A Tool-supported Proof System for Multithreaded Java*

April 2, 2003

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] CWI Amsterdam, 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 includes shared-variable concurrency via instance variables, coordination via reentrant synchronization monitors, synchronous message passing, and dynamic thread creation.
To reason about safety properties of multithreaded *Java* programs, we introduce a tool-supported *assertional proof method* for $Java_{MT}$ ("*Multi-Threaded Java*"), a small sublanguage of *Java*, covering the mentioned concurrency issues as well as the object-based core of *Java*. The verification method is formulated in terms of proof-outlines, where the assertions are layered into local ones specifying the behavior of a single instance, and global ones taking care of the connections between objects. From the annotated program, a number of verification conditions are generated and handed over to the interactive theorem prover *PVS*. We illustrate the use of the proof system on an example.

1 Introduction

The semantical foundations of *Java* [20] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [6, 39, 14]). 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. [23, 42, 36]). In this paper we present a tool-supported assertional proof system for *Java*'s monitor concept, and illustrate its use on examples. 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

* 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).

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) [43], properties of object-structures are described in terms of a navigation or dereferencing operator.

The assertional proof system for verifying safety properties of *Java$_{MT}$* is formulated in terms of *proof outlines* [30], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [19, 21]. 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* [30, 28], 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 [11] and in [28] 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 [31] to verify the conditions.

Overview The paper is organized as follows. Section 2 described syntax and semantics of *Java$_{MT}$*. After introducing the assertional logic in Section 3, the main Section 4 presents the proof system and sketches the use of the verification tool. Section 5 discusses related and future work.

2 The programming language *Java$_{MT}$*

In this section we briefly describe the programming language *Java$_{MT}$*, a small sublanguage of *Java* concentrating on the concurrency aspects of the language. We start with the syntax in the following section, before in Section 2.2 we sketch the operational semantics. For a more thorough treatment, we refer to [4]

2.1 Syntax

Java$_{MT}$ is a strongly typed language; besides class types, it supports booleans and integers as primitive types, and pairs and lists as composite type. We will use $\alpha, \beta, \ldots$ as typical elements for class-typed values. Each domain is equipped with a standard set of operators.

Expressions are built from instance and local variables, the self-reference this, the empty reference null, and using a standard set of operators for each domain. We will use e as typical element for expressions. Conditional expressions can be defined for all types. In this paper we use the notation if e_1 then e_2 else e_3 fi; *Verger* supports *Java* syntax.

As *statements stm*, we allow assignments, object creation, method invocation, and standard control constructs like sequential composition, conditional statements, and iteration. We do not allow nested side-effect expressions. For clarity, we will sometimes write assignments in the form $y := e$ instead of the *Java*-syntax y = e.

A *method* definition specifies the return type of the method, its name, a list of formal parameters, and a method body, declaring the local variables of the method, and the statement that is to be executed if the method is invoked. All local variables must be declared at the beginning of the method; thus they all have their scope over the whole method body. To simplify the proof system we require that method bodies are terminated by a single return statement, giving back the control and possibly a return value. A *class* is defined by its name, its instance variables, and its methods. Its *instances*, i.e., *objects,* are dynamically created, and communicate via *method invocation. Programs,* finally, are collections of classes; a special main class defines the entry point of the program execution by its main-method.[3]

Java$_{MT}$ does not support inheritance, and consequently neither subtyping, overriding, and late-binding. The only class which may be extended is *Java*'s Thread class, allowing for a multithreaded flow of control. This class specifies a start-method, which can be invoked at most once for each of its instances, spawning a new thread of execution. The new thread starts to execute the user-defined run-method of the instance, while the initiating thread continues its own execution.

As a mechanism of concurrency control, methods can be declared as *synchronized* by the modifier keyword synchronized; In the sequel we also refer to statements in the body of a synchronized method as being 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. If the thread does not own the lock, it 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. 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

[3] *Java*$_{MT}$ does not allow static methods and variables. The only static method is the main-method, whose body may only create an instance of the main class and start its 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.

Besides the mentioned simplifications, we impose for technical reasons the following restrictions: To support a clean interface between internal and external object behavior, *Java$_{MT}$* does not allow qualified references $e.x$ to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object, only, but not across object boundaries. Furthermore, we require that none of the expressions $e_0, \ldots, e_n$ in method invocation statements $e_0.m(e_1, \ldots, e_n)$ contains instance variables, and that formal parameters are not assigned to. This restriction implies that during method execution the actual and formal parameter values are not changed. Finally, the result of an object creation or method invocation statement may not be assigned to instance variables. This restriction allows for a proof system with separated verification conditions for interference freedom and cooperation. It should be clear that it is possible to transform a program to adhere to this restrictions at the expense of additional local variables and thus new interleaving points.

