An Assertion-based Proof System for Multithreaded Java

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

[a] *Christian-Albrechts-University Kiel, Germany*
[b] *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 synchronous message passing, dynamic thread creation, shared-variable concurrency via instance variables, and coordination via reentrant synchronization monitors.

To reason about safety properties of multithreaded *Java* programs, we introduce an *assertional proof method* for a multithreaded sublanguage of *Java*, covering the mentioned concurrency issues as well as the object-based core of *Java*. The verification method is formulated in terms of proof-outlines, where the assertions are layered into local ones specifying the behavior of a single instance, and global ones taking care of the connections between objects. We establish the soundness and the relative completeness of the proof system. From an annotated program, a number of verification conditions are generated and handed over to the interactive theorem prover *PVS*.

Key words: *Java*, multithreading, monitors, verification, Hoare-logic, soundness and relative completeness

Email addresses: eab@informatik.uni-freiburg.de (Erika Ábrahám), F.S.de.Boer@cwi.nl (Frank S. de Boer), wpr@informatik.uni-kiel.de (Willem-Paul de Roever), ms@informatik.uni-kiel.de (Martin Steffen).
[1] Currently at University Freiburg

1 Introduction

Besides the features of a class-based object-oriented language, *Java* integrates *concurrency* via its thread classes. *Java*'s semantical foundations [1] have been thoroughly studied ever since the language gained widespread popularity (e.g. [2–4]). The research concerning *Java*'s proof theory mainly focussed on *sequential* sub-languages (e.g. [5–7]). This work presents a sound and relatively complete assertional proof system for $Java_{synch}$, a subset of *Java*, featuring dynamic object creation, object references with aliasing, method invocation, and, specifically, concurrency and *Java*'s monitor discipline.

The behavior of a $Java_{synch}$ program results from the concurrent execution of methods. To support a clean interface between internal and external object behavior, the state of an object can be accessed from the outside only via the object's methods and not directly via qualified references $e.x$ to instance variables x. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only. To mirror this modularity, the assertional logic and the proof system are formulated at 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.

The language and the proof system for partial correctness are presented incrementally in three stages, starting with a sequential sublanguage, which is extended by concurrency and monitor synchronization in next steps. The proof systems are formulated in terms of *proof outlines* [8], i.e., of programs *augmented* by auxiliary variables and *annotated* with Hoare-style assertions [9,10]. To obtain a complete proof system, i.e., which allows to prove each invariant property of a program, it must be possible to express the strongest invariant property, which is reachability and which in general depends not only on the current values of variables, but also on other control information. Therefore, the standard route to achieve completeness is to represent the missing control information in the states in so-called *auxiliary variables.* Of course, The incremental development shows, which information must be additionally represented at the different stages for completeness. For method calls, already in the sequential case, we use auxiliary variables to identify communicating partners in method calls. Additionally, in the multithreaded case, we additionally need auxiliary variables to identify threads, and to capture monitor synchronization at the third stage.

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* [8,11], 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 two cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules in [12] and in [11] for CSP.

Our proof method is *modular* in the sense that it allows for separate interference freedom and cooperation tests (Figure 1). 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. This restriction could be released, without loosing the mentioned modularity, but it would increase the complexity of the proof system.

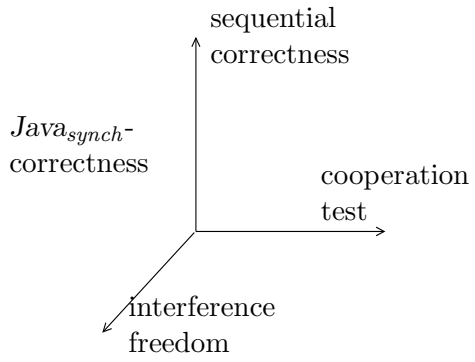

Fig. 1. Modularity of the proof system

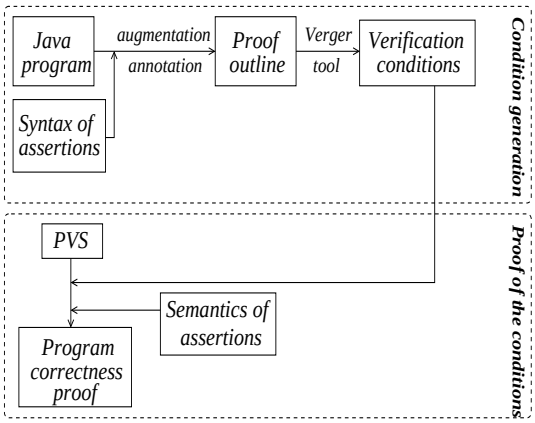

Fig. 2. The verification process

For readability, the verification conditions in this paper are formulated as standard Hoare-triples $\{\varphi\}stm\{\psi\}$. The meaning of these partial correctness formulas is, that if *stm* is executed in a state satisfying φ, and the execution terminates, then the resulting state satisfies ψ. In [13] we reformulate these Hoare-triples to logical implications using substitutions.

Computer-support is given by the tool *Verger* (*VERification condition GEneratoR*), taking a proof outline as input and generating the verification conditions as output. We use the interactive theorem prover PVS [14] to verify the conditions, for which we only need to encode the semantics of the assertion language (cf. Figure 2). In the verification conditions we model assignments by substitutions, instead of more semantic approaches [15,16,6,7], which use an explicit encoding of the semantics of assignments.

The remainder of the paper is structured as follows. We start in Section 2 with a *sequential,* class-based sublanguage of *Java* and its proof system, featuring dynamic object creation and method invocation. This level shows how to handle activities of a single *thread* of execution. At the second stage we include *concurrency* in Section 3. The proof system is extended to handle dynamic thread creation, interleaving, and shared variable concurrency. Finally, we integrate *Java*'s *monitor synchronization* mechanism in Section 4. Soundness and completeness are discussed in Section 5. Section 6 shows how we can prove deadlock freedom, and Section 7 discusses related and future work.

2 The sequential sublanguage

In this section we start with a sequential language, ignoring concurrency issues of *Java*, which will be added in later sections. Furthermore —and throughout the paper— we concentrate on the object-based core of *Java*, i.e., we disregard *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.

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 created dynamically, and communicate via *method invocation,* i.e., synchronous message passing.

The languages we consider are strongly typed. Besides class types c, they support booleans Bool and integers Int as primitive types, furthermore pairs $t \times t$ and lists $\mathsf{list}\, t$ as composite types. Each domain is equipped with a standard set of operators. Without inheritance and subtyping, the type system is rather straightforward. Throughout the paper, we tacitly assume all constructs of the abstract syntax to be well-typed, without further explicating the static semantics here. We thus work with a type-annotated abstract syntax where we omit the explicit mentioning of types when no confusion can arise.

2.1 Syntax

The abstract syntax of the sequential language $Java_{seq}$ is summarized in Table 1. Though we use the abstract syntax for the theoretical part of this work, our tool supports *Java* syntax.

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

Table 1
$Java_{seq}$ abstract syntax

For variables, we notationally distinguish between *instance* variables $x \in IVar$ and *local* or *temporary* variables $u \in TVar$. Instance variables 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. We use $Var = IVar \dot{\cup} TVar$ for the set of program variables with typical element y. The set $IVar^c$ of instance variables of a class c is given implicitly by the instance variables occurring in the class; the set of local variables of method declarations is given similarly.

Besides using instance and local variables, *expressions* $e \in Exp$ are built from the self-reference this, the empty reference null, and from subexpressions using the given operators. To support a clean interface between internal and external object behavior, we disallow qualified references to instance variables.

As *statements* $stm \in Stm$, we allow assignments, object creation, method invocation, and standard control constructs like sequential composition, conditional statements, and iteration. We 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 of the form $stm; \mathsf{return}\ e_{ret}$, i.e., we require that method bodies are terminated by a single return statement, giving back the control and possibly a return value. The set $Meth_c$ contains the methods of class c. We denote the body of method m of class c by $body_{m,c}$. A *class* is defined by its name c and its methods, whose names are assumed to be distinct. 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.[2] The run-method cannot be invoked.

Besides the mentioned simplifications on the type system, we impose for technical reasons the following restrictions: We require that method invocation and object creation statements contain only local variables, i.e., that none of the expressions $e_0, \ldots, e_n$ 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, and thus we can use their equality to describe caller-callee dependencies when returning from a method call. The above restrictions could be released by storing the identity of the callee object and the values of the formal and actual parameters in additional built-in auxiliary variables. However, the restrictions simplify the proof system and thus they make it easier to understand the basic ideas of this work. 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. Also this restriction could be released, but it would increase the complexity of the proof system.

2.2 Semantics

2.2.1 States and configurations

Let Val^t be the disjoint domains of the various types t and $Val = \dot{\bigcup}_t\ Val^t$, where $\dot{\cup}$ is the disjoint union operator. For class names c, the disjunct sets Val^c with typical elements $\alpha, \beta, \ldots$ denote infinite sets of *object identifiers.* The value of the empty reference null in type c is $null^c \notin Val^c$. 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, and $Val_{null} = \dot{\bigcup}_t\ Val^t_{null}$. Let $Init : Var \to Val_{null}$ be a function assigning an initial value to each variable $y \in Var$, 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 this $\notin Var$, i.e., the self-reference is not in the

[2] In *Java*, the entry point of a program is given by the static main-method of the main class. Relating the abstract syntax to that of *Java*, we assume that the main class is a `Thread`-class whose main-method just creates an instance of the main class and starts its thread. The reason for this restriction is, that *Java*'s main-method is static, but our proof system does not support static methods and variables.

domain of *Init*.[3]

A *local state* $\tau \in \Sigma_{loc}$ of type $TVar \rightharpoonup Val_{null}$ is a partial function holding the values of the local variables of a method. The initial local state $\tau_{init}^{m,c}$ of method m of class c assigns to each local variable u of m the value $Init(u)$. A *local configuration* (α, τ, stm) of a thread executing within an object α 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 $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$, representing the call chain of the thread. We write $\xi \circ (\alpha, \tau, stm)$ for pushing a new local configuration onto the stack.

An object is characterized by its *instance state* $\sigma_{inst} \in \Sigma_{inst}$, a partial function of type $IVar \dot\cup \{\mathsf{this}\} \rightharpoonup Val_{null}$, which assigns values to the self-reference this and to the instance variables. The initial instance state $\sigma_{inst}^{c,init}$ of instances of class c assigns a value from Val^c to this, and to each of its remaining instance variables x the value $Init(x)$. A *global state* $\sigma \in \Sigma$ of type $(\dot\bigcup_c Val^c) \rightharpoonup \Sigma_{inst}$ stores for each currently *existing* object, i.e., an object belonging to the domain $dom(\sigma)$ of σ, its instance state. The set of existing objects of type c in a state σ is given by $Val^c(\sigma)$, and $Val^c_{null}(\sigma) = Val^c(\sigma) \dot\cup \{null^c\}$. For the remaining types, $Val^t(\sigma)$ and $Val^t_{null}(\sigma)$ are defined correspondingly, $Val(\sigma) = \dot\bigcup_t Val^t(\sigma)$, and $Val_{null}(\sigma) = \dot\bigcup_t Val^t_{null}(\sigma)$. A *global configuration* $\langle T, \sigma\rangle$ describes the currently existing objects by the global state σ, where the set T contains the configuration of the executing thread. For the concurrent languages of the later sections, T will be the set of configurations of all currently executing threads. 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.

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; $\sigma_{inst}[x \mapsto v]$ is defined analogously, where $\sigma[\alpha.x \mapsto v]$ results from σ by assigning v to the instance variable x of object α. We use these operators analogously for vectors of variables. We use $\tau[\vec{y} \mapsto \vec{v}]$ also for arbitrary variable sequences, where instance variables are untouched; $\sigma_{inst}[\vec{y} \mapsto \vec{v}]$ and $\sigma[\alpha.\vec{y} \mapsto \vec{v}]$ are analogous. Finally for global states, $\sigma[\alpha \mapsto \sigma_{inst}]$ equals σ except on α; note that in case $\alpha \notin Val(\sigma)$, the operation extends the set of existing objects by α, which has its instance state initialized to σ_{inst}.

