An Assertional Proof System for Multithreaded Java*

June 29, 2003

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

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

Abstract. In this paper we report on a tool-supported *assertional proof method* for a concurrent sublanguage of *Java*, covering shared-variable concurrency via instance variables, coordination via reentrant synchronization monitors, synchronous message passing, and dynamic thread creation. The verification method is formulated in terms of proof-outlines, where the assertions are layered into local ones specifying the behavior of a single instance, and global ones taking care of the connections between objects.

1 Introduction

The semantical foundations of *Java* [7] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [3, 14, 5]). Besides standard object-oriented features, *Java* integrates multithreading and monitor synchronization. The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sub-languages (see e.g. [10, 15, 13]). In this paper we present a tool-supported assertional proof system for *Java*'s monitor concept. The proof system generates verification conditions for $Java_{MT}$, a subset of *Java*, featuring dynamic object creation, method invocation, object references with aliasing, and, specifically, concurrency and *Java*'s monitor discipline.

The behavior of a $Java_{MT}$ program results from the concurrent execution of methods. To support a clean interface between internal and external object behavior, $Java_{MT}$ does not allow qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries. In order to capture program behavior in a modular way, the assertional logic and the proof system are formulated at two levels, a local and a global one. The local assertion language describes the internal object behavior. The global behavior, including the communication topology of the objects, is expressed in the global language. As in the Object Constraint Language (OCL) [16], properties of object-structures are described in terms of a navigation or dereferencing operator.

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

The assertional proof system for verifying safety properties of $Java_{MT}$ is formulated in terms of *proof outlines* [12], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [6, 8]. The satisfaction of the program properties specified by the assertions is guaranteed by the verification conditions of the proof system. The execution of a single method body in isolation is captured by standard *local correctness* conditions, using the local assertion language. Interference between concurrent method executions is covered by the *interference freedom test* [12, 11], formulated also in the local language. It has especially to accommodate for reentrant code and the specific synchronization mechanism. Possibly affecting more than one instance, communication and object creation is treated in the *cooperation test*, using the global language. The communication can take place within a single object or between different objects. As these cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules in [4] and in [11] for CSP.

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

In the following we informally describe the language and the proof system. For formal syntax, semantics, verification conditions, soundness, and completeness we refer to [1].

2 The programming language $Java_{MT}$

$Java_{MT}$ is a strongly typed sublanguage of *Java*. The abstract syntax is summarized in Table 1. Programs are collections of classes containing method declarations. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation.* As we focus on the concurrency aspects of *Java*, all classes in $Java_{MT}$ are thread classes in the sense of *Java*: They contain a start-method which can be invoked only once for each object. The method spawns a new thread of execution which begins its live in the run-method while the initiating thread continues its own execution.

As a mechanism of concurrency control, methods can be declared as *synchronized.* Each object has a *lock* which can be owned by at most one thread. Synchronized methods of an object can be invoked only by a thread that owns the lock of that object. Without the lock, a thread has to wait until the lock becomes free. The owner of an object's lock can recursively invoke several synchronized methods of that object, which corresponds to the notion of reentrant monitors. Besides synchronization, objects offer the methods wait, notify, and notifyAll as means to facilitate efficient thread coordination at the object boundary. All four methods are parameterless and without a return value. A thread owning the lock of an object can block itself and free the lock by invoking wait on the given object. The blocked thread can be reactivated by another thread via the object's notify-method; the reactivated thread must re-apply for the lock before it may continue its execution. The method notifyAll, finally, generalizes notify in that it informs all threads blocked on the object.

