Verification for Java's Reentrant Multithreading Concept

October 21, 2001

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

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

Abstract. Besides the features of a class-based object-oriented language, *Java* integrates concurrency via its thread-classes, allowing for a *multithreaded* flow of control. The concurrency model 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* [14] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [4, 28, 11]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sublanguages (see e.g. [19, 31, 26]). 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* [23], i.e., of annotated programs where Hoare-style assertions [13, 17] 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 and which are standard.

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 [21] 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 [23] and in [21]. 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) [32], 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 run-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 and supports class types and primitive, i.e., non-reference types. 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 $\mathsf{list}\, t$. Side-effect expressions without a value, i.e., 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 *temporary* variables, where instance variables are always private in $Java_{MT}$. They hold the state of an object and exist throughout the object's lifetime. Temporary variables play the role of formal parameters and local variables of method definitions and only exist during the execution of the method to which they belong. Therefore they store the local state of a thread of execution. The set of variables $Var = IVar \dot{\cup} TVar$ with typical element y is given as the disjoint union of the instance and the temporary variables. Var^t denotes the set of all variables of type t, and correspondingly for $IVar^t$ and $TVar^t$. As we assume a monomorphic type discipline, $TVar^t \cap TVar^{t'} = \emptyset$ for distinct types t and t'. We use $x, x', x_1, \ldots$ as typical elements from $IVar$ and $u, u', u_1, \ldots$ as typical elements from $TVar$.

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 $e \in Exp_c^t$ and those with side effects $sexp \in SExp_c^t$, 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*. In the sequel we also refer to statements in the body of a synchronized method as being synchronized. As mentioned earlier, all classes in $Java_{MT}$ are thread classes; the corresponding run-methods are denoted by $meth_{\mathsf{run}}$ with body stm^c_{run}. The main class $class_{\mathsf{main}}$ contains as entry point of the program execution the specific method $meth_{\mathsf{main}}$ with body $stm^{prog}_{\mathsf{main}}$.

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	$::= x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp,\ldots,exp)$	$e \in Exp^t_c$	expressions
$sexp$	$::= \mathsf{new}^{c'} \mid exp.m(exp,\ldots,exp) \mid exp.\mathsf{start}()$	$sexp \in SExp^t_c$	side-effect exp.
stm	$::= sexp \mid x := exp \mid u := exp \mid u := sexp$		
	$\mid\ \epsilon \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm$		
	$\mid\ \mathsf{while}\ exp\ \mathsf{do}\ stm \ldots$	$stm \in Stm_c$	statements
$modif$	$::= \mathsf{nsync} \mid \mathsf{sync}$		modifiers
$rexp$	$::= \mathsf{return} \mid \mathsf{return}\ exp$		
$meth$	$::= modif\ m(u,\ldots,u)\{\ stm; rexp\}$	$meth \in Meth_c$	methods
$meth_{\mathsf{run}}$	$::= modif\ \mathsf{run}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{run}} \in Meth_c$	run-method
$meth_{\mathsf{main}}$	$::= \mathsf{nsync}\ \mathsf{main}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{main}} \in Meth_c$	main-method
$class$	$::= c\{meth\ \ldots\ meth\ meth_{\mathsf{run}}\}$	$class \in Class$	class defn's
$class_{\mathsf{main}}$	$::= c\{meth\ \ldots\ meth\ meth_{\mathsf{run}}\ meth_{\mathsf{main}}\}$	$class_{\mathsf{main}} \in Class$	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. 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 minimal value $\perp^c \notin Val^c$ stands for the undefined value, i.e., the value of nil in the corresponding type, and we define $Val^c_\perp$ as $Val^c \cup \{\perp^c\}$. In general we will just write $\perp$ when c is clear from the context. The set of all possible non-nil values $\bigcup_t Val^t$ is written as Val, and $Val_\perp$ denotes $Val \cup \bigcup_c \{\perp^c\}$.

