Inductive Proof Outlines for Monitors in Java*

September 1, 2003

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

[1] Christian-Albrechts-University Kiel, Germany
{eab,wpr,ms}@informatik.uni-kiel.de
[2] CWI Amsterdam, The Netherlands
F.S.de.Boer@cwi.nl

Abstract. The research concerning *Java*'s semantics and proof theory has mainly focussed on various aspects of sequential sub-languages. *Java*, however, integrates features of a class-based object-oriented language with the notion of multi-threading, where multiple threads can concurrently execute and exchange information via shared instance variables. Furthermore, each object can act as a monitor to assure mutual exclusion or to coordinate between threads.
In this paper we present a sound and relatively complete assertional proof system for *Java*'s monitor concept, which generates verification conditions for a concurrent sublanguage Java_{MT} of *Java*. This work extends previous results by incorporating *Java*'s monitor methods.
Keywords: OO, Java, multithreading, monitors, deductive verification, proof-outlines

1 Introduction

From its inception, *Java* [12] has attracted interest from the formal methods community: The widespread use of *Java* across platforms made the need for formal studies and verification support more urgent, the grown awareness and advances of formal methods for real-life languages made it more acceptable, and last but not least the array of non-trivial language features made it challenging.

Nevertheless, research concerning *Java*'s proof theory concentrated mainly on various aspects of *sequential* sub-languages (see e.g. [14, 23, 20]). In [5], we presented a sound and complete proof method for multithreaded *Java*, where threads can concurrently execute and exchange information via shared instance variables. In this paper we extend our results to deal with *Java*'s *monitor synchronization* mechanism: each object can act as a monitor to assure mutual exclusion or to coordinate between threads.

To support a clean interface between internal and external object behavior, we exclude 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

* 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).

in a modular way, the assertional logic and the proof system are formulated in 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) [24], properties of object-structures are described in terms of a navigation or dereferencing operator.

The assertional proof system for safety properties is formulated in terms of *proof outlines* [19], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [11, 13]. Invariance of the asserted program properties 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* [19, 16], 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 are treated in the *cooperation test*, using the global language. The theory presented here forms the theoretical foundation for a verification tool (*Verger*) which takes asserted *Java* programs as input and generates verification conditions for the *PVS* theorem prover as output; the use of the tool on a number of examples is reported in [4].

Overview The paper is organized as follows. Section 2 defines syntax and semantics of the programming language. After introducing the assertional logic in Section 3, the main Section 4 presents the proof system. Section 5 discusses related and future work.

2 The programming language $Java_{MT}$

Similar to *Java*, the language $Java_{MT}$ is strongly typed; besides class types c, it supports booleans and integers as primitive types, and products and lists as composite types. Each corresponding value domain is equipped with a standard set of operators with typical element f; we use $\alpha, \beta, \ldots$ as typical object identities.

2.1 Syntax

The $Java_{MT}$ syntax is summarized in Table 1. We notationally distinguish between *instance* variables $x \in IVar$, and stack-allocated *local* or *temporary* variables $u \in TVar$ of methods; we use y to denote variables from $Var = IVar \mathbin{\dot{\cup}} TVar$. Programs are collections of classes containing method declarations, where we use c and m for class names resp. method names, and $body_{m,c}$ to refer to the body of method m in class c. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation.* To increase readability of the proof outlines, we deviate from standard *Java* syntax, in that method invocation is syntactically split into a sending and a receiving statement.

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

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

Each class contains the predefined methods start, wait, notify, and notifyAll, and furthermore a user-defined run-method. The entry point of the program is given by the main-method of the program's main class. Invocation of the start-method, which can be done successfully only once, spawns a new thread of execution 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. The monitor methods wait, notify, and notifyAll facilitate efficient thread coordination at the object boundary. Their definitions use the auxiliary statements !signal, !signal_all, ?signal, and $\mathsf{return}_{getlock}$. A thread owning the lock of an object can block itself and free the lock by invoking wait on the 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, notifies all threads blocked on the object.