Example 1 (Account). The following program implements a simple account, offering interfaces for deposit and withdraw. To assure that the balance x remains non-negative, the withdraw method is synchronized; implicitly, the balance does not get decreased between the evaluation of $x \geq i$ (written `x >= i` in concrete syntax) and the withdrawal. We will use this program to demonstrate the proof system: We show that for each class instance the balance x is always non-negative, under the assumption that the methods are called with positive parameters, only.

```
public class Account{
    private int x;

    private void change_balance(int i){
        x = x+i;
    }

    public void deposit(int i){
        change_balance(i);
    }

    public synchronized void withdraw(int i){
        if (x>=i) { change_balance(-i); }
    }
}
```

2.2 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 α specifies, in addition to its local state τ, its point of execution represented by the statement *stm*. A *thread configuration* ξ is a stack of local configurations $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$, representing the call chain of the thread. We write $\xi \circ (\alpha, \tau, \mathit{stm})$ for pushing a new local configuration onto the stack.

An object is characterized by its *instance state* σ_{inst} which assigns values to the self-reference this and to the instance variables. A *global state* σ stores for each currently *existing* object, i.e., an object belonging to the domain $\mathit{dom}(\sigma)$ of σ, its instance state. We denote by $\mathit{dom}^c(\sigma)$ set of all instances of class c from the domain of σ. 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 describing the currently existing objects.

Expressions are evaluated with respect to an *instance local* state $(\sigma_{\mathit{inst}}, \tau)$, where the instance state gives meaning to the instance variables $[\![x]\!]_{\mathcal{E}}^{\sigma_{\mathit{inst}},\tau} = \sigma_{\mathit{inst}}(x)$ and the self-reference $[\![\mathsf{this}]\!]_{\mathcal{E}}^{\sigma_{\mathit{inst}},\tau} = \sigma_{\mathit{inst}}(\mathsf{this})$, whereas the local state determines the values of the local variables $[\![u]\!]_{\mathcal{E}}^{\sigma_{\mathit{inst}},\tau} = \tau(u)$. The operational semantics is given as transition system between global configurations. In an initial configuration, the only existing object is an instance of the main class and the only thread executes the body of its run-method, with all variables are set to their initial values.

For the semantics of assignments, object and thread creation, and ordinary, i.e., non-monitor method invocation we refer to [4]. The rules of Table 1 handle *Java$_{MT}$*'s monitor methods wait, notify, and $\mathsf{notifyAll}$, offering a typical monitor synchronization mechanism. The bodies body_m of the predefined monitor methods m are specified by

$$\begin{array}{llll} \mathit{body}_{\mathsf{wait}} & = ?\mathsf{signal}; \mathsf{return}_{\mathit{getlock}} & & \\ \mathit{body}_{\mathsf{notify}} & = !\mathsf{signal}; \mathsf{return} & \mathit{body}_{\mathsf{notifyAll}} & = !\mathsf{signal\_all}; \mathsf{return} \end{array}$$

using the auxiliary statements $!\mathsf{signal}$, $!\mathsf{signal\_all}$, $?\mathsf{signal}$, and $\mathsf{return}_{\mathit{getlock}}$. In all three cases the caller must own the lock of the callee object (cf. rule $\textsc{Call}_{\mathit{monitor}}$), as expressed by the predicate *owns*, defined below. If not, the caller will deadlock, as, once devoid of the lock, the caller stops and will never obtain it. In *Java*, invoking a monitor method without owning the lock raises an exception, which terminates the culprit thread, but lets the rest of the program continue. In this sense, our model is faithful to the behavior in *Java*.

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 (cf. rule $\textsc{Call}_{\mathit{monitor}}$). Formally, the wait set $\mathit{wait}(T, \alpha)$ of an object is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?\mathsf{signal}; \mathit{stm})$. After having put itself on ice, the thread awaits notification by another thread which invokes the notify-method of the object. The notifier must own the lock of the object in question. The $!\mathsf{signal}$-statement in the notify-method thus reactivates a single thread waiting for notification on the given object (see rule $\textsc{Signal}$). Analogously to the wait set, the notified set $\mathit{notified}(T, \alpha)$ of α consists of all stacks in T with top element of the form $(\alpha, \tau, \mathsf{return}_{\mathit{getlock}})$, i.e., threads which have been notified and trying to get hold of the lock again. According to rule $\textsc{Return}_{\mathit{wait}}$, the receiver can continue after notification in executing $\mathsf{return}_{\mathit{getlock}}$ only if the lock is free. Note that the notifier does not hand

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

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

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

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

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

Table 1. Operational semantics of the monitor methods

over the lock to the one being notified but continues to own it. This behavior is known as *signal-and-continue* monitor discipline [9].

If no threads are waiting on the object, the !signal of the notifier is without effect (cf. rule $\textsc{Signal}_{\mathit{skip}}$). 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 $(T \setminus \mathit{wait}(T, \alpha)) \cup \{\xi \circ (\beta, \tau, \mathit{stm}) \mid \xi \circ (\beta, \tau, \mathsf{?signal}; \mathit{stm}) \in \mathit{wait}(T, \alpha)\}$.

