Verification for Java's Reentrant Multithreading Concept

January 21, 2002

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

[1] Christian-Albrechts-Universität zu Kiel, Germany
eab,wpr,ms@informatik.uni-kiel.de
[2] Utrecht University, The Netherlands *frankb@cs.uu.nl*

Abstract. Besides the features of a class-based object-oriented language, *Java* integrates concurrency via its thread-classes, allowing for a *multithreaded* flow of control. The concurrency model offers *coordination* via lock-synchronization, and *communication* by synchronous message passing, including re-entrant method calls, and by instance variables shared among threads.
To reason about multithreaded programs, we introduce in this paper an *assertional proof method* for *Java*$_{MT}$ (*"Multi-Threaded Java"*), a small concurrent sublanguage of *Java*, covering the mentioned concurrency issues as well as the object-based core of *Java*, i.e., object creation, side effects, and aliasing, but leaving aside inheritance and subtyping.

1 Introduction

The semantical foundations of *Java* [15] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [4, 29, 12]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sublanguages (see e.g. [20, 33, 27]). This paper presents a proof system for *multithreaded Java* programs. Concentrating on the issues of concurrency, we introduce an abstract programming language *Java*$_{MT}$, a subset of *Java* featuring object creation, method invocation, object references with aliasing, and specifically concurrency. Threads are the units of concurrency and are created as instances of specific Thread-classes.

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

The assertional proof system for verifying safety properties of *Java*$_{MT}$ is formulated in terms of *proof outlines* [24], i.e., of annotated programs where Hoare-style assertions [14, 18] are associated with every control point. Soundness and completeness of the proof system is shown in [3].

Recall that the global behaviour of a *Java* program results from the concurrent execution of method bodies, that can interact by shared-variable concurrency, synchronous message passing for method calls, and object creation. In order to capture these features, the proof system is split into three parts.

The execution of a single method body in isolation is captured by *local correctness* conditions that show the inductiveness of the annotated method bodies.

Interaction via synchronous message passing and via object creation cannot be established locally but only relative to assumptions about the communicated values. These assumptions are verified in the *cooperation test.* The communication can take place within a single object or between different objects. As these two cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules used in [8] and in [22] for CSP.

Finally, the effect of shared-variable concurrency is handled, as usual, by the *interference freedom test,* which is modeled after the corresponding tests in the proof systems for shared-variable concurrency in [24] and in [22]. In the case of *Java* it additionally has to accommodate for reentrant code and the specific synchronization mechanism. To simplify the proof system we reduce the potential of interference by disallowing public instance variables in $Java_{MT}$.

The assertion language consists of two different levels: The local assertion language specifies the behaviour on the level of method execution, and is used to annotate programs. The global behaviour, including the communication topology of the objects, is expressed in the global language used in the cooperation test. As in the Object Constraint Language (OCL) [34], global assertions describe properties of object-structures in terms of a navigation or dereferencing operator.

This paper is organized as follows: Section 2 defines the syntax of $Java_{MT}$ and sketches its semantics. After introducing the assertion language in Section 3, the main Section 4 presents the proof system. In Section 5, we discuss related and future work.

2 The programming language $Java_{MT}$

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

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

All programs are assumed to be well-typed, i.e, each method invoked on an object must be supported by the object, the types of the formal and actual parameters of the invocation must match, etc. As the static relationships between classes are orthogonal to multithreading aspects, we ignore in Java_{MT} the issues of *inheritance*, and consequently subtyping, overriding, and late binding. For simplicity, we also do not allow method *overloading*, i.e., we require that each method name is assigned a unique list of formal parameter types and a return type. In short, being concerned with the verification of the run-time behavior, we assume a simple *monomorphic* type discipline for Java_{MT}.

2.1 Abstract syntax

Similar to *Java*, the language Java_{MT} is strongly typed. We use t as typical element of types. As built-in primitive types we restrict to integers and booleans. Besides the built-in types Int and Bool, the set of user-definable types is given by a set of class names $\mathcal{C}$, with typical element c. Furthermore, the language allows pairs of type $t_1 \times t_2$ and sequences of type $\mathsf{list}\, t$. Methods without a return value will get the type Void.

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

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

For variables, we notationally distinguish between *instance* and *local* variables, where instance variables are always private in Java_{MT}. They hold the state of an object and exist throughout the object's lifetime. Local variables play the role of formal parameters and variables of method definitions. They only exist during the execution of the method to which they belong and store the local state of a thread of execution.

The set of variables $\mathit{Var}{=}\mathit{IVar} \,\dot\cup\, \mathit{TVar}$ with typical element y is given as the disjoint union of the instance and the local variables. Var^t denotes the set of all variables of type t, and correspondingly for IVar^t and TVar^t. As we assume a monomorphic type discipline, $\mathit{Var}^t \cap \mathit{Var}^{t'}{=}\emptyset$ for distinct types t and t'. We use as typical elements $x, x', x_1, \ldots$ for IVar and $u, u', u_1, \ldots$ for TVar.

