INSTITUT FÜR INFORMATIK UND PRAKTISCHE MATHEMATIK

LEHRSTUHL FÜR SOFTWARETECHNOLOGIE

A Hoare Logic for Monitors in Java

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

Bericht Nr. TR-ST-03-1
April 2003

CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

A Hoare Logic for Monitors in Java*

April 4, 2003

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

[1] Christian-Albrechts-University Kiel, Germany
[2] CWI Amsterdam, 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 safety-properties of multithreaded programs, we introduce in this paper a tool-supported *assertional proof method* 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.

* Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2).

Table of Contents

1 Introduction

The semantical foundations of *Java* [18] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [6, 38, 14]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sub-languages (see e.g. [24, 42, 35]). In [4] a sound and complete proof system is presented for *Java*'s reentrant multithreading concept. In this paper we

illustrate how information about other mechanisms can be incorporated in the proof system by means of *auxiliary variables* which describe the corresponding flow of control: We describe an extension of the proof system to *monitor synchronization*. 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 and *Java*'s monitor discipline.

The behavior of a *Java*$_{MT}$ program results from the concurrent execution of methods. To support a clean interface between internal and external object behavior, *Java*$_{MT}$ does not allow qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries. In order to capture program behavior 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) [43], properties of object-structures are described in terms of a navigation or dereferencing operator.

The assertional proof system for verifying safety properties of *Java*$_{MT}$ is formulated in terms of *proof outlines* [30], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [17, 21]. The satisfaction of the program properties specified by the assertions is guaranteed by the verification conditions of the proof system. The execution of a single method body in isolation is captured by standard *local correctness* conditions, using the local assertion language. Interference between concurrent method executions is covered by the *interference freedom test* [30, 28], formulated also in the local language. It has especially to accommodate for reentrant code and the specific synchronization mechanism. Possibly affecting more than one instance, communication and object creation is treated in the *cooperation test*, using the global language. The communication can take place within a single object or between different objects. As these cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules in [11] and in [28] for CSP.

Computer-support is given by the *Verger* (*"VERification condition GEneratoR"*) tool, taking a proof outline as input, and generating the corresponding verification conditions as output. We use the theorem prover PVS [31] to verify the conditions.

Overview This paper is organized as follows. Section 2 defines the syntax and semantics of *Java*$_{MT}$. After introducing the assertion language in Section 3, the main Section 4 presents the proof system, whose soundness and completeness is shown in Section 5. The proofs of the results are included in the appendix. The last Section 6 discusses related and future work.

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 user-defined run-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.* Each object has a *lock* which can be owned by at most one thread. Synchronized methods of an object can be invoked only by a thread which owns the lock of that object. If the thread does not own the lock, it has to wait until the lock gets free. If a thread owns the lock of an object, it can recursively invoke several synchronized methods of that object. This corresponds to the notion of reentrant monitors and eliminates the possibility that a single thread deadlocks itself on an object's synchronization barrier.

Besides mutual exclusion, using the lock-mechanism for synchronized methods, objects offer the methods wait, notify, and notifyAll as means to facilitate efficient thread coordination at the object boundary. A thread owning the lock of an object can block itself and free the lock by invoking wait on the given object. The blocked thread can be reactivated by another thread via the object's notify method; the reactivated thread must re-apply for the lock before it may continue its execution. The method notifyAll, finally, generalizes notify in that it notifies all threads blocked on the object.

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 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, 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. 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 (temporary) 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 $TVar$.

The abstract syntax is summarized in the Table 1. It slightly differs from the *Java* syntax: For example, we use := to build assignments instead of *Java*'s =, while we use = for equality instead of ==, and write conditional expressions in the form $\mathsf{if}\ \ e_1\, \mathsf{then}\ \ e_2\, \mathsf{else}\ \ e_3\, \mathsf{fi}$ instead of $(e_1 ? e_2 : e_3)$. Though we use the abstract syntax for the theoretical part of this work, the *Verger* tool supports *Java* syntax.

Basic constructs Besides using instance and local variables, *expressions* $exp \in Exp$ are built from this, null, and from subexpressions using the given operators. We will use e as typical element for expressions, and $Exp^t_{m,c}$ 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 start, run, wait, notify, and notifyAll. The sets Exp^t and $Exp_{m,c}$ are defined correspondingly. The expression this is used for self-reference within an object, and null is a constant representing an empty reference.

To support a clean interface between internal and external object behavior, *Java$_{MT}$* does not allow qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries.

As *statements* $stm \in Stm$, we allow assignments, object creation, method invocation, and standard control constructs like sequential composition, conditional statements, and iteration. We refer by $Stm_{m,c}$ to the set of statements in method m of class c, and write ϵ for the empty statement.

A *method* definition 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; return e_{ret}. 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 return e_{ret}, giving back the control and possibly a return value. 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* is defined by its name c and its methods, whose names are assumed to be distinct. As mentioned earlier, each class contains the methods start and run. These methods serve to launch a thread, where run contains the actual body of the thread, and start is a "wrapper" around run which allows the initiator of the thread to asynchronously continue after setting off the execution of the new thread. The run-method cannot be invoked directly.

A *program,* finally, is a collection of class definitions having different class names, where $class_{\mathsf{main}}$ defines by its run-method the entry point of the program execution. We call the body of the run-method of the main class the *main statement* of the program. In *Java*, the entry point of a program is given by the main-method of the main class. Relating the abstract syntax to that of *Java*, we assume that the main-method of *Java$_{MT}$* programs creates an instance of the main class, starts its thread, and terminates. The reason to make this restriction is, that *Java*'s main-method is static, but our proof system does not support static methods and variables.

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, 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.

Thread coordination Besides the user-definable methods, we consider the predefined ones start, wait, notify, and notifyAll. The start-method is not implemeted syntactically; see the next section for its semantics. The definition of the monitor methods wait, notify, and notifyAll uses the auxiliary statements !signal, !signal_all, ?signal, and return$_{getlock}$. All three methods are non-synchronized, parameterless, and without a return value. They can be used to block and reactivate threads at

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

$$
\begin{aligned}
exp &::= x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(exp, \ldots, exp) \\
exp_{ret} &::= \epsilon \mid exp \\
stm &::= x := exp \mid u := exp \mid u := \mathsf{new}^c \\
&\quad \mid u := exp.m(exp, \ldots, exp) \mid exp.m(exp, \ldots, exp) \\
&\quad \mid \epsilon \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm\ \mathsf{fi} \mid \mathsf{while}\ exp\ \mathsf{do}\ stm\ \mathsf{od} \ldots \\
modif &::= \mathsf{nsync} \mid \mathsf{sync} \\
meth &::= modif\, m(u, \ldots, u)\{\ stm; \mathsf{return}\ exp_{ret}\} \\
meth_{\mathsf{run}} &::= \mathsf{nsync}\ \mathsf{run}()\{\ stm; \mathsf{return}\ \} \\
class &::= c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{predef}\} \\
class_{\mathsf{main}} &::= class \\
prog &::= \langle class \ldots class\ class_{\mathsf{main}} \rangle
\end{aligned}
$$

Table 1. *Java$_{MT}$* abstract syntax

the object boundary, as informally described in Section 2.1. *Java*'s `Thread` class additionally support methods for suspending, resuming, and stopping a thread, but they are deprecated and thus not considered here.

$$
\begin{aligned}
meth_{predef} &::= meth_{\mathsf{start}}\ meth_{\mathsf{wait}}\ meth_{\mathsf{notify}}\ meth_{\mathsf{notifyAll}} \\
meth_{\mathsf{wait}} &::= \mathsf{nsync}\ \mathsf{wait}()\{\ ?\mathsf{signal}; \mathsf{return}_{getlock}\ \} \\
meth_{\mathsf{notify}} &::= \mathsf{nsync}\ \mathsf{notify}()\{\ !\mathsf{signal}\ ; \mathsf{return}\ \} \\
meth_{\mathsf{notifyAll}} &::= \mathsf{nsync}\ \mathsf{notifyAll}()\{\ !\mathsf{signal\_all}; \mathsf{return}\ \}
\end{aligned}
$$

Table 2. *Java$_{MT}$* predefined methods

Restrictions Besides the mentioned simplifications on the type system, we impose for technical reasons the following restrictions on the language: We require that method invocation and object creation statements contain only local variables, i.e., that none of the expressions $e_0, \ldots, e_n$ in a method invocation $e_0.m(e_1, \ldots, e_n)$ contains instance variables. Furthermore, formal parameters must not occur on the left-hand side of assignments. These restrictions imply 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.

Example 1. The following class implements a simple account, offering interfaces for deposit and withdraw. To assure that the balance x remains non-negative, the

withdraw method is synchronized; implicitly, the balance does not get decreased between the evaluation of $x >= i$ and the withdrawing. We will use this program to demonstrate the proof system: We show that for each class instance, under the assumption, that the methods deposit and withdraw are called with positive parameters only, the balance x has always a non-negative value.

```
public class Account{
    private int x;

    private void change_balance(int i){
        x = x+i;
    }

    public void deposit(int i){
        change_balance(i);
    }

    public synchronized void withdraw(int i){
        if (x>=i) { change_balance(-i); }
    }
}
```

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. Section 2.3.3 shows how states are represented in *PVS*.

2.3.1 States and configurations To give semantics to the expressions, we first fix the domains Val^t of the various types t. Let Val^{Int} and Val^{Bool} denote the set of integers and booleans, $Val^{\mathsf{list}\,t}$ be finite sequences over values from Val^t, and let $Val^{t_1 \times t_2}$ stand 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, $null^c \notin Val^c$ represents the value of null in the corresponding type. In general we will just write *null*, when c is clear from the context. We define Val^c_{null} as $Val^c \dot\cup \{null^c\}$, and correspondingly for compound types. The set of all possible non-nil values $\bigcup_t Val^t$ is written as Val, and Val_{null} denotes $\bigcup_t Val^t_{null}$.

Let $Init : Var \to Val_{null}$ be a function assigning the initial value of type t to each variable $y \in Var^t$, i.e., *null*, *false*, and 0 for class, boolean, and integer types, respectively, and analogously for compound types, where sequences are initially empty. We define $\mathsf{this} \notin Var$, such that the self-reference is not in the domain of *Init*.

The configuration of a program consists of all currently executing threads together with the set of existing objects and the values of their instance variables

(cf. Figure 1). Before formalizing the global configurations of a program, we define local states and local configurations. In the sequel we identify the occurrence of a statement in a program with the statement itself.

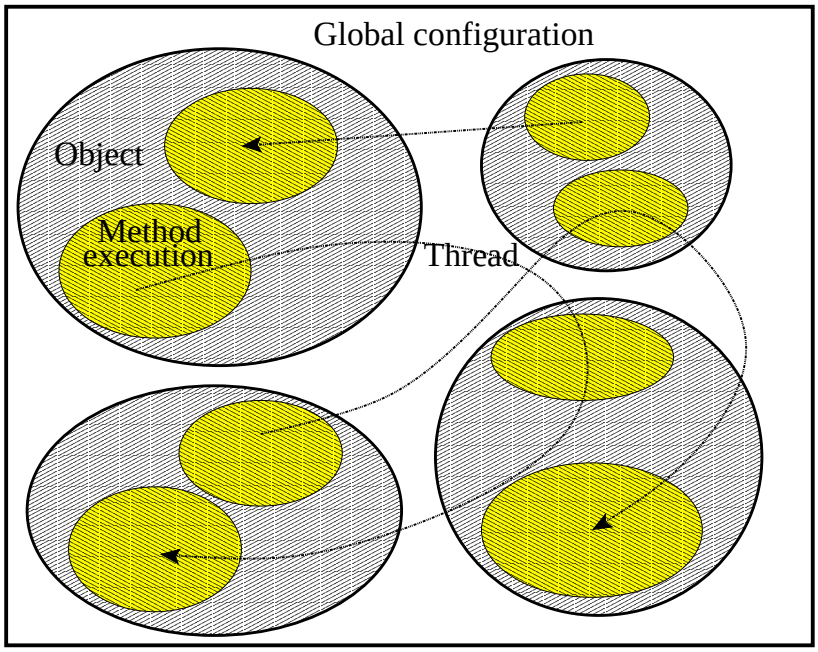

Fig. 1. Global configuration

A *local state* $\tau \in \Sigma_{loc}$ of a thread holds the values of its local variables and is modeled as a partial function of type $TVar \rightharpoonup Val_{null}$. We use the notation $\tau^{m,c}$ to refer to a local state of a thread executing method m of an instance of class c. The initial local state τ_{init} or $\tau_{init}^{m,c}$ assigns to each local variable u from its domain the value $Init(u)$.

A *local configuration* (α, τ, stm) of a thread executing within an object $\alpha \neq null$ specifies, in addition to its local state τ, its point of execution represented by the statement stm. A *thread configuration* ξ is a stack of local configurations $(\alpha_0, \tau_0, stm_0)(\alpha_1, \tau_1, stm_1) \ldots (\alpha_n, \tau_n, stm_n)$, representing the chain of method invocations of the given thread. We write $\xi \circ (\alpha, \tau, stm)$ for pushing a new local configuration onto the stack.

The state of an object is characterized by its *instance state* $\sigma_{inst} \in \Sigma_{inst}$ of type $IVar \dot\cup \{\mathsf{this}\} \rightharpoonup Val_{null}$ which assigns values to the self-reference this and to instance variables.[4] 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 $IVar_c \dot\cup \{\mathsf{this}\} \rightarrow Val_{null}$.

[4] In *Java*, this is a "final" instance variable, which for instance implies, it cannot be assigned to.

The initial instance state σ^{init}_{inst} or $\sigma^{c,init}_{inst}$ assigns a value from Val^c to this, and to each of its remaining instance variables x the value $Init(x)$. The semantics will maintain $\sigma^c_{inst}(\mathsf{this}) \in Val^c$ as invariant.

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 $Val^c(\sigma)$, and $Val^c_{null}(\sigma)$ is defined by $Val^c(\sigma) \dot\cup \{null^c\}$. For the built-in types Int and Bool we define $Val^t(\sigma)$ and $Val^t_{null}(\sigma)$, independently of σ, as the set of pre-existing values Val^{Int} and Val^{Bool}, respectively. For compound types, $Val^t(\sigma)$ and $Val^t_{null}(\sigma)$ are defined correspondingly. We refer to the set $\bigcup_t Val^t(\sigma)$ by $Val(\sigma)$; $Val_{null}(\sigma)$ denotes $\bigcup_t Val^t_{null}(\sigma)$. The instance state of an object $\alpha \in Val(\sigma)$ is given by $\sigma(\alpha)$ with the invariant property $\sigma(\alpha)(\mathsf{this}) = \alpha$. We say that objects $\alpha \in Val(\sigma)$ *exists* 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., $\sigma(\alpha)(x) \in Val_{null}(\sigma)$ for all $\alpha \in Val^c(\sigma)$. 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 (α, τ, stm) in $\langle T, \sigma\rangle$ refer only to existing object identities, i.e., $\alpha \in Val(\sigma)$ and $\tau(u) \in Val_{null}(\sigma)$ for all variables u from the domain of τ; again this will be an invariant of the operational semantics. In the following, we write $(\alpha, \tau, stm) \in T$ if there exists a local configuration (α, τ, stm) within one of the execution stacks of T.

Expressions $e \in Exp_{m,c}$ are evaluated with respect to an *instance local* state $(\sigma^c_{inst}, \tau^{m,c}) \in \Sigma_{inst} \times \Sigma_{loc}$. The semantic function $[\![\_]\!]_{\mathcal{E}} : (\Sigma_{inst} \times \Sigma_{loc}) \to (Exp \rightharpoonup Val_{null})$, shown in Table 3, evaluates in the context of an instance local state (σ_{inst}, τ) expressions containing 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; this evaluates to $\sigma_{inst}(\mathsf{this})$, and null has the *null*-reference as value, where compound expressions are evaluated by homomorphic lifting.

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

Table 3. Expression evaluation

We denote by $\tau[u \mapsto v]$ the local state which assigns the value v to u and agrees with τ on the values of all other variables. The semantic update $\sigma_{inst}[x \mapsto v]$ of in-

stance states is defined analogously. Correspondingly for global states, $\sigma[\alpha.x \mapsto v]$ denotes the global state which results from σ by assigning v to the instance variable x of object α. We use these operators analogously for simultaneously setting the values of vectors 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, $\sigma[\alpha \mapsto \sigma_{inst}]$ equals σ except on α; note that in case $\alpha \notin \mathit{Val}(\sigma)$, the operation extends the set of existing objects by α, which has its instance state initialized to σ_{inst}.

2.3.2 Operational semantics The operational semantics of *Java$_{MT}$* is given inductively by the rules of Tables 4 and 5 as transitions between global configurations. Before having a closer look at the semantical rules for the transition relation $\longrightarrow$, let us start by defining the starting point of a program. The initial configuration $\langle T_0, \sigma_0 \rangle$ of a program satisfies $\mathit{dom}(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma_{inst}^{c,init}[\mathsf{this} \mapsto \alpha]$, and $T_0 = \{(\alpha, \tau_{init}^{\mathsf{run},c}, \mathit{body}_{\mathsf{run},c})\}$, where c is the main class, and $\alpha \in \mathit{Val}^c$.

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 $\longrightarrow$. A local configuration $(\alpha, \tau, \mathit{stm}) \in T$ 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 $\langle T, \sigma \rangle \rightarrow \langle T', \sigma' \rangle$ executing *stm* in the local state τ and object α.

Assignments to instance or local variables update the corresponding state component, i.e., either the instance state or the local state (cf. rules Ass$_{inst}$ and Ass$_{loc}$). Object creation by $u := \mathsf{new}^c$ as shown in rule New creates a new object of type c with a fresh identity stored in the local variable u, and initializes its instance variables. Invoking a method extends the call chain by a new local configuration (cf. rule Call). We have a similar rule not shown in the table for the invocation of methods without return value. After initializing the local state and passing the parameters, the thread begins to execute the method body. The condition $\mathit{sync}(c, m) \rightarrow \neg \mathit{owns}(T, \beta)$ expresses that a synchronized method of an object can be invoked by a thread only if no other threads holds its lock, i.e., if the lock is free or if the executing thread already owns it.

Threads being blocked or waiting at an object, though, temporarily relinquish the lock. Formally, the wait set $\mathit{wait}(T, \alpha)$ of an object α is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?\mathsf{signal}; \mathit{stm})$. Analogously, we need the set $\mathit{notified}(T, \alpha)$ of threads that have been notified and trying to get hold of the lock again: It is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, \mathsf{return}_{\mathit{getlock}})$.

Thus a thread owns the lock of an object, if it currently executes some synchronized methods of that object, but not its wait-method. Formally, the pred-

icate $owns(\xi, \alpha)$ is true iff there exists a $(\alpha, \tau, stm) \in \xi$ with stm synchronized and $\xi \notin wait(\{\xi\}, \alpha) \cup notified(\{\xi\}, \alpha)$. The definition is used analogously for sets of threads. An invariant of the semantics is that at most one thread can own the lock of an object at a time, i.e., for all reachable $\langle T, \sigma\rangle$, for all ξ and ξ' in T and $\alpha \in Val(\sigma)$, $owns(\xi, \alpha)$ and $owns(\xi', \alpha)$ imply $\xi = \xi'$.

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.

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

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

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

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

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

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

Table 4. Operational semantics (1)

The remaining rules of Table 5 handle Java$_{MT}$'s methods for thread manipulation. The invocation of a start-method brings a new thread into being (cf. rule CALL$_{start}$), thereby initializing the first activation record of a 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 $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n) \in T$ such that $\alpha = \alpha_0$. Further invocations of the start-method are without effect (cf. rule CALL$_{start}^{skip}$).[5] A thread ends its lifespan by arriving at the end of its earliest local configura-

[5] In Java an exception is thrown if the thread is already started but not yet terminated.

tion (cf. rule RETURN$_{run}$), that is by returning from a run-method.[6] Note that, since the initial thread begins its execution in the initial object, according to the definition of the *started* predicate, the start-method of the initial object cannot be invoked.

The remaining three methods offer typical monitor synchronization mechanism at the object boundary, whose calls are described in rule CALL$_{monitor}$. In all three cases it is necessary that the caller owns the lock of the object in question. If not, the caller will deadlock, as, once devoid of the lock, the caller stops and will never obtain it. In contrast, the successful call of synchronized methods as formalized by rule CALL of Table 4 depends contra-positively on the non-ownership of the lock by the rest of the program, which of course changes if another thread gives it free again. In *Java*, invoking a monitor-method without owning the lock raises an exception, which terminates the culprit thread, but lets the rest of the program continue. In this sense, our model is faithful to the behavior in *Java*.

A thread can *block* itself on an object whose lock it owns by invoking the object's wait-method, thereby relinquishing the lock and placing itself into α's *wait set* (cf. rule CALL$_{monitor}$). In our formalization, this is indicated in that the thread is about to execute the statement ?signal after successful invocation of the wait-method. Remember that according to the predicate *owns* the thread releases the lock thereby.

After having put itself on ice, the thread awaits notification to be reactivated by another thread which invokes the notify-method of the object. The notifier must own the lock of the object in question. The !signal-statement in the above method thus reactivates a thread waiting for notification on the given object (cf. rule SIGNAL). It reactivates one of the blocked threads at least insofar as it is given the chance to re-apply for the lock: According to rule RETURN$_{wait}$, the receiver can continue after notification in executing return$_{getlock}$ only if the lock is free. Note that the notifier does not hand over the lock to the one being notified but continues to own it. This behavior is known as *signal-and-continue* monitor discipline [9].

If there are no threads waiting on the object, then the !signal of the notifier is without effect (rule SIGNAL$_{skip}$). The notifyAll-method generalizes notify in that all waiting threads are notified via the !signal_all-broadcast (cf. rule SIGNALALL). The effect of this statement is given by setting $signal(T, \alpha)$ as $\{\xi \circ (\beta, \tau, stm) \mid \xi \circ (\beta, \tau, stm) \in T \setminus wait(T, \alpha) \vee \xi \circ (\beta, \tau, ?\mathsf{signal}; stm) \in wait(T, \alpha)\}$.

2.3.3 Representation of states in *PVS* Before dealing with verification conditions, let us have a look how objects are represented in *PVS*. Besides a theory defining objects, two additional theories are generated for each class: One defining the reference type, and one specifying the state of class instances. In this way, the classes can use each other's type definition without mutual dependency.

[6] The worked-off local configuration (α, τ, ϵ) is kept in the global configuration to ensure that the thread of α cannot be started twice.

$$\frac{\beta = [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}} \in Val^c(\sigma) \qquad \neg started(T \cup \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \beta)}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, e.\mathsf{start}(); stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm), (\beta, \tau^{\mathsf{run},c}_{init}, body_{\mathsf{run},c})\}, \sigma\rangle}\ \textsc{Call}_{start}$$

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

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

$$\frac{\begin{array}{c} m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\} \\ \beta = [\![e]\!]^{\sigma(\alpha),\tau}_{\mathcal{E}} \in Val^c(\sigma) \qquad owns(\xi \circ (\alpha, \tau, e.m(); stm), \beta) \end{array}}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, e.m(); stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{receive}; stm) \circ (\beta, \tau^{m,c}_{init}, body_{m,c})\}, \sigma\rangle}\ \textsc{Call}_{monitor}$$

$$\frac{\neg owns(T, \beta)}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, \mathsf{receive}; stm) \circ (\beta, \tau', \mathsf{return}_{getlock})\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle}\ \textsc{Return}_{wait}$$

$$\frac{}{\begin{array}{l}\langle T \dot\cup \{\xi \circ (\alpha, \tau, !\mathsf{signal}; stm)\} \dot\cup \{\xi' \circ (\alpha, \tau', ?\mathsf{signal}; stm')\}, \sigma\rangle \longrightarrow \\ \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm)\} \dot\cup \{\xi' \circ (\alpha, \tau', stm')\}, \sigma\rangle\end{array}}\ \textsc{Signal}$$

$$\frac{wait(T, \alpha) = \emptyset}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, !\mathsf{signal}; stm)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle}\ \textsc{Signal}_{skip}$$

$$\frac{T' = signal(T, \alpha)}{\langle T \dot\cup \{\xi \circ (\alpha, \tau, !\mathsf{signal\_all}; stm)\}, \sigma\rangle \longrightarrow \langle T' \dot\cup \{\xi \circ (\alpha, \tau, stm)\}, \sigma\rangle}\ \textsc{SignalAll}$$

Table 5. Operational semantics (2)

Note that we do not define states in general, but specify an arbitrary single state. The type `Object` (which would in our case be the integers) is not represented, but the *PVS* definition specifies all objects existing in the given state. The verification conditions should be satisfied by all states. Instead of showing the quantification, the *PVS* implementation assures validity of the conditions for the given arbitrary state. This simple representation increases the proof automation.