2.2.2 Operational semantics

Expressions are evaluated with respect to an *instance local* state (σ_{inst}, τ), where the instance state gives meaning to the instance variables and the self-reference, whereas the local state determines the values of the local vari-

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

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

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

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

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

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

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

Fig. 3. *Java*$_{seq}$ operational semantics

ables. The main cases of the evaluation function are $[\![x]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \sigma_{inst}(x)$ and $[\![u]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \tau(u)$. The operational semantics of *Java*$_{seq}$ is given inductively by the rules of Figure 3 as transitions between global configurations. The rules are formulated such a way that we can re-use them for the concurrent languages of the later sections. Note that for the sequential language, the sets T in the rules are empty, since there is only one single thread in global configurations. We elide the rules for the remaining sequential constructs —sequential composition, conditional statement, and iteration— as they are standard.

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 $dom(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma_{inst}^{c,init}[\mathsf{this} \mapsto \alpha]$, and $T_0 = \{(\alpha, \tau_{init}^{\mathsf{run},c}, body_{\mathsf{run},c})\}$, where c is the main class, and $\alpha \in Val^c$.

A configuration $\langle T, \sigma\rangle$ of a program is *reachable* if there exists a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$ where $\langle T_0, \sigma_0\rangle$ is the initial configuration of the program and

$\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$. A local configuration $(\alpha, \tau, stm) \in T$ is *enabled* in $\langle T, \sigma\rangle$, if it can be executed, 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 (see 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. Call). After initializing the local state and passing the parameters, the thread begins to execute the method body. When returning from a method call (cf. 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 have similar rules not shown in the figure for the invocation of methods without return value. The executing thread ends its lifespan by returning from the run-method of the initial object (see Return_{run}).

2.3 The assertion language

The assertion logic consists of a local and a global sublanguage. *Local* assertions $p, q, \ldots$ 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 $P, Q, \ldots$ describe a whole system of objects and their communication structure and will be used in the cooperation test. In the assertion language we add the type Object as the supertype of all classes, and we introduce *logical variables* z different from all program variables. Logical variables are used for quantification and as free variables to represent local variables in the global assertion language. Expressions and assertions are interpreted relative to a logical environment ω, assigning values to logical variables.

Assertions are boolean program expressions, extended by logical variables and quantification.[4] Global assertions may furthermore contain qualified references. Quantification can be used for all types, also for reference types. However, the existence of objects dynamically depends on the *global* state, something one cannot speak about at the local level. Nevertheless, one can assert the existence of objects on the local level, provided one is explicit about the domain of quantification. Thus quantification over objects in the local assertion language is restricted to $\forall z \in e.\, p$ for objects and to $\forall z \sqsubseteq e.\, p$ for lists of objects, and correspondingly for existential quantification and for composite types. Unrestricted quantification $\forall z.\, p$ can be used in the local language

[4] In this paper we use mathematical notation like $\forall z.\, p$ etc. for phrases in abstract syntax. The concrete syntax used by the *Verger* tool is an adaptation of *JML*.

for boolean and integer domains only. Global assertions are evaluated in the context of a global state. Thus, quantification is allowed unrestricted for all types and ranges over the set of *existing* values.

The evaluations of local and global assertions are given by $[\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}$ and $[\![P]\!]_{\mathcal{G}}^{\omega,\sigma}$. The main cases are shown in Table 2. We write $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $[\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = \mathit{true}$, and $\models_{\mathcal{L}} p$ if p holds in all contexts; we use analogously $\models_{\mathcal{G}}$ for global assertions.

$$
\begin{array}{rcl}
([\![\exists z.\, p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = \mathit{true}) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega[z \mapsto v],\sigma_{inst},\tau} = \mathit{true} \text{ for some } v \in \mathit{Val}) \\
([\![\exists z{\in}e.\, p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = \mathit{true}) & \text{iff} & ([\![z{\in}e \wedge p]\!]_{\mathcal{L}}^{\omega[z \mapsto v],\sigma_{inst},\tau} {=} \mathit{true} \text{ for some } v \in \mathit{Val}_{null}) \\
[\![E.x]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \\
([\![\exists z.\, P]\!]_{\mathcal{G}}^{\omega,\sigma} = \mathit{true}) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega[z \mapsto v],\sigma} = \mathit{true} \text{ for some } v \in \mathit{Val}_{null}(\sigma))
\end{array}
$$

Table 2
Semantics of assertions

To express a local property p in the global assertion language, we define the *lifting substitution* $p[z/\mathsf{this}]$ by simultaneously replacing in p all occurrences of this by z, and transforming all occurrences of instance variables x into qualified references $z.x$. We assume z 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 will write $P(z)$ for $p[z/\mathsf{this}]$, and similarly for expressions.

2.4 The proof system

The proof system has to accommodate for dynamic object creation, aliasing, and method invocation. Before describing the proof method we first show how to augment and annotate programs resulting in *proof outlines* or *asserted programs.*

2.4.1 Proof outlines

For a complete proof system it is necessary that the transition semantics of Java_{seq} 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. Invariant program properties are specified by the *annotation.*

An *augmentation* extends a program by *observations*, which are atomically executed multiple assignments $\vec{y} := \vec{e}$ to auxiliary variables. Furthermore, the observations have to be "attached" in an atomic manner to statements they observe. For object creation this is syntactically represented by the augmentation $u := \mathsf{new}^c\langle \vec{y} := \vec{e}\rangle^{new}$ which attaches the observation to the object creation statement. Observations $\vec{y}_1 := \vec{e}_1$ of a method call and observations $\vec{y}_4 := \vec{e}_4$ of the corresponding reception of a return value[5] are denoted by $u := e_0.m(\vec{e})\langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\langle \vec{y}_4 := \vec{e}_4\rangle^{?ret}$. The augmentation $\langle \vec{y}_2 := \vec{e}_2\rangle^{?call} stm; \mathsf{return}\ e_{ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}$ of method bodies specifies $\vec{y}_2 := \vec{e}_2$ as the observation of the reception of the method call, and $\vec{y}_3 := \vec{e}_3$ as the observation attached to the return statement. Assignments can be observed using $\vec{y} := \vec{e}\langle \vec{y}' := \vec{e}'\rangle^{ass}$. 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.

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 calls, communication, sender, and receiver observations are executed in a single computation step, in this order. Points between a statement and its observation are no *control points*, since the statement and its observation are executed in a single computation step; we call them *auxiliary points*. We use the expression *multiple assignment* to refer generally to statements of the following forms: assignment statements together with their observations, unobserved assignments, stand-alone observations, as well as observations of communication and object creation.

Besides the auxiliary variables defined by the user, we have three *built-in* auxiliary variables, described in the following. In order to express the transition semantics in the logic, we identify each local configuration by a pair consisting of the object in which it executes and a unique object-internal identifier. The latter is stored in a built-in auxiliary local variable conf, and its uniqueness is assured by the auxiliary instance variable $\mathsf{counter}$, incremented for each new local configuration in that object. The callee receives the "return address" as auxiliary formal parameter caller of type $\mathsf{Object} \times \mathsf{Int}$, storing the identities of the caller object and the calling local configuration. The parameter caller of the initial invocation of the run-method of the initial object gets the value $(null, 0)$.

Syntactically, the built-in augmentation translates each method definition $m(\vec{u})\{stm\}$ into $m(\vec{u}, \mathsf{caller})\{\langle \mathsf{conf}, \mathsf{counter} := \mathsf{counter}, \mathsf{counter} + 1\rangle^{?call} stm\}$.

[5] To exclude the possibility, that two multiple assignments get executed in a single computation step in the same object, we require that caller observations in a self-communication may not change the values of instance variables [17].

Correspondingly, method invocation statements $u := e_0.m(\vec{e})$ get extended to $u := e_0.m(\vec{e}, (\mathsf{this}, \mathsf{conf}))$.

For readability, in the examples of the following sections we will not explicitly list the built-in augmentation; they are meant to be automatically included.

To specify invariant properties of the system, the augmented programs are *annotated* by attaching local assertions to each control and auxiliary point. 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

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

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 the observation. The annotation

$$\{p_0\}\, u := e_0.m(\vec{e}) \quad \{p_1\}^{!call}\, \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \quad \{p_2\}^{wait} \quad \{p_3\}^{?ret}\, \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} \quad \{p_4\}$$

assigns p_0 and p_4 as pre- and postconditions to the method invocation; p_1 and p_3 are assumed to hold directly after method call and return, resp., but prior to their observations; p_2 describes the control point of the caller after method call and before return. The annotation of method bodies $stm; \mathsf{return}\ e$ is as follows:

$$\{p_0\}^{?call}\ \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call}\ \{p_1\} \quad stm; \quad \{p_2\}\ \mathsf{return}\ e\ \{p_3\}^{!ret}\ \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret}\ \{p_4\}$$

The callee postcondition of the method call is p_1; the callee pre- and postconditions of return are p_2 and p_4. The assertions p_0 resp. p_3 specify the states of the callee between method call resp. return and its observation. Note that method annotations do not syntactically specify the state prior to call, i.e., there is no precondition of method invocations from the callee side. Semantically, this precondition is the class invariant.

Besides pre- and postconditions, the annotation defines for each class c a local assertion I_c called *class invariant*, specifying invariant properties of instances of c in terms of its instance variables. 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. We require that in the annotation no free logical variables occur. In the following we will also use partial annotation. Assertions which are not explicitly specified are by definition true.

2.4.2 Verification conditions

The proof system generates a number of *verification conditions* which inductively ensure that for each reachable configuration the local assertions attached to the current control points in the thread configuration as well as the global and the class invariants hold. The conditions are grouped, as usual, into initial conditions (which are not discussed in this paper, see [17]), and for the inductive step into local correctness and tests for interference freedom and cooperation.

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.

Local correctness A proof outline is *locally correct*, if the properties of method instances, as specified by the annotation, are invariant under the execution of the given method instance. For example, the precondition of an assignment must imply its postcondition after its execution. The following condition is required to hold for all multiple assignments being an assignment statement with its observation, an unobserved assignment, or a stand-alone observation:

Definition 2.1 (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 object creation or communication,*

$$\models_{\mathcal{L}} \{p_1\} \quad \vec{y} := \vec{e} \quad \{p_2\}\ . \tag{1}$$

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*.

The interference freedom test Invariance of local assertions under computation steps in which they are not involved is assured by the proof obligations of the *interference freedom test*. Its definition covers also invariance of the class invariants. Since $Java_{seq}$ does not support qualified references to instance variables, we only have to deal with 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 cover invariance of assertions at control points over assignments in the same object, including observations of communication and object creation. To distinguish local variables of the different local configurations, we rename those of the assertion.

Let q be an assertion at a control point and $\vec{y} := \vec{e}$ a multiple assignment in the same class c. In which cases does q have to be invariant under the execution of the assignment? Since the language is sequential, i.e., q and $\vec{y} := \vec{e}$ belong to the *same* thread, the only assertions endangered are those at control points waiting for return earlier in the current execution stack. Invariance of a local configuration under its own execution, however, need not be considered and is excluded by requiring $\mathsf{conf} \neq \mathsf{conf}'$. Interference with the *matching* return statement in a self-communication need also not be considered, because communicating partners execute simultaneously. Let $\mathsf{caller\_obj}$ be the first and $\mathsf{caller\_conf}$ the second component of caller. We define $\mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})$ by

- $\mathsf{conf}' \neq \mathsf{conf}$, for assertions $\{q\}^{wait}$ attached to control points waiting for return, if $\vec{y} := \vec{e}$ is not the observation of return;
- $\mathsf{conf}' \neq \mathsf{conf} \wedge (\mathsf{this} \neq \mathsf{caller\_obj} \vee \mathsf{conf}' \neq \mathsf{caller\_conf})$, for assertions $\{q\}^{wait}$, if $\vec{y} := \vec{e}$ observes return;
- false, otherwise.