Table 1 contains the abstract syntax of Java_{MT}. Since most of the syntax is standard, we mention only a few salient constructs and restrictions. We distinguish between side-effect-free expressions $e \in \mathit{Exp}_c^t$ and those with side effects $\mathit{sexp} \in \mathit{SExp}_c^t$, where c is the class in which the expression occurs and t is its type. We will use similar conventions concerning sub- and superscripts for other constructs. Methods can be declared as *non-synchronized* or *synchronized*. The body of a method m of class c we denote by $\mathit{body}_{m,c}$. As mentioned earlier, all classes in Java_{MT} are thread classes; the corresponding start- and run-methods are denoted by $meth_{\mathsf{start}}$ and $meth_{\mathsf{run}}$. The main class $class_{\mathsf{main}}$ contains as entry point of the program execution the specific method $meth_{\mathsf{main}}$ with body $body_{\mathsf{main}}$.

To simplify the proof system, we make the following additional restrictions: Method invocation and object creation statements may not refer to instance variables. We also disallow assignments to formal parameters. For methods, we assume that their bodies are terminated by a single return statement.

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

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

2.2 Semantics

States and configurations For each type t, Val^t denotes its value domain, where Val is given by $\bigcup_t Val^t$. Specifically, for class names $c \in \mathcal{C}$, the set Val^c with typical elements $\alpha, \beta, \ldots$ denotes an infinite set of *object identifiers,* where the domains for different class names are assumed to be disjoint. For each class name c, the constant $nil^c \notin Val^c$ defines the value of nil of type c. In general we will just write nil when c is clear from the context. We define Val^c_{nil} as $Val^c \cup \{nil^c\}$, and correspondingly for compound types; the set of all possible values Val_{nil} is given by $\bigcup_t Val^t_{nil}$.

A *local state* $\tau \in \Sigma_{\mathsf{loc}}$ of type $TVar \cup \{\mathsf{this}\} \rightharpoonup Val_{nil}$ holds the values of the local variables. Especially it contains a reference $\mathsf{this} \in dom(\tau)$ to the object in which the corresponding thread is currently executing. A *local configuration* (τ, stm) specifies, in addition to a local state, the point of execution. A *thread configuration* ξ is a non-empty stack $(\tau_0, stm_0)(\tau_1, stm_1) \ldots (\tau_n, stm_n)$ of local configurations, representing the chain of method invocations of the given thread. An *instance state* $\sigma^c_{inst} \in \Sigma_{inst}$ of type $IVar \rightharpoonup Val_{nil}$ assigns values to the instance variables of class c. In the following we write σ_{inst} when the class c is clear from the context. A *global state* $\sigma \in \Sigma$ is a partial function of type

$(\bigcup_{c\in C} \mathit{Val}^c) \rightharpoonup \Sigma_{inst}$ and stores for each currently *existing* object $\alpha \in \mathit{Val}^c$ its object state σ^c_{inst}. The state of an existing object α in a global state σ is given by $\sigma(\alpha)$. The set of existing objects of type c in a state σ is given by $\mathit{dom}^c(\sigma)$. We define $\mathit{dom}^{\mathsf{Int}}(\sigma)=\mathsf{Int}$ and $\mathit{dom}^{\mathsf{Bool}}(\sigma)=\mathsf{Bool}$, and correspondingly for compound types, where $\mathit{dom}(\sigma) = \bigcup_t \mathit{dom}^t(\sigma)$. We write $\mathit{dom}^c_{nil}(\sigma) = \mathit{dom}^c(\sigma) \cup \mathit{nil}^c$. For compound types t the set dom^t_{nil} is defined analogously, and $\mathit{dom}_{nil}(\sigma)$ is given by $\bigcup_t \mathit{dom}^t_{nil}$. A *global configuration* $\langle T, \sigma\rangle$ consists of a set T of thread configurations describing the currently executing threads, and a global state σ describing the currently existing objects.

Operational semantics Computation steps of a program are represented by transitions between global configurations. In the informal description we concentrate on the object-oriented constructs and those dealing with concurrency. The formalization as structural operational semantics is given in [3]. Executing $u{:=}\mathsf{new}^c$ creates a new object of type c and initializes its instance variables, but does not yet start the thread of the new object. This is done by the first invocation of the start-method, thereby initializing the first activation record of the new stack. Further invocations of the start-method are without effect.

The invocation of a method extends the call chain by creating a new local configuration. After initializing the local state, the values of the actual parameters are assigned to the formal parameters, and the thread begins to execute the body of the corresponding method. Synchronized methods of an object can be invoked only if no other threads are currently executing any synchronized methods of the same object. This mutual exclusion requirement is expressed in terms of a predicate on a set of thread configurations. This way, the semantics abstracts from any particular implementation of the synchronization mechanism.

When returning from a method call, the callee evaluates its return expression and passes it to the caller which subsequently updates its local state. The execution of the method body then terminates and the caller can continue. Returning from a method without return value is analogous. Returning from the body of the main-method or of a start-method is treated differently in that there does not exist an explicit local configuration of the caller in the stack.

The initial configuration $\langle T_0, \sigma_0\rangle$ of a program satisfies the following: $T_0 = \{(\tau_{nil}, \mathit{body}_{\mathsf{main}})\}$, where c is the type of the main class, $\alpha \in \mathit{Val}^c$, and τ_{nil} is the initial local state assigning α to this, and *nil*, 0, and *false* to class-typed, integer, and boolean variables, and correspondingly for composed types. Moreover, $\mathit{dom}(\sigma_0)=\{\alpha\}$ and $\sigma_0(\alpha)=\sigma^{nil}_{inst}$, where σ^{nil}_{inst} is the initial object state assigning initial values to variables as in τ_{nil}. We call a configuration $\langle T, \sigma\rangle$ *reachable* iff there exists a computation $\langle T_0, \sigma_0\rangle \longrightarrow^* \langle T, \sigma\rangle$, where $\langle T_0, \sigma_0\rangle$ is the initial configuration and $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$.

