$_{MT}$ is given inductively by the rules of Table 3.

Rule $\textsc{Ass}_{inst}$ states that assigning e to the instance variable x executed in an object α updates the instance state of the respective object with a new value of x as given by evaluation of the expression in the respective instance local state. Assignments to local variables are handled correspondingly by rule $\textsc{Ass}_{loc}$, where the local state is updated. Executing $u := \mathsf{new}^c$, as shown in rule NEW, creates a new object of type c, initializes its instance variables, but does not yet add a new execution stack to the global configuration.[4] This is done by the first invocation of the start-method (cf. rule START), thereby initializing the first activation record of the new stack. Only the first invocation of the start-method has this effect. This is captured by the predicate *started* which holds for a global configuration T and an instance α iff there exists a stack $(\tau_0, stm_0) \ldots (\tau_n, stm_n) \in T$ such that $\tau_0(\mathsf{this}) = \alpha$. Further invocations of the start-method are without effect (cf. rule $\textsc{Start}_{skip}$).[5]

Invoking a method extends the call chain by a new local configuration (cf. rule CALL for methods with return value). After initializing the local state, the values of the actual parameters are assigned to the formal parameters and the thread begins to execute the method body. We introduce the statements receive and receive u to denote that the execution of a configuration is suspended until the invoked method terminates, where the return value, if any, will be stored in the variable u. Note that these statements are not part of the syntax of *Java*$_{MT}$. Statements *stm* in the operational semantics are assumed to be program statements possibly containing receive statements. An analogous rule not shown here takes care of method invocation without return value.

Different threads execute synchronized methods mutually exclusive on a given object. This is expressed by the condition $sync(c, m) \rightarrow isfree(T, \beta)$, where *isfree* is a predicate over a set of stacks and an object such that $isfree(T, \beta)$ is true iff no stack in T contains any local configuration (τ, stm) with $\tau(\mathsf{this}) = \beta$ and *stm* synchronized. This means, a synchronized method of an object can be invoked if and only if currently no other thread executes any synchronized methods of this object.

When returning from a method call (cf. rule RETURN) the callee evaluates its return expression and passes it to the caller which subsequently updates its local state. The method body terminates its execution and the caller can continue. An analogous rule, not shown in the table, deals with returning from a method without return value. Returning from the initial invocation of the main-method or from a start-method is specific in that there is no caller configuration in the stack (cf. rule TERMINATE). The worked-off local configuration (τ, ϵ) is kept in the global configuration to ensure that the thread of $\tau(\mathsf{this})$ cannot be started twice.

---

[4] The statement new^c is handled similarly but without changing the local state.

[5] In *Java* an exception is thrown if the thread is already terminated.

We elide the rules for the remaining sequential constructs —sequential composition, conditional statement, and iteration— since they are standard.

We conclude the section with the definition of *initial* and *reachable* configurations. The initial configuration $\langle T_0, \sigma_0 \rangle$ of a program satisfies the following: $T_0 = \{(\tau_{init}^{\mathsf{main},c}[\mathsf{this} \mapsto \alpha], body_{\mathsf{main}})\}$, where c is the main class, and $\alpha \in Val^c$. Moreover, $dom(\sigma_0) = \{\alpha\}$ and $\sigma_0(\alpha) = \sigma_{inst}^{c,init}$. We call a configuration $\langle T, \sigma \rangle$ of a program *reachable* iff there exists a computation $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle T, \sigma \rangle$ such that $\langle T_0, \sigma_0 \rangle$ is the initial configuration of the program and $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$.

In *Java*, the main method of a program is static. Since Java_{MT} does not have static methods and variables, we define the initial configuration as having a single initial object in which an initial thread starts to execute the main-method. Note that according to the definition of the *started* predicate, the start-method of the initial object cannot be invoked.

3 The assertion language

In this section we introduce *assertions* to specify properties of Java_{MT} programs. The assertion logic consists of a *local* and a *global* sublanguage. The *local* assertion language is used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong. The *global* assertion language describes a whole system of objects and their communication structure and will be used in the cooperation test.

To be able to argue about communication histories, represented as lists of objects, we add the type Object as the supertype of all classes into the assertion language. Note that we allow this type solely in the assertion language, but not in the programming language, thus preserving the assumption of monomorphism.

After fixing the syntax of the assertions in the next section, we define its semantics and provide basic substitution properties.

3.1 Syntax

In the language of assertions, we introduce a countably infinite set $LVar$ of well-typed *logical variables* with typical element z, where we assume that instance variables, local variables, and this are not in $LVar$. Logical variables are used for quantification in both the local and the global language. Besides that, they are used as free variables to represent local variables in the global assertion language: To express a local property on the global level, each local variable in a given local assertion will be replaced by a fresh logical variable.

Table 4 defines the syntax of the assertion language. *Local expressions* $exp_l \in LExp$ are expressions of the programming language possibly containing logical variables. The set $LExp_{m,c}^t$ consists of all local expressions of type t in method m of class c, where $LExp^t$ is defined by $\bigcup_{m,c} LExp_{m,c}^t$. In abuse of notation, we use e, $e' \ldots$ not only for program expressions of Table 1, but also for typical elements of

$$\frac{\tau(\mathsf{this}) = \alpha}{\langle T \dot\cup \{\xi \circ (\tau, x{:=}e; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau, stm)\}, \sigma[\alpha.x \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]\rangle} \ \textsc{Ass}_{inst}$$

$$\frac{\tau(\mathsf{this}) = \alpha}{\langle T \dot\cup \{\xi \circ (\tau, u{:=}e; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau[u \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}], stm)\}, \sigma\rangle} \ \textsc{Ass}_{loc}$$

$$\frac{\beta \in Val^c \backslash dom^c(\sigma) \qquad \sigma' = \sigma[\beta \mapsto \sigma_{inst}^{c,init}]}{\langle T \dot\cup \{\xi \circ (\tau, u{:=}\mathsf{new}^c; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau[u \mapsto \beta], stm)\}, \sigma'\rangle} \ \textsc{New}$$

$$\frac{\begin{array}{c}\tau(\mathsf{this}) = \alpha \qquad \beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in dom^c(\sigma) \qquad \tau' = \tau_{init}^{\mathsf{start},c}[\mathsf{this} \mapsto \beta] \\ \neg started(T \cup \{\xi \circ (\tau, e.\mathsf{start}(); stm)\}, \beta)\end{array}}{\langle T \dot\cup \{\xi \circ (\tau, e.\mathsf{start}(); stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau, stm), (\tau', body_{\mathsf{start},c})\}, \sigma\rangle} \ \textsc{Start}$$

$$\frac{\begin{array}{c}\tau(\mathsf{this}) = \alpha \qquad \beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in dom^c(\sigma) \\ started(T \cup \{\xi \circ (\tau, e.\mathsf{start}(); stm)\}, \beta)\end{array}}{\langle T \dot\cup \{\xi \circ (\tau, e.\mathsf{start}(); stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau, stm)\}, \sigma\rangle} \ \textsc{Start}_{skip}$$

$$\frac{\begin{array}{c}\tau(\mathsf{this}) = \alpha \qquad \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in dom^c(\sigma) \qquad modif\ m(\vec{u})\{\ body\ \} \in Meth_c \\ m \neq \mathsf{start} \qquad \tau' = \tau_{init}^{m,c}[\mathsf{this} \mapsto \beta][\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \qquad sync(c, m) \rightarrow isfree(T, \beta)\end{array}}{\langle T \dot\cup \{\xi \circ (\tau, u{:=}e_0.m(\vec{e}); stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau, \mathsf{receive}\ u; stm) \circ (\tau', body)\}, \sigma\rangle} \ \textsc{Call}$$

$$\frac{\tau'(\mathsf{this}) = \beta \qquad \tau'' = \tau[u \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\beta),\tau'}]}{\langle T \dot\cup \{\xi \circ (\tau, \mathsf{receive}\ u; stm) \circ (\tau', \mathsf{return}\ e)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau'', stm)\}, \sigma\rangle} \ \textsc{Return}$$

$$\frac{}{\langle T \dot\cup \{(\tau, \mathsf{return})\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{(\tau, \epsilon)\}, \sigma\rangle} \ \textsc{Terminate}$$

Table 3. Operational semantics

local expressions. *Local assertions* $ass_l \in LAss$, with typical elements $p, p', q, \ldots$, are standard logical formulas over boolean local expressions; local assertions in method m of class c form the set $LAss_{m,c}$. We allow three forms of quantification over the logical variables: Unrestricted quantification $\exists z(p)$ is solely allowed for integer and boolean domains, i.e., z is required to be of type Int or Bool. For reference types c, this form of quantification is not allowed, as for those types, the existence of a value dynamically depends on the *global* state, something one cannot speak about on the local level, or more formally: Disallowing unrestricted quantification for object types ensures that the value of a local assertion indeed only depends on the values of the instance and local variables, but not on the global state. Nevertheless, one can assert the existence of objects on the local level satisfying a predicate, provided one is explicit about the set of objects to range over. Thus, the restricted quantifications $\exists z \in e(p)$ or $\exists z \sqsubseteq e(p)$ assert the existence of an element, respectively, the existence of a subsequence of a given sequence e, for which a property p holds.

Global expressions $exp_g \in GExp$, with typical elements $E, E', \ldots$, are constructed from logical variables, nil, operator expressions, and qualified references $E.x$ to instance variables x of objects E. We write $GExp^t$ for the set of global expressions of type t. *Global assertions* $ass_g \in GAss$, with typical elements $P, Q \ldots$, are logical formulas over boolean global expressions. Unlike the local language, the meaning of the global one is defined in the context of a global state. Thus unrestricted quantification is allowed for all types and is interpreted to range over the set of *existing* values, i.e., the set of values $dom_{nil}(\sigma)$ in a global configuration $\langle T, \sigma \rangle$.

$exp_l ::= z \mid x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp_l, \ldots, exp_l)$	$e \in LExp$	local expressions
$ass_l ::= exp_l \mid \neg ass_l \mid ass_l \wedge ass_l$		
$\mid\ \exists z(ass_l) \mid \exists z \in exp_l(ass_l) \mid \exists z \sqsubseteq exp_l(ass_l)$	$p \in LAss$	local assertions
$exp_g ::= z \mid \mathsf{nil} \mid \mathsf{f}(exp_g, \ldots, exp_g) \mid exp_g.x$	$E \in GExp$	global expressions
$ass_g ::= exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists z(ass_g)$	$P \in GAss$	global assertions

Table 4. Syntax of assertions

3.2 Semantics

Next, we define the interpretation of the assertion language. The semantics is fairly standard, except that we have to cater for dynamic object creation when interpreting quantification.

Expressions and assertions are interpreted relative to a logical environment $\omega \in \Omega$, a partial function of type $LVar \rightharpoonup Val_{nil}$, assigning values to logical variables. We denote by $\omega[z \mapsto val]$ the logical environment that assigns $val \in$

Val_{nil} to z, and agrees with ω on all other variables. For a logical environment ω and a global state σ we say that ω refers only to values existing in σ, if $\omega(z) \in dom_{nil}(\sigma)$ for all $z \in dom(\omega)$. This property matches with the definition of quantification which ranges only over existing values and *nil*, and with the fact that in reachable configurations local variables may refer only to existing values or to *nil*. Correspondingly for local states, we say that a local state τ refers only to values existing in σ, if $\tau(u) \in dom_{nil}(\sigma)$ for all $u \in dom(\tau)$.

The semantic function $[\![\_]\!]_{\mathcal{L}}$ of type $(\Omega \times \Sigma_{inst} \times \Sigma_{\mathsf{loc}}) \rightarrow (LExp \cup LAss \rightharpoonup Val_{nil})$ evaluates local expressions and assertions in the context of a logical environment ω and an instance local state (σ_{inst}, τ) (cf. Table 5). The evaluation function is defined for expressions and assertions that contain only variables from $dom(\omega) \cup dom(\sigma_{inst}) \cup dom(\tau)$. The instance local state provides the context for giving meaning to programming language expressions as defined by the semantic function $[\![\_]\!]_{\mathcal{E}}$; the logical environment evaluates logical variables. An unrestricted quantification $\exists z(p)$ is evaluated to true in the logical environment ω and instance local state (σ_{inst}, τ) if and only if there exists a value $val \in Val^t$ such that p holds in the logical environment $\omega[z \mapsto val]$ and instance local state (σ_{inst}, τ), where for the type t of z only Int or Bool is allowed. The evaluation of a restricted quantification $\exists z \in e(p)$ with $z \in LVar^t$ and $e \in LExp^{\mathsf{list}\,t}$ is defined analogously, where the existence of an element in the sequence is required. An assertion $\exists z \sqsubseteq e(p)$ with $z \in LVar^{\mathsf{list}\,t}$ and $e \in LExp^{\mathsf{list}\,t}$ states the existence of a subsequence of e for which p holds. In the following we also write $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $[\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true$. By $\models_{\mathcal{L}} p$, we express that $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ holds for arbitrary logical environments, instance states, and local states.

$$
\begin{array}{rcl}
[\![z]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \omega(z) \\
[\![x]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \sigma_{inst}(x) \\
[\![u]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \tau(u) \\
[\![\mathsf{this}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \tau(\mathsf{this}) \\
[\![\mathsf{nil}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & nil \\
[\![\mathsf{f}(e_1,\ldots,e_n)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & f([\![e_1]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau},\ldots,[\![e_n]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}) \\
([\![\neg p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = false) \\
([\![p_1 \wedge p_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![p_1]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true \text{ and } [\![p_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) \\
([\![\exists z(p)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega[z\mapsto val],\sigma_{inst},\tau} = true \text{ for some } val \in Val) \\
([\![\exists z{\in}e(p)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![z{\in}e \wedge p]\!]_{\mathcal{L}}^{\omega[z\mapsto val],\sigma_{inst},\tau}{=}true \text{ for some } val{\in}Val_{nil}) \\
([\![\exists z{\sqsubseteq}e(p)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![z{\sqsubseteq}e \wedge p]\!]_{\mathcal{L}}^{\omega[z\mapsto val],\sigma_{inst},\tau}{=}true \text{ for some } val{\in}Val_{nil})
\end{array}
$$

Table 5. Local evaluation

Since *global* assertions do not contain local variables and non-qualified references to instance variables, the global assertional semantics does not refer to instance local states but to global states. The semantic function $[\![\_]\!]_{\mathcal{G}}$ of type $(\Omega \times \Sigma) \rightharpoonup (GExp \cup GAss \rightharpoonup Val_{nil})$, shown in Table 6, gives meaning to global expressions and assertions in the context of a global state σ and a logical environment ω. To be well-defined, ω is required to refer only to values existing in σ, and the expression respectively assertion may only contain free variables from $dom(\omega) \cup dom(\sigma)$. Logical variables, nil, and operator expressions are evaluated analogously to local assertions. The value of a global expression $E.x$ is given by the value of the instance variable x of the object referred to by the expression E. The evaluation of an expression $E.x$ is defined only if E refers to an object existing in σ. Note that when E and E' refer to the same object, that is, E and E' are *aliases*, then $E.x$ and $E'.x$ denote the same variable. The semantics of negation and conjunction is standard. A quantification $\exists z(P)$ evaluates to true in a logical environment ω and global state σ if and only if P evaluates to true in the logical environment $\omega[z \mapsto val]$ and global state σ, for some value $val \in dom_{nil}(\sigma)$. Note that quantification over objects ranges over the set of *existing* objects and *nil*, only.

$$
\begin{array}{rcl}
[\![z]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \omega(z) \\
[\![\mathsf{nil}]\!]_{\mathcal{G}}^{\omega,\sigma} & = & nil \\
[\![\mathsf{f}(E_1,\ldots,E_n)]\!]_{\mathcal{G}}^{\omega,\sigma} & = & f([\![E_1]\!]_{\mathcal{G}}^{\omega,\sigma},\ldots,[\![E_n]\!]_{\mathcal{G}}^{\omega,\sigma}) \\
[\![E.x]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \\
([\![\neg P]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega,\sigma} = false) \\
([\![P_1 \wedge P_2]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P_1]\!]_{\mathcal{G}}^{\omega,\sigma} = true \text{ and } [\![P_2]\!]_{\mathcal{G}}^{\omega,\sigma} = true) \\
([\![\exists z(P)]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega[z \mapsto val],\sigma} = true \text{ for some } val \in dom_{nil}(\sigma))
\end{array}
$$

Table 6. Global evaluation

For a global state σ and a logical environment ω referring only to values existing in σ we write $\omega, \sigma \models_{\mathcal{G}} P$ when P is true in the context of ω and σ. We write $\models_{\mathcal{G}} P$ if P holds for arbitrary global states σ and logical environments ω that refers only to values existing in σ.

The verification conditions defined in the next section involve the following substitution operations: The standard capture-avoiding substitution $p[\vec{e}/\vec{y}]$ replaces in the local assertion p all occurrences of the given distinct variables $\vec{y}$ by the local expressions $\vec{e}$. We apply the substitution also to local expressions. The following lemma expresses the standard property of the above substitution, relating it to state-update. The relation between substitution and update formulated in the lemma asserts that $p[\vec{e}/\vec{y}]$ is the *weakest precondition* of p wrt. to the assignment. The lemma will be used for proving invariance of local assertions under assignments.

Lemma 1 (Local substitution). *For arbitrary logical environments ω, instance local states (σ_{inst}, τ), local expressions e', and local assertions p, we have*

$$\begin{array}{rcl} [\![e'[\vec{e}/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & [\![e']\!]_{\mathcal{L}}^{\omega,\sigma_{inst}[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}],\tau[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}]}, \textit{ and} \\ \omega,\sigma_{inst},\tau \models_{\mathcal{L}} p[\vec{e}/\vec{y}] & \textit{iff} & \omega,\sigma_{inst}[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}],\tau[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}] \models_{\mathcal{L}} p\,. \end{array}$$

The effect of assignments to instance variables is expressed on the *global* level by the substitution $P[\vec{E}/z.\vec{x}]$, which replaces in the global assertion P the instance variables $\vec{x}$ of the object referred to by z by the global expressions $\vec{E}$. To accommodate properly for the effect of assignments, though, we must not only syntactically replace the occurrences $z.x_i$ of the instance variables, but also all their *aliases* $E'.x_i$, when z and the result of the substitution applied to E' refer to the same object. As the aliasing condition cannot be checked syntactically, we define the main case of the substitution by a conditional expression [6]:

$$(E'.x_i)[\vec{E}/z.\vec{x}] = (\mathsf{if}\ \ E'[\vec{E}/z.\vec{x}] = z\ \ \mathsf{then}\ \ E_i\ \ \mathsf{else}\ (E'[\vec{E}/z.\vec{x}]).x_i\ \ \mathsf{fi})\ .$$

The substitution is extended to global assertions homomorphically. We use this substitution to express that a property defined in the global assertion language is invariant under assignments. For the sake of convenience, we also use the substitution $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences $\vec{y}$ possibly containing local variables, whose semantics is defined by $P[\vec{E}_x/z.\vec{x}]$, where $\vec{x}$ is the sequence of the instance variables of $\vec{y}$ and $\vec{E}_x$ is the corresponding subsequence of $\vec{E}$. That the substitution accurately catches the semantical update, and thus represents the weakest precondition relation, is expressed by the following lemma:

Lemma 2 (Global substitution). *For arbitrary global states σ, logical environments ω referring only to values existing in σ, global expressions E', global assertions P, and class-typed logical variables z:*

$$\begin{array}{rcl} [\![E'[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = & [\![E']\!]_{\mathcal{G}}^{\omega,\sigma[[\![z]\!]_{\mathcal{G}}^{\omega,\sigma}.\vec{x}\mapsto[\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]}, \textit{and} \\ \omega,\sigma \models_{\mathcal{G}} P[\vec{E}/z.\vec{x}] & \textit{iff} & \omega,\sigma[[\![z]\!]_{\mathcal{G}}^{\omega,\sigma}.\vec{x}\mapsto[\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}] \models_{\mathcal{G}} P\,. \end{array}$$

To express a local property p in the global assertion language, we define the substitution $p[z,\vec{E}/\mathsf{this},\vec{u}]$ by simultaneously replacing in p all occurrences of the self-reference this by the logical variable z, which is assumed to occur neither in p nor in $\vec{E}$, and all occurrences of the local variables $\vec{u}$ by the global expressions $\vec{E}$. The main cases of the substitution are defined as follows:

$$\begin{array}{rcl} \mathsf{this}[z,\vec{E}/\mathsf{this},\vec{u}] & = & z \\ x[z,\vec{E}/\mathsf{this},\vec{u}] & = & z.x \\ u_i[z,\vec{E}/\mathsf{this},\vec{u}] & = & E_i \\ (\exists z'(p))[z,\vec{E}/\mathsf{this},\vec{u}] & = & \exists z'(p[z,\vec{E}/\mathsf{this},\vec{u}]) \\ (\exists z' \in e(p))[z,\vec{E}/\mathsf{this},\vec{u}] & = & \exists z'((z' \in e[z,\vec{E}/\mathsf{this},\vec{u}]) \wedge p[z,\vec{E}/\mathsf{this},\vec{u}]) \\ (\exists z' \sqsubseteq e(p))[z,\vec{E}/\mathsf{this},\vec{u}] & = & \exists z'((z' \sqsubseteq e[z,\vec{E}/\mathsf{this},\vec{u}]) \wedge p[z,\vec{E}/\mathsf{this},\vec{u}])\ , \end{array}$$

where $z \neq z'$ in the cases for existential quantification. The substitution replaces all occurrences of the self-reference this by z, transforms all occurrences of instance variables x into qualified references $z.x$, and substitutes all local variables u_i by the given global expressions E_i. For unrestricted quantifications $(\exists z'(p))[z, \vec{E}/\mathsf{this}, \vec{u}]$ the substitution applies to the assertion p. Local restricted quantifications are transformed into global unrestricted ones where the relations $\in$ and $\sqsubseteq$ are expressed at the global level as operators.

For notational convenience we sometimes view the local variables occurring in the global assertion $p[z/\mathsf{this}]$ as logical variables. Formally, these local variables are replaced by fresh logical variables.

This substitution will be used to combine properties of instance local states on the global level. The substitution $[z, \vec{E}/\mathsf{this}, \vec{u}]$ preserves the meaning of local assertions, provided the meaning of the local variables $\vec{u}$ and this is matchingly represented by the global expressions $\vec{E}$ and z:

Lemma 3 (Lifting substitution). *Let σ be a global state, ω and τ a logical environment and local state, both referring only to values existing in σ. Let furthermore e and p be a local expression and a local assertion containing local variables $\vec{u}$. If $\tau(\vec{u}) = [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}$ and z a fresh logical variable with $\omega(z) = \tau(\mathsf{this})$, then*

$$\begin{array}{rcl} [\![e[z, \vec{E}/\mathsf{this}, \vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = & [\![e]\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau}\ , \ \textit{and} \\ \omega, \sigma \models_{\mathcal{G}} p[z, \vec{E}/\mathsf{this}, \vec{u}] & \textit{iff} & \omega, \sigma(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} p\ . \end{array}$$

4 The proof system

This section presents the assertional proof system for reasoning about $Java_{MT}$ programs, formulated in terms of *proof outlines* [26, 15], i.e., where Hoare-style pre- and postconditions [16, 20] are associated with each program statement. The proof system has to accommodate for dynamic object creation, shared-variable concurrency, aliasing, method invocation, and synchronization.

The following section defines how to augment and annotate programs into proof outlines, before Section 4.2 describes the proof method.

4.1 Proof outlines

To reason about multithreading and communication, we first define a program transformation, introducing new communication statements that model explicitly the communication mechanism of method invocations, then augment the program by auxiliary variables, and, finally, introduce bracketed sections.

To be able to reason about the communication mechanism of method invocations from the view of the caller, who sends the actual parameter values and receives the return value, we split each method invocation $u := e_0.m(\vec{e})$ different from the invocation of the start-method of an object into the sequential composition of the statements $e_0.m(\vec{e})$ and $\mathsf{receive}\, u$. Execution of a *method call* $e_0.m(\vec{e})$ sends the actual parameter values, whereas the corresponding *receive* statement $\mathsf{receive}\, u$ models the reception of the result value. Correspondingly, for methods without return value, the pure receive statement receive is used, instead.

To express properties of the multithreaded flow of control we need to augment the program by fresh *auxiliary* variables, disjoint from the program variables, both as local and as instance variables. They are added only for the sake of verification and do not influence the control flow. These additional variables represent information about the global configuration within local and instance states.

Formally, assignments $y := e$ of expressions without side-effects to instance or local variables can be extended to multiple assignments $y, \vec{y} := e, \vec{e}$ by inserting additional assignments to auxiliary variables, where $(y, \vec{y})$ denotes a vector of distinct variables and $(e, \vec{e})$ a corresponding sequence of side-effect-free expressions. Besides the above extension of already occurring assignments, additional multiple assignments to auxiliary variables can be inserted at any point of the program.

Finally, we extend programs by *bracketed sections,* a conceptual notion, which is introduced for the purpose of proof and does not influence the control flow. Semantically, a bracketed section $\langle stm \rangle$ expresses that the statements inside are executed without interleaving with other threads. To make object creation and communication observable, we attach auxiliary assignments to the corresponding statements; to do the observation immediately after these statements, we enclose the statement and the assignment in bracketed sections. The replacement of communication and object creation statements, and method bodies is defined in Table 7, where $\vec{y}, \vec{y}_1, \ldots, \vec{y}_4$ are arbitrary auxiliary variable sequences.

Replace		by
call	$e_0.m(\vec{e})$	$\langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle$
method body	$stm; rexp$	$\langle \vec{y}_2 := \vec{e}_2 \rangle; stm; \langle rexp; \vec{y}_3 := \vec{e}_3 \rangle$
receive	$\mathsf{receive}\, u$	$\langle \mathsf{receive}\, u; \vec{y}_4 := \vec{e}_4 \rangle$
receive	$\mathsf{receive}$	$\langle \mathsf{receive}; \vec{y}_4 := \vec{e}_4 \rangle$
object creation	$u := \mathsf{new}$	$\langle u := \mathsf{new}; \vec{y} := \vec{e} \rangle$
object creation	new	$\langle \mathsf{new}; \vec{y} := \vec{e} \rangle$

Table 7. Bracketed sections

As auxiliary variables do not change the control flow of the original program, we can schedule the execution order of the augmented program as follows: For method call statements, after communication of the parameters, first the auxiliary assignment of the caller and then that of the callee is executed. Conversely for return, where the communication of the return value is followed by the execution of the assignment of the callee and then that of the caller, in this order. Note that these three steps for method invocation and return may not be interleaved by other threads.

Control points within a bracketed section and at the beginning of a method body we call *non-interleaving* points. All other control points are called *interleaving* points. A global configuration $\langle T, \sigma \rangle$ is *stable*, if for all local configurations (τ, stm) in T, *stm* represents an interleaving point. Restricted to an object, $\langle T, \sigma \rangle$ is *stable in* α, if for all local configurations (τ, stm) in T with $\tau(\mathsf{this}) = \alpha$, *stm* represents an interleaving point. A local configuration $(\tau, stm) \in T$ is *enabled* in $\langle T, \sigma \rangle$, if the statement *stm* can be executed at the current point, i.e., if there is a computation step $\langle T, \sigma \rangle \to \langle T', \sigma' \rangle$ executing *stm* in the local state τ.

A *transformation* of a program is given by, first, introducing communication statements, then adding assignments to auxiliary variables, and, finally, extending the program by bracketed sections. The operational semantics of transformed programs is given in Appendix A. A transformation does not change the original behavior of a program (cf. Lemma 9), except that it introduces additional non-interleaving points.

The definition of a complete proof system requires that we can formulate the transition semantics of $Java_{MT}$ in the assertion language. As the assertion language can reason about the local and global states, only, we have to augment the program with auxiliary variables to represent information about the control points and stack structures within the local and global states. We introduce the specific auxiliary variables

$$\mathsf{callerobj}, \mathsf{id}, \mathsf{lock}, \mathsf{started}, \text{and } \mathsf{stable}\ ,$$

described in the following.

An important point of the proof system is the identification of the communicating objects and threads. Roughly speaking, the local state of the execution of a method must represent information about the caller object to distinguish self-calls from others. Additionally, information about its thread membership and its position within the call stack is needed to detect local configurations in caller-callee relationship and reentrant calls. As these distinctions determine whether and how the auxiliary assignments accompanying the communication statements affect the *instance* states of objects, they will be crucial in the formulation of the interference freedom test.

We identify a thread by the object in which it has begun its execution, i.e., by the self-reference of the deepest local configuration in the thread's stack. This identification is unique since the start-method of an object can be invoked only once, i.e., at most one thread can begin its execution in a single object. A local configuration is identified by the stack it appears in together with its position in the stack, i.e., the stack depth at which it occurs.

Formally, each method definition is extended by the auxiliary formal parameters $\mathsf{callerobj}$ and id. The variable $\mathsf{callerobj}$ of type Object stores the identity of the caller object. The variable id of type $\mathsf{Object} \times \mathsf{Int}$ is used to identify the executing thread via the object in which it has begun its execution, and the position of the corresponding local configuration in the stack of the thread. Each formal parameter list $\vec{u}$ is extended to $(\mathsf{callerobj}, \mathsf{id}, \vec{u})$. When executing the main-method in the initial configuration, $\mathsf{callerobj}$ is initialized to nil, and id gets the initial value $(\alpha, 0)$, where α is the initial object. Correspondingly for each method invocation, $e_0.m(\vec{e})$ is extended to $e_0.m(\mathsf{this}, \mathsf{callee}(\mathsf{id}), \vec{e})$, where $callee(\alpha, n) = (\alpha, n+1)$ for all $n \geq 0$. If m is the start-method, the method call statement is replaced by $e_0.\mathsf{start}(nil, (e_0, 0))$, instead.

To express if two local configurations appear in the same stack let the function $samethread : (\mathsf{Object} \times \mathsf{Int})^2 \to \mathsf{Bool}$ be defined by $samethread((\alpha, n), (\beta, m))$ iff $\alpha = \beta$. Similarly, the relation $<$ of the same type is given by $(\alpha_1, n_1) < (\alpha_2, n_2)$ iff $\alpha_1 = \alpha_2$ and $n_1 < n_2$. The following lemma formalizes some basic invariant properties of the auxiliary variables $\mathsf{callerobj}$ and id.

Lemma 4 (Identification). *Let $\langle T, \sigma \rangle$ be a reachable configuration of a transformed program. Then*

1. *for all stacks $\xi, \xi' \in T$ and for all local configurations $(\tau, stm) \in \xi$ and $(\tau', stm') \in \xi'$ we have $samethread(\tau(\mathsf{id}), \tau'(\mathsf{id})) = true$ iff $\xi = \xi'$, and*
2. *for each stack $(\tau_0, stm_0) \dots (\tau_n, stm_n)$ in T and each index $i \in \{0, \dots, n\}$ $\tau_i(\mathsf{id}) = (\tau_0(\mathsf{this}), i)$; furthermore, $\tau_0(\mathsf{callerobj}) = nil$ and $\tau_j(\mathsf{callerobj}) = \tau_{j-1}(\mathsf{this})$ for all $j \in \{1, \dots, n\}$.*

To be able to reason about the synchronization mechanism of $Java_{MT}$, we extend each class definition by the auxiliary instance variable lock of type $\mathsf{Object} \times \mathsf{Int}$. Its initial value $(nil, 0)$ states that no thread is currently executing any synchronized method of the given object; otherwise, the value (α, n) identifies the thread which acquired the lock by invoking a synchronized method of the given object. Besides the identity α of the lock-holder, lock remembers the stack depth n, at which the thread has gotten the lock. I.e., if a thread is currently executing some synchronized methods in an object α, then the variable lock of α stores the identity of the deepest local configuration in the thread's stack which represents the execution of a synchronized method of α.

Formally, lock reservation for a synchronized method with body $\langle \vec{y}_2 := \vec{e}_2 \rangle; stm; \langle rexp; \vec{y}_3 := \vec{e}_3 \rangle$, is represented by including the assignment $\mathsf{lock} := \mathsf{getlock}(\mathsf{lock}, \mathsf{id})$ in $\vec{y}_2 := \vec{e}_2$, and $\mathsf{lock} := \mathsf{release}(\mathsf{lock}, \mathsf{id})$ into $\vec{y}_3 := \vec{e}_3$ for lock release. The interpretation of the operators $\mathsf{getlock}$ and $\mathsf{release}$ is defined by

$$getlock(lock, id) = \begin{cases} lock \text{ if } lock \neq (nil, 0) \\ id \quad \text{otherwise} \end{cases}$$
$$release(lock, id) = \begin{cases} lock \quad\ \text{ if } lock \neq id \\ (nil, 0) \text{ otherwise}. \end{cases}$$

The following lemma shows how to express enabledness of the invocation of synchronized methods using the auxiliary variable lock of the callee object:

Lemma 5 (Lock). *Let $\langle T, \sigma\rangle$ be a reachable stable configuration of a transformed program, $\alpha \in dom(\sigma)$, and $\xi \in T$ a stack with $\xi = \xi' \circ (\tau, stm)$. Then*

$$isfree(T \backslash \{\xi\}, \alpha) \quad \textit{iff} \quad \sigma(\alpha)(\mathsf{lock}) = (nil, 0) \vee \sigma(\alpha)(\mathsf{lock}) \leq \tau(\mathsf{id}) \ .$$

The auxiliary boolean instance variable started represents the semantic function *started* and states whether there is a thread in the global configuration which started its execution in the given object. For each object, started is initialized to *false*. Bracketed sections at the beginning of the main-method and at the beginning of start-methods contain the assignment started := true. The following lemma states that the variable started adequately represents the predicate *started*.

Lemma 6 (Started). *For all reachable stable configurations $\langle T, \sigma\rangle$ of a transformed program and all objects $\alpha \in dom(\sigma)$,*

$$started(T, \alpha) \quad \textit{iff} \quad \sigma(\alpha)(\mathsf{started}) \ .$$

The proof system of Section 4.2 generates verification conditions assuring invariance of assertions under the execution of statements, indeed of enabled statements. Now, in the transformed semantics with its bracketed sections, enabledness of a statement is a *global* notion, as it depends on whether the global configuration is stable or not. In order not to stipulate too strong proof obligations and thus loose completeness of the proof system, invariance at the level of local proof obligations needs to be shown only if there exists a corresponding global state with the statement enabled.

Ordinary statements outside bracketed sections are enabled in stable configurations, only. Nevertheless, concentrating on the local verification conditions for a single thread visiting a single object α, the fact whether another thread executing exclusively *outside* of α is currently at a *non-interleaving* point is immaterial for the local proof obligations. It is immaterial, as, from the perspective of the thread for which we formulate the verification conditions, the next stable configuration after the non-interleaving section as well as the one in front of it are identical with the globally instable one in between wrt. the instance state of α, since the threads do not have common variables, neither local ones nor instance variables. If a second thread is currently at a *non-interleaving* point and visits the *same object*, the situation is similar. For non-interleaving points immediately after object creation $\langle \mathsf{new}; \vec{y} := \vec{e}\rangle$, the instable instance state after object creation is identical with the stable one just before the bracketed section. Also method calls and returns (cf. Table 7) *across different objects* can be handled analogously, since either the caller or the callee object is different from the object under current consideration. Hence again the stable global configuration either before or after the non-interleaving execution of the method call or return agrees with the instance state of α in between, and thus the invariance needs to be shown.

The only situation which cannot be argued away in this manner is for *self-calls* affecting the same object as for which we are formulating the local proof obligation: The non-stable configuration at the non-interleaving point in between the caller's and the callee's observation does not necessarily correspond to any stable configuration with identical instance local state. We must therefore explicitly exclude from the proof-conditions this case, lest to loose completeness of the proof method.

To be able to do so, we introduce for each class an auxiliary boolean instance variable stable, asserting the existence of a global stable configuration with corresponding instance state. Formally, we define the augmentation as follows: The initial value of stable is *true*. Bracketed sections of method call and return statements, representing the sending parts of communication, contain the assignments $\mathsf{stable} := (e_0 \neq \mathsf{this})$ and $\mathsf{stable} := (\mathsf{callerobj} \neq \mathsf{this})$, respectively, thereby distinguishing between self-calls and others. Correspondingly for the receiver part, the bracketed sections at the beginning of method bodies and those of receive statements include the assignment $\mathsf{stable} := \mathsf{true}$.

With this augmentation, we define the assertion $\mathsf{enabled}$ for multiple assignments $\vec{y} := \vec{e}$ as true for assignments in the bracketed sections attached to object creation, and as stable for assignments occurring outside bracketed sections. For assignments in bracketed sections accompanying communication, we define:

$$\begin{array}{lcl} \mathsf{enabled}(\vec{y}_1 := \vec{e}_1) & = & \mathsf{true} \\ \mathsf{enabled}(\vec{y}_2 := \vec{e}_2) & = & (\mathsf{callerobj} = \mathsf{this}) \rightarrow \neg\mathsf{stable} \\ \mathsf{enabled}(\vec{y}_3 := \vec{e}_3) & = & \mathsf{true} \\ \mathsf{enabled}(\vec{y}_4 := \vec{e}_4) & = & (e_0 = \mathsf{this}) \rightarrow \neg\mathsf{stable}\ , \end{array}$$

where e_0 specifies the callee object of the method invocation under consideration. That the assertion $\mathsf{enabled}$ accurately captures enabledness as seen from the local perspective of a single instance is expressed in the following lemma:

Lemma 7 (Enabled). *Let $\langle T, \sigma\rangle$ be a reachable configuration of a transformed program and $(\tau, stm_{ass}; stm)$ a local configuration in T where stm_{ass} is $\vec{y} := \vec{e}$ or $\langle \vec{y} := \vec{e}\rangle$. Let furthermore $\sigma_{inst} = \sigma(\tau(\mathsf{this}))$.*

1. *If $(\tau, stm_{ass}; stm)$ is enabled in $\langle T, \sigma\rangle$, then $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$.*
2. *If $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$, then $(\tau, stm_{ass}; stm)$ is enabled in some reachable $\langle T', \sigma'\rangle$ with $\sigma'(\tau(\mathsf{this})) = \sigma_{inst}$.*

The values of the auxiliary variables $\mathsf{callerobj}$, id, lock, $\mathsf{started}$, and stable are changed only in the bracketed sections as described above.