2.2 Operational semantics

A *local state* τ holds the values of the local variables of a method. A *local configuration* (α, τ, stm) of a thread executing within an object $\alpha \neq nil$ specifies, in addition to its local state τ, its point of execution represented by the statement *stm*. A *thread configuration* $\xi = (\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$ is a stack of local

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

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

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

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

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

Table 2. Operational semantics of the monitor methods

configurations, representing the call chain of the thread. We write $\xi \circ (\alpha, \tau, stm)$ for pushing a new local configuration onto the stack.

An object is characterized by its *instance state* σ_{inst} which assigns values to the self-reference this and to instance variables. The initial states τ_{init} and σ_{inst}^{init} assign initial values to all variables. A *global state* or *heap* σ maps each currently *existing* object, i.e., an object of its domain, to its instance state. A *global configuration* $\langle T, \sigma\rangle$ consists of a set T of thread configurations of the currently executing threads, together with a global state σ.

Expressions are evaluated with respect to an *instance local* state (σ_{inst}, τ); the base cases are $[\![u]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \tau(u)$, $[\![x]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \sigma_{inst}(x)$, and $[\![\mathsf{this}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \sigma_{inst}(\mathsf{this})$. The operational semantics is given as transitions between global configurations. A program's initial configuration $\langle T_0, \sigma_0\rangle$ satisfies $T_0 = \{(\alpha, \tau_{init}, body_{\mathsf{main},c})\}$ and $\sigma_0(\alpha) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \alpha]$, where the domain of σ_0 is $\{\alpha\}$ and α is an instance of the main class c.

For the semantics of assignment, object and thread creation, and ordinary method invocation we refer to [4]. The rules of Table 2 handle $Java_{MT}$'s monitor methods wait, notify, and notifyAll, offering a typical monitor synchronization mechanism. In all three cases the caller must own the lock of the callee object (cf. rule $\textsc{Call}_{monitor}$), as expressed by the predicate *owns*, defined below.

A thread can *block* itself on an object whose lock it owns by invoking the object's wait-method, thereby relinquishing the lock and placing itself into the object's wait set. Formally, the wait set $wait(T, \alpha)$ of an object is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?\mathsf{signal}; stm)$. After having "put itself on ice", the thread awaits notification by another thread which invokes the notify-method of the object. The !signal-statement in the notify-method thus reactivates a single thread waiting for notification on the given object (cf. rule SIGNAL). Analogous to the wait set, the notified set $\mathit{notified}(T, \alpha)$ of α is the set of all stacks in T with top element of the form $(\alpha, \tau, \mathsf{return}_{getlock})$, i.e., threads which have been notified and trying to get hold of the lock again. According to rule RETURN_{wait}, the receiver can continue after notification in executing $\mathsf{return}_{getlock}$ only if the lock is free. Note that the notifier does not hand over the lock to the one being notified but continues to own it. This behavior is known as *signal-and-continue* monitor discipline [6].

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

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

3 The assertion language

Java_{MT} does not allow qualified references to instance variables, to support a clean interface between between internal and external object behavior. To mirror this modularity, the assertion logic consists of a *local* and a *global* sublanguage. *Local* assertions are used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong. *Global* assertions describe a whole system of objects and their communication structure and will be used in the cooperation test. In the assertion language we add the type Object as the supertype of all classes, and we introduce *logical variables* $z \in \mathit{LVar}$ different from all program variables. Logical variables are used for quantification and as free variables to represent local variables in the global assertion language. Expressions and assertions are interpreted relative to a logical environment ω, assigning values to logical variables.