$$
\begin{array}{rcl}
exp & ::= & x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(exp, \ldots, exp) \\
exp_{ret} & ::= & \epsilon \mid exp \\
stm & ::= & x := exp \mid u := exp \mid u := \mathsf{new}^c \\
& \mid & u := exp.m(exp, \ldots, exp) \mid exp.m(exp, \ldots, exp) \\
& \mid & \epsilon \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm\ \mathsf{fi} \mid \mathsf{while}\ exp\ \mathsf{do}\ stm\ \mathsf{od} \ldots \\
modif & ::= & \mathsf{nsync} \mid \mathsf{sync} \\
meth & ::= & modif\, m(u, \ldots, u)\{\ stm; \mathsf{return}\ exp_{ret}\} \\
meth_{\mathsf{run}} & ::= & \mathsf{nsync\ run}()\{\ stm; \mathsf{return}\ \} \\
meth_{\mathsf{wait}} & ::= & \mathsf{nsync\ wait}()\{\ ?\mathsf{signal}; \mathsf{return}_{getlock}\ \} \\
meth_{\mathsf{notify}} & ::= & \mathsf{nsync\ notify}()\{\ !\mathsf{signal}\ ; \mathsf{return}\ \} \\
meth_{\mathsf{notifyAll}} & ::= & \mathsf{nsync\ notifyAll}()\{\ !\mathsf{signal\_all}; \mathsf{return}\ \} \\
meth_{predef} & ::= & meth_{\mathsf{start}}\ meth_{\mathsf{wait}}\ meth_{\mathsf{notify}}\ meth_{\mathsf{notifyAll}} \\
class & ::= & c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{predef}\} \\
class_{\mathsf{main}} & ::= & class \\
prog & ::= & \langle class \ldots class\ class_{\mathsf{main}}\rangle
\end{array}
$$

Table 1. Java$_{MT}$ syntax

$$
\begin{array}{ll}
e ::= z \mid x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(e, \ldots, e) & e \in LExp \\
p ::= e \mid \neg p \mid p \wedge p & \\
\quad \mid\ \exists z.\ p \mid \exists z \in e.\ p \mid \exists z \sqsubseteq e.\ p & p \in LAss \\
 & \\
E ::= z \mid \mathsf{null} \mid \mathsf{f}(E, \ldots, E) \mid E.x & E \in GExp \\
P ::= E \mid \neg P \mid P \wedge P \mid \exists z.\ P & P \in GAss
\end{array}
$$

Table 2. Syntax of assertions

3 The assertion language

The assertion logic consists of a *local* and a *global* sublanguage (cf. Table 2). The *local* assertion language is used to annotate methods 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 communication structure. In the assertion language we add the type Object as the supertype of all classes, and we introduce *logical variables* z different from all program variables. Logical variables are used for quantification and as free variables to represent local variables in the global assertion language.

Assertions are built using the usual constructs from predicate logic, where only the difference between the local and the global level deserves mention which concerns the form of quantification. In the local language, unrestricted quantification $\exists z.p$ is solely allowed for integer and boolean domains, but not for reference types, as for those types the range of quantification dynamically depends on the *global* state, something which is not available on the local level. This does not mean one cannot speak about the existence of objects on the local level at all as long as one avoids reference to the global state, To be able to do so, the local language syntactically contains two restricted forms of quantifications where the range of quantification is explicitely mentionend in the formula. The existential formula $\exists z{\in}e.p$ asserts that p holds for *some object from* e, where the expression e evaluates locally to a list of objects. Simularly for i$\exists z{\sqsubseteq}e.p$, where $\sqsubseteq$ is mean to denote all subsequences of e. Global assertions are evaluated in the context of a global state. Thus, unrestricted quantification is allowed for all types and ranges over the set of *existing* values. Qualified references may be used in the global language only. We use $\models_{\mathcal{L}} p$ and $\models_{\mathcal{G}} P$ to state that the assertions p and P are satisfied in all contexts.

4 The proof system

The proof system has to accommodate for dynamic object creation, shared-variable concurrency, aliasing, method invocation, synchronization, and, especially, reentrant monitors and thread coordination. The following section shows how to augment and annotate programs resulting in proof outlines, before Section 4.2 describes the proof method.