The interference freedom test can now be formulated as follows:

Definition 2.2 (Interference freedom) *A proof outline is* interference free, *if for all classes c and multiple assignments $\vec{y} := \vec{e}$ with precondition p in c,*

$$\models_{\mathcal{L}} \{p \wedge I_c\} \quad \vec{y} := \vec{e} \quad \{I_c\} \,. \tag{2}$$

Furthermore, for all assertions q at control points in c,

$$\models_{\mathcal{L}} \{p \wedge q' \wedge \mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})\} \quad \vec{y} := \vec{e} \quad \{q'\} \,. \tag{3}$$

Remember that q' stands for the assertion q with each local variable appropriately renamed, e.g., the variable conf is replaced by conf' etc. Note further that if we would allow program expressions to contain qualified references to instance variables, we would have to show interference freedom of all assertions under all assignments in programs, not only for those occurring in the same class. For a program with n classes where each class contains k assignments and l assertions at control points, the number of interference freedom conditions is in $O(n{\cdot}k{\cdot}l)$, instead of $O((n{\cdot}k) \cdot (n{\cdot}l))$ with qualified references.

Example 2.3 *Let* $\{p_1\}\mathsf{this}.m(\vec{e})\ \{p_2\}^{!call}\langle stm_1\rangle^{!call}\{p_3\}^{wait}\{p_4\}^{?ret}\langle stm_2\rangle^{?ret}\{p_5\}$ *be an annotated method call statement in a method* m' *of a class* c *with an integer auxiliary instance variable* x*, such that each assertion implies* $\mathsf{conf} = x$*. I.e., the identity of the executing local configuration is stored in the instance variable* x*. The annotation expresses that no pairs of control points in* m' *of* c *can be simultaneously reached.*

The assertions p_2 *and* p_4 *need not be shown invariant, since they are attached to auxiliary points. Interference freedom neither requires invariance of the assertions* p_1 *and* p_5*, since they are not at control points waiting for return, and thus the antecedents of the corresponding conditions evaluate to false. Invariance of* p_3 *under the execution of the observation* stm_1 *with precondition* p_2 *requires validity of* $\models_{\mathcal{L}} \{p_2 \wedge p_3' \wedge \mathsf{waits\_for\_ret}(p_3, stm_1)\}\ stm_1\ \{p_3'\}$*. The assertion* $p_2 \wedge p_3' \wedge \mathsf{waits\_for\_ret}(p_3, stm_1)$ *implies* $(\mathsf{conf} = x) \wedge (\mathsf{conf}' = x) \wedge (\mathsf{conf}' \neq \mathsf{conf})$*, which evaluates to false. Invariance of* p_3 *under* stm_2 *follows analogously.*

Example 2.4 *Assume a partially*[6] *annotated method invocation statement of the form* $\{p_1\}\mathsf{this}.m(\vec{e})\ \{\mathsf{conf} = x \wedge p_2\}^{wait}\{p_3\}$ *in a class* c *with an integer auxiliary instance variable* x*, and assume that method* m *of* c *has the annotated return statement* $\{q_1\}\ \mathsf{return}\ \{\mathsf{caller} = (\mathsf{this}, x)\}^{!ret}\langle stm\rangle^{!ret}\{q_2\}$*. The annotation expresses that the local configurations containing the above statements are in caller-callee relationship. Thus upon return, the control point of the caller moves from the point at* $\mathsf{conf} = x \wedge p_2$ *to that at* p_3*, i.e.,* $\mathsf{conf} = x \wedge p_2$ *does not have to be invariant under the observation of the return statement.*

Again, the assertion $\mathsf{caller} = (\mathsf{this}, x)$ *at an auxiliary point does not have to be shown invariant. For the assertions* p_1*,* p_3*,* q_1*, and* q_2*, which are not at a control point waiting for return, the antecedent is false. Invariance of* $\mathsf{conf} = x \wedge p_2$ *under the observation stm with precondition* $\mathsf{caller} = (\mathsf{this}, x)$ *is covered by the interference freedom condition*

$$\models_{\mathcal{L}} \{\ \mathsf{caller} = (\mathsf{this}, x) \wedge (\mathsf{conf}' = x \wedge p_2') \wedge \mathsf{waits\_for\_ret}((\mathsf{conf} = x \wedge p_2), stm)\ \}\ stm\ \{\mathsf{conf}' = x \wedge p_2'\}\ .$$

The $\mathsf{waits\_for\_ret}$ *assertion implies* $\mathsf{caller} \neq (\mathsf{this}, \mathsf{conf}')$*, which contradicts the assumptions* $\mathsf{caller} = (\mathsf{this}, x)$ *and* $\mathsf{conf}' = x$*; thus the antecedent of the condition is false.*

Satisfaction of $\mathsf{conf} = x \wedge p_2$ *after the call, satisfaction of* $\mathsf{caller} = (\mathsf{this}, x)$ *directly after return, and satisfaction of* p_3 *and* q_2 *after the observation stm is assured by the cooperation test.*

[6] As already mentioned, missing assertions are by definition true.

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. Additionally, the preconditions of the corresponding observations must hold immediately after communication. The global invariant expresses global invariant properties using auxiliary instance variables which can be changed by observations of communication, only. Consequently, the global invariant 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 start with the cooperation test for method invocation. The communication pattern of method call and return and the involved local assertions are illustrated in Figure 4. Control points are represented by •'s, and auxiliary points by ∘'s.

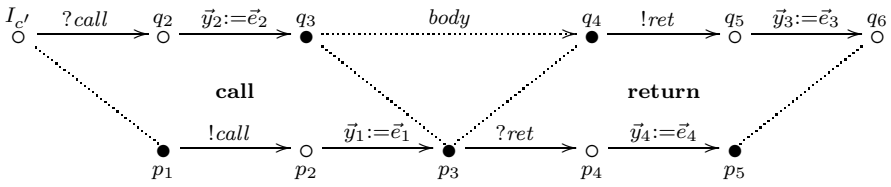

Fig. 4. Cooperation test: Communication

Since different objects may be involved, the cooperation test is formulated in the global assertion language. Local properties are expressed in the global language using the lifting substitution. 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. To avoid name clashes between local variables of the partners, we rename those of the callee. Remember that after communication, i.e., after creating and initializing the callee local configuration and passing on the actual parameters, first the caller, and then the callee execute their corresponding observations, all in a single computation step. Correspondingly for return, after communicating the result value, first the callee and then the caller observation gets executed.

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 communicating partners is captured in the assertion comm, which

depends on the type of communication: For method invocation $e_0.m(\vec{e})$, the assertion $E_0(z) = z'$ states, that z' is indeed the callee object. 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 at returning from a method to identify partners being in caller-callee relationship, using the built-in auxiliary variables. Thus for the return case, comm additionally states $\vec{u}' = \vec{E}(z)$, where $\vec{u}$ and $\vec{e}$ are the formal and the actual parameters. Returning from the run-method terminates the executing thread, which does not have communication effects.

As in the previous conditions, state changes are represented by assignments. For the example of method invocation, communication is represented by the assignment $\vec{u}' := \vec{E}(z)$, where initialization of the remaining local variables $\vec{v}$ is covered by $\vec{v}' := \mathsf{Init}(\vec{v})$. The assignments $z.\vec{y}_1 := \vec{E}_1(z)$ and $z'.\vec{y}_2' := \vec{E}_2'(z')$ stand for the caller and callee observations $\vec{y}_1 := \vec{e}_1$ and $\vec{y}_2 := \vec{e}_2$, executed in the objects z and z', respectively. Note that we rename all local variables of the callee to avoid name clashes.

Definition 2.5 (Cooperation test for communication) *A proof outline satisfies the* cooperation test for communication*, if*

$$\begin{aligned} &\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\} \\ &\qquad\qquad f_{comm} \\ &\qquad\qquad \{P_2(z) \wedge Q_2'(z')\} \qquad (4) \\ &\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\} \\ &\qquad\qquad f_{comm};\quad f_{obs_1};\quad f_{obs_2} \\ &\qquad\qquad \{GI \wedge P_3(z) \wedge Q_3'(z')\} \qquad (5) \end{aligned}$$

hold for distinct fresh logical variables z of type c and z' of type c', in the following cases:

(1) Call*: For all statements $\{p_1\}\, u_{ret} := e_0.m(\vec{e})\, \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait}$ (or those without receiving a value) in class c with e_0 of type c', where method m of c' has body $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\}\, stm;\, \mathsf{return}\ e_{ret}$, formal parameters $\vec{u}$, and local variables $\vec{v}$ except the formal parameters. The callee class invariant is $q_1 = I_{c'}$. The assertion* comm *is given by $E_0(z) = z'$. Furthermore, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$, f_{obs_1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs_2} is $z'.\vec{y}_2' := \vec{E}_2'(z')$.*

(2) Return*: For all $u_{ret} := e_0.m(\vec{e}) \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_1\}^{wait} \{p_2\}^{?ret} \langle \vec{y}_4 := \vec{e}_4 \rangle^{?ret} \{p_3\}$ (or those without receiving a value) occurring in c with e_0 of type c', such that method m of c' has the return statement $\{q_1\}\ \mathsf{return}\ e_{ret} \{q_2\}^{!ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret} \{q_3\}$, and formal parameter list $\vec{u}$, the above equations must hold with*

comm *given by* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$*, and where* f_{comm} *is* $u_{ret} := E'_{ret}(z')$*,* f_{obs_1} *is* $z'.\vec{y}_3 := \vec{E}'_3(z')$*, and* f_{obs_2} *is* $z.\vec{y}_4 := \vec{E}_4(z)$*.*

(3) $\textsc{Return}_{run}$*: For the statement* $\{q_1\}\,\mathsf{return}\,\{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ *of the* run*-method of the main class,* $p_1 = p_2 = p_3 = \mathsf{true}$*,* $\mathsf{comm} = \mathsf{true}$*,* f_{obs_1} *is* $z'.\vec{y}'_3 := \vec{E}'_3(z')$*, and furthermore* f_{comm} *and* f_{obs_2} *are the empty statement.*

Example 2.6 *This example illustrates how one can prove properties of parameter passing. Let* $\{p\}\,e_0.m(v, \vec{e})$*, with* p *given by* $v > 0$*, be a (partially) annotated statement in a class* c *with* e_0 *of type* c'*, and let method* $m(u, \vec{w})$ *of* c' *have a body of the form* $\{q\}\,stm;\mathsf{return}$ *where* q *is* $u > 0$*. Inductivity of the proof outline requires that if* p *is valid prior to the call (besides validity of the global and class invariants), then* q *is satisfied after the invocation. Omitting irrelevant details, Condition 5 of the cooperation test requires proving* $\models_{\mathcal{G}} \{P(z)\}\ u' := v\ \{Q'(z')\}$*, which expands to* $\models_{\mathcal{G}} \{v > 0\}\ u' := v\ \{u' > 0\}$*.*

Example 2.7 *The following example demonstrates how one can express dependencies between instance states in the global invariant and use this information in the cooperation test.*

Let $\{p\}\,e_0.m(\vec{e})$*, with* p *given by* $x > 0 \wedge e_0 = o$*, be an annotated statement in a class* c *with* e_0 *of type* c'*,* x *an integer instance variable, and* o *an instance variable of type* c'*, and let method* $m(\vec{u})$ *of* c' *have the annotated body* $\{q\}\,stm;\mathsf{return}$ *where* q *is* $y > 0$ *and* y *an integer instance variable. Let furthermore* $z \in LVar^c$ *and let the global invariant be given by* $\forall z.\,(z \neq \mathsf{null} \wedge z.o \neq \mathsf{null} \wedge z.x > 0) \rightarrow z.o.y > 0$*. Inductivity requires that if* p *and the global invariant are valid prior to the call, then* q *is satisfied after the invocation (again, we omit irrelevant details). The cooperation test Condition 5, i.e.,* $\models_{\mathcal{G}} \{GI \wedge P(z) \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\}\ \vec{u}' := \vec{E}(z)\ \{Q'(z')\}$ *expands to*

$$\begin{aligned}\models_{\mathcal{G}} \{&(\forall z.\,(z \neq \mathsf{null} \wedge z.o \neq \mathsf{null} \wedge z.x > 0) \rightarrow z.o.y > 0) \wedge \\ &(z.x > 0 \wedge E_0(z) = z.o) \wedge E_0(z) = z' \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\ \} \\ &\vec{u}' := \vec{E}(z) \\ &\{z'.y > 0\}\ .\end{aligned}$$