A *local state* $\tau \in \Sigma_{\mathsf{loc}}$ of type $TVar \rightharpoonup Val_\perp$ holds the values of the temporary 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 $IVar \rightharpoonup Val_\perp$ 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 $(\bigcup_{c \in \mathcal{C}} Val^c) \rightharpoonup \Sigma_{inst}$ and stores for each currently *existing* object $\alpha \in Val^c$ its object state σ^c_{inst}. The set of existing objects of type c in a state σ is given by $dom^c(\sigma)$; we refer to the set of all existing objects by $dom(\sigma)$. We define $dom^{\mathsf{Int}}(\sigma) = \mathsf{Int}$ and $dom^{\mathsf{Bool}}(\sigma) = \mathsf{Bool}$. For compound types, dom^t is defined correspondingly. Note that the state of an existing object α in a global state σ is given by $\sigma(\alpha)$. 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 the Appendix and 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 which calls the object's run-method, thereby initializing the first activation record of the new stack. Only the first invocation of the start-method has this effect. This is captured by the predicate *started*. 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. Note that 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, and returning from the body of the main-method or of a run-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 *prog* satisfies the following: $T_0 = \{(\tau_\perp, stm^{prog}_{\mathsf{main}})\}$, where c is the type of the main class, $\alpha \in Val^c$, and $\tau_\perp$ is the initial local state assigning α to this, and $\perp$, 0, and false to class-typed, integer, and boolean variables, and correspondingly for composed types. Moreover, $dom(\sigma_0) = \{\alpha\}$ and $\sigma_0(\alpha) = \sigma^\perp_{inst}$, where $\sigma^\perp_{inst}$ is the initial object state assigning initial values to variables as in $\tau_\perp$. We call a configuration $\langle T, \sigma \rangle$ of *prog 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 temporary variables and 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 language of assertions, we introduce as usual a countably infinite set $LVar$ of well-typed *logical variables* $LVar^t$ of type t with typical element z disjoint from the instance and the local variables occurring in programs. Logical variables are used as bound variables in quantifications and, on the global level, to represent the values of local variables.

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, $e' \ldots$ 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 allowed for integer and boolean domains only, 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 temporary variables.

Global expressions $exp_g \in GExp^t$ 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. In deference to the local assertion language, quantification on the global level is allowed for all types. Quantifications $\exists z(P)$ over objects and sequences of objects, i.e., for $z \in LVar^t$, where t stands for c, Object, $\mathsf{list}\,c$, and $\mathsf{list}\,\mathsf{Object}$, range over the set of *existing* objects only, i.e., the set of objects $dom(\sigma)$ in a global configuration $\langle T, \sigma\rangle$.

$exp_l ::= z \mid x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp_l, \ldots, exp_l)$	$e \in LExp_c^t$	local expressions
$ass_l ::= exp_l \mid \neg ass_l \mid ass_l \wedge ass_l$		
$\quad\mid\ \exists z(ass_l) \mid \exists z \in exp_l(ass_l) \mid \exists z \sqsubseteq exp_l(ass_l)$	$p \in LAss_c$	local assertions
$exp_g ::= z \mid \mathsf{nil} \mid \mathsf{f}(exp_g, \ldots, exp_g) \mid exp_g.x$	$E \in GExp^t$	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 semantics of the assertion languages is standard and thus omitted (cf. [6]).

The verification conditions defined in the next section involve the following substitution operations: By $p[\vec{e}/\vec{u}]$ and $p[\vec{e}/\vec{x}]$ we denote the standard capture-avoiding substitutions. We additionally use $p[\vec{e}/\vec{y}]$ to denote the simultaneous replacement of temporary and instance variables. 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 if $E'[\vec{E}/z.\vec{x}] = z$ then E_i else $(E'[\vec{E}/z.\vec{x}]).x_i$ fi [6]. We also use $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences $\vec{y}$, where local variables are untouched. To express a property defined by a local assertion p on the global level, 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 temporary variables occurring in $p[z/\mathsf{this}]$ as logical variables. Formally, these temporary variables should be replaced by fresh logical variables.