Example 2. For the class

```
class c { int x; }
```

Verger generates the following type definitions:

```
Object: THEORY
BEGIN
    null: int
    Object_type: NONEMPTY_TYPE = {p:PRED[int] | p(null)}
        CONTAINING (LAMBDA (i:int): TRUE)
    Object?: Object_type
    Object: NONEMPTY_TYPE = (Object?) CONTAINING null
    class_name: NONEMPTY_TYPE = {cn:string | cn = "c"}
        CONTAINING "c"
    class: [Object->class_name]
END Object

c_type: THEORY
BEGIN
  IMPORTING Object
    c?: [Object->bool] = LAMBDA (i:Object): i=null OR class
        (i)="c"
    c: NONEMPTY_TYPE = (c?) CONTAINING null
    c_nn: TYPE = {i:c | i/=null}
END c_type

c: THEORY
BEGIN
  IMPORTING c_type
    x : [c_nn -> int] ...
END c
```

The instance state definitions are used in global conditions only. Local conditions define the instance variables of the given object locally in the theories containing the verification conditions. Also local variables are represented this way.

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. *Local* assertions describe instance local states, and are used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong. *Global* assertions describe the global state, i.e., a whole system of objects and their communication structure.

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 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 6 defines the syntax of the assertion language. For readability, we use in the theoretical part $\forall z.\, p$ and $\exists z.\, p$ for quantification; the *Verger* tool supports quantification in JML syntax: ($\backslash$*forall* $t\, z;\ p_1;\ p_2$) expresses that all values z of type t with the property p_1 satisfy p_2, where ($\backslash$*exists* $t\, z;\ p_1;\ p_2$) expresses that there is a value z of type t with the property p_1, which satisfies p_2.

Local expressions $exp_l \in LExp$ are expressions of the programming language possibly containing logical variables. The sets $LExp^t_{m,c}$, $LExp^t$, and $LExp_{m,c}$ are defined as for program expressions. In abuse of notation, we use e, $e' \ldots$ not only for program expressions of Table 1, but also for typical elements of 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, Bool, or compound types built from them. 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, null, 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 $Val_{null}(\sigma)$ in a global configuration $\langle T, \sigma \rangle$.

$exp_l ::= z \mid x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(exp_l, \ldots, exp_l)$		$e \in LExp$	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
$exp_g ::= z \mid \mathsf{null} \mid \mathsf{f}(exp_g, \ldots, exp_g) \mid exp_g.x$		$E \in GExp$	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 6. 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.

Logical variables are interpreted relative to a logical environment $\omega \in \Omega$, a partial function of type $LVar \rightharpoonup Val_{null}$, assigning values to logical variables. We denote by $\omega[\vec{z} \mapsto \vec{v}]$ the logical environment that assigns the values $\vec{v}$ to the variables $\vec{z}$, and agrees with ω on all other variables. Similarly to local and instance state updates, the occurrence of instance variables in $\vec{z}$ is without effect. For a logical environment ω and a global state σ we say that ω refers only to values existing in σ, if $\omega(z) \in Val_{null}(\sigma)$ for all $z \in dom(\omega)$. This property matches with the definition of quantification which ranges only over existing values and *null*, and with the fact that in reachable configurations local variables may refer only to existing values or to *null*.

The semantic function $[\![\_]\!]_{\mathcal{L}}$ of type $(\Omega \times \Sigma_{inst} \times \Sigma_{loc}) \to (LExp \cup LAss \rightharpoonup Val_{null})$ evaluates local expressions and assertions in the context of a logical environment ω and an instance local state (σ_{inst}, τ) (cf. Table 7). 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 $\exists z.\ p$ with $z \in LVar^t$ is evaluated to true in the logical environment ω and instance local state (σ_{inst}, τ) if and only if there exists a value $v \in Val^t$ such that p holds in the logical environment $\omega[z \mapsto v]$ and instance local state (σ_{inst}, τ), where for the type t of z only Int, Bool, or compund types built from them are 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.

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

Table 7. 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 $(\Omega \times \Sigma) \rightharpoonup (GExp \cup GAss \rightharpoonup Val_{null})$, shown in Table 8, 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 the domain of ω or σ. Logical variables, null, 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 $\exists z.\ P$ with $z \in LVar^t$ evaluates to true in the context of ω and σ if and only if P evaluates to true in the context of $\omega[z \mapsto v]$ and σ, for some value $v \in Val^t_{null}(\sigma)$. Note that quantification over objects ranges over the set of *existing* objects and *null*, only.

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 referring only to values existing in σ.

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

Table 8. Global evaluation

3.3 Substitution operations

The verification conditions defined in the next section involve three substitution operations: the local, the global, and the lifting substitution. The local substitution will be used to express the effect of assignments in local assertions. The global substitution is used similarly for global assertions. The lifting substitution, finally, allows to express local assertions in the global language.

The *local substitution* $p[\vec{e}/\vec{y}]$ is the standard capture-avoiding substitution, replacing 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 $\vec{y} := \vec{e}$. As for all three substitutions, the lemma is formulated for assertions, but the same property holds for expressions.

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

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p[\vec{e}/\vec{y}] \quad \textit{iff} \quad \omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]^{\omega,\sigma_{inst},\tau}_{\mathcal{L}}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]^{\omega,\sigma_{inst},\tau}_{\mathcal{L}}] \models_{\mathcal{L}} p\,.$$

The effect of assignments to instance variables is expressed on the global level by the *global 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 [8]:

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

The substitution is extended to global assertions homomorphically. We will also use the substitution $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences $\vec{y}$ possibly containing logical variables, whose semantics is defined by the simultaneous substitutions $[\vec{E}_x/z.\vec{x}]$ and $[\vec{E}_u/\vec{u}]$, where $\vec{x}$ and $\vec{u}$ are the sequences of the instance

and local variables of $\vec{y}$, and $\vec{E}_x$ and $\vec{E}_u$ the corresponding subsequences of $\vec{E}$; if only logical variables are substituted, we simply write $P[\vec{E}/\vec{u}]$. 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 and logical environments ω referring only to values existing in σ we have*

$$\omega, \sigma \models_{\mathcal{G}} P[\vec{E}/z.\vec{y}] \quad \textit{iff} \quad \omega', \sigma' \models_{\mathcal{G}} P \ ,$$

where $\omega' = \omega[\vec{y} \mapsto [\![\vec{E}]\!]^{\omega,\sigma}_{\mathcal{G}}]$ *and* $\sigma' = \sigma[[\![z]\!]^{\omega,\sigma}_{\mathcal{G}}.\vec{y} \mapsto [\![\vec{E}]\!]^{\omega,\sigma}_{\mathcal{G}}]$.

To express a local property p in the global assertion language, we define the substitution $p[z/\mathsf{this}]$ by simultaneously replacing in p all occurrences of the self-reference this by the logical variable z, which is assumed not to occur in p. For notational convenience we 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. We write $P(z)$ for $p[z/\mathsf{this}]$, and similarly for expressions. The substitution replaces all occurrences of the self-reference this by z, and transforms all occurrences of instance variables x into qualified references $z.x$. For unrestricted quantifications $(\exists z'.\ p)[z/\mathsf{this}]$ 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. The main cases of the substitution are defined as follows:

$$\begin{aligned}
\mathsf{this}[z/\mathsf{this}] &= z \\
x[z/\mathsf{this}] &= z.x \\
u[z/\mathsf{this}] &= u \\
(\exists z'.\ p)[z/\mathsf{this}] &= \exists z'.\ p[z/\mathsf{this}] \\
(\exists z' \in e.\ p)[z/\mathsf{this}] &= \exists z'.\ (z' \in e[z/\mathsf{this}] \wedge p[z/\mathsf{this}]) \\
(\exists z' \sqsubseteq e.\ p)[z/\mathsf{this}] &= \exists z'.\ (z' \sqsubseteq e[z/\mathsf{this}] \wedge p[z/\mathsf{this}]) \ ,
\end{aligned}$$

where $z \neq z'$ in the cases for existential quantification.

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

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 p be a local assertion containing local variables $\vec{u}$. If $\tau(\vec{u}) = \omega(\vec{u})$ and z a fresh logical variable, then*

$$\omega, \sigma \models_{\mathcal{G}} p[z/\mathsf{this}] \quad \textit{iff} \quad \omega, \sigma(\omega(z)), \tau \models_{\mathcal{L}} p \ .$$

4 The proof system

This section presents the assertional proof system to reason about Java_{MT} programs, formulated in terms of *proof outlines* [30, 16], i.e., where Hoare-style pre-

and postconditions [17, 21] are associated with each control point. The proof system has to accommodate for dynamic object creation, shared-variable concurrency, aliasing, method invocation, synchronization, and, especially, the monitor concept.

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

4.1 Proof outlines

4.1.1 Augmentation For a complete proof system it is necessary that the transition semantics of $Java_{MT}$ can be encoded in the assertion language. As the assertion language reasons about the local and global states, we have to augment the program with fresh auxiliary variables to represent information about the control points and stack structures within the local and global states. An augmentation extends a program by atomically executed multiple assignments $\vec{y} := \vec{e}$ to auxiliary variables, which we call *observations*. Furthermore, the observations have, in general, to be "attached" to statements they observe in a non-interleavable manner. This is syntactically represented using the special comment $/^*1\langle \vec{y} := \vec{e} \rangle^*/$ which attaches the observation to the preceding statement. As method calls $u := e_0.m(\vec{e})$ conceptually consist of two steps —handing over the parameters and reception of the result being stored in u— we need an additional form to observe atomically the reception of the return value. This form is represented as $/^*2\langle \vec{y} := \vec{e} \rangle^*/$. A stand-alone observation not attached to any statement is written as $/^*\langle \vec{y} := \vec{e} \rangle^*/$; it can be inserted at any point in the program. For readability, in the following we use the shortcuts $\langle stm \rangle$, $\langle stm \rangle^1$, and $\langle stm \rangle^2$ for $/^*\langle stm \rangle^*/$, $/^*1\langle stm \rangle^*/$, and $/^*2\langle stm \rangle^*/$.

Formally, assignments $y := e$ of the program can be extended to multiple assignments $y := e; \langle \vec{y}' := \vec{e}' \rangle^1$ by inserting additional assignments to distinct auxiliary variables $\vec{y}'$, which are executed simultaneously with the program statement. Besides the above extension of already occurring assignments, additional multiple assignments $\langle \vec{y}' := \vec{e}' \rangle$ to auxiliary variables can be inserted at any point in the program. Object creation can be observed by

$$u := \mathsf{new}^c; \quad \langle \vec{y} := \vec{e}; \rangle^1 \ .$$

Object creation and its observation are executed atomically in one computation step and in this order; the execution is not simultaneous, in order to allow to observe the identity of the new object.

An observation $\vec{y}_1 := \vec{e}_1$ of a call $u := e_0.m(\vec{e})$ and the observation $\vec{y}_4 := \vec{e}_4$ of the reception of a return value are indicated by

$$u := e_0.m(\vec{e}); \quad \langle \vec{y}_1 := \vec{e}_1; \rangle^1 \quad \langle \vec{y}_4 := \vec{e}_4; \rangle^2 \ .$$

Similarly for the callee, the observation $\vec{y}_2 := \vec{e}_2$ of the reception of a call invoking method m and the observation $\vec{y}_3 := \vec{e}_3$ of its return are indicated by extending the body $stm; \mathsf{return}\, u_{ret}$ of m to

$$\langle \vec{y}_2 := \vec{e}_2; \rangle^1 \quad stm; \quad \mathsf{return}\, u_{ret}; \quad \langle \vec{y}_3 := \vec{e}_3; \rangle^1 \ .$$

The augmentation does not influence the control flow of the program but enforce a particular scheduling policy. An assignment statement and its observation are executed simultaneously. Object creation and its observation are executed in a single computation step, in this order. For method call, communication, sender, and receiver observations are executed in a single computation step, in this order. I.e., they are executed atomically in the sense that they cannot be interleaved by other threads. Points which can be interleaved we call control points. Points between communication and its observation cannot be interleaved; we call them *auxiliary points*.

The proof system defines conditions which should hold for all simultaneously executed assignments. To collect these cases, in the following we will use multiple assigments $\vec{y} := \vec{e}$ to denote an assignment statement with its observation, an unobserved assignment, an alone-standing observation, or an observation of communication or object creation.

In [4] we allowed that in a self-communication both the caller and the callee may change the instance state. This complicated the proof system, especially the interference freedom test. Here we avoid this complication in that we require that the observation on the caller side in a self-communication may not change the values of instance variables. Formally, each observation of a method invocation statement $e_0.m(\vec{e})$ assigning a new value to an instance variable must have the form $x := \mathsf{if}\ e_0 = \mathsf{this}\ \mathsf{then}\ x\ \mathsf{else}\ e\ \mathsf{fi}$. Invoking the start-method by a self-call is specific in that, when the thread is already started, the caller is the only active entity (cf. rule $\textsc{Call}^{skip}_{start}$). In this case, it has to be the caller that updates the instance state; the corresponding observation has the form $x := \mathsf{if}\ e_0 = \mathsf{this} \wedge \neg\mathsf{started}\ \mathsf{then}\ x\ \mathsf{else}\ e\ \mathsf{fi}$.

Example 3. Extending an assignment $x := e$ to $x := e; \langle u := x; \rangle^1$ stores the value of x *prior* to the execution of $x := e$ in the auxiliary variable u.

Extending the assignment $x := e$ to $x := e; \langle u := x; \rangle$ stores the value of x in u *after* the execution of $x := e$.

Example 4. We can store the number objects created by an instance in an auxiliary instance variable n of type Int by extending each object creation statement $u := \mathsf{new}$ in the given class to $u := \mathsf{new}\ \langle n := n + 1; \rangle^1$.

Example 5. We extend Example 4 by additionally extending each call $u := e_0.m(\vec{e})$ with $m \neq \mathsf{start}$ in c to $u := e_0.m(\vec{e}); \langle m := n; \rangle^1 \langle m := n - m; \rangle^2$. Then the value of m after method call, but before return stores the number of objects created up to the call; after return, it stores the number of objects created during the method evaluation.

Example 6. Let k of type Int be an auxiliary instance variable of class c. We can count the number of local configurations executing in an instance of c by extending the body $stm; \mathsf{return}\ e_{ret}$ of each method in class c to $\langle k = k + 1; \rangle^1\ stm;\ \mathsf{return}\ e_{ret}; \langle k = k - 1; \rangle^1$

The above examples show how to count objects, local configurations in an object, etc. But those informations are not sufficient for a complete proof system: We have to be able to *identify* those entities. In the following we define some specific auxiliary variables, which we will use to formulate the verification conditions.They are automatically included in all augmentations; the user may not change their values. The built-in augmentation is not visible to the user, but the values of the built-in variables may be used in the augmentation and annotation.

An important point of the proof system is the identification of 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 an object-unique identification is needed, to detect local configurations in caller-callee relationship and reentrant calls.

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. For each method invocation, the callee thread identity is handed over in the auxiliary formal parameter thread of type Object. For readability, we use in the following the type Thread = Object for the domain of thread identities.

A local configuration is identified by the object in which it executes together with the value of its auxiliary local variable conf storing a unique object-internal identifier. Its uniqueness is assured by the auxiliary instance variable counter, incremented for each new local configuration in that object. The callee receives the "return address" as auxiliary formal parameter caller of type Object×Int×Thread, storing the identities of the caller object, the calling local configuration, and the caller thread. Note that the thread identities of caller and callee are the same in all cases but the invocation of a start-method. The run-method of the initial object is executed with the parameters $(\mathsf{thread}, \mathsf{caller}) = (\alpha_0, (null, 0, null))$, where α_0 is the initial object.

To capture mutual exclusion and the monitor discipline, the instance variable lock of Type Thread × Int stores the identity of the thread who owns the lock, if any, together with the number of synchronized calls in the call chain. Its initial value $(null, 0)$, for which we also write *free*, indicates, that the lock is free. We write thread(lock) to refer to the first component of the lock value, i.e., to the thread owning the lock. The instance variables wait and notified of Type list(Thread × Int) are the analogues of the *wait*- and *notified*-sets of the semantics and store the threads waiting at the monitor, respectively those having been notified. Besides the thread identity, the number of synchronized calls is stored. In other words, these variables remember the old lock-value prior to suspension which is restored when the thread becomes active again. The boolean instance variable started, finally, remembers whether the object's start-method has already been invoked. All auxiliary variables are initialized as usual.

In general, the specific auxiliary variables are needed to make the global predicates of the semantics expressible in the assertion language. That the variables and the predicates match will be shown in Section 5.1 as part of the soundness. Note that while object creation statements are observable, we do not introduce a specific augmentation as for the communication statements. The crucial predicate for object creation, the freshness-proviso, is already expressible in the global assertion language by existential quantification over existing objects. Therefore we do not need to prescribe a specific augmentation.

For the update of lists, which are represented in *PVS* by finite sequences `finseq[t]` of type `t`, we need the following functions, whose PVS definition is automatically generated by *Verger*: Given a sequence `s` of type `finseq[t]` and a value `e` of type `t`, the function `index` retrieves the index of an occurrence of `e` in `s`, if any, and gives `-1` otherwise. The function `choose` assigns to each non-empty sequence a non-negative integer smaller then the length of the sequence; for the empty sequence its value is `-1`. The expression `remove(s,i)` gives `s` without its `i`th element if $0 \leq i \leq |s|$, and returns `s` otherwise. The predicate $e \in s$ is syntactically represented by `includes(s,e)`. The function `append` appends an element at the end of a sequence, and finally `o` concatenates two sequences. The above functions are deterministic. The use of the specific auxiliary variables is illustrated by the following example, where `[:`t_1`,...,`t_n`:]` and `(:`e_1`,...,`e_n`:)` define product types and tuples, and `proj(s,i)` gives the `i`th tuple element.:

Example 7. For the class

```
public class Annotation extends Thread{

    void m1(){}

    synchronized void m2(){}

    public void run(){
        this.m1();
        this.start();
    }
}
```

Verger generates the following proof outline by extending the class with the built-in augmentation:

```
public class Annotation extends Thread {
  /*< finseq[[:Thread,int:]] wait; >*/
  /*< finseq[[:Thread,int:]] notified; >*/
  /*< boolean started; >*/
  /*< int counter; >*/
  /*< [:Thread,int:] lock; >*/

  void m1(Thread thread, [:Object,int,Thread:] caller) {
    /*< int conf; >*/
    /*1<conf = counter; counter = counter+1;>*/
    return;
  }
```

```
  synchronized void m2(Thread thread, [:Object,int,Thread
      :] caller) {
    /*< int conf; >*/
    /*1<conf = counter; counter = counter+1;
         lock = (:thread,proj(lock,2)+1:);>*/
    return;
    /*1<lock = (:proj(lock,2) == 1 ? null : proj(lock,1),proj(lock,2)-1:);>
        */
  }

  public void run(Thread thread, [:Object,int,Thread:]
      caller) {
    /*< int conf; >*/
    /*1<conf = counter; counter = counter+1; started = true;>*/
    this.m1(thread, (:this,conf,thread:));
    this.start(this, (:this,conf,thread:));
    return;
  }
}
```

The class is further extended with the specification of the monitor methods:

```
public void wait(Thread thread, [:Object,int,Thread:]
    caller) {
  /*< int conf; >*/
  /*1<conf = counter; counter = counter+1;
      wait = append(wait,lock); lock = (:null,0:);>*/
  return;
  /*1<lock = notified[get(notified,thread)];
      notified = remove(notified,get(notified,thread));>*/
}

public void notify(Thread thread, [:Object,int,Thread:]
    caller) {
  /*< int conf; >*/
  /*1<conf = counter; counter = counter+1;>*/
  /*<wait = remove(wait,choose(wait));
      notified = append(notified,wait[choose(wait)]);>*/
  return;
}

public void notifyAll(Thread thread, [:Object,int,Thread:]
    caller) {
  /*< int conf; >*/
  /*1<conf = counter; counter = counter+1;>*/
  /*<notified = o(notified,wait); wait = empty_seq();>*/
  return;
}
```

The user may additionally augment and annotate the monitor methods using special comments. Note that the statements of the monitor methods, generated by *Verger*, do not use the auxiliary statements !signal, !signal_all, and ?signal of the semantics. Instead we implement the wait and notify methods by means of the auxiliary instance variables wait and notified which represent the corresponding sets of the semantics. In the augmented wait-method both the waiting and the notified status of the executing thread are represented by a single control point.

The two statuses can be distinguished by the values of the wait and notified variables.

4.1.2 Annotation To specify invariant properties of the system, the augmented programs are *annotated* by attaching local assertions to each control and auxiliary point. In *Verger* syntax, assertions are special comments /*{p}*/, /*1{p}*/, etc., as shown in the example below. For readability, we also use the shortcuts {p}, {p}¹, etc. We use the triple notation $\{p\}\ stm\ \{q\}$ and write $pre(stm)$ and $post(stm)$ to refer to the pre- and the post-condition of a statement. For assertions at auxiliary points we use the following notation: The annotation

$$\{p_0\} \quad u := \mathsf{new}\ c; \quad \{p_1\}^1 \quad \langle y := e; \rangle^1 \quad \{p_2\}$$

of an object creation statement specifies p_0 and p_2 as pre- and postconditions, where p_1 at the auxiliary point should hold directly after object creation but before observation. The annotation

$$\begin{array}{llll} \{p_0\} \quad u := e_0.m(\vec{e}); & \{p_1\}^1 & \langle y_1 := e_1; \rangle^1 & \{p_2\}^2 \\ & \{p_3\}^3 & \langle y_4 := e_4; \rangle^2 & \{p_4\} \end{array}$$

assigns p_0 and p_4 as pre- and postconditions to the method invocation; p_1 is assumed to hold directly after method call, but prior to its observation; p_2 describes the control point of the caller after method call and before return; finally, p_3 specifies the state directly after return but before its observation. The annotation of method bodies stm; return e; is as follows:

$$\begin{array}{lllll} & \{p_1\}^1 & \langle y_2 := e_2; \rangle^1 & \{p_2\} \\ stm; & & & \{p_3\} \\ \mathsf{return}\ e; & \{p_4\}^1 & \langle y_3 := e_3; \rangle^1 & \{p_5\} \end{array}$$

The callee postcondition of the method call is p_2; the callee pre- and postconditions of return are p_3 and p_5. The assertions p_1/p_4 specify the states of the callee between method call/return and its observation.