To specify invariant properties of the system, the transformed programs are *annotated* by attaching local assertions to each control point. Besides that, for each class c, the annotation defines a local assertion I_c called *class invariant* that expresses invariant properties of the instances of the class.[6] Finally, the *global invariant* $GI \in GAss$ specifies properties of communication between objects.

[6] Note that the notion of class invariant used, for instance, in [22] differs from our notion since they require the class invariant to hold only after the termination of the class constructor and to be preserved by whole method calls, but not necessarily in between.

Definition 1 (Annotation, proof outline). *An* annotation *of a transformed program associates with each control point in some method m of a class c a local assertion* $p \in LAss_{m,c}$. *Furthermore, it assigns to each class c a* class invariant $I_c \in LAss_{m,c}$ *which may refer only to the instance variables of c. Finally, the program is assigned a* global invariant $GI \in GAss$. *We require that in the annotation no free logical variables occur, and that for all qualified references E.x in GI with* $E \in GExp^c$, *all assignments to x in class c are enclosed in bracketed sections. An annotated transformation of prog, denoted by prog′, is called a* proof outline.

For annotated programs, we use the standard notation $\{p\}\ stm\ \{q\}$ to express that p and q are the *pre-* and *postconditions* of *stm*, i.e., the assertions in front of and after *stm*, and write $pre(stm)$ and $post(stm)$ to refer to them.

4.2 Proof system

The proof system formalizes a number of *verification conditions* which inductively ensure that for each *reachable* configuration $\langle T, \sigma \rangle$ and for each local configuration (τ, stm) in T the precondition of the statement *stm* is satisfied and the class invariants and the global invariant hold. More precisely, the global invariant is required to hold in reachable *stable* configurations only, since its satisfaction can be expected only if the auxiliary variables are up-to-date. To cover concurrency and communication, the verification conditions are grouped, as usual, into initial conditions, local correctness conditions, an interference freedom test, and a cooperation test.

A proof outline is *initially correct*, if the precondition of the main statement and the global invariant are satisfied in the initial configuration. *Local correctness* ensures that local properties of a thread are invariant under its own execution. This invariance can be guaranteed by local correctness conditions only if no communication or object creation takes place, since their effect depends on the communicated values and cannot be determined locally. They will be analyzed in the *cooperation test* whose conditions are formalized in the global language. The invariance of local properties of a thread that currently executes in a given object can also be influenced by other threads executing in the same object which possibly changes the instance state. The corresponding verification conditions are formalized in the *interference freedom test.*

Our proof method is *modular* in the sense that it allows for separate interference freedom and cooperation tests. This modularity, which in practice simplifies correctness proofs considerably, is obtained by disallowing the assignment of side-effect expressions to instance variables. Clearly, such assignments can be avoided by additional assignments to fresh local variables and thus at the expense of new interleaving points.

Before specifying the verification conditions for a proof outline, we first fix some auxiliary functions and notations. Let $\mathsf{InitVal}$ be a syntactical operator with interpretation $InitVal : Var \rightarrow Val$ assigning *true* to stable and the initial value of type t to each other variable $y \in Var^t$, i.e., *nil*, *false*, and 0 for class, boolean, and integer types, respectively, and analogously for compound types, where sequences are initially empty. Note that since $\mathsf{this} \notin Var$, the self-reference is not in the domain of *InitVal*. Given $IVar_c$ as the set of instance variables of class c and $z \in LVar^c$, then $\mathsf{InitState}(z)$ denotes the global assertion $z \neq \mathsf{nil} \wedge \bigwedge_{x \in IVar_c} z.x = \mathsf{InitVal}(x)$, expressing that the object denoted by z is in its initial instance state.

4.2.1 Initial correctness A proof outline is *initially correct*, if the precondition of the main statement is satisfied by the initial instance and local states, where id identifies the first local configuration of a thread, and all other variables have their initial values. Furthermore, the global invariant must be satisfied by the first reachable stable configuration, i.e., by the initial global state after the execution of the bracketed section at the beginning of the main-method.

Definition 2 (Initial correctness). *A proof outline is* initially correct, *if*

$$\models_{\mathcal{L}} pre(body_{\mathsf{main}})[(\mathsf{this}, 0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}] \ , \tag{1}$$

$$\models_{\mathcal{G}} \mathsf{InitState}(z) \wedge \forall z'(z' = \mathsf{nil} \vee z = z') \rightarrow GI[\vec{E}_2/z.\vec{y}_2] \ , \tag{2}$$

where $body_{\mathsf{main}} = \langle \vec{y}_2 := \vec{e}_2 \rangle; stm$ *is the body and* $\vec{y}$ *the local and instance variables of the main-method,* $\vec{E}_2 = \vec{e}_2[(\mathsf{this}, 0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}][z/\mathsf{this}]$*,* z *is of the type of the main class, and* $z' \in LVar^{\mathsf{Object}}$*.*

4.2.2 Local correctness A proof outline is *locally correct*, if the usual verification conditions [7] for standard sequential constructs hold. Especially, the precondition of an enabled assignment, as given in the proof-outline, must imply its postcondition after the execution of the assignment (cf. Equation (3)). Besides invariance under assignments, local correctness requires that all assertions of a class imply the class invariant:

Definition 3 (Local correctness). *A proof outline is* locally correct*, if for each class* c *with class invariant* I_c*, all multiple assignments* $\vec{y} := \vec{e}$*, and all assertions* p *in class* c*,*

$$\models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge \mathsf{enabled}(\vec{y} := \vec{e}) \rightarrow post(\vec{y} := \vec{e})[\vec{e}/\vec{y}] \tag{3}$$

$$\models_{\mathcal{L}} p \rightarrow I_c \ . \tag{4}$$

Note that we have no local verification conditions for communication and object creation statements. The postcondition of a receive statement expresses an *assumption* about the method's return value. Similarly, the precondition of a method body expresses an assumption about the actual parameters received and the postcondition of an object creation statement an assumption about the identity of the new object. These assumptions will be verified in the *cooperation test*.

Other threads concurrently executing in the same object may influence or *interfere with* the invariance of the local assertions. This is covered in the interference freedom test.

4.2.3 The interference freedom test Next we formalize conditions that ensure the invariance of local properties of a local configuration under the activities of others. Since we disallow qualified reference to instance variables in $Java_{MT}$, we only have to deal with the invariance of properties under the execution of statements within the *same* object. Containing only local variables, communication and object creation do not change the state of the executing objects. Thus we only have to take assignments into account. In the following let p and $\vec{y} := \vec{e}$ be an assertion and an assignment occurring in the same class of a program.

Satisfaction of an assertion describing a local property of a thread may clearly be affected by the execution of an assignment by a *different* thread in the same object, provided that not both belong to a synchronized method of the object. Note that this applies only for assertions at interleaving points, since control points within bracketed sections are protected against interleaving by another thread.[7] This situation covering shared-variable interaction between different threads is captured by the predicate $\mathsf{diff\_threads}(p, \vec{y} := \vec{e})$ defined as $\neg\mathsf{samethread}(\mathsf{id}, \mathsf{id}')$ if p is at an interleaving point and not both p and $\vec{y} := \vec{e}$ occur in a synchronized method, and by false otherwise (see page 20 for the definition of $\mathsf{samethread}$). The variable id represents the identity of the thread executing $\vec{y} := \vec{e}$ and id' the identity of the thread of p.

If, otherwise, the assertion describes the *same* thread that executes the assignment, the only *interleaving* points endangered are those waiting for a return value earlier in the current execution stack. In other words, an assignment belonging to a *reentrant* code segment can affect the precondition of a receive statement whose execution is suspended earlier in the same call chain. However, the assignment belonging to the *matching* return statement need not be considered. To express this kind of interference, we define $\mathsf{wait\_for\_ret}(p, \vec{y} := \vec{e})$ by $\mathsf{id}' < \mathsf{id}$ if p is the precondition of a receive statement and $\vec{y} := \vec{e}$ is not in the bracketed section of a return statement, by $\mathsf{callee}(\mathsf{id}') < \mathsf{id}$ if p is the precondition of a receive statement and $\vec{y} := \vec{e}$ is in the bracketed section of a return statement, and by false otherwise.

For *self-calls,* the auxiliary assignments at the caller interferes with the precondition of method body, since both reside in the same object. The case for return to the same object is analogous. Unlike the situation captured by $\mathsf{wait\_for\_ret}$, p here represents a non-interleaving point that has to be shown interference free. For method calls, we define $\mathsf{self\_call}(p, \vec{y} := \vec{e})$ by $\mathsf{id}' = \mathsf{callee}(\mathsf{id}) \wedge e_0 = \mathsf{this}$, if p is the precondition of a method m and $\vec{y} := \vec{e}$ occurs in a bracketed section invoking method m of e_0, and by false otherwise. For self-calls of the start-method, we use $\mathsf{id}' = (\mathsf{this}, 0)$ for identification, i.e., $\mathsf{self\_start}(p, \vec{y} := \vec{e})$ is $\mathsf{id}' = (\mathsf{this}, 0) \wedge e_0 = \mathsf{this}$, if p is the precondition of start and $\vec{y} := \vec{e}$ occurs in a bracketed section invoking the start-method of e_0. For all other assignments and assertions the predicate is false.

[7] Strictly speaking, interference in the same object by different threads can occur also when the start-method is executed by a self-call. This will be handled together with self-calls in general.

The case for returning is specified by the assertion $\mathsf{self\_ret}(p, \vec{y} := \vec{e})$ which is $\mathsf{id} = \mathsf{callee}(\mathsf{id}') \wedge e_0' = \mathsf{this}$ if $\vec{y} := \vec{e}$ occurs in the bracketed section of return in a method m and p is the postcondition of a receive statement which is preceded by the invocation of method m of e_0. In all other cases $\mathsf{self\_ret}(p, \vec{y} := \vec{e})$ is false. The expression e_0' denotes e_0 with every local variable u different from this is replaced by a fresh one u'.

Collecting the above cases, we define $\mathsf{interleavable}(p, \vec{y} := \vec{e})$ for assertions p and assignments $\vec{y} := \vec{e}$ in the same class by

$$\mathsf{diff\_threads}(p, \vec{y} := \vec{e}) \vee \mathsf{wait\_for\_ret}(p, \vec{y} := \vec{e}) \vee$$
$$\mathsf{self\_call}(p, \vec{y} := \vec{e}) \vee \mathsf{self\_start}(p, \vec{y} := \vec{e}) \vee \mathsf{self\_ret}(p, \vec{y} := \vec{e}) \ .$$

The interference freedom test assures invariance of a property under the execution of an assignment in the same object, if both local configurations are in a configuration in that the assignment is enabled. We use the assertion and the precondition of the assignment to express reachability of the given control points, where the predicate $\mathsf{interleavable}(p, stm)$ denotes that they are also reachable in a common computation. That an assignment $\vec{y} := \vec{e}$ can be enabled in the given instance local state is stated by the assertion $\mathsf{enabled}(\vec{y} := \vec{e})$, as defined on page 22.

Definition 4 (Interference freedom). *A proof outline is* interference free, *if for all classes c, all assignments $\vec{y} := \vec{e}$ and assertions p in c,*

$$\models_{\mathcal{L}} p' \wedge pre(\vec{y} := \vec{e}) \wedge \mathsf{this} = \mathsf{this}' \wedge \mathsf{interleavable}(p, \vec{y} := \vec{e}) \wedge \mathsf{enabled}(\vec{y} := \vec{e}) \rightarrow p'[\vec{e}/\vec{y}] \ , \tag{5}$$

where p' denotes p with all local variables u and this *replaced by some fresh local variables u' and* this', *respectively.*

4.2.4 The cooperation test Whereas the verification conditions associated with local correctness and interference freedom cover the effects of assignments, the *cooperation test* deals with method invocation and object creation. Since different objects may be involved, it is formulated in the global assertion language. Besides ensuring invariance of the global invariant over bracketed sections, it specifies conditions under which the local properties of the communicating partners, i.e., the postconditions of statements involving communication or object creation, are satisfied. We start with the cooperation test for method invocation.

In the following definition, the logical variable z denotes the object calling a method and z' refers to the callee. The cooperation test assures that the local assertions at both ends of the communication hold, immediately after the values have been communicated. When calling a method, the postcondition of the method invocation statement and the precondition of the invoked method's body must hold after passing the parameters (Equation (6)). In the stable global state prior to the call, we can assume that the global invariant, the precondition of the method invocation at the caller side, and the class invariant of the callee hold.

For synchronized methods, additionally the lock of the callee object is free, or the lock has been acquired in the call chain of the executing thread. This is expressed by the predicate $\mathsf{isfree}(z'.\mathsf{lock},\mathsf{id})$ defined as $z'.\mathsf{lock} = (\mathsf{nil},0) \vee z'.\mathsf{lock} \leq \mathsf{id}$, where id is the identity of the caller. Equation (7) works similarly, where the postconditions of the corresponding return- and receive-statements are required to hold after returning from a method. In the global state prior to the call the global invariant and the preconditions of the return and receive statements are assumed to hold.

The global invariant GI is not allowed to refer to instance variables whose values are changed outside bracketed sections. Consequently, it will be automatically invariant under the execution of statements outside bracketed sections. For the bracketed sections, however, the invariance must be shown as part of the cooperation test. A difference between the treatment of the local assertions and the global invariant is, that the latter does not necessarily hold immediately after communication, but only after the accompanying assignments to the auxiliary variables of both the caller and callee have been performed. This is reflected in the two substitutions applied to the global invariant on the right-hand sides of the implications. For instance in Equation (6), $GI[\vec{E}_2'/z'.\vec{y}_2][\vec{E}_1/z.\vec{y}_1]$ is the weakest precondition of GI wrt. the assignments $\vec{y}_1 := \vec{e}_1$ and $\vec{y}_2 := \vec{e}_2$, in this order. Note that the order in which the syntactic substitutions are applied to GI is reverse compared with the order in which the corresponding assignments update the state.

Invoking the start-method of an object whose thread is already started, or returning from a start-method or from the first execution of the main-method does not have communication effects; Equations (8) and (9) take care about the validity of the postconditions and the invariance of the global invariant.

Definition 5 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test *for* communication*, if for all classes c and all statements $\langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1\rangle; \langle \mathsf{receive}\, v; \vec{y}_4 := \vec{e}_4\rangle$ in c with $e_0 \in Exp_c^{c'}$, Equations (6) and (7) hold, where m is a synchronized method of c' with $body_{m,c'} = \langle \vec{y}_2 := \vec{e}_2\rangle; stm; \langle \mathsf{return}\, e_{ret}; \vec{y}_3 := \vec{e}_3\rangle$, formal parameter list $\vec{u}$, and local variables $\vec{v}$ except the formal parameters and* this*.*

$$\begin{aligned}
\models_{\mathcal{G}}\ & GI \wedge pre(e_0.m(\vec{e}))[z/\mathsf{this}] \wedge I_{c'}[z'/\mathsf{this}] \wedge \\
& \quad e_0[z/\mathsf{this}] = z' \wedge \mathsf{isfree}(z'.\mathsf{lock},\mathsf{id}) \wedge z \neq \mathsf{nil} \wedge z' \neq \mathsf{nil} \\
& \to post(e_0.m(\vec{e}))[z/\mathsf{this}] \wedge pre'(body_{m,c'})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge \\
& \quad GI[\vec{E}_2'/z'.\vec{y}_2][\vec{E}_1/z.\vec{y}_1] \qquad (6) \\
\models_{\mathcal{G}}\ & GI \wedge pre'(\mathsf{return}\, e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge pre(\mathsf{receive}\, v)[z/\mathsf{this}] \wedge \\
& \quad e_0[z/\mathsf{this}] = z' \wedge z \neq \mathsf{nil} \wedge z' \neq \mathsf{nil} \\
& \to post'(\mathsf{return}\, e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge post(\mathsf{receive}\, v)[z, E_{ret}'/\mathsf{this}, v] \wedge \\
& \quad GI[\vec{E}_4/z.\vec{y}_4][\vec{E}_3'/z'.\vec{y}_3]\ . \qquad (7)
\end{aligned}$$

In the equations, $z \in LVar^c$ and $z' \in LVar^{c'}$ are distinct fresh logical variables and local variables are viewed as logical ones on the global level. We define

$pre'(body_{m,c'}) = pre(body_{m,c'})[\mathsf{InitVal}(\vec{v})/\vec{v}]$, $\vec{e}_2' = \vec{e}_2[\mathsf{InitVal}(\vec{v})/\vec{v}]$, and e_{ret}', $\vec{e}_3'$, $pre'(\mathsf{return}\ e_{ret})$, and $post'(\mathsf{return}\ e_{ret})$ denote the given expressions and assertions with every local variable except the formal parameters and this *replaced by a fresh one. Furthermore, $\vec{E}_1 = \vec{e}_1[z/\mathsf{this}]$, $\vec{E}_i' = \vec{e}_i'[z', \vec{E}/\mathsf{this}, \vec{u}]$ for $i = 2, 3$, $\vec{E}_4 = \vec{e}_4[z, E_{ret}'/\mathsf{this}, v]$, where $\vec{E} = \vec{e}[z/\mathsf{this}]$ and $E_{ret}' = e_{ret}'[z', \vec{E}/\mathsf{this}, \vec{u}]$. For* non-synchronized methods, *the antecedent* $\mathsf{isfree}(z'.\mathsf{lock}, \mathsf{id})$ *is dropped. The verification conditions for methods without return value are analogous.*

For invocations of start*-methods, only (6) applies with the additional antecedent $\neg z'.\mathsf{started}$. For the case that the thread is already started,*

$$\begin{aligned}\models_{\mathcal{G}}\ & GI \wedge pre(e_0.\mathsf{start}(\vec{e}))[z/\mathsf{this}] \wedge I_{c'}[z'/\mathsf{this}] \wedge \\ & \quad e_0[z/\mathsf{this}] = z' \wedge z'.\mathsf{started} \wedge z \neq \mathsf{nil} \wedge z' \neq \mathsf{nil} \\ & \to post(e_0.\mathsf{start}(\vec{e}))[z/\mathsf{this}] \wedge GI[\vec{E}_1/z.\vec{y}_1] \end{aligned} \tag{8}$$

has to be satisfied. Finally, for statements $\langle \mathsf{return}; \vec{y}_3 := \vec{e}_3 \rangle$ in the main-method or in a start-method,

$$\begin{aligned}\models_{\mathcal{G}}\ & GI \wedge pre(\mathsf{return})[z'/\mathsf{this}] \wedge \mathsf{id} = (z', 0) \wedge z' \neq \mathsf{nil} \\ & \to post(\mathsf{return})[z'/\mathsf{this}] \wedge GI[\vec{E}_3/z'.\vec{y}_3]\ . \end{aligned} \tag{9}$$

Note that we replace the local variables u of the callee by fresh ones denoted by u' in order to avoid name clashes with local variables of the caller. The resulting assertions and expressions we denote by a primed version.

The substitution of $\vec{u}$ by $\vec{E}$ in the condition $pre'(body_{m,c'})[z', \vec{E}/\mathsf{this}, \vec{u}]$ reflects the parameter-passing mechanism, where $\vec{E}$ are the actual parameters $\vec{e}$ represented at the global assertional level. This substitution also identifies the callee, as specified by its formal parameter id. Note that the actual parameters do not contain *instance* variables, i.e., their interpretation does not change during the execution of the method body. Therefore, $\vec{E}$ can be used not only to logically capture the conditions at the entry of the method body, but at the exit of the method body, as well, as shown in Equation (7).

Besides method calls and return, the cooperation test needs to handle bracketed sections containing object creation statements, taking care of the preservation of the global invariant, the postcondition of the new-statement, and the new object's class invariant. We can assume that the precondition of the object creation statement and the global invariant hold in the stable configuration prior to the instantiation. The extension of the global state with a freshly created object is formulated in a strongest postcondition style, i.e., it is requested to hold immediately *after* the instantiation. We use existential quantification to refer to the old value: z' of type $LVar^{\mathsf{list\ Object}}$ represents the existing objects prior to the extension. Moreover, that the created object's identity stored in u is fresh and that the new instance is properly initialized is captured by the global assertion $\mathsf{Fresh}(z', u)$ defined as $\mathsf{InitState}(u) \wedge u \notin z' \wedge \forall v(v \in z' \vee v = u)$, where $\mathsf{InitState}(u)$ is as defined in Section 4.2. To express that an assertion refers to the set of existing objects *before* the new-statement, we need to *restrict* any existential quantification to range over objects from z', only. So let P be a global assertion and $z' \in LVar^{\mathsf{list\ Object}}$ a logical variable not occurring in P. Then $P \downarrow z'$ is the global assertion P with all quantifications $\exists z(P')$ replaced by $\exists z(\mathsf{within}(z, z') \wedge P')$, where the semantic interpretation $within(v, v')$ for object sequences $v' \in Val^{\mathsf{list\ Object}}$ and arbitrary values $v \in Val$ is defined recursively by

$$within(v, v') = \begin{cases} true & \text{if } v \in Val^{\mathsf{Bool}} \cup Val^{\mathsf{Int}} \\ v \in v' & \text{if } v \in \bigcup_c Val^c_{nil} \\ within(v_1, v') \wedge within(v_2, v') & \text{if } v{=}(v_1, v_2){\in} \bigcup_{t_1,t_2} Val^{t_1 \times t_2}_{nil} \\ \forall v_i \in v(within(v_i, v')) & \text{if } v \in \bigcup_t Val^{\mathsf{list}\, t}_{nil} \,. \end{cases}$$

The following lemma formulates the basic property of the projection operator:

Lemma 8. *Assume a global state σ, an extension $\sigma' = \sigma[\alpha \mapsto \sigma^{c,init}_{inst}]$ for some $\alpha \in Val^c$, $\alpha \notin dom(\sigma)$, and a logical environment ω referring only to values existing in σ. Let v be the sequence consisting of all elements of $\bigcup_c dom^c_{nil}(\sigma)$. Then for all global assertions P and logical variables $z' \in LVar^{\mathsf{list\ Object}}$ not occurring in P,*

$$\omega, \sigma \models_{\mathcal{G}} P \text{ iff } \omega[z' \mapsto v], \sigma' \models_{\mathcal{G}} P \downarrow z'.$$

Thus the predicates $GI \downarrow z'$ and $\exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}]) \downarrow z'$ express that the global invariant and the precondition of the object creation statement hold for the old value of u prior to the creation of the new object.

This leads to the following definition of the cooperation test for object creation:

Definition 6 (Cooperation test: Instantiation). *A proof outline satisfies the* cooperation test *for* object creation, *if for all classes c' and statements $\langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle$ in c':*

$$\begin{aligned} \models_{\mathcal{G}} z \neq \mathsf{nil} \wedge \exists z'\Big(&\mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}])) \downarrow z'\Big) \qquad (10)\\ &\to post(u := \mathsf{new}^c)[z/\mathsf{this}] \wedge I_c[u/\mathsf{this}] \wedge GI[\vec{E}/z.\vec{y}]\,, \end{aligned}$$

with fresh logical variables $z \in LVar^{c'}$ and $z' \in LVar^{\mathsf{list\ Object}}$, and $\vec{E} = \vec{e}[z/\mathsf{this}]$.

5 Soundness and completeness

This section contains soundness and completeness of the proof method of Section 4. Given a program together with its annotation, the proof system stipulates a number of induction conditions for the various types of assertions and program constructs. *Soundness* for the inductive method means that for a proof outline satisfying the verification conditions, all configurations reachable in the operational semantics satisfy the given assertions, *completeness* conversely means that if a program does satisfy an annotation, this is provable. For convenience, let us introduce the following notations. Given a program *prog*, we will write φ_{prog} or just φ for its annotation, and write $prog \models \varphi$, if *prog* satisfies all requirements stated in the assertions, more precisely, satisfaction of the assertions for all reachable configurations, where in case of the global invariant, satisfaction is required for stable configurations, only:

Definition 7. *Given a program prog with annotation φ, then $prog \models \varphi$ iff for all reachable configurations $\langle T, \sigma\rangle$ of prog, for all $(\tau, stm) \in T$ with $\alpha = \tau(\mathsf{this})$, and for all logical environments ω referring only to values existing in σ:*

1. $\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} pre(stm)$, *and*
2. *if $\langle T, \sigma\rangle$ is stable, then* $\omega, \sigma \models_{\mathcal{G}} GI$.

Furthermore, for all classes c, objects $\beta \in dom^c(\sigma)$, and local states τ':

3. $\omega, \sigma(\beta), \tau' \models_{\mathcal{L}} I_c$.

The definition is applied both to transformed and original programs. For proof outlines, i.e., annotated transformed programs, we write $prog' \vdash \varphi'$ iff $prog'$ satisfies the verification conditions of the proof system.

5.1 Soundness

Soundness, as mentioned, means that all reachable configurations do satisfy their assertions for an annotated program that has been verified using the proof conditions. Soundness of the method is proved by a straightforward, albeit rather tedious induction on the computation steps.

Before embarking upon the soundness formulation and its proof, we need to clarify the connection between the original program and the transformed one, i.e., the one decorated with assertions, extended by auxiliary variables, sprinkled with bracketed sections, and transformed as far as the method calls are concerned (cf. Section 4.1). The transformation is done for the sake of verification, only, and as far as the un-augmented portion of the states and the configurations is concerned, the behavior of the original and the transformed program are the same, modulo some additional non-interleaving points caused by the transformation.

To make the connection between original program and transformed one precise, we define a projection operation $\downarrow prog$, that jettisons all additions of the transformation. So let $prog'$ a transformation of *prog*, and $\langle T', \sigma'\rangle$ a global configuration of $prog'$. Then $\sigma' \downarrow prog$ is defined by removing all auxiliary instance variables from the instance state domains. For the set of thread configurations $T' \downarrow prog$ is given by restricting the domains of the local states to non-auxiliary variables, removing all annotations, augmentations, and bracketed sections, and transforming back the explicit communication statements to $Java_{MT}$ syntax. The following lemma expresses that the transformation does not change the behavior of programs:

Lemma 9 (Transformation). *Let $prog'$ be a transformation of a program prog. Then $\langle T, \sigma\rangle$ is a reachable configuration of prog iff there exists a reachable configuration $\langle T', \sigma'\rangle$ of $prog'$ with $\langle T' \downarrow prog, \sigma' \downarrow prog\rangle = \langle T, \sigma\rangle$.*

Let *prog* be a program with annotation φ, and $prog'$ a transformation of *prog* with annotation φ'. Let GI' be the global invariant of φ', I'_c denote its class invariants, and for an assertion p of φ let p' denote the assertion of φ' associated with the same control point. We write $\models \varphi' \to \varphi$ iff $\models_{\mathcal{G}} GI' \to GI$, $\models_{\mathcal{L}} I'_c \to I_c$ for all classes c, and $\models_{\mathcal{L}} p' \to p$, for all assertions p of φ associated with some control point. To give meaning to the auxiliary variables, the above implications are evaluated in the context of states of the transformed program. The following theorem states the soundness of the proof method.

Theorem 1 (Soundness). *Given a proof outline $prog'$ with annotation $\varphi_{prog'}$.*

$$\textit{If} \quad prog' \vdash \varphi_{prog'} \quad \textit{then} \quad prog' \models \varphi_{prog'} \ .$$

The soundness proof is contained in the appendix in Section B.2, basically an induction on the length of computation, simultaneous on all three parts from the definition of satisfaction (Definition 7). The property of Theorem 1 is formulated for reachability of transformed programs. With the help of the transformation Lemma 9, we immediately get:

Corollary 1. *If $prog' \vdash \varphi_{prog'}$ and $\models \varphi'_{prog'} \to \varphi_{prog}$, then $prog \models \varphi_{prog}$.*

5.2 Completeness

Next we conversely show that if a program satisfies the requirements asserted in its proof outline, then this is indeed provable, i.e., then there exists a proof outline which can be shown to hold and which implies the given one:

$$\forall prog.\ prog \models \varphi_{prog} \ \Rightarrow\ \exists prog'.\ prog' \vdash \varphi_{prog'} \wedge\ \models \varphi_{prog'} \to \varphi_{prog} \ .$$

Given a program satisfying an annotation $prog \models \varphi_{prog}$, the consequent can be uniformly shown, i.e., independently of the given assertional part φ_{prog}, by instantiating $\varphi_{prog'}$ to the strongest annotation still provable, thereby discharging the last clause $\models \varphi_{prog'} \to \varphi_{prog}$. Since the strongest annotation still satisfied by the program corresponds to reachability, the key to completeness is to

1. augment each program with enough information, to be able to
2. express reachability in the annotation, i.e., annotate the program such that a configuration satisfies its local and global assertions exactly if reachable (see Definition 9 below), and finally
3. to show that this augmentation indeed satisfies the verification conditions.

We begin with the augmentation, using the transformation from Section 4.1 as starting point, where method invocation statements are replaced by method call and receive statements, the programs are augmented with the specific auxiliary variables lock, $\mathsf{started}$, stable, $\mathsf{callerobj}$, and id, and finally equipped with bracketed sections.

Now to make visible within a configuration whether or not it is reachable, the standard trick is to add information into the states about the way it has been reached, i.e., the *history* of the computation leading to the configuration. It is recorded in history variables, containing enough information to distinguish reachable from unreachable configurations.

The assertional language is split into a local and a global level, and likewise the proof-system is tailored to separate local proof obligations from global ones to obtain a modular proof system. The history will be recorded in instance variables, and thus each instance can keep track only of its own past. To mirror the split into a local and a global level in the proof system, the history per instance is recorded separately for *internal* and *external* behavior. The sequence of internal state changes local to that instance are recorded in the *local* history and the external behavior in the communication history.

The communication history keeps information about the kind of communication, the communicated values, and the identity (both object and local configuration identities) of the communication partners involved. For the kind of communication, we distinguish as cases object creation, ingoing and outgoing method calls, and likewise ingoing and outgoing communication for the return value. We use the set of constants $\{\mathsf{new}, \mathsf{call}, \mathsf{called}, \mathsf{return}, \mathsf{receive}\}$ for this purpose. Note in passing that the information stored in the communication history matches exactly the information needed to decorate the transitions in order to obtain a compositional variant of the operational semantics of Section 2.3.2. See [3] for such a compositional semantics.

To facilitate reasoning, we introduce an additional auxiliary local variable loc, which stores the current control point of the execution of a thread. Given a function which assigns to all control points unique location labels, we extend each assignment $\vec{y} := \vec{e}$ by the update of the variable loc to $\vec{y}, \mathsf{loc} := \vec{e}, l$, where l is the label of the control point after the given occurrence of the assignment. We extend bracketed sections which do not contain assignments by $\mathsf{loc} := l$, where l is the label of the control point following the bracketed section. We write $l \equiv stm$ if l represents the control point in front of stm in a method body $stm'; stm$.

Definition 8 (Augmentation with histories). *Each class is further extended by two auxiliary instance variables h_{inst} and h_{comm}, both initialized to the empty sequence. They are updated as follows:*

1. *Each multiple assignment $\vec{y} := \vec{e}$ in a class c is replaced by*

$$\vec{y}, \mathsf{h}_{inst} := \vec{e}, \mathsf{h}_{inst} \circ ((\vec{x}, \vec{u})[\vec{e}/\vec{y}]) ,$$

   where $\vec{x}$ are the instance variables of class c containing also h_{comm} but without h_{inst}, and $\vec{u}$ are the local variables of the executing thread including this.
2. *Every bracketed section $\langle stm; \vec{y} := \vec{e}\rangle$ is extended to*

$$\langle stm; \vec{y}, \mathsf{h}_{comm} := \vec{e}, \mathsf{h}_{comm} \circ (\mathsf{kind}, \mathsf{id}, \mathsf{partner}, \mathsf{values})\rangle .$$

   The value of kind *is* (new, c) *for bracketed sections creating an object of type c,* (call, m) *for bracketed sections invoking method m of an object,* (called, m)

for the bracketed section at the beginning of a method m*,* (return, m) *for returning from method* m*, and* $(\mathsf{receive}, m)$ *for receiving the return value from a method* m*. The communication partner* partner *is given by* e_0 *for method invocation* $e_0.m(\vec{e})$ *and its subsequent receive statement, if any, and by* callerobj *for bracketed sections at the beginning of method bodies and for return statements. The sequence* values *contain the actual parameters for method call, the formal parameters for bracketed sections at the beginning of method bodies, the return value for return statements, and the received value for receive statements. In the case of object creation* partner *is* nil*, and* values *is the identity of the new object, if it is assigned to some variable, and the empty sequence otherwise.*

In the update of the history variable h_{inst}, the expression $(\vec{x}, \vec{u})[\vec{e}/\vec{y}]$ identifies the active thread by the local variable id, and specifies its instance local state after the execution of the assignment. Note that especially the values of the auxiliary variables introduced in the program transformation are recorded in the history h_{inst}. In the following we will also write (σ_{inst}, τ) when referring to elements of h_{inst}. For a non-empty sequence h we define $head(h)$ as the sequence without its last element and $tail(h)$ as the last element of the sequence.

We introduce the following annotation for the transformed program:

Definition 9 (Reachability annotation).

1. *For each class* c *of the transformed program we define* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} I^c$ *iff there exists a reachable configuration* $\langle T, \sigma\rangle$ *of the program such that* $\sigma(\tau(\mathsf{this})) = \sigma_{inst}$*;*
2. *We define* $post(body_{m,c}) = I_c$ *for each class* c *and method* m *of* c*. For all other control points in front of a statement* stm *we define* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(stm)$ *iff there exists a reachable configuration* $\langle T, \sigma\rangle$ *of the program with a local configuration* $(\tau, stm; stm_2)$ *in* T*, with* $\sigma(\tau(\mathsf{this})) = \sigma_{inst}$*;*
3. *Finally,* $\omega, \sigma \models_{\mathcal{G}} GI$ *iff there exists a reachable stable configuration* $\langle T, \sigma'\rangle$ *of the program such that* $dom(\sigma) = dom(\sigma')$*, and for all objects* $\alpha \in dom(\sigma)$*,* $\sigma(\alpha)(\mathsf{h}_{comm}) = \sigma'(\alpha)(\mathsf{h}_{comm})$*.*

It can be shown that these assertions are expressible in the assertion language [33]. The transformed program together with the above annotation build a proof outline that we denote by *prog'*.

What remains to be shown for completeness is that the proof-outline *prog'* indeed satisfies the verification conditions of the proof system. Initial and local correctness are straightforward, where for local correctness we use the fact of Lemma 7 that the enabled-predicate used in the local correctness condition captures enabledness from the perspective of an instance. The full proofs are shown in the appendix in Section B.3.

Completeness for the interference freedom test and the cooperation test are more complex, since, unlike initial and local correctness, the verification conditions in these cases mention more than one local configuration in the assertions of their respective antecedents. Now, the reachability assertions of *prog'* guarantee that, when satisfied by an instance local state, there *exists* a reachable global configuration responsible for the satisfaction. So a crucial step in the completeness proof for interference freedom and the cooperation test is to show that individual reachability of two local configurations implies that they are reachable in a *common* computation. This is also the key property for the history variables: they record enough information such that they allow to uniquely determine the way a configuration has been reached; in the case of instance history, uniqueness of course, only as far as the instance under consideration is concerned. This property is stated formally in the following local merging lemma, where the global configurations are required to be *stable in the object*, so that the history variable indeed contains an up-to-date representation of all steps performed within the instance.

Lemma 10 (Local merging lemma). *Let $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$ be two reachable global configurations of prog′ and $(\tau, stm) \in T_1$, such that both $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$ are stable in $\tau(\mathsf{this}) \in dom(\sigma_1) \cap dom(\sigma_2)$. Then $\sigma_1(\tau(\mathsf{this}))(\mathsf{h}_{inst}) = \sigma_2(\tau(\mathsf{this}))(\mathsf{h}_{inst})$ implies $(\tau, stm) \in T_2$.*

For completeness of the cooperation test, connecting two possibly different instances, we need an analogous property for the communication histories. Arguing on the global level, the cooperation test can assume that two control points are individually reachable but agreeing on the communication histories of the objects. This information must be enough to ensure common reachability. Such a common computation can be constructed, since the internal computations of different objects are independent from each other, i.e., in a global computation, the local behavior of an object is interchangeable, as long as the external behavior does not change. This leads to the following lemma:

Lemma 11 (Global merging lemma). *Let $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$ be two reachable stable global configurations of prog′ and $\alpha \in dom(\sigma_1) \cap dom(\sigma_2)$ with $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$. Then there exists a reachable stable configuration $\langle T, \sigma\rangle$ with $\sigma(\alpha) = \sigma_1(\alpha)$, and $\sigma(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2) \setminus \{\alpha\}$.*

Note that together with the local merging lemma this implies that all local configurations of α in $\langle T_1, \sigma_1\rangle$ all local configurations of $\beta \neq \alpha$ in $\langle T_2, \sigma_2\rangle$ are contained in the commonly reached configuration $\langle T, \sigma\rangle$.