4 Proof outlines

This section presents the assertional proof system for reasoning about $Java_{MT}$ programs, formulated in terms of *proof outlines* [23, 12], i.e., where Hoare-style pre- and postconditions [13, 17] are associated with each program statement. The proof system has to accommodate for shared-variable concurrency, aliasing, method invocation, 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.

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 three specific auxiliary variables id, lock, and $\mathsf{started}$ to represent information about the global configuration at the proof-theoretical level. The temporary variable id is used as additional formal parameter of type $\mathsf{Object} \times \mathsf{Int}$. It stores the identity of the object in which the corresponding thread has begun its execution, together with the current depth of its stack. Each output statement $e_0.m(\vec{e})$ gets extended to $e_0.m(callee(\mathsf{id}), \vec{e})$, where $callee(\alpha, n) = (\alpha, n+1)$. If m is the start-method, the output statement is extended to $e_0.m((e_0, 0), \vec{e})$, instead. The formal parameter lists get extended correspondingly. The temporary variable id of the thread executing the main-method in the initial configuration is initialized to $(\bot, 0)$. The auxiliary instance variable lock of the same type $\mathsf{Object} \times \mathsf{Int}$ is used to reason about thread synchronization: The value $\bot$ 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 boolean instance variable $\mathsf{started}$ 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 $\langle stm \rangle$ expresses that the statements inside are executed without interleaving with 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. Formally, we replace output statements $e_0.m(\vec{e})$ by $\langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1 \rangle$, where $\vec{y}_1$ does not contain variables occurring in $\vec{e}$. Input statements $\mathsf{receive}\, u$ and $\mathsf{receive}$ get replaced by $\langle \mathsf{receive}\, u; \vec{y}_4 := \vec{e}_4 \rangle$ and $\langle \mathsf{receive}; \vec{y}_4 := \vec{e}_4 \rangle$. Object creation statements $u := \mathsf{new}$ and new are transformed into $\langle u := \mathsf{new}; \vec{y} := \vec{e} \rangle$ and $\langle \mathsf{new}; \vec{y} := \vec{e} \rangle$, respectively. Furthermore, we introduce additional critical sections at the beginning and at the end of each method body $stm; rexp$, yielding $\langle \vec{y}_2 := \vec{e}_2 \rangle; stm; \langle rexp; \vec{y}_3 := \vec{e}_3 \rangle$. Critical sections at the beginning of run-methods contain the assignment $\mathsf{started} := \mathsf{true}$; in case of a synchronized method, $\vec{y}_2 := \vec{e}_2$ and $\vec{y}_3 := \vec{e}_3$ include the assignments $\mathsf{lock} := \mathsf{getlock}(\mathsf{lock}, \mathsf{id})$ representing the lock reservation and $\mathsf{lock} := \mathsf{release}(\mathsf{lock}, \mathsf{id})$ representing the lock release, respectively, where $getlock(lock, id)$ is $lock$ if $lock \neq \bot$ and id, otherwise, and correspondingly $release(lock, id)$ denotes $lock$ if $lock \neq id$ and $\bot$, otherwise. The vectors $\vec{y}, \vec{y}_1, \ldots, \vec{y}_4$ from above are sequences of auxiliary variables; the values of the three specific auxiliary variables 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 output 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 of the caller, in that order. Note that these three steps for method invocation and return may not be interleaved by other threads, since the control points in between are non-interleaving points.

To specify invariant properties of the system, the transformed programs are *annotated* by attaching pre- and postconditions, formulated in the local assertion language, to all occurrences of statements. 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 properties of communication between objects. We require that for all qualified references $E.x$ in GI, all assignments to x in class c are enclosed in critical sections. An annotated transformation of $prog$ is denoted by $prog'$. For annotated programs, we use the standard notation $\{p\}\ stm\ \{q\}$ to express that $pre(stm) = p$ and $post(stm) = q$.

An annotated transformation $prog'$ is a *proof outline* when the usual sequential control-flow requirements for the assertions of composed statements hold; for example, for the sequential composition $stm_1; stm_2$ we have $pre(stm_1) = pre(stm_1; stm_2)$, $post(stm_1) = pre(stm_2)$, and $post(stm_2) = post(stm_1; stm_2)$.