3 The assertion language

In this section we define two different assertion languages. The *local* assertion language is used to annotate methods directly in terms of their local variables and of the instance variables of the class to which they belong. The *global* assertion language describes a whole system of objects and their topology, and will be used in the cooperation test.

In the assertion language, we introduce as usual a countably infinite set $LVar$ of well-typed *logical variables* disjoint from the instance and the local variables occurring in programs. We use z as typical element of $LVar$, and write $LVar^t$ when specific about the type. Logical variables are used as bound variables in quantifications and, on the global level, to represent the values of local variables. To be able to argue about communication histories, we add the type Object as the supertype of all classes into the assertion language.

Table 2 defines the syntax of the assertion language. *Local expressions* $exp_l \in LExp_c^t$ of type t in class c are expressions of the programming language possibly containing logical variables. In abuse of notation, we use e, $e' \dots$ not only for program expressions of Table 1, but also for typical elements of local expressions. *Local assertions* $ass_l \in LAss_c$ in class c, with typical elements p, q, and r, are standard logical formulas over local expressions, where unrestricted quantification $\exists z(p)$ is only allowed for integer and boolean domains, i.e., z is required to be of type Int or Bool. Besides that, $\exists z \in e(p)$, resp., $\exists z \sqsubseteq e(p)$ assert the existence of an element denoted by $z \in LVar^t$, resp., a subsequence $z \in LVar^{\mathsf{list}\, t}$ of a given sequence $e \in LExp_c^{\mathsf{list}\, t}$, for which a property p holds. Restricted quantification involving objects ensures that the evaluation of a local assertion indeed only depends on the values of the instance and local variables.

Global expressions $exp_g \in GExp^t$ of type t with typical element E are constructed from logical variables, nil, operator expressions, and qualified references $E.x$ to instance variables x of objects E. *Global assertions* $ass_g \in GAss$, with typical elements P, Q, and R, are logical formulas over global expressions. Different to the local assertion language, quantification on the global level is allowed for all types. Quantifications $\exists z(P)$ range over the set of *existing* values only, i.e., the set of objects $dom_{nil}(\sigma)$ in a global configuration $\langle T, \sigma \rangle$. The semantics of the assertion languages is standard and omitted (cf. [6]).

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

Table 2. Syntax of assertions

The verification conditions defined in the next section involve the following substitution operations: By $p[\vec{e}/\vec{y}]$ we denote the standard capture-avoiding substitution. The effect of assignments to instance variables is expressed on the *global* level by the substitution $P[\vec{E}/z.\vec{x}]$, which replaces in P the instance variables $\vec{x}$ of the object referred to by z by the global expressions $\vec{E}$. To accommodate properly for the effect of assignments, though, we must not only syntactically replace the occurrences $z.x_i$ of the instance variables, but also all their *aliases* $E'.x_i$, when z and the result of the substitution applied to E' refer to the same object. As the aliasing condition cannot be checked syntactically, we define the main case $(E'.x_i)[\vec{E}/z.\vec{x}]$ of the substitution by the conditional expression $\mathsf{if}\ E'[\vec{E}/z.\vec{x}]{=}z\ \mathsf{then}\ E_i\ \mathsf{else}\ (E'[\vec{E}/z.\vec{x}]).x_i\ \mathsf{fi}$ [6]. We also use $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences $\vec{y}$, where local variables are untouched. To express on the global level a property defined by a local assertion p, we define the substitution $p[z,\vec{E}/\mathsf{this},\vec{u}]$, where the logical variable z is assumed to occur neither in p nor in $\vec{E}$, by simultaneously replacing in p all occurrences of the self-reference this by z, transforming all occurrences of instance variables x into qualified references $z.x$, and substituting all local variables u_i by the given global expressions E_i. For unrestricted quantifications $(\exists z'(p))[z,\vec{E}/\mathsf{this},\vec{u}]$ the substitution applies to the assertion p. Local restricted quantifications are transformed into global unrestricted ones where the relations $\in$ and $\sqsubseteq$ are expressed at the global level as operators. For notational convenience, we sometimes view the local variables occurring in $p[z/\mathsf{this}]$ as logical variables. Formally, these local variables should be replaced by fresh logical variables.

4 Proof system

This section presents the assertional proof system for reasoning about $Java_{MT}$ programs, formulated in terms of *proof outlines* [24, 13], i.e., where Hoare-style pre- and postconditions [14, 18] are associated with each control point. The proof system has to accommodate for shared-variable concurrency, aliasing, method invocation, synchronization, and dynamic object creation.

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

4.1 Program transformation

To be able to reason about the communication mechanism of method invocations, we split each invocation $u{:=}e_0.m(\vec{e})$ of a method different from the start-method into the sequential composition of the *communication statements* $e_0.m(\vec{e})$ and receive u. Similarly for methods without a return value, $e_0.m(\vec{e})$ gets replaced by $e_0.m(\vec{e})$; receive.

