A Tool-supported Proof System for Multithreaded Java[⋆]

July 23, 2003

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

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

Abstract. Besides the features of a class-based object-oriented language, *Java* integrates concurrency via its thread classes. The concurrency model includes shared-variable concurrency via instance variables, coordination via reentrant synchronization monitors, synchronous message passing, and dynamic thread creation.
To reason about safety properties of multithreaded *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. From the annotated program, a translator tool generates a number of verification conditions which are handed over to the interactive theorem prover *PVS*.

1 Introduction

Besides the features of a class-based object-oriented language, *Java* integrates *concurrency* via its thread classes. The semantical foundations of *Java* [GJS96] have been thoroughly studied ever since the language gained widespread popularity (e.g. [AF99,SSB01,CKRW99]). The research concerning *Java*'s proof theory mainly concentrated on *sequential* sub-languages (e.g. [Hui01,vON02,PHM99]). This work presents a tool-supported assertional proof system for $Java_{synch}$, a subset of *Java*, featuring dynamic object creation, method invocation, object references with aliasing, and, specifically, concurrency and *Java*'s monitor discipline.

The behavior of a $Java_{synch}$ program results from the concurrent execution of methods. To support a clean interface between internal and external object behavior, $Java_{synch}$ does not allow qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution

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

within a single object, only, but not across object boundaries. 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. As in the Object Constraint Language (OCL) [WK99], properties of object-structures are described in terms of a navigation or dereferencing operator.

The proof system is formulated in terms of *proof outlines* [OG76], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [Flo67,Hoa69]. 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* [OG76,LG81], formulated also in the local language. It has especially to accommodate for reentrant code and the specific synchronization mechanism. Possibly affecting more than one instance, communication and object creation is treated in the *cooperation test*, using the global language. The communication can take place within a single object or between different objects. As these cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules in [AFdR80] and in [LG81] for CSP.

Our proof method is *modular* in the sense that it allows for separate interference freedom and cooperation tests (Fig. 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.

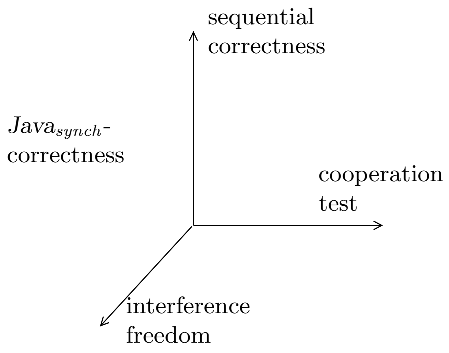

Fig. 1. Modularity of the proof system

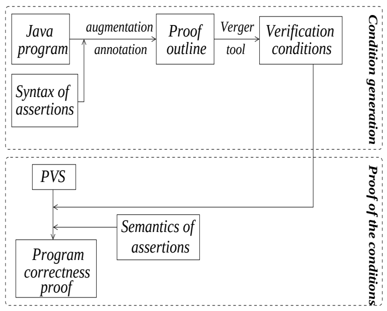

Fig. 2. The verification process

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 [ORS92] to verify the conditions, for which we only need to encode the semantics of the assertion language (cf. Figure 2). The verification conditions are generated by a syntax-directed Hoare logic based on a logical modeling of assignments by means of substitutions, instead of more semantic approaches using the global store model [AL97,JKW03,vON02,PHM99], which requires an explicit encoding of the semantics of assignments.

To transparently describe the proof system, we present it incrementally in three stages: We start with a *sequential,* class-based sublanguage of *Java* and its proof system in Section 2, featuring dynamic object creation and method invocation. This level shows how to handle activities of a single *thread* of execution. On the second stage we include *concurrency* in Section 3, where the proof system is extended to handle dynamic thread creation and aspects of interleaving and shared variable concurrency. Finally, we integrate *Java*'s *monitor synchronization* mechanism in Section 4. Section 5 shows how we can prove deadlock freedom, and Section 6 discusses related and future work.

The incremental development shows how the proof system can be extended stepwise to deal with additional features of the programming language. Further extensions with for example the concepts of inheritance and subtyping build topics for future work.