4.2 Proof system

The proof system formalizes a number of *verification conditions* which ensure that in each reachable configuration the precondition of each existing thread's statement is satisfied and 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 $InitValue : Var \to Val$ be a function assigning the value of $\bot$ to the variable lock, and the initial value of type t to each other variable $y \in Var^t$, that is $\bot$, false, and 0 for class, boolean, and integer types, respectively, and analogously for composed types, especially for sequences. For each class c, let $IVar_c$ be the set of instance variables in class c, and let $Init(z)$ denote the global assertion $\bigwedge_{x \in IVar_c} z.x = InitValue(x)$, for all $z \in LVar^c$, expressing that the object denoted by z is in its initial object state. To characterize the relationship between threads, we define $samethread$ by $samethread((\alpha_1, n_1), (\alpha_2, n_2))$ iff $\alpha_1 = \alpha_2$. Similarly, the relation $<$ of the same type is given by $(\alpha_1, n_1) < (\alpha_2, n_2)$ iff $\alpha_1 = \alpha_2$ and $n_1 < n_2$. Note that $\bot \not< (\alpha, n)$.

Initial correctness *Initial correctness* means that the precondition of the main statement and the global invariant are satisfied by the initial configuration, i.e., $pre(stm^{prog}_{\mathsf{main}})$ is satisfied by the initial local state in that $\mathsf{id}{=}(\mathsf{nil}, 0)$ and all other variables have their initial values, and the global assertion GI is satisfied by the initial global state in that the only existing object is in its initial state.

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

$$\models_{\mathcal{L}} pre(stm^{prog}_{\mathsf{main}})[(\mathsf{nil},0)/\mathsf{id}][InitValue(\vec{y})/\vec{y}] \tag{1}$$

$$\models_{\mathcal{G}} \exists z\Big((\forall z'(z=z')) \wedge Init(z)\Big) \rightarrow GI, \tag{2}$$

where $\vec{y}$ denotes the variables occurring in $pre(stm^{prog}_{\mathsf{main}})$, 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 to instance and local variables must imply the postcondition after execution of the assignment. As output and return statements do not affect the state of the executing thread, their preconditions must directly imply their postconditions. Finally, the pre- and postconditions of all statements of a class are required to imply the class invariant.

Definition 2. *A proof outline is* locally correct, *if for each class $c \in \mathcal{C}$ with class invariant I_c, and all multiple assignments $\vec{y} := \vec{e}$, input statements $e_0.m(\vec{e})$, return statements $rexp$, and statements $stm \in Stm_c$ in class c, we have, besides the usual verification conditions for sequential control flow,*

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

$$\models_{\mathcal{L}} pre(e_0.m(\vec{e})) \rightarrow post(e_0.m(\vec{e})) \tag{4}$$

$$\models_{\mathcal{L}} pre(rexp) \rightarrow post(rexp) \tag{5}$$

$$\models_{\mathcal{L}} (pre(stm) \rightarrow I_c) \wedge (post(stm) \rightarrow I_c). \tag{6}$$

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 temporary 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 (Equation (7)). 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 (Equation (8)). 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, the caller, after communicating the actual parameters, 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 (Equation (9)). Likewise when returning from a method and after communicating the return value, first the callee executes the multiple assignment of the critical section $\langle rexp; \vec{y}_3 := \vec{e}_3 \rangle$, which can affect the postcondition of the receive statement of the caller (Equation (10)).