Assertions are built using the usual constructs from predicate logic (cf. Table 3), where only the difference between the local and the global level deserves mention which concerns the form of quantification. On 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 one cannot speak about on the local level. Nevertheless, one can assert the existence of objects on the local level, provided one is explicit about the domain of quantification, as in the

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

Table 3. Syntax of assertions

restricted quantifications $\exists z{\in}e.p$ or $\exists z{\sqsubseteq}e.p$. The local assertion $\exists z{\in}e.p$ states that there is a value z in the sequence e, for which p holds; $\exists z{\sqsubseteq}e.p$ states the existence of a subsequence. 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 $E.x$ may be used in the global language only. We write $[\![\_]\!]_{\mathcal{L}}$ and $[\![\_]\!]_{\mathcal{G}}$ for the semantic functions evaluating local and global assertions, and $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $[\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true$, and $\models_{\mathcal{L}} p$ if p holds in all contexts; we use analogously $\models_{\mathcal{G}}$ for global assertions.

To express a local property p in the global assertion language, we define the lifting substitution $p[z/\mathsf{this}]$ by simultaneously replacing in p all occurrences of this by z, and transforming all occurrences of instance variables x into qualified references $z.x$, where z is assumed not to occur in p. For notational convenience we view the local variables occurring in the global assertion $p[z/\mathsf{this}]$ as logical variables. Formally, these local variables are replaced by fresh logical variables. We will write $P(z)$ for $p[z/\mathsf{this}]$, and similarly for expressions.

For technical convenience, in this paper we formulate verification conditions as standard Hoare-triples $\{\varphi\}\, stm\, \{\psi\}$. The statements of these Hoare-triples may also contain assignments involving qualified references as given by the global assertion language. The formal semantics is given by means of a weakest precondition calculus [8, 4].

4 The proof system

The following section defines how to augment and annotate programs resulting in proof outlines, before Section 4.2 describes the proof method. The proof system accommodates for dynamic object creation, shared-variable concurrency, aliasing, method invocation, synchronization, and, especially, the reentrant monitors. Missing details can be found in [4].

4.1 Proof outlines

The definition of a relatively complete proof system requires that we can encode the transition semantics of $Java_{MT}$ in the assertion language. As the assertion language can reason about the local and global states, only, we have to *augment* programs with fresh auxiliary variables to represent information about

```
...⟨e0.m(this,conf,thread,e);⟩ ⟨receive u_ret;⟩ ... // meth. call

sync m(caller,thread,u) {
       ⟨conf:=counter, counter:=counter+1, lock:=inc(lock)⟩ ...
       ⟨return e_ret; lock:=dec(lock)⟩ }

nsync start(caller,thread,caller_thread) {
       ⟨conf:=counter, counter:=counter+1, started:=true⟩ ...
       ⟨return;⟩}
```

Fig. 1. Augmentation and annotation: Synchronized method call, start

the control points and stack structures within the local and global states: Assignments $y := e$ can be extended to multiple assignments $y, \vec{y}_{aux} := e, \vec{e}_{aux}$ by inserting additional assignments to distinct auxiliary variables $\vec{y}_{aux}$. Additional auxiliary assignments can be inserted at any control point. Observations $\vec{y} := \vec{e}$ of communication and object creation *stm* are enclosed in *bracketed sections* $\langle stm; \vec{y} := \vec{e}\rangle$. Method calls and the reception of the return value is observed by $\langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1\rangle$ and $\langle \mathsf{receive}\, u_{ret}; \vec{y}_4 := \vec{e}_4\rangle$. Similarly for the callee, methods are extended to $m(\vec{u})\{\langle \vec{y}_2 := \vec{e}_2\rangle; stm; \langle \mathsf{return}\, u_{ret}; \vec{y}_3 := \vec{e}_3\rangle\}$. To be uniform, we will sometimes write $\langle ?m(\vec{u}); \vec{y} := \vec{e}\rangle$ to indicate that the assignment observes the reception of a method call. We require that the caller observation in a self-communication does not change the values of instance variables.