In this paper, the verification conditions are formulated as standard Hoare-triples $\{\varphi\}stm\{\psi\}$. Their meaning is, that if *stm* is executed in a state satisfying φ, and the execution terminates, then the resulting state satisfies ψ. For the formal semantics of Hoare-triples, given by means of a weakest precondition calculus, for soundness and completeness of the proof method, and for the description of the tool support see [ÁdBdRS03].

2 The sequential sublanguage

In this section we start with a sequential part of our 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 *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 dynamically created, and communicate via *method invocation,* i.e., synchronous message passing.

The languages we consider are strongly typed languages. 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.

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

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, *Java$_{seq}$* does not allow 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.[3] 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 $e_0, \dots, e_n$ in a method invocation $e_0.m(e_1, \dots, e_n)$ 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, without loosing the mentioned modularity, but it would increase the complexity of the proof system.

2.2 Semantics

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 $\alpha, \beta, \dots$ 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 $\mathsf{this} \notin Var$, such that the self-reference is not in the domain of $Init$.[4]

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 $\tau^{m,c}_{init}$ of method m

[3] 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 to make this restriction is, that *Java*'s main-method is static, but our proof system does not support static methods and variables.

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

of class c assigns to each local variable u of m the value $\mathit{Init}(u)$. A *local configuration* $(\alpha, \tau, \mathit{stm})$ 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 $(\alpha_0, \tau_0, \mathit{stm}_0) \ldots (\alpha_n, \tau_n, \mathit{stm}_n)$, representing the call chain of the thread. We write $\xi \circ (\alpha, \tau, \mathit{stm})$ for pushing a new local configuration onto the stack.

An object is characterized by its *instance state* $\sigma_{\mathit{inst}} \in \Sigma_{\mathit{inst}}$, a partial function of type $\mathit{IVar} \,\dot\cup\, \{\mathsf{this}\} \rightharpoonup \mathit{Val}_{\mathit{null}}$, which assigns values to the self-reference this and to the instance variables. The initial instance state $\sigma_{\mathit{inst}}^{c,\mathit{init}}$ of instances of class c assigns a value from Val^c to this, and to each of its remaining instance variables x the value $\mathit{Init}(x)$. A *global state* $\sigma \in \Sigma$ of type $(\dot\bigcup_c \mathit{Val}^c) \rightharpoonup \Sigma_{\mathit{inst}}$ stores for each currently *existing* object, i.e., an object belonging to the domain $\mathit{dom}(\sigma)$ of σ, its instance state. The set of existing objects of type c in a state σ is given by $\mathit{Val}^c(\sigma)$, and $\mathit{Val}^c_{\mathit{null}}(\sigma) = \mathit{Val}^c(\sigma) \,\dot\cup\, \{\mathit{null}^c\}$. For the remaining types, $\mathit{Val}^t(\sigma)$ and $\mathit{Val}^t_{\mathit{null}}(\sigma)$ are defined correspondingly, $\mathit{Val}(\sigma) = \dot\bigcup_t \mathit{Val}^t(\sigma)$, and $\mathit{Val}_{\mathit{null}}(\sigma) = \dot\bigcup_t \mathit{Val}^t_{\mathit{null}}(\sigma)$. 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 $(\alpha, \tau, \mathit{stm}) \in T$ if there exists a local configuration $(\alpha, \tau, \mathit{stm})$ 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; $\sigma_{\mathit{inst}}[x \mapsto v]$ 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; $\sigma_{\mathit{inst}}[\vec{y} \mapsto \vec{v}]$ and $\sigma[\alpha.\vec{y} \mapsto \vec{v}]$ are analogous. Finally for global states, $\sigma[\alpha \mapsto \sigma_{\mathit{inst}}]$ equals σ except on α; note that in case $\alpha \notin \mathit{Val}(\sigma)$, the operation extends the set of existing objects by α, which has its instance state initialized to σ_{inst}.