Definition 3. *A proof outline is* interference free, *if for all classes c the following conditions hold, where for an assertion p we denote by p' the result of replacing each temporary variable u in p by a fresh one denoted by u':*

– *For all statements $\vec{y} := \vec{e}$ and assertions p in class c, if p represents an interleaving point, then*

$$\models_{\mathcal{L}} p' \wedge pre(\vec{y} := \vec{e}) \wedge \mathsf{this} = \mathsf{this}' \wedge \neg samethread(\mathsf{id}', \mathsf{id}) \rightarrow p'[\vec{e}/\vec{y}]. \quad (7)$$

– *For all statements $\vec{y} := \vec{e}$ and assertions p in c, if p is the precondition of a receive statement, then*

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

– *For all statements $\langle e_0.m(\vec{u}); \vec{y} := \vec{e} \rangle$ in c with $e_0 \in Exp_c^c$, if p is the precondition of the body of $m \neq \mathsf{start}$ in c, then*

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

If m=start, then p is $pre(stm^c_{\mathsf{run}})$, and id'=$(\mathsf{this}', 0)$ replaces id'=$callee(\mathsf{id})$.

– *For all statements $\langle rexp; \vec{y} := \vec{e} \rangle$ occurring 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, then*

$$\models_{\mathcal{L}} p' \wedge pre(\vec{y} := \vec{e}) \wedge e_0 = \mathsf{this} = \mathsf{this}' \wedge \mathsf{id} = callee(\mathsf{id}') \rightarrow p'[\vec{e}/\vec{y}]. \quad (10)$$

Note that we have to replace the temporary variables 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 invocation and object creation. Since different objects may be involved, it is formulated in the global assertion language. Besides defining verification conditions that ensure the invariance of the global invariant over critical sections, it specifies conditions under which properties, whose evaluation depend on communicated values, are satisfied. Those properties are given by the preconditions of method bodies, and by the postconditions of receive and object creation statements.

The global invariant GI, which describes invariant properties of a program, is not allowed to refer to instance variables whose values are changed outside critical sections. Consequently, it will be automatically invariant over the execution of statements outside critical sections. For the critical sections themselves, however, the invariance must be shown as part of the cooperation test. 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 assertion in the object at the receiving end of the communication holds, immediately after the values has been communicated. For method call, the receiver is the callee and conversely, the caller when returning from a method. When calling a method, the precondition of the method body must hold after passing the parameters (cf. Equation (11)). 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, i.e., $z'.\mathsf{lock} = \mathsf{nil}$, or the lock has been acquired in the call chain of the executing thread, i.e., $z'.\mathsf{lock} \leq \mathsf{id}$, where id refers to the identity of the caller. Equation (12) works similarly, where the postcondition of the receive-statement is 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 local variables of the caller.

The global invariant must also be preserved under the communication statements as formalized in Equations (13) and (14), which use the same antecedents as the corresponding conditions for the local assertions. A difference between the treatment of the local assertions and the global invariant is, that the latter does not necessarily hold immediately after the communication statement, 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.

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 $e_0 \in Exp_c^{c'}$ and m a synchronized method in class c' with body $\langle \vec{y}_2 := \vec{e}_2\rangle; stm; \langle \mathsf{return}\, e_{ret}; \vec{y}_3 := \vec{e}_3\rangle$ and formal parameter list $\vec{u}$, then*

$$\models_{\mathcal{G}} GI \wedge pre(e_0.m(\vec{e}))[z/\mathsf{this}] \wedge I_{c'}[z'/\mathsf{this}] \wedge e_0[z/\mathsf{this}] = z' \wedge (z'.\mathsf{lock} = \mathsf{nil} \vee z'.\mathsf{lock} \leq \mathsf{id}) \rightarrow pre'(body)[z', \vec{E}/\mathsf{this}, \vec{u}] \tag{11}$$

$$\models_{\mathcal{G}} GI \wedge pre(\mathsf{receive}\, v)[z/\mathsf{this}] \wedge \; pre'(\mathsf{return}\, e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge e_0[z/\mathsf{this}] = z' \rightarrow post(\mathsf{receive}\, v)[z, E_{ret}/\mathsf{this}, v] \tag{12}$$

$$\models_{\mathcal{G}} GI \wedge pre(e_0.m(\vec{e}))[z/\mathsf{this}] \wedge I_{c'}[z'/\mathsf{this}] \wedge e_0[z/\mathsf{this}] = z' \wedge (z'.\mathsf{lock} = \mathsf{nil} \vee z'.\mathsf{lock} \leq \mathsf{id}) \rightarrow GI[\vec{E}_2/z'.\vec{y}_2][\vec{E}_1/z.\vec{y}_1] \tag{13}$$