This brings us to the last result of the paper:

Theorem 2 (Completeness). *Given a program prog, the proof outline prog′ satisfies the verification conditions of the proof system from Section 4.2.*

6 Conclusion

Related work This paper presents the first sound and complete assertional proof method for a multithreaded sublanguage of *Java*. In [2] the basic ideas have been introduced for proof outlines by means of a modular integration of the interference freedom and the cooperation test for a more restricted version of

Java. The present paper offers such an integration for a more concrete version of *Java* by incorporating *Java*'s *reentrant* synchronization mechanism. This requires a non-trivial extension of the proof method by a more refined mechanism for the identification of threads.

Most papers in the literature focus on *sequential* subsets of Java [30, 12, 10, 28, 29, 13, 34, 1, 35, 36]. Formal semantics of *Java*, including multithreaded execution, and its virtual machine in terms of abstract state machines is given in [31]. A structural operational semantics of multithreaded *Java* can be found in [14].

Future work In the context of the bilateral NWO/DFG project MobiJ and the European Fifth Framework RTD project Omega we are currently developing a front-end tool for the computer-aided specification and verification of *Java* programs based on our proof method. Such a front-end tool consists of an editor and a parser for annotating *Java* programs, and of a compiler which generates corresponding proof obligations. A theorem prover (HOL or PVS) is used for verifying the validity of these verifications conditions. Of particular interest in this context is an integration of our method with related approaches like the Loop project [19, 25].

As future work, we plan to extend $Java_{MT}$ by further constructs, especially adding further synchronization primitives for monitor synchronization such as wait and notify, but also extending the language in the direction of "object-orientedness", adding inheritance, subtyping, and other concepts featured in *Java*. To deal with subtyping on the logical level requires a notion of behavioral subtyping [5].

Acknowledgments We thank Marcel Kyas and Cees Pierik for fruitful discussions and suggestions. This work was partly supported by the NWO/DFG project MobiJ and the European Fifth Framework RTD project Omega.

References

1. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In Bidoit and Dauchet [11], pages 682–696. An extended version of this paper appeared as SRC Research Report 161 (September 1998).
2. E. Ábrahám-Mumm and F. de Boer. Proof-outlines for threads in Java. In Palamidessi [27].
3. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. A compositional semantics for $java_{mt}$. Technical report, Lehrstuhl für Software-Technologie, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Mar., 2002.
4. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS State-of-the-Art-Survey. Springer-Verlag, 1999.
5. P. America. A behavioural approach to subtyping in object-oriented programming languages. 443, Phillips Research Laboratories, January/April 1989.
6. P. America and F. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1993.

7. K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.
8. K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.
9. I. Attali and T. Jensen, editors. *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, 2001.
10. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In Attali and Jensen [9], pages 6–24.
11. M. Bidoit and M. Dauchet, editors. *Theory and Practice of Software Development, Proceedings of the 7th International Joint Conference of CAAP/FASE, TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, Lille, France, Apr. 1997. Springer-Verlag.
12. R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* PhD thesis, Universität Passau, 1991. See also Springer LNCS 562.
13. P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, Mar. 1995.
14. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [4].
15. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods.* Cambridge University Press, 2001.
16. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
17. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
18. C. Hankin, editor. *Programming Languages and Systems: Proceedings of the 7th European Symposium on Programming (ESOP '98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal, March/April 1998)*, volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
19. J. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Hankin [18].
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [21].
21. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.
22. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.
23. H. Hussmann, editor. *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
24. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
25. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.
26. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
27. C. Palamidessi, editor. *CONCUR 2000: Concurrency Theory (11th International Conference, University Park, PA, USA)*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 2000.

28. A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* Technische Universität München, Jan. 1997. Habilitationsschrift.
29. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In Swierstra [32], pages 162–176.
30. B. Reus, R. Hennicker, and M. Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Hussmann [23], pages 300–316.
31. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine.* Springer-Verlag, 2001.
32. S. Swierstra, editor. *Proceedings of the 8th European Symposium on Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science.* Springer, 1999.
33. J. V. Tucker and J. I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, volume 6 of *CWI Monograph Series.* North-Holland, 1988.
34. D. von Oheimb. Axiomatic sematics for Java^{light} in Isabelle/HOL. In S. Drossopoulo, S. Eisenbach, B. Jacobs, G. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, number 269, 5/2000 in Technical Report. Fernuniversität Hagen, 2000.
35. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency – Practice and Experience*, 2001. to appear.
36. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. submitted for publication, 2002.
37. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml.* Object Technology Series. Addison-Wesley, 1999.

A Semantics of transformed programs

Section 4 describes how to transform programs to reason about their properties. This transformation implies slight changes in the semantics. The operational semantics of transformed programs is given in the Tables 8 and 9.

Multiple assignments are executed simultaneously (Ass). Method invocation consists of three steps executed without interleaving with other threads not involved in the method invocation: First, a new local configuration is created ready to execute the method body and the actual parameters are passed on (Call), afterwards, the caller thread executes the multiple assignment in the bracketed section of the output statement ($\textsc{Ass}^1_{crit}$), and third, the bracketed section at the beginning of the method body is executed ($\textsc{Ass}^2_{crit}$). For the invocation of start-methods, the rules Start, $\textsc{Ass}^4_{crit}$, and $\textsc{Ass}^2_{crit}$ are used, instead. If the thread is already started, only one local configuration is involved, and only the rules Start and $\textsc{Ass}^2_{crit}$ apply. Analogously, the semantics of return in a transformed program is defined by substituting the return value in a first step (Return), executing the callee-assignment in a second step ($\textsc{Ass}^3_{crit}$), and, finally, executing the caller-assignment ($\textsc{Ass}^2_{crit}$). In the case of termination no communication takes place, and only rules Terminate and $\textsc{Ass}^2_{crit}$ apply.

Since the stability of a global configuration depends only on the thread configurations, we use here the predicate *stable*, defined in Section 4.1 for global configurations, also for sets of thread configurations. Note that the statement of a local configuration represents a non-interleaving control point if and only if its statement begins with a bracketed section containing only a multiple assignment. Lemma 9 shows that reachable configurations of a program *prog* correspond to reachable configurations of its transformation *prog'* in that all control points are interleaving points, i.e., all started communications are completed.

The transitions for the remaining sequential constructs are standard and omitted.

$$\frac{T \text{ stable} \qquad \beta \in Val^c \setminus dom^c(\sigma) \qquad \tau' = \tau[u \mapsto \beta] \qquad \sigma' = \sigma[\beta \mapsto \sigma_{inst}^{c,init}]}{\langle T \,\dot\cup\, \{\xi \circ (\tau, \langle u{:=}\mathsf{new}^c; \vec{y}{:=}\vec{e}\rangle; stm)\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\tau', \langle \vec{y}{:=}\vec{e}\rangle; stm)\}, \sigma'\rangle} \text{ New}$$

$$\frac{\begin{array}{c} T \text{ stable} \qquad \tau(\mathsf{this}) = \alpha \qquad \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in dom^c(\sigma) \\ \tau' = \tau_{init}^{\mathsf{start},c}[\mathsf{this} \mapsto \beta][\vec{u} \mapsto [\![\vec{e}']\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \\ \neg started(T \cup \{\xi \circ (\tau, \langle e_0.\mathsf{start}(\vec{e}'); \vec{y} := \vec{e}\rangle; stm)\}, \beta) \end{array}}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\tau, \langle e_0.\mathsf{start}(\vec{e}'); \vec{y} := \vec{e}\rangle; stm)\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\tau, \langle \vec{y} := \vec{e}\rangle; stm), (\tau', body_{\mathsf{start},c})\}, \sigma\rangle \end{array}} \text{ Start}$$

$$\frac{\begin{array}{c} T \text{ stable} \qquad \tau(\mathsf{this}) = \alpha \qquad \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in dom^c(\sigma) \\ started(T \cup \{\xi \circ (\tau, \langle e_0.\mathsf{start}(\vec{e}'); \vec{y} := \vec{e}\rangle; stm)\}, \beta) \end{array}}{\langle T \dot\cup \{\xi \circ (\tau, \langle e_0.\mathsf{start}(\vec{e}'); \vec{y}{:=}\vec{e}\rangle; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau, \langle \vec{y}{:=}\vec{e}\rangle; stm)\}, \sigma\rangle} \text{ Start}_{skip}$$

$$\frac{\begin{array}{c} T \text{ stable} \qquad \tau(\mathsf{this}) = \alpha \qquad \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in dom^c(\sigma) \\ modif\ m(\vec{u})\{body_{m,c}\} \in Meth_c \qquad m \neq \mathsf{start} \qquad sync(c,m) \rightarrow isfree(T,\beta) \\ \tau' = \tau_{init}^{m,c}[\mathsf{this} \mapsto \beta][\vec{u} \mapsto [\![\vec{e}']\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \end{array}}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\tau, \langle e_0.m(\vec{e}'); \vec{y} := \vec{e}\rangle; stm)\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\tau, \langle \vec{y} := \vec{e}\rangle; stm) \circ (\tau', body_{m,c})\}, \sigma\rangle \end{array}} \text{ Call}$$

$$\frac{T \text{ stable} \qquad \tau'(\mathsf{this}) = \beta \qquad \tau'' = \tau[u \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\beta),\tau'}]}{\begin{array}{l} \langle T \,\dot\cup\, \{\xi \circ (\tau, \langle \mathsf{receive}\ u; \vec{y}_4 := \vec{e}_4\rangle; stm) \circ (\tau', \langle \mathsf{return}\ e; \vec{y}_3 := \vec{e}_3\rangle)\}, \sigma\rangle \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\tau'', \langle \vec{y}_4 := \vec{e}_4\rangle; stm) \circ (\tau', \langle \vec{y}_3 := \vec{e}_3\rangle)\}, \sigma\rangle \end{array}} \text{ Return}$$

$$\frac{T \text{ stable}}{\langle T \,\dot\cup\, \{(\tau, \langle \mathsf{return}; \vec{y}_3 := \vec{e}_3\rangle)\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{(\tau, \langle \vec{y}_3 := \vec{e}_3\rangle)\}, \sigma\rangle} \text{ Terminate}$$

Table 8. Operational semantics of transformed programs I

$$\frac{T \text{ stable} \qquad \tau(\mathsf{this}) = \alpha \qquad \tau' = \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \qquad \sigma' = \sigma[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]}{\langle T \,\dot{\cup}\, \{\xi \circ (\tau, \vec{y} := \vec{e}; stm)\}, \sigma\rangle \longrightarrow \langle T \,\dot{\cup}\, \{\xi \circ (\tau', stm)\}, \sigma'\rangle} \,\mathrm{Ass}$$

$$\frac{stm' \neq \epsilon \qquad \tau(\mathsf{this}) = \alpha \qquad \tau'' = \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \qquad \sigma' = \sigma[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]}{\begin{array}{l}\langle T \dot{\cup}\{\xi \circ (\tau, \langle \vec{y} := \vec{e}\rangle; stm) \circ (\tau', \langle \vec{y}' := \vec{e}'\rangle; stm')\}, \sigma\rangle \longrightarrow \\ \langle T \dot{\cup}\{\xi \circ (\tau'', stm) \circ (\tau', \langle \vec{y}' := \vec{e}'\rangle; stm')\}, \sigma'\rangle\end{array}} \,\mathrm{Ass}^1_{crit}$$

$$\frac{\begin{array}{c}T \cup \{\xi\} \text{ stable} \qquad stm \neq \epsilon \vee \xi = \epsilon \qquad \tau(\mathsf{this}) = \alpha \\ \tau' = \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \qquad \sigma' = \sigma[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]\end{array}}{\langle T \dot{\cup}\{\xi \circ (\tau, \langle \vec{y} := \vec{e}\rangle; stm)\}, \sigma\rangle \longrightarrow \langle T \dot{\cup}\{\xi \circ (\tau', stm)\}, \sigma'\rangle} \,\mathrm{Ass}^2_{crit}$$

$$\frac{\xi \neq \epsilon \qquad \tau(\mathsf{this}) = \alpha \qquad \sigma' = \sigma[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]}{\langle T \dot{\cup}\{\xi \circ (\tau, \langle \vec{y} := \vec{e}\rangle)\}, \sigma\rangle \longrightarrow \langle T \dot{\cup}\{\xi\}, \sigma'\rangle} \,\mathrm{Ass}^3_{crit}$$

$$\frac{\tau(\mathsf{this}) = \alpha \qquad \tau'' = \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \qquad \sigma' = \sigma[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}]}{\begin{array}{l}\langle T \dot{\cup}\{\xi \circ (\tau, \langle \vec{y} := \vec{e}\rangle; stm), (\tau', body_{\mathsf{start},c})\}, \sigma\rangle \longrightarrow \\ \langle T \dot{\cup}\{\xi \circ (\tau'', stm), (\tau', body_{\mathsf{start},c})\}, \sigma'\rangle\end{array}} \,\mathrm{Ass}^4_{crit}$$

Table 9. Operational semantics of transformed programs II

B Proofs

B.1 Properties of substitutions

Proof (of Lemma 1). By straightforward induction on the structure of local expressions and assertions. In the case for local variables $u = y_i$ we get

$$\begin{aligned}
[\![u[\vec{e}/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} &= [\![e_i]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} \\
&= \tau[u \mapsto [\![e_i]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}](u) \\
&= [\![u]\!]_{\mathcal{L}}^{\omega,\sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}],\tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}]} \ .
\end{aligned}$$

For instance variables $x = y_i$ similarly:

$$\begin{aligned}
[\![x[\vec{e}/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} &= [\![e_i]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} \\
&= \sigma_{inst}[x \mapsto [\![e_i]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}](x) \\
&= [\![x]\!]_{\mathcal{L}}^{\omega,\sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}],\tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}]} \ .
\end{aligned}$$

The remaining cases are straightforward. □

Proof (of Lemma 2). Let $\sigma' = \sigma[[\![z]\!]_{\mathcal{G}}^{\omega,\sigma}.\vec{x} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]$. We proceed by induction on the structure of global expressions and assertions. The base cases are straightforward:

$$\begin{aligned}
[\![\mathsf{nil}[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= \mathsf{nil} = [\![\mathsf{nil}]\!]_{\mathcal{G}}^{\omega,\sigma'} \\
[\![z'[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![z']\!]_{\mathcal{G}}^{\omega,\sigma} = [\![z']\!]_{\mathcal{G}}^{\omega,\sigma'} \ .
\end{aligned}$$

Furthermore, we get the following induction cases. We start with the crucial one for qualified reference to instance variables.

$$[\![(E'.x_i)[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![\mathsf{if}\ E'[\vec{E}/z.\vec{x}] = z\ \mathsf{then}\ E_i\ \mathsf{else}\ (E'[\vec{E}/z.\vec{x}]).x_i\ \mathsf{fi}]\!]_{\mathcal{G}}^{\omega,\sigma} \ .$$

This conditional assertion evaluates to $[\![E_i]\!]_{\mathcal{G}}^{\omega,\sigma}$ if $[\![E'[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![z]\!]_{\mathcal{G}}^{\omega,\sigma}$ and to $[\![(E'[\vec{E}/z.\vec{x}]).x_i]\!]_{\mathcal{G}}^{\omega,\sigma}$ otherwise. So in the first case we get

$$\begin{aligned}
[\![(E'.x_i)[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![E_i]\!]_{\mathcal{G}}^{\omega,\sigma} \\
&= \sigma'([\![z]\!]_{\mathcal{G}}^{\omega,\sigma})(x_i) && \text{by def. of } \sigma' \\
&= \sigma'([\![E'[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma})(x_i) && \text{by the case assumption} \\
&= \sigma'([\![E']\!]_{\mathcal{G}}^{\omega,\sigma'})(x_i) && \text{by induction} \\
&= [\![E'.x_i]\!]_{\mathcal{G}}^{\omega,\sigma'} && \text{by def. of } [\![_]\!]_{\mathcal{G}} \ .
\end{aligned}$$

If otherwise $[\![E'[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} \neq [\![z]\!]_{\mathcal{G}}^{\omega,\sigma}$, then

$$\begin{aligned}
[\![(E'.x_i)[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![(E'[\vec{E}/z.\vec{x}]).x_i]\!]_{\mathcal{G}}^{\omega,\sigma} \\
&= \sigma([\![E'[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma})(x_i) && \text{by def. of } [\![_]\!]_{\mathcal{G}} \\
&= \sigma'([\![E'[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma})(x_i) && \text{case assumption and def. of } \sigma' \\
&= \sigma'([\![E']\!]_{\mathcal{G}}^{\omega,\sigma'})(x_i) && \text{by induction} \\
&= [\![E'.x_i]\!]_{\mathcal{G}}^{\omega,\sigma'} && \text{by def. of } [\![_]\!]_{\mathcal{G}} \ .
\end{aligned}$$

For operator expressions we get:

$$\begin{array}{lll}
& [\![(\mathsf{f}(E_1,\ldots,E_n))[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} & \\
= & [\![\mathsf{f}(E_1[\vec{E}/z.\vec{x}],\ldots,E_n[\vec{E}/z.\vec{x}])]\!]_{\mathcal{G}}^{\omega,\sigma} & \text{def. of substitution} \\
= & f([\![E_1[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma},\ldots,[\![E_n[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma}) & \text{def. of } [\![_]\!]_{\mathcal{G}} \\
= & f([\![E_1]\!]_{\mathcal{G}}^{\omega,\sigma'},\ldots,[\![E_n]\!]_{\mathcal{G}}^{\omega,\sigma'}) & \text{by induction} \\
= & [\![\mathsf{f}(E_1,\ldots,E_n)]\!]_{\mathcal{G}}^{\omega,\sigma'} & \text{def. of } [\![_]\!]_{\mathcal{G}}\ .
\end{array}$$

For global assertions, the cases of negation and conjunction are straightforward. For quantification,

$$\begin{array}{lll}
& [\![(\exists z'(P))[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega,\sigma} = \mathit{true} & \\
\iff & [\![\exists z'(P[\vec{E}/z.\vec{x}])]\!]_{\mathcal{G}}^{\omega,\sigma} = \mathit{true} & \text{def. of substitution} \\
\iff & [\![P[\vec{E}/z.\vec{x}]]\!]_{\mathcal{G}}^{\omega[z'\mapsto v],\sigma} = \mathit{true} \text{ for some } v \in \mathit{dom}_{\mathit{nil}}(\sigma) & \text{def. of } [\![_]\!]_{\mathcal{G}} \\
\iff & [\![P]\!]_{\mathcal{G}}^{\omega[z'\mapsto v],\sigma'} = \mathit{true} \text{ for some } v \in \mathit{dom}_{\mathit{nil}}(\sigma) & \text{by induction} \\
\iff & [\![\exists z'(P)]\!]_{\mathcal{G}}^{\omega,\sigma'} = \mathit{true} & \mathit{dom}(\sigma) = \mathit{dom}(\sigma')\ .
\end{array}$$

□

Proof (of Lemma 3). By induction on the structure of local expressions and assertions. The base cases for local expressions are listed below, where the ones for instance and local variables are covered by the respective provisos of the lemma. Note that $\vec{u}$ is the vector of *all* local variables of the expression.

$$\begin{array}{rl}
[\![x[z,\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = [\![z.x]\!]_{\mathcal{G}}^{\omega,\sigma} = \sigma([\![z]\!]_{\mathcal{G}}^{\omega,\sigma})(x) = \sigma(\omega(z))(x) = \sigma(\tau(\mathsf{this}))(x) \\
& = [\![x]\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau} \\
[\![u_i[z,\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = [\![E_i]\!]_{\mathcal{G}}^{\omega,\sigma} = \tau(u_i) = [\![u_i]\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau} \\
[\![\mathsf{this}[z,\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = [\![z]\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(z) = \tau(\mathsf{this}) = [\![\mathsf{this}]\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau} \\
[\![\mathsf{nil}[z,\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = \mathsf{nil} = [\![\mathsf{nil}]\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau} \\
[\![z'[z,\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = [\![z']\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(z') = [\![z']\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau}\ .
\end{array}$$

Compound expressions are treated by straightforward induction:

$$\begin{array}{lll}
& [\![\mathsf{f}(e_1,\ldots,e_n)[z,\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} & \\
= & f([\![e_1[z,\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma},\ldots,[\![e_n[z,\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma}) & \text{semantics of assertions} \\
= & f([\![e_1]\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau},\ldots,[\![e_n]\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau}) & \text{by induction} \\
= & [\![\mathsf{f}(e_1,\ldots,e_n)]\!]_{\mathcal{L}}^{\omega,\sigma(\tau(\mathsf{this})),\tau} & \text{semantics of assertions}\ .
\end{array}$$

For local assertions, negation and conjunction are straightforward. Unrestricted quantification $\exists z'(p)$ in the local assertion language is only allowed for variables of type $t \in \{\mathsf{Int}, \mathsf{Bool}\}$, for which $dom^t_{nil}(\sigma) = Val^t$. We get

$$\begin{array}{lll}
& [\![(\exists z'(p))[z, \vec{E}/\mathsf{this}, \vec{u}]]\!]^{\omega,\sigma}_{\mathcal{G}} = true & \\
\iff & [\![\exists z'(p[z, \vec{E}/\mathsf{this}, \vec{u}])]\!]^{\omega,\sigma}_{\mathcal{G}} = true & \text{def. of substitution} \\
\iff & [\![p[z, \vec{E}/\mathsf{this}, \vec{u}]]\!]^{\omega[z' \mapsto v],\sigma}_{\mathcal{G}} = true \text{ for some } v \in Val^t & \text{assertion semantics} \\
\iff & [\![p]\!]^{\omega[z' \mapsto v],\sigma(\tau(\mathsf{this})),\tau}_{\mathcal{L}} = true \text{ for some } v \in Val^t & \text{by induction} \\
\iff & [\![\exists z'(p)]\!]^{\omega,\sigma(\tau(\mathsf{this})),\tau}_{\mathcal{L}} = true & \text{assertion semantics.}
\end{array}$$

For restricted quantification over elements of a sequence let $z' \in LVar^t$. Then

$$\begin{array}{lll}
& [\![(\exists z' \in e(p))[z, \vec{E}/\mathsf{this}, \vec{u}]]\!]^{\omega,\sigma}_{\mathcal{G}} = true & \\
\iff & [\![\exists z'((z' \in e[z, \vec{E}/\mathsf{this}, \vec{u}]) \wedge (p[z, \vec{E}/\mathsf{this}, \vec{u}]))]\!]^{\omega,\sigma}_{\mathcal{G}} = true & \text{by definition} \\
\iff & [\![(z' \in e[z, \vec{E}/\mathsf{this}, \vec{u}]) \wedge p[z, \vec{E}/\mathsf{this}, \vec{u}]]\!]^{\omega',\sigma}_{\mathcal{G}} = true & \text{semantics} \\
& \text{for some } v \in dom^t_{nil}(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] & \\
\iff & \left(([\![z']\!]^{\omega',\sigma}_{\mathcal{G}} \in [\![e[z, \vec{E}/\mathsf{this}, \vec{u}]]\!]^{\omega',\sigma}_{\mathcal{G}}) \wedge [\![p[z, \vec{E}/\mathsf{this}, \vec{u}]]\!]^{\omega',\sigma}_{\mathcal{G}}\right) = true & \text{semantics} \\
& \text{for some } v \in dom^t_{nil}(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] & \\
\iff & \left(([\![z']\!]^{\omega',\sigma(\tau(\mathsf{this})),\tau}_{\mathcal{L}} \in [\![e]\!]^{\omega',\sigma(\tau(\mathsf{this})),\tau}_{\mathcal{L}}) \wedge [\![p]\!]^{\omega',\sigma(\tau(\mathsf{this})),\tau}_{\mathcal{L}}\right) = true & \text{by induction} \\
& \text{for some } v \in dom^t_{nil}(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] & \\
\iff & [\![(z' \in e) \wedge p]\!]^{\omega',\sigma(\tau(\mathsf{this})),\tau}_{\mathcal{L}} = true & \text{semantics} \\
& \text{for some } v \in dom^t_{nil}(\sigma) \text{ and } \omega' = \omega[z' \mapsto v] & \\
\iff & [\![\exists z' \in e(p)]\!]^{\omega,\sigma(\tau(\mathsf{this})),\tau}_{\mathcal{L}} = true & \text{semantics .}
\end{array}$$

The last equation uses the assumption that the local state τ and the instance state $\sigma(\tau(\mathsf{this}))$ assign values from $dom_{nil}(\sigma)$ to all variables, i.e., e does not refer to values of non-existing objects. Consequently, $v \in Val^t_{nil}$ together with $[\![z' \in e]\!]^{\omega[z' \mapsto v],\sigma(\tau(\mathsf{this})),\tau}_{\mathcal{L}} = true$ implies $v \in dom^t_{nil}(\sigma)$.

The case for restricted quantification is analogous. □

B.2 Soundness

This section contains the inductive proof of soundness of the proof method. We start with some ancillary lemmas about basic invariant properties of transformed and annotated programs, for instance properties of the auxiliary variables added in the transformation. Afterwards, we show soundness of the verification conditions of Section 4.2, which then straightforwardly lead to the soundness of the proof-system.

B.2.1 Invariant properties

Proof (of transformation Lemma 9). Both directions by straightforward induction on the length of reduction. The crucial point in the "if"-direction is that for all reachable global configurations $\langle T', \sigma' \rangle$ of a transformed program *prog'* there is also a reachable *stable* configuration $\langle T'', \sigma'' \rangle$ of *prog'* representing the same configuration of the original program, i.e., such that $\langle T'' \downarrow prog, \sigma'' \downarrow prog\rangle = \langle T' \downarrow prog, \sigma' \downarrow prog\rangle$. The stable configuration $\langle T'', \sigma''\rangle$ is the next stable configuration after completing the assignments in the bracketed sections accompanying object creation and communication statements in $\langle T', \sigma'\rangle$. Note that a local configuration is enabled in a reachable stable configuration $\langle T'', \sigma''\rangle$ of a transformed program iff the corresponding local configuration is enabled in the projection $\langle T'' \downarrow prog, \sigma'' \downarrow prog\rangle$. □

Lemma 12. *Let σ be a global state and ω a logical environment referring only to values existing in σ. Then $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in dom_{nil}(\sigma)$ for all global expressions $E \in GExp$ that can be evaluated in the context of ω and σ.*

Proof (of Lemma 12). By structural induction on the global assertion. The case for logical variables $z \in LVar^t$ is immediate by the assumption about ω, the ones for nil and operator expressions are trivial, respectively follows by induction. For qualified references $E.x$ with $E \in GExp^c$ and $x \in IVar^t$ an instance variable of class c, if $E.x$ can be evaluated in the context of ω and σ, then $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \neq nil$. Hence by induction $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in dom_{nil}(\sigma)$, more specifically $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in dom(\sigma)$. Therefore by definition of global states $\sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \in dom_{nil}(\sigma)$. □

Proof (of Lemma 8). By structural induction on the global assertion P. Let $\omega' = \omega[z' \mapsto v]$. For logical variables z in P we know $z \neq z'$ and thus $[\![z]\!]_{\mathcal{G}}^{\omega,\sigma} = \omega(z) = \omega'(z) = [\![z]\!]_{\mathcal{G}}^{\omega',\sigma'}$. For qualified references to instance variables, the argument is as follows:

$$
\begin{array}{lll}
[\![E.x]\!]_{\mathcal{G}}^{\omega,\sigma} & = \sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) & \\
 & = \sigma'([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) & [\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \neq \alpha \text{ by Lemma 12 and } \alpha \notin dom(\sigma) \\
 & = \sigma'([\![E \downarrow z']\!]_{\mathcal{G}}^{\omega',\sigma'})(x) & \text{by induction} \\
 & = \sigma'([\![E]\!]_{\mathcal{G}}^{\omega',\sigma'})(x) & \text{by the definition of } \downarrow z' \\
 & = [\![E.x]\!]_{\mathcal{G}}^{\omega',\sigma'} & \text{semantics of global expressions .}
\end{array}
$$

The interesting case is the one for quantification. For $z \in LVar^t$:

$$
\begin{array}{llll}
 & \omega, \sigma \models_{\mathcal{G}} \exists z(P) & & \\
\iff & \omega[z \mapsto v], \sigma \models_{\mathcal{G}} P \text{ for some } v \in dom^t_{nil}(\sigma) & & \text{semantics} \\
\iff & \omega[z \mapsto v][z' \mapsto v], \sigma' \models_{\mathcal{G}} P \downarrow z' \text{ for some } v \in dom^t_{nil}(\sigma) & & \text{induction} \\
\iff & \omega[z \mapsto v][z' \mapsto v], \sigma' \models_{\mathcal{G}} \mathsf{within}(z, z') \wedge P \downarrow z' & & dom^t_{nil}(\sigma) \subseteq v \\
 & \qquad \text{for some } v \in dom^t_{nil}(\sigma) & & \\
\iff & \omega[z' \mapsto v], \sigma' \models_{\mathcal{G}} \exists z(\mathsf{within}(z, z') \wedge P \downarrow z') & & \text{semantics} \\
\iff & \omega[z' \mapsto v], \sigma' \models_{\mathcal{G}} (\exists z(P)) \downarrow z'. & &
\end{array}
$$

The remaining cases are straightforward. □

Proof (of Lemma 4). All parts by straightforward induction on the steps of the transformed program. □

Lemma 13 (Synchronization). *Let $\langle T, \sigma\rangle$ be a reachable stable configuration of $prog'$. Then for each class c of $prog'$ and each object $\alpha \in dom^c(\sigma)$,*

1. $\sigma(\alpha)(\mathsf{lock}) = (nil, 0)$ *iff there exists no* $(\tau, stm) \in T$ *with* $\tau(\mathsf{this}) = \alpha$ *and* stm *synchronized, and*
2. $\sigma(\alpha)(\mathsf{lock}) \neq (nil, 0)$ *iff there exists a* $(\tau, stm) \in T$ *with* $\tau(\mathsf{this}) = \alpha$*,* $\tau(\mathsf{id}) = \sigma(\alpha)(\mathsf{lock})$*, and* stm *synchronized, and for all* (τ', stm') *with* $\tau'(\mathsf{this}) = \alpha$ *and* stm' *synchronized,* $\tau'(\mathsf{id}) \geq \sigma(\alpha)(\mathsf{lock})$.

Proof (of Lemma 13). By induction on the length of $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T_n, \sigma_n\rangle$.

In the base case of an initial configuration $\langle T_0, \sigma_0\rangle$ (cf. page 11), the set T_0 contains exactly one thread (τ, stm), executing the non-synchronized main-statement of the program, and initially the lock of the only object $\tau(\mathsf{this}) = \alpha$ is set to $(nil, 0)$. The first stable configuration results from $\langle T_0, \sigma_0\rangle$ by executing the bracketed section at the beginning of the non-synchronized main-method. Since the assignment in this bracketed section does not change the values of the variables lock, id, and this of any objects or threads, does not create new objects, and does not add or remove any local configuration from T, the property holds for the first reachable stable configuration.

For the inductive step, assume $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle \longrightarrow^* \langle T', \sigma'\rangle$ such that $\langle T', \sigma'\rangle$ is stable and $\langle T, \sigma\rangle$ is the last stable configuration preceding $\langle T', \sigma'\rangle$ in the computation. We distinguish whether $\langle T', \sigma'\rangle$ results from $\langle T, \sigma\rangle$ by executing assignment, object creation, method invocation, or return.

Case: Ass
Let $\langle T, \sigma\rangle \longrightarrow \langle T', \sigma'\rangle$ result from the execution of an assignment $\vec{y} := \vec{e}$ outside bracketed sections, where both $\langle T, \sigma\rangle$ and $\langle T', \sigma'\rangle$ are stable (rule Ass). The assignment does not touch the variables lock, id, and this of any objects or threads, does not create new objects, and does not push or pop local configurations, and the property follows directly by induction.

Case: New
In this case $\langle T, \sigma\rangle \longrightarrow^2 \langle T', \sigma'\rangle$, where the first step created a new object (rule New), and the second one is the trailing multiple assignment in the bracketed section of new.

Let $\alpha \in dom(\sigma')$. Then α either reference the newly created object, or $\alpha \in dom(\sigma)$. In the first case $\alpha \notin dom(\sigma)$, and by the definition of global configurations (cf. page 8) there is no local configuration $(\tau, stm) \in T$ with $\tau(\mathsf{this}) = \alpha$. Since the last two steps do not add any local configurations to T, $\tau(\mathsf{this}) \neq \alpha$ for all $(\tau, stm) \in T'$. Furthermore, since the lock of the new object is initialized to $(nil, 0)$, the required property holds for the new object. In the second case, if $\alpha \in dom(\sigma)$, the property follows directly by induction.

Case: Start_{skip}
Also in this case where the two steps of $\langle T, \sigma\rangle \longrightarrow^2 \langle T', \sigma'\rangle$ are justified by Start_{skip} and Ass, no local configurations are added to or removed from T, no new objects are created, and the values of lock, id, and this are unchanged, and the case follows by induction.

Case: Call, Start
We are given $\langle T, \sigma\rangle \longrightarrow^3 \langle T', \sigma'\rangle$, where the first step is justified by rule Call

or START, and the following two steps are the the assignments in the bracketed sections of caller and callee.

Let $\alpha \in dom(\sigma')$. Then also $\alpha \in dom(\sigma)$, since no new objects are created in the last three steps. If α is not the callee object, then the property holds directly by induction. If α is the callee object, the only new local configuration (τ, stm) in T' with $\tau(\mathsf{this}) = \alpha$ represents the execution of the invoked method.

If the invoked method is non-synchronized, then no locks are touched, and since no local configurations with synchronized statements are added to or removed from the stack, the property follows by induction.

In the case of a synchronized method, the invoked method is not a start-method, as they are non-synchronized by definition. If in the state prior to the method invocation $\sigma(\alpha)(\mathsf{lock}) = (nil, 0)$, then by induction (τ, stm) is the only local configuration in T' representing the execution of a synchronized method of α. Furthermore, in the bracketed section of the callee the assignment $\mathsf{lock} := \mathsf{getlock}(\mathsf{lock}, \mathsf{id})$ is executed, implying $\sigma'(\alpha)(\mathsf{lock}) = \tau(\mathsf{id})$, and thus the required property. Otherwise, if $\sigma(\alpha)(\mathsf{lock}) \neq (nil, 0)$, then by induction there exists $(\tau', stm') \in T$ such that $\tau'(\mathsf{this}) = \alpha$, stm' synchronized, and $\tau'(\mathsf{id}) = \sigma(\alpha)(\mathsf{lock})$, and for all $(\tau'', stm'') \in T$ representing the execution of a synchronized method in α we have $\tau'(\mathsf{id}) \leq \tau''(\mathsf{id})$. Since the callee configuration is on top of its stack, the antecedent *isfree* of rule CALL together with Lemma 4 implies also $\tau'(\mathsf{id}) \leq \tau(\mathsf{id})$. The assignment $\mathsf{lock} := \mathsf{getlock}(\mathsf{lock}, \mathsf{id})$ of the callee does not change the lock value, i.e., $\sigma(\alpha)(\mathsf{lock}) = \sigma'(\alpha)(\mathsf{lock})$. As no local configurations are removed from the stack in the last three steps, the property is satisfied.

Case: RETURN
We are given $\langle T, \sigma \rangle \longrightarrow^3 \langle T', \sigma' \rangle$, consisting of a return step by rule RETURN and the two trailing assignments in the bracketed sections of the callee and the caller.

The assumption $\alpha \in dom(\sigma')$ implies $\alpha \in dom(\sigma)$, since no new objects are created in the last three steps. If α is not the callee object, or if the invoked method is non-synchronized, then the property holds directly by induction. If otherwise α is the callee object and the invoked method is synchronized, the bracketed section of the callee contains the assignment $\mathsf{lock} := \mathsf{release}(\mathsf{lock}, \mathsf{id})$. We further distinguish two cases: If the identity $\tau(\mathsf{id})$ of the callee is greater than $\sigma(\alpha)(\mathsf{lock})$, then the lock of the callee remains unchanged, and the property follows directly by induction. Otherwise, if $\sigma(\alpha)(\mathsf{lock})$ equals the identity $\tau(\mathsf{id})$ of the callee, then in the bracketed section of the return statement the lock is set by $\mathsf{lock} := \mathsf{release}(\mathsf{lock}, \mathsf{id})$ to $(nil, 0)$, and the local configuration of the callee is removed from the stack. By induction, all local configurations (τ', stm') in T with synchronized statements stm' and representing execution in α, i.e., $\tau'(\mathsf{this}) = \alpha$, have an identity $\tau'(\mathsf{id}) \geq \sigma(\alpha)(\mathsf{lock})$, i.e., $\tau'(\mathsf{id}) \geq \tau(\mathsf{id})$. On the other hand, the callee configuration in T is by rule RETURN on the top of its stack, and consequently by Lemma 4 $\tau'(\mathsf{id}) \leq \tau(\mathsf{id})$. It follows that $\tau'(\mathsf{id}) = \tau(\mathsf{id})$, and by Lemma 4 the callee configuration (τ, stm) is the only configuration in T with synchronized statement and $\tau(\mathsf{this}) = \alpha$. Thus after removing the callee from the stack, there is no local configuration in T' representing the execution of any synchronized methods of α, and the property holds.