Operational semantics Expressions are evaluated with respect to an *instance local* state $(\sigma_{\mathit{inst}}, \tau)$, where the instance state gives meaning to the instance variables and the self-reference, whereas the local state determines the values of the local variables. The main cases of the evaluation function are $[\![x]\!]_{\mathcal{E}}^{\sigma_{\mathit{inst}},\tau} = \sigma_{\mathit{inst}}(x)$ and $[\![u]\!]_{\mathcal{E}}^{\sigma_{\mathit{inst}},\tau} = \tau(u)$. The operational semantics of $\mathit{Java}_{\mathit{seq}}$ is given inductively by the rules of Table 2 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 $\mathit{dom}(\sigma_0) = \{\alpha\}$, $\sigma_0(\alpha) = \sigma_{\mathit{inst}}^{c,\mathit{init}}[\mathsf{this} \mapsto \alpha]$, and $T_0 = \{(\alpha, \tau_{\mathit{init}}^{\mathsf{run},c}, \mathit{body}_{\mathsf{run},c})\}$, where c is the main class, and $\alpha \in \mathit{Val}^c$.

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

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

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

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

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

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

Table 2. Java_{seq} operational semantics

A configuration $\langle T, \sigma\rangle$ of a program is *reachable* if there exists a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$ such that $\langle T_0, \sigma_0\rangle$ is the initial configuration of the program and $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$. A local configuration $(\alpha, \tau, \mathit{stm}) \in T$ is *enabled* in $\langle T, \sigma\rangle$, if 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 $\textsc{Ass}_{inst}$ and $\textsc{Ass}_{loc}$). 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 table 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 $\textsc{Return}_{run}$).

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.[5] Global assertions may furthermore contain qualified references. Note that when the global expressions E and E' refer to the same object, that is, E and E' are *aliases*, then $E.x$ and $E'.x$ denote the same variable.

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 on 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 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 3. 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.

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.

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

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

Table 3. Semantics of assertions

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.*

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 atomically executed multiple assignments $\vec{y} := \vec{e}$ to auxiliary variables, which we call *observations.* Furthermore, the observations have, in general, to be "attached" to statements they observe in an atomic manner. 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[6] is 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 $\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\, \mathit{stm}; \mathsf{return}\ e_{ret}; \langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}$ 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 call, 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

[6] 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 [ÁdBdRS03].

call them *auxiliary points*. In the following we call assignment statements with their observations, unobserved assignments, alone-standing observations, or observations of communication or object creation general as multiple assignments, since they are executed simultaneously.

In order to express the transition semantics in the logic, we identify each local configuration by the object in which it executes together with the value of its built-in auxiliary local variable conf storing a unique object-internal identifier. 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 get the value $(null, 0)$.

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

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.[7] We require that the precondition of each method's body is the class invariant. 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.

Verification conditions The proof system formalizes 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 [ÁdBdRS03], 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 their own execution. For example, the precondition of an assignment must imply its postcondition after its execution. The following condition should hold for all multiple assignments being an assignment statement with its observation, an unobserved assignment, or an alone-standing observation:

Definition 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*.

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

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 (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,

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

Note that if we would allow qualified references in program expressions, 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 $O(c \cdot k \cdot l)$, instead of $O((c \cdot k) \cdot (c \cdot l))$ with qualified references.

Example 1. 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 all assertions imply $\mathsf{conf} = x$. I.e., the identity of the executing local configuration is stored in the instance variable x. The annotation expresses that the method m' of c is not called recursively. That means, in our sequential language, no pairs of control points in m' of c can be simultaneously reached.

The assertions p_2 and p_4 do not have to 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 are 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 is analogous.

Example 2. Assume a partially[8] 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

$$\begin{aligned}\models_{\mathcal{L}} \{\ & \mathsf{caller} = (\mathsf{this}, x) \wedge (\mathsf{conf}' = x \wedge p'_2) \wedge \\ & \mathsf{waits\_for\_ret}((\mathsf{conf} = x \wedge p_2), stm) \quad \}\ stm\ \{\mathsf{conf}' = x \wedge p'_2\}\ .\end{aligned}$$

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 $\mathsf{caller} = (\mathsf{this}, x)$ directly after communication and satisfaction of p_3 and q_2 after the observation is assured by the cooperation test.