$$\models_{\mathcal{G}} GI \wedge pre(\mathsf{receive}\, v)[z/\mathsf{this}] \wedge \; pre'(\mathsf{return}\, e_{ret})[z', \vec{E}/\mathsf{this}, \vec{u}] \wedge e_0[z/\mathsf{this}] = z' \rightarrow GI[\vec{E}_4/z.\vec{y}_4][\vec{E}_3/z'.\vec{y}_3], \tag{14}$$

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 $pre'(body)$ is the precondition of the method body with every local variable, i.e., temporary variable except the formal formal parameters, of* *class, boolean, or integer type, is replaced by* nil, false*, or* 0*, respectively, and correspondingly for composed types; the assertion* $pre'(\textsf{return}\, e_{ret})$ *denotes the precondition of the given statement with every local variable replaced by a fresh logical one. Furthermore,* $\vec{E}_1 = \vec{e}_1[z/\textsf{this}]$, $\vec{E}_j = \vec{e}_j[z', \vec{E}/\textsf{this}, \vec{u}]$ *for* $j = 2, 3$, $\vec{E}_4 = \vec{e}_4[z, E_{ret}/\textsf{this}, v]$*, where* $\vec{E} = \vec{e}[z/\textsf{this}]$*. The global expression* E_{ret} *denotes* $e_{ret}[z', \vec{E}/\textsf{this}, \vec{u}]$*, with every local variable replaced by a fresh logical one in correspondence to the replacement in* $pre'(\textsf{return}\, e_{ret})$*. For the invocation of* non-synchronized methods, *the antecedent* $z'.\textsf{lock} = \textsf{nil} \vee z'.\textsf{lock} \leq \textsf{id}$ *is dropped. The verification conditions for methods without return value are analogous. For invocations of* start*-methods, only (11) and (13) apply, where body denotes* $stm^{c'}_{\textsf{run}}$*; both equations have the additional antecedent* $z'.\textsf{started} = \textsf{false}$.

The substitution of $\vec{u}$ by $\vec{E}$ in the condition $pre'(body)[z', \vec{E}/\textsf{this}, \vec{u}]$ reflects the parameter-passing mechanism. 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 Equations (12) and (14). Furthermore, the cooperation test needs to handle critical sections containing object creation statements taking care of the preservation of the global invariant, the postcondition of the new-statement, and the new object's class invariant. 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 $LVar^{\textsf{list Object}}$ 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 $u \neq \textsf{nil} \wedge u \notin z' \wedge (\forall v(v \in z' \vee v = u)) \wedge Init(u)$, where $z' \in LVar^{\textsf{list Object}}$, and $Init(u)$ is as defined in Section 4.2. To have the quantification 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. So let P be a global assertion and $z' \in LVar^{\textsf{list Object}}$. Then $P \downarrow z'$ is the global assertion P with all quantifications $\exists z(P')$ replaced by $\exists z(z \in z' \wedge P')$, when z is of type c or Object, by $\exists z(z \sqsubseteq z' \wedge P')$, when z is of type list c or list Object, and correspondingly for composed types. We use $\exists_\perp z(P)$ as a shortcut for $P[\textsf{nil}/z] \vee \exists z(P)$ to express that the predicate P holds for the old value of u prior to the creation of the new object.

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

$$\models_G \exists z'\Big(Fresh(z', u) \wedge (GI \wedge \exists_\perp u(pre(u := \textsf{new}^c)[z/\textsf{this}])) \downarrow z'\Big) \qquad (15)$$
$$\rightarrow (post(u := \textsf{new}^c)[z/\textsf{this}] \wedge I_c[u/\textsf{this}] \wedge GI[\vec{E}/z.\vec{y}]),$$

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

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].