4.1 Proof outlines

For a complete proof system it is necessary that the transition semantics of $Java_{MT}$ can be encoded in the assertion language. As the assertion language reasons about the local and global states, we have to *augment* the program with fresh auxiliary variables to represent information about the control points and stack structures within the local and global states. An augmentation extends a program by assignments to auxiliary variables, which we call *observations*. The additional assignments are executed atomically as multiple assignments $\vec{y} := \vec{e}$. Observations are, in general, "attached" to statements they observe in a non-interleavable manner. This is syntactically represented using $\langle \vec{y} := \vec{e} \rangle^{1}$ which attaches the observation to the preceding statement. Though the reception of a method call is not represented by any statements, we use $\langle \vec{y} := \vec{e} \rangle^{1}$ at the beginning of method bodies also to denote the corresponding observation of the callee. As method calls $u := e_0.m(\vec{e})$ conceptually consist of two steps —handing over the parameters and reception of the result being stored in u— we need an additional form to observe atomically the reception of the return value. This form is represented as $\langle \vec{y} := \vec{e} \rangle^{2}$. A stand-alone observation not attached to any statement is written as $\langle \vec{y} := \vec{e} \rangle$; it can be inserted at any point in the program. Points between a statement and its observation are no *control points*, since they are executed in a single computation step; we call them *auxiliary points*.

The augmentation does not influence the control flow of the program but enforce a particular scheduling policy. An assignment statement and its observation are executed simultaneously. Object creation and its observation are executed in a single computation step, in this order. For method call, communication, sender, and receiver observations are executed in a single computation step, in this order.

In the following we introduce some built-in auxiliary variables used to formulate the verification conditions. An important point of the proof system to achieve completeness is the identification of communicating objects and threads. We identify a thread by the object in which it has begun its execution. We use the type Thread thus as abbreviation for the type Object. This identification is unique, since an object's start-method can be invoked only once. During a method call, the callee thread receives its own identity as an auxiliary formal parameter thread.

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

To capture mutual exclusion and the monitor discipline, the instance variable lock of type $\mathsf{Thread} \times \mathsf{Int}$ stores the identity of the thread who owns the lock, if any, together with the number of synchronized calls in the call chain. Its initial value $(null, 0)$ indicates that the lock is free. The instance variables wait and notified of type $\mathsf{list}(\mathsf{Thread} \times \mathsf{Int})$ store the threads waiting at the monitor, respectively those having been notified. Besides the thread identity, the number of synchronized calls is stored. In other words, these variables remember the old lock-value prior to suspension which is restored when the thread becomes active again. The boolean instance variable started, finally, remembers whether the object's start-method has already been invoked.

To specify invariant properties of the system, the augmented programs are *annotated* by attaching pre- and postconditions to all program statements. We use the triple notation $\{p\}\, stm\, \{q\}$ and write $pre(stm)$ and $post(stm)$ to refer to the pre- and the post-condition of a statement. For assertions at auxiliary points we use the following notation: The annotation

$$\{p_0\}\, u := e_0.m(\vec{e}); \quad \{p_1\}^{1} \langle \vec{y}_1 := \vec{e}_1; \rangle^{1} \quad \{p_2\}^{ret} \quad \{p_3\}^{2} \langle \vec{y}_4 := \vec{e}_4; \rangle^{2} \quad \{p_4\}$$

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

$$\{p_1\}^{1} \ \langle \vec{y}_2 := \vec{e}_2; \rangle^{1} \ \{p_2\} \quad stm; \quad \{p_3\} \ \ \mathsf{return}\, e; \ \{p_4\}^{1} \ \langle \vec{y}_3 := \vec{e}_3; \rangle^{1} \ \{p_5\}$$

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

Besides pre- and postconditions, for each class c, the annotation defines a local assertion I_c called *class invariant*, which may refer only to the instance variables of c, and which expresses invariant properties of instances of the class, i.e., properties required to hold for all instances of the class at each point during the execution.[3] We require that for each method of a class, the class invariant is the precondition of the method body. Finally, a global assertion GI called the *global invariant* specifies properties of communication between objects. As such, it should be invariant under object-internal computation. For that reason, we require that for all qualified references $E.x$ in GI with E of type c, all assignments to x in class c occur in the observations of communication or object creation. We require that in the annotation no free logical variables occur. An augmented and annotated program is called a *proof outline* or *asserted program.*

4.2 Verification conditions

The proof system formalizes a number of *verification conditions* which inductively ensure that for each reachable configuration, the local assertions as well as the global and the class invariants hold. The conditions are grouped, as usual, into initial conditions, and for the inductive step into local correctness and tests for interference freedom and cooperation. Here we only informally describe the latter two.