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

3 The assertion language

In this section we introduce *assertions* to specify properties of Java_{MT} programs. The assertion logic consists of a local and a global sublanguage. 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 and will be used in the cooperation test. In the assertion language we add the type Object as the supertype of all classes, and we introduce *logical variables* z different from all program variables. Logical variables are used for quantification and as free variables to represent local variables in the global assertion language. Expressions and assertions are interpreted relative to a logical environment ω, assigning values to logical variables.

Assertions are program expressions, extended by logical variables and quantification. Global assertions may furthermore contain qualified references. Note that when the global expressions E and E' refer to the same object, that is, E and E' are *aliases*, then $E.x$ and $E'.x$ denote the same variable.

Quantification can be used for all types, also for reference types.[4] However, the existence of objects dynamically depends on the *global* state, something one cannot speak about on the local level. Nevertheless, one can assert the existence of objects on the local level, provided one is explicit about the domain of quantification. Thus quantification over objects in the local assertion language is restricted to $\forall z \in e.\, p$ for objects and to $\forall z \sqsubseteq e.\, p$ for lists of objects, and correspondingly for existential quantification and for composite types. Global assertions are evaluated in the context of a global state. Thus, quantification is allowed unrestricted for all types and ranges over the set of *existing* values.

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

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 the global assertion 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 of the substitution by a conditional expression [8]:

$$(E'.x_i)[\vec{E}/z.\vec{x}] = \mathsf{if}\;\; E'[\vec{E}/z.\vec{x}] = z\,\mathsf{then}\;\; E_i\;\mathsf{else}\;(E'[\vec{E}/z.\vec{x}]).x_i\,\mathsf{fi}\;\; .$$

The substitution is extended to global assertions homomorphically. We will also use $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences $\vec{y}$ possibly containing logical variables, whose semantics is defined by simultaneously applying the substitution $[\vec{E}_x/z.\vec{x}]$ and the usual capture-avoiding substitution $[\vec{E}_u/\vec{u}]$, where $\vec{x}$ and $\vec{u}$ are the sequences of the instance and local variables of $\vec{y}$, and $\vec{E}_x$ and $\vec{E}_u$ the corresponding subsequences of $\vec{E}$.

[4] In this paper we use mathematical notation like $\forall x.\, p$ etc. for phrases in abstract syntax. The concrete syntax used by *Verger* is an adaptation of JML, which we employ in typewriter font when displaying concretely annotated program text.

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. Soundness and completeness of the proof method is shown in [4]. The following section defines how to augment and annotate programs resulting in proof outlines, before Section 4.2 describes the proof method, the tool support, and two examples.

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 $\vec{y}' := \vec{e}'$. Furthermore, the observations have, in general, to be "attached" to statements they observe in a non-interleavable manner. This is syntactically represented using the special comment $/^{*}1\langle\vec{y} := \vec{e}\rangle^{*}/$ which attaches the observation to the preceding statement. 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 $/^{*}2\langle\vec{y} := \vec{e}\rangle^{*}/$. 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. For readability, in the following we use the shortcuts $\langle stm\rangle$, $\langle stm\rangle^{1}$, and $\langle stm\rangle^{2}$ for $/^{*}\langle stm\rangle^{*}/$, $/^{*}1\langle stm\rangle^{*}/$, and $/^{*}2\langle stm\rangle^{*}/$.

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. This means they are executed atomically in the sense that they cannot be interleaved by other threads. Points which can be interleaved we call control points. Points between communication and its observation cannot be interleaved; we call them *auxiliary points.* We require that the caller observation in a self-communication may not change the values of instance variables.

Example 2. Extending an assignment $x := e$ to $x := e; \langle u := x; \rangle^{1}$ stores the value of x *prior* to the execution of $x := e$ in the auxiliary variable u. Extending it to $x := e; \langle u := x; \rangle$ stores the value of x in u *after* the execution of $x := e$.

Example 3. We can store the number objects created by an instance using an auxiliary integer instance variable and by extending each object creation statement in the class of the instance to $u := \mathsf{new}^{c};\ \langle n := n + 1; \rangle^{1}$.

Example 4. We extend Example 3 by additionally extending each call $u := e_0.m(\vec{e})$ with $m \neq \mathsf{start}$ in c to $u := e_0.m(\vec{e}); \langle m := n; \rangle^{1} \langle m := n - m; \rangle^{2}$. Then the value of m after method call, but before return stores the number of objects created up to the call. After return, it stores the number of objects created during method evaluation.

Example 5. Let k be an auxiliary integer instance variable of class c. We can count the number of local configurations executing in an instance of c by extending each method body $stm; \mathsf{return}\ e_{ret}$ in c to $\langle k := k+1; \rangle^{1}\ stm;\ \mathsf{return}\ e_{ret}; \langle k := k-1; \rangle^{1}$.

The above examples show how to count objects, local configurations in an object, etc. But this information is not sufficient for a complete proof system: we have to be able to *identify* those entities. In the following we define a number of specific auxiliary variables used to formulate the verification conditions. The variables are automatically included in all augmentations. The built-in augmentation is not visible to the user, but may be used in the augmentation and annotation.