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

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

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

$$\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\}$$
$$f_{comm}$$
$$\{P_2(z) \wedge Q_2'(z')\} \tag{4}$$
$$\models_{\mathcal{G}} \{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm} \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null}\}$$
$$f_{comm}; \quad f_{obs1}; \quad f_{obs2}$$
$$\{GI \wedge P_3(z) \wedge Q_3'(z')\} \tag{5}$$

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 such without receiving a value) in class c with e_0 of type c', where method m of c' has body $\{q_2\}^{?call}\langle \vec{y}_2 := \vec{e}_2\rangle^{?call}\{q_3\}\, stm;$* 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_{obs1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs2} 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 such without receiving a value) occurring in c with e_0 of type c', such that method m of c' has the return statement $\{q_1\}$* 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_{obs1} is $z'.\vec{y}_3' := \vec{E}_3'(z')$, and f_{obs2} is $z.\vec{y}_4 := \vec{E}_4(z)$.*
3. RETURN$_{run}$*: For $\{q_1\}$* return*$;\{q_2\}^{!ret}\langle \vec{y}_3 := \vec{e}_3\rangle^{!ret}\{q_3\}$ in the* run*-method of the main class, $p_1 = p_2 = p_3 = \mathsf{true}$, $\mathsf{comm} = \mathsf{true}$, f_{obs1} is $z'.\vec{y}_3' := \vec{E}_3'(z')$, and furthermore f_{comm} and f_{obs2} are the empty statement.*

Example 3. 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 the body $\{q\}\, stm;$ return where q is $u > 0$. Inductivity of the proof outline requires that if p is valid prior to the call (besides 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 4. 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

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

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 5. 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 the body $\{\neg b\}^{?call} \langle b := \mathsf{true}\rangle^{?call} \{b\}\, stm; \mathsf{return}$. Condition 4 of the cooperation test assures inductivity for the precondition of the observation. We have to show $\models_{\mathcal{G}} \{\neg z.b \wedge \mathsf{comm}\}\vec{u}' := \vec{E}(z)\{\neg z'.b\}$, 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 Init is a syntactical operator with interpretation *Init* (cf. page 5), *IVar* is the set of instance variables of u, and $\mathsf{InitState}(u)$ 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. 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 4 (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':*

$$\begin{aligned} &\models_{\mathcal{G}} \quad z{\neq}\mathsf{null} \wedge z{\neq}u \wedge \exists z'.\ \big(\mathsf{Fresh}(z',u) \wedge (GI \wedge \exists u.\ P_1(z)) \downarrow z'\big) \\ &\qquad\qquad\qquad\qquad \rightarrow P_2(z) \wedge I_c(u) \end{aligned} \tag{6}$$

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

with z of type c' and z' of type $\mathsf{list\,Object}$ *fresh.*

Example 6. 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. 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 and statements can be constructed as in $Java_{seq}$. The abstract syntax of the remaining constructs is summarized in Table 4. As we focus on concurrency aspects, all classes are `Thread` classes in the sense of *Java*: Each class contains the pre-defined methods start and run. The run-methods cannot be invoked directly. The parameterless start-method without return value is not implemented syntactically; its semantics is described below. Note, 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.

$$\begin{aligned}
\mathit{meth} &::= m(u,\ldots,u)\{\ \mathit{stm}; \mathsf{return}\ e_{\mathit{ret}}\} \\
\mathit{meth}_{\mathsf{run}} &::= \mathsf{run}()\{\ \mathit{stm}; \mathsf{return}\ \} \\
\mathit{class} &::= c\{\mathit{meth}\ldots\mathit{meth}\ \mathit{meth}_{\mathsf{run}}\ \mathit{meth}_{\mathsf{start}}\} \\
\mathit{class}_{\mathsf{main}} &::= \mathit{class} \\
\mathit{prog} &::= \langle \mathit{class}\ldots\mathit{class}\ \mathit{class}_{\mathsf{main}}\rangle
\end{aligned}$$

Table 4. *Java*$_{conc}$ abstract syntax

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

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

Table 5. *Java*$_{conc}$ operational semantics

The operational semantics of *Java*$_{conc}$ extends the semantics of *Java*$_{seq}$ by dynamic thread creation. The additional rules are shown in Table 5. The first invocation of a start-method brings a new thread into being ($\textsc{Call}_{start}$). 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 ($\textsc{Call}_{start}^{skip}$).[9] 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 ($\textsc{Return}_{run}$ of Table 2).[10]

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.

Proof outlines As mentioned, an important point of the proof system to achieve completeness is the identification of communicating partners. For the concurrent language we additionally have to be able to identify *threads*. We

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

[10] 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.