Case: TERMINATE
For termination, we are given $\langle T, \sigma\rangle \longrightarrow^2 \langle T', \sigma'\rangle$, where the first step is the execution of the return-statement of a start-method or of the initial invocation of the main-method (rule TERMINATE), and the second step executed the assignment in the bracketed section of the return-statement. Since main- and start-methods are non-synchronized by definition, no lock-values are changed during these steps, no local configurations with synchronized statements are pushed or popped, and no new objects are created, and the property holds by induction. □

Proof (of Lemma 5). For the "only-if"-direction, $\mathit{isfree}(T\backslash\{\xi\}, \alpha)$ implies by definition (cf. page 10) that there is no $(\tau', \mathit{stm}') \in T\backslash\{\xi\}$ with $\tau'(\mathsf{this}) = \alpha$ and stm' synchronized. If neither in ξ there exist such a configuration, then by Lemma 13 $\sigma(\alpha)(\mathsf{lock}) = (\mathit{nil}, 0)$. Otherwise, $\sigma(\alpha)(\mathsf{lock}) \neq (\mathit{nil}, 0)$ by Lemma 13, and there exists $(\tau', \mathit{stm}') \in T$ with $\tau'(\mathsf{this}) = \alpha$ and stm' synchronized, such that $\sigma(\alpha)(\mathsf{lock}) = \tau'(\mathsf{id})$. The assumption $\mathit{isfree}(T\backslash\{\xi\}, \alpha)$ implies that $(\tau', \mathit{stm}') \in \xi$. Furthermore, since (τ, stm) is on top of the stack ξ, Lemma 4 implies that $\tau'(\mathsf{id}) \leq \tau(\mathsf{id})$, i.e., $\sigma(\alpha)(\mathsf{lock}) \leq \tau(\mathsf{id})$.

For the reverse direction, we are given $\sigma(\alpha)(\mathsf{lock}) = (\mathit{nil}, 0) \vee \sigma(\alpha)(\mathsf{lock}) \leq \tau(\mathsf{id})$. If $\sigma(\alpha)(\mathsf{lock}) = (\mathit{nil}, 0)$, then $\mathit{isfree}(T\backslash\{\xi\}, \alpha)$ directly by Lemma 13. If $\sigma(\alpha)(\mathsf{lock}) \leq \tau(\mathsf{id})$, then by Lemma 13 there exists a $(\tau', \mathit{stm}') \in T$ with $\tau'(\mathsf{this}) = \alpha$ and stm' synchronized, such that $\sigma(\alpha)(\mathsf{lock}) = \tau'(\mathsf{id})$, and for all (τ'', stm'') with $\tau''(\mathsf{this}) = \alpha$ and stm'' synchronized, $\sigma(\alpha)(\mathsf{lock}) \leq \tau''(\mathsf{id})$. Lemma 4 implies that all such local configurations representing the execution of a synchronized method in α are in the same stack as (τ, stm), i.e., in ξ, hence $\mathit{isfree}(T\backslash\{\xi\}, \alpha)$. □

Proof (of Lemma 6). Straightforward by the definition of augmentation. □

Lemma 14 (Stable). *For all reachable configurations $\langle T, \sigma\rangle$ of a program prog' and for all objects $\alpha \in \mathit{dom}(\sigma)$, $\sigma(\alpha)(\mathsf{stable}) = \mathit{false}$ iff the last two steps in the computation leading to $\langle T, \sigma\rangle$ were a self-communication (call or return) within α and the execution of the auxiliary assignment of the sender but not yet that of the receiver.*

Proof (of Lemma 14). By straightforward induction, using the definition of augmentation. □

Proof (of Lemma 7). Let $\langle T, \sigma\rangle$ be a reachable configuration of a transformed program and $(\tau, \mathit{stm}_{ass}; \mathit{stm})$ a local configuration in T where stm_{ass} equals $\vec{y} := \vec{e}$ or $\langle \vec{y} := \vec{e}\rangle$. Let furthermore $\sigma_{inst} = \sigma(\tau(\mathsf{this}))$.

Case: Part 1
Let $(\tau, \mathit{stm}_{ass}; \mathit{stm})$ be enabled in $\langle T, \sigma\rangle$. If $\vec{y} := \vec{e}$ is the observation of either object creation, or the sender part in a communication, or the receiver part in a non-self-communication, then by definition $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$. If

$\vec{y} := \vec{e}$ is the observation of the receiver part in a self-communication, then, since the assignment is enabled, the sender has already executed its observation, and Lemma 14 assures that $\sigma_{inst}(\mathsf{stable}) = \mathit{false}$, and hence by the clause for self-calls $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$. If finally $\vec{y} := \vec{e}$ does not occur within a bracketed section, $\langle T, \sigma \rangle$ is stable since the assignment is enabled. By Lemma 14 $\sigma_{inst}(\mathsf{stable}) = \mathit{true}$ and thus also $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$.

Case: Part 2
Let $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$. If $\vec{y} := \vec{e}$ is the observation of object creation or of the sender part of a communication, then it is enabled in $\langle T, \sigma \rangle$.

If $\vec{y} := \vec{e}$ is the observation of the receiver part of a self-communication within $\tau(\mathsf{this})$, then Lemma 4 together with $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$ and the definition of $\mathsf{enabled}$ imply $\sigma_{inst}(\mathsf{stable}) = \mathit{false}$. Using Lemma 14 we get that the sender already executed its observation, i.e., $\vec{y} := \vec{e}$ is enabled in $\langle T, \sigma \rangle$.

If $\vec{y} := \vec{e}$ is the observation of the receiver part of a non-self-communication, then either the sender has already executed its observation, or not. In the first case $\vec{y} := \vec{e}$ is enabled in $\langle T, \sigma \rangle$. In the second case, executing the observation of the caller does not change the instance state of the callee object $\tau(\mathsf{this})$. Thus the resulting global configuration satisfies the requirements.

Finally, assume that $\vec{y} := \vec{e}$ occurs outside bracketed sections and let $\langle T', \sigma' \rangle$ be the last stable configuration in the computation leading to $\langle T, \sigma \rangle$. If σ' and σ define the same instance state for $\tau(\mathsf{this})$, then $\langle T', \sigma' \rangle$ satisfies the requirements. Otherwise, $\langle T', \sigma' \rangle \longrightarrow^* \langle T, \sigma \rangle$ executes some communication and the observation of the sender in the object $\tau(\mathsf{this})$. Furthermore, $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$ imply $\sigma_{inst}(\mathsf{stable}) = \mathit{true}$, and Lemma 14 assures that the receiver object is different from $\tau(\mathsf{this})$. This means, executing the observation of the receiver does not change the instance state of $\tau(\mathsf{this})$, and the resulting configuration satisfies the requirements. $\square$

Lemma 15 (Interleavable). *Let $(\tau_1, stm_{ass}; stm_1)$ be enabled in a reachable configuration $\langle T, \sigma \rangle$ of a proof outline, where stm_{ass} is $\vec{y} := \vec{e}$ or $\langle \vec{y} := \vec{e} \rangle$. Let furthermore $(\tau_1, stm_{ass}; stm_1) \neq (\tau_2, stm; stm_2) \in T$ with $\tau_1(\mathsf{this}) = \tau_2(\mathsf{this}) = \alpha$. Then*

$$\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$$

for arbitrary $\omega \in \Omega$ and $\tau = \tau_1[\vec{u}', \mathsf{this}' \mapsto \tau_2(\vec{u}), \tau_2(\mathsf{this})]$, where $\vec{u}$ are the local variables from the domain of τ_2 and $\vec{u}'$ fresh variables of corresponding types.

Proof (of Lemma 15). We distinguish whether or not the local configurations $(\tau_1, stm_{ass}; stm_1)$ and $(\tau_2, stm; stm_2)$ occur in the same stack.

Case: $(\tau_1, stm_{ass}; stm_1)$ and $(\tau_2, stm; stm_2)$ reside in the same stack
Then Lemma 4 yields $\mathit{samethread}(\tau_2(\mathsf{id}), \tau_1(\mathsf{id}))$. Furthermore, if stm represents an *interleaving* point, we have $\tau_2(\mathsf{id}) < \tau_1(\mathsf{id})$, and stm begins with a receive statement. If additionally $stm_{ass} = \langle \vec{y} := \vec{e} \rangle$ is the trailing observation of a return statement, then $(\tau_2, stm; stm_2)$ cannot be the matching callee, since, as said, stm begins with a receive statement. I.e., we have the stronger condition

$callee(\tau_2(\mathsf{id})) < \tau_1(\mathsf{id})$. Hence by definition of wait_for_ret (cf. page 25) and by the definition of τ

$$\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} \mathsf{wait\_for\_ret}(pre(stm), \vec{y} := \vec{e}) \ .$$

If otherwise *stm* represents a *non-interleaving* point, then, according to the semantics of communication for transformed programs, either the last computation step leading to $\langle T, \sigma \rangle$ was a self-call within α or the communication of a return value within α, such that $(\tau_1, \langle \vec{y} := \vec{e} \rangle; stm_1)$ is the resulting configuration of the sender and $(\tau_2, stm; stm_2)$ that of the receiver. For the self-call we have $\tau_2(\mathsf{id}) = callee(\tau_1(\mathsf{id}))$ and in the case of return conversely $\tau_1(\mathsf{id}) = callee(\tau_2(\mathsf{id}))$. Note that for the invocation of a start-method, $(\tau_1, stm_{ass}; stm_1)$ and $(\tau_2, stm; stm_2)$ would not belong to the same stack; this kind of communication is handled in the proof case below. Thus, by definition of self_call and self_ret (cf. page 25) we get

$$\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} \mathsf{self\_call}(pre(stm), \vec{y} := \vec{e}) \vee \mathsf{self\_ret}(pre(stm), \vec{y} := \vec{e}) \ .$$

Case: $(\tau_1, stm_{ass}; stm_1)$ and $(\tau_2, stm; stm_2)$ are in different stacks
According to the semantics of synchronization, since the local configurations $(\tau_1, stm_{ass}; stm_1)$ and $(\tau_2, stm; stm_2)$ belong to different threads executing in the same object, not both statements are synchronized. Furthermore by Lemma 4 $\neg samethread(\tau_1(\mathsf{id}), \tau_2(\mathsf{id}))$. If additionally *stm* represents an interleaving point, then

$$\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} \mathsf{diff\_threads}(pre(stm), \vec{y} := \vec{e}) \ .$$

If otherwise *stm* represents a non-interleaving point, then the last computation step leading to $\langle T, \sigma \rangle$ was the self-invocation of the start-method of α, where $(\tau_2, stm; stm_2)$ represents the initial stack of the new thread, and $(\tau_1, \langle \vec{y} := \vec{e} \rangle; stm_1)$ is the caller. By the definition of the augmentation $\tau_2(\mathsf{id}) = (\alpha, 0)$, and therefore

$$\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} \mathsf{self\_start}(pre(stm), \vec{y} := \vec{e}) \ .$$

Hence we have

$$\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e}) \ .$$

□

B.2.2 Soundness of the proof-conditions This section shows one by one the soundness of the verification conditions of Section 4.2, from Equation (1) to (10).

Lemma 16 (Initial correctness). *Let the proof outline prog′ be initially correct. Let* $\langle T_0, \sigma_0 \rangle$ *the initial configuration of prog′ with* $T_0 = \{(\tau, body_{\mathsf{main}})\}$, *and* $\langle T_0, \sigma_0 \rangle \longrightarrow \langle T_0', \sigma_0' \rangle$. *Then* $\omega, \sigma_0(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} pre(body_{\mathsf{main}})$ *and* $\omega, \sigma_0' \models_{\mathcal{G}} GI$, *for all logical environments* ω *referring only to values existing in* σ_0.

Proof (of Lemma 16). Let α be the initial object. Then by definition $\tau = \tau_{init}[\mathsf{this}\mapsto\alpha][\mathsf{id}\mapsto(\alpha,0)]$, $dom(\sigma_0) = \{\alpha\}$ and $\sigma_0(\alpha) = \sigma_{inst}^{init}$. Furthermore, the first stable configuration $\langle T_0', \sigma_0'\rangle$ results from $\langle T_0, \sigma_0\rangle$ by executing the multiple assignment $\vec{y}_2 := \vec{e}_2$ in the bracketed section at the beginning of the main-method, i.e., $\sigma_0' = \sigma_0[\alpha.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_0(\alpha),\tau}]$.

Condition (1) of the initial correctness on page 24 implies

$$\omega, \sigma_{inst}^{init}, \tau \models_{\mathcal{L}} pre(body_{\mathsf{main}})[(\mathsf{this},0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}],$$

and with Lemma 1, $\omega, \sigma_{inst}^{init}, \tau \models_{\mathcal{L}} pre(body_{\mathsf{main}})$, which means, $\omega, \sigma_0(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} pre(body_{\mathsf{main}})$.

For the global invariant we argue as follows. As in σ_0 there exists exactly one object α being in its initial instance state, we have

$$\omega[z\mapsto\alpha], \sigma_0 \models_{\mathcal{G}} \mathsf{InitState}(z) \wedge \forall z'(z'{=}\mathsf{nil} \vee z{=}z')\ ,$$

where z is of the type of the main class, and z' is a logical variable of type Object. Using condition (2) of the initial correctness we get

$$\omega[z\mapsto\alpha], \sigma_0 \models_{\mathcal{G}} GI[\vec{E}_2/z.\vec{y}_2]\ ,$$

where $\vec{E}_2 = \vec{e}_2[(\mathsf{this},0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}][z/\mathsf{this}]$. Applying Lemma 2 we get

$$\omega[z\mapsto\alpha], \sigma_0[\alpha.\vec{y}_2 \mapsto [\![\vec{E}_2]\!]_{\mathcal{G}}^{\omega[z\mapsto\alpha],\sigma_0}] \models_{\mathcal{G}} GI\ .$$

Using Lemma 3 on page 17 and Lemma 1 gives

$$\begin{aligned}
[\![\vec{E}_2]\!]_{\mathcal{G}}^{\omega[z\mapsto\alpha],\sigma_0} &= [\![\vec{e}_2[(\mathsf{this},0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}][z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega[z\mapsto\alpha],\sigma_0} \\
&= [\![\vec{e}_2]\!]_{\mathcal{L}}^{\omega[z\mapsto\alpha],\sigma_0(\alpha),\tau} \\
&= [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_0(\alpha),\tau}\ ,
\end{aligned}$$

i.e., $\omega[z\mapsto\alpha], \sigma_0' \models_{\mathcal{G}} GI$. Again, the value of GI does not depend on the logical environment, and therefore $\omega, \sigma_0' \models_{\mathcal{G}} GI$. □

Lemma 17 (Local correctness). *Let $\langle T_n, \sigma_n\rangle$ be a reachable configuration of a locally correct proof outline, and let $\langle T_n, \sigma_n\rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1}\rangle$ result from the execution of a multiple assignment $\vec{y} := \vec{e}$ in a local configuration $(\tau, stm_{ass}; stm) \in T_n$, where stm_{ass} is $\vec{y} := \vec{e}$ or $\langle\vec{y} := \vec{e}\rangle$. Then $\omega, \sigma_n(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} pre(stm_{ass})$ implies $\omega, \sigma_{n+1}(\tau'(\mathsf{this})), \tau' \models_{\mathcal{L}} pre(stm)$ for arbitrary $\omega \in \Omega$, where $\tau' = \tau[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_n(\tau(\mathsf{this})),\tau}]$.*

Proof (of Lemma 17). As the assignment is enabled in $\langle T_n, \sigma_n\rangle$, so by Lemma 7 $\omega, \sigma_n(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$. The verification condition (3) for local correctness on page 24 gives $\omega, \sigma_n(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} post(\vec{y} := \vec{e})[\vec{e}/\vec{y}]$, and since $post(\vec{y} := \vec{e}) = pre(stm)$, we get by Lemma 1 $\omega, \sigma_{n+1}(\tau'(\mathsf{this})), \tau' \models_{\mathcal{L}} pre(stm)$. □

Lemma 18 (Interference freedom test). *Let $\langle T_n, \sigma_n\rangle$ be a reachable configuration of an interference free proof outline, and let $\langle T_n, \sigma_n\rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1}\rangle$ result from the execution of a multiple assignment $\vec{y} := \vec{e}$ in a local configuration $(\tau_1, stm_{ass}; stm_1) \in T_n$, where stm_{ass} is $\vec{y} := \vec{e}$ or $\langle \vec{y} := \vec{e}\rangle$. Let furthermore $(\tau_2, stm; stm_2) \in T_n \cap T_{n+1}$.*

Then $\omega, \sigma_n(\tau_1(\mathsf{this})), \tau_1 \models_{\mathcal{L}} pre(stm_{ass})$ and $\omega, \sigma_n(\tau_2(\mathsf{this})), \tau_2 \models_{\mathcal{L}} pre(stm)$ imply $\omega, \sigma_{n+1}(\tau_2(\mathsf{this})), \tau_2 \models_{\mathcal{L}} pre(stm)$ for arbitrary $\omega \in \Omega$.

Proof (of Lemma 18). We are given

$$\langle T \,\dot\cup\, \{\xi \circ (\tau_1, stm_{ass}; stm_1) \circ \xi'\}, \sigma_n\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\tau_1', stm_1) \circ \xi'\}, \sigma_{n+1}\rangle \text{ , or}$$
$$\langle T \,\dot\cup\, \{\xi \circ (\tau_1, stm_{ass})\}, \sigma_n\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi\}, \sigma_{n+1}\rangle \text{ ,}$$

where stm_{ass} is $\vec{y} := \vec{e}$ or $\langle \vec{y} := \vec{e}\rangle$, $\tau_1(\mathsf{this}) = \alpha$, $\tau_1' = \tau_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega, \sigma_n(\alpha), \tau_1}]$, and $\sigma_{n+1} = \sigma_n[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega, \sigma_n(\alpha), \tau_1}]$.

Let $\tau_2(\mathsf{this}) = \beta$ and $\tau = \tau_1[\vec{u}', \mathsf{this}' \mapsto \tau_2(\vec{u}), \tau_2(\mathsf{this})]$, where $\vec{u}$ are the local variables from the domain of τ_2 and $\vec{u}'$ fresh variables.

If $\alpha \neq \beta$, then $\sigma_n(\beta) = \sigma_{n+1}(\beta)$, and we get $\omega, \sigma_{n+1}(\beta), \tau_2 \models_{\mathcal{L}} pre(stm)$ by assumption.

Assume in the following $\alpha = \beta$. Then $\omega, \sigma_n(\beta), \tau \models_{\mathcal{L}} \mathsf{this} = \mathsf{this}'$. By assumption and by the definition of τ we get $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$ and $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} pre(stm)$. As the assignment is enabled in $\langle T_n, \sigma_n\rangle$, so by Lemma 7 $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$. Furthermore, using Lemma 15 we get that $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$. Condition (5) of the interference freedom test implies $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} pre(stm)[\vec{e}/\vec{y}]$. Using Lemma 1 and the definition of τ yields the required property. □

Lemma 19 (Cooperation test: Method invocation). *Let $\langle T_n, \sigma_n\rangle$ be a reachable configuration of a proof outline satisfying the verification conditions of the cooperation test for communication, and let $\langle T_n, \sigma_n\rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1}\rangle \longrightarrow \langle T_{n+2}, \sigma_{n+2}\rangle \longrightarrow \langle T_{n+3}, \sigma_{n+3}\rangle$ result from executing a method invocation in a local configuration $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1\rangle; stm_1) \in T_n$ and the corresponding observations. Let $(\tau_2, \langle \vec{y}_2 := \vec{e}_2\rangle; stm_2) \in T_{n+1}$ be the callee configuration after communication. Then*

$$\begin{aligned} \omega, \sigma_n(\tau_1(\mathsf{this})), \tau_1 &\models_{\mathcal{L}} pre(e_0.m(\vec{e})) \text{ ,} \\ \omega, \sigma_n(\tau_2(\mathsf{this})), \tau &\models_{\mathcal{L}} I \text{ , } \textit{and} \\ \omega, \sigma_n &\models_{\mathcal{G}} GI \end{aligned}$$

imply

$$\begin{aligned} \omega, \sigma_{n+1}(\tau_1(\mathsf{this})), \tau_1 &\models_{\mathcal{L}} pre(\langle \vec{y}_1 := \vec{e}_1\rangle; stm_1) \text{ ,} \\ \omega, \sigma_{n+1}(\tau_2(\mathsf{this})), \tau_2 &\models_{\mathcal{L}} pre(\langle \vec{y}_2 := \vec{e}_2\rangle; stm_2) \text{ , } \textit{and} \\ \omega, \sigma_{n+3} &\models_{\mathcal{G}} GI \end{aligned}$$

for arbitrary $\omega \in \Omega$ and $\tau \in \Sigma_{\mathsf{loc}}$, where I is the class invariant of the callee object.

Proof (of Lemma 19). Let caller and callee be given by $\tau_1(\mathsf{this}) = \alpha$ and $\tau_2(\mathsf{this}) = [\![e_0]\!]_{\mathcal{L}}^{\omega,\sigma_n(\alpha),\tau_1} = \beta$, both different from *nil*. Furthermore, the callee's local state is defined by $\tau_2 = \tau_{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_n(\alpha),\tau_1}][\mathsf{this} \mapsto \beta]$, where $\vec{u}$ are the formal parameters of the method invocation. In addition, if m is synchronized, then $\mathit{isfree}(T \backslash \{\xi \circ (\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1\rangle; \mathit{stm}_1)\}, \beta)$.

Since the method invocation is enabled, the global configuration $\langle T_n, \sigma_n\rangle$ prior to the method call is stable. By assumption

$$\begin{aligned}\omega, \sigma_n(\alpha), \tau_1 &\models_{\mathcal{L}} \mathit{pre}(e_0.m(\vec{e}))\\ \omega, \sigma_n(\beta), \tau &\models_{\mathcal{L}} I\\ \omega, \sigma_n &\models_{\mathcal{G}} \mathit{GI}\ ,\end{aligned}$$

where I is the class invariant of the callee.

Let z and z' be fresh logical variables of appropriate type and let $\omega' = \omega[z \mapsto \alpha][z' \mapsto \beta][\vec{v} \mapsto \tau_1(\vec{v})]$, where $\vec{v}$ are the local variables of the caller viewed as fresh logical variables in the global language. Since logical variables may not occur free in the annotation, we have

$$\begin{aligned}\omega', \sigma_n(\alpha), \tau_1 &\models_{\mathcal{L}} \mathit{pre}(e_0.m(\vec{e}))\\ \omega', \sigma_n(\beta), \tau &\models_{\mathcal{L}} I\\ \omega', \sigma_n &\models_{\mathcal{G}} \mathit{GI}\ ,\end{aligned}$$

and further with the substitution Lemma 3

$$\begin{aligned}\omega', \sigma_n &\models_{\mathcal{G}} \mathit{pre}(e_0.m(\vec{e}))[z/\mathsf{this}]\\ \omega', \sigma_n &\models_{\mathcal{G}} I[z'/\mathsf{this}]\ .\end{aligned}$$

By definition of ω' and the assumption on the value of e_0, $[\![e_0]\!]_{\mathcal{L}}^{\omega',\sigma_n(\alpha),\tau_1} = \beta$ and further $\omega', \sigma_n \models_{\mathcal{G}} (e_0[z/\mathsf{this}] = z')$. If method m is synchronized, $\mathit{isfree}(T \backslash \{\xi \circ (\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1\rangle; \mathit{stm}_1)\}, \beta)$ implies with Lemma 5 $\sigma_n(\beta)(\mathsf{lock}) = (\mathit{nil}, 0) \vee \sigma_n(\beta)(\mathsf{lock}) < \tau_1(\mathsf{id})$, consequently $\omega', \sigma_n \models_{\mathcal{G}} z'.\mathsf{lock} = (\mathsf{nil}, 0) \vee z'.\mathsf{lock} \leq \mathsf{id}$. Furthermore, if $m = \mathsf{start}$, from the predicate $\neg\mathit{started}$ in rule START we get additionally using Lemma 6 $\omega', \sigma_n \models_{\mathcal{G}} \neg z'.\mathsf{started}$. Thus, by Equation (6) of the cooperation test,

$$\begin{aligned}\omega', \sigma_n \models_{\mathcal{G}} \mathit{post}(e_0.m(\vec{e}))[z/\mathsf{this}] \wedge \mathit{pre}'(\langle \vec{y}_2 := \vec{e}_2\rangle; \mathit{stm}_2)[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge\\ \mathit{GI}[\vec{E}_2'/z'.\vec{y}_2][\vec{E}_1/z.\vec{y}_1]\ ,\end{aligned}$$

where $\vec{E} = \vec{e}[z/\mathsf{this}]$, $\vec{E}_1 = \vec{e}_1[z/\mathsf{this}]$, and $\vec{E}_2' = \vec{e}_2'[z', \vec{E}/\mathsf{this}, \vec{u}]$. For the local assertions, Lemma 3 together with $\omega'(z) = \tau_1(\mathsf{this})$, $\omega'(z') = \tau_2(\mathsf{this})$, and $\omega(\vec{v}) = \tau_1(\vec{v})$ implies

$$\begin{aligned}\omega', \sigma_n(\alpha), \tau_1 &\models_{\mathcal{L}} \mathit{pre}(\langle \vec{y}_1 := \vec{e}_1\rangle; \mathit{stm}_1)\\ \omega', \sigma_n(\beta), \tau_2 &\models_{\mathcal{L}} \mathit{pre}(\langle \vec{y}_2 := \vec{e}_2\rangle; \mathit{stm}_2)\ .\end{aligned}$$

Since logical variables may not occur free in the annotation, and since $\sigma_n = \sigma_{n+1}$, we get as required

$$\begin{aligned}\omega, \sigma_{n+1}(\alpha), \tau_1 &\models_{\mathcal{L}} pre(\langle \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \\ \omega, \sigma_{n+1}(\beta), \tau_2 &\models_{\mathcal{L}} pre(\langle \vec{y}_2 := \vec{e}_2 \rangle; stm_2) \ .\end{aligned}$$

For the global invariant we observe that the global state is not influenced by the communication itself; after the assignment $\vec{y}_1 := \vec{e}_1$ of the caller it is given by $\sigma_{n+2} = \sigma_n[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{L}}^{\sigma_n(\alpha),\tau_1}]$, and after the execution of $\vec{y}_2 := \vec{e}_2$ by the callee, the resulting global state is $\sigma_{n+3} = \sigma_{n+2}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{L}}^{\sigma_{n+2}(\beta),\tau_2}]$. Using the substitution Lemma 3 we get

$$[\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![\vec{e}_1[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![\vec{e}_1]\!]_{\mathcal{L}}^{\omega',\sigma_n(\alpha),\tau_1},$$

and hence $\sigma_{n+2} = \sigma_n[\alpha.\vec{y}_1 \mapsto [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega',\sigma_n}]$. Using the same substitution lemma once more yields

$$[\![\vec{E}_2']\!]_{\mathcal{G}}^{\omega',\sigma_{n+2}} = [\![\vec{e}_2'[z',\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega',\sigma_{n+2}} = [\![\vec{e}_2]\!]_{\mathcal{L}}^{\omega,\sigma_{n+2}(\beta),\tau_2}.$$

Therefore, $\sigma_{n+3} = \sigma_{n+2}[\beta.\vec{y}_2 \mapsto [\![\vec{E}_2']\!]_{\mathcal{G}}^{\omega',\sigma_{n+2}}]$. Now, applying twice the substitution Lemma 2 and using the above equalities yields $\omega', \sigma_{n+3} \models_{\mathcal{G}} GI$. Since GI may not contain free occurrences of logical variables, also $\omega, \sigma_{n+1} \models_{\mathcal{G}} GI$, as required. □

Lemma 20 (Cooperation test: Start$_{skip}$). *Let $\langle T_n, \sigma_n \rangle$ be a reachable configuration of a proof outline satisfying the verification conditions of the cooperation test for communication, and let $\langle T_n, \sigma_n \rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1} \rangle \longrightarrow \langle T_{n+2}, \sigma_{n+2} \rangle$ result from calling the start-method of an object whose thread is already started in a local configuration $(\tau_1, \langle e_0.\mathsf{start}(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T_n$, and executing the corresponding observation. Then*

$$\begin{aligned}\omega, \sigma_n(\tau_1(\mathsf{this})), \tau_1 &\models_{\mathcal{L}} pre(e_0.m(\vec{e})) \ , \\ \omega, \sigma_n([\![e_0]\!]_{\mathcal{L}}^{\sigma_n(\tau_1(\mathsf{this})),\tau_1}), \tau &\models_{\mathcal{L}} I \ , \ \ \textit{and} \\ \omega, \sigma_n &\models_{\mathcal{G}} GI\end{aligned}$$

imply

$$\begin{aligned}\omega, \sigma_{n+1}(\tau_1(\mathsf{this})), \tau_1 &\models_{\mathcal{L}} pre(\langle \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \ , \ \ \textit{and} \\ \omega, \sigma_{n+2} &\models_{\mathcal{G}} GI\end{aligned}$$

for arbitrary $\omega \in \Omega$ and $\tau \in \Sigma_{\mathsf{loc}}$, where I is the class invariant of the callee object.

Proof (of Lemma 20). The proof is analogous to the case of ordinary method invocation (cf. proof of Lemma 19), where the additional antecedent z'.started of condition (8) of the cooperation test is implied by the predicate *started* in rule START$_{\mathsf{skip}}$ and again by Lemma 6. □

Lemma 21 (Cooperation test: Return). *Let $\langle T_n, \sigma_n\rangle$ be a reachable configuration of a proof outline satisfying the verification conditions of the cooperation test for communication, and let $\langle T_n, \sigma_n\rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1}\rangle \longrightarrow \langle T_{n+2}, \sigma_{n+2}\rangle \longrightarrow \langle T_{n+3}, \sigma_{n+3}\rangle$ result from communicating the return value of a method in the local configurations $(\tau_1, \langle \mathsf{receive}\, u; \vec{y}_4 := \vec{e}_4\rangle; stm_2) \in T_n$ and $(\tau_2, \langle \mathsf{return}\, e_{ret}; \vec{y}_3 := \vec{e}_3\rangle) \in T_n$, and the execution corresponding observations. Let τ_1' be the local state of the callee after communication. Then*

$$\begin{aligned} \omega, \sigma_n(\tau_1(\mathsf{this})), \tau_1 &\models_{\mathcal{L}} pre(\mathsf{receive}\, u)\ , \\ \omega, \sigma_n(\tau_2(\mathsf{this})), \tau_2 &\models_{\mathcal{L}} pre(\mathsf{return}\ e_{ret})\ , \ \textit{and} \\ \omega, \sigma_n &\models_{\mathcal{G}} GI \end{aligned}$$

imply

$$\begin{aligned} \omega, \sigma_{n+1}(\tau_1'(\mathsf{this})), \tau_1' &\models_{\mathcal{L}} pre(\langle \vec{y}_4 := \vec{e}_4\rangle; stm_2)\ , \\ \omega, \sigma_{n+1}(\tau_2(\mathsf{this})), \tau_2 &\models_{\mathcal{L}} pre(\langle \vec{y}_3 := \vec{e}_3\rangle)\ , \ \textit{and} \\ \omega, \sigma_{n+3} &\models_{\mathcal{G}} GI \end{aligned}$$

for arbitrary $\omega \in \Omega$.

Proof (of Lemma 21). Let $\tau_1(\mathsf{this}) = \alpha$ be the caller object, $\tau_2(\mathsf{this}) = \beta$ the callee, both different from *nil* Then $\tau_1' = \tau[u \mapsto [\![e_{ret}]\!]_{\mathcal{L}}^{\omega,\sigma_n(\beta),\tau_2}]$ is the updated local state of the caller after receiving the return value. By the assumptions we have

$$\begin{aligned} \omega, \sigma_n(\alpha), \tau_1 &\models_{\mathcal{L}} pre(\mathsf{receive}\, u) \\ \omega, \sigma_n(\beta), \tau_2 &\models_{\mathcal{L}} pre(\mathsf{return}\ e_{ret}) \\ \omega, \sigma_n &\models_{\mathcal{G}} GI\ . \end{aligned}$$

Let z and z' be fresh logical variables not occurring in *prog'*, and let furthermore $\omega' = \omega[z \mapsto \alpha][z' \mapsto \beta][\vec{v}_1 \mapsto \tau_1(\vec{v}_1)][\vec{v}_2' \mapsto \tau_2(\vec{v}_2)]$, where $\vec{v}_1$ are the local variables of the caller, and $\vec{v}_2$ the local variables of the callee except the formal parameters and this, viewed as disjoint fresh logical variables in the global language. As the logical variables do not occur free in *prog'*, we have

$$\begin{aligned} \omega', \sigma_n(\alpha), \tau_1 &\models_{\mathcal{L}} pre(\mathsf{receive}\, u) \\ \omega', \sigma_n(\beta), \tau_2 &\models_{\mathcal{L}} pre(\mathsf{return}\ e_{ret}) \\ \omega', \sigma_n &\models_{\mathcal{G}} GI\ . \end{aligned}$$

Since actual parameters are not allowed to contain instance variables, and since formal parameters may not be assigned to, their values remain *unchanged* during the execution of the invoked method, and thus $\tau_2(\vec{u}) = [\![\vec{e}]\!]_{\mathcal{L}}^{\omega',\sigma_n(\alpha),\tau_1}$, where $\vec{u}$ are the formal parameters of the method considered, and $\vec{e}$ its actual parameters. By the substitution Lemma 3

$$\tau_2(\vec{u}) = [\![\vec{e}]\!]_{\mathcal{L}}^{\omega',\sigma_n(\alpha),\tau_1} = [\![\vec{e}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![\vec{E}]\!]_{\mathcal{G}}^{\omega',\sigma_n}\ .$$

Using the same lemma once more gives

$$\omega', \sigma_n \models_{\mathcal{G}} pre(\mathsf{receive}\, u)[z/\mathsf{this}]$$
$$\omega', \sigma_n \models_{\mathcal{G}} pre'(\mathsf{return}\, e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] ,$$

where $pre'(\mathsf{return}\, e_{ret})$ results from $pre(\mathsf{return}\, e_{ret})$ by replacing the local variables from $\vec{v}_2$ by the corresponding logical variables from $\vec{v}'_2$.

If the receive statement is preceded by a bracketed section invoking a method of object e_0, then since e_0 may not contain instance variables, i.e., its value does not change during the execution of the invoked method, the callee object β is given by $[\![e_0]\!]_{\mathcal{L}}^{\omega,\sigma_n(\alpha),\tau_1}$. This implies using the definition of ω' that $\omega', \sigma_n \models_{\mathcal{G}} (e_0[z/\mathsf{this}] = z')$. Now, Equation (7) of the cooperation test gives

$$\omega', \sigma_n \models_{\mathcal{G}} post(\mathsf{receive}\, u)[z, E'_{ret}/\mathsf{this}, u] \wedge post'(\mathsf{return}\, e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge$$
$$GI[\vec{E}_4/z.\vec{y}_4][\vec{E}'_3/z'.\vec{y}_3] ,$$

where $\vec{E}'_3 = \vec{e}'_3[z', \vec{E}/\mathsf{this}, \vec{u}]$ with $\vec{E} = \vec{e}[z/\mathsf{this}]$, and $\vec{E}_4 = \vec{e}_4[z, E'_{ret}/\mathsf{this}, u]$ with $E'_{ret} = e'_{ret}[z', \vec{E}/\mathsf{this}, \vec{u}]$.