Bracketed sections do not influence the control flow of the original program but enforce a particular scheduling policy: Communication, sender, and receiver observations are executed in this order in a single computation step, i.e., they cannot be interleaved with other threads. Points which can be interleaved we call control points. At points between communication and its observation in bracketed sections, or at the beginning and at the end of a methods, no interleaving can take place; we call them *auxiliary points*.

Next we introduce a few auxiliary variables, built into all augmentations and used in the verification conditions. Their values are changed only as described in the following. The updating of the specific auxiliary variables for ordinary synchronized methods and for the start-method is illustrated in Figure 1. Non-synchronized methods are treated analogously except that they do not change the lock value; the start-method additionally handles thread creation.

Figure 2 shows the augmentation of the monitor methods. Note that we do not use the auxiliary statements !signal, !signal_all, and ?signal in the proof outlines and implement the monitore methods with the auxiliary variables wait and notified, instead, which represent the corresponding sets of the semantics.

An important point of the proof system is the identification of communicating objects and threads. We identify a thread by the object in which it has begun its execution. The identification is unique as an object's thread can be started only once. This identity is handed over from caller to callee as auxiliary formal parameter thread. For the start-method we use caller_thread as additional formal

```
nsync wait(caller,thread) {
      ⟨conf:=counter, counter:=counter+1,
       wait:=wait ∪ {lock}, lock:=free;⟩
      ⟨return_getlock;  lock:=get(notified,thread),
       notified:=notified \ get(notified,thread);⟩ }

nsync notify(caller,thread) {
      ⟨conf:=counter, counter:=counter+1;⟩
      wait,notified:=notify(wait,notified);
      ⟨return;⟩ }

nsync notifyAll(caller,thread) {
      ⟨conf:=counter, counter:=counter+1;⟩
      notified:=notified ∪ wait, wait:=∅;
      ⟨return;⟩ }
```

Fig. 2. Augmentation and annotation: Signaling

parameter to store the identity of the caller thread. A local configuration, which represents the execution of a method, 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, given by the caller object together with the identity of the calling local configuration. The main-method is initially executed with the parameters $((nil, 0), \alpha_0)$, where α_0 is the initial object.

To capture mutual exclusion and the monitor discipline, the instance variable lock of type $\mathsf{Object} \times \mathsf{Int} + \mathsf{free}$, with initial value *free*, stores the identity of the thread that owns the lock, if any, together with the number of reentrant synchronized calls in the call chain. The semantics of incrementing the lock $[\![\mathsf{inc}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ is $(\tau(\mathsf{thread}), 0)$ for $\sigma_{inst}(\mathsf{lock}) = \mathit{free}$, and $(\alpha, n + 1)$ for $\sigma_{inst}(\mathsf{lock}) = (\alpha, n)$. Decrementing dec(lock) is done inversely. The instance variables wait and notified of type $2^{\mathsf{Object}\times\mathsf{Int}}$, with initial value $\emptyset$, are the analogues of the *wait*- and *notified*-sets of the semantics and store the threads waiting at the monitor, respectively, those having been notified. Besides the thread identity, the number of reentrant synchronized calls is stored. In other words, the wait and notified sets remember the old lock-value prior to suspension which is restored when the thread becomes active again. The old value is given by $get(notified, \alpha)$ for a thread α, whose uniqueness is assured by the semantics. The value $notify(wait, notified)$ is the pair of the given sets with one element, chosen nondeterministically, moved from the wait into the notified set; if the wait set is empty, it is the identity function. Note that in the augmented wait-method both the waiting and the notified status of the executing thread are represented by a single control point. The two statuses can be distinguished by the values of the wait and notified variables. The boolean instance variable started, finally, remembers whether the object's start-method has already been invoked. All auxiliary variables are initialized as usual, except the started-variable of the initial object which gets the value true.