Besides pre- and postconditions, for each class c, the annotation defines a local assertion I_c called *class invariant*, which may refer only to the instance variables of c, and which expresses invariant properties of instances of the class.[7] We require that $pre(body_{m,c}) = I_c$ for all methods. Finally, a global assertion GI called the *global invariant* specifies properties of communication between objects. As such, it should be invariant under object-internal computation. For that reason, we require that for all qualified references $E.x$ in GI with E of type c, all assignments to x in class c occur in the observations of communication or object creation. Note that the global invariant is not affected by the object-internal monitor signaling mechanism. We require that in the annotation no free logical variables occur. An augmented and annotated program is called a *proof outline*.

[7] The notion of class invariant commonly used for sequential object-oriented languages differs from our notion: In a sequential setting, it is sufficient that the class invariant holds initially and is preserved by whole method calls, but not necessarily in between.

Example 8. The following proof outline annotates the class of Example 1. *Verger* allows partial annotation: unspecified assertions are true by definition. Note the way how we define the functions *owns* and *free_for* and use them in the assertions. Note furthermore how the wait method is annotated, expressing that it is not called, and thus the assertions do not have to be invariant under its built-in augmentation.

```
1  //function definitions
2  /*{ boolean owns(Thread thread, [:Thread,int:] lock) =
3          thread!=null && thread==proj(lock,1) }*/
4  /*{ boolean free_for(Thread thread, [:Thread,int:] lock) =
5          thread!=null && (thread==proj(lock,1) || proj(lock,1)==null) }*/

7  public class Account{
8      private int x;

10     /*{ x>=0 }*/ //class invariant

12     /*[ wait ]*/ /*1{ false }*/ /*{ false }*/
13                  /*< return; >*/ /*1{ false }*/ /*[]*/

15     private void change_balance(int i){
16         /*{ i>0 || (x+i>=0 && owns(thread,lock)) }*/
17         x = x+i;
18         /*{ i>0 || owns(thread,lock) }*/
19     }

21     public void deposit(int i){
22         /*{i>0}*/
23         change_balance(i);
24     }

26     public synchronized void withdraw(int i){
27         /*1{ free_for(thread,lock) }*/
28         /*{ i>0 && owns(thread,lock) }*/
29         if (x>=i) {
30             /*{ x>=i && i>0 && owns(thread,lock) }*/
31             change_balance(-i);
32             /*2{ i>0 }*/
33             /*{ owns(thread,lock) }*/
34         }
35         return;
36         /*1{ owns(thread,lock) }*/
37     }
38 }
```

All verification conditions for the above proof outline, as introduced in the following section, are proven by PVS automatically, using the `grind` strategy.

4.2 Verification conditions

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 (α, τ, stm) in T, the precondition of the statement *stm* is satisfied and the class invariants and the global invariant hold. 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, the class invariant of the initial object, 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 can also be influenced by other threads executing in the same object since they are sharing the same instance state. The corresponding verification conditions are formalized in the *interference freedom test*.

We define cooperations tests for communication and object creation, but we have no cooperation test for notification. As mentioned earlier, notification takes place within a single object, and thus its effect can be captured by a single assignment to the auxiliary variables wait and notified executed by the notifier. Invariance for the notifying thread is covered by the local correctness conditions, whereas preservation of the assertions for the notified partners is assured by 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 the result of communication and object creation 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 list some notation. Let Init be a syntactical operator with interpretation *Init* (cf. page 8). Given $IVar_c$ as the set of instance variables of class c without the self-reference, and $z \in LVar^c$, then $\mathsf{InitState}(z)$ denotes the global assertion $z \neq \mathsf{null} \wedge \bigwedge_{x \in IVar_c} z.x = \mathsf{Init}(x)$, expressing that the object denoted by z is in its initial instance state.

For readability, in the following definitions we will use the notation $p \circ f$ with $f = [\vec{e}/\vec{y}]$ for the substitution $p[\vec{e}/\vec{y}]$; we use a similar notation for global assertions. Note that the substitution binds stronger than the logical operators.

Finally, arguing about two different local configurations makes it necessary to distinguish between their local variables, since they may have the same names; in such cases we will rename the local variables in one of the local states. We use primed assertions p' to denote the given assertion p with every local variable u replaced by a fresh one u', and correspondingly for expressions.

4.2.1 Initial correctness A proof outline is *initially correct*, if the precondition of the main statement, the class invariant of the initial object, and the global invariant are satisfied initially, i.e., in the initial global configuration after

the execution of the callee observation at the beginning of the main statement. Furthermore, the precondition of the observation should be satisfied prior to its execution.[8]

Definition 1 (Initial correctness). *A proof outline is* initially correct, *if*

$$\models_{\mathcal{G}} \forall z.\ \big(\mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z = z') \rightarrow \tag{1}$$

$$P_2(z) \circ f_{init} \wedge (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init}\big)\ , \tag{2}$$

where $\{p_2\}^1\,\langle \vec{y}_2 := \vec{e}_2\rangle^1\,\{p_3\}$ *stm is the body and* $\vec{v}$ *the local variables of the* run*-method of the main class, I is the class invariant of the main class, z is of the type of the main class, and* $z' \in LVar^{\mathsf{Object}}$*. Furthermore,*

$$f_{init} = [z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]\ ,\ \textit{and}$$
$$f_{obs} = [\vec{E}_2(z)/z.\vec{y}_2]\,.$$

Example 9. For the proof outline

```
//global invariant
/*{(\exists Initial z1; z1!=null; (\forall Initial z2; z2!=null; z1==z2))}*/

public class Initial extends Thread{
     int x;

     //class invariant
     /*{ started }*/

     public static void main(String[] args){
          Initial obj;
          obj = new Initial();
          obj.start();
     }

     public void run(){
          int v;
          /*< int u; >*/
          /*1{ u==0 && v==0 && x==0 }*/ //precondition of observation
          /*1< u = 1; >*/               //observation of call
          /*{ u==1 && v==0 && x==0 }*/  //postcondition of observation
     }
}
```

the following initial condition is generated:

```
FORALL (z:Initial)  :
(Initial.init(z) AND
(FORALL (obj:Object)  : (obj=null OR z=obj))) IMPLIES
%precondition of observation:
((0=0 AND 0=0 AND Initial.x(z)=0) AND
%global invariant:
(EXISTS (z1:Initial)  : z1/=null AND
FORALL (z2:Initial)  : (z2/=null IMPLIES z1=z2)) AND
```

[8] We need this condition in the interference freedom test to show invariance of assertions under the execution of the observation.

```
%postcondition of observation:
(1=1 AND 0=0 AND Initial.x(z)=0) AND
%class invariant:
true)
```

where the `init` function in theory `Initial` is defined by

```
init(o:Initial): bool = (o/=null AND Initial.x(o)=0 AND
    Initial.started(o)=false AND ...
```

4.2.2 Local correctness A proof outline is *locally correct*, if the usual verification conditions [10] for standard sequential constructs hold. Especially, the precondition of an ordinary assignment, as given in the proof outline, must imply its postcondition after the execution of the assignment (cf. Equation (3)). We can additionally use validity of the class invariant, whose invariance is assured by the interference freedom test. Note that using the class invariant as an antecedent would not be necessary for a minimal proof system, since the class invariant itself can be stated in the local assertions, also. However, it allow less annotation. In case of notification, local correctness covers also invariance for the notifying thread, as the effect of notification is captured by an auxiliary assignment.

Definition 2 (Local correctness: Assignment). *A proof outline is* locally correct*, if for all multiple assignments* $\{p_1\}\ \vec{y} := \vec{e}\ \{p_2\}$ *in class c, which is not the observation of communication or object creation,*

$$\models_{\mathcal{L}} p_1 \wedge I_c \rightarrow p_2 \circ f_{ass}\ , \tag{3}$$

with $f_{ass} = [\vec{e}/\vec{y}]$.

The conditions for loops and conditional statements are similar. Note that we have no local verification conditions for observations of communication and object creation. The postconditions of such statements express *assumptions* about the communicated values. These assumptions will be verified in the *cooperation test*.

Example 10. For the proof outline of Example 9 no local conditions are generated, since the only assignment observes communication.

Example 11. The proof outline

```
class Local{
    public void m(){
        int y;
        y = 0;
        /*{ y<=10 }*/
        while (y<10) {
            /*{ y<10 }*/
            y = y+1;
            /*{ y<=10 }*/
        }
        /*{ y==10 }*/
    }
}
```

specifies, that after the while-loop, the variable y has the value 10. The following local conditions are generated:

```
%local condition for the assignment y:=0
m_0 : LEMMA (0<=10)
% local condition for entering the loop
m_1 : LEMMA (y<=10 AND y<10) IMPLIES (y<10)
% local condition for exiting the loop
m_2 : LEMMA (y<=10 AND (NOT (y<10))) IMPLIES (y=10)
% local condition for jumping back
m_3 : LEMMA y<=10 IMPLIES y<=10
% local condition for the assignment y:=y+1
m_4 : LEMMA y<10 IMPLIES y+1<=10
```

Example 12. The following proof outline is a producer-consumer example. Instances of the WaitSynch class have a memory `obj` storing a single value. A boolean variable `written`, having the initial value *false*, remembers, if the memory is written but not yet read. If the memory is written but not yet read, producer threads, executing the synchronized `put` method, suspend themselfs and give the lock free by invoking the object's wait-method. If a consumer thread, executing the `get`-method, reads the memory, it notifies a waiting thread, and returns. Now a producer thread can have access to write the memory. After writing, it notifies a thread, and returns. However, it can happen, that a producer notifies a producer, in which case the notified thread should not have write access. In this case the notified thread notifies again and suspend itself, until the memory got read. The `get` method executed by consumer threads work similarly.

We show a very simple annotation satisfying the verification conditions, which already shows correctness of the requirement, that producers write only if the memory is not yet written at all or already read, and consumers read only if the memory contains a new, not yet read value.[9]

```
public class WaitSynch extends Thread{

    DT obj;
    boolean written;

    public synchronized DT get(){
        DT result;
        while (!written){
            try { this.wait(); }catch (InterruptedException
                e){}
            if (!written){ this.notify(); }
        }
        /*{ written }*/
        result = obj;
        written = false;
        this.notify();
        return result;
    }
```

[9] We don't handle exceptions in $Java_{MT}$. However, in order to call the wait-method, we must syntactically catch InterruptedExceptions. But, since we don't support the interrupt method, it cannot be thrown.

```
    public synchronized void put(DT value){
        while (written) {
            try { this.wait(); }catch (InterruptedException
                e){}
            if (written){ this.notify(); }
        }
        /*{ !written  }*/
        obj = value;
        written = true;
        this.notify();
        return;
    }
}

class DT{}
```

For the above example the following local conditions are generated, expressing that after the while-loop, its condition is false:

```
%local condition for exiting the loop in get
get_0 : LEMMA (NOT (NOT written)) IMPLIES written
%local condition for exiting the loop in put
put_0 : LEMMA (NOT written) IMPLIES (NOT written)
```

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

4.2.3 The interference freedom test Besides invariance of the class invariants, the interference freedom test shows that local assertions at control points are invariant under computation steps in which they are not involved.

Note that assertions at auxiliary points do not have to be invariant, since these points are not interleavable. Furthermore, the interference freedom test does not treat the global invariant: Inductivity for the global invariant is covered by the cooperation test.

Since we disallow qualified references to instance variables in $Java_{MT}$, we only have to deal with the invariance under execution within the *same* object. Affecting only local variables, communication and object creation do not change the instance states of the executing objects. Thus we only have to take assignments into account. In the following, let p be an assertion at a control point, and $\vec{y} := \vec{e}$ a multiple assignment occurring in the same class, which can be an assignment statement with its observation, an unobserved assignment, an alone-standing observation, or the observation of communication or object creation.

Synchronized methods of a single object can be executed concurrently only if one of the corresponding local configurations is waiting for return: If the executing threads are different, then one of the threads is in the *wait* or *notified* set of the object; otherwise, both executing local configurations are in the same call chain. Thus we assume that either not both the assignment and the assertion

occur in a synchronized method, or the assertion is at a control point waiting for return.[10]

To avoid name clashes, we replace the local variables u in p by fresh ones u' resulting in p'. Using the specific auxiliary variables, we next formalize the conditions when p has to be invariant under the assignment, namely if $\vec{y} := \vec{e}$ is executed independently of p.

If p and $\vec{y} := \vec{e}$ belong to the *same* thread, expressed by $\mathsf{thread} = \mathsf{thread}'$, the only assertions endangered are those at control points waiting for a return value earlier in the current execution stack. For example, for the situation in Figure 2, the assertion $p7$ is at a control point waiting for return, and it has to be invariant under the execution of the assignment. In other words, an assignment belonging to a *reentrant* code segment can affect properties of a local configuration, whose execution is suspended earlier in the same call chain. Invariance of a local con-

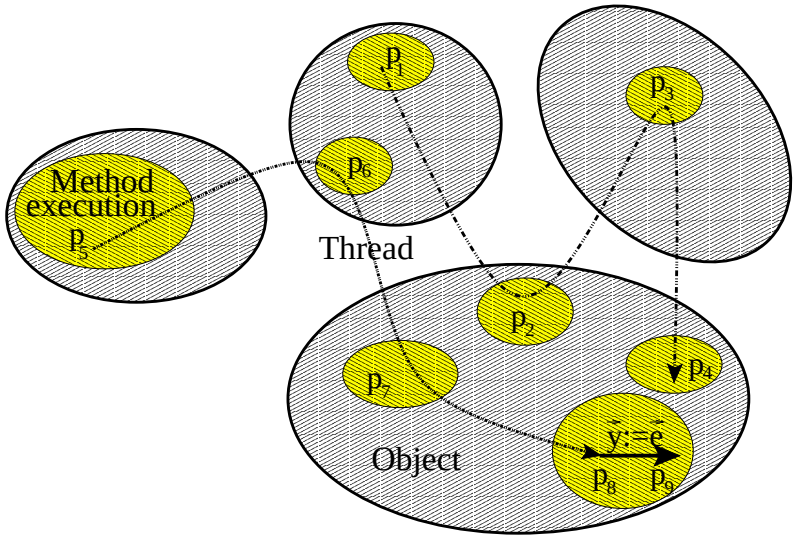

Fig. 2. Interference between threads

figuration under its own execution, however, need not be considered, expressed by requiring $\mathsf{conf} \neq \mathsf{conf}'$. Referring again to the above constallation, $p8$ do not have to be invariant under the assignment. Neither can the assertion interfere with the observation of the *matching* return statement, because communicating partners execute simultaneously. Applied to the above example, if the assignment oberves return, then $p7$ do not have to be invariant under the assignment. This requirement can be expressed by using the auxiliary variable caller, whose value identifies the caller local configuration, and by requiring that this identity

[10] This restriction is not necessary for a minimal proof system, but reduces the number of verification conditions.

is not identical to that of p, i.e., that $\mathsf{this} \neq \mathsf{proj}(\mathsf{caller}, 1) \vee \mathsf{conf}' \neq \mathsf{proj}(\mathsf{caller}, 2)$. Collecting the above observations, we define $\mathsf{waits\_for\_ret}(p, \vec{y} := \vec{e})$ by

- $\mathsf{conf}' \neq \mathsf{conf}$, if p is at a control point waiting for return and $\vec{y} := \vec{e}$ is not the observation of a return statement;
- $\mathsf{conf}' \neq \mathsf{conf} \wedge (\mathsf{this} \neq \mathsf{proj}(\mathsf{caller}, 1) \vee \mathsf{conf}' \neq \mathsf{proj}(\mathsf{caller}, 2))$ if p is at a control point waiting for return and $\vec{y} := \vec{e}$ is the observation of a return statement;
- false, otherwise.

If the assertion and the assignment belong to *different* threads, interference freedom must be shown in any case except for the self-invocation of the start-method: The precondition of such a method invocation cannot interfere with the corresponding observation of the callee. For the above example it means, that both $p2$ and $p4$ have to be shown invariant. To describe this setting, we define $\mathsf{self\_start}(p, \vec{y} := \vec{e})$ by $\mathsf{caller} = (: \mathsf{this}, \mathsf{conf}', \mathsf{thread}' :)$ iff p is the precondition of a method invocation $e_0.\mathsf{start}(\vec{e})$ and the assignment is the callee observation at the beginning of the run-method, and by false otherwise.

Using the above assertions, we define for assertions p at control points and assignments $\vec{y} := \vec{e}$ in the same class:

$$\begin{aligned}\mathsf{interleavable}(p, \vec{y} := \vec{e}) := {} & \mathsf{thread} = \mathsf{thread}' \rightarrow \mathsf{waits\_for\_ret}(p, \vec{y} := \vec{e}) \wedge \\ & \mathsf{thread} \neq \mathsf{thread}' \rightarrow \neg \mathsf{self\_start}(p, \vec{y} := \vec{e})\,.\end{aligned}$$

The interference freedom test can now be formulated as follows.

Definition 3 (Interference freedom). *A proof outline is* interference free, *if for all classes c with class invariant I, and for all multiple assignments $\vec{y} := \vec{e}$ with precondition p in c,*

$$\models_{\mathcal{L}} p \wedge I \rightarrow I \circ f_{ass}\ , \tag{4}$$

with $f_{ass} = [\vec{e}/\vec{y}]$. Furthermore, for all assertions q at control points in c, such that either not both p and q occur in a synchronized method, or q is at a control point waiting for return,

$$\models_{\mathcal{L}} p \wedge q' \wedge I \wedge \mathsf{interleavable}(q, \vec{y} := \vec{e}) \rightarrow q' \circ f_{ass}\ . \tag{5}$$

Note that including the class invariant as an antecedent in the second part of the definition is not necessary for a minimal proof system. However, it allows simpler annotations: Invariant instance properties can be expressed in the class invariant, and do not have to be included in the definition of local properties of threads.

Especially for notification, we require also invariance of the assertions for the notified thread. We do so, as notification is described by an auxiliary assignment executed by the notifier. That means, both the waiting and the notified status of the executing thread are represented by a single control point in the wait-method. The two statuses can be distinguished by the values of the wait and notified

variables. The invariance of the precondition of the return statement in the wait-method under the assignment in the notify-method represents the notification process, whereas invariance of that assertion over assignments changing the lock represents the synchronization mechanism. Information about the lock variable will be imported from the cooperation test as this information depends on the global behavior.

In the following we apply the interference freedom test to a few examples using *Verger*. Renaming is implemented by extending the name of each local variable of the local configuration executing the assignment with _1, where the names of local variables in the assertion get extended with _2; the names of instance variables get the extension `_inst`.

Verger does not generate conditions for trivial cases, for example if the assertion is true by definition, or if the substitution does not change the assertion.

Example 13. For the proof outline of Example 11 no interference freedom conditions are generated. Though the method can be executed concurrently, the assertions refer to local variables only. Due to the renaming mechanism, assigning a new value to the local variable y of a thread does not affect the assertions describing the local variable y of another thread, executing the same method.

Example 14. For the proof outline of Example 12 no interference freedom conditions are generated, since both methods are synchronized, and none of the assertions are at a control point waiting for a return value.

Example 15. Invariance of the class invariant of the proof outline of Example 8 is assured by the condition

```
%precondition assignment
((i_1>0 OR (x_inst+i_1>=0 AND owns(thread_1,lock_inst)))
    AND
%class invariant
x_inst>=0)
%class invariant after execution
IMPLIES (x_inst+i_1>=0)
```

generated for the only assignment (17), which changes the balance `x`. That (30) is invariant under the same assignment, is assured by the condition

```
%preconditions assignment
((i_1>0 OR (x_inst+i_1>=0 AND owns(thread_1,lock_inst)))
    AND
%assertion
x_inst>=i_2 AND i_2>0 AND owns(thread_2,lock_inst) AND
%class invariant
x_inst >= 0 AND
%interleavable
(thread_1=thread_2 IMPLIES false) AND
(thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(x_inst+i_1>=i_2 AND i_2>0 AND owns(thread_2,lock_inst))
```

If i_1>0, then x_inst>=i_2 implies x_inst+i_1>=i_2, and the condition is satisfied. This case corresponds to the concurrent execution of the methods withdraw and deposit. Otherwise, owns(thread_1,lock_inst), owns(thread_2,lock_inst), and thread_1/=thread_2 lead to a contradiction. This case corresponds to the concurrent execution of withdraw, which is not possible. The case that two threads are concurrently executing the change_balance method, is covered by the following condition, showing that (16) is invariant under (17):

```
%precondition assignment
((i_1>0 OR (x_inst+i_1>=0 AND owns(thread_1,lock_inst)))
    AND
%assertion
(i_2>0 OR (x_inst+i_2>=0 AND owns(thread_2,lock_inst))) AND
%class invariant
x_inst>=0 AND
%interleavable
(thread_1=thread_2 IMPLIES false) AND
(thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(i_2>0 OR (x_inst+i_1+i_2>=0 AND owns(thread_2,lock_inst)))
```

The remaining conditions are all generated for invariance under changing the lock value. We have 5 assertions at control points, which must be invariant under entering and exiting the wait method. However, since we've expressed by the annotation of the wait-method, that it is not invoked, the left-hand-side of the generated conditions is false.

The only remaining assignments changing the lock value are the observations at the beginning and at the end of the synchronized withdraw method, which do not have to be invariant under its own execution. Thus only the assertions (16) and (18) in change_balance have to be shown invariant (4 conditions). For invariance of (16) under entering the withdraw method we get the condition

```
%precondition assignment
(free_for(thread_1,lock_inst) AND
%assertion
(i_2>0 OR (x_inst+i_2>=0 AND owns(thread_2,lock_inst))) AND
%class invariant
x_inst>=0 AND
%interleavable
(thread_1=thread_2 IMPLIES false) AND
(thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(i_2>0 OR (x_inst+i_2>=0 AND owns(thread_2,(thread_1,(
    PROJ_2(lock_inst)+1)))))
```

Similarly for (18):

```
%precondition assignment
(free_for(thread_1,lock_inst) AND
%assertion
(i_2>0 OR owns(thread_2,lock_inst)) AND
%class invariant
x_inst>=0 AND
%interleavable
```

```
(thread_1=thread_2 IMPLIES false) AND
(thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(i_2>0 OR owns(thread_2,(thread_1,(PROJ_2(lock_inst)+1))))
```

Note that the predicates `free_for(thread_1,lock_inst)`, `owns(thread_2,lock_inst)`, and `thread_1/=thread_2` together lead to a contradiction. This corresponds to the fact, that if the `change_balance`-method was called from the `withdraw`-method, then no threads can enter the synchronized `withdraw`-method, since the first thread owns the lock. Finally, invariance of (16) under exiting `withdraw` is assured by

```
%precondition assignment
(owns(thread_1,lock_inst) AND
%assertion
(i_2>0 OR (x_inst+i_2>= 0 AND owns(thread_2,lock_inst)))
    AND
%class invariant
x_inst>=0 AND
%interleavable
(thread_1=thread_2 IMPLIES false) AND
(thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(i_2>0 OR (x_inst+i_2>=0 AND owns(thread_2,(IF (PROJ_2(
    lock_inst)=1) THEN null ELSE PROJ_1(lock_inst) ENDIF,(
    PROJ_2(lock_inst)-1)))))
```

and for (18) we have

```
%precondition assignment
(owns(thread_1,lock_inst) AND
%assertion
(i_2>0 OR owns(thread_2,lock_inst)) AND
%class invariant
x_inst>=0 AND
%interleavable
(thread_1=thread_2 IMPLIES false) AND
(thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(i_2>0 OR owns(thread_2,(IF (PROJ_2(lock_inst)=1) THEN null
     ELSE PROJ_1(lock_inst) ENDIF,(PROJ_2(lock_inst)-1))))
```

In these cases, similarly to invariance under entering the `withdraw` method, the predicates `owns(thread_1,lock_inst)`, `owns(thread_2,lock_inst)`, and `thread_1/=thread_2` together lead to a contradiction.