As mentioned, 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. The run-method of the initial object is executed with the parameters $(\mathsf{thread}, \mathsf{caller}) = (\alpha_0, (null, 0, null))$, where α_0 is the initial object.

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 $\mathsf{Thread} \times \mathsf{Int}$ are the analogues of the *wait*- and *notified*-sets of the semantics and store the threads waiting at the monitor, respectively those having been notified. Besides the thread identity, the number of synchronized calls is stored. In other words, these variables remember the old lock-value prior to suspension which is restored when the thread becomes active again. The boolean instance variable started, finally, remembers whether the object's start-method has already been invoked. All auxiliary variables are initialized as usual.

For the update of lists, which are represented in *PVS* by finite sequences finseq[t] of type t, we need the following functions, whose *PVS* definition is automatically generated by *Verger*: Given a sequence s and an element e, the function index retrieves the index of an occurrence of e in s, if any, and gives -1 otherwise. The function choose assigns to each non-empty sequence a non-negative integer smaller than the length of the sequence; for the empty sequence its value is -1. The expression remove(s, i) gives s without its ith element if $0 \leq i \leq |s|$, and returns s otherwise. The predicate $e \in s$ is syntactically represented by includes(s, e). The function append appends an element at the end of a sequence, and finally o concatenates two sequences. The above functions are deterministic. The use of the specific auxiliary variables is illustrated by the following example, where (: $\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{n}$:) is the notation for tuples and [: $\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}$:] for their types, i.e., product types; proj(s, i) is the projection on the ith component:

Example 6. For the class

```
public class Annotation extends Thread{
     void m1(){}
     synchronized void m2(){}
     public void run(){ this.m1();}
}
```

Verger generates the following proof outline by extending the class with the built-in augmentation:

```
public class Annotation extends Thread {
     /*< finseq[[:Thread,int:]] wait; >*/
     /*< finseq[[:Thread,int:]] notified; >*/
     /*< boolean started; >*/
     /*< int counter; >*/
     /*< [:Thread,int:] lock; >*/

void m1(Thread thread, [:Object,int,Thread:] caller) {
     /*< int conf; >*/
     /*1<conf = counter; counter = counter+1;>*/
     return;
}

synchronized void m2(Thread thread, [:Object,int,Thread:]
    caller) {
     /*< int conf; >*/
     /*1<conf = counter; counter = counter+1; lock = (:thread,proj(lock,2)
         +1:);>*/
     return;
     /*1<lock = (:proj(lock,2) == 1 ? null : proj(lock,1),proj(lock,2)-1:);>
         */
}

public void run(Thread thread, [:Object,int,Thread:] caller
    ) {
     /*< int conf; >*/
     /*1<conf = counter; counter = counter+1; started = true;>*/
     this.m1(thread, (:this,conf,thread:));
     return;
   } }
```

The class is further extended with the specification of the monitor methods:

```
public void wait(Thread thread, [:Object,int,Thread:]
    caller) {
     /*< int conf; >*/
     /*1<conf = counter; counter = counter+1;
     wait = append(wait,lock); lock = (:null,0:);>*/
     return;
     /*1<lock = notified[get(notified,thread)];
         notified = remove(notified,get(notified,thread));>*/
}

public void notify(Thread thread, [:Object,int,Thread:]
    caller) {
     /*< int conf; >*/
     /*1<conf = counter; counter = counter+1;>*/
     /*<wait = remove(wait,choose(wait));
        notified = append(notified,wait[choose(wait)]);>*/
     return;
}

public void notifyAll(Thread thread, [:Object,int,Thread:]
    caller) {
     /*< int conf; >*/
     /*1<conf = counter; counter = counter+1;>*/
     /*<notified = o(notified,wait); wait = empty_seq();>*/
     return;
}
```

The user may additionally augment and annotate the monitor methods by special comments. Note that the statements of the monitor methods, generated by *Verger*, do not use the auxiliary statements !signal, !signal_all, and ?signal of the semantics. Instead we implement the wait and notify methods by means of auxiliary instance variables wait and notified which represent the corresponding sets of the semantics. 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.

To specify invariant properties of the system, the augmented programs are *annotated* by attaching local assertions to each control and auxiliary point. In *Verger* syntax, assertions are special comments /*{p}*/, /*1{p}*/, etc., as shown in the examples below. For readability, we also use the shortcuts $\{p\}$, $\{p\}^{1}$, etc. We use the triple notation $\{p\}\ \mathit{stm}\ \{q\}$ and write $\mathit{pre}(\mathit{stm})$ and $\mathit{post}(\mathit{stm})$ to refer to the pre- and the post-condition of a statement.