Next, we augment the program by fresh *auxiliary* variables. Assignments can be extended to multiple assignments, and additional multiple assignments to auxiliary variables can be inserted at any point. We introduce the specific auxiliary variables callerobj, id, lock, critsec, and started to represent information about the global configuration at the proof-theoretical level. The local variables callerobj and id are used as additional formal parameters of types Object and Object × Int, resp. The parameter callerobj stores the identity of the caller object, where id stores the identity of the object in which the corresponding thread has begun its execution, together with the current depth of its stack. Each statement $e_0.m(\vec{e})$ gets extended to $e_0.m(\mathsf{this}, \mathit{callee}(\mathsf{id}), \vec{e})$, where $\mathit{callee}(\alpha, n) = (\alpha, n + 1)$. If m is the start-method, the method call statement is extended to $e_0.m(\mathsf{this}, (e_0, 0), \vec{e})$, instead. The formal parameter lists get extended correspondingly. The variables callerobj and id of the thread executing the main-method in the initial configuration are initialized to *nil* and $(\alpha, 0)$, resp., where α is the initial object. The auxiliary instance variable lock of the same type Object × Int is used to reason about thread synchronization: The value $(\mathit{nil}, 0)$ states that no threads are currently executing any synchronized methods of the given object; otherwise, the value (α, n) identifies the thread which acquired the lock, together with the stack depth n, at which it has gotten the lock. The auxiliary variable lock will be only used to indicate who owns the lock, i.e., it is *not* used to *implement* the synchronization mechanism (e.g. by means of semaphores and the like). The meaning of the boolean auxiliary instance variable critsec will be explained after the introduction of critical sections. The boolean instance variable started states whether the object's start-method has already been invoked.

Finally, we extend programs by *critical sections,* a conceptual notion, which is introduced for the purpose of proof and, therefore, does not influence the control flow. Semantically, a critical section $\langle \mathit{stm} \rangle$ expresses that the statements inside are executed without interleaving with other threads. To make object creation and communication observable, we attach auxiliary assignments to the corresponding statements; to do the observation immediately after these statements, we enclose the statement and the assignment in critical sections. The formal replacement of communication and object creation statements, and method bodies is defined in Table 3.

Replace each statement of the form	by
$e_0.m(\vec{e})$	$\langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle$
method body $\mathit{stm}; \mathit{rexp}$	$\langle \vec{y}_2 := \vec{e}_2 \rangle; \mathit{stm}; \langle \mathit{rexp}; \vec{y}_3 := \vec{e}_3 \rangle$
receive u	$\langle \mathsf{receive}\ u; \vec{y}_4 := \vec{e}_4 \rangle$
receive	$\langle \mathsf{receive}; \vec{y}_4 := \vec{e}_4 \rangle$
u:=new	$\langle u := \mathsf{new}; \vec{y} := \vec{e} \rangle$
new	$\langle \mathsf{new}; \vec{y} := \vec{e} \rangle$

Table 3. Critical sections

Critical sections of method call and return statements, representing the sending parts of communication, contain the assignments critsec:=(e_0=this) and critsec:=(callerobj=this), resp. Correspondingly for the receiver part, the critical sections at the beginning of method bodies and that of receive statements include the assignment critsec:=false. I.e., critsec states whether there is a thread currently communicating within the given object, i.e., executing some self-calls or returning from a method within the object, such that the observation of the sender part is already executed but not yet that of the receiver part. Critical sections at the beginning of start-methods contain additionally the assignment started:=true; in case of a synchronized method, $\vec{y}_2{:=}\vec{e}_2$ and $\vec{y}_3{:=}\vec{e}_3$ include the assignments lock:=getlock(lock, id) representing lock reservation and lock:=release(lock, id) representing lock release, where $\mathit{getlock}(\mathit{lock}, \mathit{id})$ is given by lock if $\mathit{lock}{\neq}(\mathit{nil}, 0)$ and id, otherwise; correspondingly, $\mathit{release}(\mathit{lock}, \mathit{id})$ equals lock if $\mathit{lock}{\neq}\mathit{id}$ and $(\mathit{nil}, 0)$, otherwise. The vectors $\vec{y}, \vec{y}_1, \ldots, \vec{y}_4$ from above are auxiliary variable sequences; the values of the auxiliary variables callerobj, id, lock, critsec, and started are changed only in the critical sections, as described.

As auxiliary variables do not change the control flow of the original program, we can schedule the execution order of the augmented program as follows: For method call statements, after communication of the parameters, first the auxiliary assignment of the caller and then that of the callee is executed. Conversely, for return, the communication of the return value is followed by the execution of the assignment of the callee and then that of the caller, in this order. Note that these three steps for method invocation and return may not be interleaved by other threads. Control points within a critical section and at the beginning of a method body, which we call *non-interleaving* points, do not change the program behaviour. All other control points we call *interleaving* points.

To specify invariant properties of the system, the transformed programs are *annotated* by attaching local assertions to all control points. Besides that, for each class c, the annotation defines a local assertion I_c called *class invariant,* which refers only to instance variables, and expresses invariant properties of the instances of the class. Finally, the *global invariant* $GI \in GAss$ specifies the communication structure of the program. We require that for all qualified references $E.x$ in GI with $E \in GExp^c$, all assignments to x in class c are enclosed in critical sections. An annotated transformation of prog, denoted by prog', is called a *proof outline.* For annotated programs, we use the standard notation $\{p\}\ \mathit{stm}\ \{q\}$ to express that p and q are the assertions in front of and after the statement stm. We call $\mathit{pre}(\mathit{stm}){=}p$ and $\mathit{post}(\mathit{stm}){=}q$ the *pre-* and *postconditions* of stm.