Example 16. The following example illustrates properties of the wait-method. All conditions generated for the proof outline are proven in PVS.

```
1 /*{ boolean owns(Thread thread, [:Thread,int:] lock) =
2         thread!=null && proj(lock,1)==thread }*/
3 /*{ boolean not_owns(Thread thread, [:Thread,int:] lock) =
4         thread!=null && proj(lock,1)!=thread }*/
5 /*{ boolean free_for(Thread thread, [:Thread,int:] lock) =
6        thread!=null && (thread==proj(lock,1) || lock==(:null,0:)) }*/
7 /*{ boolean disjunct(finseq[[:Thread,int:]] x) =
```

```
8         (\forall int i,j; 0<=i && 0<=j && i<length(x)&& j<length(x) && i!=j
              ; proj(x[i],1)!=proj(x[j],1)) }*/

10  public class WaitExample{
11      /*< finseq[[:Thread,int:]] x;>*/

13      /*{ disjunct(x) }*/ //class invariant

15      /*[wait]*/   /*1{ owns(thread,lock) }*/
16        /*{ not_owns(thread,lock) && proj(caller,1)==this && includes(x,(:
              thread,proj(caller,2):)) }*/
17        /*<return;>*/
18        /*1{ lock==(:null,0:) && proj(caller,1)==this && includes(x,(:thread
              ,proj(caller,2):)) && get(notified,thread)!=-1 }*/
19      /*[]*/

21      public synchronized void m(){
22          /*1{ free_for(thread,lock) && (\forall int i;true;!includes(x,(:
                thread,i:))) }*/
23          /*1< x=append(x,(:thread,counter:)); >*/
24          /*{ owns(thread,lock) && includes(x,(:thread,conf:)) }*/
25          try{ this.wait();} catch (InterruptedException e){}
26          /*2{ not_owns(thread,lock) && includes(x,(:thread,conf:)) }*/
27          /*{ owns(thread,lock) && includes(x,(:thread,conf:)) }*/
28          return;
29          /*1{ owns(thread,lock) && includes(x,(:thread,conf:)) }*/
30          /*1< x=remove(x,index(x,(:thread,conf:))); >*/
31      }
32  }
```

The wait-method can be called only by a thread owning the lock of the callee object, as expressed by the precondition (24). After invoking wait, the thread gives the lock free, as formalized in (26); when returning, it becomes the lock owner again, as stated by (27).

We use the auxiliary instance variable `x` to store for each local configuration executing `m` the thread and local configuration identities. We use this information to identify local configurations in caller-callee relationship: We can exclude from the interference freedom test for example the invariance of (26) under the built-in return-observation of its callee, setting the lock owner to the identity of the executing thread. Clearly, (26) would not be invariant under the return-observation of its callee; caller and callee execute a common step, and the control point of the caller moves from (26) to (27). We get the following interference freedom condition for setup, where the case `thread_1=thread_2` leads to a contradiction:

```
%precondition assignment
(lock_inst=(null,0) AND PROJ_1(caller_1)=this AND includes(
    x_inst,(thread_1,PROJ_2(caller_1))) AND get(
    notified_inst,thread_1)/=(-1) AND
%assertion
not_owns(thread_2,lock_inst) AND includes(x_inst,(thread_2,
    conf_2)) AND
%class invariant
disjunct(x_inst) AND
%interleavable
```

```
(thread_1=thread_2 IMPLIES (conf_1/=conf_2 AND (this/=
    PROJ_1(caller_1) OR conf_2/=PROJ_2(caller_1)))) AND
(thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(not_owns(thread_2,seq(notified_inst)(get(notified_inst,
    thread_1))) AND includes(x_inst,(thread_2,conf_2)))
```

4.2.4 The cooperation test Whereas the interference freedom test assures invariance of assertions under steps in which they are not involved, the *cooperation test* deals with inductivity for communicating partners, assuring that the global invariant, and the preconditions and the class invariants of the involved statements imply their postconditions after the joint step. The soundness of the proof system requires, that the precondition of an observation, used in the interference freedom test, describes the state in which the assignment is executed. Therefore, we additionally have to show that these assertions are valid immediately after communication.

Like in the interference freedom test, we use the class invariants as antecedents, but it would not be necessary for a minimal proof system. The only exception is method call: In this case the callee class invariant is the only assertion which can be used to describe the callee instance state. But in this case the callee class invariant is already included as the precondition of the called method, which is by definition the class invariant.

The global invariant expresses global invariant properties using auxiliary instance variables which can be changed by observations of communication, only. Consequently, it is automatically invariant under the execution of non-communicating statements. For communication and object creation, however, the invariance must be shown as part of the cooperation test.

We define the corresponding verification conditions for communication and object creation, but we have no cooperation test for notification. As mentioned earlier, notification takes place within a single object, and thus its effect can be captured by a single assignment to the auxiliary variables wait and notified executed by the notifier. Invariance for the notifying thread is covered by the local correctness conditions, whereas preservation of the assertions for the notified partners is assured by the interference freedom test.

We start with the cooperation test for communication; for the corresponding augmentation see Example 18. The semantics is intuitively shown in Fig.3. Since different objects may be involved, the cooperation test is formulated in the global assertion language. The local properties and expressions are expressed in the global language using the lifting substitution. To avoid name clashes between local variables of the partners, we rename those of the callee.

Let z and z' be logical variables representing the caller, respectively the callee object in a method call. We assume the global invariant, the class invariants of the communicating partners, and the preconditions of the communicating statements to hold prior to communication. For method invocation, the precondition of the callee is its class invariant. That the two statements indeed represent

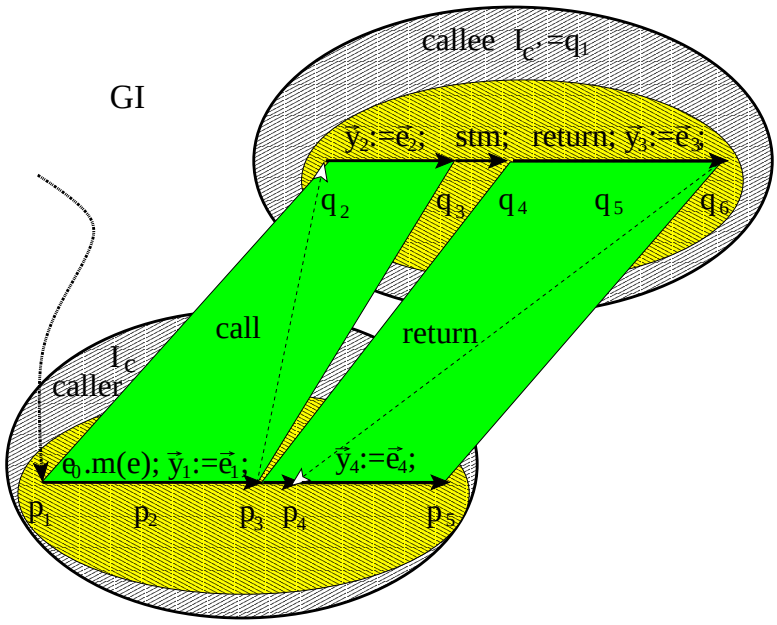

Fig. 3. Communication

communicating partners and especially that the communication is enabled is captured in the assertion comm, which depends on the type of communication. For instance, in case of a synchronized method invocation, the lock of the callee object has to be free or owned by the caller. This is expressed by $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$, where thread is the caller-thread, and where $\mathsf{thread}(z'.\mathsf{lock})$ is the first component of the lock value, i.e., the thread owning the lock of z'. For the invocation of the monitor methods we require that the executing thread is holding the lock. An additional predicate $E_0(z) = z'$ in the condition of a method call $e_0.m(\vec{e})$ states, that z' is indeed the callee object, where $E_0(z)$ is $e_0[z/\mathsf{this}]$. Invoking the start-method of an object whose thread is already started does not have communication effects. The same holds for returning from a run-method. Returning from the wait-method assumes that the thread has been notified and that the callee's lock is free.

Remember that method invocation hands over the return address, and that the values of formal parameters remain unchanged. Furthermore, actual parameters may not contain instance variables, i.e., their interpretation does not change during method execution. Therefore, the formal and actual parameters can be used to identify partners being in caller-callee relationship, using the built-in auxiliary variables.

Note the execution order: first communication takes place, followed by the sender, and then the receiver observation. To describe the common effect, we first have to substitute for the receiver, then for the sender observation, and

finally for communication. For method call, we additionally have to substitute for the initialization of the local variables.

Let again p' denote the assertion p with every local variable u replaced by a fresh one u', and similarly for expressions. As already mentioned, we use the shortcuts $P(z)$ for $p[z/\mathsf{this}]$, $Q'(z')$ for $q'[z'/\mathsf{this}]$, and similarly for expressions, where local variables are viewed on the global level as logical ones.

Definition 4 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test for communication*, if*

$$\models_{\mathcal{G}} GI \wedge P_1(z) \wedge I_c(z) \wedge Q_1'(z') \wedge I_{c'}(z') \wedge \tag{6}$$

$$\mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null} \rightarrow \tag{7}$$

$$(P_2(z) \wedge Q_2'(z')) \circ f_{comm} \wedge (GI \wedge P_3(z) \wedge Q_3'(z')) \circ f_{obs2} \circ f_{obs1} \circ f_{comm}$$

holds for distinct fresh logical variables $z \in LVar^c$ and $z' \in LVar^{c'}$, in the following cases:

1. (a) CALL*: For all statements $\{p_1\}\, u_{ret} := e_0.m(\vec{e});\ \{p_2\}^1\, \langle \vec{y}_1 := \vec{e}_1 \rangle^1\, \{p_3\}$ (or such without receiving a value) in class c with $e_0 \in Exp_c^{c'}$, where method $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ of c' is synchronized with body $\{q_2\}^1\, \langle \vec{y}_2 := \vec{e}_2 \rangle^1\, \{q_3\}$ stm, formal parameters $\vec{u}$, and local variables $\vec{v}$ except the formal parameters. The callee class invariant is $q_3 = I_{c'}$. The assertion* comm *is given by $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$. Furthermore, $f_{comm} = [\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$, $f_{obs1} = [\vec{E}_1(z)/z.\vec{y}_1]$, $f_{obs2} = [\vec{E}_2'(z')/z'.\vec{y}_2']$. If m is not synchronized, $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ in* comm *is dropped.*
   (b) CALL$_{monitor}$*: For $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$,* comm *is given by $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$.*
   (c) CALL$_{start}$*: For $m =$* start*,* comm *is $E_0(z) = z' \wedge \neg z'.\mathsf{started}$, where $\{q_2\}^1\, \langle \vec{y}_2 := \vec{e}_2 \rangle^1\, \{q_3\}$ stm is the body of the* run*-method of c'.*
   (d) CALL$_{start}^{skip}$*: For $m =$* start*, additionally, (6) must hold with* comm *given by $E_0(z) = z' \wedge z'.\mathsf{started}$, $q_2 = q_3 = \mathsf{true}$, and f_{comm} and f_{obs2} are the identity functions.*
2. (a) RETURN*: For all method call statements $u_{ret} := e_0.m(\vec{e}); \{p\}^1\, \langle \vec{y}_1 := \vec{e}_1 \rangle^1\, \{p_1\}^2\, \{p_2\}^3\, \langle \vec{y}_4 := \vec{e}_4 \rangle^2\, \{p_3\}$ (or such without receiving a value) occurring in c with $e_0 \in Exp_c^{c'}$, such that method m of c' has the return statement $\{q_1\}$* return *$e_{ret}; \{q_2\}^1\, \langle \vec{y}_3 := \vec{e}_3 \rangle^1\, \{q_3\}$, and formal parameter list $\vec{u}$, Equation (6) must hold with* comm *given by $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$, and where $f_{comm} = [E'_{ret}(z')/u_{ret}]$, $f_{obs1} = [\vec{E}_3'(z')/z'.\vec{y}_3']$, and $f_{obs2} = [\vec{E}_4(z)/z.\vec{y}_4]$.*
   (b) RETURN$_{wait}$*: For $\{q_1\}$* return$_{getlock}$*; $\{q_2\}^1\, \langle \vec{y}_3 := \vec{e}_3 \rangle^1\, \{q_3\}$ in a* wait*-method,* comm *is $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.\mathsf{lock} = \mathsf{free} \wedge \mathsf{thread}' \in z'.\mathsf{notified}$.*
   (c) RETURN$_{run}$*: For $\{q_1\}$* return*; $\{q_2\}^1\, \langle \vec{y}_3 := \vec{e}_3 \rangle^1\, \{q_3\}$ occurring in a* run*-method, $p_1 = p_2 = p_3 = \mathsf{true}$,* comm *$= \mathsf{true}$, and furthermore f_{comm} and f_{obs2} the identity function.*

Example 17. For the proof outline of Example 8 three global conditions are generated: one for the method call at (23), one for the call at (31), and one for the corresponding return for the second call. Note that we do not have any conditions for returning from the first call (23), because all postconditions are by definition true. The first condition

```
FORALL (caller:Account) : caller/=null IMPLIES
FORALL (callee:Account) : callee/=null IMPLIES
%precondition caller
((i_1>0 AND
%class invariant caller and callee + caller-callee relationship
Account.x(caller)>=0 AND Account.x(callee)>=0 AND caller=
    callee) IMPLIES
%postcondition callee
(i_1>0 OR (Account.x(callee)+i_1>=0 AND owns(thread_1,
    Account.lock(callee)))))
```

states that the class invariants and the preconditions of caller and callee imply the postcondition of the callee. The *PVS*-expression $c.x(z)$ represents the qualified reference $z.x$ for z of type c. Note that the global invariant, the postcondition of the caller, and the assertions at the auxiliary points are by definition true. The caller-callee relationship of the partners is assured by requiring caller = callee, since it is a self-call. The condition for the second call is similar:

```
FORALL (caller:Account) : caller/=null IMPLIES
FORALL (callee:Account) : callee/=null IMPLIES
%precondition caller
((Account.x(caller)>=i_1 AND i_1>0 AND owns(thread_1,
    Account.lock(caller)) AND
%class invariants caller and callee + caller-callee relationship
Account.x(caller)>=0 AND Account.x(callee)>=0 AND caller=
    callee) IMPLIES
%postcondition caller
(i_1>0 AND
%postcondition callee
(-i_1>0 OR (Account.x(callee)+(-i_1)>=0 AND owns(thread_1,
    Account.lock(callee))))))
```

The condition for return assures the caller-callee relationship of the partners by additionally requiring, that the formal parameters equal the actual ones. Applied to the built-in auxiliary parameter thread, this requirement implies for example that caller and callee are the same thread, i.e., thread_1 = thread_2, which we need to show that the caller owns the lock after communication:

```
FORALL (caller:Account) : caller/=null IMPLIES
FORALL (callee:Account) : callee/=null IMPLIES
%precondition caller
((i_1>0 AND
%class invariant caller
Account.x(caller)>=0 AND
%precondition callee
(i_2>0 OR owns(thread_2,Account.lock(callee))) AND
%class invariant callee
Account.x(callee)>=0 AND
%caller-callee relationship
```

```
caller=callee AND i_2=(-i_1) AND thread_2=thread_1 AND
    caller_2=(caller,conf_1,thread_1)) IMPLIES
%postcondition caller
owns(thread_1,Account.lock(caller)))
```

Besides method calls and return, the cooperation test needs to handle object creation, taking care of the preservation of the global invariant, the postcondition of the new-statement and its observation, and the new object's class invariant. We can assume that the precondition of the object creation statement, the class invariant of the creator, and the global invariant hold in the configuration prior to instantiation. Note again that the class invariant as antecedent would not be necessary for a minimal proof system. The extension of the global state with a freshly created object is formulated in a *strongest postcondition* style, i.e., it is required 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 expressed by the global assertion $\mathsf{Fresh}(z', u)$ defined as $\mathsf{InitState}(u) \wedge u \notin z' \wedge \forall v.\ v \in z' \vee v = u$ (see page 28 for the definition of InitState). To express that an assertion refers to the set of existing objects *prior* to the extension of the global state, we need to *restrict* any existential quantification in the assertion 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 $\exists z.\ \mathsf{obj}(z) \subseteq z' \wedge P'$, where $obj(v)$ denotes the set of objects occurring in the value v, formally

$$obj(v) = \begin{cases} \emptyset & \text{if } v \in Val^{\mathsf{Bool}} \cup Val^{\mathsf{Int}} \\ \{v\} & \text{if } v \in \bigcup_c Val^c_{null} \\ obj(v_1) \cup obj(v_2) & \text{if } v = (v_1, v_2) \in \bigcup_{t_1,t_2} Val^{t_1 \times t_2}_{null} \\ \bigcup_{v_i \in v} obj(v_i) & \text{if } v \in \bigcup_t Val^{\mathsf{list}\,t}_{null}\ . \end{cases}$$

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

Lemma 4. *Assume a global state σ, an extension $\sigma' = \sigma[\alpha \mapsto \sigma^{c,init}_{inst}]$ for some $\alpha \in Val^c$, $\alpha \notin Val(\sigma)$, and a logical environment ω referring only to values existing in σ. Let v be the sequence consisting of all elements of $\bigcup_c Val^c_{null}(\sigma)$. Then for all global assertions P and logical variables $z' \in LVar^{\mathsf{list\,Object}}$ not occurring in P,*

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

Thus a predicate $P \downarrow z'$, evaluated immediately after the instantiation, express that P holds prior to the creation of the new object. This leads to the following definition of the cooperation test for object creation:

Definition 5 (Cooperation test: Instantiation). *A proof outline satisfies the* cooperation test for object creation*, if for all classes c' and statements*

$\{p_1\}\, u := \mathsf{new}^c; \{p_2\}^1\, \langle \vec{y} := \vec{e} \rangle^1\, \{p_3\}$ *in* c'*:*

$$\models_{\mathcal{G}} z \neq \mathsf{null} \wedge z \neq u \wedge \tag{8}$$

$$\exists z'.\ \Big(\mathsf{Fresh}(z', u) \wedge (GI \wedge (\exists u.\ P_1(z)) \wedge I_{c'}(z))\ \downarrow z'\Big) \rightarrow \tag{9}$$

$$P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z)) \circ f_{obs}\ ,$$

with $z \in LVar^{c'}$ *and* $z' \in LVar^{\mathsf{list\,Object}}$ *fresh , and where* $f_{obs} = [\vec{E}(z)/z.\vec{y}]$.

Example 18. The proof outline below specifies two classes, called `Creator` and `Created`. Instances of the `Creator` class offer the method `create()` which creates an instance of the `Created` class and gives it back as a return value. The global invariant states that there exists at most one instance of the `Creator` class, and that its auxiliary instance variable `nr` stores the number of the existing `Created` instances.

```
//function definition
/*{ boolean disjunct(finseq[Created] z) =
       (\forall int i; 0<=i && i<length(z); z[i]!=null &&
       (\forall int j; 0<=j && j<length(z) && i!=j ; z[i]!=z[j]))}*/

//global invariant
/*{ (\forall Creator o; o!=null;
     (\forall Creator o2; o2!=null; o2==o) &&
     (\forall finseq[Created] z; disjunct(z) && (\forall Created z2; z2!=
         null; includes(z,z2)); o.nr == length(z)))
     }*/

public class Creator {
     /*< int nr; >*/

     public void create(){
          Created o;

          o = new Created();
          /*1< nr = nr + 1; >*/
     }
}

class Created {}
```

We apply the proof system to these two classes. Of course, the global invariant describes a program, which contains these classes, only then correctly, if the context of these classes also preserve it. Thus this example shows also, how to verify parts of a program to be correct under the assumption that the remaining verification conditions hold for the environment. *Verger* generates the following cooperation test condition for object creation, where the type $\texttt{Object}_{\texttt{old}}$ in theory `Object` represents the logical variable z' in the cooperation test.

```
Object: THEORY
BEGIN
     ...
     new_Object: Object
     Object_old: NONEMPTY_TYPE = {o:Object | o=null OR o /=
         new_Object} CONTAINING null
```

```
END Object

Created_type: THEORY
BEGIN
    ...
    Created_old: NONEMPTY_TYPE = {o:Created | o=null OR o
        /= new_Object} CONTAINING null
END Created_type

Created: THEORY
BEGIN
    ...
    init(o:Created): bool = (o=new_Object AND o/=null AND
        ... AND Created.lock(o)=(null,0))
END Created

...

global_cond_0 : THEORY
BEGIN
...
condition : LEMMA
% z /= null /\
FORALL (creator:Creator)  : creator/=null IMPLIES
% z /= u /\ Fresh(z',u)
((creator/=u AND Created.init(u) AND
% GI restricted to z'
(FORALL (o:Creator)  : o/=null IMPLIES
  ((FORALL (o2:Creator)  : o2/=null IMPLIES o2=o) AND
  (FORALL (z:finseq[Created_old])  : ((disjunct(z) AND
  (FORALL (z2:Created_old)  : (z2/=null IMPLIES includes(z,
     z2)))) IMPLIES
  (Creator.nr(o)=length(z))))))) IMPLIES
% GI after execution
(FORALL (o:Creator)  : (o/=null IMPLIES
  ((FORALL (o2:Creator)  : (o2/=null IMPLIES o2=o)) AND
  (FORALL (z:finseq[Created])  : ((disjunct(z) AND
  (FORALL (z2:Created)  : (z2/=null IMPLIES includes(z,z2))
     )) IMPLIES
  (IF (o=creator) THEN (Creator.nr(creator)+1) ELSE Creator
     .nr(o) ENDIF = length(z))))))))

END global_cond_0
```

Further examples can be found in Appendix A.

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 fact 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, and $prog' \vdash \varphi'$, if $prog'$ satisfies the verification conditions of the proof system:

Definition 6. *Given a program prog with annotation φ, then $prog \models \varphi$ iff for all reachable configurations $\langle T, \sigma\rangle$ of prog, for all $(\alpha, \tau, stm) \in T$, 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. $\omega, \sigma \models_{\mathcal{G}} GI$.

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

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

For proof outlines, 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 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 proof outline, i.e., the one decorated with assertions, and extended by auxiliary variables. The transformation is done for the sake of verification, only, and as far as the unaugmented portion of the states and the configurations is concerned, the behavior of the original and the transformed program are the same.

To make the connection between original program and the proof outline precise, we define a projection operation $\downarrow prog$, that jettisons all additions of the transformation. So let $prog'$ be a proof outline for *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 and removing all annotations and augmentations. Additionally, for local configurations $(\alpha, \tau, \mathsf{return}_{getlock}; stm) \in T'$, if the executing thread is in the wait set, i.e., $\tau(\mathsf{thread}) \in \sigma'(\alpha)(\mathsf{wait})$ then the statement $\mathsf{return}_{getlock}$ gets replaced by $?\mathsf{signal}; \mathsf{return}_{getlock}$. Furthermore, for local configurations $(\alpha, \tau, stm; \mathsf{return}; stm')$ with $stm \neq \epsilon$ an auxiliary assignment in the notify- or the notifyAll-method, the auxiliary assignment *stm* gets replaced by !signal and !signal_all, respectively. The following lemma expresses that the transformation does not change the behavior of programs:

Lemma 5. *Let prog′ be a proof outline for 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* $\langle T' \downarrow prog, \sigma' \downarrow prog\rangle = \langle T, \sigma\rangle$.

The augmentation introduced a number of specific auxiliary variables that reflect the predicates used in the semantics. That the semantics is faithfully represented by the variables is formulated in the following lemmas.