For the local assertions, Lemma 3 and using $\omega'(z') = \beta$ and $\tau_2(\vec{u}) = [\![\vec{E}]\!]_{\mathcal{G}}^{\omega',\sigma_n}$ further gives

$$[\![E'_{ret}]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![e'_{ret}[z', \vec{E}/\mathsf{this}, \vec{u}]]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![e_{ret}]\!]_{\mathcal{L}}^{\omega',\sigma_n(\beta),\tau_2} = \tau'_1(u) ,$$

where e'_{ret} results from e_{ret} by substituting all local variables from $\vec{v}_2$ by the corresponding logical variables from $\vec{v}'_2$. Using Lemma 3 again we get

$$\omega', \sigma_n(\alpha), \tau'_1 \models_{\mathcal{L}} post(\mathsf{receive}\, u)$$
$$\omega', \sigma_n(\beta), \tau_2 \models_{\mathcal{L}} post(\mathsf{return}\, e_{ret}) .$$

As $prog'$ does not contain free occurrences of logical variables, furthermore $post(\mathsf{receive}\, u) = pre(\langle \vec{y}_4 := \vec{e}_4\rangle; stm_1)$, $post(\mathsf{return}\, e_{ret}) = pre(\langle \vec{y}_3 := \vec{e}_3\rangle)$, and $\sigma_n = \sigma_{n+1}$, we obtain

$$\omega, \sigma_{n+1}(\alpha), \tau'_1 \models_{\mathcal{L}} pre(\langle \vec{y}_4 := \vec{e}_4\rangle; stm_1)$$
$$\omega, \sigma_{n+1}(\beta), \tau_2 \models_{\mathcal{L}} pre(\langle \vec{y}_3 := \vec{e}_3\rangle) .$$

For the global invariant,

For the global invariant we observe that the local state of the caller after communication is given by $\tau'_1 = \tau_1[u \mapsto [\![e_{ret}]\!]_{\mathcal{L}}^{\sigma_n(\beta),\tau_2}]$. For the global states we have $\sigma' = \sigma_n[\beta.\vec{y}_3 \mapsto [\![\vec{e}_3]\!]_{\mathcal{L}}^{\sigma_n(\beta),\tau_2}]$ and $\sigma_{n+1} = \sigma'[\alpha.\vec{y}_4 \mapsto [\![\vec{e}_4]\!]_{\mathcal{L}}^{\sigma'(\alpha),\tau'_1}]$. With the help of Lemma 3 and with $\tau_2(\vec{u}) = [\![\vec{E}]\!]_{\mathcal{G}}^{\omega',\sigma_n}$, we obtain

$$[\![\vec{E}'_3]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![\vec{e}'_3[z', \vec{E}/\mathsf{this}, \vec{u}]]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![\vec{e}_3]\!]_{\mathcal{L}}^{\omega',\sigma_n(\beta),\tau_2} = [\![\vec{e}_3]\!]_{\mathcal{E}}^{\sigma_n(\beta),\tau_2} .$$

This implies $\sigma_n[\beta.\vec{y}_3 \mapsto [\![\vec{E}'_3]\!]_{\mathcal{G}}^{\omega',\sigma_n}] = \sigma_{n+2}$, and therefore applying Lemma 2 to the global invariant in Equation (B.2.2) yields

$$\omega', \sigma_{n+2} \models_{\mathcal{G}} GI[\vec{E}_4/z.\vec{y}_4] . \tag{11}$$

We further get with Lemma 3

$$\tau_1'(u) = [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma_n(\beta),\tau_2} = [\![e_{ret}]\!]_{\mathcal{L}}^{\omega',\sigma_n(\beta),\tau_2} = [\![e'_{ret}[z'/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![E'_{ret}]\!]_{\mathcal{G}}^{\omega',\sigma_n} \;,$$

and by the same lemma again

$$[\![\vec{E}_4]\!]_{\mathcal{G}}^{\omega',\sigma_{n+2}} = [\![\vec{e}_4[z, E'_{ret}/\mathsf{this}, u]]\!]_{\mathcal{G}}^{\omega',\sigma_{n+2}} = [\![\vec{e}_4]\!]_{\mathcal{L}}^{\omega',\sigma_{n+2}(\alpha),\tau_1'},$$

and hence $\sigma_{n+2}[\alpha.\vec{y}_4 \mapsto [\![\vec{E}_4]\!]_{\mathcal{G}}^{\omega',\sigma_{n+2}}] = \sigma_{n+3}$. Therefore, applying Lemma 2 to Equation (11) yields $\omega', \sigma_{n+3} \models_{\mathcal{G}} GI$. Since GI does not contain free logical variables, also $\omega, \sigma_{n+3} \models_{\mathcal{G}} GI$, as required. □

Lemma 22 (Cooperation test: Terminate). *Let $\langle T_n, \sigma_n\rangle$ be a reachable configuration of a proof outline satisfying the verification conditions of the cooperation test for communication, and let $\langle T_n, \sigma_n\rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1}\rangle \longrightarrow \langle T_{n+2}, \sigma_{n+2}\rangle$ result from executing the return-statement of a start-method or of the initial invocation of the main-method in a local configuration $(\tau_1, \langle \mathsf{return}; \vec{y}_3 := \vec{e}_3\rangle) \in T_n$, and executing the corresponding observation. Then*

$$\omega, \sigma_n(\tau_1(\mathsf{this})), \tau_1 \models_{\mathcal{L}} pre(\mathsf{return}) \;,\; \textit{and}$$
$$\omega, \sigma_n \models_{\mathcal{G}} GI$$

imply

$$\omega, \sigma_{n+1}(\tau_1(\mathsf{this})), \tau_1 \models_{\mathcal{L}} pre(\langle \vec{y}_3 := \vec{e}_3\rangle) \;,\; \textit{and}$$
$$\omega, \sigma_{n+2} \models_{\mathcal{G}} GI$$

for arbitrary $\omega \in \Omega$.

Proof (of lemma 22). Let $\tau_1(\mathsf{this}) = \alpha$. Executing the return statement at the end of the initial invocation of the main-method or at the end of a start-method changes only the control point, but no states. By assumption

$$\omega, \sigma_n(\alpha), \tau_1 \models_{\mathcal{L}} pre(\mathsf{return})$$
$$\omega, \sigma_n \models_{\mathcal{G}} GI \;.$$

Let $\omega' = \omega[z' \mapsto \alpha][\vec{v} \mapsto \tau_1(\vec{v})]$, where $\vec{v}$ are the local variables from the domain of τ_1 viewed as logical variables on the global level, and where z' is a fresh logical variable. Since the annotation does not contain free logical variables, also

$$\omega', \sigma_n(\alpha), \tau_1 \models_{\mathcal{L}} pre(\mathsf{return})$$
$$\omega', \sigma_n \models_{\mathcal{G}} GI \;.$$

Using the substitution Lemma 3 we get

$$\omega', \sigma_n \models_{\mathcal{G}} pre(\mathsf{return})[z'/\mathsf{this}] \wedge GI \;.$$

Furthermore, $\tau_1(\mathsf{id}) = (\tau_1(\mathsf{this}), 0)$, $\omega'(z') = \alpha = \tau_1(\mathsf{this})$, and $\omega'(\mathsf{id}) = \tau_1(\mathsf{id})$ imply $\omega', \sigma_n \models_{\mathcal{G}} \mathsf{id} = (z', 0)$. By Equation (9) of the cooperation test for communication, $\omega', \sigma_n \models_{\mathcal{G}} post(\mathsf{return})[z'/\mathsf{this}] \wedge GI[\vec{E}_3/z'.\vec{y}_3]$, where $\vec{E}_3 = \vec{e}_3[z'/\mathsf{this}]$. Applying Lemma 3 again yields for the local assertion that $\omega', \sigma_n(\alpha), \tau_1 \models_{\mathcal{L}} post(\mathsf{return})$. Since the annotation does not contain free logical variables, we get with $\sigma_n = \sigma_{n+1}$ that $\omega, \sigma_{n+1}(\alpha), \tau_1 \models_{\mathcal{L}} pre(\langle \vec{y}_3 := \vec{e}_3 \rangle)$, as required.

For the global invariant we observe that $\sigma_{n+2} = \sigma_n[\alpha.\vec{y}_3 \mapsto [\![\vec{e}_3]\!]_{\mathcal{L}}^{\sigma_n(\alpha),\tau}]$. Applying Lemma 3 yields $[\![\vec{E}_3]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![\vec{e}_3[z'/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega',\sigma_n} = [\![\vec{e}_3]\!]_{\mathcal{L}}^{\omega',\sigma_n(\beta),\tau}$. Using the above equalities and Lemma 2 we get $\omega', \sigma_{n+2} \models_{\mathcal{G}} GI$. Since GI does not contain free logical variables, finally $\omega, \sigma_{n+2} \models_{\mathcal{G}} GI$. $\square$

Lemma 23 (Cooperation test: Instantiation). *Let $\langle T_n, \sigma_n \rangle$ be a reachable configuration of a proof outline satisfying the verification conditions of the cooperation test for instantiation, and let $\langle T_n, \sigma_n \rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1} \rangle \longrightarrow \langle T_{n+2}, \sigma_{n+2} \rangle$ result from the creation of a new object in a local configuration $(\tau_1, \langle u := \mathsf{new}^c; \vec{y} := \vec{e} \rangle; stm_1) \in T_n$, and executing the corresponding observation. Let τ_1' be the local state of the executing thread after object creation. Then*

$$\omega, \sigma_n(\tau_1(\mathsf{this})), \tau_1 \models_{\mathcal{L}} pre(u := \mathsf{new}^c) \text{ , } \textit{and}$$
$$\omega, \sigma_n \models_{\mathcal{G}} GI$$

imply

$$\omega, \sigma_{n+1}(\tau_1'(\mathsf{this})), \tau_1' \models_{\mathcal{L}} pre(\langle \vec{y} := \vec{e} \rangle; stm_1) \text{ ,}$$
$$\omega, \sigma_{n+1}(\tau_1'(u)), \tau \models_{\mathcal{L}} I \text{ , } \textit{and}$$
$$\omega, \sigma_{n+2} \models_{\mathcal{G}} GI$$

for arbitrary $\omega \in \Omega$ and $\tau \in \Sigma_{\mathsf{loc}}$, where I is the class invariant of the new object.

Proof (of Lemma 23). Let $\tau_1(\mathsf{this}) = \alpha$ and $\beta \notin dom_{nil}(\sigma_n)$ the newly created object. Then $\tau_1' = \tau_1[u \mapsto \beta]$, and $\sigma_{n+1} = \sigma_n[\beta \mapsto \sigma_{inst}^{c,init}]$. Note that $\langle T_n, \sigma_n \rangle$ is stable, since the object creation statement is enabled. By assumption

$$\omega, \sigma_n \models_{\mathcal{G}} GI$$
$$\omega, \sigma_n(\alpha), \tau_1 \models_{\mathcal{L}} pre(u := \mathsf{new}^c) \text{ .}$$

Let $\vec{v}$ be the local variables from the domain of τ_1. For the logical environment $\omega' = \omega[z \mapsto \alpha][\vec{v} \mapsto \tau_1(\vec{v})]$, with fresh logical variables z and $\vec{v}$, also

$$\omega', \sigma_n \models_{\mathcal{G}} GI$$
$$\omega', \sigma_n(\alpha), \tau_1 \models_{\mathcal{L}} pre(u := \mathsf{new}^c) \text{ .}$$

Applying the substitution Lemma 3 we have $\omega', \sigma_n \models_{\mathcal{G}} pre(u := \mathsf{new}^c)[z/\mathsf{this}]$. As $\tau_1(u) = \omega'(u) \in dom_{nil}(\sigma_n)$, this implies

$$\omega', \sigma_n \models_{\mathcal{G}} GI \wedge \exists u (pre(u := \mathsf{new}^c)[z/\mathsf{this}]) \text{ .}$$

As $\sigma_{n+1} = \sigma_n[\beta \mapsto \sigma_{inst}^{init}]$ and $\beta \notin dom_{nil}(\sigma_n)$, Lemma 8 gives

$$\omega'[z' \mapsto dom_{nil}(\sigma_n)], \sigma_{n+1} \models_{\mathcal{G}} (GI \wedge \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}])) \downarrow z' \ .$$

The logical variable u does not occur free in the above assertion, so we further obtain $\omega'[z' \mapsto dom_{nil}(\sigma_n)][u \mapsto \beta], \sigma_{n+1} \models_{\mathcal{G}} (GI \wedge \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}])) \downarrow z'$. Since $\beta \notin dom_{nil}(\sigma_n)$ is the unique new element in $dom(\sigma_{n+1})$ being in its initial state, we obtain that $\omega'[z' \mapsto dom_{nil}(\sigma_n)][u \mapsto \beta], \sigma_{n+1} \models_{\mathcal{G}} \mathsf{Fresh}(z', u)$. Therefore, by the semantics of global assertions,

$$\omega'[u \mapsto \beta], \sigma_{n+1} \models_{\mathcal{G}} \exists z'(\ \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}])) \downarrow z'\ ) \ .$$

From this, we get with the cooperation test for object creation (Equation (10))

$$\omega'[u \mapsto \beta], \sigma_{n+1} \models_{\mathcal{G}} post(u := \mathsf{new}^c)[z/\mathsf{this}] \wedge I_c[u/\mathsf{this}] \wedge GI[\vec{E}/z.\vec{y}] \ ,$$

where $\vec{E} = \vec{e}[z/\mathsf{this}]$. For the local assertions we get with the substitution Lemma 3

$$\begin{aligned} \omega'[u \mapsto \beta], \sigma_{n+1}(\alpha), \tau_1' \models_{\mathcal{L}} post(u := \mathsf{new}^c) \\ \omega'[u \mapsto \beta], \sigma_{n+1}(\beta), \tau \models_{\mathcal{L}} I \ . \end{aligned}$$

According to the definition of annotation on page 23, no free logical variables occur in $post(u := \mathsf{new}^c)$ and in I, and hence we get $\omega, \sigma_{n+1}(\alpha), \tau_1' \models_{\mathcal{L}} pre(stm)$ and $\omega, \sigma_{n+1}(\tau_1'(u)), \tau \models_{\mathcal{L}} I$, as required.

For the global invariant we observe that $\sigma_{n+1} = \sigma_n[\beta \mapsto \sigma_{inst}^{c,init}]$ and $\sigma_{n+2} = \sigma_{n+1}[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\sigma_{n+1}(\alpha),\tau_1'}]$. By the substitution Lemma 2 and $\omega'(z) = \alpha$ thus

$$\omega', \sigma_{n+1}[\alpha.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega',\sigma_{n+1}}] \models_{\mathcal{G}} GI \ ,$$

and further with Lemma 3 $[\![\vec{E}]\!]_{\mathcal{G}}^{\omega',\sigma_{n+1}} = [\![\vec{e}]\!]_{\mathcal{L}}^{\omega',\sigma_{n+1}(\alpha),\tau_1'}$ and thus $\omega', \sigma_{n+2} \models_{\mathcal{G}} GI$. Since there are no free occurrences of logical variables in GI, also $\omega, \sigma_{n+2} \models_{\mathcal{G}} GI$, as required. □

B.2.3 Inductive soundness proof This section collects the previous soundness lemmas for the single proof conditions into the inductive soundness proof for the whole system. We split the inductive step into preservation for the local precondition, for the class invariant, and for the global invariant, before we wrap up the results into the soundness proof of Theorem 1.

Lemma 24 (Induction step: preconditions). *Given a proof outline $prog'$ that satisfies the verification conditions and a reachable configuration $\langle T_n, \sigma_n \rangle$ of $prog'$. Assume further that $\omega, \sigma_n(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} pre(stm)$ for all $(\tau, stm) \in T_n$ and all logical environments ω referring only to values existing in σ_n. Additionally assume $\omega, \sigma_n \models_{\mathcal{G}} GI$ to hold, if $\langle T_n, \sigma_n \rangle$ is stable. Furthermore, for all classes c', objects $\beta \in dom^{c'}(\sigma_n)$, and local states τ' assume $\omega, \sigma_n(\beta), \tau' \models_{\mathcal{L}} I_{c'}$.*

Then $\langle T_n, \sigma_n \rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1} \rangle$ implies $\omega, \sigma_{n+1}(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} pre(stm)$, for all $(\tau, stm) \in T_{n+1}$.

Proof (of Lemma 24). Let $(\tau, stm) \in T_{n+1}$ with $\tau(\mathsf{this}) = \alpha$. We show that $\omega, \sigma_{n+1}(\alpha), \tau \models_{\mathcal{L}} pre(stm)$ and distinguish according to the computation step $\langle T_n, \sigma_n \rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1} \rangle$.

Case: ASS
We are given

$$\langle T \dot{\cup} \{\xi \circ (\tau_1, stm_{ass}; stm_1) \circ \xi'\}, \sigma_n \rangle \longrightarrow \langle T \dot{\cup} \{\xi \circ (\tau_1', stm_1) \circ \xi'\}, \sigma_{n+1} \rangle \text{ , or}$$
$$\langle T \dot{\cup} \{\xi \circ (\tau_1, stm_{ass})\}, \sigma_n \rangle \longrightarrow \langle T \dot{\cup} \{\xi\}, \sigma_{n+1} \rangle \text{ .}$$

By assumption $\omega, \sigma_n(\tau_1(\mathsf{this})), \tau_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$, and if $(\tau, stm) \in T_n$, then additionally $\omega, \sigma_n(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} pre(stm)$.

If $(\tau, stm) = (\tau_1', stm_1)$, then soundness of the local correctness conditions (Lemma 17) implies the required property. Otherwise $(\tau, stm) \in T_n$, and soundness of the interference freedom test (Lemma 18) implies $\omega, \sigma_{n+1}(\tau(\mathsf{this})), \tau \models_{\mathcal{L}} pre(stm)$.

Case: CALL
In this case we are given

$$\begin{array}{ll}\langle T \dot{\cup} \{\xi \circ (\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1)\}, \sigma_n \rangle & \longrightarrow \\ \langle T \dot{\cup} \{\xi \circ (\tau_1, \langle \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \circ (\tau_2, body_{m,c})\}, \sigma_{n+1} \rangle \text{ ,} & \end{array}$$

where m is not the start-method. If $(\tau, stm) = (\tau_1, \langle \vec{y}_1 := \vec{e}_1 \rangle; stm_1)$ or $(\tau, stm) = (\tau_2, body_{m,c})$, then the assumptions and soundness of the cooperation test for communication (Lemma 19) implies the required property. If otherwise $(\tau, stm) \in T$ or $(\tau, stm) \in \xi$, then directly by assumption $\omega, \sigma_{n+1}(\alpha), \tau \models_{\mathcal{L}} pre(stm)$, since method invocation does not change the global state,

Case: START, $\text{START}_{\mathsf{skip}}$
Rule START is treated analogously to the above case of ordinary method invocation using the assumptions and Lemma 19. In case of $\text{START}_{\mathsf{skip}}$ Lemma 20 is used instead, stating soundness of the verification condition (8) of the cooperation test for $\text{START}_{\mathsf{skip}}$.

Case: RETURN
In this case we are given

$$\langle T \dot{\cup} \{\xi \circ (\tau_1, \langle \mathsf{receive}\, u; \vec{y}_4 := \vec{e}_4 \rangle; stm_1) \circ (\tau_2, \langle \mathsf{return}\ e_{ret}; \vec{y}_3 := \vec{e}_3 \rangle)\}, \sigma_n \rangle \longrightarrow$$
$$\langle T \dot{\cup} \{\xi \circ (\tau_1', \langle \vec{y}_4 := \vec{e}_4 \rangle; stm_1) \circ (\tau_2, \langle \vec{y}_3 := \vec{e}_3 \rangle)\}, \sigma_{n+1} \rangle \text{ .}$$

If $(\tau, stm) = (\tau_1', \langle \vec{y}_4 := \vec{e}_4 \rangle; stm_1)$ or $(\tau, stm) = (\tau_2, \langle \vec{y}_3 := \vec{e}_3 \rangle)$, then, as the communication is enabled, the global configuration $\langle T_n, \sigma_n \rangle$ prior to the communication is stable. Thus using the assumptions and Lemma 21 stating soundness of the cooperation test for return (cf. Equation (7)) we get the required property. If otherwise $(\tau, stm) \in \xi$ or $(\tau, stm) \in T$, then $(\tau, stm) \in T_n$, and by assumption $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} pre(stm)$. From $\sigma_n = \sigma_{n+1}$ the property follows directly.

Case: TERMINATE

$$\langle T \dot\cup \{(\tau_1, \langle \mathsf{return}; \vec{y}_3 := \vec{e}_3\rangle)\}, \sigma_n\rangle \longrightarrow \langle T \dot\cup \{(\tau_1, \langle \vec{y}_3 := \vec{e}_3\rangle)\}, \sigma_{n+1}\rangle \ .$$

Then $\langle T_n, \sigma_n\rangle$ is stable. In the case if $(\tau, stm) = (\tau_1, \langle \vec{y}_3 := \vec{e}_3\rangle)$, we get the required property by using the assumptions and Lemma 22 about the soundness of the termination (cf. Equation (9) of the cooperation test for communication). If otherwise $(\tau, stm) \in T$, then $(\tau, stm) \in T_n$, and the assumption $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} pre(stm)$ and $\sigma_n = \sigma_{n+1}$ yield the required property.

Case: NEW

$$\begin{aligned}&\langle T \dot\cup \{\xi \circ (\tau_1, \langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle; stm_1)\}, \sigma_n\rangle \longrightarrow \\ &\langle T \dot\cup \{\xi \circ (\tau_1', \langle \vec{y} := \vec{e}\rangle; stm_1)\}, \sigma_{n+1}\rangle \ .\end{aligned}$$

Note that $\langle T_n, \sigma_n\rangle$ is stable, since the object creation statement is enabled. If $(\tau, stm) = (\tau_1', \langle \vec{y} := \vec{e}\rangle; stm_1)$, then by assumption and using Lemma 23 for the the soundness of the cooperation test for instantiation we get the required property. If otherwise $(\tau, stm) \in T$ or $(\tau, stm) \in \xi$, then $(\tau, stm) \in T_n$, and by assumption $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} pre(stm)$. According to the definition of global configurations, $\alpha = \tau(\mathsf{this}) \in dom(\sigma_n)$, i.e., $\alpha \neq \gamma$. Hence $\sigma_n(\alpha) = \sigma_{n+1}(\alpha)$, and finally $\omega, \sigma_{n+1}(\alpha), \tau \models_{\mathcal{L}} pre(stm)$. □

Lemma 25 (Inductive step: Class invariant). *Let the proof outline prog′ satisfy the verification conditions and $\langle T_n, \sigma_n\rangle$ be a reachable configuration of prog′ such that for all $(\tau, stm) \in T_n$ with $\alpha = \tau(\mathsf{this})$ of type c and for all logical environments ω referring only to values existing in σ_n we have $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} pre(stm)$. If $\langle T_n, \sigma_n\rangle$ is stable, assume further $\omega, \sigma_n \models_{\mathcal{G}} GI$. Furthermore, for all classes c′, objects $\beta \in dom^{c'}(\sigma_n)$, and local states τ′, let $\omega, \sigma_n(\beta), \tau' \models_{\mathcal{L}} I_{c'}$.*

Then for all $\langle T_n, \sigma_n\rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1}\rangle$ and for all existing objects $\alpha \in dom(\sigma_{n+1})$ of type c, local states $\tau \in \Sigma_{\mathsf{loc}}$, and logical environments ω referring only to values existing in σ_{n+1},

$$\omega, \sigma_{n+1}(\alpha), \tau \models_{\mathcal{L}} I_c \ .$$

Proof (of Lemma 25). Let $\alpha \in dom^c(\sigma_{n+1})$, $\tau \in \Sigma_{\mathsf{loc}}$, and $\omega \in \Omega$ referring only to values existing in σ_{n+1}. We show that $\omega, \sigma_{n+1}(\alpha), \tau \models_{\mathcal{L}} I_c$, distinguishing on the last computation step.

Case: ASS

Let $\langle T_{n+1}, \sigma_{n+1}\rangle$ result from $\langle T_n, \sigma_n\rangle$ by executing the assignment $\vec{y} := \vec{e}$ in the local configuration $(\tau_1, stm_{ass}; stm_1) \in T_n$ where stm_{ass} is $\vec{y} := \vec{e}$ or $\langle \vec{y} := \vec{e}\rangle$. Then $\alpha \in dom^c(\sigma_n)$, and by assumption $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} I_c$. If $\tau_1(\mathsf{this}) \neq \alpha$, then $\sigma_n(\alpha) = \sigma_{n+1}(\alpha)$, and thus $\omega, \sigma_{n+1}(\alpha), \tau \models_{\mathcal{L}} I_c$, as required. If otherwise $\tau_1(\mathsf{this}) = \alpha$, then condition (3) of the local correctness combined with the local substitution Lemma 1 assures that $\omega, \sigma_{n+1}(\alpha), \tau_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega, \sigma_n(\alpha), \tau_1}] \models_{\mathcal{L}} post(\vec{y} := \vec{e})$. By the local correctness condition (4) for the class invariant

$\omega, \sigma_{n+1}(\alpha), \tau_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_n(\alpha),\tau_1}] \models_{\mathcal{L}} I_c$. As the class invariant may refer only to instance variables, its evaluation does not depend on the local state, hence $\omega, \sigma_{n+1}(\alpha), \tau \models_{\mathcal{L}} I_c$, as required.

Case: CALL, START, START$_{\text{skip}}$, RETURN, TERMINATE
In these cases the global state is not changed, i.e., $\sigma_n = \sigma_{n+1}$, and the property is directly implied by the assumption $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} I_c$.

Case: NEW
Let $\langle T_{n+1}, \sigma_{n+1}\rangle$ result from $\langle T_n, \sigma_n\rangle$ by executing an object creation statement. The instance states of objects existing prior to the last computation step are unchanged, i.e., if α is not the newly created object, then $\sigma_n(\alpha) = \sigma_{n+1}(\alpha)$, and $\omega, \sigma_{n+1}(\alpha), \tau \models_{\mathcal{L}} I_c$ follows from the assumption $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} I_c$.

If α is the newly created object, then the property is implied by the assumptions and using Lemma 23 which states soundness of the cooperation test for instantiation. □

Lemma 26 (Inductive step: Global invariant). *Let the proof outline prog′ satisfy the verification conditions and $\langle T_n, \sigma_n\rangle$ be a reachable stable configuration of prog′ such that for all $(\tau, stm) \in T_n$ with $\alpha = \tau(\mathsf{this})$ of type c and for all logical environments ω referring only to values existing in σ_n we have $\omega, \sigma_n(\alpha), \tau \models_{\mathcal{L}} pre(stm)$ and $\omega, \sigma_n \models_{\mathcal{G}} GI$. Furthermore, for all classes c', objects $\beta \in dom^{c'}(\sigma_n)$, and local states τ' let $\omega, \sigma_n(\beta), \tau' \models_{\mathcal{L}} I_{c'}$.*

Let $\langle T_n, \sigma_n\rangle \longrightarrow^* \langle T_{n+1}, \sigma_{n+1}\rangle$, such that $\langle T_{n+1}, \sigma_{n+1}\rangle$ is stable, and there are no stable configurations in the computation between $\langle T_n, \sigma_n\rangle$ and $\langle T_{n+1}, \sigma_{n+1}\rangle$. Then

$$\omega, \sigma_{n+1} \models_{\mathcal{G}} GI \ .$$

Proof (of Lemma 26). We distinguish according to the computation steps in $\langle T_n, \sigma_n\rangle \longrightarrow^* \langle T_{n+1}, \sigma_{n+1}\rangle$.

Case: ASS
Assume that $\langle T_n, \sigma_n\rangle \longrightarrow \langle T_{n+1}, \sigma_{n+1}\rangle$ consists of the execution of a single assignment outside bracketed sections. The case follows by assumption and the restrictions on the global invariant (cf. Definition 1), which assure that GI is preserved under the execution of assignments outside bracketed sections.

Case: CALL, START
We are given $\langle T_n, \sigma_n\rangle \longrightarrow^3 \langle T_{n+1}, \sigma_{n+1}\rangle$, where the first step is a method call (rule CALL) and the following two steps correspond to the observations in the bracketed sections of the caller and the callee. The required property follows from the assumptions and soundness of Equation (6) of the cooperation test for method calls (Lemma 19). The case for CALL is analogous.

Case: START$_{skip}$
Analogously to the previous case CALL, using Lemma 20 instead.

Case: RETURN
We are given the sequence $\langle T_n, \sigma_n\rangle \longrightarrow^3 \langle T_{n+1}, \sigma_{n+1}\rangle$ consisting of the communication of a return value by rule RETURN followed by the assignments in

the bracketed sections of callee and caller. The required property follows from the assumptions and by soundness of the cooperation test for returning from a method (Lemma 21).

Case: TERMINATE
For termination, we are given $\langle T_n, \sigma_n\rangle \longrightarrow^2 \langle T_{n+1}, \sigma_{n+1}\rangle$, caused by the return-statement of a start-method or of the initial invocation of the main-method by rule TERMINATE, followed by the accompanying assignment in the bracketed section of the return-statement. The required property follows from the assumptions and by soundness of the cooperation test for termination (Lemma 22).

Case: NEW
We are given $\langle T_n, \sigma_n\rangle \longrightarrow^2 \langle T_{n+1}, \sigma_{n+1}\rangle$ consisting of an object creation step by rule NEW followed by the execution of the assignment in the bracketed section of new. The required property follows from the assumptions and soundness of the cooperation test for instantiation (Lemma 23). □

Proof (of the soundness Theorem 1). We proceed by induction on the length of the computation, simultaneously for all parts of Definition 7 of $\models$, where in part (2) for the global invariant we consider stable configurations, only.

The base cases of parts (1) and (2) are implied by Lemma 16. For part (3), condition (4) of the local correctness ensures $\omega, \sigma_{inst}^{init}, \tau \models_{\mathcal{L}} pre(body_{\mathsf{main}}) \rightarrow I$ where I is the class invariant of the initial object α. The class invariant contains only instance variables, i.e., its evaluation does not depend on the local state. Furthermore, α is the only existing object in σ, thus part (3) is satisfied initially.

For the inductive step in parts (1) and (3) we are given $\langle T_0, \sigma_0\rangle \longrightarrow^+ \langle T, \sigma\rangle$ and the result follow directly by induction from Lemma 24 respectively from Lemma 25. Part (2) follows by induction and Lemma 26 applied to the last stable configuration preceding $\langle T, \sigma\rangle$ in the computation $\langle T_0', \sigma_0'\rangle \longrightarrow^+ \langle T, \sigma\rangle$. □

B.3 Completeness

The following lemma states that the variable loc indeed stores the current control point of a thread:

Lemma 27. *Let $\langle T, \sigma\rangle$ be a reachable configuration of $prog_0$ and let $(\tau, stm) \in T$ such that the control point before the statement stm is an interleaving point. Then $\tau(\mathsf{loc}) \equiv stm$.*

Proof (of Lemma 27). Straightforward by the definition of augmentation. □

Lemma 28 (Initial correctness). *The proof outline $prog'$ satisfies the initial conditions of Definition 2.*

Proof (of Lemma 28). We show that the proof outline $prog'$ satisfies the initial conditions of Definition 2. Let $\omega \in \Omega$, $\sigma_{inst} \in \Sigma_{inst}$, and $\tau \in \Sigma_{\mathsf{loc}}$ with $\tau(\mathsf{this}) = \alpha$. For the precondition of the main-method we have to show

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(body_{\mathsf{main}})[(\mathsf{this}, 0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}] ,$$

where $\vec{y}$ are the local and instance variables occurring in $pre(body_{\mathsf{main}})$. We start transforming the right-hand side using Lemma 1:

$$\begin{aligned} & [\![pre(body_{\mathsf{main}})[(\mathsf{this}, 0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} \\ = & [\![pre(body_{\mathsf{main}})[(\mathsf{this}, 0)/\mathsf{id}]]\!]_{\mathcal{L}}^{\omega,\sigma_{inst}[\vec{y} \mapsto InitVal(\vec{y})],\tau[\vec{y} \mapsto InitVal(\vec{y})]} \\ = & [\![pre(body_{\mathsf{main}})]\!]_{\mathcal{L}}^{\omega,\sigma_{inst}[\vec{y} \mapsto InitVal(\vec{y})],\tau[\vec{y} \mapsto InitVal(\vec{y})][\mathsf{id} \mapsto (\alpha,0)]} \\ = & [\![pre(body_{\mathsf{main}})]\!]_{\mathcal{L}}^{\omega,\sigma_{inst}^{init},\tau_{init}[\mathsf{this} \mapsto \alpha][\mathsf{id} \mapsto (\alpha,0)]} . \end{aligned}$$

The assertion $pre(body_{\mathsf{main}})$ is satisfied by the logical environment ω and the instance local state $(\sigma_{inst}^{init}, \tau_{init}[\mathsf{this} \mapsto \alpha][\mathsf{id} \mapsto (\alpha, 0)])$ iff there exists a reachable configuration $\langle T, \sigma \rangle$ of $prog'$ with $(\tau_{init}[\mathsf{this} \mapsto \alpha][\mathsf{id} \mapsto (\alpha, 0)], body_{\mathsf{main}}) \in T$ and $\sigma(\alpha) = \sigma_{inst}^{init}$. The initial configuration satisfies these conditions.

For the global invariant we need to show that

$$\omega, \sigma \models_{\mathcal{G}} \mathsf{InitState}(z) \wedge \forall z'(z'{=}\mathsf{nil} \vee z{=}z') \rightarrow GI[\vec{E}_2/z.\vec{y}_2]$$

for arbitrary $\sigma \in \Sigma$ and $\omega \in \Omega$ referring only to values existing in σ, where $\langle \vec{y}_2 := \vec{e}_2 \rangle$ is the bracketed section at the beginning of the main-method, $\vec{E}_2 = \vec{e}_2[(\mathsf{this}, 0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}][z/\mathsf{this}]$, z is of the type of the main class, and $z' \in LVar^{\mathsf{Object}}$. We observe that

$$\omega, \sigma \models_{\mathcal{G}} \mathsf{InitState}(z) \wedge \forall z'(z' = \mathsf{nil} \vee z' = z)$$

implies that σ is the unique initial global state defining exactly one existing object $\omega(z) = \alpha$ in its initial instance state $\sigma(\alpha) = \sigma_{inst}^{init}$.

For the global expression $\vec{E}_2$ we get using the substitution Lemmas 3 and Lemma 1, together with the fact that $\vec{e}_2$ does not contain logical variables, that

$$\begin{aligned} & [\![\vec{E}_2]\!]_{\mathcal{G}}^{\omega,\sigma} & = \\ & [\![\vec{e}_2[(\mathsf{this}, 0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}][z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = \\ & [\![\vec{e}_2[(\mathsf{this}, 0)/\mathsf{id}][\mathsf{InitVal}(\vec{y})/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\sigma(\alpha),\tau[\mathsf{this} \mapsto \alpha]} & = \\ & [\![\vec{e}_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst}^{init},\tau_{init}[\mathsf{this} \mapsto \alpha][\mathsf{id} \mapsto (\alpha,0)]} & = \\ & [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_{inst}^{init},\tau_{init}[\mathsf{this} \mapsto \alpha][\mathsf{id} \mapsto (\alpha,0)]} & . \end{aligned}$$

Thus for the global invariant

$$\begin{aligned} & [\![GI[\vec{E}_2/z.\vec{y}_2]]\!]_{\mathcal{G}}^{\omega,\sigma} & = \\ & [\![GI]\!]_{\mathcal{G}}^{\omega,\sigma[\alpha.\vec{y}_2 \mapsto [\![\vec{E}_2]\!]_{\mathcal{G}}^{\omega,\sigma}]} & = \\ & [\![GI]\!]_{\mathcal{G}}^{\omega,\sigma[\alpha.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_{inst}^{init},\tau_{init}[\mathsf{this} \mapsto \alpha][\mathsf{id} \mapsto (\alpha,0)]}]} & . \end{aligned}$$

Starting from the initial one, the configuration $\langle T_1, \sigma_1\rangle$ after executing the bracketed section at the beginning of the main-method has as state component $\sigma_1 = \sigma[\alpha.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_{inst}^{init}, \tau_{init}[\mathsf{this} \mapsto \alpha][\mathsf{id} \mapsto (\alpha,0)]}]$, i.e., the initial correctness condition for the global invariant equivalently reads $\omega, \sigma_1 \models_{\mathcal{G}} GI$. Since $\langle T_1, \sigma_1\rangle$ is stable and reachable, it is satisfied. □

Lemma 29 (Local correctness). *The proof outline prog′ satisfies the locally correctness conditions from Definition 3.*

Proof (of Lemma 29). Let c be a class of $prog'$ with class invariant I_c, $\vec{y} := \vec{e}$ a multiple assignment, and p an assertion in class c. Let furthermore $\omega \in \Omega$, $\sigma_{inst} \in \Sigma_{inst}$ and $\tau \in \Sigma_{\mathsf{loc}}$. We have to show the local correctness conditions

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e}) \wedge \mathsf{enabled}(\vec{y} := \vec{e}) \rightarrow post(\vec{y} := \vec{e})[\vec{e}/\vec{y}] \text{ and}$$
$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p \rightarrow I_c \ .$$

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$ it follows that there is a reachable $\langle T, \sigma\rangle$ containing $(\tau, \vec{y} := \vec{e}; stm) \in T$ or $(\tau, \langle\vec{y} := \vec{e}\rangle; stm) \in T$, and where $\sigma(\tau(\mathsf{this})) = \sigma_{inst}$. Furthermore, $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$ implies by Lemma 7 that the local configuration is enabled in some reachable $\langle T', \sigma'\rangle$ with $\sigma'(\tau(\mathsf{this})) = \sigma_{inst}$. Executing the local configuration in $\langle T', \sigma'\rangle$ leads to a reachable global configuration $\langle T'', \sigma''\rangle$ with $\sigma''(\tau(\mathsf{this})) = \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}]$. Furthermore, if stm is not the empty statement, i.e., $\vec{y} := \vec{e}$ is not the observation in the bracketed section of a return statement, then $(\tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], stm) \in T''$. If otherwise $stm = \epsilon$, then $post(stm)$ is the class invariant. Thus by the definition of the annotation for $prog'$ we have

$$\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}] \models_{\mathcal{L}} post(\vec{y} := \vec{e}) \ ,$$

and further with the substitution Lemma 1

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} post(\vec{y} := \vec{e})[\vec{e}/\vec{y}] \ ,$$

as required.

For the local correctness condition of the class invariant, let $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$. If p is the postcondition of a method body, then it is the class invariant itself. Otherwise, let p be the precondition of the non-empty statement stm. Then by definition of the annotation there exists a reachable $\langle T, \sigma\rangle$ such that $(\tau, stm) \in T$ and $\sigma(\tau(\mathsf{this})) = \sigma_{inst}$, and with $\sigma(\tau(\mathsf{this})) = \sigma_{inst}$ immediately $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} I_c$, as required. □

The local merging Lemma 10 states that the instance history variables contain enough information, such that individual reachability at two instance local states with a common value for the the history variable implies, that the local configuration is commonly reachable. A key intuition for this property is that the information stored in the instance history suffices to uniquely determine the set of local configurations currently executing within the given instance. The instance variable h_{inst} contains the history for all local configurations shuffled into one sequence, but as each method body in execution is characterized uniquely by its value of id stored as well in the history, one can read-off the set of currently executing local configuration by looking at the *last* (τ, stm) per id and using the value of loc to determine the statement corresponding the current control point. Remember in this context that the value of loc identifies the statement to be executed next. This leads to the definition of *LocConf*, which assigns to each sequence of instance local states a set of local configurations as follows:

$$\begin{aligned} LocConf(\epsilon) &= \emptyset \\ LocConf(h \circ (\sigma_{inst}, \tau)) &= \{(\tau', stm') \in LocConf(h) \mid \tau'(\mathsf{id}) \neq \tau(\mathsf{id})\} \cup \\ &\quad\ \{(\tau, stm) \mid \tau(\mathsf{loc}) \equiv stm \wedge (stm \neq \epsilon \vee \tau(\mathsf{id}) = (\tau(\mathsf{this}), 0))\}. \end{aligned}$$

That this definition, given the value of h_{inst} in an instance, captures all local configurations currently active in the instance, is stated in Lemma 30 below. In the Lemma, the reachable $\langle T, \sigma\rangle$ must be *stable in* α for the same reason, as this is needed in the local merging lemma: Stability is required for the history variable to be up-to date.