Instantiating the quantification by z*, the antecedent implies* $z.o.y > 0 \wedge z' = z.o$*, i.e.,* $z'.y > 0$*. Invariance of the global invariant is straightforward.*

Example 2.8 *This example illustrates how the cooperation test handles observations of communication. Let* $\{\neg b\}\,\mathsf{this}.m(\vec{e})\{b\}^{wait}$ *be an annotated statement in a class* c *with boolean auxiliary instance variable* b *and let* $m(\vec{u})$ *of* c *have a body of the form* $\{\neg b\}^{?call}\langle b := \mathsf{true}\rangle^{?call}\{b\}\,stm;\mathsf{return}$*. Condition 4 of the cooperation test assures inductivity for the precondition of the observation. We*

have to show

$$\models_{\mathcal{G}} \{\neg z.b \wedge \mathsf{comm}\}\vec{u}' := \vec{E}(z)\{\neg z'.b\}$$

(again, we omit irrelevant details), i.e., since it is a self-call,

$$\models_{\mathcal{G}} \{\neg z.b \wedge z = z'\}\vec{u}' := \vec{E}(z)\{\neg z'.b\} \ ,$$

which is trivially satisfied. Condition 5 of the cooperation test for the postconditions requires

$$\models_{\mathcal{G}} \{\mathsf{comm}\}\vec{u}' := \vec{E}(z); z'.b := \mathsf{true}\{z.b \wedge z'.b\}$$

which expands to

$$\models_{\mathcal{G}} \{z = z'\}\vec{u}' := \vec{E}(z); z'.b := \mathsf{true}\{z.b \wedge z'.b\} \ ,$$

whose validity is easy to see.

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. 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 $\mathsf{list\,Object}$ 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 $\mathsf{InitState}(u) \wedge u \notin z' \wedge \forall v.\ v \in z' \vee v = u$, where $\mathsf{InitState}(u)$ is the global assertion $u \neq \mathsf{null} \wedge \bigwedge_{x \in IVar \setminus \{\mathsf{this}\}} u.x = \mathsf{Init}(x)$, expressing that the object denoted by u is in its initial instance state. In this assertion, the syntactical operator Init has the interpretation *Init* (cf. page 6), and *IVar* is the set of instance variables of u. 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' of type $\mathsf{list\,Object}$ 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. Thus a predicate $(\exists u.\, P) \downarrow z'$, evaluated immediately after the instantiation, expresses 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 2.9 (Cooperation test: Instantiation) *A proof outline satisfies the* cooperation test for object creation*, if for all classes c' and statements*

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

$$\begin{aligned} \models_{\mathcal{G}} \ & z{\neq}\mathsf{null} \wedge z{\neq}u \wedge \exists z'.\ \Big(\mathsf{Fresh}(z',u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'\Big) \\ & \qquad \rightarrow P_2(z) \wedge I_c(u) \qquad (6) \\ \models_{\mathcal{G}} \{ & z{\neq}\mathsf{null} \wedge z{\neq}u \wedge \exists z'.\ \Big(\mathsf{Fresh}(z',u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'\Big)\} \\ & \qquad z.\vec{y} := \vec{E}(z) \\ & \qquad \{GI \wedge P_3(z)\} \qquad (7) \end{aligned}$$

with z *of type* c' *and* z' *of type* list Object *fresh.*

Example 2.10 *Assume a statement* $u := \mathsf{new}^c\{u \neq \mathsf{this}\}$ *in a program, where the class invariant of* c *is* $x \geq 0$ *for an integer instance variable* x*. Condition 6 of the cooperation test for object creation assures that the class invariant of the new object holds after its creation. We have to show validity of* $\models_{\mathcal{G}} (\exists z'.\ \mathsf{Fresh}(z',u)) \rightarrow u.x \geq 0$*, i.e.,* $\models_{\mathcal{G}} u.x = 0 \rightarrow u.x \geq 0$*, which is trivial. Remember that integer variables have the initial value* 0*. For the postcondition, Condition 7 requires* $\models_{\mathcal{G}} \{z \neq u\}\ \epsilon\ \{u \neq z\}$ *with* ϵ *the empty statement (no observations are executed), which is true.*

3 Multithreading

In this section we extend the language $Java_{seq}$ to a *concurrent* language $Java_{conc}$ by allowing *dynamic thread creation.* Again, we define syntax and semantics of the language, before formalizing the proof system.

3.1 Syntax and semantics

Expressions, statements, and methods can be constructed as in $Java_{seq}$. The abstract syntax of the remaining constructs is summarized in Table 3. As we

$$\begin{aligned} \mathit{class} &::= c\{\mathit{meth} \ldots \mathit{meth}\ \mathit{meth}_{\mathsf{run}}\ \mathit{meth}_{\mathsf{start}}\} \\ \mathit{class}_{\mathsf{main}} &::= \mathit{class} \\ \mathit{prog} &::= \langle \mathit{class} \ldots \mathit{class}\ \mathit{class}_{\mathsf{main}}\rangle \end{aligned}$$

Table 3
$Java_{conc}$ abstract syntax

focus on concurrency aspects, all classes are `Thread` classes in the sense of

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

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

Fig. 5. $\mathit{Java}_{\mathit{conc}}$ operational semantics

Java: Each class contains the pre-defined method start, which is identical for all classes and therefore syntactically not represented in class definitions. Semantically, when invoked, the start-method spawns a new thread, which starts to execute the object's run-method in parallel. The run-methods cannot be invoked directly. Remember that the syntax 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.

The operational semantics of $\mathit{Java}_{\mathit{conc}}$ extends the semantics of $\mathit{Java}_{\mathit{seq}}$ by dynamic thread creation. The additional rules are shown in Figure 5. The first invocation of a start-method brings a new thread into being ($\textsc{Call}_{\mathit{start}}$). The new thread starts to execute the user-defined run-method of the given object while the initiating thread continues its own execution. Only the first invocation of the start-method has this effect ($\textsc{Call}_{\mathit{start}}^{\mathit{skip}}$).[7] This is captured by the predicate $\mathit{started}(T, \beta)$ which holds iff there is a stack $(\alpha_0, \tau_0, \mathit{stm}_0) \ldots (\alpha_n, \tau_n, \mathit{stm}_n) \in T$ such that $\beta = \alpha_0$. A thread ends its lifespan by returning from a run-method ($\textsc{Return}_{\mathit{run}}$ of Figure 3).[8]

3.2 The proof system

In contrast to the sequential language, the proof system additionally has to accommodate for dynamic thread creation and shared-variable concurrency. Before describing the proof method, we show how to extend the built-in augmentation of the sequential language.

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

[8] 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.

3.2.1 Proof outlines

As mentioned, an important point in achieving completeness of the proof system in the sequential case is the identification of communicating partners. For the concurrent language we additionally have to be able to identify *threads*. We identify a thread by the object in which it has begun its execution. This identification is unique, since an object's thread can be started only once. We use the type Thread thus as abbreviation for the type Object. During a method call, the callee thread receives its own identity as an auxiliary formal parameter thread. Additionally, we extend the auxiliary formal parameter caller by the caller thread identity, i.e., let caller be of type $\mathsf{Object} \times \mathsf{Int} \times \mathsf{Thread}$, 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 values $(\alpha_0, (null, 0, null))$ assigned to the parameters $(\mathsf{thread}, \mathsf{caller})$, where α_0 is the initial object. The boolean instance variable started, finally, remembers whether the object's start-method has already been invoked.

Syntactically, each formal parameter list $\vec{u}$ in the original program gets extended to $(\vec{u}, \mathsf{thread}, \mathsf{caller})$. Correspondingly for the caller, each actual parameter list $\vec{e}$ in statements invoking a method different from start gets extended to $(\vec{e}, \mathsf{thread}, (\mathsf{this}, \mathsf{conf}, \mathsf{thread}))$. The invocation of the parameterless start-method of an object e_0 gets the actual parameter list $(e_0, (\mathsf{this}, \mathsf{conf}, \mathsf{thread}))$. Finally, the callee observation at the beginning of the run-method executes $\mathsf{started} := \mathsf{true}$. The variables conf and counter are updated as in the previous section.

3.2.2 Verification conditions

Local correctness is not influenced by introducing concurrency. Note that local correctness applies now to all concurrently executing threads.

The interference freedom test An assertion q at a control point has to be invariant under an assignment $\vec{y} := \vec{e}$ in the same class only if the local configuration described by the assertion is not active in the computation step executing the assignment. Note that assertions at auxiliary points do not have to be shown invariant. Again, to distinguish local variables of the different local configurations, we rename those of the assertion.

If q and $\vec{y} := \vec{e}$ belong to the *same* thread, i.e., $\mathsf{thread} = \mathsf{thread}'$, then we have the same antecedent as for the sequential language. 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. To describe this setting, we define $\mathsf{self\_start}(q, \vec{y} := \vec{e})$ by $\mathsf{caller} = (\mathsf{this}, \mathsf{conf}', \mathsf{thread}')$ iff q 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.

Definition 3.1 (Interference freedom) *A proof outline is* interference free, *if the conditions of Definition 2.2 hold with* $\mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})$ *replaced by*

$$\begin{aligned}\mathsf{interleavable}(q, \vec{y} := \vec{e}) \stackrel{def}{=} \mathsf{thread} = \mathsf{thread}' \rightarrow \mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e}) \wedge \\ \mathsf{thread} \neq \mathsf{thread}' \rightarrow \neg\mathsf{self\_start}(q, \vec{y} := \vec{e})\,. \qquad (8)\end{aligned}$$

Example 3.2 *Assume an annotated assignment* $\{p\}\, stm$ *in a method, and an assertion* q *at a control point not waiting for return in the same method, such that both* p *and* q *imply* $\mathsf{thread} = \mathsf{this}$*. I.e., the method is executed only by the thread of the object to which it belongs. Clearly,* p *and* q *cannot be simultaneously reached by the same thread. For invariance of* q *under the assignment* stm*, the antecedent of the interference freedom condition implies* $p \wedge q' \wedge \mathsf{interleavable}(q, stm)$*. From* $p \wedge q'$ *we conclude* $\mathsf{thread} = \mathsf{thread}'$*, and thus by the definition of* $\mathsf{interleavable}(q, stm)$ *the assertion* q *should be at a control point waiting for return, which is not the case, and thus the antecedent of the condition evaluates to false.*

The cooperation test The cooperation test for object creation is not influenced by adding concurrency, but we have to extend the cooperation test for communication by defining additional conditions for thread creation. 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, which is already included in the conditions for the sequential language as for the termination of the only thread. Note that this condition applies now to all threads.