To specify invariant properties of the system, the augmented programs are *annotated* by attaching local assertions to each control and auxiliary point. We use the standard triple notation $\{p\}\, stm\, \{q\}$ and write $pre(stm)$ and $post(stm)$ to refer to the pre- and the post-condition of a statement. Besides that, for each class c, a local assertion I_c called *class invariant* specifies invariant properties of instances of c in terms of its instance variables. We require that the pre- and postconditions of whole method bodies, describing the instance state of the callee object directly before method call and after returning, respectively, are given by the class invariant.[3] Finally, the *global invariant* $GI \in GAss$ 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 bracketed sections of communication or object creation. Note that the global invariant is not affected by the object-internal monitor signaling mechanism. We require that in the annotation no free logical variables occur. An augmented and annotated program $prog'$ is called a *proof outline*.

4.2 Verification conditions

The proof system formalizes a number of *verification conditions* which inductively ensure that for each *reachable* configuration the assertions at all current control points are satisfied, and that the global and the class invariants hold. The conditions are grouped, as usual, into initial conditions, local correctness, interference freedom, and a cooperation test. Note that the proof method is *modular* in that it allows for separate interference freedom and cooperation tests.

Arguing about two different local configurations makes it necessary to distinguish between their local variables, since these possibly have the same names; in such cases we rename the local variables in one of the local states. We use primed assertions p' to denote a given assertion p with every local variable u replaced by a fresh one u', and do so, correspondingly, for expressions.

4.2.1 Local correctness A proof outline is *locally correct*, if the properties of method instances as specified by the annotation are invariant under their own execution. For example, an assignment's precondition must imply its postcondition after execution. Besides that, invariance of the class invariant is required.

Definition 1 (Local correctness: Assignment). *A proof outline is* locally correct, *if for all assignments* $\{p_1\}\,\vec{y} := \vec{e}\,\{p_2\}$ *outside bracketed sections and all* c*,*

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

$$\models_{\mathcal{L}} p_1 \rightarrow I_c\ . \tag{2}$$

[3] Note that the callee configuration directly *after* invocation is described by the post-condition of the callee observation $\langle stm \rangle^{?call}$, which can be an arbitrary local assertion.

The conditions for loops and conditional statements are similar. Note that we have no local verification conditions for observations of communication and object creation. The postconditions of such statements express *assumptions* about the communicated values. They will be verified in the *cooperation test*.

4.2.2 The interference freedom test Invariance of local assertions under computation steps in which they are not involved is assured by 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. Affecting only local variables, communication and object creation do not change the instance states of the executing objects. Thus we only have to cover invariance of assertions annotating control points over assignments, including those of bracketed sections. Assertions at auxiliary points do not have to be shown invariant. So let p be an assertion at a control point and $\vec{y} := \vec{e}$ an assignment in the same class. In the following we will prime local variables of the assertion to distinguish them from those of the assignment.

The assertion p has to be invariant under the assignment only if the assignment is executed independently of the control point annotated by p:

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

The definition distinguishes two cases: 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. If they belong to the *same* thread, the only assertions endangered are those at control points waiting for a return value earlier in the thread's 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 neither needs to be considered, because communicating receive and return statements are executed simultaneously. This particular case is excluded by additionally requiring $\mathsf{caller} \neq (\mathsf{this}, \mathsf{conf}')$ for such assertion-assignment pairs.

Definition 2 (Interference freedom). *A proof outline is interference free, if for all classes c, assignments $\{p\}\vec{y} := \vec{e}$, and assertions q at control points in c,*

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

4.2.3 The cooperation test Whereas the interference freedom test assures invariance of assertions under steps in which they are not involved, the *cooperation test* deals with inductivity for communicating partners, assuring that the global invariant and the preconditions of the involved bracketed sections imply their postconditions after the joint step. Additionally, the assertions at the auxiliary points must hold immediately after communication. The global invariant may refer only to auxiliary instance variables which are changed in bracketed sections. Consequently, it is automatically invariant under the execution of non-communicating statements. For bracketed sections of communication and object creation, however, the invariance must be shown as part of the cooperation test.