Lemma 30. *Let $\langle T, \sigma\rangle$ be a reachable configuration of $prog'$, and $\alpha \in dom(\sigma)$ such that $\langle T, \sigma\rangle$ is stable in α. Then*

$$\{(\tau, stm) \in T \mid \tau(\mathsf{this}) = \alpha\} = LocConf(\sigma(\alpha)(\mathsf{h}_{inst}))\ .$$

Proof (of Lemma 30). Let $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$ be a computation of $prog'$ and $\alpha \in dom(\sigma)$ such that $\langle T, \sigma\rangle$ is stable in α. We show by induction on the length of the computation that $\{(\tau, stm) \in T \mid \tau(\mathsf{this}) = \alpha\}$ equals $LocConf(\sigma(\alpha)(\mathsf{h}_{inst}))$.

If $\langle T, \sigma\rangle$ is the first configuration in the computation stable in α, then either α is a freshly created object in its initial instance state such that no local configurations are executing in α, or α is the initial object, and $\langle T, \sigma\rangle$ results from the initial configuration $\langle T_0, \sigma_0\rangle$ with $T_0 = \{(\tau, \langle \vec{y}_2 := \vec{e}_2\rangle; stm)\}$ by executing the bracketed section at the beginning of the main-method. The first case is straightforward. In the second case $T = \{(\tau', stm)\}$ with $\tau' = \tau[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma_0(\alpha),\tau}]$, and $\sigma(\alpha)(\mathsf{h}_{inst}) = [\![(\vec{x}, \vec{u})[\vec{e}_2/\vec{y}_2]]\!]_{\mathcal{E}}^{\sigma_0(\alpha),\tau} = [\![(\vec{x}, \vec{u})]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau'} = (\sigma(\alpha), \tau')$. By Lemma 27 we get $\tau'(\mathsf{loc}) \equiv stm$, and since stm contains at least a return statement, $stm \neq \epsilon$. Thus we get $LocConf((\sigma(\alpha), \tau')) = \{(\tau', stm)\} = \{(\tau, stm) \in T \mid \tau(\mathsf{this}) = \alpha\}$.

Let now $\langle T', \sigma'\rangle$ be the last configuration preceding $\langle T, \sigma\rangle$ in the computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$ that is stable in α, and let $\sigma(\alpha)(\mathsf{h}_{inst}) = \sigma'(\alpha)(\mathsf{h}_{inst}) \circ h$.

Assume first $(\tau, stm) \in T$ with $\tau(\mathsf{this}) = \alpha$. If $(\tau, stm) \in T'$, then by induction $(\tau, stm) \in LocConf(\sigma'(\alpha)(\mathsf{h}_{inst}))$. Furthermore, $(\tau, stm) \in T'$ and $(\tau, stm) \in T$ together imply that (τ, stm) does not execute in $\langle T', \sigma'\rangle \longrightarrow^* \langle T, \sigma\rangle$, i.e., for all (σ_{inst}, τ') in h, $\tau'(\mathsf{id}) \neq \tau(\mathsf{id})$. By definition of *LocConf* we get $(\tau, stm) \in LocConf(\sigma(\alpha)(\mathsf{h}_{inst}))$.

If otherwise $(\tau, stm) \notin T'$, then (τ, stm) is executed in $\langle T', \sigma'\rangle \longrightarrow^* \langle T, \sigma\rangle$. Since $\langle T, \sigma\rangle$ is stable in α, (τ, stm) also observes its execution in $\langle T', \sigma'\rangle \longrightarrow^*$

$\langle T, \sigma\rangle$, i.e., it results from the execution of a multiple assignment, possibly preceded by some communication or object creation. This assignment updates also the history h_{inst}, i.e., there exists an instance state σ_{inst} such that (σ_{inst}, τ) is contained in h. Furthermore, since $\langle T', \sigma'\rangle$ is the last configuration preceding $\langle T, \sigma\rangle$ in the computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$, we know that (σ_{inst}, τ) is the only element in h with identity $\tau(\mathsf{id})$. By the definition of *LocConf* we get that $(\tau, stm) \in LocConf(\sigma(\alpha)(\mathsf{h}_{inst}))$. Note that since $(\tau, stm) \in T$, either $stm \neq \epsilon$ or (τ, stm) represents the terminated initial execution of the main-method or that of a start-method, i.e., $\tau(\mathsf{id}) = (\alpha, 0)$. Furthermore, since $\langle T, \sigma\rangle$ is stable in α, stm represents an interleaving point, and by Lemma 27 $\tau(\mathsf{loc}) \equiv stm$.

For the reverse direction, let $(\tau, stm) \in LocConf(\sigma(\alpha)(\mathsf{h}_{inst}))$. If there is no instance state σ_{inst} such that (σ_{inst}, τ) is in h, then (τ, stm) is not executed in $\langle T', \sigma'\rangle \longrightarrow^* \langle T, \sigma\rangle$, because otherwise it would have updated the history h_{inst}. Consequently $\tau'(\mathsf{id}) \neq \tau(\mathsf{id})$ for all $(\sigma'_{inst}, \tau') \in h$. By definition of *LocConf* we have $(\tau, stm) \in LocConf(\sigma'(\alpha)(\mathsf{h}_{inst}))$, and thus by induction $(\tau, stm) \in T'$. Since (τ, stm) is not executed in $\langle T', \sigma'\rangle \longrightarrow \langle T, \sigma\rangle$, also $(\tau, stm) \in T$.

Otherwise, if (σ_{inst}, τ) is in h for some instance state σ_{inst}, then (τ, stm) results from the execution of an assignment during $\langle T', \sigma'\rangle \longrightarrow^* \langle T, \sigma\rangle$ that updates the history. Note that in $\langle T', \sigma'\rangle \longrightarrow^* \langle T, \sigma\rangle$ at most one local configuration with the identity $\tau(\mathsf{id})$ executes an assignment, i.e., (σ'_{inst}, τ) is the only element in h with identity $\tau(\mathsf{id})$. Furthermore, $(\tau, stm) \in LocConf(\sigma(\alpha)(\mathsf{h}_{inst}))$ implies by definition of *LocConf* that $stm \neq \epsilon \vee \tau(\mathsf{id}) = (\tau(\mathsf{this}), 0)$, and thus $(\tau, stm) \in T$. $\square$

Proof (of the local merging Lemma 10). Let $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$ be two reachable global configurations of $prog'$ and $(\tau, stm) \in T_1$, such that both $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$ are stable in $\tau(\mathsf{this}) \in dom(\sigma_1) \cap dom(\sigma_2)$. Assume furthermore $\sigma_1(\tau(\mathsf{this}))(\mathsf{h}_{inst}) = \sigma_2(\tau(\mathsf{this}))(\mathsf{h}_{inst})$. Then by Lemma 30

$$\{(\tau', stm') \in T_1 \mid \tau'(\mathsf{this}) = \tau(\mathsf{this})\} = LocConf(\sigma_1(\tau(\mathsf{this}))(\mathsf{h}_{inst})) ,$$
$$\{(\tau', stm') \in T_2 \mid \tau'(\mathsf{this}) = \tau(\mathsf{this})\} = LocConf(\sigma_2(\tau(\mathsf{this}))(\mathsf{h}_{inst})) .$$

With $\sigma_1(\tau(\mathsf{this}))(\mathsf{h}_{inst}) = \sigma_2(\tau(\mathsf{this}))(\mathsf{h}_{inst})$ and $(\tau, stm) \in T_1$, we get $(\tau, stm) \in T_2$. $\square$

The next lemma roughly states that when an assignment can interleave with the precondition of some statement (as given by the predicate $\mathsf{interleavable}$ from page 26) then the assignment and the statement occur in the same reachable global configuration. The lemma is a application of the local merging lemma and will be helpful in the completeness of the interference freedom test.

Lemma 31. *Let $\langle T_1, \sigma_1\rangle$ be a reachable global configuration of $prog'$ and let $(\tau_1, stm; stm_1) \in T_1$. Let furthermore stm_{ass} be an assignment $\vec{y} := \vec{e}$ or $\langle \vec{y} := \vec{e}\rangle$, and let $(\tau_2, stm_{ass}; stm_2)$ with $\tau_2(\mathsf{this}) = \tau_1(\mathsf{this}) = \alpha$ be enabled in some reachable configuration $\langle T_2, \sigma_2\rangle$ with $\sigma_2(\alpha) = \sigma_1(\alpha) = \sigma_{inst}$. Assume further a* *local state τ with $\tau(u) = \tau_2(u)$ for all $u \in dom(\tau_2)$ and $\tau(u') = \tau_1(u)$ for all $u \in dom(\tau_1)$, where u' are fresh variables. Then*

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$$

implies that $(\tau_1, stm; stm_1) \in T_2$.

Proof. From the assumptions stated in the lemma together with $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}}$ $\mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$ we show that $(\tau_1, stm; stm_1) \in T_2$.

Let $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ and $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle T_2, \sigma_2 \rangle$. We start observing that both computations contain at least one stable configuration. So see this note that the only computation without any stable configuration is the empty one, which contains just the initial configuration and in which only one local configuration exists. This means, $\sigma_1(\alpha) = \sigma_2(\alpha) = \sigma_{inst}^{init}$ implies $\tau_1(\mathsf{id}) = \tau_2(\mathsf{id})$, contradicting $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$.

So let $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ and $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle T_2', \sigma_2' \rangle \longrightarrow^* \langle T_2, \sigma_2 \rangle$ such that $\langle T_1', \sigma_1' \rangle$ and $\langle T_2', \sigma_2' \rangle$ are the last configurations in the computations that are stable in α.

We distinguish according to the steps in $\langle T_2', \sigma_2' \rangle \longrightarrow^* \langle T_2, \sigma_2 \rangle$, starting with the case where the sequence is empty.

Case: $\langle T_2', \sigma_2' \rangle = \langle T_2, \sigma_2 \rangle$
In this case $\langle T_2, \sigma_2 \rangle$ is stable in α. Therefore $\vec{y} := \vec{e}$ is enabled in $\langle T_2, \sigma_2 \rangle$ and thus the assignment occurs outside bracketed sections. Hence $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}}$ $\mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$ implies that $pre(stm)$ represents an interleaving point and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$.

For the computation $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ we know that either

1. it does not execute an assignment in α, and thus $\sigma_1'(\alpha) = \sigma_1(\alpha) = \sigma_2(\alpha)$, or
2. it executes a self-communication together with the observation of the sender, but not yet that of the receiver part, otherwise $\langle T_1, \sigma_1 \rangle$ itself would be stable in α, i.e., $\langle T_1', \sigma_1' \rangle = \langle T_1, \sigma_1 \rangle$, and the first clause would apply.

In the second case the definition of augmentation gives $\sigma_1(\alpha)(\mathsf{stable}) = \mathit{false}$, but $\omega, \sigma_{inst}, \tau_2 \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$ implies that $\sigma_2(\alpha)(\mathsf{stable}) = \mathit{true}$. Thus $\sigma_1(\alpha) = \sigma_2(\alpha)$ leads to a contradiction, and only the first case is possible.

That the computation $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ executes no assignment in α means that all local configurations with self-reference α involved in the computation represent non-interleaving points. Since $(\tau_1, stm; stm_1) \in T_1$, the statement stm represents an interleaving point, and therefore $(\tau_1, stm; stm_1)$ is already contained in T_1'. Using the local merging Lemma 10 with $\sigma_1'(\alpha) = \sigma_2(\alpha)$ we get $(\tau_1, stm; stm_1) \in T_2$, as required.

Case: NEW
Assume next that $\langle T_2', \sigma_2' \rangle \longrightarrow \langle T_2, \sigma_2 \rangle$ executes an object creation statement. Then, since the assignment $\vec{y} := \vec{e}$ is enabled in $\langle T_2, \sigma_2 \rangle$, the local configuration $(\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)$ represents the executing thread:

$$\begin{array}{l} \langle T \mathbin{\dot\cup} \{\xi \circ (\tau_2', \langle u := \mathsf{new}^c; \vec{y} := \vec{e} \rangle; stm_2)\}, \sigma_2' \rangle \longrightarrow \\ \langle T \mathbin{\dot\cup} \{\xi \circ (\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)\}, \sigma_2 \rangle \ . \end{array}$$

As in the above case, $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$ implies that $pre(stm)$ represents an interleaving point and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$. Furthermore, corresponding to the above case again, $\sigma_2(\alpha)(\mathsf{stable}) = true$ implies that the computation $\langle T'_1, \sigma'_1 \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ does not execute any assignment is α, and since stm represents an interleaving point, thus $(\tau_1, stm; stm_1) \in T'_1$. Using the local merging Lemma 10 with $\sigma'_1(\alpha) = \sigma'_2(\alpha)$ we get $(\tau_1, stm; stm_1) \in T'_2$. Finally, from $\tau_2(\mathsf{id}) \neq \tau_1(\mathsf{id})$ we conclude that $(\tau_1, stm; stm_1) \in T_2$.

Case: START_{skip}
In this case, $\langle T'_2, \sigma'_2 \rangle \longrightarrow \langle T_2, \sigma_2 \rangle$ tries to invoke the start-method of an object whose thread is already started. By Lemma 6, $\sigma'_2(\alpha)(\mathsf{started}) = true$. Since the assignment $\vec{y} := \vec{e}$ is enabled in $\langle T_2, \sigma_2 \rangle$, the local configuration $(\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)$ represents the executing thread:

$$\begin{aligned}&\langle T \,\dot{\cup}\, \{\xi \circ (\tau_2, \langle e_0.\mathsf{start}(\vec{e}'); \vec{y} := \vec{e} \rangle; stm_2)\}, \sigma'_2 \rangle \longrightarrow \\ &\langle T \,\dot{\cup}\, \{\xi \circ (\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)\}, \sigma_2 \rangle \ .\end{aligned}$$

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$ we conclude that either the assertion $pre(stm)$ is at an interleaving point and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$, or e_0 evaluates to α and stm_1 is the body of the start-method of α. But in the second case the precondition of the body of the start-method implies $\sigma_1(\alpha)(\mathsf{started}) = false$, which contradicts to $\sigma_1(\alpha) = \sigma_2(\alpha)$ and $\sigma'_2(\alpha)(\mathsf{started}) = true$.

By similar arguments as in the previous cases, $\sigma_2(\alpha)(\mathsf{stable}) = true$ implies that the computation $\langle T'_1, \sigma'_1 \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ does not execute any assignment is α, and since stm represents an interleaving point, thus $(\tau_1, stm; stm_1) \in T'_1$. Using the merging Lemma 10 with $\sigma'_1(\alpha) = \sigma'_2(\alpha)$, we get $(\tau_1, stm; stm_1) \in T'_2$. Finally, from $\tau_2(\mathsf{id}) \neq \tau_1(\mathsf{id})$ we conclude that $(\tau_1, stm; stm_1) \in T_2$, as required.

Case: TERMINATE
In this case $\langle T'_2, \sigma'_2 \rangle \longrightarrow \langle T_2, \sigma_2 \rangle$ corresponds to the termination of a start-method or of the initial execution of the main-method. Then, since the assignment $\vec{y} := \vec{e}$ is enabled in $\langle T_2, \sigma_2 \rangle$, the local configuration $(\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)$ represents the executing thread:

$$\begin{aligned}&\langle T \,\dot{\cup}\, \{(\tau_2, \langle \mathsf{return}; \vec{y} := \vec{e} \rangle; stm_2)\}, \sigma'_2 \rangle \longrightarrow \\ &\langle T \,\dot{\cup}\, \{(\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)\}, \sigma_2 \rangle \ .\end{aligned}$$

The assumption $\omega, \sigma_{inst}, \tau_2 \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$ implies that either $pre(stm)$ represents an interleaving point and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$. Note that $\tau_2(\mathsf{id}) = (\alpha, 0)$, and thus $callee(\tau_1(\mathsf{id}))$ cannot be equal to $\tau_2(\mathsf{id})$.

As in the above cases, $\sigma_2(\alpha)(\mathsf{stable}) = true$ implies that $\langle T'_1, \sigma'_1 \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ does not execute an assignment in α, and thus $(\tau_1, stm; stm_1) \in T'_1$. Using Lemma 10 with $\sigma'_1(\alpha) = \sigma'_2(\alpha)$ we get $(\tau_1, stm; stm_1) \in T'_2$. Finally, from $\tau_2(\mathsf{id}) \neq \tau_1(\mathsf{id})$ we conclude that $(\tau_1, stm; stm_1) \in T_2$.

Case: CALL
In this case, $\langle T'_2, \sigma'_2 \rangle \longrightarrow \langle T_2, \sigma_2 \rangle$ executes a method invocation statement and since $\vec{y} := \vec{e}$ is enabled in $\langle T_2, \sigma_2 \rangle$, the local configuration $(\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)$

represents the caller thread:

$$\begin{array}{ll}\langle T \,\dot{\cup}\, \{\xi \circ (\tau_2', \langle e_0.m(\vec{e}); \vec{y} := \vec{e}\rangle; stm_2)\}, \sigma_2'\rangle & \longrightarrow \\ \langle T \,\dot{\cup}\, \{\xi \circ (\tau_2, \langle \vec{y} := \vec{e}\rangle; stm_2) \circ (\tau_{\mathsf{callee}}, body_{m,c})\}, \sigma_2\rangle\ . & \end{array}$$

The assumption $\omega, \sigma_{inst}, \tau_2 \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$ implies that either $pre(stm)$ represents an interleaving point and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$, or stm_1 is the body of the invoked method m, $\tau_1(\mathsf{id}) = callee(\tau_2(\mathsf{id}))$, and $[\![e_0]\!]_{\mathcal{E}}^{\sigma_2'(\alpha),\tau_2'} = \alpha$.

The case where stm represents an interleaving point is analogous to the above cases: $\sigma_2(\alpha)(\mathsf{stable}) = true$ implies that $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ does not execute any assignment in α, and thus $(\tau_1, stm; stm_1) \in T_1'$. Using Lemma 10 with $\sigma_1'(\alpha) = \sigma_2'(\alpha)$ we get $(\tau_1, stm; stm_1) \in T_2'$. Finally, from $\tau_2(\mathsf{id}) \neq \tau_1(\mathsf{id})$ we conclude that $(\tau_1, stm; stm_1) \in T_2$.

Assume now that stm_1 is the body of the invoked method m, $\tau_1(\mathsf{id}) = callee(\tau_2(\mathsf{id}))$, and $[\![e_0]\!]_{\mathcal{E}}^{\sigma_2'(\alpha),\tau_2'} = \alpha$. Then also $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ contains the invocation of the same method of α with resulting callee configuration $(\tau_1, stm; stm_1)$. Furthermore, $\sigma_2(\alpha)(\mathsf{stable}) = true = \sigma_1(\alpha)(\mathsf{stable})$, i.e., $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ does not execute any assignment in α. Let $(\tau_{\mathsf{caller}}, stm_{\mathsf{caller}})$ be the local configuration of the caller in T_1'. Then using $\sigma_1'(\alpha) = \sigma_1(\alpha) = \sigma_2(\alpha) = \sigma_2'(\alpha)$ and the local merging Lemma 10 we get $(\tau_{\mathsf{caller}}, stm_{\mathsf{caller}}) \in T_2'$. With $\tau_1(\mathsf{id}) = callee(\tau_{\mathsf{caller}}(\mathsf{id}))$ and $\tau_1(\mathsf{id}) = callee(\tau_2(\mathsf{id}))$ we further get $\tau_{\mathsf{caller}}(\mathsf{id}) = \tau_2(\mathsf{id})$, i.e., $(\tau_{\mathsf{caller}}, stm_{\mathsf{caller}}) = (\tau_2', \langle e_0.m(\vec{e}); \vec{y} := \vec{e}\rangle; stm_2)$. Thus $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ and $\langle T_2', \sigma_2'\rangle \longrightarrow \langle T_2, \sigma_2\rangle$ execute the same method invocation creating the same callee configuration $(\tau_1, stm; stm_1) = (\tau_{\mathsf{callee}}, body_{m,c}) \in T_2$.

Case: CALL+ASS

Assume next that $\langle T_2', \sigma_2'\rangle \longrightarrow^2 \langle T_2, \sigma_2\rangle$ executes a method invocation statement and the observation of the caller. Then, since $\vec{y} := \vec{e}$ is enabled in $\langle T_2, \sigma_2\rangle$, the local configuration $(\tau_2, \langle \vec{y} := \vec{e}\rangle; stm_2)$ represents the callee thread before its observation:

$$\begin{array}{ll}\langle T \,\dot{\cup}\, \{\xi \circ (\tau_{\mathsf{caller}}', \langle e_0.m(\vec{e}); \vec{y}' := \vec{e}'\rangle; stm_{\mathsf{caller}})\}, \sigma_2'\rangle & \longrightarrow \\ \langle T \,\dot{\cup}\, \{\xi \circ (\tau_{\mathsf{caller}}', \langle \vec{y}' := \vec{e}'\rangle; stm_{\mathsf{caller}}) \circ (\tau_2, \langle \vec{y} := \vec{e}\rangle; stm_2)\}, \sigma_2'\rangle & \longrightarrow \\ \langle T \,\dot{\cup}\, \{\xi \circ (\tau_{\mathsf{caller}}, stm_{\mathsf{caller}}) \circ (\tau_2, \langle \vec{y} := \vec{e}\rangle; stm_2)\}, \sigma_2\rangle\ . & \end{array}$$

Then $\omega, \sigma_{inst}, \tau_2 \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$ implies that $pre(stm)$ represents an interleaving point and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$.

If $\sigma_2(\alpha)(\mathsf{stable}) = true$, i.e., the method invocation is not a self-call, then the computation $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ does not execute any assignment in α, and thus $(\tau_1, stm; stm_1) \in T_1'$. Otherwise, $\sigma_1(\alpha)(\mathsf{stable}) = \sigma_2(\alpha)(\mathsf{stable}) = false$ and $\sigma_1(\alpha)(\mathsf{h}_{inst}) = \sigma_2(\alpha)(\mathsf{h}_{inst})$ imply that $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ executes the same assignment in the same local configuration as $\langle T_2', \sigma_2'\rangle \longrightarrow^2 \langle T_2, \sigma_2\rangle$, and thus the same method invocation in the same local configuration with identity $\tau_{\mathsf{caller}}(\mathsf{id})$. Thus $\sigma_1'(\alpha) = \sigma_2'(\alpha)$, and with $\tau_2(\mathsf{id}) = callee(\tau_{\mathsf{caller}}(\mathsf{id}))$ and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$ we conclude that $(\tau_1, stm; stm_1)$ is not the callee configuration, i.e., either it is the caller or $(\tau_1, stm; stm_1) \in T_1'$. In the first case, if $(\tau_1, stm; stm_1)$ is the caller, then $(\tau_1, stm; stm_1) = (\tau_{\mathsf{caller}}, stm_{\mathsf{caller}}) \in T_2$. Otherwise, if $(\tau_1, stm; stm_1) \in T_1'$, then using again Lemma 10 with $\sigma_1'(\alpha) = \sigma_2'(\alpha)$ we get $(\tau_1, stm; stm_1) \in T_2'$. Finally, since $(\tau_1, stm; stm_1)$ is neither the caller nor the callee configuration, we conclude that $(\tau_1, stm; stm_1) \in T_2$.

Case: RETURN
In this case $\langle T_2', \sigma_2' \rangle \longrightarrow \langle T_2, \sigma_2 \rangle$ returns the result of a method. Then, since the assignment $\vec{y} := \vec{e}$ is enabled in $\langle T_2, \sigma_2 \rangle$, the local configuration $(\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)$ represents the callee thread:

$$\langle T \,\dot\cup\, \{\xi \circ (\tau'_{\mathsf{caller}}, \langle \mathsf{receive}\, u; \vec{y}' := \vec{e}' \rangle; stm_{\mathsf{caller}}) \circ (\tau_2, \langle \mathsf{return}\, e_{ret}; \vec{y} := \vec{e} \rangle)\}, \sigma_2' \rangle \longrightarrow$$
$$\langle T \,\dot\cup\, \{\xi \circ (\tau_{\mathsf{caller}}, \langle \vec{y}' := \vec{e}' \rangle; stm_{\mathsf{caller}}) \circ (\tau_2, \langle \vec{y} := \vec{e} \rangle)\}, \sigma_2 \rangle \ .$$

Then $\omega, \sigma_{inst}, \tau_2 \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$ implies that either $pre(stm)$ represents an interleaving point and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$ and $callee(\tau_1(\mathsf{id})) \neq \tau_2(\mathsf{id})$, or stm_1 occurs after a receive statement which is preceded by the invocation of method m of e_0, $callee(\tau_1(\mathsf{id})) = \tau_2(\mathsf{id})$, and $[\![e_0]\!]_{\mathcal{E}}^{\sigma_2'(\alpha), \tau_2} = \alpha$.

The case that stm represents an interleaving point is analogous to the above cases: $\sigma_2(\alpha)(\mathsf{stable}) = true$ implies that $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ does not execute any assignment in α, and thus $(\tau_1, stm; stm_1) \in T_1'$. Using Lemma 10 with $\sigma_1'(\alpha) = \sigma_2'(\alpha)$ we get $(\tau_1, stm; stm_1) \in T_2'$. Finally, from $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$ and $callee(\tau_1(\mathsf{id})) \neq \tau_2(\mathsf{id})$ we conclude that $(\tau_1, stm; stm_1) \in T_2$.

Assume now that stm_1 occurs after a receive statement which is preceded by the invocation of method m of e_0, $callee(\tau_1(\mathsf{id})) = \tau_2(\mathsf{id})$, and $[\![e_0]\!]_{\mathcal{E}}^{\sigma_2'(\alpha), \tau_2} = \alpha$. Then also $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ contains the return from the same method of α with resulting caller configuration $(\tau_1, stm; stm_1)$. Note that $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ does not execute any assignments in α: from $[\![e_0]\!]_{\mathcal{E}}^{\sigma_2'(\alpha), \tau_2} = \alpha$ we conclude that a self-communication is executed, and $\sigma_1(\alpha) = \sigma_2(\alpha)$ and $\sigma_2(\alpha)(\mathsf{stable}) = true$ imply also $\sigma_1(\alpha)(\mathsf{stable}) = true$.

Let $(\tau_{\mathsf{callee}}, stm_{\mathsf{callee}})$ be the local configuration of the callee in T_1'. Then using $\sigma_1'(\alpha) = \sigma_1(\alpha) = \sigma_2(\alpha) = \sigma_2'(\alpha)$ and Lemma 10 we get that $(\tau_{\mathsf{callee}}, stm_{\mathsf{callee}}) \in T_2'$. From $callee(\tau_1(\mathsf{id})) = \tau_{\mathsf{callee}}(\mathsf{id})$ and $callee(\tau_1(\mathsf{id})) = \tau_2(\mathsf{id})$ we conclude $\tau_{\mathsf{callee}}(\mathsf{id}) = \tau_2(\mathsf{id})$, i.e., $(\tau_{\mathsf{callee}}, stm_{\mathsf{callee}}) = (\tau_2, \langle \mathsf{return}\, e_{ret}; \vec{y} := \vec{e} \rangle; stm_2)$. Thus $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ and $\langle T_2', \sigma_2' \rangle \longrightarrow \langle T_2, \sigma_2 \rangle$ execute the same return and receive statements and thus $(\tau_1, stm; stm_1) = (\tau_{\mathsf{caller}}, \langle \vec{y}' := \vec{e}' \rangle; stm_{\mathsf{caller}}) \in T_2$.

Case: RETURN+ASS
In this case, $\langle T_2', \sigma_2' \rangle \longrightarrow^2 \langle T_2, \sigma_2 \rangle$ returns from a method and executes the observation of the callee. Since the assignment $\vec{y} := \vec{e}$ is enabled in $\langle T_2, \sigma_2 \rangle$, the local configuration $(\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)$ represents the caller thread before its observation:

$$\langle T \,\dot\cup\, \{\xi \circ (\tau_2', \langle \mathsf{receive}\, u; \vec{y} := \vec{e} \rangle; stm_2) \circ (\tau_{\mathsf{callee}}, \langle \mathsf{return}\, e_{ret}; \vec{y}' := \vec{e}' \rangle)\}, \sigma_2' \rangle \longrightarrow$$
$$\langle T \,\dot\cup\, \{\xi \circ (\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2) \circ (\tau_{\mathsf{callee}}, \langle \vec{y}' := \vec{e}' \rangle)\}, \sigma_2' \rangle \longrightarrow$$
$$\langle T \,\dot\cup\, \{\xi \circ (\tau_2, \langle \vec{y} := \vec{e} \rangle; stm_2)\}, \sigma_2 \rangle \ .$$

By the assumption $\omega, \sigma_{inst}, \tau_2 \models_{\mathcal{L}} \mathsf{interleavable}(pre(stm), \vec{y} := \vec{e})$, $pre(stm)$ represents an interleaving point and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$.

If $\sigma_2(\alpha)(\mathsf{stable}) = \mathit{true}$, i.e., in the case of self-communication, $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ does not execute any assignment in α, and thus $(\tau_1, \mathit{stm}; \mathit{stm}_1) \in T_1'$. Otherwise, $\sigma_1(\alpha)(\mathsf{stable}) = \sigma_2(\alpha)(\mathsf{stable}) = \mathit{false}$ and $\sigma_1(\alpha)(\mathsf{h}_{inst}) = \sigma_2(\alpha)(\mathsf{h}_{inst})$ imply that $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ executes the same assignment in the same local configuration as $\langle T_2', \sigma_2' \rangle \longrightarrow^2 \langle T_2, \sigma_2 \rangle$, and thus the same return in the same local configuration with identity $\tau_{\mathsf{callee}}(\mathsf{id})$. Thus $\sigma_1'(\alpha) = \sigma_2'(\alpha)$, and with $\mathit{callee}(\tau_2(\mathsf{id})) = \tau_{\mathsf{callee}}(\mathsf{id})$ and $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$ we conclude that $(\tau_1, \mathit{stm}; \mathit{stm}_1)$ is not the caller configuration, i.e., $(\tau_1, \mathit{stm}; \mathit{stm}_1) \in T_1'$. Furthermore, since $(\tau_1, \mathit{stm}; \mathit{stm}_1) \in T_1$, it is also not the callee configuration. Using again the merging Lemma 10 with $\sigma_1'(\alpha)=\sigma_2'(\alpha)$ we get $(\tau_1, \mathit{stm}; \mathit{stm}_1) \in T_2'$. Since finally $(\tau_1, \mathit{stm}; \mathit{stm}_1)$ is neither the caller nor the callee configuration, we conclude that $(\tau_1, \mathit{stm}; \mathit{stm}_1) \in T_2$. □

Lemma 32 (Interference freedom). *The proof outline prog′ satisfies the conditions for interference freedom from Definition 4.*

Proof (of Lemma 32). Assume an arbitrary multiple assignment $\vec{y} := \vec{e}$ in class c and an arbitrary statement stm in the same class. We prove interference freedom for the precondition of the statement stm under the execution of $\vec{y} := \vec{e}$, i.e., we have to show the verification condition from Equation (5) on page 26 for some logical environment ω together with some instance and local states σ_{inst} and τ:

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathit{pre}'(\mathit{stm}) \wedge \mathit{pre}(\vec{y} := \vec{e}) \wedge \mathsf{this}{=}\mathsf{this}' \wedge \\ \mathsf{interleavable}(\mathit{pre}(\mathit{stm}), \vec{y} := \vec{e}) \wedge \mathsf{enabled}(\vec{y} := \vec{e}) \rightarrow \mathit{pre}'(\mathit{stm})[\vec{e}/\vec{y}]\ ,$$

where $\mathit{pre}'(\mathit{stm})$ denotes $\mathit{pre}(\mathit{stm})$ with all local variables u and this replaced by some fresh local variables u' and this', respectively.

If $\mathit{stm} = \epsilon$, then by Definition 9 of the annotation $\mathit{pre}(\mathit{stm})(= \mathit{pre}'(\mathit{stm}))$ is the class invariant, whose invariance under execution is shown in the local correctness. So assume $\mathit{stm} \neq \epsilon$ in the following.

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{this} = \mathsf{this}'$ we get $\tau(\mathsf{this}) = \tau(\mathsf{this}')$, i.e., $\tau_2(\mathsf{this}) = \tau(\mathsf{this})$ for $\tau_2 = \tau$, and $\tau_1(\mathsf{this}) = \tau(\mathsf{this})$, where τ_1 coincides with τ modulo renaming of the local variables, i.e., $\tau_1(u) = \tau(u')$ for all local variables $u' \in \mathit{dom}(\tau)$. Let $\alpha = \tau_1(\mathsf{this}) = \tau_2(\mathsf{this})$.

The first clause $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathit{pre}'(\mathit{stm})$ implies $\omega, \sigma_{inst}, \tau_1 \models_{\mathcal{L}} \mathit{pre}(\mathit{stm})$. Remember that we assume that the annotation does not contain free logical variables, hence the logical environment ω does not play a role. According to Definition 9 of the annotation, $\omega, \sigma_{inst}, \tau_1 \models_{\mathcal{L}} \mathit{pre}(\mathit{stm})$ implies that there exists a reachable configuration $\langle T_1, \sigma_1 \rangle$ and a local configuration $(\tau_1, \mathit{stm}; \mathit{stm}_1) \in T_1$ with $\sigma_1(\alpha) = \sigma_{inst}$. For the assignment $\vec{y} := \vec{e}$ we similarly get using $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathit{pre}(\vec{y} := \vec{e})$ that there exists a computation reaching $\langle \hat{T}_2, \hat{\sigma}_2 \rangle$ with $(\tau_2, \mathit{stm}_{ass}; \mathit{stm}_2) \in \hat{T}_2$ with $\mathit{stm}_{ass} = \vec{y} := \vec{e}$ or $\mathit{stm}_{ass} = \langle \vec{y} := \vec{e} \rangle$ such that $\hat{\sigma}_2(\alpha) = \sigma_{inst}$. It follows that $\sigma_1(\alpha) = \hat{\sigma_2}(\alpha)$. Furthermore $\omega, \sigma_{inst}, \tau_2 \models_{\mathcal{L}} \mathsf{enabled}(\vec{y} := \vec{e})$ implies with Lemma 7 that $(\tau_2, \mathit{stm}_{ass}; \mathit{stm}_2)$ is enabled in some reachable $\langle T_2, \sigma_2 \rangle$ with $\sigma_2(\alpha) = \sigma_{inst} = \sigma_1(\alpha)$.

Using Lemma 31 we get $(\tau_1, \mathit{stm}; \mathit{stm}_1) \in T_2$. Furthermore, $(\tau_2, \mathit{stm}_{ass}; \mathit{stm}_2)$ is enabled in $\langle T_2, \sigma_2 \rangle$, and additionally $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{interleavable}(\mathit{pre}(\mathit{stm}), \vec{y} :=$

$\vec{e}$) implies that $\tau_1(\mathsf{id}) \neq \tau_2(\mathsf{id})$. Thus executing in $\langle T_2, \sigma_2\rangle$ the assignment $\vec{y} := \vec{e}$ in the local configuration $(\tau_2, stm_{ass}; stm_2)$ results in a reachable global configuration $\langle T, \sigma\rangle$ with $(\tau_1, stm; stm_1) \in T$ and $\sigma(\alpha) = \sigma_2(\alpha)[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_2(\alpha),\tau_2}]$. We get $\omega, \sigma_2(\alpha)[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_2(\alpha),\tau_2}], \tau_1 \models_{\mathcal{L}} pre(stm)$, and renaming back the local variables of $pre(stm)$ also $\omega, \sigma_2(\alpha)[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_2(\alpha),\tau}], \tau \models_{\mathcal{L}} pre'(stm)$. Finally, by the substitution Lemma 1 together with $\sigma_2(\alpha) = \sigma_{inst}$ we get the required property $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre'(stm)[\vec{e}/\vec{y}]$. Note that due to renaming, no local variables of $\vec{y}$ occur in $pre'(stm)$, and for the same reason $\tau_1 = \tau_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_2(\alpha),\tau_2}]$. □

Proof (of the global merging Lemma 11). Let $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$ be two reachable stable global configurations of $prog'$ and $\alpha \in dom(\sigma_1) \cap dom(\sigma_2)$ with $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$. We show that there exists a reachable stable configuration $\langle T, \sigma\rangle$ with $\sigma(\alpha) = \sigma_1(\alpha)$, and $\sigma(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2)\backslash\{\alpha\}$. We proceed by induction on the sum of the lengths of the computations.

In the base case of the first reachable stable configurations, we are given $\langle T_1, \sigma_1\rangle = \langle T_2, \sigma_2\rangle$ and the property trivially holds.