Definition 3.3 (Cooperation test: Communication) *A proof outline satisfies the* cooperation test for communication*, if the conditions of Definition 2.5 hold for the statements listed there with* $m \neq \mathsf{start}$*, and additionally in the following cases:*

(1) $\textsc{Call}_{start}$*: For all statements* $\{p_1\}\, e_0.\mathsf{start}(\vec{e})\, \{p_2\}^{!call}\langle \vec{y}_1 := \vec{e}_1\rangle^{!call}\, \{p_3\}$ *in class* c *with* e_0 *of type* c'*,* comm *is given by* $E_0(z) = z' \wedge \neg z'.\mathsf{started}$*, where* $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\, stm$ *is the body of the* run*-method of* c' *having*

$$
\begin{aligned}
\mathit{modif} &::= \mathsf{nsync} \mid \mathsf{sync} \\
\mathit{meth} &::= \mathit{modif}\, m(u,\ldots,u)\{\ \mathit{stm}; \mathsf{return}\ e_{ret}\} \\
\mathit{meth}_{\mathsf{run}} &::= \mathsf{nsync\ run}()\{\ \mathit{stm}; \mathsf{return}\ \} \\
\mathit{meth}_{\mathsf{wait}} &::= \mathsf{nsync\, wait}()\{\ ?\mathsf{signal}; \mathsf{return}_{\mathit{getlock}}\ \} \\
\mathit{meth}_{\mathsf{notify}} &::= \mathsf{nsync\, notify}()\{\ !\mathsf{signal}\ ; \mathsf{return}\ \} \\
\mathit{meth}_{\mathsf{notifyAll}} &::= \mathsf{nsync\, notifyAll}()\{\ !\mathsf{signal\_all}; \mathsf{return}\ \} \\
\mathit{meth}_{\mathit{predef}} &::= \mathit{meth}_{\mathsf{start}}\ \mathit{meth}_{\mathsf{wait}}\ \mathit{meth}_{\mathsf{notify}}\ \mathit{meth}_{\mathsf{notifyAll}} \\
\mathit{class} &::= c\{\mathit{meth}\ldots\mathit{meth}\ \mathit{meth}_{\mathsf{run}}\ \mathit{meth}_{\mathit{predef}}\} \\
\mathit{class}_{\mathsf{main}} &::= \mathit{class} \\
\mathit{prog} &::= \langle \mathit{class}\ldots\mathit{class}\ \mathit{class}_{\mathsf{main}}\rangle
\end{aligned}
$$

Table 4
$Java_{synch}$ abstract syntax

formal parameters $\vec{u}$ and local variables $\vec{v}$ except the formal parameters. As in the CALL *case, $q_1 = I_{c'}$, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$, f_{obs_1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs_2} is $z'.\vec{y}_2' := \vec{E}_2'(z')$.*

(2) $\text{CALL}^{skip}_{start}$*: For the above statements, the equations must additionally hold with the assertion* comm *given by $E_0(z) = z' \wedge z'.\mathsf{started}$, $q_1 = I_{c'}$, $q_2 = q_3 = \mathsf{true}$, f_{obs_1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{comm} and f_{obs_2} are the empty statement.*

4 The language *Java*$_{synch}$

In this section we extend the language $Java_{conc}$ with *monitor synchronization*. Again, we define syntax and semantics of the language $Java_{synch}$, before formalizing the proof system.

4.1 Syntax and semantics

Expressions and statements can be constructed as in the previous languages. The abstract syntax of the remaining constructs is summarized in the Table 4. Formally, methods get decorated by a modifier *modif* distinguishing between *non-synchronized* and *synchronized* methods.[9] In the sequel we also

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

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

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

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

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

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

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

Fig. 6. $\mathit{Java}_{\mathit{synch}}$ Operational semantics

refer to statements in the body of a synchronized method as being synchronized. Furthermore, we consider the additional predefined methods wait, notify, and notifyAll, whose definitions use the auxiliary statements !signal, !signal_all, ?signal, and $\mathsf{return}_{\mathit{getlock}}$.[10]

The operational semantics extends the semantics of $\mathit{Java}_{\mathit{conc}}$ by the rules of Figure 6, where the Call rule is replaced. 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 (Call), as

[10] *Java*'s `Thread` class additionally support methods for suspending, resuming, and stopping a thread, but they are deprecated and thus not considered here.

expressed by the predicate *owns*, defined below. If the thread does not own the lock, it has to wait until the lock gets free. A thread owning the lock of an object can recursively invoke several synchronized methods of that object, which corresponds to the notion of reentrant monitors.

The remaining rules handle the monitor methods wait, notify, and notifyAll. In all three cases the caller must own the lock of the callee object ($\textsc{Call}_{monitor}$). 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 the object's wait set. Formally, the wait set $wait(T, \alpha)$ of an object is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?\mathsf{signal}; stm)$. After having put itself on ice, the thread awaits notification by another thread which invokes the notify-method of the object. The !signal-statement in the notify-method thus reactivates a non-deterministically chosen single thread waiting for notification on the given object (Signal). Analogously to the wait set, the notified set $notified(T, \alpha)$ of α is the set of all stacks in T with top element of the form $(\alpha, \tau, \mathsf{return}_{getlock})$, i.e., threads which have been notified and trying to get hold of the lock again. According to rule $\textsc{Return}_{wait}$, the receiver can continue after notification in executing $\mathsf{return}_{getlock}$ 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 [18]. If no threads are waiting on the object, the !signal of the notifier is without effect ($\textsc{Signal}_{skip}$). The notifyAll-method generalizes notify in that all waiting threads are notified via the !signal_all-broadcast (SignalAll). The effect of this statement is given by setting $signal(T, \alpha)$ as $(T \setminus wait(T, \alpha)) \cup \{\xi \circ (\beta, \tau, stm) \mid \xi \circ (\beta, \tau, ?\mathsf{signal}; stm) \in wait(T, \alpha)\}$.

Using the wait and notified sets, we can now formalize the *owns* predicate: A thread ξ owns the lock of β iff ξ executes some synchronized method of β, but not its wait-method. Formally, $owns(T, \beta)$ is true iff there exists a thread $\xi \in T$ and a $(\beta, \tau, stm) \in \xi$ with stm synchronized and $\xi \notin wait(T, \beta) \cup notified(T, \beta)$. The definition is used analogously for single threads. An invariant of the semantics is that at most one thread can own the lock of an object at a time.

4.2 The proof system

The proof system has additionally to accommodate for synchronization, reentrant monitors, and thread coordination via the wait and notify constructs. First we define how to extend the augmentation of $Java_{conc}$, before we describe the proof method.

4.2.1 Proof outlines

To capture mutual exclusion and the monitor discipline, the built-in auxiliary instance variable lock of type $\mathsf{Thread} \times \mathsf{Int}$ 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 $\mathit{free} = (\mathit{null}, 0)$ indicates that the lock is free. The instance variables wait and notified of type $\mathsf{list}(\mathsf{Thread} \times \mathsf{Int})$ 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. All auxiliary variables are initialized as usual. For values *thread* of type Thread and *wait* of type $\mathsf{list}(\mathsf{Thread} \times \mathsf{Int})$, we will also write $\mathit{thread} \in \mathit{wait}$ instead of $(\mathit{thread}, n) \in \mathit{wait}$ for some n.

Syntactically, besides the augmentation of the previous section, the callee observation at the beginning and at the end of each synchronized method body executes $\mathsf{lock} := \mathsf{inc}(\mathsf{lock})$ and $\mathsf{lock} := \mathsf{dec}(\mathsf{lock})$, respectively. The semantics of incrementing the lock $[\![\mathsf{inc}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ is $(\tau(\mathsf{thread}), n+1)$ for $\sigma_{inst}(\mathsf{lock}) = (\alpha, n)$. Decrementing $\mathsf{dec}(\mathsf{lock})$ is inverse.

Instead of the auxiliary statements of the semantics, notification is represented in the proof system by auxiliary assignments operating on the wait and notified variables. If the order of elements in sequences in not important, we use in the sequel also set notation. Entering the wait-method gets the observation $\mathsf{wait}, \mathsf{lock} := \mathsf{wait} \cup \{\mathsf{lock}\}, \mathsf{free}$; returning from the wait-method observes $\mathsf{lock}, \mathsf{notified} := \mathsf{get}(\mathsf{notified}, \mathsf{thread}), \mathsf{notified} \setminus \{\mathsf{get}(\mathsf{notified}, \mathsf{thread})\}$. For a thread α, the *get* function retrieves the value (α, n) from a wait or notified set. The semantics assures uniqueness of the association. The !signal-statement of the notify-method is represented by the auxiliary assignment $\mathsf{wait}, \mathsf{notified} := \mathsf{notify}(\mathsf{wait}, \mathsf{notified})$, where $\mathit{notify}(\mathit{wait}, \mathit{notified})$ is the pair of the given sets with one element, chosen nondeterministically, moved from the wait into the notified set; if the wait set is empty, it is the identity function. Finally, the !signal_all-statement of the notifyAll-method is represented by the auxiliary assignment $\mathsf{notified}, \mathsf{wait} := \mathsf{notified} \cup \mathsf{wait}, \emptyset$.

4.2.2 Verification conditions

Local correctness agrees with that for Java_{conc}. In case of notification, local correctness covers also invariance for the notifying thread, as the effect of notification is captured by an auxiliary assignment.

The interference freedom test 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.

Definition 4.1 (Interference freedom) *A proof outline is* interference free, *if Definition 3.1 holds in all cases, such that if both p and q occur in a synchronized method, then q is at a control point waiting for return.*

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 value will be imported from the cooperation test as this information depends on the global behavior.

Example 4.2 *This example shows how the fact that at most one thread can own the lock of an object can be used to show mutual exclusion. We use the assertion* $\mathsf{owns}(\mathsf{thread},\mathsf{lock})$ *for* $\mathsf{thread} \neq \mathsf{null} \wedge \mathsf{thread}(\mathsf{lock}) = \mathsf{thread}$*, where thread(lock) is the first component of the lock value. Let* $\mathsf{free\_for}(\mathsf{thread},\mathsf{lock})$ *be* $\mathsf{thread} \neq \mathsf{null} \wedge (\mathsf{owns}(\mathsf{thread},\mathsf{lock}) \vee \mathsf{lock} = \mathsf{free})$.

Let q, given by $\mathsf{owns}(\mathsf{thread},\mathsf{lock})$*, be an assertion at a control point and let* $\{p\}^{?call}\langle stm\rangle^{?call}$ *with* $p \stackrel{def}{=} \mathsf{free\_for}(\mathsf{thread},\mathsf{lock})$ *be the callee observation at the beginning of a synchronized method in the same class. Note that the observation stm changes the lock value. The interference freedom condition* $\models_{\mathcal{L}} \{p \wedge q' \wedge \mathsf{interleavable}(q, stm)\} stm \{q'\}$ *assures invariance of q under the observation stm. The assertions p and q' imply* $\mathsf{thread} = \mathsf{thread}'$*. The points at p and q can be simultaneously reached by the same thread only if q describes a point waiting for return. This fact is mirrored by the definition of the* interleavable *predicate: If q is not at a control point waiting for return, then the antecedent of the condition evaluates to false. Otherwise, after the execution of the built-in augmentation* $\mathsf{lock} := \mathsf{inc}(\mathsf{lock})$ *in stm we have* $\mathsf{owns}(\mathsf{thread},\mathsf{lock})$*, i.e.,* $\mathsf{owns}(\mathsf{thread}',\mathsf{lock})$*, which was to be shown.*

The cooperation test We extend the cooperation test for $Java_{conc}$ with synchronization and the invocation of the monitor methods. In the previous languages, the assertion comm expressed, that the given statements indeed represent communicating partners. In the current language with monitor synchronization, communication is not always enabled. Thus the assertion comm has additionally to capture enabledness of the communication: 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. Returning from the wait-method assumes that the thread has been notified and that the callee's lock is free. Note that the global invariant is not affected by the object-internal monitor signaling mechanism, which is represented by auxiliary assignments.