4.3 Example

In this section we sketch an application of the proof method on a simple toy example. Let c be a class, with instance variable x, defined by

```
c{  nsync m(){ this.start(); this.dec(); return }
    nsync run(){ this.dec(); return }
    sync dec(){ if (x > 0) then x := x − 1; return } }
```

Now let *prog* be a program containing class c together with a main class. The non-synchronized main statement of the program just creates a new object of class c and invokes its method m which in turn starts the thread of the current object; both m and run invoke the synchronized method dec decreasing x by 1. We show that $x \geq 0$ for each instance of type c. The invariance of this predicate is ensured by the fact that, being synchronized, dec cannot be executed simultaneously by two different threads, and since it does not contain method invocations, it cannot be called recursively. The following proof outline for class c expresses the above properties using the thread identification mechanism, where the class invariant I of class c is defined by $x \geq 0$:

```
c{
  nsync m(id){                    {id=(nil, 1) ∧ I}
    ⟨this.start((this, 0))⟩;      {id=(nil, 1) ∧ I}
    ⟨this.dec(callee(id))⟩;       {I}
    ⟨receive⟩;                    {I}
    ⟨return⟩                      {I}   }
  nsync run(id){                  {id=(this, 0) ∧ I}
    ⟨started := true⟩;            {id=(this, 0) ∧ I}
    ⟨this.dec(callee(id))⟩;       {I}
    ⟨receive⟩;                    {I}
    ⟨return⟩                      {I}   }
  sync dec(id){                   {(id=(nil, 2) ∨ id=(this, 1)) ∧ (lock=nil ∨ lock<id) ∧ I}
    ⟨lock := getlock(lock, id)⟩;  {(id=(nil, 2) ∨ id=(this, 1)) ∧ lock≤id ∧ I}
    if (x > 0) then               {(id=(nil, 2) ∨ id=(this, 1)) ∧ lock≤id ∧ x > 0 ∧ I}
    x := x − 1;                   {(id=(nil, 2) ∨ id=(this, 1)) ∧ lock≤id ∧ I}
    ⟨return;                      {(id=(nil, 2) ∨ id=(this, 1)) ∧ lock≤id ∧ I}
     lock := release(lock, id)⟩   {I}   }   }
```

Local correctness is straightforward. Interference freedom holds since the simultaneous execution of two different incarnations of dec is excluded logically by the additional information $\neg samethread(id, id')$ of Equation (7). Note that the precondition of the body of dec represents a non-interleaving point and that $(\mathsf{id}{=}(\mathsf{nil},2) \vee \mathsf{id}{=}(\mathsf{this},1)) \wedge \mathsf{lock}{\leq}\mathsf{id}$ together with $(\mathsf{id}'{=}(\mathsf{nil},2) \vee \mathsf{id}'{=}(\mathsf{this},1)) \wedge \mathsf{lock}{\leq}\mathsf{id}'$ implies $\mathsf{id} = \mathsf{id}'$. In the cooperation test we use the trivial global invariant true. Equation (11) of the cooperation test applied to the invocation of dec by the thread executing m requires that $z'.\mathsf{lock} = \mathsf{nil} \vee z'.\mathsf{lock} \leq \mathsf{id}$, where id denotes the identity of the caller, and z' identifies the callee object. Furthermore, the class invariant $I[z'/\mathsf{this}]$ and the precondition of the method invocation statement $\mathsf{id} = (\mathsf{nil},1) \wedge I[z/\mathsf{this}]$ are true, where z identifies the caller object. These conditions directly imply the precondition of the body of dec lifted to the global level and with the formal parameter id substituted by the actual one, expressing that the lock is owned by the executing thread of identity $(\mathsf{nil},2)$ or $(z',1)$.

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 $\mathsf{wait}()$, $\mathsf{notify}()$, and $\mathsf{notifyAll}()$, and the important feature of exception handling [19].

Most papers in the literature focus on *sequential* subsets of Java [27, 9, 25, 26, 10, 29, 1, 30, 31]. Formal semantics of *Java*, including multithreaded execution, and its virtual machine in terms of abstract state machines is given in [28]. A structural operational semantics of multithreaded *Java* can be found in [11].