4.2.1 The interference freedom test Invariance of local assertions under computation steps in which they are not involved is assured by the proof obligations of the *interference freedom test.* Since Java$_{MT}$ does not support qualified references to instance variables, we only have to deal with invariance under execution within the *same* object. Due to restrictions on the programming language, communication and object creation do not change the instance states of the executing objects. Implicitly, we only have to cover invariance of assertions annotating control points over assignments, including observations. Preconditions of observations do not have to be shown invariant, since the observations are executed together with the statements they observe in a single computation step, i.e., the preconditions of observations describe points at which no interleaving can take place. To distinguish local variables of the assertion from those of the assignment we rename the local variables in the assertion. We use primed assertions p' to denote a given assertion p with every local variable u replaced by a fresh one u', and correspondingly for expressions.

[3] The notion of class invariant commonly used for sequential object-oriented languages differs from our notion: In a sequential setting, it is sufficient that the class invariant holds initially and is preserved by whole method calls, but not necessarily in between. This weaker notion is adequate in a sequential language, whose semantics is based on the input/output behavior of programs, but not in a setting with shared variable concurrency, where also intermediate states are relevant for the behavior.

Generally, the interference freedom test defines for each assertion p at a control point and each assignment $\vec{y} := \vec{e}$ in the same class c a condition of the form $\models_{\mathcal{L}} \mathit{pre}(\vec{y} := \vec{e}) \wedge p' \wedge I_c \wedge \mathsf{interleavable} \rightarrow p'[\vec{e}/\vec{y}]$, where the effect of assignments is expressed by the usual capture-avoiding substitution. The additional antecedent $\mathsf{interleavable}$ is explained in the following: An assertion p has to be invariant under an assignment $\vec{y} := \vec{e}$ only if the local configuration described by the assertion is not active in the computation step executing the assignment. If p and $\vec{y} := \vec{e}$ belong to the *same* thread, i.e., $\mathsf{thread} = \mathsf{thread}'$, then the only assertions endangered are those at control points waiting for a return value earlier in the current execution stack. Invariance of a local configuration under its own execution, however, need not be considered and is excluded by requiring $\mathsf{conf} \neq \mathsf{conf}'$. Interference with the *matching* return statement in a self-communication need also not be considered, because communicating partners execute simultaneously; in this case we additionally require $\mathsf{this} \neq \mathsf{caller\_obj} \vee \mathsf{conf}' \neq \mathsf{caller\_conf}$, where $\mathsf{caller\_obj}$ is the first and $\mathsf{caller\_conf}$ the second component of caller. If the assertion and the assignment belong to *different* threads, interference freedom must be shown in any case except for the self-invocation of the start-method.

Example 1. Assume an assignment $\{p\}\, \mathit{stm}$ in a method m of c, and an assertion q at a control point, which is not waiting for return, in the same method such that both p and q imply $\mathsf{thread} = \mathsf{this}$, i.e., the method is executed only by the thread of the object to which it belongs. Then, for invariance of q under the assignment stm, the antecedent of the interference freedom condition implies $p \wedge q' \wedge \mathsf{interleavable}(q, \mathit{stm})$. From $p \wedge q'$ we conclude $\mathsf{thread} = \mathsf{thread}'$, and thus by the definition of $\mathsf{interleavable}(q, \mathit{stm})$ the assertion q should be at a control point waiting for return, which is not the case, and thus the antecedent of the condition leads to a contradiction.

Example 2. This example shows how we can use the fact, that at most one thread can own the lock of an object, to show mutual exclusion. We use the assertion $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$ for $\mathsf{thread} \neq \mathsf{null} \wedge \mathsf{thread}(\mathsf{lock}) = \mathsf{thread}$, where $\mathit{thread}(\mathit{lock})$ is the first component of the lock value. Let furthermore $\mathsf{free\_for}(\mathsf{thread}, \mathsf{lock})$ be $\mathsf{thread} \neq \mathsf{null} \wedge (\mathsf{owns}(\mathsf{thread}, \mathsf{lock}) \vee \mathsf{lock} = \mathsf{free})$.