For the inductive step, assume $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ and $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ as computations of $prog'$ such that $\langle T_1', \sigma_1'\rangle$ and $\langle T_2', \sigma_2'\rangle$ are the last stable configurations preceding $\langle T_1, \sigma_1\rangle$ respectively $\langle T_2, \sigma_2\rangle$ in the computations. We distinguish whether in the computations from $\langle T_1', \sigma_1'\rangle$ to $\langle T_1, \sigma_1\rangle$ and from $\langle T_1', \sigma_1'\rangle$ to $\langle T_1, \sigma_1\rangle$, the communication histories are updated or not.

Case: $\sigma_1'(\alpha)(\mathsf{h}_{comm}) = \sigma_1(\alpha)(\mathsf{h}_{comm})$
In this case $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ does not execute any communication or object creation involving α. By induction there is a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T', \sigma'\rangle$ leading to a stable configuration such that $\sigma'(\alpha) = \sigma_1'(\alpha)$ and $\sigma'(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2)\backslash\{\alpha\}$.

In case $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ does not execute in α at all, i.e., $\sigma_1'(\alpha) = \sigma_1(\alpha)$, the computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T', \sigma'\rangle$ already satisfies the required properties.

Otherwise, $\langle T_1', \sigma_1'\rangle \longrightarrow \langle T_1, \sigma_1\rangle$ executes an assignment outside bracketed sections in α, say in the local configuration also (τ, stm). For $(\tau, stm) \in T_1'$ we know $\tau(\mathsf{this}) = \alpha$ and $\sigma'(\alpha) = \sigma_1'(\alpha)$, and therefore by the local merging Lemma 10 $(\tau, stm) \in T'$. Assignments outside bracketed sections are enabled in all stable configurations. Thus we can execute (τ, stm) in $\langle T', \sigma'\rangle$, leading to the computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T', \sigma'\rangle \longrightarrow \langle T, \sigma\rangle$ with $\sigma(\alpha) = \sigma_1(\alpha)$ and $\sigma(\beta) = \sigma'(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2)\backslash\{\alpha\}$, as required.

Case: $\sigma_2'(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$
In this case $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ does not execute any communication or object creation involving α. By induction, there is a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T', \sigma'\rangle$ leading to a stable configuration such that $\sigma'(\alpha) = \sigma_1(\alpha)$ and $\sigma'(\beta) = \sigma_2'(\beta)$ for all $\beta \in dom(\sigma_2')\backslash\{\alpha\}$.

If $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ performs a step within α, then according to the case assumption it executes an assignment outside bracketed sections within

α. This means, $\sigma_2'(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2)\backslash\{\alpha\}$, and the computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T', \sigma'\rangle$ already satisfies the required properties.

If otherwise $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ does not execute in α, then all local configurations in T_2', which define a self-reference different from α, are also in T'; this follows from stability of $\langle T', \sigma'\rangle$ and $\langle T_2', \sigma_2'\rangle$, from $\sigma_2'(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2)\backslash\{\alpha\}$, and with the help of the local merging Lemma 10 applied to $\langle T', \sigma'\rangle$ and $\langle T_2', \sigma_2'\rangle$. Furthermore, the enabledness of local configurations, whose execution does not involve α, are independent of the instance state of α. Thus in $\langle T', \sigma'\rangle$ we can execute the same computation steps as in $\langle T_2', \sigma_2'\rangle \longrightarrow \langle T_2, \sigma_2\rangle$, leading to a reachable stable configuration $\langle T, \sigma\rangle$ with the required properties.

Case: $\sigma_1'(\alpha)(\mathsf{h}_{comm}) \neq \sigma_1(\alpha)(\mathsf{h}_{comm})$ and $\sigma_2'(\alpha)(\mathsf{h}_{comm}) \neq \sigma_2(\alpha)(\mathsf{h}_{comm})$
In this case finally both $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ and $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ execute some object creation or communication involving the object α. We show that in this case $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$ implies also $\sigma_1'(\alpha)(\mathsf{h}_{comm}) = \sigma_2'(\alpha)(\mathsf{h}_{comm})$, and thus by induction there is a computation leading to a configuration $\langle T', \sigma'\rangle$ such that $\sigma'(\alpha) = \sigma_1'(\alpha)$ and $\sigma'(\beta) = \sigma_2'(\beta)$ for all other objects $\beta \in dom(\sigma_2')\backslash\{\alpha\}$.

Furthermore, combining those computation steps in $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ which execute within α with those in $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ which execute outside α, we can define a computation $\langle T', \sigma'\rangle \longrightarrow^* \langle T, \sigma\rangle$ resulting in a reachable stable global configuration with $\sigma(\alpha) = \sigma_1(\alpha)$ and $\sigma(\beta) = \sigma_2(\beta)$ for all other objects $\beta \in dom(\sigma_2)\backslash\{\alpha\}$.

We distinguish according to the steps in the computation $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$:

Subcase: NEW
In this case $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ is the execution of an object creation statement and its observation in α of the form

$$\begin{array}{ll}\langle T \,\dot\cup\, \{\xi \circ (\tau', \langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle; stm)\}, \sigma_1'\rangle & \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\tau'', \langle \vec{y} := \vec{e}\rangle; stm)\}, \sigma_1''\rangle & \longrightarrow \\ \langle T \,\dot\cup\, \{\xi \circ (\tau, stm)\}, \sigma_1\rangle & \end{array}$$

with $\tau(\mathsf{this}) = \alpha$. The assignment $\vec{y} := \vec{e}$ contains the communication history update $\mathsf{h}_{comm} := \mathsf{h}_{comm} \circ (\mathsf{kind}, \mathsf{id}, \mathsf{partner}, \mathsf{values})$, resulting in $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_1'(\alpha)(\mathsf{h}_{comm}) \circ ((\mathsf{new}, c), \tau'(\mathsf{id}), nil, \gamma)$, where γ is the newly created object.

Since $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$, the last element of $\sigma_2(\alpha)(\mathsf{h}_{comm})$ contains the same information $((\mathsf{new}, c), \tau'(\mathsf{id}), nil, \gamma)$. With the case assumption $\sigma_2'(\alpha)(\mathsf{h}_{comm}) \neq \sigma_2(\alpha)(\mathsf{h}_{comm})$ we additionally know that the tuple represents information observed in $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$. This means, executing $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ creates the same new object γ and leaves the states of all objects from $dom(\sigma_2')\backslash\{\alpha\}$ untouched. By the definition of the augmentation it means $\sigma_1'(\alpha)(\mathsf{h}_{comm}) = \sigma_2'(\alpha)(\mathsf{h}_{comm})$ and $\sigma_2'(\beta) = \sigma_2(\beta)$ for all objects $\beta \in dom(\sigma_2')$ different from α. Thus by induction there is a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T', \sigma'\rangle$ with $\langle T', \sigma'\rangle$ stable and where $\sigma'(\alpha) = \sigma_1'(\alpha)$ and $\sigma'(\beta) = \sigma_2'(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2')\backslash\{\alpha\}$.

As $\sigma'(\alpha) = \sigma'_1(\alpha)$, the local merging Lemma 10 implies that all local configurations in T'_1 with self-reference α are also contained in T'. From the above observation and the fact that for stable configurations the enabledness of execution exclusively within α is independent from the instance states of other objects, we conclude that the same computation steps as in $\langle T'_1, \sigma'_1\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ can be executed also in $\langle T', \sigma'\rangle$, leading to a reachable stable configuration $\langle T, \sigma\rangle$ with $\sigma(\alpha) = \sigma_1(\alpha)$, and $\sigma(\beta) = \sigma'(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma'_2)\backslash\{\alpha\}$. Finally, for the newly created object we have $\sigma(\gamma) = \sigma_2(\gamma) = \sigma^{init}_{inst}$, and thus $\sigma(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2)\backslash\{\alpha\}$.

The case for object creation without storing the identity of the new object $\langle \mathsf{new}^c; \vec{y} := \vec{e}\rangle$ is similar, where the communication history does not contain information about the identity of the new object. Thus the fact that σ_1 and σ_2 define the same communication history for α does not ensure that the last steps $\langle T'_1, \sigma'_1\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ and $\langle T'_2, \sigma'_2\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ create a new object with the same identity. But in this case the instance states of objects existing prior to the object creation do not depend on the identity of the new object, and we can replace the identity of the new object in σ by the one in σ_2, still getting a valid computation with the required properties.

Subcase: CALL
In this case $\langle T'_1, \sigma'_1\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ executes a method invocation and the corresponding observations:

$$
\begin{array}{ll}
\langle T \dot{\cup} \{\xi \circ (\tau'_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1\rangle; stm_1)\}, \sigma'_1\rangle & \longrightarrow \\
\langle T \dot{\cup} \{\xi \circ (\tau'_1, \langle \vec{y}_1 := \vec{e}_1\rangle; stm_1) \circ (\tau'_2, \langle \vec{y}_2 := \vec{e}_2\rangle; stm_2)\}, \sigma'_1\rangle & \longrightarrow \\
\langle T \dot{\cup} \{\xi \circ (\tau_1, stm_1) \circ (\tau'_2, \langle \vec{y}_2 := \vec{e}_2\rangle; stm_2)\}, \sigma''_1\rangle & \longrightarrow \\
\langle T \dot{\cup} \{\xi \circ (\tau_1, stm_1) \circ (\tau_2, stm_2)\}, \sigma_1\rangle \ . &
\end{array}
$$

Since $\sigma_1(\alpha)(\mathsf{h}_{comm}) \neq \sigma'_1(\alpha)(\mathsf{h}_{comm})$, we know α is involved in the method invocation, as caller or the callee object, or both.

If the caller object is α but the callee is different, then by the semantics of method invocation and by the augmentation definition

$$\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma'_1(\alpha)(\mathsf{h}_{comm}) \circ ((\mathsf{call}, m), \tau'_1(\mathsf{id}), \tau'_2(\mathsf{this}), \tau'_2(\vec{u}))\ ,$$

where $\vec{u}$ are the formal parameters of the invoked method. Correspondingly in case the callee is α and the caller is different, we have

$$\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma'_1(\alpha)(\mathsf{h}_{comm}) \circ ((\mathsf{called}, m), \tau'_2(\mathsf{id}), \tau'_1(\mathsf{this}), \tau'_2(\vec{u}))\ .$$

Finally, in case of a self-call within α we have

$$
\begin{aligned}
\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma'_1(\alpha)(\mathsf{h}_{comm}) &\circ ((\mathsf{call}, m), \tau'_1(\mathsf{id}), \alpha, \tau'_2(\vec{u})) \\
&\circ ((\mathsf{called}, m), \tau'_2(\mathsf{id}), \alpha, \tau'_2(\vec{u}))\ .
\end{aligned}
$$

The assumption $\sigma'_2(\alpha)(\mathsf{h}_{comm}) \neq \sigma_2(\alpha)(\mathsf{h}_{comm})$ implies, that the last element of the communication history of α was appended in the computation $\langle T'_2, \sigma'_2\rangle \longrightarrow^*$

$\langle T_2, \sigma_2\rangle$. Since $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$, the computation $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ invokes the same method of the same callee object by the same caller object and with the same actual parameter list, thereby creating the same new local configuration representing the method execution. Thus $\sigma_1'(\alpha)(\mathsf{h}_{comm}) = \sigma_2'(\alpha)(\mathsf{h}_{comm})$, and by induction there is a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T', \sigma'\rangle$ leading to a stable configuration such that $\sigma'(\alpha) = \sigma_1'(\alpha)$, and $\sigma'(\beta) = \sigma_2'(\beta)$ for all $\beta \in dom(\sigma_2')\backslash\{\alpha\}$.

Let $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ be of the form

$$\begin{array}{ll}
\langle \hat{T} \,\dot\cup\, \{\xi' \circ (\hat\tau_1', \langle e_0'.m(\vec{e}'); \vec{y}_1' := \vec{e}_1'\rangle; stm_1')\}, \sigma_2'\rangle & \longrightarrow \\
\langle \hat{T} \,\dot\cup\, \{\xi' \circ (\hat\tau_1', \langle \vec{y}_1' := \vec{e}_1'\rangle; stm_1') \circ (\tau_2', \langle \vec{y}_2 := \vec{e}_2\rangle; stm_2)\}, \sigma_2'\rangle & \longrightarrow \\
\langle \hat{T} \,\dot\cup\, \{\xi' \circ (\hat\tau_1, stm_1') \circ (\tau_2', \langle \vec{y}_2 := \vec{e}_2\rangle; stm_2)\}, \sigma_2''\rangle & \longrightarrow \\
\langle \hat{T} \,\dot\cup\, \{\xi' \circ (\hat\tau_1, stm_1') \circ (\tau_2, stm_2)\}, \sigma_2\rangle \ . &
\end{array}$$

where $\hat\tau_1'(\mathsf{this}) = \tau_1'(\mathsf{this})$. As $\sigma'(\alpha) = \sigma_1'(\alpha)$, the local merging lemma Lemma 10 implies that all local configurations in T_1' with self-reference α are also contained in T'. Similarly for objects from $\beta \in dom(\sigma_2')\backslash\{\alpha\} = dom(\sigma_2)\backslash\{\alpha\}$, $\sigma'(\beta) = \sigma_2'(\beta)$ implies that all local configurations in T_2' with self-reference β are also contained in T'.

Consequently, if α is the caller object, then the local configuration of the caller in $\langle T_1', \sigma_1'\rangle$ is enabled in $\langle T', \sigma'\rangle$. Note that since $\sigma_2'(\beta) = \sigma'(\beta)$ for all $\beta \in dom(\sigma_2')\backslash\{\alpha\}$ and $\sigma_1'(\alpha) = \sigma'(\alpha)$, and since the invocation of the method m of the callee object is enabled both in $\langle T_1', \sigma_1'\rangle$ and in $\langle T_2', \sigma_2'\rangle$, in the case of a synchronized method the lock of the callee object is free in $\langle T', \sigma'\rangle$. Thus the same computation steps as in $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$ can be executed in $\langle T', \sigma'\rangle$ leading to a reachable stable configuration $\langle T, \sigma\rangle$ with $\sigma(\alpha) = \sigma_1(\alpha)$, and $\sigma(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2)\backslash\{\alpha\}$.

Similarly, if the caller object is not α, then the same computation steps as in $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$ can be executed in $\langle T', \sigma'\rangle$ leading to a reachable stable configuration satisfying the requirements.

The remaining subcases are analogous. For the case of return, the computation $\langle T', \sigma'\rangle \longrightarrow^* \langle T, \sigma\rangle$ is constructed from the execution of those local configurations with self-reference α which execute in $\langle T_1', \sigma_1'\rangle \longrightarrow^* \langle T_1, \sigma_1\rangle$, and the execution of those local configurations with self-reference different from α which execute in $\langle T_2', \sigma_2'\rangle \longrightarrow^* \langle T_2, \sigma_2\rangle$. □

Lemma 33 (Cooperation test: Method call, return, and terminate). *The annotated program prog′ and the global invariant GI satisfy the verification conditions of the cooperation test for communication of Definition 5.*

Proof (of Lemma 33). Let $\langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1\rangle; \langle \mathsf{receive}\ v; \vec{y}_4 := \vec{e}_4\rangle$ be a statement in a class c of $prog'$ with $e_0 \in Exp_c^{c'}$, where method m of c' is synchronized with formal parameter list $\vec{u}$, local variables without the formal parameters and this given by $\vec{v}$, and $body_{m,c'} = \langle \vec{y}_2 := \vec{e}_2\rangle; stm; \langle \mathsf{return}\ e_{ret}; \vec{y}_3 := \vec{e}_3\rangle$.

Case: Method call
Assume

$$\omega, \sigma \models_{\mathcal{G}} GI \wedge pre(e_0.m(\vec{e}))[z/\mathsf{this}] \wedge I_{c'}[z'/\mathsf{this}] \wedge \\ e_0[z/\mathsf{this}]{=}z' \wedge \mathsf{isfree}(z'.\mathsf{lock}, \mathsf{id}) \ ,$$

where $z \in LVar^c$ and $z' \in LVar^{c'}$ are distinct fresh logical variables.

By definition of the global invariant, the assumption $\omega, \sigma \models_{\mathcal{G}} GI$ implies that there exists a reachable stable configuration $\langle T', \sigma' \rangle$ with $dom(\sigma) = dom(\sigma')$ and $\sigma(\beta)(\mathsf{h}_{comm}) = \sigma'(\beta)(\mathsf{h}_{comm})$ for all $\beta \in dom(\sigma)$.

Assuming α as the identity of the caller, i.e., $\omega(z) = \alpha$, then $\omega, \sigma \models_{\mathcal{G}} pre(e_0.m(\vec{e}))[z/\mathsf{this}]$ implies

$$\omega, \sigma(\alpha), \tau_1 \models_{\mathcal{L}} pre(e_0.m(\vec{e}))$$

by the substitution Lemma 3, for some local state τ_1 with $\tau_1(\mathsf{this}) = \omega(z) = \alpha$ and $\tau_1(u) = \omega(u)$ for all local variables occurring in $pre(e_0.m(\vec{e}))$. By definition of the precondition, there exists a (not necessarily stable) reachable configuration $\langle T_1, \sigma_1 \rangle$ such that $\sigma_1(\alpha) = \sigma(\alpha)$ and $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T_1$. We define the configuration $\langle T_1', \sigma_1' \rangle$

1. as the last stable global configuration in the computation leading to $\langle T_1, \sigma_1 \rangle$, if its instance state for α coincides with $\sigma_1(\alpha)$, and
2. as the next configuration following $\langle T_1, \sigma_1 \rangle$ after extending the computation by one computation step, otherwise. Note that the extension is deterministic, since a reachable, unstable configuration can proceed in only one way.

In the first case, either $\langle T_1, \sigma_1 \rangle$ itself is stable, or the computation $\langle T_1', \sigma_1' \rangle \longrightarrow^* \langle T_1, \sigma_1 \rangle$ executes some object creation or communication possibly followed by observations in objects different from α, but it does not execute any observations in α. From $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T_1$ we conclude that also $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T_1'$, since otherwise its statement would begin with an assignment. Furthermore we know $\sigma_1(\alpha) = \sigma_1'(\alpha)$.

In the second case, $\langle T_1, \sigma_1 \rangle$ is not stable. Furthermore, since the instance state $\sigma_1(\alpha)$ differs from the instance state of α in the last stable configuration in the computation, we know that in the last two steps of the computation a communication has taken place, where the sender object is α which already executed its observation, but not yet the receiver object.

Since $\langle T', \sigma' \rangle$ is stable, the last element in the sequence $\sigma'(\alpha)(\mathsf{h}_{comm})$ is not the observation of the sender part of a self-communication. Since $\sigma(\alpha)(\mathsf{h}_{comm}) = \sigma'(\alpha)(\mathsf{h}_{comm})$, the last element in the sequence $\sigma(\alpha)(\mathsf{h}_{comm}) = \sigma_1(\alpha)(\mathsf{h}_{comm})$ is neither the observation of the sender part of a self-communication. Thus the receiver object differs from α. Executing the observation of the receiver part outside of α leads to a configuration $\langle T_1', \sigma_1' \rangle$ defining the same instance state for α as σ_1 and still containing $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1)$.

Thus we conclude that $\langle T_1', \sigma_1' \rangle$ is a reachable stable configuration such that $\sigma_1'(\alpha) = \sigma_1(\alpha)$ and $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T_1'$.

Recall that $\langle T', \sigma' \rangle$ is a reachable stable configuration with $\sigma'(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, especially $\sigma'(\alpha)(\mathsf{h}_{comm}) = \sigma(\alpha)(\mathsf{h}_{comm}) = \sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_1'(\alpha)(\mathsf{h}_{comm})$. Using the global merging Lemma 11 applied to $\langle T_1', \sigma_1' \rangle$ and $\langle T', \sigma' \rangle$ we get that there is a reachable stable configuration $\langle T'', \sigma'' \rangle$ with $\sigma''(\alpha) = \sigma_1'(\alpha) = \sigma_1(\alpha) = \sigma(\alpha)$ and $\sigma''(\gamma) = \sigma'(\gamma)$ for all objects $\gamma \neq \alpha$ from the domain of σ', which equals the domain of σ. Furthermore, since $\langle T_1', \sigma_1' \rangle$ and $\langle T'', \sigma'' \rangle$ are stable, $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T_1'$, $\tau_1(\mathsf{this}) = \alpha$, and $\sigma_1'(\alpha) = \sigma''(\alpha)$, the local merging Lemma 10 implies that $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T''$.

We get that $\langle T'', \sigma'' \rangle$ is a reachable stable configuration containing the local configuration $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T''$, and furthermore $\sigma''(\alpha) = \sigma(\alpha)$ and $\sigma''(\gamma) = \sigma'(\gamma)$, for all other objects $\gamma \neq \alpha$ from the domain of σ

Similarly for the callee, say β, the assumption $\omega, \sigma \models_{\mathcal{G}} I_{c'}[z'/\mathsf{this}]$ implies $\omega, \sigma(\beta), \tau_2 \models_{\mathcal{L}} I_{c'}$ for some local state τ_2 with $\tau_2(\mathsf{this}) = \omega(z') = \beta$. Note that $I_{c'}$ contains instance variables, only. By definition of the class invariant, there is a reachable global configuration $\langle T_2, \sigma_2 \rangle$ such that $\sigma_2(\beta) = \sigma(\beta)$.

In the case of a self-call, i.e., for $\alpha = \beta$, we directly get that $\langle T'', \sigma'' \rangle$ is a reachable stable configuration such that $\sigma''(\alpha) = \sigma(\alpha)$, $\sigma''(\beta) = \sigma(\beta)$, and $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T''$.

For non-self-calls, i.e., when $\alpha \neq \beta$, we need to fall back upon the two merging lemmas once more to obtain a common reachable configuration: Analogously to the caller part we can show that there is a reachable stable configuration $\langle T_2', \sigma_2' \rangle$ with $\sigma_2'(\beta) = \sigma_2(\beta) = \sigma(\beta)$. The global merging Lemma 11 applied to $\langle T_2', \sigma_2' \rangle$ and $\langle T'', \sigma'' \rangle$ yields that there is a reachable stable configuration $\langle T''', \sigma''' \rangle$ with $\sigma'''(\beta) = \sigma_2'(\beta) = \sigma(\beta)$ and $\sigma'''(\alpha) = \sigma''(\alpha) = \sigma(\alpha)$. Furthermore, since $\langle T'', \sigma'' \rangle$ and $\langle T''', \sigma''' \rangle$ are stable, $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T_1''$, $\tau_1(\mathsf{this}) = \alpha$, and $\sigma'''(\alpha) = \sigma''(\alpha)$, the local merging Lemma 10 implies $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T'''$. Thus $\langle T''', \sigma''' \rangle$ is a reachable stable global configuration with $\sigma'''(\alpha) = \sigma(\alpha)$, $\sigma'''(\beta) = \sigma(\beta)$, and $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T'''$.

The antecedent $\omega, \sigma \models_{\mathcal{G}} \mathsf{isfree}(z'.\mathsf{lock}, \mathsf{id})$ of the cooperation test expands to $\omega, \sigma \models_{\mathcal{G}} z'.\mathsf{lock}{=}(\mathsf{nil}, 0) \vee z'.\mathsf{lock} \leq \mathsf{id}$, where id is the identity of the caller, i.e., $\sigma(\beta)(\mathsf{lock}) = (nil, 0) \vee \sigma(\beta)(\mathsf{lock}) = \tau_1(\mathsf{id})$. By Lemma 5 $isfree(T''' \backslash \{\xi\}, \alpha)$, where ξ is the stack with $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1)$ on top. This means, the local configuration of the method invocation is enabled in $\langle T''', \sigma''' \rangle$. Furthermore, using the substitution Lemma 3, $\omega, \sigma \models_{\mathcal{G}} e_0[z/\mathsf{this}]{=}z'$ implies that $[\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha), \tau_1} = \beta$. Executing the method invocation results in a reachable global configuration with still the same global state σ''', and containing the local configurations for caller and callee $(\tau_1, \langle \vec{y}_1 := \vec{e}_1 \rangle; stm_1)$ and $(\tau_{init}[\mathsf{this} \mapsto \beta][\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma'''(\alpha), \tau_1}], body_{m,c'})$.

The definition of the augmentation and $\sigma'''(\alpha) = \sigma(\alpha)$ gives $\omega, \sigma(\alpha), \tau_1 \models_{\mathcal{L}} post(e_0.m(\vec{e}))$, which by the substitution Lemma 3 yields the required postcondition of the method call in α:

$$\omega, \sigma \models_{\mathcal{G}} post(e_0.m(\vec{e}))[z/\mathsf{this}] \; .$$

For the precondition of the callee's method body we argue similarly: By definition of the precondition of the method body

$$\omega, \sigma(\beta), \tau_{init}[\mathsf{this} \mapsto \beta][\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma'''(\alpha), \tau_1}] \models_{\mathcal{L}} pre(body_{m,c'}) \ .$$

Defining $pre'(body_{m,c'}) = pre(body_{m,c'})[\mathsf{InitVal}(\vec{v})/\vec{v}]$ gives

$$\omega, \sigma(\beta), \tau[\mathsf{this} \mapsto \beta][\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma'''(\alpha), \tau_1}] \models_{\mathcal{L}} pre'(body_{m,c'})$$

for an arbitrary local state τ. Note that all variables occurring in $pre'(body_{m,c'})$ are either instance variables, this, or formal parameters, since the remaining local variables are substituted by their initial values. For the global expression $\vec{E} = \vec{e}[z/\mathsf{this}]$ we obtain by the substitution Lemma 3

$$[\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![\vec{e}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma(\alpha),\tau_1} = [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma'''(\alpha),\tau_1} \ ,$$

and further with the same lemma, the precondition of the method body in the form as required by the cooperation test:

$$\omega, \sigma \models_{\mathcal{G}} pre'(body_{m,c'})[z', \vec{E}/\mathsf{this}, \vec{u}] \ .$$

Let $\langle T''', \sigma''' \rangle \longrightarrow \langle T'''_{comm}, \sigma'''_{comm} \rangle \longrightarrow \langle T'''_{obs1}, \sigma'''_{obs1} \rangle \longrightarrow \langle T'''_{obs2}, \sigma'''_{obs2} \rangle$ be the computation executing the method invocation in the local configuration $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T'''$ and the observations of both the caller and the callee, in this order. Then

$$\begin{aligned} \sigma'''_{comm} &= \sigma''' \ , \\ \sigma'''_{obs1} &= \sigma'''_{comm}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}] \ , \ \text{and} \\ \sigma'''_{obs2} &= \sigma'''_{obs1}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma'''_{obs1}(\beta),\tau_{callee}}] \ , \end{aligned}$$

where the local state τ_{callee} is given by $\tau_{init}[\mathsf{this} \mapsto \beta][\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}]$. We have to show that

$$\omega, \sigma \models_{\mathcal{G}} GI[\vec{E}'_2/z'.\vec{y}_2][\vec{E}_1/z.\vec{y}_1] \ ,$$

where $\vec{E}_1 = \vec{e}_1[z/\mathsf{this}]$ and $\vec{E}'_2 = \vec{e}'_2[z', \vec{E}/\mathsf{this}, \vec{u}]$ with $\vec{e}'_2 = \vec{e}_2[\mathsf{InitVal}(\vec{v})/\vec{v}]$ and $\vec{E} = \vec{e}[z/\mathsf{this}]$. Applying Lemma 2 yields

$$\begin{aligned} [\![GI[\vec{E}'_2/z'.\vec{y}_2][\vec{E}_1/z.\vec{y}_1]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![GI[\vec{E}'_2/z'.\vec{y}_2]]\!]_{\mathcal{G}}^{\omega,\sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma}]} \\ &= [\![GI]\!]_{\mathcal{G}}^{\omega,\sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma}][\beta.\vec{y}_2 \mapsto [\![\vec{E}'_2]\!]_{\mathcal{G}}^{\omega,\sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma}]}]} \ . \end{aligned}$$

Let $\sigma_{obs1} = \sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma}]$ and $\sigma_{obs2} = \sigma_{obs1}[\beta.\vec{y}_2 \mapsto [\![\vec{E}'_2]\!]_{\mathcal{G}}^{\omega,\sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma}]}]$. With the substitution Lemmas 3 and 1 we can get the following equations.

$$\begin{aligned}
[\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![\vec{e}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau_1} \\
&= [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma'''(\alpha),\tau_1} = [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1} , \\
[\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![\vec{e}_1[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau_1} \\
&= [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''(\alpha),\tau_1} = [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1} , \\
[\![\vec{E}'_2]\!]_{\mathcal{G}}^{\omega,\sigma_{obs1}} &= [\![\vec{e}'_2[z',\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma_{obs1}} \\
&= [\![\vec{e}_2[\mathsf{InitVal}(\vec{v})/\vec{v}][z',\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma_{obs1}} \\
&= [\![\vec{e}_2]\!]_{\mathcal{L}}^{\omega,\sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}](\beta),\tau_{init}[\mathsf{this}\mapsto\beta][\vec{u}\mapsto[\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]} \\
&= [\![\vec{e}_2]\!]_{\mathcal{L}}^{\omega,\sigma'''_{obs1}(\beta),\tau_{init}[\mathsf{this}\mapsto\beta][\vec{u}\mapsto[\![\vec{e}]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}]} \\
&= [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma'''_{obs1}(\beta),\tau_{callee}} .
\end{aligned}$$

This means,

$$\sigma_{obs2} = \sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}][\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma'''_{obs1}(\beta),\tau_{callee}}] .$$

Remember that

$$\sigma'''_{obs2} = \sigma'''_{comm}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}][\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\sigma'''_{obs1}(\beta),\tau_{callee}}] .$$

At the end, the communication histories all coincide, i.e., $\sigma'''_{comm}(\gamma)(\mathsf{h}_{comm}) = \sigma'''(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, and consequently σ_{obs2} agrees with σ'''_{obs2} also on all h_{comm} values. Consequently, $\langle T'''_{obs2}, \sigma'''_{obs2}\rangle$ is a reachable stable configuration satisfying $\sigma'''_{obs2}(\gamma)(\mathsf{h}_{comm}) = \sigma_{obs2}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, and we get by the definition of the annotation $\omega, \sigma_{obs2} \models_{\mathcal{G}} GI$, and finally $\omega, \sigma \models_{\mathcal{G}} GI[\vec{E}'_2/z'.\vec{y}_2][\vec{E}_1/z.\vec{y}_1]$, as required

The case for non-synchronized methods is shown analogously, only the antecedent $\mathsf{isfree}(z'.\mathsf{lock}, \mathsf{id})$ is dropped.

The case for the invocation of start-methods, whose enabledness requires $\neg started(T''', \beta)$, we have the additional antecedent $\neg z'.\mathsf{started}$, which implies $\neg\sigma'''(\beta)(\mathsf{started})$. So by Lemma 6 implies that $started(T''', \beta)$ iff $\sigma'''(\beta)(\mathsf{started})$.

Case: Return

In this case assume

$$\begin{aligned}
&\omega, \sigma \models_{\mathcal{G}} GI \wedge pre'(\mathsf{return}\ e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge pre(\mathsf{receive}\ v)[z/\mathsf{this}] \wedge \\
&\qquad e_0[z/\mathsf{this}] = z' .
\end{aligned}$$

By definition of the global invariant, $\omega, \sigma \models_{\mathcal{G}} GI$ implies that there is a reachable stable configuration $\langle T', \sigma'\rangle$ such that $dom(\sigma) = dom(\sigma')$ and $\sigma(\beta)(\mathsf{h}_{comm}) = \sigma'(\beta)(\mathsf{h}_{comm})$ for all $\beta \in dom(\sigma)$.

Assuming α as the identity of the caller, i.e., $\omega(z) = \alpha$, then Lemma 3 $\omega, \sigma \models_{\mathcal{G}} pre(\mathsf{receive}\, v)[z/\mathsf{this}]$ gives

$$\omega, \sigma(\alpha), \tau_1 \models_{\mathcal{L}} pre(\mathsf{receive}\, v)$$

for some local state τ_1 with $\tau_1(\mathsf{this}) = \omega(z) = \alpha$ and $\tau_1(u) = \omega(u)$ for all local variables occurring in $pre(\mathsf{receive}\, v)$. This implies by definition of the annotation that there exists a reachable configuration $\langle T_1, \sigma_1\rangle$ such that $\sigma_1(\alpha) = \sigma(\alpha)$ and $(\tau_1, \langle \mathsf{receive}\, v; \vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T_1$. Similar as in the case for method calls, we define the configuration $\langle T_1', \sigma_1'\rangle$

1. as the last stable global configuration in the computation leading to $\langle T_1, \sigma_1\rangle$, if its instance state for α coincides with $\sigma_1(\alpha)$, and
2. as the next configuration following $\langle T_1, \sigma_1\rangle$ after extending the computation by one computation step, otherwise. Note again that the extension is deterministic.

As in the case of method invocation, we conclude that $\langle T_1', \sigma_1'\rangle$ is a reachable stable configuration with $\sigma_1'(\alpha) = \sigma_1(\alpha)$ and $(\tau_1, \langle \mathsf{receive}\, v; \vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T_1'$.

Recall that $\langle T', \sigma'\rangle$ is a reachable stable configuration with $\sigma'(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, especially $\sigma'(\alpha)(\mathsf{h}_{comm}) = \sigma(\alpha)(\mathsf{h}_{comm}) = \sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_1'(\alpha)(\mathsf{h}_{comm})$. Using the global merging Lemma 11 on $\langle T_1', \sigma_1'\rangle$ and $\langle T', \sigma'\rangle$ we get that there is a reachable stable configuration $\langle T'', \sigma''\rangle$ with $\sigma''(\alpha) = \sigma_1'(\alpha) = \sigma_1(\alpha) = \sigma(\alpha)$ and $\sigma''(\gamma) = \sigma'(\gamma)$ for all objects $\gamma \neq \alpha$ from the domain of σ', which equals the domain of σ. Furthermore, since $\langle T_1', \sigma_1'\rangle$ and $\langle T'', \sigma''\rangle$ are stable, $(\tau_1, \langle \mathsf{receive}\, v; \vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T_1'$, $\tau_1(\mathsf{this}) = \alpha$, and $\sigma_1'(\alpha) = \sigma''(\alpha)$, the local merging Lemma 10 implies that $(\tau_1, \langle \mathsf{receive}\, v; \vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T''$. So $\langle T'', \sigma''\rangle$ is a reachable stable configuration containing the local configuration $(\tau_1, \langle \mathsf{receive}\, v; \vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T''$, and with $\sigma''(\alpha) = \sigma(\alpha)$ and $\sigma''(\gamma) = \sigma'(\gamma)$ for all other objects $\gamma \neq \alpha$ from the domain of σ.

Similarly for the callee, say β, $\omega, \sigma \models_{\mathcal{G}} pre'(\mathsf{return}\; e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}]$ implies that $\omega, \sigma(\beta), \tau_2 \models_{\mathcal{L}} pre(\mathsf{return}\; e_{ret})$ for some local state τ_2 with $\tau_2(\mathsf{this}) = \omega(z') = \beta$, $\tau_2(\vec{u}) = [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}$ for the formal parameters $\vec{u}$, and $\tau_2(\vec{v}) = \omega(\vec{v}')$ for the local variables $\vec{v}$ without the formal parameters and this. By definition of the annotation there is a reachable global configuration $\langle T_2, \sigma_2\rangle$ such that $\sigma_2(\beta) = \sigma(\beta)$ and $(\tau_2, \langle \mathsf{return}\; e_{ret}; \vec{y}_3 := \vec{e}_3\rangle) \in T_2$.

In the case of a self-call, i.e., if the caller and the callee are the same object, we get directly that $\langle T'', \sigma''\rangle$ is a reachable stable configuration such that $\sigma''(\alpha) = \sigma(\alpha)$, $\sigma''(\beta) = \sigma(\beta)$, and $(\tau_1, \langle \mathsf{receive}\, v; \vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T''$. Furthermore, by the local merging Lemma 10 using $(\tau_2, \langle \mathsf{return}\; e_{ret}; \vec{y}_3 := \vec{e}_3\rangle) \in T_2$ also $(\tau_2, \langle \mathsf{return}\; e_{ret}; \vec{y}_3 := \vec{e}_3\rangle) \in T''$.

Assume now that $\alpha \neq \beta$. Analogously to the caller part, we can show that there is a reachable stable configuration $\langle T_2', \sigma_2'\rangle$ with $\sigma_2'(\beta) = \sigma_2(\beta) = \sigma(\beta)$ and $(\tau_2, \langle \mathsf{return}\; e_{ret}; \vec{y}_3 := \vec{e}_3\rangle) \in T_2'$. Using the global merging Lemma 11 applied to $\langle T_2', \sigma_2'\rangle$ and $\langle T'', \sigma''\rangle$ we get that there is a reachable stable configuration $\langle T''', \sigma'''\rangle$ with $\sigma'''(\beta) = \sigma_2'(\beta) = \sigma(\beta)$ and $\sigma'''(\alpha) = \sigma''(\alpha) = \sigma(\alpha)$. Furthermore, since $\langle T'',\sigma''\rangle$ and $\langle T''',\sigma'''\rangle$ are stable, $(\tau_1,\langle \mathsf{receive}\, v;\vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T''_1$, $\tau_1(\mathsf{this}) = \alpha$, and $\sigma'''(\alpha) = \sigma''(\alpha)$, the local merging Lemma 10 implies that $(\tau_1,\langle \mathsf{receive}\, v;\vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T'''$. Correspondingly for the callee, $\langle T'_2,\sigma'_2\rangle$ and $\langle T''',\sigma'''\rangle$ are stable, $(\tau_2,\langle \mathsf{return}\; e_{ret};\vec{y}_3 := \vec{e}_3\rangle) \in T'_2$, $\tau_2(\mathsf{this}) = \beta$, $\sigma'''(\beta) = \sigma'_2(\beta)$, and the local merging Lemma 10 implies that $(\tau_2,\langle \mathsf{return}\; e_{ret};\vec{y}_3 := \vec{e}_3\rangle) \in T'''$. Thus $\langle T''',\sigma'''\rangle$ is a reachable stable global configuration with $\sigma'''(\alpha) = \sigma(\alpha)$, $\sigma'''(\beta) = \sigma(\beta)$, $(\tau_1,\langle \mathsf{receive}\, v;\vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T'''$, and $(\tau_2,\langle \mathsf{return}\; e_{ret};\vec{y}_3 := \vec{e}_3\rangle) \in T'''$.