In the following we define the cooperation test for method call. Since different objects may be involved, the cooperation test is formulated in the global assertion language. Local properties are expressed in the global language using the lifting substitution. To avoid name clashes between local variables of the partners, we rename those of the callee by priming them.

Let z and z' be logical variables representing the caller, respectively, the callee object. We assume the global invariant and the preconditions of the communicating statements to hold prior to communication. For method invocation, the precondition of the callee is its class invariant, as defined in the annotation. That the assertions indeed represent communicating partners and that the communication is enabled is captured in the assertion 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 $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$, where thread is the caller thread and $\mathit{thread}(\alpha, n) = \alpha$. For the invocation of the monitor methods the executing thread must hold the lock.

Let the function $\mathit{Init} : \mathit{Var} \to \mathit{Val}_{\mathit{nil}}$ assign to each variable its initial value.

Definition 3 (Cooperation test: Method invocation). *A proof outline satisfies the cooperation test for method invocation, if for all statements of the form* $\{p_1\}\langle e_0.m(\vec{e}); \{p_2\}\vec{y}_1 := \vec{e}_1\rangle\{p_3\}$ *in class* c *with* e_0 *of type* c'*, where method* m *of* c' *has body* $\{q_1\}\langle ?m(\vec{u}); \{q_2\}\vec{y}_2 := \vec{e}_2\rangle; \{q_3\}\mathit{stm}$ *and local variables* $\vec{v}$ *except the formal parameters:*

$$\models_{\mathcal{G}} \quad \begin{array}{c} \{\mathit{GI} \wedge P_1(z) \wedge Q'_1(z') \wedge \mathsf{comm}\} \\ \vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v}) \\ \{P_2(z) \wedge Q'_2(z')\} \end{array}$$

$$\models_{\mathcal{G}} \quad \begin{array}{c} \{\mathit{GI} \wedge P_1(z) \wedge Q'_1(z') \wedge \mathsf{comm}\} \\ \vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v}); \quad z.\vec{y}_1 := \vec{E}_1(z); \quad z'.\vec{y}'_2 := \vec{E}'_2(z') \\ \{\mathit{GI} \wedge P_3(z) \wedge Q'_3(z')\}, \end{array}$$

where $z \in \mathit{LVar}^c$ *and* $z' \in \mathit{LVar}^{c'}$ *are distinct and fresh;* comm *is given by* $E_0(z) = z' \wedge z \neq \mathsf{nil} \wedge z' \neq \mathsf{nil} \wedge \mathsf{synch}$*, and with* synch *defined as*

- true *for* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *non-synchronized,*
- $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *for* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *synchronized,*
- $\mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *for* $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$*, and*
- $\neg z'.\mathsf{started}$ *for* $m = \mathsf{start}$.

For $m = \mathsf{start}$*, the conditions must hold with additionally* synch *as* $z'.\mathsf{started}$*, where* $q_2 = q_3 = \mathsf{true}$ *and without* $\vec{u}', \vec{v}' := \vec{E}(z), \mathsf{Init}(\vec{v})$ *and* $z'.\vec{y}'_2 := \vec{E}'_2(z')$.

```
1        ...{owns(thread, lock)}
2  ⟨this.wait(this,conf,thread)⟩;
3        {¬owns(thread, lock) ∧ (this, conf) = getcaller(x, thread)}
4  ⟨receive⟩
5        {owns(thread, lock)} ...

7  {true}  nsync wait (caller,thread) {
8        {owns(thread, lock)}
9    ⟨conf:=counter, counter:=counter + 1, lock:=free,
10    wait:=wait ∪ {lock}, x:=x ∘ (thread,caller)⟩;
11       {¬owns(thread, lock) ∧ caller = getcaller(x, thread)}
12   ⟨return_getlock;
13       {lock = free ∧ caller = getcaller(x, thread) ∧ thread ∈ notified}
14    lock:=get(notified,thread),
15    notified:=notified \ get(notified,thread)⟩ } {true}
```

Fig. 3. Example

The first verification condition justifies the assertions at the auxiliary points after parameter passing, whereas the second verification condition justifies the postconditions of the bracketed sections and invariance of the global invariant.