4.2 Proof system

The proof system formalizes a number of *verification conditions* which ensure that in each reachable configuration all preconditions are satisfied, and that the class and global invariants hold. To cover concurrency and communication, the verification conditions are grouped into local correctness conditions, an interference freedom test, and a cooperation test.

Before specifying the verification conditions, we first fix some auxiliary functions and notations. Let $\mathit{InitValue} : \mathit{Var} \to \mathit{Val}$ be a function assigning an initial value to each variable, that is nil, false, and 0 for class, boolean, and integer types, respectively, and analogously for composed types, where sequences are initially empty. For each class c, let IVar_c be the set of instance variables in class c, and let $\mathit{Init}(z)$ denote the global assertion $\bigwedge_{x \in IVar_c} z.x = \mathit{InitValue}(x)$, for all $z \in LVar^c$, expressing that the object denoted by z is in its initial object state. The predicate $\mathit{samethread}((\alpha_1, n_1), (\alpha_2, n_2))$, defined by $\alpha_1 = \alpha_2$, characterizes the relationship between threads. Similarly, the relation $<$ of the same type is given by $(\alpha_1, n_1) < (\alpha_2, n_2)$ iff $\alpha_1 = \alpha_2$ and $n_1 < n_2$.

Initial correctness *Initial correctness* means that the precondition of the main statement is satisfied by the initial object and local states, where $\mathsf{id} = (\mathsf{this}, 0)$ and all other variables have their initial values. Furthermore, the global invariant is satisfied by the first reachable stable configuration, i.e., by the global state after the execution of the critical section at the beginning of the main-method.

Definition 1. *A proof outline prog′ is* initially correct, *if*

$$\models_{\mathcal{L}} \mathit{pre}(\mathit{body}_{\mathsf{main}})[(\mathsf{this}, 0)/\mathsf{id}][\mathit{InitValue}(\vec{y})/\vec{y}], \tag{1}$$

$$\models_{\mathcal{G}} \exists z \Big(z \neq \mathsf{nil} \wedge \mathit{Init}(z) \wedge \forall z' (z' = \mathsf{nil} \vee z = z') \Big) \rightarrow \exists z \left(\mathit{GI}[\vec{e}_2 / z.\vec{y}_1] \right), \tag{2}$$

where $\mathit{body}_{\mathsf{main}} = \langle \vec{y}_1 := \vec{e}_1 \rangle; \mathit{stm}$ *is the body of the main-method,* $\vec{y}$ *are the variables occurring in* $\mathit{pre}(\mathit{body}_{\mathsf{main}})$*,* z *is of the type of the main class, and* $z' \in LVar^{\mathsf{Object}}$.

Local correctness A proof outline is *locally correct*, if the usual verification conditions [7] for standard sequential constructs hold: The precondition of a multiple assignment must imply its postcondition after the execution of the assignment. For assignments occurring outside of critical sections, $\neg\mathsf{critsec}$ expresses the enabledness of the assignment. Furthermore, all assertions of a class are required to imply the class invariant. Inductivity for statements involving object creation or communication are verified on the global level in the cooperation test.

Definition 2. *A proof outline is* locally correct*, if for each class c with class invariant I_c, all multiple assignments $\vec{y} := \vec{e}$ and $\vec{y}_{crit} := \vec{e}_{crit}$ occurring outside and inside of critical sections, resp., and all statements stm in class c,*

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

$$\models_{\mathcal{L}} \mathit{pre}(\vec{y}_{crit} := \vec{e}_{crit}) \rightarrow \mathit{post}(\vec{y}_{crit} := \vec{e}_{crit})[\vec{e}_{crit}/\vec{y}_{crit}] \tag{4}$$

$$\models_{\mathcal{L}} (\mathit{pre}(\mathit{stm}) \rightarrow I_c) \wedge (\mathit{post}(\mathit{stm}) \rightarrow I_c). \tag{5}$$

The interference freedom test In this section we formalize conditions that ensure the invariance of local properties of a thread under the activities of other threads. Since we disallow public instance variables in Java_{MT}, we only have to deal with the invariance of properties under the execution of statements within the same object. Containing only local variables, communication and object creation statements do not change the state of the executing object. Thus we only have to take assignments $\vec{y} := \vec{e}$ into account.

Satisfaction of a local property of a thread at an *interleaving* point may clearly be affected by the execution of assignments by a *different* thread in the same object (Eq. (6)). If, otherwise, the property describes the *same* thread that executes the assignment, the only interleaving points endangered are those waiting for a return value earlier in the current execution stack, i.e., we have to show the invariance of preconditions of receive statements (Eq. (7)). Since an object can call a method of itself, the preconditions of method bodies and the postconditions of receive statements, representing *non-interleaving* points, must be proven interference free, as well: For method invocation, after communication, the caller executes the assignment of its critical section $\langle e_0.m(\vec{e}); \vec{y}_1{:=}\vec{e}_1\rangle$, which may affect the precondition of the body of the called method (Eq. (8)). Likewise when returning from a method, after communicating the return value, first the callee executes the multiple assignment of its critical section $\langle \mathit{rexp}; \vec{y}_3{:=}\vec{e}_3\rangle$, which can affect the postcondition of the receive statement of the caller (Eq. (9)).