Let q, given by $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$, be an assertion in a class and let $\{p\}^{1}\, \langle \vec{y} := \vec{e} \rangle^{1}$ with $p \stackrel{def}{=} \mathsf{free\_for}(\mathsf{thread}, \mathsf{lock})$ be the callee observation at the beginning of a synchronized method in the same class. Note that built-in augmentation in $\vec{y} := \vec{e}$ changes the lock value setting the lock owner to the executing thread. Invariance of q under the observation $\vec{y} := \vec{e}$ is assured by the interference freedom condition $\models_{\mathcal{L}} p \wedge q' \wedge \mathsf{interleavable}(q, \vec{y} := \vec{e}) \rightarrow q'[\vec{e}/\vec{y}]$. The assertions p and q' imply $\mathsf{thread} = \mathsf{thread}'$. After the execution of the built-in augmentation setting the lock owner to the identity of the executing thread in $\vec{y} := \vec{e}$ we have $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$, i.e., $\mathsf{owns}(\mathsf{thread}', \mathsf{lock})$, which was to show.

4.2.2 The cooperation test Whereas the interference freedom test assures invariance of assertions under steps in which they are not involved, the *cooperation test* deals with inductivity for communicating partners, assuring that the global invariant, the preconditions, and the class invariants of the involved statements imply their postconditions after the joint step. Additionally, the preconditions of the corresponding observations must hold immediately after communication. The global invariant expresses global properties of the auxiliary instance variables which can be changed by observations of communication, only. Thus the global invariant is automatically preserved by the execution of non-communicating statements. For communication and object creation, however, the invariance must be shown as part of the cooperation test.

Here we describe only the cooperation test for method invocation. Since different objects may be involved, the cooperation test is formulated in the global assertion language. Local properties are expressed in the global language via substitution. To avoid name clashes between local variables of the partners, we rename those of the callee.

Let z and z' be logical variables representing the caller, respectively the callee object in a method call. We assume the global invariant, the class invariants of the communicating partners, and the preconditions of the communicating statements to hold prior to communication. For method invocation, the precondition of the callee is its class invariant. That the assertions indeed represent communicating partners and that the communication is enabled is captured in the additional antecedent comm. In case of a synchronized method invocation, the lock of the callee object has to be free or owned by the caller, which is expressed by $z'.\mathsf{lock} = (\mathsf{null}, 0) \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$, where thread is the caller-thread and $thread(\alpha, n) = \alpha$. For the invocation of the monitor methods the executing thread must hold the lock. An additional predicate $e'_0[z/\mathsf{this}] = z'$ in the condition of a method call $e_0.m(\vec{e})$ states, that z' is indeed the callee object where $e'_0[z/\mathsf{this}]$ is the local assertion e'_0 expressed in the global language. The effect of communication and its observations are expressed by substitutions.

The verification conditions assure, that under the above assumptions, the preconditions of the corresponding caller and callee observations hold immediately after communication, and that the postconditions of the observations hold after their execution. The effects of communication and observation are expressed by substitutions.

For returning from a method we have similar conditions. Returning from the wait-method assumes that the thread has been notified and that the callee's lock is free. Due to the built-in auxiliary variables, method invocation hands over the "return address". Furthermore, due to some restrictions, the values of formal and actual parameters do not change during method execution. Therefore, they can be used in the return conditions to identify partners being in caller-callee relationship.

Example 3. This example shows how we can prove properties of parameter passing. Let $\{p\}\, e_0.m(v)$, with p given by $v > 0$, be a statement in a class c with e_0 of type c', and let method $m(u)$ of c' have the body $\{q\}\, stm;$ return where q is $u > 0$. Inductivity of the proof outline requires that if p is valid prior to the call (besides the global and class invariants), then the method's precondition q is satisfied after the invocation. Omitting irrelevant details, the cooperation test requires $\models_{\mathcal{G}} p[z/\mathsf{this}] \to q'[z'/\mathsf{this}][v/u']$, which expands to $\models_{\mathcal{G}} v > 0 \to (u' > 0)[v/u']$.

Example 4. The following example demonstrates how we can express dependencies between instance states in the global invariant, and use this information in the cooperation test.