For returning from a method and for object creation, there are similar conditions. Here we remark only that returning from the wait-method assumes that the thread has been notified and that the lock of the given object is free; in this case, synch is defined by $z'.\mathsf{lock} = \mathsf{free} \wedge \mathsf{thread}' \in z'.\mathsf{notified}$ (with $\alpha \in \mathit{notified}$ iff there is a $(\alpha, n) \in \mathit{notified}$ for some n).

The verification conditions presented above give rise to a *sound* and *relative complete* proof system. This means if all the verification conditions are valid, then at each reachable point all assertions hold, and conversely, if a program satisfies the requirements asserted in its proof outline, then this is indeed provable, i.e., then there exists a proof outline which can be shown to hold and which implies the given one. For the exact (standard) formulation of soundness and completeness and their proofs, we refer to the technical report [4].

4.2.4 Example We conclude this section with an example shown in Figure 3 which presents an annotation of the wait-method and its self-invocation. The annotation expresses basic properties of the lock ownership. The history of all invocations of the wait method is recorded in the auxiliary instance variable x of type $\mathsf{list}(\mathsf{Object} \times (\mathsf{Object} \times \mathsf{Int}))$. The operation $\mathit{getcaller}(x, \mathit{thread})$ returns *caller* where $(\mathit{thread}, \mathit{caller})$ is the last element in x with first component *thread*. We use $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$ as shorthand for $\mathsf{thread} = \mathsf{thread}(\mathsf{lock})$.

The proof uses the interference freedom test and the cooperation test; we start with the latter. The postconditions (3) and (9) of the method invocation at (2) and its observation at (8) is justified in the cooperation test as the postcondition of the parameter passing followed by the observation; note that (7) follows directly from (1) by parameter passing. Remember that we rename the local variables of the callee. The cooperation test for returning from the wait-method, i.e., the simultaneous execution of (4) and (10), additionally imports the information described by the assertion synch, namely that the lock is free and the executing thread is already notified.

To show interference freedom, we proceed by case analysis. For interference between different threads, assume $\mathsf{thread} \neq \mathsf{thread}'$. For example, the assertion at (1) is invariant under the execution of (8) by a different thread, because the lock can be owned by at most one thread. Formally, the conjunction of (1) and (7) gives us $\mathsf{owns}(\mathsf{thread}', \mathsf{lock}) \wedge \mathsf{owns}(\mathsf{thread}, \mathsf{lock})$ contradicting our assumption $\mathsf{thread} \neq \mathsf{thread}'$. The other cases follow similarly from the assumption.

More interesting is interference with respect to one thread, where we only need to consider invariance of the primed assertion at (3) over the assignments at (8) and (12). For (8), it suffices to observe that the assertions (3) and (7) lead to a contradiction with the assumption $\mathsf{thread} = \mathsf{thread}'$. The most interesting case is the invariance of the assertion (3) under the execution of (12). The information about the auxiliary variable x and the assumption that we deal with a single thread implies $\mathsf{caller} = (\mathsf{this}, \mathsf{conf}')$. This information contradicts the $\mathsf{interleavable}$ predicate of the interference freedom test, which excludes the situation of a matching return-receive statements in a self-communication.

5 Conclusion

This paper presents the first sound and complete assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline. It extends earlier work [5] by integrating *Java*'s wait and notify constructs into the proof system and by moving towards a more compositional formulation of the identification mechanism for threads, corresponding to the compositional semantics in [3]. Moreover, this particular extension shows how to incorporate further control mechanism by means of auxiliary variables which describe the corresponding flow of control. Based on the proof theory presented here, we have developed a front-end tool *Verger* which automatically extends programs with the built-in augmentation and generates the verification conditions for the theorem prover *PVS*. In [4], we explore the tool on a few examples, and present an extension of the proof theory to show absence of deadlock.