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 $(\alpha_0, (\mathit{null}, 0, \mathit{null}))$ assigned to the parameters $(\mathsf{thread}, \mathsf{caller})$, where α_0 is the initial object. The boolean instance variable $\mathsf{started}$, 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 $\mathsf{counter}$ are updated as in the previous section.

Verification conditions Local correctness is not influenced by the new issue of 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 5 (Interference freedom). *A proof outline is* interference free, *if the conditions of Definition 2 hold with* $\mathsf{waits\_for\_ret}(q, \vec{y} := \vec{e})$ *replaced by*

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

Example 7. Assume an assignment $\{p\}\, stm$ in an annotated method m of c, and an assertion q at a control point in the same method, which is not waiting for return, 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 6 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test for communication*, if the conditions of Definition 3 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 formal parameters $\vec{u}$ and local variables $\vec{v}$ except the formal parameters. As in the* $\textsc{Call}$*case, $q_1 = I_{c'}$, f_{comm} is $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$, f_{obs1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{obs2} is $z'.\vec{y}'_2 := \vec{E}'_2(z')$.*
2. $\textsc{Call}^{skip}_{start}$*: 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_{obs1} is $z.\vec{y}_1 := \vec{E}_1(z)$, and f_{comm} and f_{obs2} 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 6. Formally, methods get decorated by a modifier *modif* distinguishing between

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

Table 6. *Java*$_{synch}$ abstract syntax

non-synchronized and *synchronized* methods.[11] We use $sync(c, m)$ to state that method m in class c is synchronized. In the sequel we also refer to statements in the body of a synchronized method as being synchronized. Furthermore, we consider the additional predefined methods wait, notify, and notifyAll, whose definitions use the auxiliary statements !signal, !signal_all, ?signal, and $\mathsf{return}_{getlock}$.[12]

The operational semantics extends the semantics of *Java*$_{conc}$ by the rules of Table 7, 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 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 (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 RETURN$_{wait}$, the receiver can continue after notification in executing $\mathsf{return}_{getlock}$ only if the lock

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

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

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

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

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

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

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

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

Table 7. *Java$_{synch}$* Operational semantics

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 [And00]. If no threads are waiting on the object, the !signal of the notifier is without effect ($\textsc{Signal}_{\mathit{skip}}$). The notifyAll-method generalizes notify in that all waiting threads are notified via the !signal_all-broadcast ($\textsc{SignalAll}$). The effect of this statement is given by setting $\mathit{signal}(T,\alpha)$ as $(T \setminus \mathit{wait}(T,\alpha)) \cup \{\xi \circ (\beta, \tau, \mathit{stm}) \mid \xi \circ (\beta, \tau, ?\mathsf{signal}; \mathit{stm}) \in \mathit{wait}(T,\alpha)\}$.

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, $\mathit{owns}(T,\beta)$ is true iff there exists a thread $\xi \in T$ and a $(\beta, \tau, \mathit{stm}) \in \xi$ with *stm* synchronized and $\xi \notin \mathit{wait}(T,\beta) \cup \mathit{notified}(T,\beta)$. 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. First we define how to extend the augmentation of $Java_{conc}$, before we describe the proof method.

Proof outlines To capture mutual exclusion and the monitor discipline, the 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 $\mathsf{notified}$ 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 $\mathsf{notified}$ variables. Entering the wait-method gets the observation $\mathsf{wait}, \mathsf{lock} := \mathsf{wait} \cup \{\mathsf{lock}\}, \mathsf{free}$; returning from the wait-method observes $\mathsf{lock}, \mathsf{notified} := \mathsf{get}(\mathsf{notified}, \mathsf{thread}), \mathsf{notified} \backslash \{\mathsf{get}(\mathsf{notified}, \mathsf{thread})\}$. Given 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 $\mathsf{!signal}$-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 $\mathsf{!signal\_all}$-statement of the $\mathsf{notifyAll}$-method is represented by the auxiliary assignment $\mathsf{notified}, \mathsf{wait} := \mathsf{notified} \cup \mathsf{wait}, \emptyset$.