Definition 3. *A proof outline is* interference free, *if for all classes c the following conditions hold, where we denote by p' the assertion p with each local variable u different from* this *replaced by a fresh one denoted by u':*

- *For all statements $\vec{y}{:=}\vec{e}$ in a critical section and all assertions p representing an interleaving point in class c, if not both the statement and the assertion occur in a synchronized method, then*

$$\models_{\mathcal{L}} p' \wedge \mathit{pre}(\vec{y}{:=}\vec{e}) \wedge \neg \mathit{samethread}(\mathsf{id}', \mathsf{id}) \rightarrow p'[\vec{e}/\vec{y}]. \tag{6}$$

  For statements $\vec{y}{:=}\vec{e}$ in critical sections we have the additional antecedent $\neg$critsec.
- *For all statements $\vec{y}{:=}\vec{e}$ in a critical section and all assertions p in c, if p is the precondition of a receive statement, then*

$$\models_{\mathcal{L}} p' \wedge \mathit{pre}(\vec{y}{:=}\vec{e}) \wedge \mathsf{id}' < \mathsf{id} \rightarrow p'[\vec{e}/\vec{y}]. \tag{7}$$

  If $\vec{y}{:=}\vec{e}$ occurs outside of critical sections, we have the additional antecedent $\neg$critsec.
- *For all statements $\langle e_0.m(\vec{u}); \vec{y}{:=}\vec{e}\rangle$ in c with $e_0 \in \mathit{Exp}_c^c$, if p is the precondition of the body of $m{\neq}\mathsf{start}$ in c, then*

$$\models_{\mathcal{L}} p' \wedge \mathit{pre}(\vec{y}{:=}\vec{e}) \wedge e_0{=}\mathsf{this} \wedge \mathsf{id}'{=}\mathit{callee}(\mathsf{id}) \rightarrow p'[\vec{e}/\vec{y}]. \tag{8}$$

  If $m{=}\mathsf{start}$, then $\mathsf{id}'{=}(\mathsf{this}, 0)$ replaces $\mathsf{id}'{=}\mathit{callee}(\mathsf{id})$.
- *For all statements $\langle \mathit{rexp}; \vec{y}{:=}\vec{e}\rangle$ in a method m of c, if p is the postcondition of a receive statement preceded by a critical section invoking method m of $e_0 \in \mathit{Exp}_c^c$, then*

$$\models_{\mathcal{L}} p' \wedge \mathit{pre}(\vec{y}{:=}\vec{e}) \wedge e_0'{=}\mathsf{this} \wedge \mathsf{id}{=}\mathit{callee}(\mathsf{id}') \rightarrow p'[\vec{e}/\vec{y}]. \tag{9}$$

Note that we have to replace the local variables different from this occurring in p to avoid name clashes with those in $\vec{y}{:=}\vec{e}$ and its associated precondition.

The cooperation test Whereas the verification conditions associated with local correctness and interference freedom cover the effects of assigning side-effect-free expressions to variables, the *cooperation test* deals with method call and object creation. Since different objects may be involved, it is formulated in the global assertion language. We start with the cooperation test for method invocations.

In the following definition, the logical variable z denotes the object calling a method and z' refers to the callee. The cooperation test assures that the local assertions at both ends of the communication hold, immediately after the values have been communicated. When calling a method, the postcondition of the method invocation statement and the precondition of the invoked method's body must hold after passing the parameters (Eq. (10)). In the global state prior to the call, we can assume that the global invariant, the precondition of the method invocation at the caller side, and the class invariant of the callee hold. For synchronized methods, additionally the lock of the callee object is free, or the lock has been acquired in the call chain of the executing thread. This is expressed by the predicate $\mathit{isfree}(z'.\mathsf{lock}, \mathsf{id})$ defined as $z'.\mathsf{lock}{=}(\mathsf{nil}, 0) \vee z'.\mathsf{lock} \leq \mathsf{id}$, where id is the identity of the caller. Equation (11) works similarly, where the postconditions of the corresponding return- and receive-statements are required to hold after the communication when returning from a method. Note that we rename the local variables of the callee in order to avoid name clashes with that of the caller.

The global invariant GI, which describes invariant properties of a program, is not allowed to refer to instance variables whose values are changed outside of critical sections. Consequently, it will be automatically invariant over the execution of statements outside of critical sections. For the critical sections themselves, however, the invariance must be shown as part of the cooperation test. A difference between the treatment of the local assertions and the global invariant is, that the latter does not necessarily hold immediately after communication, but only after the accompanying assignments to the auxiliary variables of both the caller and callee have been completed. This is reflected in the two substitutions applied to the global invariant on the right-hand sides of the implications.

Invoking the start-method of an object whose thread is already started, or returning from a start-method or from the first execution of the main-method does not have communication effects; Equations (12) and (13) take care about the validity of the postconditions and the invariance of the global invariant.

Definition 4. *A proof outline satisfies the* cooperation test *for* communication, *if for all classes c and statements $\langle e_0.m(\vec{e}); \vec{y}_1{:=}\vec{e}_1\rangle; \langle \mathsf{receive}\, v; \vec{y}_4{:=}\vec{e}_4\rangle$ in c with $e_0 \in Exp_c^{c'}$, where method m of c' with formal parameter list $\vec{u}$ is synchronized* *with* $body_{m,c'} = \langle \vec{y}_2 := \vec{e}_2 \rangle; stm; \langle \mathsf{return}\ e_{ret}; \vec{y}_3 := \vec{e}_3 \rangle$,