Let $\{p\}\, e_0.m()$, with p given by $x > 0 \wedge e_0 = o$, be a statement in a class c with e_0 of type c' and with x an integer instance variable and o an instance variable of type c', and let method m of c' have the body $\{q\}\, stm; \mathsf{return}$ where q is $y > 0$ and y an integer instance variable. Let furthermore z be a logical variable of type c, and let the global invariant be given by $\forall z.\, (z \neq \mathsf{null} \wedge z.o \neq \mathsf{null} \wedge z.x > 0) \to z.o.y > 0$. Inductivity requires that if p and the global invariant are valid prior to the call, then the method's precondition q is satisfied after the invocation (again, we omit irrelevant details). The cooperation test requires $\models_{\mathcal{G}} GI \wedge p[z/\mathsf{this}] \wedge \mathsf{comm} \to q'[z'/\mathsf{this}]$ which expands to

$$\begin{aligned}\models_{\mathcal{G}} (\ &(\forall z.\, (z \neq \mathsf{null} \wedge z.o \neq \mathsf{null} \wedge z.x > 0) \to z.o.y > 0) \wedge \\ &(z.x > 0 \wedge e_0[z/\mathsf{this}] = z.o) \wedge (e_0[z/\mathsf{this}] = z' \wedge z \neq \mathsf{null} \wedge z' \neq \mathsf{null})\ \ ) \\ &\to z'.y > 0\end{aligned}$$

Instantiating the quantification by z, the antecedent implies $z.o.y > 0 \wedge z' = z.o$, i.e., $z'.y > 0$.

Example 5. This example illustrates how the cooperation test handles observations of communication. Let $\{\neg b\}\, \mathsf{this}.m()\{b\}^{ret}$ be a statement in a class c with auxiliary instance variable b and let m of c have the body $\{\neg b\}^{1}\, \langle b := \mathsf{true}\rangle^{1}\, \{b\}\, stm; \mathsf{return}$. The cooperation test assures inductivity for the precondition of the observation. We have to show $\models_{\mathcal{G}} (\neg z.b \wedge \mathsf{comm}) \to \neg z'.b$, i.e., as it is a self-call, $\models_{\mathcal{G}} (\neg z.b \wedge z = z') \to \neg z'.b$, which is trivially satisfied. The cooperation test requires for the postconditions $\models_{\mathcal{G}} \mathsf{comm} \to (z.b \wedge z'.b)[\mathsf{true}/z'.b]$ which expands to $\models_{\mathcal{G}} z = z' \to (z.b \wedge z'.b)[\mathsf{true}/z'.b]$, whose validity is easy to see.

Example 6. Assume the invocation of a synchronized method m of a class c, where m of c has the body $\langle \vec{y} := \vec{e}\rangle^{1}\, \{\mathsf{thread}(\mathsf{lock}) = \mathsf{thread}\}\, stm; \mathsf{return}$. Note that the built-in augmentation in $\vec{y} := \vec{e}$ sets the lock owner. Omitting irrelevant details again, the cooperation test requires $\models_{\mathcal{G}} \mathsf{true} \to (\mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}')[\vec{e}/\vec{y}]$, which holds due to the built-in augmentation, setting the lock owner.

5 Conclusion

The contribution reports on an assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline. It builds on earlier work, especially [2]). Currently we are implementing a translator, which, given an asserted $Java_{MT}$ program, generates a number verification conditions to be handed-over to *PVS* for interactive verification. The underlying theory, the proof rules, and their soundness and completeness are presented in detail in [1].

As future work, we plan to extend $Java_{MT}$ by further constructs, like inheritance, subtyping, exceptions, etc. An extension of the semantics and the proof theory to detect deadlocks and termination is also of interest.

References

1. E. Ábrahám, F. de Boer, W.-P. de Roever, and M. Steffen. A Hoare logic for monitors in Java. Techical report TR-ST-03-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Apr. 2003.
2. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In M. Nielsen and U. H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, Apr. 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.
3. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. LNCS State-of-the-Art-Survey. Springer-Verlag, 1999.
4. 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.
5. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [3], pages 157–200.
6. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
7. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [9].
9. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.
10. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
11. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
12. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
13. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.
14. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.
15. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P.-A. Lindsay, editors, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2002.
16. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Object Technology Series. Addison-Wesley, 1999.