Lemma 6 (Identification). *Let* $\langle T, \sigma\rangle$ *be a reachable configuration of a proof outline. Then*

1. *for all stacks* ξ *and* ξ' *in* T *and for all local configurations* $(\alpha, \tau, stm) \in \xi$ *and* $(\alpha', \tau', stm') \in \xi'$ *we have* $\tau(\mathsf{thread}) = \tau'(\mathsf{thread})$ *iff* $\xi = \xi'$, *and*
2. *for each stack* $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$ *in* T *and indices* $i < j$,
   (a) $\tau_i(\mathsf{thread}) = \alpha_0$;
   (b) $\alpha_i = \alpha_j$ *implies* $\tau_i(\mathsf{conf}) < \tau_j(\mathsf{conf}) < \sigma(\alpha_i)(\mathsf{counter})$,
   (c) $\tau_j(\mathsf{caller}) = (\alpha_{j-1}, \tau_{j-1}(\mathsf{conf}), \tau_{j-1}(\mathsf{thread}))$, *and*
   (d) $\mathsf{proj}(\tau_0(\mathsf{caller}), 3)) \neq \tau_0(\mathsf{thread})$.

Lemma 7 (Lock, Wait, Notify). *Let* $\langle T, \sigma\rangle$ *be a reachable configuration of a proof outline for the original program prog,* $\alpha \in Val(\sigma)$ *and* $\xi = (\alpha_0, \tau_0, stm_0) \circ \xi' \in T$. *Let furthermore* n *be the number synchronized method executions of* ξ *in* α, *i.e.,* $n = |\{(\alpha, \tau, stm) \in \xi \mid stm\ synchr.\}|$. *Then*

1. (a) $\neg owns(T \downarrow prog, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{lock}) = free$
   (b) $owns(\xi \downarrow prog, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{lock}) = (\alpha_0, n)$
2. (a) $proj(\sigma(\alpha)(\mathsf{wait})[i], 1) = proj(\sigma(\alpha)(\mathsf{wait})[j], 1)$ *implies* $i = j$
   (b) $proj(\sigma(\alpha)(\mathsf{notified})[i], 1) = proj(\sigma(\alpha)(\mathsf{notified})[j], 1)$ *implies* $i = j$
   (c) *if* $(\alpha_0, m) \in \sigma(\alpha)(\mathsf{wait})$ *or* $(\alpha_0, m) \in \sigma(\alpha)(\mathsf{notified})$ *then* $m = n$
   (d) $\sigma(\alpha)(\mathsf{wait}) \cap \sigma(\alpha)(\mathsf{notified}) = \emptyset$
   (e) $\xi \in wait(T \downarrow prog, \alpha)$ *iff* $(\alpha_0, n) \in \sigma(\alpha)(\mathsf{wait})$
   (f) $\xi \in notified(T \downarrow prog, \alpha)$ *iff* $(\alpha_0, n) \in \sigma(\alpha)(\mathsf{notified})$.

The above Lemma assures disjunctness of the sequences stored in the wait and notified variables; if the order of the elements is unimportant, in the following we sometimes use set notation for their values.

Lemma 8 (Started). *For all reachable configurations* $\langle T, \sigma\rangle$ *of a proof outline and all objects* $\alpha \in Val(\sigma)$, *we have* $started(T \downarrow prog, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{started})$.

Let *prog* be a program with annotation φ, and *prog′* a a corresponding proof outline 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 $\models \varphi' \rightarrow \varphi$ iff $\models_{\mathcal{G}} GI' \rightarrow GI$, $\models_{\mathcal{L}} I'_c \rightarrow I_c$ for all classes c, and $\models_{\mathcal{L}} p' \rightarrow p$, 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 augmented 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 basically an induction on the length of computation, simultaneous on all three parts from Definition 6. Theorem 1 is formulated for reachability of augmented programs. With the help of Lemma 5, we immediately get:

Corollary 1. *If $prog' \vdash \varphi_{prog'}$ and $\models \varphi_{prog'} \rightarrow \varphi_{prog}$, 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:

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

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 $\models \varphi_{prog'} \rightarrow \varphi_{prog}$. 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 8 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 the programs are augmented with the specific auxiliary variables. 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 with the update $\mathsf{loc} := l$, where l is the label of the control point after the given occurrence of the assignment. Also unobserved statements are extended with the update. We write $l \equiv stm$ if l represents the control point in front of *stm*.

The standard way for completeness augmentation is to add information into the states about the way how it has been reached, i.e., the *history* of the computation leading to the configuration. This information is recorded using history variables.

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 is recorded in the *local* history and the external behavior in the *communication* history.

The local history keeps track of the state updates due to local steps of threads, i.e., steps which does not communicate or create a new object. We store in the local history the updated local and instance states of the executing local configuration and the object in which the execution takes place. Note that the local history stores also the values of the built-in auxiliary variables, and thus the identities of the executing thread and especially the executing local configuration.

The communication history keeps information about the kind of communication, the communicated values, and the identity 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 $\bigcup_{c\in\mathcal{C}}\{\mathsf{new}^c\}\cup\bigcup_{m\in\mathcal{M}}\{!m,?m\}\cup\{!\mathsf{return},?\mathsf{return}\}$ of constants for this purpose. Notification does not update the communication history, since it is object-internal computation. For the same reason, we don't record self-communication in h_{comm}. 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 [5] for such a compositional semantics.

Definition 7 (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 assignment* $\vec{y} := \vec{e}$ *in each class* c *that is not the observation of a method call or of the reception of a return value is extended with*

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

   where $\vec{x}$ *are the instance variables of class* c *containing also* h_{comm} *but without* h_{inst}*, and* $\vec{v}$ *are the local variables. Observations* $\vec{y} := \vec{e}$ *of* $u_{ret} := e_0.m(\vec{e}')$ *and of the corresponding reception of the return value get extended with the assignment*

$$\mathsf{h}_{inst} := \mathsf{if}\,(e_0 = \mathsf{this})\,\mathsf{then}\ \mathsf{h}_{inst}\ \mathsf{else}\ \mathsf{h}_{inst} \circ ((\vec{x},\vec{v})[\vec{e}/\vec{y}])\ \mathsf{fi}\,,$$

   instead, if $m \neq \mathsf{start}$*. For* $e_0.\mathsf{start}(\vec{e}')$*;* $/^{*1}\langle\vec{y} := \vec{e}\rangle^{*}/$ *we use the same update with the condition* $e_0 = \mathsf{this}$ *replaced by* $e_0 = \mathsf{this} \wedge \neg\mathsf{started}$*.*
2. *Every communication and object creation gets observed by*

$$\begin{aligned}\mathsf{h}_{comm} := \mathsf{if}\,(\mathsf{partner} = \mathsf{this})\,\mathsf{then}\ \mathsf{h}_{comm}\ \mathsf{else}\\ \mathsf{h}_{comm} \circ (\mathsf{sender},\mathsf{receiver},\mathsf{values})\ \mathsf{fi}\ ,\end{aligned}$$

where the expressions partner, sender, receiver, *and* values *are defined depending on the kind of communication statements as follows:*

communication statement	partner	sender	receiver	values
$u := \mathsf{new}^c$	null	this	null	$\mathsf{new}^c\, u, \mathsf{thread}$
$u_{ret} := e_0.m(\vec{e})$	e_0	this	e_0	$!m(\vec{e})$
corresponding reception of return	e_0	e_0	this	$?\,\mathsf{return}\, u$
reception of method call $m(\vec{u})$	caller_obj	caller_obj	this	$?m(\vec{u})$
return e_{ret}	caller_obj	this	caller_obj	$!\,\mathsf{return}\, e_{ret}$

where caller_obj *is the first component of the variable* caller.

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 variables thread and conf, and specifies its instance local state after the execution of the assignment. Note that especially the values of the auxiliary variables introduced in the augmentation are recorded in the local history. In the following we will also write (σ_{inst}, τ) when referring to elements of h_{inst}.

Note furthermore that the communication history records also the identities of the communicating threads in values.

Next we introduce the annotation for the augmented program.

Definition 8 (Reachability annotation).

1. $\omega, \sigma \models_{\mathcal{G}} GI$ *iff there exists a reachable* $\langle T, \sigma' \rangle$ *such that* $Val(\sigma) = Val(\sigma')$, *and for all* $\alpha \in Val(\sigma)$, $\sigma(\alpha)(\mathsf{h}_{comm}) = \sigma'(\alpha)(\mathsf{h}_{comm})$.
2. *For each class* c, *let* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} I_c$ *iff there is a reachable* $\langle T, \sigma \rangle$ *such that* $\sigma(\alpha) = \sigma_{inst}$, *where* $\alpha = \sigma_{inst}(\mathsf{this})$. *For each class* c *and method* m *of* c, *the pre- and postconditions of* m *are given by* I_c.
3. *For assertions at control points,* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(stm)$ *iff there is a reachable* $\langle T, \sigma \rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$ *for* $\alpha = \sigma_{inst}(\mathsf{this})$, *and with* $(\alpha, \tau, stm; stm') \in T$.
4. *For preconditions* p *of observations of communication or object creation, let* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ *iff there is a reachable* $\langle T, \sigma \rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$ *for* $\alpha = \sigma_{inst}(\mathsf{this})$, *and with* $(\alpha, \tau', stm; stm') \in T$ *enabled to communicate resulting in the local state* τ *directly after communication, where* stm *is the corresponding communication statement.*
   For observing the reception of a method call, instead of the existence of the enabled $(\alpha, \tau', stm; stm') \in T$, *we require that a call of method* m *of* α *is enabled with resulting callee local state* τ *directly after communication.*

It can be shown that these assertions are expressible in the assertion language [40]. The augmented 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.

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 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 chosen instance is concerned. This property is stated formally in the following local merging lemma.

Lemma 9 (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* $(\alpha, \tau, stm) \in T_1$ *with* $\alpha \in Val(\sigma_1) \cap Val(\sigma_2)$. *Then* $\sigma_1(\alpha)(\mathsf{h}_{inst}) = \sigma_2(\alpha)(\mathsf{h}_{inst})$ *implies* $(\alpha, \tau, stm) \in T_2$.

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

Note that together with the local merging lemma this implies that all local configurations in $\langle T_1, \sigma_1 \rangle$ executing in α and all local configurations in $\langle T_2, \sigma_2 \rangle$ executing in $\beta \neq \alpha$ 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.*

6 Conclusion

This paper presents the first sound and complete assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline. It extends earlier work ([3] and especially [4]) by integrating *Java*'s wait and notify constructs into the assertional proof system and by moving towards a more compositional identification mechanism for threads.

Related work From its inception, *Java* attracted interest from the formal methods community: The widespread use of *Java* across platforms made the need for formal studies and verification support more urgent, the grown awareness and advances of formal methods for real-life applications and languages made it more acceptable, and last not least the array of non-trivial language features made it challenging and interesting. Thus *Java* offered a rich field for formal studies, ranging from formal semantics [38, 6] over bytecode verification and static analysis [27] to model checking [19].

As far as proof systems and verification support for object-oriented programs is concerned, research mostly concentrated on *sequential* languages resp. sequential subsets of *Java*. For instance, Poetzsch-Heffter and Müller [35, 33, 32, 34] develop a Hoare-style programming logic presented in sequent formulation for a sequential kernel of *Java*. Their *Java*-fragment does not contain multithreading, but goes beyond this work in featuring interfaces, subtyping, and inheritance. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer assisted soundness proof of the proof system. Two papers dealing with an axiomatic semantics for object-oriented languages in a weakest liberal precondition style are [15] and [26].

[37, 36] use (a modification of) the *object constraint language* OCL as assertional language to annotate UML class diagrams and to generate proof conditions for *Java*-programs. The work [12] presents a model checking algorithm and its implementation in Isabelle/HOL to check type correctness of Java bytecode. The work [41] formalizes a large subset of *JavaCard*, including exception handling, in *Isabelle/HOL*, and its soundness and completeness is shown within the theorem prover. The work in [2] presents a Hoare-style proof system for a sequential object-oriented calculus [1]. Their language features heap-allocated objects (but no classes), side-effects and aliasing, and its type system supports subtyping. Furthermore, their language allows nested statically let-bound variables, which requires a more complex semantical treatment for variables based on closures, and ultimately renders their proof system incomplete. Their assertion language is presented as an extension of the object calculus' language of type and analogously, the proof system extends the type derivation system, where the types cater for the static properties of the program and the "specification" takes care of the dynamic behavior. The close connection of types and specifications in the presentation in is exploited in [39] for the generation of verification conditions. Applying *type inference* technology to (a syntax-directed version of) Abadi and Leino's proof system allows to factor out the proof obligations from the structural part of the proof. In [23], the logic is implemented in the *Lego* theorem prover. A survey about *monitors* in general, including proof rules for various monitor semantics, can be found in [13].

Formal semantics of *Java*, including multithreaded execution, and its virtual machine in terms of abstract state machines is given in [38]. A structural operational semantics of multithreaded *Java* can be found in [14].

Future work Based on the proof theory presented, we are currently developing a verification condition generator for *Java*, where a theorem prover is used to

verify the conditions. Of particular interest in this context is an integration of our method with related approaches like the LOOP project [20, 29].

As future work, we plan to extend $Java_{MT}$ by further constructs, especially 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 [7]. An extension of the semantics and the proof theory to detect deadlocks and termination is also of interest.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT '97*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696, Lille, France, Apr. 1997. Springer-Verlag. An extended version of this paper appeared as SRC Research Report 161 (September 1998).
3. E. Ábrahám-Mumm and F. de Boer. Proof-outlines for threads in Java. In C. Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 2000.
4. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In M. Nielsen and U. H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, Apr. 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.
5. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. A compositional operational semantics for $Java_{MT}$. 2003. To appear. A preliminary version appeared as Technical Report TR-ST-02-2, May 2002.
6. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. LNCS State-of-the-Art-Survey. Springer-Verlag, 1999.
7. P. America. A behavioural approach to subtyping in object-oriented programming languages. 443, Phillips Research Laboratories, January/April 1989.
8. P. America and F. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1993.
9. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.
10. 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.
11. 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.
12. D. Basin, S. Friedrich, and M. Gawkowski. Verified bytecode model checkers. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs'02)*, pages 47–66, August 2002.
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 [6], pages 157–200.

15. F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–156. Springer-Verlag, 1999.
16. 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.
17. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
18. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
19. K. Havelund. *Java Pathfinder User Manual*. NASA, Aug. 1999. NASA Ames Technical Report, available at `http://ase.arc.nasa.gov/havelund/jpf.html`.
20. U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In C. Hankin, editor, *Proceedings of ESOP '98*, volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
21. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [22].
22. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.
23. M. Hofmann and F. Tang. Implementing a program logic of objects n a higher-order logic theorem prover. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics (TPHOL 2000)*, volume 1869 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 2000.
24. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
25. H. Hussmann, editor. *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
26. K. R. M. Leino. Exstatic: An object-oriented programming language with axiomatic semantics. In B. C. Pierce, editor, *Proceedings of FOOL 4*. Free electronic publication, Jan. 1997. Available electronically through `http://www.cs.indiana.edu/hyplan/pierce/fool/`.
27. X. Leroy. Java bytecode verification: An overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV '01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.
28. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
29. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.
30. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
31. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.
32. A. Poetzsch-Heffter. A logic for the verification of object-oriented programs. In R. Berghammer and F. Simon, editors, *Proceedings of Programming Languages and Fundamentals of Programming*, pages 31–42. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Nov. 1997. Bericht Nr. 9717.
33. A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. Technische Universität München, Jan. 1997. Habilitationsschrift.

34. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W.-P. de Roever, editors, *Proceedings of PROCOMET '98*. International Federation for Information Processing (IFIP), Chapman & Hall, 1998.
35. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.
36. B. Reus, R. Hennicker, and M. Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Hussmann [25], pages 300–316.
37. B. Reus and M. Wirsing. A Hoare-logic for object-oriented programs. Technical report, LMU München, 2000.
38. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.
39. F. Tang and M. Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects (extended abstract). In *Proceedings of the 9th International Workshop on Foundations of Object-Oriented Languages (FOOL'02)*, 2002. A longer version is available as LFCS technical report.
40. 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.
41. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.
42. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P.-A. Lindsay, editors, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.
43. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Object Technology Series. Addison-Wesley, 1999.

A Examples

B Proofs

B.1 Properties of substitutions and projection

Proof (of Lemma 1). We prove the lemma by straightforward induction on the structure of local assertions. Let $\acute{\sigma}_{inst} = \grave{\sigma}_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}}]$ and $\acute{\tau} = \grave{\tau}[u \mapsto [\![e_i]\!]_{\mathcal{L}}^{\omega,\grave{\sigma}_{inst},\grave{\tau}}]$. In the case for local variables $u = y_i$ we get

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

For instance variables $x = y_i$ similarly:

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

The remaining cases are straightforward. □

Proof (of Lemma 2). Let $\acute{\omega} = \grave{\omega}[\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}]$ and let $\acute{\sigma}$ be defined by $\grave{\sigma}[[\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}}]$. We proceed by induction on the structure of global expressions and assertions. The base cases for null and z' are straightforward. For the induction cases, we start with the crucial one for qualified reference to instance variables:

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

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

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

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

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

For operator expressions we get:

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

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

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

□

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.

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

Compound expressions are treated by straightforward induction:

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

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 $\mathit{Val}^t_{null}(\sigma) = \mathit{Val}^t$. We get

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

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

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

The last equation uses the assumption that the local state τ and the instance state $\sigma(\omega(z))$ assign values from $\mathit{Val}_{null}(\sigma)$ to all variables, i.e., e does not refer to values of non-existing objects. Consequently, $v \in \mathit{Val}^t_{null}$ together with $[\![z' \in e]\!]_{\mathcal{L}}^{\omega[z' \mapsto v],\sigma(\omega(z)),\tau} = \mathit{true}$ implies $v \in \mathit{Val}^t_{null}(\sigma)$. The case for restricted quantification over subsequences is analogous. □

Lemma 11. *Let σ be a global state and ω a logical environment referring only to values existing in σ. Then $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in \mathit{Val}_{null}(\sigma)$ for all global expressions $E \in \mathit{GExp}$ that can be evaluated in the context of ω and σ.*

Proof (of Lemma 11). By structural induction on the global assertion. The case for logical variables $z \in \mathit{LVar}^t$ is immediate by the assumption about ω, the ones for null and operator expressions are trivial, respectively follows by induction. For qualified references $E.x$ with $E \in \mathit{GExp}^c$ and $x \in \mathit{IVar}^t_c$ an instance variable of class c, if $E.x$ can be evaluated in the context of ω and σ, then $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \neq \mathit{null}$. Hence by induction $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in \mathit{Val}_{null}(\sigma)$, more specifically $[\![E]\!]_{\mathcal{G}}^{\omega,\sigma} \in \mathit{Val}(\sigma)$. Therefore by definition of global states $\sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \in \mathit{Val}_{null}(\sigma)$. □

Proof (of Lemma 4). By structural induction on global assertions. Assume a global state $\grave{\sigma}$, an extension $\acute{\sigma} = \grave{\sigma}[\alpha \mapsto \sigma^{c,init}_{inst}]$ for some $\alpha \in \mathit{Val}^c$, $\alpha \notin \mathit{Val}(\grave{\sigma})$, and a logical environment ω referring only to values existing in $\grave{\sigma}$. Let v be the sequence consisting of all elements of $\bigcup_c \mathit{Val}^c_{null}(\grave{\sigma})$. Let finally P be a global assertion, $z' \in \mathit{LVar}^{\mathsf{list\,Object}}$ a logical variable not occurring in P, and $\acute{\omega} = \grave{\omega}[z' \mapsto v]$. Since z' is fresh in P, we have for all logical variables z in P that $[\![z]\!]_{\mathcal{G}}^{\grave{\omega},\grave{\sigma}} = \grave{\omega}(z) = \acute{\omega}(z) = [\![z]\!]_{\mathcal{G}}^{\acute{\omega},\acute{\sigma}}$. For qualified references to instance variables, the

argument is as follows:

$$\begin{array}{lll}
[\![E.x]\!]^{\grave{\omega},\grave{\sigma}}_{\mathcal{G}} = \grave{\sigma}([\![E]\!]^{\grave{\omega},\grave{\sigma}}_{\mathcal{G}})(x) & & \text{semantics} \\
= \acute{\sigma}([\![E]\!]^{\grave{\omega},\grave{\sigma}}_{\mathcal{G}})(x) & & [\![E]\!]^{\grave{\omega},\grave{\sigma}}_{\mathcal{G}} \neq \alpha \text{ by Lemma 11 and } \alpha \notin \mathit{Val}(\grave{\sigma}) \\
= \acute{\sigma}([\![E \downarrow z']\!]^{\acute{\omega},\acute{\sigma}}_{\mathcal{G}})(x) & & \text{by induction} \\
= [\![(E \downarrow z').x]\!]^{\acute{\omega},\acute{\sigma}}_{\mathcal{G}} & & \text{semantics} \\
= [\![(E.x) \downarrow z']\!]^{\acute{\omega},\acute{\sigma}}_{\mathcal{G}} & & \text{def. } \downarrow z' \text{ .}
\end{array}$$

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

$$\begin{array}{lll}
 & \grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \exists z.\ P & \\
\iff & \grave{\omega}[z \mapsto u], \grave{\sigma} \models_{\mathcal{G}} P \text{ for some } u \in \mathit{Val}^t_{null}(\grave{\sigma}) & \text{semantics} \\
\iff & \acute{\omega}[z \mapsto u], \acute{\sigma} \models_{\mathcal{G}} P \downarrow z' \text{ for some } u \in \mathit{Val}^t_{null}(\grave{\sigma}) & \text{induction} \\
\iff & \acute{\omega}[z \mapsto u], \acute{\sigma} \models_{\mathcal{G}} \mathsf{obj}(z) \subseteq z' \wedge P \downarrow z' & \mathit{Val}^t_{null}(\grave{\sigma}) \subseteq v \\
 & \qquad \text{for some } u \in \mathit{Val}^t_{null}(\grave{\sigma}) & \\
\iff & \acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} \exists z.\ \mathsf{obj}(z) \subseteq z' \wedge P \downarrow z' & \text{semantics} \\
\iff & \acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} (\exists z.\ P) \downarrow z'. &
\end{array}$$

The remaining cases are straightforward. □

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 proof outlines, for instance properties of the auxiliary variables added in the transformation. Afterwards, we show soundness of the proof system.

B.2.1 Invariant properties

Proof (of the transformation Lemma 5). Both directions can be shown by straightforward induction on the length of reduction. The only interesting property of the transformation is the representation of notification by a single auxiliary assignment of the notifier. For this case we use Lemma 7 showing soundness of the representation of the wait and notified sets by the auxiliary instance variables wait and notified. □

Proof (of Lemma 6). All parts by straightforward induction on the steps of proof outlines. □

Proof (of Lemma 7). The cases 2e and 2f are satisfied by the definition of the projection operator. Inductivity for the cases 2a and 2b are easy to show using Lemma 6 and the cases 2e and 2f of this lemma. If the order of the elements is unimportant, in the following we also use set notation for the values of the wait and notified variables. Correctness of the projection operation uses the results of

this lemma and is formulated in Lemma 5. For the other cases we proceed by induction on the length of the run $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \acute{T}, \acute{\sigma}\rangle$ of the proof outline $prog'$.

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 (α, τ, stm), executing the non-synchronized main-statement of the program, i.e., $\neg owns(T_0 \downarrow prog, \alpha)$, and initially the lock of the only object α is set to *free*. Furthermore, the instance variables wait and notified of the initial object are set to $\emptyset$.