For assertions at auxiliary points we use the following notation: The annotation

$$\{p_0\} \quad u := \mathsf{new}\ c; \quad \{p_1\}^{1} \quad \langle \vec{y} := \vec{e}; \rangle^{1} \quad \{p_2\}$$

of an object creation statement specifies p_0 and p_2 as pre- and postconditions, where p_1 at the auxiliary point should hold directly after object creation but before the observation. The annotation

$$\begin{array}{lllll} \{p_0\} \quad u := e_0.m(\vec{e}); & \{p_1\}^{1} & \langle \vec{y}_1 := \vec{e}_1; \rangle^{1} & \{p_2\}^{2} \\ & \{p_3\}^{3} & \langle \vec{y}_4 := \vec{e}_4; \rangle^{2} & \{p_4\} \end{array}$$

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. For the callee side, the annotation of method bodies *stm*; return e; works analogous (cf. [4]).

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.[5] 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. 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 is called a *proof outline* or *asserted program.*

Example 7. The following proof outline annotates the code of Example 1. Note how we define the functions *owns* and *free_for* and use them in the assertions. Note also the annotation of the wait method, expressing that it is not called, and thus the assertions need not be invariant under its built-in augmentation.

```
//function definitions
/*{ boolean owns(Thread thread, [:Thread,int:] lock) =
        thread!=null && thread==proj(lock,1) }*/
/*{ boolean free_for(Thread thread, [:Thread,int:] lock) =
        thread!=null && (thread==proj(lock,1) || proj(lock,1)==null) }*/

public class Account{
    private int x;

    /*{ x>=0 }*/ //class invariant

    //annotation of the wait method
    /*[ wait ]*/ /*1{ false }*/ /*{ false }*/
                /*< return; >*/ /*1{ false }*/ /*[]*/

    private void change_balance(int i){
        /*{ i>0 || (x+i>=0 && owns(thread,lock)) }*/
        x = x+i;
        /*{ i>0 || owns(thread,lock) }*/
    }

    public void deposit(int i){
        /*{i>0}*/
        change_balance(i);
    }
```

[5] 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.

```
    public synchronized void withdraw(int i){
        /*1{ free_for(thread,lock) }*/
        /*{ i>0 && owns(thread,lock) }*/
        if (x>=i) {
            /*{ x>=i && i>0 && owns(thread,lock) }*/
            change_balance(-i);
            /*2{ i>0 }*/
            /*{ owns(thread,lock) }*/
        }
        return;
        /*1{ owns(thread,lock) }*/
    }
}
```

All verification conditions generated from the above proof outline (see the following section) have been proven automatically by *PVS*, using the grind strategy.

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. They are grouped, as usual, into initial conditions, and for the inductive step into local correctness and tests for interference freedom and cooperation. Again, for details and for soundness and completeness we refer to [4]. Here, we informally introduce the proof system and illustrate it by applying the *Verger* tool to the proof outline of Example 7.

4.2.1 The *Verger* tool The tool supports the automated application of the proof system to proof outlines. Its input is a proof outline, its output contains the definitions of the reference types used in the proof outline, and the verification conditions in *PVS* syntax. The verification conditions can be verified interactively in *PVS*.

Before dealing with verification conditions, let us have a look how objects are represented in *PVS*. Besides a theory defining objects, two additional theories are generated for each class: One defining the reference type, and one specifying the state of class instances. In this way, the classes can use each other's type definition without direct mutual dependency.

Note that we do not define states in general, but specify an arbitrary single state. The type Object of the assertion language is not represented, but the *PVS* definition specifies all objects existing in the given state. The verification conditions should be satisfied by all states. Instead of showing the quantification, the *PVS* implementation assures validity of the conditions for the given arbitrary state. This simple representation increases the proof automation.

Example 8. For the class `class c { int x; }`, *Verger* generates the following type definitions:

```
Object: THEORY
BEGIN
```

```
    null: int
    Object_type: NONEMPTY_TYPE = {p:PRED[int] | p(null)}
        CONTAINING (LAMBDA (i:int): TRUE)
    Object?: Object_type
    Object: NONEMPTY_TYPE = (Object?) CONTAINING null
    class_name: NONEMPTY_TYPE = {cn:string | cn = "c"}
        CONTAINING "c"
    class: [Object->class_name]
END Object

c_type: THEORY
BEGIN
  IMPORTING Object
    c?: [Object->bool] = LAMBDA (i:Object): i=null OR class
        (i)="c"
    c: NONEMPTY_TYPE = (c?) CONTAINING null
    c_nn: TYPE = {i:c | i/=null}
END c_type

c: THEORY
BEGIN
  IMPORTING c_type
    x : [c_nn -> int]
    started : [c_nn -> bool] ...
END c
```

4.2.2 Local correctness A proof outline is *locally correct*, if the usual verification conditions [10] for standard sequential constructs hold. For instance, the effect of an assignment $\vec{y} := \vec{e}$ is expressed by substituting $\vec{e}$ for $\vec{y}$ in the assertions. We have no local verification conditions for communication or object creation, and neither for the observations attached to them. The postconditions of such statements express *assumptions* about the communicated values and is verified in the cooperation test. Invariance of the class invariant is covered by the interference freedom test.