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 [16, 22].

More in general, our future work focusses on the formalization of full-featured multithreading, inheritance, and polymorphic extensions involving behavioral subtyping [5].

References

1. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT '97*, LNCS 1214, pages 682–696. Springer, 1997.
2. E. Ábrahám-Mumm and F. de Boer. Proof-outlines for threads in Java. In Palamidessi [24].
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, 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 Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.
8. K. R. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.
9. R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*. PhD thesis, Universität Passau, 1991. See also Springer LNCS 562.
10. P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, Mar. 1995.
11. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [4].
12. 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.
13. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
14. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
15. C. Hankin, editor. *Proceedings ESOP '98*, LNCS 1381. Springer, 1998.
16. J. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Hankin [15].
17. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [18].
18. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.
19. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
20. H. Hussmann, editor. *Fundamental Approaches to Software Engineering*, LNCS 2029. Springer, 2001.
21. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
22. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.

23. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
24. C. Palamidessi, editor. Proceedings of *CONCUR 2000*, LNCS 1877. Springer, 2000.
25. A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* Technische Universität München, Jan. 1997. Habilitationsschrift.
26. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. Swierstra, editor, *Programming Languages and Systems*, LNCS 1576, pages 162–176. Springer, 1999.
27. B. Reus, R. Hennicker, and M. Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Hussmann [20], pages 300–316.
28. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine.* Springer, 2001.
29. 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.
30. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency – Practice and Experience*, 2001. To appear.
31. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. Submitted for publication, 2001.
32. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml.* Object Technology Series. Addison-Wesley, 1999.

A Structural Operational Semantics

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

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

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

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

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

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

$$\frac{\tau'(\mathsf{this}) = \beta \qquad \tau'' = \tau[u \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma(\beta),\tau'}]}{\langle T \dot\cup \{\xi \circ (\tau, u := e_0.m(\vec{e}); stm) \circ (\tau', \mathsf{return}\ e)\}, \sigma\rangle \longrightarrow \langle T \dot\cup \{\xi \circ (\tau'', stm)\}, \sigma\rangle} \ \textsc{Return}$$

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

We use $sync(c, m)$ to state that method m in class c is synchronized. For a set of thread configurations T and object α, the predicate $started(T, \alpha)$ holds iff there exists a $(\tau_0, stm_0) \ldots (\tau_n, stm_n) \in T$ such that $\tau_0(\mathsf{this}) = \alpha$. Furthermore, *isfree* is a predicate of the same type such that $isfree(T, \alpha)$ is true iff no stack in T contains any local configuration (τ, stm) with $\tau(\mathsf{this}) = \alpha$ and stm synchronized.

The semantic function $[\![\_]\!]_{\mathcal{E}} : (\Sigma_{inst} \times \Sigma_{\mathsf{loc}}) \rightarrow (Exp \rightarrow Val)$ evaluates instance variables and temporary variables in the context of an instance and a local state (σ_{inst}, τ) to $\sigma_{inst}(x)$ and $\tau(u)$, respectively. The value of this refers to the object in which the expression is evaluated, nil has the undefined value $\perp$. Finally, the evaluation of compound expressions are defined by homomorphic lifting.

For a local state τ, a local variable u of type t, and $value \in Val^t$, we denote by $\tau[u \mapsto value]$ the local state which assigns $value$ to u and agrees on the values of all other variables with τ. Correspondingly for global states, $\sigma(\alpha)[x \mapsto value]$ denotes the global state resulting from σ by assigning $value$ to the instance variable x of object α. We use these operators analogously for setting the values of a sequence of variables. By $\sigma[\alpha \mapsto \sigma_{inst}]$ for some $\alpha \in Val^c$, $\alpha \notin dom^c(\sigma)$ we denote the global state resulting from σ by extending the domain, i.e., the set of existing objects, by α and initializing its object state to σ_{inst}. The semantic update $\sigma_{inst}[x \mapsto v]$ of object states is defined analogously.