With Lemma 3, the antecedent $\omega,\sigma \models_{\mathcal{G}} e_0[z/\mathsf{this}]{=}z'$ implies that $[\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau_1}$ equals β. Executing the communication of the return value results in a reachable global configuration still having the same global state σ''', and containing the local configuration $(\tau_1[v \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma'''(\beta),\tau_2}],\langle \vec{y}_4 := \vec{e}_4\rangle; stm_1)$ of the caller, and the local configuration $(\tau_2,\langle \vec{y}_3 := \vec{e}_3\rangle)$ of the callee.

By the definition of the augmentation and using $\sigma'''(\alpha) = \sigma(\alpha)$ we get that $\omega,\sigma(\alpha),\tau_1[v \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma'''(\beta),\tau_2}] \models_{\mathcal{L}} \mathit{post}(\mathsf{receive}\, v)$. Applying the lifting Lemma 3 once more, but in the other direction as before, and using

$$[\![e_{ret}]\!]_{\mathcal{E}}^{\sigma'''(\beta),\tau_2} = [\![e_{ret}]\!]_{\mathcal{L}}^{\omega,\sigma(\beta),\tau_2} = [\![e'_{ret}[z',\vec{E}/\mathsf{this},\vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![E'_{ret}]\!]_{\mathcal{G}}^{\omega,\sigma}$$

gives

$$\omega,\sigma \models_{\mathcal{G}} \mathit{post}(\mathsf{receive}\, v)[z,E'_{ret}/\mathsf{this},v]\ .$$

Similarly for the callee, $\omega,\sigma(\beta),\tau_2 \models_{\mathcal{L}} \mathit{post}(\mathsf{return}\; e_{ret})$. By the definition of τ_2 we have $\tau_2(\mathsf{this}) = \omega(z') = \beta$, $\tau_2(\vec{u}) = [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}$ for the formal parameters $\vec{u}$, and $\tau_2(\vec{v}) = \omega(\vec{v}')$ for the local variables $\vec{v}$ without the formal parameters and this. Applying the lifting Lemma 3 again yield the required post-condition from the cooperation test.

$$\omega,\sigma \models_{\mathcal{G}} \mathit{post}'(\mathsf{return}\; e_{ret})[z',\vec{E}/\mathsf{this},\vec{u}]\ .$$

For the global invariant, let $\langle T''',\sigma'''\rangle \longrightarrow \langle T'''_{comm},\sigma'''_{comm}\rangle \longrightarrow \langle T'''_{obs1},\sigma'''_{obs1}\rangle \longrightarrow \langle T'''_{obs2},\sigma'''_{obs2}\rangle$ be the computation executing the return and receive statements in the local configurations $(\tau_1,\langle \mathsf{receive}\, v;\vec{y}_4 := \vec{e}_4\rangle; stm_1) \in T'''$ and $(\tau_2,\langle \mathsf{return}\; e_{ret};\vec{y}_3 := \vec{e}_3\rangle) \in T'''$, and the observations of both the callee and the caller, in this order. The communication does not change any instance states, i.e., $\sigma'''_{comm} = \sigma'''$. Furthermore, after the execution of the assignment $\vec{y}_3 := \vec{e}_3$ in the object β we get $\sigma'''_{obs1} = \sigma'''_{comm}[\beta.\vec{y}_3 \mapsto [\![\vec{e}_3]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2}]$. The execution of the observation $\vec{y}_4 := \vec{e}_4$ in the caller object α results finally in the global state σ'''_{obs2} defined by $\sigma'''_{obs1}[\alpha.\vec{y}_4 \mapsto [\![\vec{e}_4]\!]_{\mathcal{E}}^{\sigma'''_{obs1}(\alpha),\tau_1[v \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2}]}]$.

We have to show that

$$\omega,\sigma \models_{\mathcal{G}} GI[\vec{E}_4/z.\vec{y}_4][\vec{E}'_3/z'.\vec{y}_3]\ ,$$

where $\vec{E}'_3 = \vec{e}'_3[z', \vec{E}/\mathsf{this}, \vec{u}]$, $\vec{E}_4 = \vec{e}_4[z, E'_{ret}/\mathsf{this}, v]$, $E'_{ret} = e'_{ret}[z', \vec{E}/\mathsf{this}, \vec{u}]$, and e'_{ret} and $\vec{e}'_3$ denote the given expressions with every local variable except the formal parameters and this replaced by a fresh one. Applying Lemma 2 yields

$$\begin{aligned}
[\![GI[\vec{E}_4/z.\vec{y}_4][\vec{E}'_3/z'.\vec{y}_3]]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![GI[\vec{E}_4/z.\vec{y}_4]]\!]_{\mathcal{G}}^{\omega,\sigma[\beta.\vec{y}_3 \mapsto [\![\vec{E}'_3]\!]_{\mathcal{G}}^{\omega,\sigma}]} \\
&= [\![GI]\!]_{\mathcal{G}}^{\omega,\sigma[\beta.\vec{y}_3 \mapsto [\![\vec{E}'_3]\!]_{\mathcal{G}}^{\omega,\sigma}][\alpha.\vec{y}_4 \mapsto [\![\vec{E}_4]\!]_{\mathcal{G}}^{\omega,\sigma[\beta.\vec{y}_3 \mapsto [\![\vec{E}'_3]\!]_{\mathcal{G}}^{\omega,\sigma}]}]} .
\end{aligned}$$

Let $\sigma_{obs1} = \sigma[\beta.\vec{y}_3 \mapsto [\![\vec{E}'_3]\!]_{\mathcal{G}}^{\omega,\sigma}]$ and $\sigma_{obs2} = \sigma_{obs1}[\alpha.\vec{y}_4 \mapsto [\![\vec{E}_4]\!]_{\mathcal{G}}^{\omega,\sigma[\beta.\vec{y}_3 \mapsto [\![\vec{E}'_3]\!]_{\mathcal{G}}^{\omega,\sigma}]}]$. Then we have to show that

$$\omega, \sigma_{obs2} \models_{\mathcal{G}} GI .$$

With the substitution Lemmas 3 and 1, we get the following equations

$$\begin{aligned}
[\![\vec{E}'_{ret}]\!]_{\mathcal{G}}^{\omega,\sigma_{obs1}} &= [\![e'_{ret}[z', \vec{E}/\mathsf{this}, \vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma_{obs1}} \\
&= [\![e_{ret}]\!]_{\mathcal{L}}^{\omega,\sigma_{obs1}(\beta),\tau_2} \\
&= [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma(\beta),\tau_2} = \\
&= [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2} , \\
[\![\vec{E}_4]\!]_{\mathcal{G}}^{\omega,\sigma_{obs1}} &= [\![\vec{e}_4[z, E'_{ret}/\mathsf{this}, v]]\!]_{\mathcal{G}}^{\omega,\sigma_{obs1}} \\
&= [\![\vec{e}_4]\!]_{\mathcal{E}}^{\sigma_{obs1}(\alpha),\tau_1[v \mapsto [\![E'_{ret}]\!]_{\mathcal{G}}^{\omega,\sigma_{obs1}}]} \\
&= [\![\vec{e}_4]\!]_{\mathcal{E}}^{\sigma_{obs1}(\alpha),\tau_1[v \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2}]} , \\
[\![\vec{E}'_3]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![\vec{e}'_3[z', \vec{E}/\mathsf{this}, \vec{u}]]\!]_{\mathcal{G}}^{\omega,\sigma} \\
&= [\![\vec{e}_3]\!]_{\mathcal{L}}^{\omega,\sigma(\beta),\tau_2} \\
&= [\![\vec{e}_3]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2} .
\end{aligned}$$

This means,

$$\sigma_{obs2} = \sigma[\beta.\vec{y}_3 \mapsto [\![\vec{e}_3]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2}][\alpha.\vec{y}_4 \mapsto [\![\vec{e}_4]\!]_{\mathcal{E}}^{\sigma'''_{obs1}(\alpha),\tau_1[v \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2}]}] .$$

Remember that

$$\sigma'''_{obs2} = \sigma'''_{comm}[\beta.\vec{y}_3 \mapsto [\![\vec{e}_3]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2}][\alpha.\vec{y}_4 \mapsto [\![\vec{e}_4]\!]_{\mathcal{E}}^{\sigma'''_{obs1}(\alpha),\tau_1[v \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\beta),\tau_2}]}] .$$

Since $\sigma'''_{comm}(\gamma)(\mathsf{h}_{comm}) = \sigma'''(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm})$ for all objects $\gamma \in dom(\sigma)$, the state σ_{obs2} agrees with σ'''_{obs2} also on all h_{comm} values. Therefore $\langle T'''_{obs2}, \sigma'''_{obs2}\rangle$ is a reachable stable configuration with $\sigma'''_{obs2}(\gamma)(\mathsf{h}_{comm}) = \sigma_{obs2}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, and we get by the definition of the annotation $\omega, \sigma_{obs2} \models_{\mathcal{G}} GI$, and thus finally $\omega, \sigma \models_{\mathcal{G}} GI[\vec{E}_4/z.\vec{y}_4][\vec{E}'_3/z'.\vec{y}_3]$, as required.

The case for methods without a return value is shown analogously.

Case: Start_{skip}
In case the thread is already started, assume the left-hand side of Equation 8

$$\omega, \sigma \models_{\mathcal{G}} GI \wedge pre(e_0.\mathsf{start}(\vec{e}))[z/\mathsf{this}] \wedge I_{c'}[z'/\mathsf{this}] \wedge e_0[z/\mathsf{this}]{=}z' \wedge z'.\mathsf{started}\ .$$

As in the case of method invocation, we can show that there is a reachable stable global configuration $\langle T''', \sigma''' \rangle$ with $\sigma'''(\alpha) = \sigma(\alpha)$, $\sigma'''(\beta) = \sigma(\beta)$, and $(\tau_1, \langle e_0.start(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T'''$.

Using the lifting Lemma 3, $\omega, \sigma \models_{\mathcal{G}} e_0[z/\mathsf{this}]{=}z'$ implies that $[\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau_1} = \beta$. The additional antecedent $z'.\mathsf{started}$ implies $\sigma'''(\beta)(\mathsf{started})$, which equals by Lemma 6 $started(T''', \beta)$. This means, executing $e_0.start(\vec{e})$ in the local configuration $(\tau_1, \langle e_0.start(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T'''$ does not start a new thread and results in a reachable global configuration with still the same global state σ''', and containing the local configuration $(\tau_1, \langle \vec{y}_1 := \vec{e}_1 \rangle; stm_1)$ of the caller.

By the definition of the augmentation and using $\sigma'''(\alpha) = \sigma(\alpha)$ we get $\omega, \sigma(\alpha), \tau_1 \models_{\mathcal{L}} post(e_0.m(\vec{e}))$. Applying the lifting Lemma 3 once more gives the required post-condition

$$\omega, \sigma \models_{\mathcal{G}} post(e_0.m(\vec{e}))[z/\mathsf{this}]$$

For the global invariant, let $\langle T''', \sigma''' \rangle \longrightarrow \langle T'''_{comm}, \sigma'''_{comm} \rangle \longrightarrow \langle T'''_{obs1}, \sigma'''_{obs1} \rangle$ be the computation executing the method invocation in the local configuration $(\tau_1, \langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle; stm_1) \in T'''$ and the observations of the caller. Then $\sigma'''_{comm} = \sigma'''$ and $\sigma'''_{obs1} = \sigma'''_{comm}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}]$. We have to show that

$$\omega, \sigma \models_{\mathcal{G}} GI[\vec{E}_1/z.\vec{y}_1]\ ,$$

where $\vec{E}_1 = \vec{e}_1[z/\mathsf{this}]$. Applying Lemma 2 yields

$$[\![GI[\vec{E}_1/z.\vec{y}_1]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![GI]\!]_{\mathcal{G}}^{\omega,\sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma}]}\ .$$

Let $\sigma_{obs1} = \sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma}]$. For the expression $\vec{E}_1$ we get using the lifting Lemma 3

$$\begin{aligned} [\![\vec{E}_1]\!]_{\mathcal{G}}^{\omega,\sigma} &= [\![\vec{e}_1[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau_1} \\ &= [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''(\alpha),\tau_1} = [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}\ . \end{aligned}$$

This means,

$$\sigma_{obs1} = \sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}]\ .$$

Remember that

$$\sigma'''_{obs1} = \sigma'''_{comm}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}]\ .$$

Furthermore, $\sigma'''_{comm}(\gamma)(\mathsf{h}_{comm}) = \sigma'''(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, and thus $\sigma[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}]$ agrees also on all h_{comm} values with

$\sigma'''_{obs2} = \sigma'''_{comm}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma'''_{comm}(\alpha),\tau_1}]$. Therefore $\langle T'''_{obs1}, \sigma'''_{obs1}\rangle$ is a reachable stable configuration satisfying the requirement $\sigma'''_{obs2}(\gamma)(\mathsf{h}_{comm}) = \sigma_{obs2}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, and we get by the definition of the annotation $\omega, \sigma_{obs1} \models_{\mathcal{G}} GI$, and thus $\omega, \sigma \models_{\mathcal{G}} GI[\vec{E}_1/z.\vec{y}_1]$.

Case: TERMINATE
This proof case is analogous to the proof case of $\textsc{Start}_{skip}$, where the additional antecedent $\mathsf{id} = (z', 0)$ implies by the definition of the augmentation, that the return statement is executed in a local configuration being on the bottom of its stack. This means, the thread terminates, and Rule TERMINATE applies. □

Lemma 34 (Cooperation test: Instantiation). *The annotated program prog′ and the global invariant GI satisfy the verification conditions of the cooperation test for object creation of Definition 6.*

Proof (of Lemma 34). Let $\langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle$ be a statement in a class c' of $prog'$, and assume

$$\omega, \sigma \models_{\mathcal{G}} \exists z' \Big(\mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}])) \downarrow z'\Big)$$

with fresh logical variables $z \in LVar^{c'}$ and $z' \in LVar^{\mathsf{list\ Object}}$. Let $\omega(z) = \alpha$ and $\omega(u) = \beta$. According to the semantics of assertions we have that

$$\omega', \sigma \models_{\mathcal{G}} \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}])) \downarrow z'$$

for some logical environment ω' that assigns to z' a sequence of objects from $dom_{nil}^{\mathsf{Object}}(\sigma) = \bigcup_c dom_{nil}^c(\sigma)$, and agrees on the values of all other variables with ω. The assertion $\mathsf{Fresh}(z', u)$ is defined by

$$u \notin z' \wedge \mathsf{InitState}(u) \wedge \forall v(v \in z' \vee v = u) ,$$

where $\mathsf{InitState}(u)$ expands into $u \neq \mathsf{nil} \wedge \bigwedge_{x \in IVar_c} u.x = \mathsf{InitVal}(x)$. Thus, $\omega', \sigma \models_{\mathcal{G}} \mathsf{Fresh}(z', u)$ implies that $\beta \in dom^c(\sigma)$ with $\sigma(\beta) = \sigma_{inst}^{init}$, and $dom_{nil}^{\mathsf{Object}}(\sigma) = \omega'(z') \,\dot\cup\, \{\beta\}$. Let σ' be the global state with domain $dom^{\mathsf{Object}}(\sigma')$ given as $dom^{\mathsf{Object}}(\sigma)\backslash\{\beta\}$ and such that $\sigma'(\gamma) = \sigma(\gamma)$ for all $\gamma \in dom^{\mathsf{Object}}(\sigma)\backslash\{\beta\}$. Then $\sigma = \sigma'[\beta \mapsto \sigma_{inst}^{init}]$, and from

$$\omega', \sigma \models_{\mathcal{G}} (GI \wedge \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}])) \downarrow z'$$

we get with Lemma 8

$$\omega, \sigma' \models_{\mathcal{G}} GI \wedge \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}]) .$$

By definition of the annotation, $\omega, \sigma' \models_{\mathcal{G}} GI$ implies that there is a reachable stable configuration $\langle T_1, \sigma_1\rangle$ such that $dom(\sigma_1) = dom(\sigma')$ and $\sigma_1(\gamma)(\mathsf{h}_{comm}) = \sigma'(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma')$.

The precondition of the object creation statement

$$\omega, \sigma' \models_{\mathcal{G}} \exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}])$$

implies

$$\omega[u \mapsto Z], \sigma' \models_{\mathcal{G}} pre(u := \mathsf{new}^c)[z/\mathsf{this}]$$

for some $Z \in dom_{nil}^{\mathsf{Object}}(\sigma')$. Applying the lifting Lemma 3 we get that

$$\omega, \sigma'(\alpha), \tau \models_{\mathcal{L}} pre(u := \mathsf{new}^c)$$

for a local state τ with $\tau(\mathsf{this}) = \omega(z) = \alpha$, $\tau(u) = Z$, and $\tau(v) = \omega(v)$ for all other local variables v. By definition of the annotation, there is a reachable global configuration $\langle T_2, \sigma_2 \rangle$ such that $\sigma_2(\alpha) = \sigma'(\alpha)$ and $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle; stm) \in T_2$.

To be able to apply the global merging lemma and analogous to the situation in the proof of the cooperation test for communication (Lemma 33), we define the configuration $\langle T_2', \sigma_2' \rangle$ either

1. as the last stable global configuration in the computation leading to $\langle T_2, \sigma_2 \rangle$, if it defines the same instance state for α as σ_2, or
2. as the configuration following $\langle T_2, \sigma_2 \rangle$ when extending the computation by one computation step, otherwise. Again note that the extension is deterministic.

In the first case, either $\langle T_2, \sigma_2 \rangle$ itself is stable or the computation $\langle T_2', \sigma_2' \rangle \longrightarrow^* \langle T_2, \sigma_2 \rangle$ executes some object creation or communication possibly followed by observations in objects different from α, but it does not execute any observations in α. From $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle; stm) \in T_2$ we conclude that also $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle; stm) \in T_2'$ (otherwise its statement would begin with an assignment), and that $\sigma_2(\alpha) = \sigma_2'(\alpha)$. In the second case, $\langle T_2, \sigma_2 \rangle$ is not stable since otherwise the first case would apply. Furthermore, since the instance state $\sigma_2(\alpha)$ differs from the instance state of α in the last stable configuration in the computation, we know that in the last two steps of the computation a communication has taken place, where the sender object is α which already executed its observation, but not yet the receiver object.

Since $\langle T_1, \sigma_1 \rangle$ is stable, the last element in the sequence $\sigma_1(\alpha)(\mathsf{h}_{comm})$ is not the observation of the sender part of a self-communication. Since $\sigma_2(\alpha)(\mathsf{h}_{comm}) = \sigma'(\alpha)(\mathsf{h}_{comm}) = \sigma_1(\alpha)(\mathsf{h}_{comm})$, neither is the last element in $\sigma_2(\alpha)(\mathsf{h}_{comm})$. Thus the receiver object differs from α. Executing the observation of the receiver part outside α leads to a configuration $\langle T_2', \sigma_2' \rangle$ with the same instance state for α as σ_2 and still containing $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle; stm)$.

Thus we conclude that $\langle T_2', \sigma_2' \rangle$ is a reachable stable configuration such that $\sigma_2'(\alpha) = \sigma_2(\alpha) = \sigma'(\alpha)$ and $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e}\rangle; stm) \in T_2'$.

Recall that $\langle T_1, \sigma_1 \rangle$ is a reachable stable configuration with $\sigma_1(\gamma)(\mathsf{h}_{comm}) = \sigma'(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma')$, especially $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma'(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm}) = \sigma_2'(\alpha)(\mathsf{h}_{comm})$. Using the global merging Lemma 11 applied to the reachable stable global configurations $\langle T_2', \sigma_2' \rangle$ and $\langle T_1, \sigma_1 \rangle$ we get that there is a reachable stable configuration $\langle T_3, \sigma_3 \rangle$ with $\sigma_3(\alpha) = \sigma_2'(\alpha) = \sigma_2(\alpha) = \sigma'(\alpha)$

and $\sigma_3(\gamma) = \sigma_1(\gamma)$ for all objects γ different from α from the domain of σ_1, which equals the domain of σ'. Furthermore, since $\langle T_2', \sigma_2' \rangle$ and $\langle T_3, \sigma_3 \rangle$ are stable, $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e} \rangle; stm) \in T_2'$, $\tau(\mathsf{this}) = \alpha$, and $\sigma_2'(\alpha) = \sigma_3(\alpha)$, the local merging Lemma 10 implies that $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e} \rangle; stm) \in T_3$.

So we know that $\langle T_3, \sigma_3 \rangle$ is a reachable stable configuration containing the local configuration $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e} \rangle; stm) \in T_3$, and defining $\sigma_3(\alpha) = \sigma'(\alpha)$ and $\sigma_3(\beta) = \sigma_1(\gamma)$ for all other objects γ different from α from the domain of σ. Now executing the instantiation statement in the local configuration $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e} \rangle; stm)$ in $\langle T_3, \sigma_3 \rangle$ creates a new object $\beta \notin dom(\sigma_3)$ and results in a new global configuration $\langle T_3', \sigma_3' \rangle$ with $\sigma_3' = \sigma_3[\beta \mapsto \sigma_{inst}^{init}]$. Especially, $\langle T_3', \sigma_3' \rangle$ is a reachable global configuration with $\sigma_3'(\alpha) = \sigma_3(\alpha) = \sigma'(\alpha) = \sigma(\alpha)$ and $(\tau, \langle \vec{y} := \vec{e} \rangle; stm)$ in $\langle T_3, \sigma_3 \rangle \in T_3'$, i.e.,

$$\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} post(u := \mathsf{new}^c) \; .$$

and with the lifting Lemma 3 together with the definition of τ this means

$$\omega, \sigma \models_{\mathcal{G}} post(u := \mathsf{new}^c)[z/\mathsf{this}]$$

as required in the cooperation test.

As $\langle T_3', \sigma_3' \rangle$ is a reachable global configuration with $\sigma_3'(\beta) = \sigma(\beta) = \sigma_{inst}^{init}$ we know

$$\omega, \sigma(\beta), \tau' \models_{\mathcal{L}} I_c$$

for some local state τ with $\tau(\mathsf{this}) = \beta$. Applying the lifting Lemma 3 again with $\omega(u) = \beta$ yields the required condition for the class invariant.

$$\omega, \sigma \models_{\mathcal{G}} I_c[u/\mathsf{this}] \; ,$$

as required.

To show finally satisfaction of the global invariant, let $\langle T_3, \sigma_3 \rangle \longrightarrow \langle T_3', \sigma_3' \rangle \longrightarrow \langle T_3'', \sigma_3'' \rangle$ be the computation executing the object creation and its observation in the local configuration $(\tau, \langle u := \mathsf{new}^c; \vec{y} := \vec{e} \rangle; stm) \in T_3$ such that $\beta \notin dom(\sigma_3)$ is the identity of the new object. Then $\sigma_3' = \sigma_3[\beta \mapsto \sigma_{inst}^{init}]$, and $\sigma_3'' = \sigma_3'[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_3'(\alpha), \tau}]$. We have to show that

$$\omega, \sigma \models_{\mathcal{G}} GI[\vec{E}/z.\vec{y}] \; ,$$

where $\vec{E} = \vec{e}[z/\mathsf{this}]$. Applying the substitution Lemma 2 yields

$$[\![GI[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![GI]\!]_{\mathcal{G}}^{\omega,\sigma[\alpha.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]} \; .$$

Now let $\sigma'' = \sigma[\alpha.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]$. Transforming the expression $\vec{E}$

$$[\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![\vec{e}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} = [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} = [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_3(\alpha),\tau} = [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_3'(\alpha),\tau} \; ,$$

with the help of the lifting Lemma 3 and the definition of τ, this means

$$\sigma'' = \sigma'[\beta \mapsto \sigma_{inst}^{init}][\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_3'(\alpha),\tau}] \ .$$

Remember that

$$\sigma_3'' = \sigma_3[\beta \mapsto \sigma_{inst}^{init}][\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_3'(\alpha),\tau}] \ .$$

Furthermore, $\sigma_3'(\gamma)(\mathsf{h}_{comm}) = \sigma_3(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma')$, and consequently σ'' agrees also on all h_{comm} values with σ_3''.

Thus $\langle T_3'', \sigma_3'' \rangle$ is a reachable stable configuration such that $\sigma_3''(\gamma)(\mathsf{h}_{comm}) = \sigma''(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, and we get by the definition of the annotation $\omega, \sigma'' \models_{\mathcal{G}} GI$, and thus $\omega, \sigma \models_{\mathcal{G}} GI[\vec{E}/z.\vec{y}]$, as required. $\square$

C Notation

typ. element, notation	symbol	definition, domain	explanation
		Syntax	
c	$\mathcal{C}$		class names
m	$\mathcal{M}$		method names
t	$\mathcal{T}$		types
x	$IVar^t$		instance variables
u, v	$TVar^t$		local variables
y	Var^t		local or instance variables
f	F		operators
e	Exp_c^t		expressions
$sexp$	$SExp_c^t$		side-effect expressions
stm	Stm_c		statements
$meth$	$Meth_c$		methods
$class$	$Class$		classes
$prog$			program
		Semantics	
α	Val^c		object identities
	Val^t		values of type t
	Val	$\bigcup_t Val^t$	all values (except nil)
nil^c			empty reference of type c
	Val^c_{nil}	$Val^c \dot{\cup} \{nil^c\}$	identities or empty reference
val	Val_{nil}	$\bigcup_t \{Val^t_{nil}\}$	all values
τ	Σ_{loc}	$TVar \dot{\cup} \{\mathsf{this}\} \rightharpoonup Val_{nil}$	local state
σ_{inst}	Σ_{inst}	$IVar \rightharpoonup Val_{nil}$	instance state
σ	Σ	$\bigcup_c Val^c \rightharpoonup \Sigma_{inst}$	global state
(σ_{inst}, τ)		$\Sigma_{inst} \times \Sigma_{\mathsf{loc}}$	instance local state
$dom^c(\sigma)$		2^{Val^c}	existing instances of c in σ
$dom^t(\sigma)$		2^{Val^t}	existing values of type t
$dom(\sigma)$		2^{Val}	existing values
$dom_{nil}(\sigma)$		$2^{Val_{nil}}$	existing values including nil
(τ, stm)		$\Sigma_{\mathsf{loc}} \times Stm$	local configuration
ξ	$Thread$	Stack of $\Sigma_{\mathsf{loc}} \times Stm$	thread configuration
$\langle T, \sigma\rangle$		$2^{Thread} \times \Sigma$	global configuration
$[\![\_]\!]_{\mathcal{E}}$		$(\Sigma_{inst} \times \Sigma_{\mathsf{loc}}) \to (Exp \rightharpoonup Val_{nil})$	local evaluation function
$isfree$		$2^{Thread} \times \bigcup_c Val^c \to \mathsf{Bool}$	entering possible?
$started$		$2^{Thread} \times \bigcup_c Val^c \to \mathsf{Bool}$	thread of an object started?
$sync$		$\mathcal{C} \times \mathcal{M} \to \mathsf{Bool}$	m of c synchronized?
		Assertions (syntax)	
z	$LVar^t$		logical var's
e	$LExp$		local expressions
p, q	$LAss$		local assertions
E	$GExp$		global expressions
P, Q	$GAss$		global assertions
		Assertions (semantics)	
ω	Ω	$LVar \rightharpoonup Val_{nil}$	logical environment
$[\![\_]\!]_{\mathcal{L}}$		$(\Omega \times \Sigma_{inst} \times \Sigma_{\mathsf{loc}}) \to$ $(LExp \cup LAss \rightharpoonup Val_{nil})$	local evaluation
$[\![\_]\!]_{\mathcal{G}}$		$(\Omega \times \Sigma) \to (GExp \cup GAss \rightharpoonup Val_{nil})$	global evaluation

D Example

The following class implements interfaces for read and write access to some database. Several threads may concurrently read the database. Before entering the reading section the threads must log in for reading, which increases a counter named *readers* by one; after finishing reading, the threads log out by decreasing the counter.

Entering the critical section of writing is possible only if there are no threads currently reading, i.e., if the counter *readers* has the value 0. Since the methods for write access and for logging in for reading are synchronized, no threads can log in for reading or begin to write if another thread is currently writing. Thus we conclude that write access is mutually exclusive wrt. reading and writing, whereas concurrent reading is possible. The example is formulated in *Java*-syntax, which is slightly different from $Java_{MT}$.

```
class Resource{
  private int readers = 0;
  public void read(){
    login_read();
    // Critical section of reading
    logout_read();
  }
  private synchronized void login_read(){
    readers = readers + 1;
  }
  private void logout_read(){
    readers = readers − 1;
  }
  public synchronized void write(){
    while (readers != 0) {;}
    // Critical section of writing
  }
}
```

To define an inductive assertion network expressing the program properties mentioned above, we first have to transform the program. In the following transformation the type Id denotes the type $\mathsf{Object} \times \mathsf{Int}$ of the auxiliary formal parameter id; the type list Id is denoted by IdSequence. The operation "$\circ$" applied to a sequence of type IdSequence and an identity of type Id appends the given identity at the end of the sequence. The operator "$-$" applied again to a sequence of type IdSequence and an identity of type Id removes exactly one occurrence of the given identity from the sequence, if any, and lets the sequence untouched otherwise.

Besides the built-in transformation introducing the explicit communication statements with their bracketed sections and the augmentation with the built-in auxiliary variables, we introduce the auxiliary instance variable *reader_threads*, which we use to store the identities of all currently reading threads. For the sake of readability, we don't show the augmentation with the auxiliary variables callerobj, lock, started, and stable, since they don't occur in the annotation, and thus their values do not influence the inductiveness of the network.

```
class Resource{
  private int readers = 0;
  private IdSequence reader_threads;
  public void read(Id id){
    ⟨login_read(callee(id))⟩; ⟨receive⟩;
    // Critical section of reading
    ⟨logout_read(callee(id))⟩; ⟨receive⟩;
    ⟨return⟩;
  }
  private synchronized void login_read(Id id){
    readers, reader_threads = readers + 1, reader_threads ∘ id;
    ⟨return⟩;
  }
  private void logout_read(Id id){
    readers, reader_threads = readers − 1, reader_threads − id;
    ⟨return⟩;
  }
  public synchronized void write(Id id){
    while (readers != 0) {;}
    // Critical section of writing
    ⟨return⟩;
  }
}
```

Each reading thread is logged in, and consequently its identity is appended to the sequence *reader_threads*. Using the fact that a thread can remove only its *own* identity from the sequence *reader_threads*, we state as invariant that the identity of a thread is represented in *reader_threads* iff the thread is logged in for reading. Thus we observe that the value of *reader_threads* of each instance of the class Resource contains the identities of all threads that are currently logged in for reading. Its length equals the number of reading threads, i.e., $readers = |reader\_threads|$. Combining the above observations leads to the following annotation of the transformed program:

```
class Resource{                                      I = (readers = |reader_threads|)
  private int readers = 0;
  private IdSequence reader_threads;
  public void read(Id id){                           {I}
    ⟨login_read(callee(id))⟩; {I} ⟨receive⟩;         {callee(id) ∈ reader_threads ∧ I}
    // Critical section of reading                   {callee(id) ∈ reader_threads ∧ I}
    ⟨logout_read(callee(id))⟩; {I} ⟨receive⟩;        {I}
```

```
    ⟨return⟩;                                        {I}
  }
  private synchronized void login_read(Id id){       {I}
    readers, reader_threads = readers + 1, reader_threads ∘ id;
                                                     {id ∈ reader_threads ∧ I}
    ⟨return⟩;                                        {I}
  }
  private void logout_read(Id id){                   {id ∈ reader_threads ∧ I}
    readers, reader_threads = readers − 1, reader_threads − id;
                                                     {I}
    ⟨return⟩;                                        {I}
  }
  public synchronized void write(Id id){             {I}
    while (readers != 0) {;}                         {readers = 0 ∧ I}
    // Critical section of writing                   {readers = 0 ∧ I}
    ⟨return⟩;                                        {I}
  }
}
```

Next we show that the above transformed and annotated class definition satisfies the verification conditions. Initial and local correctness are straightforward. For the interference freedom test we have to show the invariance of assertions under the execution of the assignments in the methods login_read and logout_read. Invariance of the class invariant is straightforward. Invariance of the assertion $callee(id) \in reader\_threads \wedge I$ under appending an element to *reader_threads* in the method login_read is easy to see. The assertion is also invariant under the execution of the assignment in logout_read which removes an element from the sequence, since the assertion interleavable in the verification condition implies that the assertion describes a thread different from the one executing the assignment. I.e., $\neg\mathsf{samethread}(\mathsf{id}', \mathsf{id})$, and thus $\mathsf{callee}(\mathsf{id}') \neq \mathsf{id}$, where id' denotes the identity of the thread described by the assertion, and id is the identity of the thread executing the assignment. The same arguments apply to the invariance of the assertion $id \in reader\_threads \wedge I$. The assertion $readers = 0 \wedge I$ cannot interfere with the execution of any assignments: Since the methods write and login_read are both synchronized, the assertion interleavable applied to $readers = 0 \wedge I$ and the assignment in login_read evaluates to *false*, and thus we don't have to show interference freedom for this case. Finally, the assertion $readers = 0 \wedge I$ imply that the sequence *reader_threads* is empty, that contradicts to the precondition $id \in reader\_threads \wedge I$ of the assignment in the method logout_read. Thus the verification conditions of the interference freedom test are satisfied.

For the cooperation test we define the trivial global invariant $GI = \mathsf{true}$. The cooperation test for the invocation of login_read and for returning from logout_read are straightforward. The cooperation test defines the following condition for returning from the method login_read invoked by a thread executing the method read:

$$\begin{aligned}\omega,\sigma \models_{\mathcal{G}} & \ (\mathsf{id} \in \text{reader\_threads} \wedge I)[z', \mathsf{callee}(\mathsf{id})/\mathsf{this}, \mathsf{id}] \wedge I[z/\mathsf{this}] \wedge \\ & \quad \mathsf{this}[z/\mathsf{this}] = z' \wedge z{\neq}\mathsf{nil} \wedge z'{\neq}\mathsf{nil} \\ & \quad \rightarrow I[z', \mathsf{callee}(\mathsf{id})/\mathsf{this}, \mathsf{id}] \wedge (\mathsf{callee}(\mathsf{id}) \in \text{reader\_threads} \wedge I)[z/\mathsf{this}] \ ,\end{aligned}$$

i.e.,

$$\begin{aligned}\omega,\sigma \models_{\mathcal{G}} & \ \mathsf{callee}(\mathsf{id}) \in z'.\text{reader\_threads} \wedge I[z'/\mathsf{this}] \wedge I[z/\mathsf{this}] \wedge z = z' \\ & \quad \rightarrow I[z'/\mathsf{this}] \wedge \mathsf{callee}(\mathsf{id}) \in z.\text{reader\_threads} \wedge I[z/\mathsf{this}] \ ,\end{aligned}$$

whose satisfaction is easy to see. Similarly for the invocation of the method logout_read by a thread executing the method read we have to show that

$$\begin{aligned}\omega,\sigma \models_{\mathcal{G}} & \ (\mathsf{callee}(\mathsf{id}) \in \text{reader\_threads} \wedge I)[z/\mathsf{this}] \wedge I[z'/\mathsf{this}] \wedge \\ & \quad \mathsf{this}[z/\mathsf{this}] = z' \wedge z \neq \mathsf{nil} \wedge z' \neq \mathsf{nil} \\ & \quad \rightarrow I[z/\mathsf{this}] \wedge (\mathsf{id} \in \text{reader\_threads} \wedge I)[z', \mathsf{callee}(\mathsf{id})/\mathsf{this}, \mathsf{id}] \ ,\end{aligned}$$

i.e.,

$$\begin{aligned}\omega,\sigma \models_{\mathcal{G}} & \ \mathsf{callee}(\mathsf{id}) \in z.\text{reader\_threads} \wedge I[z/\mathsf{this}] \wedge I[z'/\mathsf{this}] \wedge z = z' \\ & \quad \rightarrow I[z/\mathsf{this}] \wedge \mathsf{callee}(\mathsf{id}) \in z'.\text{reader\_threads} \wedge I[z'/\mathsf{this}] \ ,\end{aligned}$$

whose validity is again straightforward.

This example shows that with our proof method we can prove properties of instances of a class without explicitly defining the whole program, i.e., the context of the class. The preconditions of the methods in the class define *assumptions* about the behavior of the context. The object properties of instances of the class are invariant, if these assumptions are satisfied for all invocations of instance methods by the context.