Example 9 (Local correctness). For the proof outline of Example 7, two local conditions are generated. The first one for the assignment in line 18 expresses that the class invariant together with the precondition of the assignment imply the assignment's postcondition. The second one shows, that the precondition of the if-statement in line 29 in withdraw, the class invariant, and the boolean condition of the if-statement together imply the assertion of line 31:

```
change_balance_0 : LEMMA
((i>0 OR (x+i>=0 AND owns(thread,lock))) AND x>=0)
IMPLIES (i>0 OR owns(thread,lock))

withdraw_0 : LEMMA
((i>0 AND owns(thread,lock)) AND x>=0 AND x>=i)
IMPLIES (x>=i AND i>0 AND owns(thread,lock))
```

4.2.3 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. 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 observations of communication. Assertions $\{p\}^1$ and $\{p\}^3$ at auxiliary points do not have to be shown invariant. To distinguish local variables of the assertion from those of the assignment we rename the variables. All local variables of the thread executing the assignment get the extension $\_1$, all local variables of the assertion are extended by $\_2$, and all instance variables by $\_\mathsf{inst}$. If the assignment does not change the values of any variables occurring in the assertion, no conditions get generated.

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 $p \wedge pre(\vec{y} := \vec{e}) \wedge I_c \wedge \mathsf{interleavable} \rightarrow p[\vec{e}/\vec{y}]$. 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, which we express in terms of the auxiliary variables thread, caller, and conf, introduced earlier. If p and $\vec{y} := \vec{e}$ belong to the *same* thread, i.e., $\mathsf{thread\_1} = \mathsf{thread\_2}$, 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 $\mathsf{conf\_1} \neq \mathsf{conf\_2}$. Interference with the *matching* return statement in a self-communication need also not be considered, because communicating partners execute simultaneously. The interference freedom condition gets the antecedent $\mathsf{thread\_1} = \mathsf{thread\_2} \rightarrow ret$, where ret is

- $\mathsf{conf\_2} \neq \mathsf{conf\_1}$, for assertions $\{p\}^2$ attached to control points waiting for return, if $\vec{y} := \vec{e}$ is not the observation of return;
- $\mathsf{conf\_2} \neq \mathsf{conf\_1} \wedge (\mathsf{this} \neq \mathsf{proj}(\mathsf{caller\_1}, 1) \vee \mathsf{conf\_2} \neq \mathsf{proj}(\mathsf{caller\_1}, 2))$, for assertions $\{p\}^2$, if $\vec{y} := \vec{e}$ observes return;
- false, otherwise.

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. Furthermore, concurrent execution of synchronized methods by different threads is possible only, if the assertion is at a control point waiting for a return value. The interference freedom test becomes an additional antecedent of the form $\mathsf{thread\_1} \neq \mathsf{thread\_2} \rightarrow cond$; for technical details we refer to [4].

Example 10 (Interference freedom). Satisfaction of the class invariant of the proof outline of Example 7 is assured by the condition

```
%precondition assignment
```

```
((i_1>0 OR (x_inst+i_1>=0 AND owns(thread_1,lock_inst)))
    AND
     %class invariant
     x_inst >= 0)
     %class invariant after execution
IMPLIES (x_inst+i_1>=0)
```

generated for the only assignment (18), which changes the balance x. That (31) is invariant under the same assignment, is assured by the condition

```
%preconditions assignment
((i_1>0 OR (x_inst+i_1>=0 AND owns(thread_1,lock_inst)))
    AND
     %assertion
     x_inst>=i_2 AND i_2>0 AND owns(thread_2,lock_inst) AND
     %class invariant
     x_inst >= 0 AND
     %interleavable
     (thread_1=thread_2 IMPLIES false) AND
     (thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(x_inst+i_1>=i_2 AND i_2>0 AND owns(thread_2,lock_inst))
```

If $\mathsf{i\_1} > 0$, then $\mathsf{x\_inst} \geq \mathsf{i\_2}$ implies $\mathsf{x\_inst} + \mathsf{i\_1} \geq \mathsf{i\_2}$, and the condition is satisfied, which corresponds to the concurrent execution of the methods withdraw and deposit. Otherwise, owns(thread_1, lock_inst), owns(thread_2, lock_inst), and $\mathsf{thread\_1} \neq \mathsf{thread\_2}$ lead to a contradiction. This case corresponds to the concurrent execution of withdraw, which is not possible. There is a similar condition for the case that two threads are concurrently executing the change_balance method, showing that (17) is invariant under (18).

The remaining conditions are all generated for invariance under changing the lock value. There are altogether 5 assertions at control points, which have to be shown invariant under entering and exiting the wait method. As the wait-method, however, is not invoked, as expressed by its annotation, the left-hand side of the generated implications is false.