Definition 4.3 (Cooperation test: Communication) *A proof outline satisfies the* cooperation test for communication*, if the conditions of Definition 3.3 hold for the statements listed there with the exception of the* Call*-case, and additionally in the following cases:*

(1) Call*: For all statements* $\{p_1\}\, u_{ret} := e_0.m(\vec{e})\, \{p_2\}^{!call} \langle \vec{y}_1 := \vec{e}_1 \rangle^{!call} \{p_3\}^{wait}$ *(or such without receiving a value) in class* c *with* e_0 *of type* c'*, where method* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *of* c' *is synchronized with body* $\{q_2\}^{?call} \langle \vec{y}_2 := \vec{e}_2 \rangle^{?call} \{q_3\}\, stm$*, formal parameters* $\vec{u}$*, and local variables* $\vec{v}$ *except the formal parameters. The callee class invariant is* $q_1 = I_{c'}$*. 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} *is* $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$*,* f_{obs_1} *is given by* $z.\vec{y}_1 := \vec{E}_1(z)$*, and* f_{obs_2} *is* $z'.\vec{y}_2' := \vec{E}_2'(z')$*. If* m *is not synchronized,* $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *in* comm *is dropped.*

(2) $\text{Call}_{monitor}$*: For* $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$*,* comm *is given by* $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$*.*

(3) Return_{wait}*: For* $\{q_1\}\, \mathsf{return}_{getlock} \{q_2\}^{!ret} \langle \vec{y}_3 := \vec{e}_3 \rangle^{!ret} \{q_3\}$ *in a* wait*-method,* comm *is* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.\mathsf{lock} = \mathsf{free} \wedge \mathsf{thread}' \in z'.\mathsf{notified}$*.*

Example 4.4 *Assume the invocation of a synchronized method* m *of a class* c*, where* m *of* c *has the body* $\langle stm \rangle^{?call} \{\mathsf{thread}(\mathsf{lock}) = \mathsf{thread}\}\, stm'; \mathsf{return}$*. Note that the built-in augmentation in stm sets the lock owner by the assignment* $\mathsf{lock} := \mathsf{inc}(\mathsf{lock})$*. Omitting irrelevant details again, the cooperation test requires* $\models_{\mathcal{G}} \{\mathsf{true}\} z'.\mathsf{lock} := \mathsf{inc}(z'.\mathsf{lock}) \{\mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}'\}$*, which holds by the definition of inc.*

5 Soundness and relative completeness

This section contains soundness and relative completeness of the proof method of Section 4.2. The proofs can be found in [17]. 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 $prog'$ with annotation φ' satisfies the verification conditions of the proof system:

Definition 5.1 *Given a program prog with annotation φ, then $prog \models \varphi$ iff for all reachable configurations $\langle T, \sigma \rangle$ of prog, for all $\alpha \in dom(\sigma)$ with class invariant I_c, for all $(\alpha, \tau, stm) \in T$, for all logical environments ω referring only to values existing in σ, and for all local states τ':*

(1) $\omega, \sigma \models_{\mathcal{G}} GI$,
(2) $\omega, \sigma(\alpha), \tau' \models_{\mathcal{L}} I_c$, *and*
(3) $\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} pre(stm)$.

For proof outlines, we write $prog' \vdash \varphi'$ iff $prog'$ with annotation φ' 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 un-augmented 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 augmentations. Additionally, for local configurations $(\alpha, \tau, \mathsf{return}_{getlock}) \in T'$, 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 $(\alpha, \tau, stm; \mathsf{return})$ 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 5.2 *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 [17].

Let $prog$ be a program with annotation φ, and $prog'$ 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 5.3 (Soundness) *Given a proof outline $prog'$ with annotation $\varphi_{prog'}$, then*

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

Theorem 5.3 is formulated for augmented programs. We get immediately with the help of Lemma 5.2:

Corollary 5.4 *If $prog' \vdash \varphi_{prog'}$ and $\models \varphi_{prog'} \rightarrow \varphi_{prog}$, then $prog \models \varphi_{prog}$.*

The soundness proof is basically an induction on the length of computations, simultaneously on all three parts from Definition 5.1. After handling the initial case, the inductive step assumes $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle \grave{T}, \grave{\sigma} \rangle \longrightarrow \langle \acute{T}, \acute{\sigma} \rangle$ such that $\langle \grave{T}, \grave{\sigma} \rangle$ satisfies the conditions of Definition 5.1, and ω a logical environment referring only to values existing in $\acute{\sigma}$. In the proof cases we distinguish between possible

kinds of the computation step $\langle \grave{T}, \grave{\sigma}\rangle \longrightarrow \langle \acute{T}, \acute{\sigma}\rangle$. We illustrate the soundness proof on the case for synchronized method invocation.

Case: CALL
Let $(\alpha, \grave{\tau}_1, u_{ret} := e_0.m(\vec{e}); \langle \vec{y}_1 := \vec{e}_1\rangle^{!call} stm_1) \in \grave{T}$ 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 $\langle \vec{y}_2 := \vec{e}_2\rangle^{?call} stm_2$ be the invoked method's body, $\vec{u}$ its formal parameters, and $\vec{v}$ its local variables except the 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 $\acute{\sigma} = \hat{\sigma}[\beta.\vec{y}_2 \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta),\hat{\tau}_2}]$.

Since the invoked method is synchronized, if ξ is the stack of the executing thread in $\grave{T}$, then according to the transition rule $\neg owns(\grave{T}\backslash\{\xi\}, \beta)$. Using the correctness of the representation of the lock ownership and uniqueness of the identification mechanism by the built-in auxiliary variables 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 where $\grave{\omega}$ is given by $\omega[z \mapsto \alpha][z' \mapsto \beta][\vec{v}_1 \mapsto \grave{\tau}_1(\vec{v}_1)]$. Similarly, $\beta = [\![e_0]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}$ implies $\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} E_0(z) = z'$. Remember that $E_0(z)$ is a shortcut for $e_0[z/\mathsf{this}]$.

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

$$\begin{aligned}\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \{&GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \wedge z{\neq}\mathsf{null} \wedge z'{\neq}\mathsf{null}\}\\ &\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v}); \quad z.\vec{y}_1 := \vec{E}_1(z); \quad z'.\vec{y}_2' := \vec{E}_2'(z')\\ &\{GI \wedge P_3(z) \wedge Q_3'(z')\}\end{aligned}$$

where comm is $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$. Note that the above assignments represent exactly the state changes caused by communication and the observations of caller and callee. Thus we have

$$\acute{\omega}, \acute{\sigma} \models_{\mathcal{G}} GI \wedge P_3(z) \wedge Q_3'(z')$$

with $\acute{\omega}$ given by $\grave{\omega}[\vec{u}' \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}][\vec{v}' \mapsto \mathsf{Init}(\vec{v})][\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\grave{\sigma}(\alpha),\grave{\tau}_1}][\vec{y}_2' \mapsto [\![\vec{e}_2]\!]_{\mathcal{E}}^{\hat{\sigma}(\beta),\hat{\tau}_2}]$. 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. I.e.,

$\omega, \acute{\sigma} \models_{\mathcal{G}} GI$, and thus part (1). Using correctness of the lifting substitution we get similarly $\omega, \acute{\sigma}(\alpha), \acute{\tau}_1 \models_{\mathcal{L}} p_3$ and $\omega, \acute{\sigma}(\beta), \acute{\tau}_2 \models_{\mathcal{L}} q_3$.

Thus part (3) 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 in the following that γ is either the caller α or the callee β. We need to apply the interference freedom test to show invariance of the corresponding assertions. To do so, we use the cooperation test to show that the preconditions of the observations are satisfied in the state in which they get executed. The cooperation test assures

$$\begin{aligned}\grave{\omega}, \grave{\sigma} \models_{\mathcal{G}} \{&GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \wedge z{\neq}\mathsf{null} \wedge z'{\neq}\mathsf{null}\} \\ &\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v}) \\ &\{P_2(z) \wedge Q_2'(z')\} \ .\end{aligned}$$

As above, the precondition is satisfied, and we get that $\grave{\sigma}(\alpha), \grave{\tau}_1 \models_{\mathcal{L}} p_2$ and $\grave{\sigma}(\beta), \hat{\tau}_2 \models_{\mathcal{L}} q_2$.

We distinguish three cases: γ can be the caller object, the callee object, or both in case of a self-call. Assume first $\gamma = \alpha$ and $\alpha \neq \beta$, and let τ be $\grave{\tau}_1[\vec{v}' \mapsto \tau_3(\vec{v})]$, where $\vec{v} = dom(\tau_3)$. The interference freedom test assures

$$\begin{aligned}\omega, \grave{\sigma}(\alpha), \tau \models_{\mathcal{L}} \{&p_2 \wedge pre'(stm_3) \wedge \mathsf{interleavable}(pre(stm_3), \vec{y}_1 := \vec{e}_1)\} \\ &\vec{y}_1 := \vec{e}_1 \\ &\{pre'(stm_3)\} \ .\end{aligned}$$

With the definition of $\mathsf{interleavable}$ this yields $\omega, \hat{\sigma}(\alpha), \tau[\vec{y}_1 \mapsto [\![\vec{e}_1]\!]_{\mathcal{E}}^{\hat{\sigma}(\alpha), \tau}] \models_{\mathcal{L}} pre(stm_3)$. Due to the renaming mechanism, no local variables in $\vec{v}'$ occur in $\vec{y}_1$. Renaming back the variables leads to $\omega, \hat{\sigma}(\alpha), \tau_3 \models_{\mathcal{L}} pre(stm_3)$. Now, since $\beta \neq \alpha$, the callee observation neither changes 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 applied to the states $\hat{\sigma}(\beta)$ and τ with $\tau = \hat{\tau}_2[\vec{v}' \mapsto \tau_3(\vec{v})]$ results $\omega, \acute{\sigma}(\beta), \acute{\tau} \models_{\mathcal{L}} pre'(stm_3)$ 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 (3) is satisfied.

Part (2) 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 does not 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. □

5.2 Relative 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:

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

Given a program satisfying an annotation $\mathit{prog} \models \varphi_{\mathit{prog}}$, the consequent can be uniformly shown, i.e., independently of the given assertional part φ_{prog}, by instantiating $\varphi_{\mathit{prog}'}$ to the strongest annotation still provable, thereby discharging the last clause $\models \varphi_{\mathit{prog}'} \rightarrow \varphi_{\mathit{prog}}$. Since the strongest annotation still satisfied by the program corresponds to reachability, the key to (relative) 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 5.6 below), and finally
(3) to show that this augmentation indeed satisfies the verification conditions.

We begin with the augmentation, using the transformation of the previous sections 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 $l \equiv \mathit{stm}$ if l represents the control point in front of *stm*.

The standard way for relative 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* behavior in the *local* history, and for *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 communicating partners. 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 $\bigcup_c \{\mathsf{new}^c\} \cup \bigcup_m \{!m, ?m\} \cup \{!\mathsf{return}, ?\mathsf{return}\}$ 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 in this paper. See [19] for such a compositional semantics.

Definition 5.5 (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* $e_0.\mathsf{start}(\vec{e}')\langle\vec{y} := \vec{e}\rangle^{!call}$ *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	$!m(\vec{e})$
reception of return	e_0	e_0	this	$?\,\mathsf{return}\, u_{ret}, \mathsf{thread}$
reception of call $m(\vec{u})$	caller_obj	caller_obj	this	$?m(\vec{u})$
return e_{ret}	caller_obj	this	caller_obj	$!\,\mathsf{return}\ e_{ret}, \mathsf{thread}$

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

Note that the communication history records also the identities of the communicating threads in values. Next we introduce the annotation for the augmented program.

Definition 5.6 (Reachability annotation) *We define*

(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 such that* $(\alpha, \tau, stm; stm') \in T$.

(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 [20]. The augmented program together with the above annotation build a proof outline *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 their verification conditions 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 5.7 (Local merging lemma) *Assume two reachable global configurations* $\langle T_1, \sigma_1 \rangle$ *and* $\langle T_2, \sigma_2 \rangle$ *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 5.8 (Global merging lemma) *Assume two reachable global configurations* $\langle T_1, \sigma_1 \rangle$ *and* $\langle T_2, \sigma_2 \rangle$ *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 both merging lemmas together imply 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 5.9 (Completeness) *For a program prog, the proof outline* $prog'$ *satisfies the verification conditions of the proof system from Section 4.2.*

The completeness proof handles all cases for the different verification condition groups. Here we illustrate the proof by the case of interference freedom:

Case: Interference freedom
Assume an arbitrary assignment $\vec{y} := \vec{e}$ with precondition p in class c, and an arbitrary assertion q at a control point in the same class. We show that the proof outline $prog'$ satisfies the conditions for interference freedom, i.e.,

$$\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} \{p \wedge q' \wedge \mathsf{interleavable}(q, \vec{y} := \vec{e})\} \;\; \vec{y} := \vec{e} \;\; \{q'\}$$

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'. We do so by proving that $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p \wedge q' \wedge \mathsf{interleavable}(q, \vec{y} := \vec{e})$ implies $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}] \models_{\mathcal{L}} q'$.