$$\begin{aligned}
&\models_{\mathcal{G}} GI \wedge pre(e_0.m(\vec{e}))[z/\mathsf{this}] \wedge I_{c'}[z'/\mathsf{this}] \wedge e_0[z/\mathsf{this}]{=}z' \wedge \mathit{isfree}(z'.\mathsf{lock}, \mathsf{id}) \\
&\quad \rightarrow post(e_0.m(\vec{e}))[z/\mathsf{this}] \wedge pre'(body_{m,c'})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge \\
&\quad\quad GI[\vec{E}_2/z'.\vec{y}_2][\vec{E}_1/z.\vec{y}_1] \qquad (10) \\
&\models_{\mathcal{G}} GI \wedge pre'(\mathsf{return}\ e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge pre(\mathsf{receive}\ v)[z/\mathsf{this}] \wedge \ e_0[z/\mathsf{this}]{=}z' \\
&\quad \rightarrow post'(\mathsf{return}\ e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge post(\mathsf{receive}\ v)[z, E_{ret}/\mathsf{this}, v] \wedge \\
&\quad\quad GI[\vec{E}_4/z.\vec{y}_4][\vec{E}_3/z'.\vec{y}_3], \qquad (11)
\end{aligned}$$

where $z \in LVar^c$ *and* $z' \in LVar^{c'}$ *are distinct fresh logical variables, and* id *is the auxiliary local variable of the caller viewed as logical variable on the global level. The assertion* $pre'(body_{m,c'})$ *is* $pre(body_{m,c'})$ *with every local variable except the formal parameters and* this, *of class, boolean, or integer type replaced by* nil, false, *or* 0, *respectively, and correspondingly for composed types;* $pre'(\mathsf{return}\ e_{ret})$, $post'(\mathsf{return}\ e_{ret})$, *and* e'_{ret} *denote the given assertions and expressions with every local variable except the formal parameters and* this *replaced by a fresh one. Furthermore,* $\vec{E}_1 = \vec{e}_1[z/\mathsf{this}]$, $\vec{E}_j = \vec{e}_j[z', \vec{E}/\mathsf{this}, \vec{u}]$ *for* $j{=}2, 3$, $\vec{E}_4 = \vec{e}_4[z, E_{ret}/\mathsf{this}, v]$, *where* $\vec{E} = \vec{e}[z/\mathsf{this}]$ *and* $E_{ret} = e'_{ret}[z', \vec{E}/\mathsf{this}, \vec{u}]$. *For the invocation of* non-synchronized methods, *the antecedent* $\mathit{isfree}(z'.\mathsf{lock}, \mathsf{id})$ *is dropped. The verification conditions for methods without return value are analogous. For invocations of* start*-methods, only (10) applies with the additional antecedent* $\neg z'.\mathsf{started}$. *For the case that the thread is already started,*

$$\begin{aligned}
&\models_{\mathcal{G}} GI \wedge pre(e_0.\mathsf{start}(\vec{e}))[z/\mathsf{this}] \wedge I_{c'}[z'/\mathsf{this}] \wedge e_0[z/\mathsf{this}]{=}z' \wedge z'.\mathsf{started} \\
&\quad \rightarrow post(e_0.\mathsf{start}(\vec{e}))[z/\mathsf{this}] \wedge GI[\vec{E}_1/z.\vec{y}_1] \qquad (12)
\end{aligned}$$

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

$$\begin{aligned}
&\models_{\mathcal{G}} GI \wedge pre(\mathsf{return})[z'/\mathsf{this}] \wedge \mathsf{id}{=}(z', 0) \\
&\quad \rightarrow post(\mathsf{return})[z'/\mathsf{this}] \wedge GI[\vec{E}_3/z'.\vec{y}_3]. \qquad (13)
\end{aligned}$$

The substitution of $\vec{u}$ by $\vec{E}$ in the condition $pre'(body)[z', \vec{E}/\mathsf{this}, \vec{u}]$ reflects the parameter-passing mechanism, where $\vec{E}$ are the actual parameters $\vec{e}$ represented at the global assertional level. This substitution also identifies the callee, as specified by its formal parameter id. Note that the actual parameters do not contain *instance* variables, i.e., their interpretation does not change during the execution of the method body. Therefore, $\vec{E}$ can be used not only to logically capture the conditions at the entry of the method body, but at the exit of the method body, as well, as shown in Equation (11).

Furthermore, the cooperation test needs to handle critical sections of object creation taking care of the preservation of the global invariant, the postcondition of the new-statement, and the new object's class invariant. The extension of the global state with a freshly created object is formulated in a strongest postcondition style, using existential quantification to refer to the old, changed value, i.e., z' of type $LVar^{\mathsf{list\,Object}}$ represents the existing objects prior to the extension. Moreover, that the created object's identity is fresh and that the new instance u is properly initialized is captured by the global assertion $Fresh(z', u)$ defined as $u{\neq}\mathsf{nil} \wedge u \notin z' \wedge Init(u) \wedge \forall v(v \in z' \vee v{=}u)$, where $z' \in LVar^{\mathsf{list\,Object}}$, and $Init(u)$ is as defined in Section 4.2. To have quantifications on the left-hand side of the implication to refer to the set of existing objects *before* the new-statement, we need to *restrict* any existential quantification to range over objects from z', only. For a global assertion P we define its restriction $P \downarrow z'$ by replacing all quantifications $\exists z(P')$ in P by $\exists z(z \in z' \wedge P')$, when z is of type c or Object, and by $\exists z(z \sqsubseteq z' \wedge P')$, when z is of type list c or list Object, and correspondingly for composed types.