The only remaining assignments changing the lock value are the observations at the beginning and at the end of the synchronized $mathsfwithdraw$ method. Assertions in that method that are not at a control point waiting for return, does not have to be invariant under the execution of withdraw. Thus only the assertions (17) and (19) in change_balance have to be shown invariant, which yields 4 conditions. For invariance of (17) under entering the withdraw method we get:

```
%precondition assignment
(free_for(thread_1,lock_inst) AND
     %assertion
     (i_2>0 OR (x_inst+i_2>=0 AND owns(thread_2,lock_inst)))
          AND
     %class invariant
     x_inst>=0 AND
     %interleavable
     (thread_1=thread_2 IMPLIES false) AND
     (thread_1/=thread_2 IMPLIES true))      IMPLIES
%assertion after execution
```

```
  (i_2>0  OR
     (x_inst+i_2>=0 AND
      owns(thread_2,(thread_1,(PROJ_2(lock_inst)+1)))))
```

Note that the predicates free_for(thread_1, lock_inst), owns(thread_2, lock_inst), and thread_1 $\neq$ thread_2 together lead to a contradiction: If a thread executing the private change_balance-method owns the lock, then no other thread can enter the synchronized withdraw-method. The remaining three conditions are analogous.

Example 11 (Wait). The following example illustrates properties of the wait-method.[6] All conditions generated for the proof outline are proven in *PVS*.

```
1   /*{ boolean owns(Thread thread, [:Thread,int:] lock) =
2             thread!=null && proj(lock,1)==thread }*/
3   /*{ boolean not_owns(Thread thread, [:Thread,int:] lock) =
4             thread!=null && proj(lock,1)!=thread }*/
5   /*{ boolean free_for(Thread thread, [:Thread,int:] lock) =
6           thread!=null && (thread==proj(lock,1) || lock==(:null,0:)) }*/
7   /*{ boolean disjunct(finseq[[:Thread,int:]] x) =
8           (\forall int i,j; 0<=i && 0<=j && i<length(x)&& j<length(x) && i!=
                j; proj(x[i],1)!=proj(x[j],1)) }*/

10  public class WaitExample{
11      /*< finseq[[:Thread,int:]] x;>*/

13      /*{ disjunct(x) }*/ //class invariant

15      /*[wait]*/  /*1{ owns(thread,lock) }*/
16        /*{ not_owns(thread,lock) && proj(caller,1)==this && includes(x,(:
               thread,proj(caller,2):)) }*/
17        /*<return;>*/
18        /*1{ lock==(:null,0:) && proj(caller,1)==this && includes(x,(:
               thread,proj(caller,2):)) && get(notified,thread)!=-1 }*/
19      /*[]*/

21      public synchronized void m(){
22          /*1{ free_for(thread,lock) && (\forall int i;true;!includes(x,(:
                 thread,i:))) }*/
23          /*1< x=append(x,(:thread,counter:)); >*/
24          /*{ owns(thread,lock) && includes(x,(:thread,conf:)) }*/
25          try{ this.wait();} catch (InterruptedException e)
                 {}
26          /*2{ not_owns(thread,lock) && includes(x,(:thread,conf:)) }*/
27          /*{ owns(thread,lock) && includes(x,(:thread,conf:)) }*/
28          return;
29          /*1{ owns(thread,lock) && includes(x,(:thread,conf:)) }*/
30          /*1< x=remove(x,index(x,(:thread,conf:))); >*/
31      }
32  }
```

The wait-method can be called only by a thread owning the lock of the callee object, as expressed by the precondition (24). After invoking wait, the thread

[6] We currently do not handle exceptions in Java$_{MT}$ and its proof theory. To call the wait-method, however, we must syntactically catch InterruptedExceptions. But, since we do not support the interrupt method, it cannot be thrown.

gives the lock free, as formalized in (26). When returning, it becomes the lock owner again, as stated by (27).

We use the auxiliary instance variable x to store, for each local configuration executing m, the identities of the thread and the local configuration, to identify local configurations in caller-callee relationship. We can exclude, for instance, from the interference freedom test the invariance of (26) under the built-in return-observation of its callee, setting the lock owner to the identity of the executing thread. Clearly, (26) would not be invariant under the return-observation of its callee; caller and callee execute a common step and the control point of the caller moves from (26) to (27). We get the following interference freedom condition for setup, where thread_1 = thread_2 leads to a contradiction:

```
%precondition assignment
(lock_inst=(null,0) AND PROJ_1(caller_1)=this AND
  includes(x_inst,(thread_1,PROJ_2(caller_1))) AND
  get(notified_inst,thread_1)/=(-1) AND
  %assertion
  not_owns(thread_2,lock_inst) AND includes(x_inst,(
      thread_2,conf_2)) AND
  %class invariant
  disjunct(x_inst) AND
  %interleavable
  (thread_1=thread_2 IMPLIES
      (conf_1/=conf_2 AND (this/=PROJ_1(caller_1) OR
          conf_2/=PROJ_2(caller_1)))) AND
  (thread_1/=thread_2 IMPLIES true)) IMPLIES
%assertion after execution
(not_owns(thread_2,seq(notified_inst)(get(notified_inst,
    thread_1))) AND
   includes(x_inst,(thread_2,conf_2)))
```

4.2.4 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 assertions at the auxiliary points before 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.