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. Assume that q is 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 5.7 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 stm_p in 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, the caller observation $\vec{y} := \vec{e}$ does not change the caller instance state. Thus, due to the renaming mechanism, $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst}, \tau}] \models_{\mathcal{L}} q'$.

Otherwise, if $(\alpha, \tau_p, stm_p; stm'_p)$ does not 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 in the case of 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 $(\alpha, \tau', stm_q; stm'_q) \in \acute{T}_p$.

Using the annotation definition 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 $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], \tau \models_{\mathcal{L}} q'$. 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. This implies the required property $\omega, \sigma_{inst}[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}], \tau[\vec{y} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}] \models_{\mathcal{L}} q'$.

Validity of the verification condition 2 for the class invariant is similar, where we additionally use the fact that the class invariant refers to instance variables only. □

6 Proving deadlock freedom

The previous sections described a proof system which can be used to prove safety properties of $Java_{synch}$ programs. In this section we show how to apply the proof system to show *deadlock freedom*.

A system of processes is in a deadlocked configuration, if no one of them is enabled to compute, but not yet all started processes are terminated. A typical deadlock situation can occur, if two threads t_1 and t_2 both try to gather the locks of two objects z_1 and z_2, but in reverse order: t_1 first applies for access to synchronized methods of z_1, and then for those of z_2, while t_2 first collects the lock of z_2, and tries to become the lock owner of z_1. Now, it can happen, that t_1 gets the lock of z_1, t_2 gets the lock of z_2, and both are waiting for the other lock, which will never become free. Another typical source of deadlock situations are threads which suspended themselves by calling wait and which will never get notified.

What kind of $Java_{synch}$-statements can be disabled and under which conditions? The important cases, to which we restrict, are

- the invocation of synchronized methods, if the lock of the callee object is neither free nor owned by the executing thread,
- if a thread tries to invoke a monitor method of an object whose lock it does not own, or
- if a thread tries to return from a wait-method, but either the lock is not free or the thread is not yet notified.

To be exact, the semantics specifies method calls to be disabled also, if the callee object is the empty reference. However, we won't deal with this case; it can be excluded in the preconditions by stating that the callee object is not null.

Assume a proof outline with global invariant GI. For a logical variable z of

type Object, let $I(z) = I[z/\mathsf{this}]$ be the class invariant of z expressed on the global level. Let the assertion $\mathsf{terminated}(z)$ express that the thread of z is already terminated. Formally, we define $\mathsf{terminated}(z) = q[z/\mathsf{thread}][z/\mathsf{this}]$, where q is the postcondition of the run-method of z. For assertions p in z' let furthermore $\mathsf{blocked}(z, z', p)$ express that the thread of z is disabled in the object z' at control point p. Formally, we define $\mathsf{blocked}(z, z', p)$ by

- $\exists \vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge e_0.\mathsf{lock} \neq \mathsf{free} \wedge \mathsf{thread}(e_0.\mathsf{lock}) \neq \mathsf{thread}$ if p is the precondition of a call invoking a synchronized method of e_0,
- $\exists \vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge \mathsf{thread}(e_0.\mathsf{lock}) \neq \mathsf{thread}$ if p is the precondition of a call invoking a monitor method of e_0,
- $\exists \vec{v}.\, p[z/\mathsf{thread}][z'/\mathsf{this}] \wedge (z'.\mathsf{lock} \neq \mathsf{free} \vee z \notin z'.\mathsf{notified})$ if p is the precondition of the return-statement in the wait-method, and
- false otherwise,

where $\vec{v}$ is the vector of local variables in the given assertion without thread, and z and z' fresh. Let finally $\mathsf{blocked}(z, z')$ express that the thread of object z is blocked in the object z'. Formally, it is defined by the assertion $\bigvee_{p \in Ass(z')} \mathsf{blocked}(z, z', p)$, where $Ass(z')$ is the set of all assertions at control points in z'. Now we can formalize the verification condition for deadlock freedom:

Definition 6.1 *A proof outline satisfies the test for* deadlock freedom*, if*

$$
\begin{aligned}
&\models_{\mathcal{G}} (GI \wedge \\
&\quad (\forall z.\, z \neq \mathsf{null} \rightarrow (I(z) \wedge \\
&\qquad (z.\mathsf{started} \rightarrow (\mathsf{terminated}(z) \vee (\exists z'.\, z' \neq \mathsf{null} \wedge \mathsf{blocked}(z, z')))))) \wedge \\
&\quad (\exists z.\, z \neq \mathsf{null} \wedge z.\mathsf{started} \wedge (\exists z'.\, z' \neq \mathsf{null} \wedge \mathsf{blocked}(z, z')))) \\
&\quad \rightarrow \mathsf{false}\ .
\end{aligned}
\tag{9}
$$

The above condition states, that the assumptions that all started processes are terminated or disabled, and that at least one thread is not yet terminated, i.e., that the program is in a deadlocked configuration, lead to a contradiction. Soundness of the above condition, i.e., that the condition indeed assures absence of deadlock, is easy to show. Completeness results directly from the completeness of the proof method.

Example 6.2 *The proof outline below defines two classes,* Producer *and* Consumer*, where* Producer *is the main class. The initial thread of the initial* Producer*-instance creates a* Consumer*-instance and calls its synchronized* produce *method. This method starts the consumer thread and enters a non-terminating loop, producing some results, notifying the consumer, and suspending itself by calling* wait*. After the producer suspended itself, the consumer thread calls the synchronized* consume *method, which consumes the result of the pro-*

ducer, notifies, and calls wait*, again in a non-terminating loop.*

The assertion owns *is as in Example 4.2,* $proj(v, i)$ *denotes the* i*th component of the tuple* v*, and* not_owns(thread, lock) *is* thread $\neq$ null $\wedge$ proj(lock, 1) $\neq$ thread*. Again, the built-in augmentation is not listed in the code. We additionally list instance and local variable declarations* type name;*, where* ⟨type name;⟩ *declares auxiliary variables. We sometimes skip return statements without giving back a value, and write explicitly* $\forall(z : t).p$ *for quantification over* t*-typed values.*

For readability, we only list a partial annotation and augmentation, which already implies deadlock freedom. Invariance of the properties listed below has been shown in PVS using an extended augmentation and annotation [13]. Also deadlock freedom has been proven in PVS.

$$
\begin{array}{l}
GI \stackrel{def}{=} \\
\quad (\forall(p : Producer).(p \neq null \wedge \neg p.outside \wedge p.consumer \neq null) \rightarrow \\
\qquad p.consumer.lock = (null, 0)) \wedge \\
\quad (\forall(c : Consumer).(c \neq null \wedge c.started) \rightarrow \\
\qquad (c.producer \neq null \wedge c.producer.started)) \wedge \\
\quad (\forall(c1 : Consumer).(c1 \neq null \rightarrow (\forall(c2 : Consumer).c2 \neq null \rightarrow c1 = c2))
\end{array}
$$

$$
\begin{array}{l}
I_{Producer} \stackrel{def}{=} true \\
I_{Consumer} \stackrel{def}{=} length(wait) \leq 1 \wedge \\
\quad (lock = (null, 0) \vee (owns(this, lock) \wedge started) \vee owns(producer, lock))
\end{array}
$$

```
class Producer {
    ⟨ Consumer consumer; ⟩
    ⟨ Bool outside; ⟩

    nsync Void wait(){ {false} }

    nsync Void run(){
        Consumer c;
        c := new^Consumer ; ⟨consumer := c⟩^new
        {c = consumer ∧ ¬outside ∧ consumer ≠ null ∧ consumer ≠ this ∧
           thread = this}
        c.produce() ⟨outside := (if c = this then outside else true fi)⟩^!call
        {false}
    }
}

class Consumer {
    Int buffer;
    ⟨ Producer producer; ⟩

    nsync Void wait(){
        {started ∧ not_owns(thread, lock) ∧ (thread = this ∨ thread = producer) ∧
```

```
            (thread ∈ wait ∨ thread ∈ notified)}
    }

    sync Void produce(){
        Int i;

        ⟨producer := proj(caller, 1)⟩?call
        i:=0;
        start();
        while (true) do
            //produce i here
            buffer := i;
            {owns(thread, lock)}
            notify();
            {owns(thread, lock)}
            wait()
        od
    }

    nsync Void run(){
        {not_owns(thread, lock) ∧ thread = this}
        consume()
        {false}
    }

    sync Void consume(){
        Int i;

        while (true) do
            i := buffer;
            //consume i here
            {owns(thread, lock)}
            notify();
            {owns(thread, lock)}
            wait()
        od
    }
}
```

Both run*-methods have* false *as postcondition, stating that the corresponding threads don't terminate. The preconditions of all monitor method invocations express that the executing thread owns the lock, and thus execution cannot be enabled at these control points. The* wait*-method of* Producer*-instances is not invoked; we define* false *as the precondition of its return-statement, implying that disabledness is excluded also at this control point.*

The condition for deadlock freedom assumes that there is a thread which is started but not yet terminated, and whose execution is disabled. This thread is either the thread of a Producer*-instance, or that of a* Consumer*-instance.*

We discuss only the case that the disabled thread belongs to a Producer*-instance z different from null; the other case is similar. Note that the control of the thread of z cannot stay in the* run*-method of a* Consumer*-instance, since the corresponding local assertion implies* thread $=$ this*, which would contradict the type assumptions. Thus the thread can have its control point prior to the method call in the* run*-method of a* Producer*-instance, or in the* wait*-method of a* Consumer*-instance. In the first case, the corresponding local assertion and the global invariant imply that the lock of the callee is free, i.e., that the execution is enabled, which is a contradiction. In the second case, if the thread of z executes in the* wait*-method of a* Consumer*-instance z', the local assertion in* wait *together with the type assumptions implies* $z'.\mathsf{started} \wedge \mathsf{not\_owns}(z, z'.\mathsf{lock}) \wedge z = z'.\mathsf{producer}$*, and that z is either in the wait- or in the notified-set of z'.*

By the assumptions of the deadlock freedom condition, also the started thread of z' is disabled or terminated; its control point cannot be in a Producer*-instance, since that would contradict to the type assumptions. Thus the control of z' stays in the* run*- or in the* wait*-method of a* Consumer*-instance; the annotation implies that the instance is z' itself.*

If the control stays in the run*-method, then the corresponding local assertion and the class invariant imply that the lock is free, since neither the producer, nor the consumer owns it, which leads to a contradiction, since in this case the execution of the thread of z' would be enabled. Finally, if the control of the thread of z' stays in the* wait*-method of z', then the annotation assures that the thread does not own the lock of z'; again, using the class invariant we get that the lock is free.*

Now, both threads of z and z' have their control points in the wait*-method of z', and the lock of z' is free. Furthermore, both threads are disabled, and are in the wait- or in the notified set. If one of them is in the notified set, then its execution is enabled, which is a contradiction. If both threads are in the wait set, then from $z \neq z'$ we imply that the wait-set of z' has at least two elements, which contradicts the class invariant of z'.*

Thus the assumptions lead to a contradiction, which was to be shown.

7 Conclusion

Extending earlier work, this paper presents a sound and relatively complete assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline. We also provide conditions for deadlock freedom.

In [21] we develop a proof system for a concurrent *Java* subset without reen-

trant lock synchronization and without the wait and notify constructs. The proof system was extended in [22] to deal with reentrant monitor synchronization. The wait and notify constructs are incorporated in [23]. The extension of the proof system to prove deadlock freedom can be found in [24]. Currently we are working on the incorporation of *Java*'s exception handling mechanism [25]. We formalize the semantics of our programming language in a compositional manner in [19]. The underlying theory, the proof rules, their soundness and completeness, and tool support for the automatic generation of verification conditions are presented in detail in [13].

Related work As far as proof systems and verification support for object-oriented programs is concerned, research mostly concentrated on *sequential* languages. Early examples of Hoare-style proof systems for sequential object-oriented languages are [26] and [27,28]. America and de Boer [29] formulate for the first time a cooperation test for an object-oriented language with synchronous message passing.