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.[13]

Definition 7 (Interference freedom). *A proof outline is* interference free, *if Definition 5 holds in all cases, such that either not both p and q occur in a synchronized method, or 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 $\mathsf{notified}$ 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 8. 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 $\mathit{thread}(\mathit{lock})$ is the first component of the lock value. Let furthermore $\mathsf{free\_for}(\mathsf{thread}, \mathsf{lock})$ 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 $p \overset{def}{=} \mathsf{free\_for}(\mathsf{thread}, \mathsf{lock})$ be the callee observation at the beginning of a synchronized method in the same class. Note that the observation 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 $\mathsf{interleavable}$ 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.

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

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 8 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test for communication*, if the conditions of Definition 6 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_{obs1} *is given by* $z.\vec{y}_1 := \vec{E}_1(z)$*, and* f_{obs2} *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. $\mathrm{CALL}_{monitor}$*: 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 9. 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 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 doesn't 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'. 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 9. *A proof outline satisfies the test for* deadlock freedom, *if*

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

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 10. In the following example, the assertion owns is as in Example 8, $proj(v, i)$ denotes the ith component of the tuple v, and $\mathsf{not\_owns}(\mathsf{thread}, \mathsf{lock})$ is $\mathsf{thread} \neq \mathsf{null} \wedge \mathsf{proj}(\mathsf{lock}, 1) \neq \mathsf{thread}$. 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.

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 producer, notifies, and calls `wait`, again in a non-terminating loop.

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. Also deadlock freedom has been proven in *PVS*.

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

$$I_{Producer} \overset{def}{=} true$$

$$\begin{aligned}&I_{Consumer} \overset{def}{=} (lock = (null, 0) \vee (owns(this, lock) \wedge started) \vee owns(producer, lock)) \wedge \\ &\quad length(wait) \leq 1\end{aligned}$$

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

    nsync wait(){ {false} }

    nsync run(){
        Consumer c;
        c = new^Consumer; ⟨$consumer = c$⟩^$new$
        {$c = consumer \wedge \neg outside \wedge consumer \neq null \wedge consumer \neq this \wedge thread = this$}
        c.produce(); ⟨$outside =$ (if $c = this$ then $outside$ else $true$ fi)⟩^$!call$
        {$false$}
    }
}

class Consumer {
    int buffer;
    ⟨ Producer producer; ⟩

    nsync wait(){
        {$started \wedge not\_owns(thread, lock) \wedge (thread = this \vee thread = producer) \wedge$
            $(thread \in wait \vee thread \in notified)$}
    }

    sync produce(){
        int i;

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

    nsync run(){
        {$not\_owns(thread, lock) \wedge thread = this$}
        consume();
        {$false$}
    }

    sync consume(){
        int i;

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

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 $\mathsf{thread} = \mathsf{this}$, 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'.

According to 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 doesn't 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.

6 Conclusion

This paper presents a tool-supported assertional proof method for a multi-threaded sublanguage of *Java* including its monitor discipline. This builds on earlier work ([ÁMdB00] and especially [ÁMdBdRS02]). The underlying theory, the proof rules, their soundness and completeness, and the tool support are presented in greater detail in [ÁdBdRS03].

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 a sequential object-oriented languages are [dF95] and [LW90,LW94]. With *Java*'s rise to prominence, research more intensively turned to (sublanguages of) *Java*, as opposed of capturing object-oriented language features in the abstract. In this direction, JML [LBR98,LCC$^+$03] has emerged as some kind of common ground for asserting *Java* programs. Another trend is to offer mechanized proof support for the languages.

For instance, Poetzsch-Heffter and Müller [PHM99,PH97b,PH97a,PHM98] 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. [Loo01,JvdBH$^+$98]) also concentrates on a sequential subpart of *Java*, translating the proof-theory into *PVS* and *Isabelle/HOL*.

The work [RW00,RHW01] use a modification of the *object constraint language* OCL as assertional language to annotate UML class diagrams and to generate proof conditions for *Java*-programs. The work [BFG02] presents a model checking algorithm and its implementation in *Isabelle/HOL* to check type correctness of Java bytecode. In [vO01] 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 [AL97] presents a Hoare-style proof-system for a sequential object-oriented calculus [AC96]. 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 [TH02] 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. [dB99] 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 [PdB03,dBP03,dBP02] 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 [BFC95].