For the inductive step, assume $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$. We distinguish on the kind of the last computation step.

Case: $\textsc{Call}_{start}$, $\textsc{Call}^{skip}_{start}$, $\textsc{Return}_{run}$
In these cases none of the concerned variables or predicates are touched, no new objects are created, and no local configurations are pushed or popped, and the property follows directly by induction.

Case: $\textsc{Ass}_{inst}$, $\textsc{Ass}_{loc}$, Signal, $\textsc{Signal}_{skip}$, SignalAll
Note that this case handles assignments, but not the observations of communication and object creation. Remember furthermore that the signaling mechanism is implemented in proof outlines by auxiliary assignments.

If the assignment is not in a notify- or in a notifyAll-method representing notification, then the case is analogous to the above one.

Assume now that the assignment in the last computation step represents notification in a notify-method of the proof outline. I.e., a thread $\xi_1 \in \grave{T}$ notifies another thread $\xi_2 = (\alpha_2, \tau, stm) \circ \xi_2' \in \grave{T}$ in the *wait* set of α. Remember that notification is represented by a single assignment of the notifier, and thus the stack of the notified thread ξ_2 does not change. However, according to the projection definition, as the notifier changes the value of wait of α, the projection $\xi_2 \downarrow prog$ represents a thread being in the wait set in $\langle \grave{T}, \grave{\sigma}\rangle$ and being in the notified set in $\langle \acute{T}, \acute{\sigma}\rangle$.

The only relevant effect of the step is moving $(\alpha_2, n) \in \grave{\sigma}(\alpha)(\mathsf{wait})$ from the wait set into the notified set of α, where n is by induction the number of synchronized invocations of ξ_2 in α. Thus the properties 1a, 1b and 2c are automatically invariant. Induction implies also uniqueness of the representation of the wait and notified sets, i.e., α_2 is contained neither in $\grave{\sigma}(\alpha)(\mathsf{notified})$ nor in $\acute{\sigma}(\alpha)(\mathsf{wait})$. Thus moving the thread of α_2 from the wait into the notified set does not violate uniqueness of the representation.

If the wait set $\grave{\sigma}(\alpha)(\mathsf{wait})$ is empty, then no notification takes place; the property follows directly by induction.

The case for the assignment in the notifyAll-method is analogous, with the difference that all threads in the wait set get notified by ξ_1. The notifier observation sets the value of the auxiliary instance variable notified of α to $\grave{\sigma}(\alpha)(\mathsf{notified}) \dot\cup \grave{\sigma}(\alpha)(\mathsf{wait})$, whereas the corresponding wait variable gets the value $\emptyset$. By induction we have $\grave{\sigma}(\alpha)(\mathsf{notified}) \cap \grave{\sigma}(\alpha)(\mathsf{wait}) = \emptyset$, and thus the required properties are invariant under notification.

Case: New
Assume that the last step creates a new object (rule New), and executes the corresponding observation. Let $\alpha \in dom(\acute{\sigma})$. Then α either reference the newly

created object, or $\alpha \in dom(\grave{\sigma})$. In the first case $\alpha \notin dom(\grave{\sigma})$, and by the definition of global configurations (cf. page 10) there is no local configuration $(\alpha, \tau, stm) \in \grave{T}$. Since the last step doesn't add any local configurations to $\grave{T}$, we have $\alpha \neq \beta$ for all $(\beta, \tau, stm) \in \acute{T}$ and thus $\neg owns(\acute{T} \downarrow prog, \alpha)$. Since the lock of the new object is initialized to *free*, and wait and notified of α get the value $\emptyset$, the required property holds for the new object. In the second case, if $\alpha \in dom(\grave{\sigma})$, the property follows directly by induction.

Case: Call
Let $\alpha \in dom(\acute{\sigma})$. Then also $\alpha \in dom(\grave{\sigma})$. 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 (α, τ, stm) in $\acute{T}$ represents the execution of the invoked method.

If the invoked method is non-synchronized, then the property follows by induction. In the case of a synchronized method, let $\xi \in \grave{T}$ be the executing thread. The antecedent $\neg owns(\grave{T} \backslash \{\xi\} \downarrow prog, \alpha)$ implies by induction that, if there is no local configuration in the thread's stack executing a synchronized method of α then $\grave{\sigma}(\alpha)(\mathsf{lock}) = free$, and $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$ otherwise, where $(\alpha_0, \tau_0, stm_0)$ is the deepest configuration in the thread's stack and n the number of synchronized method invocations in the stack ξ. If in the state prior to the method invocation $\grave{\sigma}(\alpha)(\mathsf{lock}) = free$, then (α, τ, stm) is the only local configuration in $\acute{T}$ representing the execution of a synchronized method of α by a thread not in the wait or notified sets of α. Furthermore, the callee observation sets $\acute{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, 1)$, and thus the required property holds. In the second case, using the fact that the callee configuration is on top of its stack, the callee observation changes $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$ to $\acute{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n+1)$, and we get the property by Lemma 6 and by induction.

Case: Call$_{monitor}$
Similarly to the case Call, for $\alpha \in dom(\acute{\sigma})$ also $\alpha \in dom(\grave{\sigma})$, and if α is not the callee object, then the property holds by induction. In the case of the non-synchronized notify- and notifyAll-methods, none of the concerned variables or predicates are touched, and thus the property holds by induction again. So let $\xi \in \grave{T}$ be the executing thread invoking the non-synchronized wait-method of α.

The antecedent $owns(\xi \downarrow prog, \alpha)$ implies by induction $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$, where $(\alpha_0, \tau_0, stm_0)$ is the deepest configuration in the stack ξ and n is the number of its synchronized method invocations in α. Furthermore, since ξ does not yet execute a wait-method prior to the call, from $\xi \notin wait(\grave{T} \downarrow prog, \alpha) \cup notified(\grave{T} \downarrow prog, \alpha)$ we conclude by induction that α_0 is contained neither in wait or in notified of α in $\grave{\sigma}$.

The execution places the thread into α's wait set and, since at most one thread can own a lock at a time, it gives the lock of α free, i.e., we have $\neg owns(\acute{T} \downarrow prog, \alpha)$. The corresponding callee observation extends $\grave{\sigma}(\alpha)(\mathsf{wait})$ with (α_0, n), and sets the lock-value of α to *free*. Thus the case follows by induction.

Case: Return
Assume $\alpha \in dom(\acute{\sigma}) = dom(\grave{\sigma})$. If α is not the callee object, or if the invoked method is non-synchronized, then the property holds directly by induction. So

let $\xi \in \grave{T}$ be the thread of α_0 returning from a synchronized method of α; we denote the thread after execution by $\xi' \in \acute{T}$.

Since the wait-method is terminated by a syntactically different $\mathsf{return}_{getlock}$-statement, ξ is neither in the wait nor in the notified set of α, and we get by definition $owns(\xi \downarrow prog, \alpha)$ prior to execution. If the given method is the only synchronized method of α executed by ξ, then in the successor configuration $\neg owns(\xi' \downarrow prog, \alpha)$, and from the invariant property that at most one thread can own a lock at a time we imply $\neg owns(\acute{T} \downarrow prog, \alpha)$. Otherwise, if ξ has reentrant synchronized method invocations in α, then the thread doesn't give the lock free upon return, i.e., in the successor state we still have $owns(\xi' \downarrow prog, \alpha)$.

Using $owns(\xi \downarrow prog, \alpha)$, we get by induction $\grave{\sigma}(\alpha)(\mathsf{lock}) = (\alpha_0, n)$, where n is the number of invocations of synchronized methods of α by ξ. The auxiliary variable lock of α is set by the callee augmentation to *free*, if $n = 1$, and to $(\alpha_0, n-1)$, otherwise. Since the auxiliary variables wait and notified are not touched, the property follows by induction.

Case: $\textsc{Return}_{wait}$
Assume that the thread $\xi \in \grave{T}$ of an object α_0 is returning from the wait-method of $\alpha \in dom(\acute{\sigma}) = dom(\grave{\sigma})$; we denote the thread after execution by $\xi' \in \acute{T}$.

The semantics assures $\neg owns(\grave{T} \downarrow prog, \alpha)$ and by definition $\xi \in notified(\grave{T} \downarrow prog, \alpha)$. We get by induction $\grave{\sigma}(\alpha)(\mathsf{lock}) = free$ and $(\alpha_0, n) \in \grave{\sigma}(\alpha)(\mathsf{notified})$, where n is the number of invocations of synchronized methods of α by ξ. After returning, the thread gets removed from the *notified*-set of α and gathers the lock of α, i.e., $\xi' \notin notified(\acute{T} \downarrow prog, \alpha)$ and $owns(\xi' \downarrow prog, \alpha)$.

The augmentation of the wait-method removes (α_0, n) from $\grave{\sigma}(\alpha)(\mathsf{notified})$; from the uniqueness of the representation follows $\alpha_0 \neq \beta$ for all $(\beta, m) \in \acute{\sigma}(\alpha)(\mathsf{notified})$. Furthermore, the observation sets the lock of α to (α_0, n), by which we get the required property. □

Proof (of Lemma 8). Straightforward by the definition of augmentation. □

B.2.2 Proof of the soundness theorem

Proof (of the soundness Theorem 1). We proceed by induction on the length of the computation, simultaneously for all parts of Definition 6.

For the initial case let $dom(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \alpha]$, $\tau_0 = \tau_{init}[\mathsf{thread} \mapsto \alpha]$, and let $\langle \vec{y}_2 := \vec{e}_2 \rangle^{\iota}\ stm$ be the main statement. Then the initial configuration $\langle T'_0, \sigma'_0 \rangle$ of the proof outline satisfies the following: $\sigma'_0 = \sigma_0[\alpha.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]^{\sigma_0(\alpha),\tau_0}_{\mathcal{E}}]$, and for the stack we have $T'_0 = \{(\alpha, \tau'_0, stm)\}$ with $\tau'_0 = \tau_0[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]^{\sigma_0(\alpha),\tau_0}_{\mathcal{E}}]$.

Let ω be a logical environment referring only to values existing in σ_0. As in σ_0 there exists exactly one object α being in its initial instance state, we have

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

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

$$\omega[z \mapsto \alpha], \sigma_0 \models_{\mathcal{G}} (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init}$$

with I the class invariant of α,

$$f_{init} = [\mathsf{this}, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{u})/\vec{u}] \ , \text{ and}$$
$$f_{obs} = [\vec{E}_2(z)/z.\vec{y}_2] \, .$$

Applying Lemma 2, we get for the global invariant $\omega', \sigma_0' \models_{\mathcal{G}} GI$ for $\omega' = \omega[z \mapsto \alpha][\vec{u} \mapsto \tau_0'(\vec{u})]$. Since GI may not contain free logical variables, its value does not depend on the logical environment, and therefore $\omega, \sigma_0' \models_{\mathcal{G}} GI$.

Similarly for the local property $p_3 = pre(stm)$, we get with Lemma 2 that $\omega', \sigma_0' \models_{\mathcal{L}} P_3$. With Lemma 3 we get $\omega', \sigma_0'(\alpha), \tau_0' \models_{\mathcal{L}} pre(stm)$. Since $pre(stm)$ does not contain free logical variables, we get finally $\omega, \sigma_0'(\alpha), \tau_0' \models_{\mathcal{L}} pre(stm)$. Part 3 is analogous.

For the inductive step, assume $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$ such that $\langle \grave{T}, \grave{\sigma} \rangle$ satisfies the conditions of Definition 6. Let ω be a logical environment referring only to values existing in $\acute{\sigma}$. We distinguish on the kind of the computation step $\langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$.

Case: Ass_{inst}, Ass_{loc}
Note that signaling is represented in proof outlines by auxiliary assignments, thus this case covers also the rules Signal, SignalAll, and Signal_{skip}. Note furthermore that this case does not cover observations of communication or object creation.

Let the last computation step be the execution of an assignment in the local configuration $(\alpha, \grave{\tau}_1, \vec{y} := \vec{e}; stm_1) \in \grave{T}$ resulting in $(\alpha, \acute{\tau}_1, stm_1) \in \acute{T}$. According to the semantics, $\acute{\tau}_1 = \grave{\tau}_1[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$ and $\acute{\sigma} = \grave{\sigma}[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$.

Since assignments, that does not observe object creation or communication, don't change the values of variables occurring in GI, part (2) is satisfied.

For part (1), assume $(\beta, \tau_2, stm_2) \in \acute{T}$. If $(\beta, \tau_2, stm_2) = (\alpha, \acute{\tau}_1, stm_1)$ is the executing local configuration, then by induction $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$. The local correctness condition implies that $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} pre(stm_1)[\vec{e}/\vec{y}]$. Using the properties of the local substitution formulated in Lemma 1 we get $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} pre(stm_1)$.

If otherwise (β, τ_2, stm_2) is not the executing local configuration, then it is contained in $\grave{T}$. If $\alpha \neq \beta$, i.e., the execution didn't take place in β, then $\grave{\sigma}(\beta) = \acute{\sigma}(\beta)$, and thus $\omega, \acute{\sigma}(\beta), \tau_2 \models_{\mathcal{L}} pre(stm_2)$ by induction. Otherwise let τ be $\grave{\tau}_1[\vec{u}' \mapsto \tau_2(\vec{u})]$, where $\vec{u} = dom(\tau_2)$ and $\vec{u}'$ fresh. Then Lemma 6, the induction assumptions, and the definition of interleavable imply

$$\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre(stm_1) \wedge pre'(stm_2) \wedge \mathsf{interleavable}(pre(stm_2), \vec{y} := \vec{e}) \ ,$$

and with the interference freedom test we get $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}/\vec{y}]$. Using the substitution Lemma 1 and the fact that, due to the renaming mechanism, no variables in $\vec{u}'$ may occur in $\vec{y}$, yields $\omega, \acute{\sigma}(\alpha), \tau_2 \models_{\mathcal{L}} pre(stm_2)$.

Part (3) is similar, using the fact that the class invariant may contain instance variables only, and thus its evaluation doesn't depend on the local state.

Case: CALL

Let $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^\iota \, stm_1) \in \grave{T}$ be the caller configuration prior to method invocation, and let $(\alpha, \acute{\tau}_1, stm_1) \in \acute{T}$ and $(\beta, \acute{\tau}_2, stm_2) \in \acute{T}$ be the local configurations of the caller and the callee after execution. Let furthermore $\langle \vec{y}_2 := \vec{e}_2 \rangle^\iota \, stm_2$ be the invoked method's body and $\vec{u}$ its formal parameters. Then $\beta = [\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1} \neq null$. Directly after communication the callee has the local state $\hat{\tau}_2 = \tau_{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$; after the caller observation, the global state is $\hat{\sigma} = \grave{\sigma}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$ and the caller's local state is updated to $\acute{\tau}_1 = \grave{\tau}_1[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$. Finally, the callee observation updates its local state to $\acute{\tau}_2 = \hat{\tau}_2[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$ and the global state to $\acute{\sigma} = \hat{\sigma}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$.

If the method is synchronized and ξ is the stack of the executing thread in $\grave{T}$, then according to the transition rule $\neg owns(\grave{T} \backslash \{\xi\} \downarrow prog, \beta)$. Using Lemma 7 and Lemma 6 we get $\grave{\sigma}(\beta)(\mathsf{lock}) = free \vee thread(\grave{\sigma}(\beta)(\mathsf{lock})) = \grave{\tau}_1(\mathsf{thread})$ and thus $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ with $\vec{v}_1 = dom(\grave{\tau}_1)$ and $\grave{\omega} = \omega[z \mapsto \alpha][z' \mapsto \beta][\vec{v}_1 \mapsto \grave{\tau}_1(\vec{v}_1)]$.

Similarly, $\beta = [\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}$ implies with Lemma 3 and the definition of $\grave{\omega}$ that $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} E_0(z) = z'$.

In the following let $p_1 = pre(u_{ret} := e_0.m(\vec{e}))$, $p_2 = pre(\vec{y}_1 := \vec{e}_1)$, $p_3 = post(\vec{y}_1 := \vec{e}_1)$, $q_1 = I_q$, $q_2 = pre(\vec{y}_2 := \vec{e}_2)$, and $q_3 = post(\vec{y}_2 := \vec{e}_2)$, where I_q is the class invariant of the callee. Let I_p be caller class invariant. Then we have by induction $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI, \grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} I_p$, $\grave{\omega}, \grave{\sigma}(\beta), \grave{\tau}_1 \models_{\mathcal{L}} I_q$, and $\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_1$. Using the lifting lemma the cooperation test for communication implies

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z) \wedge Q_3'(z'))[E_2'(z')/z'.\vec{y}_2'][E_1(z)/z.\vec{y}_1][\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}'] \; ,$$

where $\vec{v}$ contains the variables from $dom(\tau_2)$ without the formal parameters $\vec{u}$. Using the lifting lemma again but in the reverse direction and Lemma 2 results $\omega, \acute{\sigma} \models_{\mathcal{G}} GI$, and thus part (2). Note that in the annotation no free logical variables occur, and thus the values of assertions in a proof outline do not depend on the logical environment.

Furthermore, using the same lemmas we get

$$\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3 \; \omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_3 \; .$$

Thus part (1) is satisfied for the local configurations involved in the last computation step. All other configurations (γ, τ_3, stm_3) in $\acute{T}$ are also in $\grave{T}$. If $\gamma \neq \alpha$ and $\gamma \neq \beta$, then $\grave{\sigma}(\gamma) = \acute{\sigma}(\gamma)$, and thus $\omega, \acute{\sigma}(\gamma), \tau_3 \models_{\mathcal{L}} pre(stm_3)$ by induction.

Assume next $\gamma = \alpha$ and $\alpha \neq \beta$, and let τ be $\grave{\tau}_1[\vec{v}' \mapsto \tau_3(\vec{v})]$, where $\vec{v} = dom(\tau_3)$. Then Lemma 6, the induction assumptions, and the definition of the assertion $\mathsf{interleavable}$ imply with the interference freedom test $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_3)[\vec{e}_1/\vec{y}_1]$. The substitution Lemma 1 and the fact that, due to the renaming mechanism, no local variables in $\vec{v}'$ occur in $\vec{y}_1$, yield $\omega, \hat{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}}$

$pre(stm_3)$. Now, since $\beta \neq \alpha$, the callee observation also does not change the caller's instance state, and we have $\hat{\sigma}(\alpha) = \acute{\sigma}(\alpha)$. Thus we get $\omega, \acute{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}} pre(stm_3)$.

The case $\gamma = \beta$ and $\alpha \neq \beta$ is similar. Communication and caller observation do not change the instance state of β, i.e., $\grave{\sigma}(\beta) = \hat{\sigma}(\beta)$. The interference freedom test results $\omega, \hat{\sigma}(\beta), \tau \models_{\mathcal{L}} pre'(stm_3)[\vec{e}_2/\vec{y}_2]$ with $\tau = \hat{\tau}_2[\vec{v}' \mapsto \tau_3(\vec{v})]$. Due to the renaming mechanism, we conclude with the local substitution lemma that $\omega, \acute{\sigma}(\beta), \acute{\tau} \models_{\mathcal{L}} pre'(stm_3)$ with $\acute{\tau}(\vec{v}') = \tau_3(\vec{v})$, and thus $\omega, \acute{\sigma}(\beta), \tau_3 \models_{\mathcal{L}} pre(stm_3)$.

For the last case $\gamma = \alpha = \beta$ note that, according to the restrictions on the augmentation, the caller may not change the instance state. Thus the same arguments as for $\gamma = \beta$ and $\alpha \neq \beta$ apply. I.e., part (1) is satisfied.

Part (3) is analogous: Let I be the class invariant of α. The interference freedom test implies $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} I$. Since I may contain instance variables only, its evaluation doesn't depend on the local state. Similarly for the callee, $\omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} I$. The state of other objects is not changed in the last computation step, and we get the required property.

Case: $\textsc{Call}_{start}$, $\textsc{Call}^{skip}_{start}$
These cases are analogous to the above one, where we additionally need $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \neg z'.\mathsf{started}$ and $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{started}$, resp., to be able to apply the cooperation test. The above properties result from the transition antecedents $\neg started(\grave{T}, \beta)$ and $started(\grave{T}, \beta)$, resp., using Lemma 8 and $\grave{\omega}(z') = \beta$.

Case: $\textsc{Call}_{monitor}$
As above, where $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ is implied by the transition antecedent $owns(\xi \downarrow prog, \beta)$ and Lemma 6.

Case: Return
This case is analogous to the Call case, where we define q_1 as the precondition of the corresponding return statement instead of the class invariant. The additional requirement $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$ of the cooperation test results from the fact that the values of formal parameters may not change during method execution, and that the method invocation statements may not contain instance variables, so that the values of the formal parameters and the expressions in the method invocation statement are untouched during the execution of the invoked method.

For the application of the interference freedom test, to show the validity of the $\mathsf{interleavable}$ predicate, we use the fact that the assertion $pre(stm_3)$ neither describes the caller nor the callee, since the corresponding local configuration is not involved in the execution.

Case: $\textsc{Return}_{run}$
Similar to the return case.

Case: $\textsc{Return}_{wait}$
In this case the antecedent $\neg owns(\grave{T} \downarrow prog, \beta)$ of the transition rule together with Lemma 7 imply $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free}$. Furthermore, the executing thread is in the notified set prior to execution, and the same lemma yields that the executing thread is registered in $\grave{\sigma}(\beta)(\mathsf{notified})$, i.e., $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{thread}' \in z'.\mathsf{notified}$.

Case: NEW
Let $(\alpha, \grave{\tau}_1, u := \mathsf{new}; \langle \vec{y}_1 := \vec{e}_1 \rangle^\iota \, stm_1) \in \grave{T}$ be the local configuration of the executing thread prior to object creation, and $(\alpha, \acute{\tau}_1, stm_1) \in \acute{T}$ after it. Object creation updates the global state to $\hat{\sigma} = \grave{\sigma}[\beta \mapsto \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta]]$, where $\beta \notin dom(\grave{\sigma})$; the executing thread's local state gets updated to $\hat{\tau}_1 = \grave{\tau}_1[u \mapsto \beta]$. After observation we have $\acute{\tau}_1 = \hat{\tau}_1[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$ and for the global state $\acute{\sigma} = \hat{\sigma}[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}_1}]$.

In the following let $p_1 = pre(u := \mathsf{new})$, $p_2 = pre(\vec{y}_1 := \vec{e}_1)$, and $p_3 = post(\vec{y}_1 := \vec{e}_1)$. Then we have by induction $\omega, \grave{\sigma} \models_{\mathcal{G}} GI$ and $\omega, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_1 \wedge I$, where I is the class invariant of the creator α. Using the lifting lemma we get $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI \wedge P_1(z) \wedge I(z)$ for $\grave{\omega} = \omega[z \mapsto \alpha][\vec{v}_1 \mapsto \grave{\tau}_1(\vec{v}_1)]$ and $\vec{v}_1$ the variables from the domain of $\grave{\tau}_1$. With Lemma 4 $\grave{\omega}[z' \mapsto dom(\grave{\sigma})][u \mapsto \beta], \hat{\sigma} \models_{\mathcal{G}} (GI \wedge (\exists u.\ P_1(z)) \wedge I(z)) \downarrow z'$. Note that GI may not contain free logical variables, and thus its evaluation does not depend on the logical environment. Since the newly created object with a fresh identity is in its initial instance state, $\grave{\omega}[z' \mapsto dom(\grave{\sigma})][u \mapsto \beta], \hat{\sigma} \models_{\mathcal{G}} \mathsf{Fresh}(z', u)$. Thus

$$\grave{\omega}[u \mapsto \beta], \hat{\sigma} \models_{\mathcal{G}} z \neq \mathsf{null} \wedge \exists z'.\ \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z' \ .$$

The cooperation test for object creation implies

$$\grave{\omega}[u \mapsto \beta], \hat{\sigma} \models_{\mathcal{G}} I_{\mathsf{new}}(u) \wedge (GI \wedge P_3(z))[\vec{E}_1(z)/z.\vec{y}_1] \ ,$$

where I_{new} is the class invariant of the new object. Using the lifting lemma again but in the reverse direction and Lemma 2 results $\omega, \acute{\sigma} \models_{\mathcal{G}} GI$, and thus part (2). Note that in the annotation no free logical variables occur, and thus the values of assertions do not depend on the logical environment.