Related work As far as proof systems and verification support for object-oriented programs is concerned, research has mostly concentrated on *sequential* languages. For instance, the Loop-project [14, 17] develops methods and tools for the verification of sequential object-oriented languages, based on coalgebras and using proof *PVS* and *Isabelle/HOL*. Poetzsch-Heffter and Müller [20] develop a Hoare-style programming logic presented in sequent formulation for a sequential kernel of *Java*, featuring interfaces, subtyping, and inheritance. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer assisted soundness proof. The work [21] uses a modification of the *object constraint language* OCL as assertional language to annotate UML class diagrams and to generate proof conditions for *Java*-programs. In [23] a large subset of *JavaCard*, including exception handling, is formalized in *Isabelle/HOL*, and its soundness and completeness is shown within the theorem prover. [18] presents an exectuable formalization of a simplified JVM within the theorem prover ACL2. The work in [2] presents a Hoare-style proof-system for a sequential object-oriented calculus [1]. The language features heap-allocated objects (but no classes), side-effects and aliasing, and its type system supports subtyping. Furthermore, the language allows nested statically let-bound variables, which requires a more complex semantical treatment for variables based on closures, and ultimately renders their proof-system incomplete. Its assertion language is presented as an extension of the object calculus' language of type and analogously, the proof system extends the type derivation system. The close connection of types and specifications in the presentation is exploited in [22] for the generation of verification conditions. A survey about *monitors* in general, including proof-rules for various monitor semantics, can be found in [7].

The *extended static checking* approach [9, 15] occupies a middle-ground between verification and static analysis. Based on an intermediate guarded command language, the ESC-tool statically tries to detect (amongst other static properties) programming errors typical in a multithreaded setting such as synchronization errors and race conditions. In this direction, [10] presents a thread-modular checking approach based on assume-guarantee reasoning and implemented in the *Calvin*-tool.

Future work We plan to extend Java$_{MT}$ by further constructs, like exceptions, inheritance, and subtyping. To deal with subtyping on the logical level requires a notion of behavioral subtyping.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.
2. 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, Lille, France, Apr. 1997. Springer-Verlag. An extended version of this paper appeared as SRC Research Report 161 (September 1998).
3. E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. A compositional operational semantics for Java$_{MT}$. In N. Derschowitz, editor, *International Symposium on Verification (Theory and Practice)*, LNCS 2772. Springer-Verlag, 2003. To appear. A preliminary version appeared as Technical Report TR-ST-02-2, May 2002.
4. E. Ábrahám, F. S. 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.
5. E. Ábrahám-Mumm, F. S. 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*, LNCS 2303, 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.
6. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.
7. P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, Mar. 1995.
8. F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, LNCS 1578, pages 135–156. Springer-Verlag, 1999.
9. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Technical Note 159, Compaq, Dec. 1998.
10. C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In E. Brinksma and K. G. Larsen, editors, *Proceedings of CAV '02*, LNCS 2404, pages 180–194. Springer-Verlag, 2002.
11. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
12. J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
14. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.
15. K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. SRC Technical Note 1999-002, Compaq, May 1999.
16. G. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
17. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.
18. J. S. Moore and G. M. Porter. An executable formal Java Virtual Machine thread model. In *Proceedings of the 2001 JVM Usenix Symposium in Monterey, California*, 2001.
19. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
20. 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.
21. B. Reus, R. Hennicker, and M. Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, LNCS 2029, pages 300–316. Springer-Verlag, 2001.
22. F. Tang and M. Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects (extended abstract). In *Proceedings of the 9th International Workshop on Foundations of Object-Oriented Languages (FOOL'02)*, 2002. A longer version is available as LFCS technical report.
23. 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)*, LNCS 2391, pages 89–105. Springer-Verlag, 2002.
24. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml.* Object Technology Series. Addison-Wesley, 1999.