Future work As ti future work, we plan to extend $Java_{synch}$ by further constructs, like inheritance, subtyping, and exception handling. Dealing with subtyping on the logical level requires a notion of behavioral subtyping [Ame89].

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

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.

[ÁdBdRS03] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin 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, April 2003.

[AF99] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS State-of-the-Art-Survey.* Springer-Verlag, 1999.

[AFdR80] K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Proceedings of TAPSOFT '97*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696, Lille, France, April 1997. Springer-Verlag. An extended version of this paper appeared as SRC Research Report 161 (September 1998).

[ÁMdB00] Erika Ábrahám-Mumm and Frank S. de Boer. Proof-outlines for threads in Java. In Catuscia Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, August 2000.

[ÁMdBdRS02] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In Mogens Nielsen and Uffe H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, April 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.

[Ame89] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. 443, Phillips Research Laboratories, January/April 1989.

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[BdBdRG03] Marcello Bosangue, Frank S. de Boer, Willem-Paul de Roever, and Susanne Graf, editors. *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002), Leiden*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[BFG02] David Basin, Stefan Friedrich, and Marek Gawkowski. Verified bytecode model checkers. In Victor A. Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs'02)*, volume 2410 of *Lecture Notes in Computer Science*, pages 47–66. Springer-Verlag, August 2002.

[CKRW99] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [AF99], pages 157–200.

[dB99] Frank S. de Boer. A WP-calculus for OO. In Wolfgang Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–156. Springer-Verlag, 1999.

[dBP02] Frank S. de Boer and Cees Pierik. Computer-aided specification and verification of annotated object-oriented programs. In Bart Jacobs and Arend Rensink, editors, *Proceedings of the Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, volume 209, pages 163–177. Kluwer, 2002.

[dBP03] Frank S. de Boer and Cees 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, January 2003.

[dF95] C. C. de Figueiredo. A proof system for a sequential object-oriented language. Technical Report UMCS-95-1-1, University of Manchester, 1995.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[HJ89] C. A. R. Hoare and Cliff B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [HJ89].

[Hui01] Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.

[JKW03] Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In Bosangue et al. [BdBdRG03].

[JvdBH$^+$98] Bart Jacobs, Jan van den Berg, Marijke Huisman, M. van Barkum, Ulrich Hensel, and Hendrik Tews. Reasoning about classes in Java (preliminary report). In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA) '98*, pages 329–340. ACM, 1998. in *SIGPLAN Notices*.

[LBR98] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java modelling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.

[LCC$^+$03] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of jml accommodates both runtime assertion checking and formal verification. In Bosangue et al. [BdBdRG03].

[LG81] G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.

[Loo01] The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.

[LW90] Gary T. Leavens and William E. Wheil. Reasoning about object-oriented programs that use subtypes. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '90 (Ottawa, Canada)*, pages 212–223. ACM, 1990. Extended Abstract.

[LW94] Gary T. Leavens and William E. Wheil. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 1994. An expanded version appeared as Iowa State Unversity Report, 92-28d.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.

[PdB03] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. Technical report UU-CS-2003-010, Institute of Information and Computing Sciences, University of Utrecht, 2003.

[PH97a] Arnd Poetzsch-Heffter. A logic for the verification of object-oriented programs. In Rudolf Berghammer and Friedeman Simon, editors, *Proceedings of Programming Languages and Fundamentals of Programming*, pages 31–42. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, November 1997. Bericht Nr. 9717.

[PH97b] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. Technische Universität München, January 1997. Habilitationsschrift.

[PHM98] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In David Gries and Willem-Paul de Roever, editors, *Proceedings of PROCOMET '98*. International Federation for Information Processing (IFIP), Chapman & Hall, 1998.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.

[RHW01] Bernhard Reus, Rolf Hennicker, and Martin Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 300–316. Springer-Verlag, 2001.

[RW00] Bernhard Reus and Martin Wirsing. A Hoare-logic for object-oriented programs. Technical report, LMU München, 2000.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.

[TH02] Francis Tang and Martin Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects (extended abstract). In *Proceedings of the 9th International Workshop on Foundations of Object-Oriented Languages (FOOL'02)*, 2002. A longer version is available as LFCS technical report.

[vO01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[vON02] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P.-A. Lindsay, editors, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.

[WK99] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Object Technology Series. Addison-Wesley, 1999.