Furthermore, using the substitution lemmas we get

$$\begin{aligned} \omega, \acute{\sigma}(\alpha), \acute{\tau}_1 &\models_{\mathcal{L}} p_3 \\ \omega, \acute{\sigma}(\beta), \tau &\models_{\mathcal{L}} I_{\mathsf{new}} \end{aligned}$$

for all τ. For the class invariant of the executing thread, the interference freedom test implies $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} I$, where I is the class invariant of α. Since I may contain instance variables only, its evaluation doesn't depend on the local state, and the required property holds. The state of other objects not involved in the last step is not changed in the last computation step, and part (3) is satisfied.

Furthermore, part (1) is satisfied for the local configuration involved in the last computation step. All other configurations $(\gamma, \grave{\tau}_2, stm_2)$ in $\acute{T}$ are also in $\grave{T}$ and $\gamma \neq \beta$. If $\gamma \neq \alpha$, then $\grave{\sigma}(\gamma) = \acute{\sigma}(\gamma)$, and thus $\omega, \acute{\sigma}(\gamma), \grave{\tau}_2 \models_{\mathcal{L}} pre(stm_2)$ by induction.

Assume now $\gamma = \alpha$, and let τ be $\hat{\tau}_1[\vec{v}' \mapsto \grave{\tau}_2(\vec{v})]$, where $\vec{v} = dom(\grave{\tau}_2)$. Then, since $\grave{\sigma}(\alpha) = \hat{\sigma}(\alpha)$, Lemma 6, the induction assumptions, and the definition of $\mathsf{interleavable}$ imply with the interference freedom test $\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} pre'(stm_2)[\vec{e}_1/\vec{y}_1]$. The substitution Lemma 1 and the fact that, due to the renaming mechanism, no local variables in $\vec{v}'$ occur in $\vec{y}_1$, yields $\omega, \acute{\sigma}(\alpha), \grave{\tau}_2 \models_{\mathcal{L}} pre(stm_2)$. I.e., part (1) is satisfied.

Proof (of the soundness Corollary 1). The proof is straightforward using the soundness Lemma 1.

B.3 Completeness

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

Lemma 12. *Let $\langle T, \sigma\rangle$ be a reachable configuration of $prog_0$ and assume $(\alpha, \tau, stm) \in T$. Then $\tau(\mathsf{loc}) \equiv stm$.*

Proof (of Lemma 12). Straightforward by the definition of augmentation. □

Proof (of the local merging Lemma 9). Let be given two computations $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle \acute{T}_2, \acute{\sigma}_2\rangle$ of $prog'$, and assume $(\alpha, \tau, stm) \in \acute{T}_1$ with $\alpha \in dom(\acute{\sigma}_1) \cap dom(\acute{\sigma}_2)$ and $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$. We prove $(\alpha, \tau, stm) \in \acute{T}_2$ by induction over the sum of the length of the computations.

In the initial case both $\acute{T}_1$ and $\acute{T}_2$ contain the same single initial local configuration, and thus the property holds.

For the inductive case, let $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ be the last steps of the computations. The augmentation definition implies that each computation step appends at most one element to the instance history of α. If $\grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{inst})$, then, by the definition of the augmentation, $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ did not execute in α, i.e., $(\alpha, \tau, stm) \in \grave{T}_1$, and the property follows by induction. The case for $\grave{\sigma}_2(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ is analogous. Thus assume in the following $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) \circ (\sigma^1_{inst}, \tau_1)$ and $\acute{\sigma}_2(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{inst}) \circ (\sigma^2_{inst}, \tau_2)$. From $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ we conclude that $\grave{\sigma}_1(\alpha)(\mathsf{h}_{inst}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{inst})$ and $(\sigma^1_{inst}, \tau_1) = (\sigma^2_{inst}, \tau_2)$.

Since $\acute{\sigma}_1(\alpha)(\mathsf{h}_{inst}) \neq \grave{\sigma}_1(\alpha)(\mathsf{h}_{inst})$, the computation step $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ executed some statements in α. If there is only one local configuration in α that was involved in the step, then the augmentation definition and the local substitution lemma imply that its resulting local configuration in $\acute{T}_1$ is given by (α, τ_1, stm_1) with $stm_1 \equiv \tau_1(\mathsf{loc})$. From $(\sigma^1_{inst}, \tau_1) = (\sigma^2_{inst}, \tau_2)$ we conclude that the same local configuration executed in $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$. Thus, either $(\alpha, \tau, stm) \in \acute{T}_1$ is the executing configuration (α, τ_1, stm_1) and then it is also in $\acute{T}_2$, or not, and then it is in $\grave{T}_1$, by induction in $\grave{T}_2$, and since it wasn't involved in the execution $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$, also in $\acute{T}_2$.

If otherwise there are two local configurations in α involved in $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$, then $(\sigma^1_{inst}, \tau_1)$ specifies the callee's local configuration for communication. However, due to the built-in auxiliary variables, the identity of the caller local configuration is also stored in τ_1. In the case of a method invocation, the identity of the caller local configuration is uniquely specified by the formal parameter caller of the callee. The caller configuration is in $\grave{T}_1$, and by induction in $\grave{T}_2$. Furthermore, since there are no two local configurations with the same identity in a reachable configuration, both steps execute in the same instance local configuration.

Thus, either $(\alpha, \tau, stm) \in \acute{T}_1$ is one of the executing configurations and then it is also in $\acute{T}_2$, or not, and then it is in $\grave{T}_1$, by induction in $\grave{T}_2$, and since it wasn't involved in the execution, also in $\acute{T}_2$. □

Proof (of the global merging Lemma 10). Let the configurations $\langle \acute{T}_1, \acute{\sigma}_1 \rangle$ and $\langle \acute{T}_2, \acute{\sigma}_2 \rangle$ be reachable and let $\alpha \in dom(\acute{\sigma}_1) \cap dom(\acute{\sigma}_2)$ satisfying $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$. We show that there exists a reachable $\langle \acute{T}, \acute{\sigma} \rangle$ with $dom(\acute{\sigma}) = dom(\acute{\sigma}_2)$, $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$, and $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \backslash \{\alpha\}$. We proceed by induction on the sum of the lengths of the computations.

In the base case we are given $\langle \acute{T}_1, \acute{\sigma}_1 \rangle = \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ and the property trivially holds.

For the inductive step, let $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ be the last steps of the computations.

If $\alpha \notin dom(\grave{\sigma}_1)$ or $\alpha \notin dom(\grave{\sigma}_2)$, then α was created in one of the last steps, and thus $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm}) = \epsilon$. That means, no methods of α were involved yet, i.e., α is in its initial instance state $\acute{\sigma}_1(\alpha) = \acute{\sigma}_2(\alpha) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \alpha]$; in this case $\langle \acute{T}_2, \acute{\sigma}_2 \rangle$ already satisfies the requirements. Assume in the following $\alpha \in dom(\grave{\sigma}_1) \cap dom(\grave{\sigma}_2)$.

We distinguish whether the last computation steps update the communication history of α or not.

Case: $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$
In this case $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ doesn't execute any non-self communication or object creation in α. By induction there is a computation $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma} \rangle$ leading to a configuration such that $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$ and $\grave{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \backslash \{\alpha\}$.

In case $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ does not execute in α at all, i.e., $\grave{\sigma}_1(\alpha) = \acute{\sigma}_1(\alpha)$, then $\langle \grave{T}, \grave{\sigma} \rangle$ already satisfies the requirements.

Otherwise, the local configurations in $\grave{T}_1$ which execute in α and which are involved in the computation step $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ are by the local merging Lemma 9 also in $\grave{T}$. Furthermore, from $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ we conclude that they don't execute any non-self communication or object creation, and thus their enabledness and effect depends only on the instance state of α. We conclude that the same computation as in $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$ can be executed in $\langle \grave{T}, \grave{\sigma} \rangle$, leading to a reachable global configuration satisfying the requirements.

Case: $\grave{\sigma}_2(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$
In this case $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ does not execute any non-self communication or object creation involving α. By induction, there is a reachable $\langle \grave{T}, \grave{\sigma} \rangle$ with $\grave{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$ and $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ for all $\beta \in dom(\grave{\sigma}_2) \backslash \{\alpha\}$.

If $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ performs a step within α, then, according to the case assumption, it executes exclusively within α. This means, $\grave{\sigma}_2(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \backslash \{\alpha\}$, and $\langle \grave{T}, \grave{\sigma} \rangle$ already satisfies the required properties.

If otherwise $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ does not execute in α, then all local configurations in $\grave{T}_2$, executing in an object different from α, are also in $\grave{T}$; this

follows from $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$ for all $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$, and with the help of the local merging Lemma 9 applied to $\langle \grave{T}, \grave{\sigma}\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle$. The enabledness of local configurations, whose execution does not involve α, are independent of the instance state of α; furthermore, the effect of their execution neither influences the instance state of α nor depends on it. Thus in $\langle \grave{T}, \grave{\sigma}\rangle$ we can execute the same computation steps as in $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$, leading to a reachable configuration with the required properties.

Case: $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) \neq \acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ and $\grave{\sigma}_2(\alpha)(\mathsf{h}_{comm}) \neq \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$
In this case finally both $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ execute some object creation or non-self communication in α. We show that in this case $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ implies also $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$, and thus by induction there is a computation leading to a configuration $\langle \grave{T}, \grave{\sigma}\rangle$ such that $dom(\grave{\sigma}) = dom(\grave{\sigma}_2)$, $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$, and $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ for all other objects $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$.

Furthermore, combining those local configurations involved in $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ which execute within α with those in $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ which execute outside α, we can define a computation $\langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$ such that $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$ and $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all other objects $\beta \in dom(\acute{\sigma}_2)\backslash\{\alpha\}$.

The case assumptions imply, that the last elements of the communication histories $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ and $\acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ were appended in the last computation steps; $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ imply that the last elements are equal.

According to the augmentation, each computation step extends the communication history of α with at most one element. Thus we get $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$, and by induction there is a reachable $\langle \grave{T}, \grave{\sigma}\rangle$ with $dom(\grave{\sigma}) = dom(\grave{\sigma}_2)$, $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$, and $\grave{\sigma}(\beta) = \grave{\sigma}_2(\beta)$ for all $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$.

Note that the last elements of the communication histories $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm})$ and $\acute{\sigma}_2(\alpha)(\mathsf{h}_{comm})$ record the kind of execution, and so we know that both steps execute the same kind of communication in α. Furthermore, the last elements record also the identity of the local configuration executing in α, the communication partner of α, and the communicated values, which are consequently also equal.

We distinguish on the kind of the computation step $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$:

Subcase: NEW
In this case $\acute{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) \circ (\alpha, null, (\mathsf{new}^c\gamma, thread_\alpha))$, where $thread_\alpha$ is the identity of the creator thread as specified by its local variable thread, and γ is the newly created object.

From the preliminary observations we conclude that $\langle \grave{T}_2, \grave{\sigma}_2\rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2\rangle$ creates the same new object γ being in the same initial state; furthermore, it leaves the states of all objects from $dom(\grave{\sigma}_2)\backslash\{\alpha\}$ untouched.

As $\grave{\sigma}(\alpha) = \grave{\sigma}_1(\alpha)$, the local merging Lemma 9 implies that the local configuration of the creator in $\grave{T}_1$ is also contained in $\grave{T}$. Thus, since $\gamma \notin dom(\grave{\sigma}_2) = dom(\grave{\sigma})$, the same computation step as in $\langle \grave{T}_1, \grave{\sigma}_1\rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1\rangle$ can be executed also in $\langle \grave{T}, \grave{\sigma}\rangle$, leading to a reachable configuration $\langle \acute{T}, \acute{\sigma}\rangle$ with $Val^{\mathsf{Object}}(\acute{\sigma}) = Val^{\mathsf{Object}}(\grave{\sigma}) \dot{\cup} \{\gamma\} = Val^{\mathsf{Object}}(\grave{\sigma}_2) \dot{\cup} \{\gamma\} = Val^{\mathsf{Object}}(\acute{\sigma}_2)$, $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$, and $\acute{\sigma}(\beta) = \grave{\sigma}(\beta) = \grave{\sigma}_2(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\grave{\sigma}_2)\backslash\{\alpha\}$. Finally, for the newly

created object we have $\acute{\sigma}(\gamma) = \acute{\sigma}_2(\gamma) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \gamma]$, and thus $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$ for all $\beta \in dom(\acute{\sigma}_2) \backslash \{\alpha\}$.

Subcase: CALL
Assume first that α is the caller object and $\beta \neq \alpha$ the callee. According to the preliminary observations, also $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ executes the invocation of the same method of β, where α is the caller and β the callee. Furthermore, by the local merging lemma, the caller local configuration from $\grave{T}_1$ is also in $\grave{T}$, and its execution is also enabled in $\langle \grave{T}, \grave{\sigma} \rangle$. The last property holds also for synchronized and monitor methods, since the invocation of the same method of β by the same thread is enabled in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$, and $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$.

Thus the caller local configuration from $\grave{T}_1$ can execute the method invocation in $\langle \grave{T}, \grave{\sigma} \rangle$, leading to a reachable configuration $\langle \acute{T}, \acute{\sigma} \rangle$ with $\acute{\sigma}(\alpha) = \acute{\sigma}_1(\alpha)$. Furthermore, $\langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$ and $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ execute the same callee observation in the same instance state $\grave{\sigma}_2(\beta) = \grave{\sigma}(\beta)$ and the same initial local state after the communication of the same actual parameter values, and thus $\acute{\sigma}(\beta) = \acute{\sigma}_2(\beta)$. The states of other objects are not touched, and thus $\langle \acute{T}, \acute{\sigma} \rangle$ satisfies the required properties.

Similarly, if the callee object is α, then the same caller local configuration as in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$ can execute in $\langle \grave{T}, \grave{\sigma} \rangle$ leading to a reachable configuration satisfying the requirements.

Subcase: RETURN
This case is analogous to the above case for CALL. The computation $\langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$ is constructed from the execution of the local configuration in α which executes in $\langle \grave{T}_1, \grave{\sigma}_1 \rangle \longrightarrow \langle \acute{T}_1, \acute{\sigma}_1 \rangle$, together with the execution of the communication partner of α which executes in $\langle \grave{T}_2, \grave{\sigma}_2 \rangle \longrightarrow \langle \acute{T}_2, \acute{\sigma}_2 \rangle$.

□

Lemma 13 (Initial correctness). *The proof outline prog′ satisfies the initial conditions of Definition 1.*

Proof (of Lemma 13). Let σ_0 be a global state with $dom(\sigma_0) = \{\alpha\}$ and $\sigma_0(\alpha) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \alpha]$, let $\{p_2\}^{\iota}\, \langle \vec{y}_2 := \vec{e}_2 \rangle^{\iota}\, \{p_3\}\ stm$ be the main statement with local variables $\vec{v}$, and let I be the class invariant of the main class. Then the initial configuration $\langle T'_0, \sigma'_0 \rangle$ of the proof outline $prog'$ satisfies $\sigma'_0 = \sigma_0[\alpha.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]^{\sigma_0(\alpha),\tau_0}_{\mathcal{E}}]$, and $T'_0 = \{(\alpha, \tau'_0, stm)\}$ with the local state τ_0 defined by $\tau_{init}[\mathsf{thread} \mapsto \alpha][\mathsf{caller} \mapsto (null, 0, null)]$ and with $\tau'_0 = \tau_0[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]^{\sigma_0(\alpha),\tau_0}_{\mathcal{E}}]$.

We have to show for arbitrary $\sigma \in \Sigma$ and $\omega \in \Omega$ referring only to values existing in σ, that

$$\omega, \sigma \models_{\mathcal{G}} \forall z.\ \mathsf{InitState}(z) \wedge (\forall z'.\ z' = \mathsf{null} \vee z = z') \rightarrow \\ P_2(z) \circ f_{init} \wedge (GI \wedge P_3(z) \wedge I(z)) \circ f_{obs} \circ f_{init}\ ,$$

where z is of the type of the main class, z' of type Object, and where $f_{init} = [z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]$ and $f_{obs} = [\vec{E}_2(z)/z.\vec{y}_2]$.

So let $v' \in Val^{\mathsf{main}}_{null}$. We observe that

$$\omega[z \mapsto v'], \sigma \models_{\mathcal{G}} \mathsf{InitState}(z) \wedge \forall z'.\ z' = \mathsf{null} \vee z' = z$$

implies that σ is the initial global state σ_0 defining exactly one existing object $\omega(z) = \alpha$ being in its initial instance state. We start transforming the right-hand side using the substitution Lemmas 2 and 3:

$$\begin{aligned}
&[\![P_2(z)[z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]]\!]^{\omega[z \mapsto \alpha], \sigma_0}_{\mathcal{G}} \\
&= [\![P_2(z)[z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}]]\!]^{\omega[z \mapsto \alpha][\vec{v} \mapsto \mathsf{Init}(\vec{v})], \sigma_0}_{\mathcal{G}} \\
&= [\![P_2(z)]\!]^{\omega[z \mapsto \alpha][\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha], \sigma_0}_{\mathcal{G}} \\
&= [\![p_2]\!]^{\omega, \sigma_0(\alpha), \tau_0}_{\mathcal{L}}
\end{aligned}$$

which evaluates to *true*, since the run-method of the main class is initially invoked in the given context.

For the global invariant we get similarly

$$\begin{aligned}
&[\![GI[\vec{E}_2(z)/z.\vec{y}_2][z, (\mathsf{null}, 0, \mathsf{null})/\mathsf{thread}, \mathsf{caller}][\mathsf{Init}(\vec{v})/\vec{v}]]\!]^{\omega[z \mapsto \alpha], \sigma_0}_{\mathcal{G}} \\
&= [\![GI[\vec{E}_2(z)/z.\vec{y}_2]]\!]^{\omega[z \mapsto \alpha][\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha], \sigma_0}_{\mathcal{G}} \\
&= [\![GI]\!]^{\omega', \sigma'_0}_{\mathcal{G}} \\
&= [\![GI]\!]^{\omega, \sigma'_0}_{\mathcal{G}}
\end{aligned}$$

for some logical environment ω'. In the last step we used the restriction that the global invariant may not contain free logical variables. The step before made use of the following equation for $\vec{E}_2(z)$, which we get using Lemma 3 and with the fact that $\vec{e}_2$ does not contain logical variables:

$$\begin{aligned}
[\![\vec{E}_2(z)]\!]^{\omega[z \mapsto \alpha][\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha], \sigma_0}_{\mathcal{G}} &= [\![\vec{e}_2[z/\mathsf{this}]]\!]^{\omega[z \mapsto \alpha][\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha], \sigma_0}_{\mathcal{G}} \\
&= [\![\vec{e}_2]\!]^{\omega[z \mapsto \alpha][\vec{v} \mapsto \mathsf{Init}(\vec{v})][\mathsf{thread} \mapsto \alpha], \sigma_0(\alpha), \tau_0}_{\mathcal{G}} \\
&= [\![\vec{e}_2]\!]^{\omega', \sigma_0(\alpha), \tau_0}_{\mathcal{G}} .
\end{aligned}$$

Since $\langle T'_0, \sigma'_0 \rangle$ is reachable, the initial condition for the global invariant is satisfied. The cases for p_3 and I are similar to that of GI, where we additionally use the lifting substitution Lemma 3 to show that $[\![P(z)]\!]^{\omega', \sigma'_0}_{\mathcal{G}} = [\![p]\!]^{\omega', \sigma'_0(\alpha), \tau'_0}_{\mathcal{L}}$. □

Lemma 14 (Local correctness). *The proof outline prog′ satisfies the conditions of local correctness from Definition 2.*

Proof (of Lemma 14). Let c be a class of *prog′*, $\omega \in \Omega$, $\sigma_{inst} \in \Sigma_{inst}$, and $\tau \in \Sigma_{loc}$ with $\sigma_{inst}(\mathsf{this}) = \alpha$. Assume an assignment $\{p_1\}\ \vec{y} := \vec{e}\ \{p_2\}$ in c with class invariant I. We have to show that

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p_1 \wedge I \rightarrow p_2[\vec{e}/\vec{y}] \ . \tag{10}$$

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(\vec{y} := \vec{e})$ it follows that there is a reachable $\langle \grave{T}, \grave{\sigma} \rangle$ with $\grave{\sigma}(\alpha) = \sigma_{inst}$ and $(\alpha, \tau, \vec{y} := \vec{e}; stm) \in \grave{T}$. Executing the local configuration in $\langle \grave{T}, \grave{\sigma} \rangle$ leads to a reachable global configuration $\langle \acute{T}, \acute{\sigma} \rangle$ with $\acute{\sigma}(\alpha) = \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}]$ and $(\alpha, \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], stm) \in \acute{T}$. 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. □

Lemma 15 (Interference freedom). *The proof outline $prog'$ satisfies the conditions for interference freedom from Definition 3.*

Proof (of Lemma 15). Assume an arbitrary assignment $\vec{y} := \vec{e}$ with precondition p in class c with class invariant I, and an arbitrary assertion q at a control point in the same class. We show the verification condition from Equation (5) on page 34

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p \wedge q' \wedge I \wedge \mathsf{interleavable}(q, \vec{y} := \vec{e}) \rightarrow q'[\vec{e}/\vec{y}] \ , \tag{11}$$

for some logical environment ω together with some instance and local states σ_{inst} and τ, where q' denotes q with all local variables u replaced by some fresh local variables u'.

Let $\alpha = \sigma_{inst}(\mathsf{this})$. The first clause $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ implies that there exists a computation reaching $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ with $\grave{\sigma}_p(\alpha) = \sigma_{inst}$, and an enabled configuration $(\alpha, \tau_p, stm_p; stm'_p) \in \grave{T}_p$, where stm_p is $\vec{y} := \vec{e}$ if the assignment does not observe method call or object creation, and the corresponding communication statement with its observation otherwise. The local state τ_p is τ if stm_p does not receive any values. Otherwise $\tau_p = \tau[\vec{u} \mapsto \vec{v}]$, where $\vec{u}$ are the variables storing the received values and $\vec{v}$ some value sequence, such that the local configuration is enabled to receive the values $\tau(\vec{u})$. If p is the precondition of a method body, then additionally $\tau_p(\vec{w}) = \mathsf{Init}(\vec{w})$ for the sequence $\vec{w}$ of local variables in p that are not formal parameters.

From $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'$ we get by renaming back the local variables that $\omega, \sigma_{inst}, \tau' \models_{\mathcal{L}} q$ for $\tau'(u) = \tau(u')$ for all local variables u in q. Let q be the precondition of the statement stm_q. Note that q is an assertion at a control point. Applying the annotation definition we conclude that there is a reachable $\langle \grave{T}_q, \grave{\sigma}_q \rangle$ with $\grave{\sigma}_q(\alpha) = \sigma_{inst} = \grave{\sigma}_p(\alpha)$ and $(\alpha, \tau', stm_q; stm'_q) \in \grave{T}_q$. The local merging Lemma 9 implies that $(\alpha, \tau', stm_q; stm'_q) \in \grave{T}_p$.