With *Java*'s rise to prominence, research more concretely turned to (sublanguages of) *Java*, as opposed to object-oriented language features in the abstract. In this direction, *JML* [30,31] has emerged as common ground for asserting *Java* programs. Another trend is to offer mechanized proof support. For instance, Poetzsch-Heffter and Müller [7,32–34] develop a Hoare-style programming logic presented in sequent formulation for a sequential kernel of *Java*, featuring interfaces, subtyping, and inheritance. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer-assisted soundness proof. The work in the LOOP-project (cf. e.g. [35,36]) also concentrates on a sequential subpart of *Java*, translating the proof-theory into *PVS* and *Isabelle/HOL*.

The work [37,38] 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. In [39] a large subset of *JavaCard*, including exception handling, is formalized in *Isabelle/HOL*, and its soundness and completeness is shown within the theorem prover. The work in [15] presents a Hoare-style proof system for a sequential object-oriented calculus [40]. 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. The close connection of types and specifications in the presentation is exploited in [41] for the generation of verification conditions.

Work on proof systems for parallel object-oriented languages or in particular the multithreading aspects of *Java* is more scarce. [42] presents a sound and complete proof system in weakest precondition formulation for a parallel object-based language, i.e., without inheritance and subtyping, and also without reentrant method calls. Later work [43–45] includes more features, especially catering for Hoare logic for inheritance and subtyping.

A survey about *monitors* in general, including proof-rules for various monitor semantics, can be found in [46]. Besides deductive verification, there are several other research areas for *Java* program analysis. For example, the paper [47] presents a model checking algorithm and its implementation in *Isabelle/HOL* to check type correctness of Java bytecode. See [48,49] for an overview.

Future work As to future work, we plan to extend $Java_{synch}$ by further constructs, like inheritance and subtyping. Dealing with subtyping on the logical level requires a notion of behavioral subtyping [50].

Acknowledgments

We thank Cees Pierik for fruitful discussions and suggestions, and furthermore Tim D'Avis for careful reading and commenting on an earlier version of this document.

References

[1] J. Gosling, B. Joy, G. L. Steele, The Java Language Specification, Addison-Wesley, 1996.

[2] J. Alves-Foss (Ed.), Formal Syntax and Semantics of Java, Vol. 1523 of Lecture Notes in Computer Science State-of-the-Art-Survey, Springer-Verlag, 1999.

[3] R. Stärk, J. Schmid, E. Börger, Java and the Java Virtual Machine: Definition, Verification, Validation, Springer-Verlag, 2001.

[4] P. Cenciarelli, A. Knapp, B. Reus, M. Wirsing, An event-based structural operational semantics of multi-threaded Java, in: Alves-Foss [2], pp. 157–200.

[5] M. Huisman, Java program verification in higher-order logic with PVS and Isabelle, Ph.D. thesis, University of Nijmegen (2001).

[6] D. von Oheimb, T. Nipkow, Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited, in: L.-H. Eriksson, P. A. Lindsay (Eds.), Proceedings of FME'02, Vol. 2391 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 89–105.

[7] A. Poetzsch-Heffter, P. Müller, A programming logic for sequential Java, in: S. Swierstra (Ed.), Proceedings of ESOP'99, Vol. 1576 of Lecture Notes in Computer Science, Springer, 1999, pp. 162–176.

[8] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Informatica 6 (4) (1976) 319–340.

[9] R. W. Floyd, Assigning meanings to programs, in: J. T. Schwartz (Ed.), Proc. Symp. in Applied Mathematics, Vol. 19, 1967, pp. 19–32.

[10] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (1969) 576–580.

[11] G. Levin, D. Gries, A proof technique for communicating sequential processes, Acta Informatica 15 (3) (1981) 281–302.

[12] K. R. Apt, N. Francez, W.-P. de Roever, A proof system for communicating sequential processes, ACM Transactions on Programming Languages and Systems 2 (1980) 359–385.

[13] E. Ábrahám, An assertional proof system for multithreaded Java — theory and tool support, Ph.D. thesis, University of Leiden, to appear. A preliminary version can be found at `http://www.informatik.uni-freiburg.de/~eab/phd.ps` (2004).

[14] S. Owre, J. M. Rushby, N. Shankar, PVS: A prototype verification system, in: D. Kapur (Ed.), Automated Deduction (CADE-11), Vol. 607 of Lecture Notes in Computer Science, Springer-Verlag, 1992, pp. 748–752.

[15] M. Abadi, K. R. M. Leino, A logic of object-oriented programs, in: M. Bidoit, M. Dauchet (Eds.), Proceedings of TAPSOFT '97, Vol. 1214 of Lecture Notes in Computer Science, Springer-Verlag, Lille, France, 1997, pp. 682–696, an extended version of this paper appeared as SRC Research Report 161 (September 1998).

[16] B. Jacobs, J. Kiniry, M. Warnier, Java program verification challenges, in: Bonsangue et al. [51], pp. 202–220.

[17] E. Ábrahám, F. S. de Boer, W.-P. de Roever, M. Steffen, A Hoare logic for monitors in Java, Techical report TR-ST-03-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel (Apr. 2003).
URL `http://www.informatik.uni-kiel.de/inf/deRoever/techreports/03/tr-st-03-1.pdf`

[18] G. R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley, 2000.

[19] E. Ábrahám, F. S. de Boer, W.-P. de Roever, M. Steffen, A compositional operational semantics for Java$_{MT}$, in: N. Derschowitz (Ed.), International Symposium on Verification (Theory and Practice), Vol. 2772 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 290–303, a preliminary version appeared as Technical Report TR-ST-02-2, May 2002.

[20] J. V. Tucker, J. I. Zucker, Program Correctness over Abstract Data Types, with Error-State Semantics, Vol. 6 of CWI Monograph Series, North-Holland, 1988.

[21] E. Ábrahám-Mumm, F. S. de Boer, Proof-outlines for threads in Java, in: C. Palamidessi (Ed.), Proceedings of CONCUR'00, Vol. 1877 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 229–242.

[22] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, M. Steffen, Verification for Java's reentrant multithreading concept, in: M. Nielsen, U. H. Engberg (Eds.), Proceedings of FoSSaCS'02, Vol. 2303 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 4–20, a longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

[23] E. Ábrahám, F. S. de Boer, W.-P. de Roever, M. Steffen, Inductive proof-outlines for monitors in Java, in: Najm et al. [52], pp. 155–169, a longer version appeared as technical report TR-ST-03-1, April 2003 (`http://www.informatik.uni-kiel.de/inf/deRoever/techreports/03/tr-st-03-1.pdf`).

[24] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, M. Steffen, A tool-supported proof system for monitors in Java, in: Bonsangue et al. [51], pp. 1–32.

[25] E. Ábrahám, F. S. de Boer, W.-P. de Roever, M. Steffen, Inductive proof outlines for multithreaded Java with exceptions, Technical Report 0313, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel (Dec. 2003).
URL `http://www.informatik.uni-kiel.de/reports/2003/0313.html`

[26] C. C. de Figueiredo, A proof system for a sequential object-oriented language, Technical Report UMCS-95-1-1, University of Manchester (1995).

[27] G. T. Leavens, W. E. Wheil, Reasoning about object-oriented programs that use subtypes, in: Proceedings of OOPSLA'90, ACM, 1990, pp. 212–223, extended abstract.

[28] G. T. Leavens, W. E. Wheil, Specification and verification of object-oriented programs using supertype abstraction, Acta Informatica 32 (8) (1995) 705–778, an expanded version appeared as Iowa State University Report, 92-28d.

[29] P. America, F. S. de Boer, A sound and complete proof system for SPOOL, Technical Report 505, Philips Research Laboratories (1990).

[30] G. T. Leavens, A. L. Baker, C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java, Tech. Rep. TR #98-06f, Iowa State University, revised version from July 1999 (2000).

[31] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, D. R. Cok, How the design of JML accommodates both runtime assertion checking and formal verification, in: Bonsangue et al. [51], pp. 262–284.

[32] A. Poetzsch-Heffter, Specification and Verification of Object-Oriented Programs, Technische Universität München, 1997, Habilitationsschrift.

[33] A. Poetzsch-Heffter, A logic for the verification of object-oriented programs, in: R. Berghammer, F. Simon (Eds.), Proceedings of Programming Languages and Fundamentals of Programming, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 1997, pp. 31–42, Bericht Nr. 9717.

[34] A. Poetzsch-Heffter, P. Müller, Logical foundations for typed object-oriented languages, in: D. Gries, W.-P. de Roever (Eds.), Proceedings of PROCOMET '98, International Federation for Information Processing (IFIP), Chapman & Hall, 1998, pp. 404–423.

[35] The LOOP project: Formal methods for object-oriented systems, `http://www.cs.kun.nl/~bart/LOOP/` (2001).

[36] B. Jacobs, J. van den Berg, M. Huisman, M. van Barkum, U. Hensel, H. Tews, Reasoning about classes in Java (preliminary report), in: Proceedings of OOPSLA'98, ACM, 1998, pp. 329–340, in *SIGPLAN Notices* 30(10).

[37] B. Reus, M. Wirsing, A Hoare-logic for object-oriented programs, Technical report, LMU München (2000).

[38] B. Reus, R. Hennicker, M. Wirsing, A Hoare calculus for verifying Java realizations of OCL-constrained design models, in: H. Hussmann (Ed.), Fundamental Approaches to Software Engineering, Vol. 2029 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 300–316.

[39] D. von Oheimb, Hoare logic for Java in Isabelle/HOL, Concurrency and Computation: Practice and Experience 13 (13) (2001) 1173–1214.

[40] M. Abadi, L. Cardelli, A Theory of Objects, Monographs in Computer Science, Springer, 1996.

[41] F. Tang, M. Hofmann, Generation of verification conditions for Abadi and Leino's logic of objects (extended abstract), in: Proceedings of FOOL'02, 2002, a longer version is available as LFCS technical report.

[42] F. S. de Boer, A WP-calculus for OO, in: W. Thomas (Ed.), Proceedings of FoSSaCS'99, Vol. 1578 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 135–156.

[43] C. Pierik, F. S. de Boer, A syntax-directed Hoare logic for object-oriented programming concepts, in: Najm et al. [52], pp. 64–78, an extended version appeared as University of Utrecht Technical Report UU-CS-2003-010.

[44] F. S. de Boer, C. Pierik, Towards an environment for the verification of annotated object-oriented programs, Technical report UU-CS-2003-002, Institute of Information and Computing Sciences, University of Utrecht (Jan. 2003).

[45] F. S. de Boer, C. Pierik, Computer-aided specification and verification of annotated object-oriented programs, in: B. Jacobs, A. Rensink (Eds.), Proceedings of FMOODS'02, Vol. 209, Kluwer, 2002, pp. 163–177.

[46] P. A. Buhr, M. Fortier, M. H. Coffin, Monitor classification, ACM Computing Surveys 27 (1) (1995) 63–107.

[47] D. Basin, S. Friedrich, M. Gawkowski, Verified bytecode model checkers, in: V. A. Carreño, C. A. Muñoz, S. Tahar (Eds.), Proceedings of TPHOLs'02, Vol. 2410 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 47–66.

[48] X. Leroy, Java bytecode verification: An overview, in: G. Berry, H. Comon, A. Finkel (Eds.), Proceedings of CAV'01, Vol. 2102 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 265–285.

[49] P. H. Hartel, L. Moreau, Formalizing the safety of Java, the Java virtual machine, and Java Card, ACM Computing Surveys 33 (4) (2001) 517–558.

[50] P. America, A behavioural approach to subtyping in object-oriented programming languages, 443, Phillips Research Laboratories (January/April 1989).

[51] M. Bonsangue, F. S. de Boer, W.-P. de Roever, S. Graf (Eds.), Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO'02), Leiden, Vol. 2852 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

[52] E. Najm, U. Nestmann, P. Stevens (Eds.), Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'03), Paris, Vol. 2884 of Lecture Notes in Computer Science, Springer-Verlag, 2003.