Definition 5. *A proof outline satisfies the* cooperation test *for* object creation*, if for all classes c' and statements $\langle u{:=}\mathsf{new}^c; \vec{y}{:=}\vec{e}\rangle$ in c':*

$$\begin{aligned}\models_{\mathcal{G}} \exists z'\Big(&Fresh(z', u) \wedge (GI \wedge \exists u(pre(u{:=}\mathsf{new}^c)[z/\mathsf{this}])) \downarrow z'\Big) \\ &\rightarrow post(u{:=}\mathsf{new}^c)[z/\mathsf{this}] \wedge I_c[u/\mathsf{this}] \wedge GI[\vec{E}/z.\vec{y}],\end{aligned} \tag{14}$$

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

Theorem 1. *The proof method is sound and relative complete.*

The soundness of our method is shown by a standard albeit tedious induction on the length of the computation. Proving its completeness involves the introduction of appropriate assertions expressing reachability and auxiliary *history variables.* The details of the proofs can be found in [3].

5 Conclusion

In this paper we introduce an assertional proof method for a multithreaded sublanguage of *Java*. In [2] the basic ideas have been introduced for proof outlines by means of a modular integration of the interference freedom and the cooperation test for a more restricted version of *Java*. The present paper offers such an integration for a more concrete version of *Java* by incorporating *Java*'s *reentrant* synchronization mechanism. This requires a non-trivial extension of the proof method by a more refined mechanism for the identification of threads. Its soundness and completeness is proved in [3]. As such, our paper presents a first assertional proof method for reasoning about threads in Java which is complete in the sense that it forms a basis for extending our proof method to more specific *Java*-synchronization methods, such as wait(), notify(), and notifyAll(), and the important feature of exception handling [20].

Most papers in the literature focus on *sequential* subsets of Java [28, 10, 26, 27, 11, 31, 1, 32, 33]. Formal semantics of *Java*, including multithreaded execution, and its virtual machine in terms of abstract state machines is given in [29]. A structural operational semantics of multithreaded *Java* can be found in [12].

Currently we are developing in the context of the European Fifth Framework RTD project Omega and the bilateral NWO/DFG project MobiJ a front-end tool for the computer-aided specification and verification of *Java* programs based on our proof method. Such a front-end tool consists of an editor and a parser for annotating *Java* programs, and of a compiler which translates these annotated *Java* programs into corresponding verification conditions. A theorem prover (HOL or PVS) is used for verifying the validity of these verifications conditions. Of particular interest in this context is an integration of our method with related approaches like the LOOP project [17, 23].

More in general, our future work focuses on including more features of multithreading, inheritance, and polymorphic extensions involving behavioral subtyping [5].

Acknowledgment This work was partly supported by the bilateral NWO/DFG project MobiJ. We also thank the reviewers for their helpful comments.

References

1. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In Bidoit and Dauchet [9], pages 682–696.
2. E. Ábrahám-Mumm and F. de Boer. Proof-outlines for threads in Java. In Palamidessi [25].
3. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept: Soundness and completeness. Technical Report TR-ST-01-2, Lehrstuhl für Software-Technologie, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität Kiel, 2001.
4. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS State-of-the-Art-Survey. Springer, 1999.
5. P. America. A behavioural approach to subtyping in object-oriented programming languages. Technical report 443, Phillips Research Laboratories, 1989.
6. P. America and F. Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1993.
7. K. R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transact. on Progr. Lang. and Syst.*, 3(4):431–483, 1981.
8. K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transact. on Progr. Lang. and Syst.*, 2:359–385, 1980.
9. M. Bidoit and M. Dauchet, editors. *Theory and Practice of Software Development, Proc. of the 7th Int. Joint Conf. of CAAP/FASE, TAPSOFT'97*, volume 1214 of *LNCS.* Springer, 1997.
10. R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* PhD thesis, Universität Passau, 1991. See also Springer LNCS 562.
11. P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, 1995.
12. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [4].
13. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods.* Cambridge University Press, 2001.

14. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
15. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
16. C. Hankin, editor. *Programming Languages and Systems: Proc. of ESOP '98, Held as Part of ETAPS '98*, volume 1381 of *LNCS.* Springer, 1998.
17. J. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Hankin [16].
18. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [19].
19. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science.* International Series in Computer Science. Prentice Hall, 1989.
20. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.
21. H. Hussmann, editor. *Fundamental Approaches to Software Engineering*, volume 2029 of *LNCS.* Springer, 2001.
22. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
23. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.
24. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
25. C. Palamidessi, editor. *CONCUR 2000*, volume 1877 of *LNCS.* Springer, 2000.
26. A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* Technische Universität München, 1997. Habilitationsschrift.
27. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In Swierstra [30], pages 162–176.
28. B. Reus, R. Hennicker, and M. Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Hussmann [21], pages 300–316.
29. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine.* Springer, 2001.
30. S. Swierstra, editor. *Proc. of ESOP '99*, volume 1576 of *LNCS.* Springer, 1999.
31. D. von Oheimb. Axiomatic sematics for Java^{light} in Isabelle/HOL. In S. Drossopoulo, S. Eisenbach, B. Jacobs, G. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, number 269, 5/2000 in Technical Report. Fernuniversität Hagen, 2000.
32. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency – Practice and Experience*, 2001. To appear.
33. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. Submitted for publication, 2002.
34. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml.* Object Technology Series. Addison-Wesley, 1999.