Let $\langle \acute{T}_p, \acute{\sigma}_p \rangle$ result from $\langle \grave{T}_p, \grave{\sigma}_p \rangle$ by executing the enabled local configuration $(\alpha, \tau_p, stm_p; stm'_p)$. If the local configuration is the caller part in a self-communication, then, due to the restriction on the augmentation, $[\vec{e}/\vec{y}]$ does not

substitute any instance variables. Thus, due to the renaming mechanism, $q'[\vec{e}/\vec{y}]$ equals q', and thus $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$.

Otherwise, if $(\alpha, \tau_p, stm_p; stm'_p)$ doesn't represent the caller part in a self-communication, then $\acute{\sigma}_p(\alpha) = \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}]$. Note that for self-communication, the caller part does not change the instance state. Thus the only update of the instance state of α is given by the effect of $\vec{y} := \vec{e}$. From the assumption $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \mathsf{interleavable}(q, \vec{y} := \vec{e})$ we get that $(\alpha, \tau', stm_q; stm'_q)$ cannot be the communication partner of $(\alpha, \tau_p, stm_p; stm'_p)$, and thus $(\alpha, \tau', stm_q; stm'_q) \in \acute{T}_p$.

We get $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], \tau' \models_{\mathcal{L}} q$, and after renaming the local variables of q also $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], \tau \models_{\mathcal{L}} q'$. Finally, by the substitution Lemma 1 we get the required property $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} q'[\vec{e}/\vec{y}]$. Note that due to renaming, no local variables of q' occur in $\vec{y}$, and thus $\tau(u') = \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}](u')$ for all local variables u in q.

Validity of the verification condition 4 for the class invariant is similar, where we additionally use the fact the the class invariant refers to instance variables only. □

Lemma 16 (Cooperation test: Communication). *The proof outline $prog'$ satisfies the verification conditions of the cooperation test for communication of Definition 4.*

Proof (of Lemma 16). We distinguish on the kind of communication starting with the verification condition for synchronized method invocation.

Case: Call

Let $\{p_1\}\, u_{ret} := e_0.m(\vec{e}); \{p_2\}^1\, \langle \vec{y}_1 := \vec{e}_1 \rangle^1\, \{p_3\}^2$ be a statement in a class c of $prog'$ with $e_0 \in Exp_c^{c'}$, where method $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ of c' is synchronized with body $\{q_2\}^1\, \langle \vec{y}_2 := \vec{e}_2 \rangle^1\, \{q_3\}\, stm$, formal parameters $\vec{u}$, local variables without the formal parameters given by $\vec{v}$, and let $q_1 = I_{c'}$ be the callee class invariant. Assume

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI \wedge P_1(z) \wedge Q'_1(z') \wedge \mathsf{communicating} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}$$

for distinct and fresh $z \in LVar^c$ and $z' \in LVar^{c'}$, and where $\mathsf{communicating} = E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$. Note that for completeness we don't need the information stored in the caller class invariant. By definition of the global invariant, the assumption $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} GI$ implies that there exists a reachable $\langle T, \sigma \rangle$ with

$$dom(\grave{\sigma}) = dom(\sigma) \text{ and } \grave{\sigma}(\gamma)(\mathsf{h}_{comm}) = \sigma(\gamma)(\mathsf{h}_{comm}) \text{ for all } \gamma \in dom(\sigma)\,.$$

Assuming $\grave{\omega}(z) = \alpha$ as caller identity, $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_1(z)$ implies $\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_1$ by the substitution Lemma 3, for some local state $\grave{\tau}_1$ with $\grave{\tau}_1(u) = \grave{\omega}(u)$ for all local variables u occurring in p_1. By the annotation definition there exists a reachable configuration $\langle T_1, \sigma_1 \rangle$ such that

$$\sigma_1(\alpha) = \grave{\sigma}(\alpha) \text{ and } (\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1 \rangle^1\, stm_1) \in T_1\,.$$

Recall that $\sigma(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\sigma)$, and especially for the caller $\sigma(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}(\alpha)(\mathsf{h}_{comm}) = \sigma_1(\alpha)(\mathsf{h}_{comm})$. Using the global merging Lemma 10 applied to $\langle T_1, \sigma_1\rangle$ and $\langle T, \sigma\rangle$ we get that there is a reachable $\langle T', \sigma'\rangle$ with $dom(\sigma') = dom(\sigma)$ and

$$\sigma'(\alpha) = \sigma_1(\alpha) \text{ and } \sigma'(\gamma) = \sigma(\gamma) \text{ for all } \gamma \in dom(\sigma)\backslash\{\alpha\}\,.$$

Furthermore, $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^\iota\ stm_1) \in T_1$, $\sigma_1(\alpha) = \sigma'(\alpha)$, and the local merging Lemma 9 implies that

$$(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^\iota\ stm_1) \in T'\,.$$

Let $\beta = \grave{\omega}(z')$ be the callee object. In case of a self-call, i.e., for $\alpha = \beta$, we directly get that $\langle T'', \sigma''\rangle = \langle T', \sigma'\rangle$ is a reachable configuration such that $\sigma''(\alpha) = \grave{\sigma}(\alpha)$, $\sigma''(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\grave{\sigma})$, and $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^\iota\ stm_1) \in T''$.

Otherwise, the assumption $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} I_{c'}(z')$ implies $\grave{\omega}, \grave{\sigma}(\beta), \tau_2 \models_{\mathcal{L}} I_{c'}$ for some local state τ_2. Note that the class invariant contains instance variables, only. By definition of the class invariant, there is a reachable global configuration $\langle T_2, \sigma_2\rangle$ such that

$$\sigma_2(\beta) = \grave{\sigma}(\beta)\,.$$

We need to fall back upon the two merging lemmas once more to obtain a common reachable configuration: Analogously to the caller part, the global merging Lemma 10 applied to $\langle T_2, \sigma_2\rangle$ and $\langle T', \sigma'\rangle$ yields that there is a reachable configuration $\langle T'', \sigma''\rangle$ with $dom(\sigma'') = dom(\sigma')$ and

$$\sigma''(\beta) = \sigma_2(\beta) \text{ and } \sigma''(\gamma) = \sigma'(\gamma) \text{ for all } \gamma \in dom(\sigma')\backslash\{\beta\}\,.$$

Furthermore, $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^\iota\ stm_1) \in T'$, $\sigma''(\alpha) = \sigma'(\alpha)$, and the local merging Lemma 9 implies $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^\iota\ stm_1) \in T''$.

Thus $\langle T'', \sigma''\rangle$ is a reachable configuration with $\sigma''(\alpha) = \grave{\sigma}(\alpha)$, $\sigma''(\beta) = \grave{\sigma}(\beta)$, $\sigma''(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\grave{\sigma})$, and $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^\iota\ stm_1) \in T''$.

With the antecedent $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ of the cooperation test we get $\grave{\sigma}(\beta)(\mathsf{lock}) = \mathit{free} \vee \mathit{thread}(\grave{\sigma}(\beta)(\mathsf{lock})) = \grave{\tau}_1(\mathsf{thread})$. With $\grave{\sigma}(\beta) = \sigma''(\beta)$ and Lemma 7 we get $\neg \mathit{owns}(T''\backslash\{\xi\}, \beta)$, where ξ is the stack with $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^\iota\ stm_1)$ on top. Furthermore, $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \mathsf{comm}$ implies $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} E_0(z) = z'$, and by the lifting substitution lemma $[\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1} = [\![e_0]\!]_{\mathcal{E}}^{\sigma''(\alpha), \grave{\tau}_1} = \omega(z') = \beta$. This means, the invocation of method m of β is enabled in the local configuration $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^\iota\ stm_1)$ in $\langle T'', \sigma''\rangle$.

The definition of the augmentation, and $\sigma''(\alpha) = \grave{\sigma}(\alpha)$ gives

$$\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_2\,,$$

which by the substitution Lemma 3 and with the definition of $\grave{\tau}_1$ yields $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_2(z)$. Due to the renaming mechanism we get

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} P_2(z) \circ f_{comm}$$

for $f_{comm} = [\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$. For the precondition of the method body, the annotation definition implies

$$\grave{\omega}, \grave{\sigma}(\beta), \hat{\tau}_2 \models_{\mathcal{L}} q_2$$

with $\hat{\tau}_2 = \tau_{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}]$. For the actual parameters we obtain by the substitution Lemma 3 $[\![\vec{E}(z)]\!]_{\mathcal{G}}^{\grave{\omega}, \grave{\sigma}} = [\![\vec{e}]\!]_{\mathcal{L}}^{\grave{\omega}, \grave{\sigma}(\alpha), \grave{\tau}_1} = [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}$, and further with the same lemma

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} Q_2'(z')[\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$$

as required by the cooperation test.

Directly after communication we have a global configuration with still the same global state σ''. The caller observation evolves its own local state to $\acute{\tau}_1 = \grave{\tau}_1[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma''(\alpha), \grave{\tau}_1}]$, and the global state to $\hat{\sigma} = \sigma''[\alpha.\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\sigma''(\alpha), \grave{\tau}_1}]$. Finally, the callee observation changes the global state to $\acute{\sigma} = \hat{\sigma}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$, where its own local state is updated to $\acute{\tau}_2 = \hat{\tau}_2[\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$. According to the annotation definition we get

$$\grave{\omega}, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3, \quad \grave{\omega}, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_3, \quad \text{and} \quad \grave{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI\,.$$

Let $\acute{\omega} = \grave{\omega}[\vec{v}' \mapsto \mathit{Init}(\vec{v})][\vec{u}' \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}][\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha), \grave{\tau}_1}][\vec{y}_2' \mapsto [\![\vec{e}_2']\!]_{\mathcal{E}}^{\hat{\sigma}(\beta), \hat{\tau}_2}]$. The lifting lemma implies $\acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI \wedge P_3(z) \wedge Q_3'(z')$; with the global substitution lemma finally

$$\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z) \wedge Q_3'(z'))[\vec{E}_2'(z')/z'.\vec{y}_2'][\vec{E}_1(z)/z.\vec{y}_1][\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']\,,$$

and thus the cooperation test is satisfied for the invocation of synchronous methods.

The case for non-synchronized methods is analogous, where the antecedent $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ is dropped.

Case: $\textsc{Call}_{monitor}$
This case is similar to the above one of Call, where for the invocation of a method $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$, the assertion comm is given by $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$, implying $\mathit{owns}(\xi, \beta)$ for the caller thread ξ and the callee object β.

Case: $\textsc{Call}_{start}$
Enabledness of starting the thread of an object β requires $\neg \mathit{started}(T'', \beta)$. Due to the definition of comm, we have additionally $\grave{\omega}, \sigma'' \models_{\mathcal{G}} \neg z'.\mathsf{started}$, which implies $\neg \sigma''(\beta)(\mathsf{started})$. We get enabledness by Lemma 8 assuring $\mathit{started}(T'', \beta)$ iff $\sigma''(\beta)(\mathsf{started})$.

Case: $\textsc{Call}^{skip}_{start}$
The enabledness argument is similar for $\textsc{Call}^{skip}_{start}$, where we use $\dot{\omega}, \sigma'' \models_{\mathcal{G}} z'.\mathsf{started}$ to imply the enabledness predicate $started(T'', \beta)$.

Case: $\textsc{Return}$
For return, the construction of $\langle T'', \sigma'' \rangle$ is similar, where we get instead of the enabledness of the caller that the callee configuration $(\beta, \dot{\tau}_2, \mathsf{return}\ e_{ret}; \langle \vec{y}_3 := \vec{e}_3 \rangle^{\iota})$ is in $\langle T'', \sigma'' \rangle$, and thus enabled to execute.

Case: $\textsc{Return}_{wait}$
In this case we additionally have to show $\neg owns(T'', \beta)$, which we get from the comm assertion implying $\dot{\omega}, \dot{\sigma} \models_{\mathcal{G}} z'.\mathsf{lock} = \mathsf{free}$ and using Lemma 7.

Case: $\textsc{Return}_{run}$
Since the run-method cannot be invoked directly, we conclude that the executing local configuration is the only one in its stack, i.e., the transition rule $\textsc{Return}_{run}$ of the semantics can be applied in $\langle T'', \sigma'' \rangle$ to terminate the callee $(\beta, \dot{\tau}_2, \mathsf{return}; \langle \vec{y}_3 := \vec{e}_3 \rangle^{\iota})$.

□

Lemma 17 (Cooperation test: Instantiation). *The proof outline $prog'$ satisfies the verification conditions of the cooperation test for object creation of Definition 5.*

Proof (of Lemma 17). Let $\{p_1\}\, u := \mathsf{new}^c; \{p_2\}^{\iota}\, \langle \vec{y} := \vec{e} \rangle^{\iota}\, \{p_3\}$ be a statement in class c' of $prog'$, and assume

$$\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} z \neq \mathsf{null} \wedge z \neq u \wedge \exists z'.\ \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u(P_1(z))) \downarrow z'$$

with $z \in LVar^{c'}$ and $z' \in LVar^{\mathsf{list\,Object}}$ fresh. Note that we don't need the class invariant of the creator for completeness. We show that

$$\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z))[\vec{E}(z)/z.\vec{y}]\ .$$

Let $\hat{\omega}(z) = \alpha$ and $\hat{\omega}(u) = \beta$. According to the semantics of assertions we have that

$$\omega, \hat{\sigma} \models_{\mathcal{G}} \mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'$$

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

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

where $\mathsf{InitState}(u)$ expands to $u \neq \mathsf{null} \wedge \bigwedge_{x \in IVar_c} u.x = \mathsf{Init}(x)$. Thus, $\omega, \hat{\sigma} \models_{\mathcal{G}} \mathsf{Fresh}(z', u)$ implies that $\beta \in Val^c(\hat{\sigma})$ with $\hat{\sigma}(\beta) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta]$, and additionally $Val^{\mathsf{Object}}_{null}(\hat{\sigma}) = \omega(z') \dot{\cup} \{\beta\}$. Let $\dot{\sigma}$ be the global state with domain

$Val^{\mathsf{Object}}(\grave{\sigma}) = Val^{\mathsf{Object}}(\hat{\sigma})\backslash\{\beta\}$ and such that $\grave{\sigma}(\gamma) = \hat{\sigma}(\gamma)$ for all $\gamma \in Val^{\mathsf{Object}}(\grave{\sigma})$. Then $\hat{\sigma} = \grave{\sigma}[\beta \mapsto \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta]]$, and from

$$\omega, \hat{\sigma} \models_{\mathcal{G}} (GI \wedge \exists u.\ P_1(z)) \downarrow z'$$

we get with Lemma 4

$$\omega, \grave{\sigma} \models_{\mathcal{G}} GI \wedge \exists u.\ P_1(z)\ .$$

By definition of the annotation, $\omega, \grave{\sigma} \models_{\mathcal{G}} GI$ implies that there is a reachable configuration $\langle \grave{T}_1, \grave{\sigma}_1 \rangle$ such that

$$dom(\grave{\sigma}_1) = dom(\grave{\sigma}) \text{ and } \grave{\sigma}_1(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm}) \text{ for all } \gamma \in dom(\grave{\sigma})\,.$$

The precondition of the object creation statement

$$\omega, \grave{\sigma} \models_{\mathcal{G}} \exists u.\ P_1(z)$$

implies

$$\omega[u \mapsto Z], \grave{\sigma} \models_{\mathcal{G}} P_1(z)$$

for some $Z \in Val^{\mathsf{Object}}_{null}(\grave{\sigma})$. Applying the lifting Lemma 3 we get that

$$\omega, \grave{\sigma}(\alpha), \grave{\tau} \models_{\mathcal{L}} p_1$$

for a local state $\grave{\tau}$ with $\grave{\tau}(u) = Z$ and $\grave{\tau}(v) = \omega(v)$ for all other local variables v. By definition of the annotation, there is a reachable global configuration $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$ such that

$$\grave{\sigma}_2(\alpha) = \grave{\sigma}(\alpha) \text{ and } (\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{\iota}\ stm) \in \grave{T}_2\,.$$

Recall that $\grave{\sigma}_1(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}(\gamma)(\mathsf{h}_{comm})$ for all $\gamma \in dom(\grave{\sigma})$; especially we have $\grave{\sigma}_1(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}(\alpha)(\mathsf{h}_{comm}) = \grave{\sigma}_2(\alpha)(\mathsf{h}_{comm})$. Using the global merging Lemma 10 applied to the reachable global configurations $\langle \grave{T}_2, \grave{\sigma}_2 \rangle$ and $\langle \grave{T}_1, \grave{\sigma}_1 \rangle$ we get that there is a reachable configuration $\langle \grave{T}_3, \grave{\sigma}_3 \rangle$ with

$dom(\grave{\sigma}_3){=}dom(\grave{\sigma}_1)$, $\grave{\sigma}_3(\alpha) = \grave{\sigma}_2(\alpha)$, and $\grave{\sigma}_3(\gamma) = \grave{\sigma}_1(\gamma)$ for all $\gamma{\in}dom(\grave{\sigma}_1)\backslash\{\alpha\}$.

Furthermore, $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{\iota}\ stm) \in \grave{T}_2$, $\grave{\sigma}_2(\alpha) = \grave{\sigma}_3(\alpha)$, and the local merging Lemma 9 implies that $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{\iota}\ stm) \in \grave{T}_3$.

So we know that $\langle \grave{T}_3, \grave{\sigma}_3 \rangle$ is a reachable configuration containing the local configuration $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^{\iota}\ stm) \in \grave{T}_3$. With $Val^{\mathsf{Object}}(\grave{\sigma}) = Val^{\mathsf{Object}}(\hat{\sigma})\backslash\{\beta\}$, $dom(\grave{\sigma}_1) = dom(\grave{\sigma})$, and $dom(\grave{\sigma}_3){=}dom(\grave{\sigma}_1)$ we get that $\beta \notin dom(\grave{\sigma}_3)$, i.e., the local configuration is enabled to create the fresh object $\beta = \omega(u)$. With $\grave{\sigma}_3(\alpha) = \grave{\sigma}_2(\alpha) = \hat{\sigma}(\alpha)$ we get

$$\omega, \hat{\sigma}(\alpha), \hat{\tau} \models_{\mathcal{L}} p_2\ ,$$

where $\hat{\tau} = \grave{\tau}[u \mapsto \beta]$; with the lifting Lemma 3 together with the definition of $\grave{\tau}$ this means $\omega, \hat{\sigma} \models_{\mathcal{G}} P_2(z)$, as required in the cooperation test.

Executing the instantiation in the local configuration $(\alpha, \grave{\tau}, u := \mathsf{new}^c; \langle \vec{y} := \vec{e} \rangle^\iota \; stm)$ in $\langle \grave{T}_3, \grave{\sigma}_3 \rangle$, creating a new object $\beta \notin dom(\grave{\sigma}_3)$, results in $\langle \hat{T}_3, \hat{\sigma}_3 \rangle$ with $\hat{\sigma}_3 = \grave{\sigma}_3[\beta \mapsto \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta]]$; executing the creator observation leads to a reachable $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ with $\acute{\sigma}_3 = \hat{\sigma}_3[\alpha.\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}_3(\alpha), \hat{\tau}}]$ and $(\alpha, \acute{\tau}, stm)$ in $\acute{T}_3$ with $\acute{\tau} = \hat{\tau}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}_3(\alpha), \hat{\tau}}]$.

As $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ is reachable with $\acute{\sigma}_3(\beta) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta] = \hat{\sigma}(\beta)$ we know

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

As I_c may not contain local variables, applying the lifting Lemma 3 again with $\omega(u) = \beta$ yields the required condition $\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} I_c(u)$ for the class invariant. It remains to show that

$$\hat{\omega}, \hat{\sigma} \models_{\mathcal{G}} (GI \wedge P_3(z))[\vec{E}(z)/z.\vec{y}] \,.$$

Applying the substitution Lemma 2 and the fact that GI does not contain free logical variables yields

$$[\![GI[\vec{E}(z)/z.\vec{y}]]\!]_{\mathcal{G}}^{\hat{\omega}, \hat{\sigma}} = [\![GI]\!]_{\mathcal{G}}^{\hat{\omega}, \acute{\sigma}}$$

with $\acute{\sigma} = \hat{\sigma}[\alpha.\vec{y} \mapsto [\![\vec{E}(z)]\!]_{\mathcal{G}}^{\hat{\omega}, \hat{\sigma}}]$. Thus we have to show the existence of a reachable configuration with a global state defining the same object domain and communication history values as $\acute{\sigma}$. The configuration $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ satisfies the above requirements, since, first, it is reachable with

$$\begin{aligned} Val^{\mathsf{Object}}(\acute{\sigma}_3) &= Val^{\mathsf{Object}}(\grave{\sigma}_3) \dot{\cup} \{\beta\} = Val^{\mathsf{Object}}(\grave{\sigma}_1) \dot{\cup} \{\beta\} \\ &= Val^{\mathsf{Object}}(\grave{\sigma}) \dot{\cup} \{\beta\} = Val^{\mathsf{Object}}(\hat{\sigma}) = Val^{\mathsf{Object}}(\acute{\sigma}) \,. \end{aligned}$$

Furthermore, $\acute{\sigma}_3(\alpha) = \hat{\sigma}_3(\alpha)[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}_3(\alpha), \hat{\tau}}]$, and with $\hat{\sigma}_3(\alpha) = \grave{\sigma}_3(\alpha) = \grave{\sigma}_2(\alpha) = \hat{\sigma}(\alpha)$ and

$$[\![\vec{E}(z)]\!]_{\mathcal{G}}^{\hat{\omega}, \hat{\sigma}} = [\![\vec{e}[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\hat{\omega}, \hat{\sigma}} = [\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}(\alpha), \hat{\tau}} = [\![\vec{e}]\!]_{\mathcal{E}}^{\hat{\sigma}_3(\alpha), \hat{\tau}} \,,$$

we get $\acute{\sigma}_3(\alpha) = \acute{\sigma}(\alpha)$. For the new object, $\acute{\sigma}_3(\beta) = \hat{\sigma}_3(\beta) = \sigma^{init}_{inst}[\mathsf{this} \mapsto \beta] = \hat{\sigma}(\beta) = \acute{\sigma}(\beta)$. Finally, for all other objects γ different from both α and β from the domain of $\acute{\sigma}$ we have $\acute{\sigma}_3(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}_3(\gamma)(\mathsf{h}_{comm}) = \grave{\sigma}_1(\gamma)(\mathsf{h}_{comm}) = \acute{\sigma}(\gamma)(\mathsf{h}_{comm})$.

Similarly for the postcondition p_3 of the observation,

$$[\![P_3(z)[\vec{E}(z)/z.\vec{y}]]\!]_{\mathcal{G}}^{\hat{\omega}, \hat{\sigma}} = [\![P_3(z)]\!]_{\mathcal{G}}^{\hat{\omega}, \acute{\sigma}} = [\![p_3[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\hat{\omega}, \acute{\sigma}} = [\![p_3]\!]_{\mathcal{L}}^{\hat{\omega}, \acute{\sigma}(\alpha), \acute{\tau}} = [\![p_3]\!]_{\mathcal{L}}^{\hat{\omega}, \acute{\sigma}_3(\alpha), \acute{\tau}} \,.$$

Thus we have to show the existence of a reachable configuration with a global state defining the same instance state for α as $\acute{\sigma}_3$ and containing the local configuration $(\alpha, \acute{\tau}, stm)$. The configuration $\langle \acute{T}_3, \acute{\sigma}_3 \rangle$ satisfies the above requirements.

□

Proof (of Theorem 2). Straightforward using the Lemmas 13, 14, 15, 16, and 17.