In the following we introduce the cooperation test for method invocation. For object creation we again refer to [4]. 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 extend the names of local variables of the caller by _1 and those of the callee by _2.

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. In case of a synchronized method invocation, the communication must be enabled, i.e., the lock of the callee object has to be free or owned by the caller. This is expressed by $z'.\mathsf{lock} = (\mathsf{null}, 0) \vee \mathsf{proj}(z'.\mathsf{lock}, 1) = \mathsf{thread\_1}$, where $\mathsf{thread\_1}$ is the caller-thread. For the invocation of the monitor methods we require that the executing thread is holding the lock. An additional predicate $E_0 = z'$ in the condition of a method call $e_0.m(\vec{e})$ states, that z' is indeed the callee object, where E_0 is $e_0[z/\mathsf{this}]$ with every local variable name suffixed by $\mathsf{\_1}$.

For returning from a method we have similar verification conditions (see [4]). Here we remark only that returning from the wait-method assumes that the thread has been notified and that the callee's lock is free. Remember that method invocation hands over the return address, and that the values of formal parameters remain unchanged. Furthermore, actual parameters may not contain instance variables, i.e., their interpretation does not change during method execution. Therefore, the formal and actual parameters can be used to identify partners being in caller-callee relationship, using the built-in auxiliary variables.

The effect of communication, changing local states only, is expressed by simultaneous substitution. This means, in case of a method call, the formal parameters get replaced by the actual ones expressed in the global language. The effect of the caller observation $\langle \vec{y} := \vec{e} \rangle^{1}$ to a global assertion P is expressed by the substitution $P[\vec{e}/z.\vec{y}]$. The effect of the callee-observation is handled similarly. Note the order: first communication takes place, followed by the sender and then the receiver observation.

Example 12 (Cooperation test). For the proof outline of Example 7, three global conditions are generated: one for the method call at (24), one for the call at (32), and one for corresponding the return for the second call. Note that we do not have any conditions for returning from the first call (24), because all postconditions are true by definition. The first condition

```
FORALL (caller:Account) : caller/=null IMPLIES
FORALL (callee:Account) : callee/=null IMPLIES
  %precondition caller
  ((i_1>0 AND
       %class invariant caller and callee + caller-callee relationship
       Account.x(caller)>=0 AND
       Account.x(callee)>=0 AND
       caller=callee)            IMPLIES
  %postcondition callee
  (i_1>0 OR (Account.x(callee)+i_1>=0 AND owns(thread_1,
      Account.lock(callee)))))
```

states that the class invariants and the preconditions of caller and callee imply the postcondition of the callee. The *PVS*-expression $c.x(z)$ represents the qualified reference $z.x$ for z of type c. Note that the global invariant, the postcondition of the caller, and the assertions at the auxiliary points are by definition true. The caller-callee relationship of the partners is assured by requiring caller = callee, since it is a self-call. The condition for the second call is similar. The condition for return assures the caller-callee relationship of the partners by additionally requiring, that the formal parameters equal the actual ones. Applied to the built-in auxiliary parameter thread, this requirement implies for example that caller and callee are the same thread, i.e., thread_1 = thread_2, which we need to show that the caller owns the lock after communication:

```
FORALL (caller:Account) : caller/=null IMPLIES
FORALL (callee:Account) : callee/=null IMPLIES
  %precondition caller
  ((i_1>0 AND
      %class invariant caller
      Account.x(caller)>=0 AND
      %precondition callee
      (i_2>0 OR owns(thread_2,Account.lock(callee))) AND
      %class invariant callee
      Account.x(callee)>=0 AND
      %caller-callee relationship
      caller=callee AND i_2=(-i_1) AND thread_2=thread_1
         AND
      caller_2=(caller,conf_1,thread_1))
                                   IMPLIES
  %postcondition caller
  owns(thread_1,Account.lock(caller)))
```

5 Conclusion

This paper presents a tool-supported assertional proof method for a multi-threaded sublanguage of *Java* including its monitor discipline. This builds on earlier work ([3] and especially [5]). The underlying theory, the proof rules, and their soundness and completeness are presented in greater detail in [4], where here we introduced the assertional proof system on a small number of examples, which have been verified using the *Verger* tool.

Related work As far as proof-systems and verification support for object-oriented programs is concerned, research mostly concentrated on *sequential* languages. Early examples of Hoare-style proof systems for a sequential object-oriented languages are [18] and [26, 27, 18]. With *Java*'s rise to prominence, research more intensively turned to (sublanguages of) *Java*, as opposed of capturing object-oriented language features in the abstract. In this direction, JML [25] has emerged as some kind of common ground for asserting *Java* programs. Another trend is to offer mechanized proof support for the languages. For instance, Poetzsch-Heffter and Müller [36, 34, 33, 35] develop a Hoare-style programming logic presented in sequent formulation for a sequential kernel of *Java*, featuring interfaces, subtyping, and inheritance. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer-assisted soundness proof. The work in the LOOP-project (cf. eg. [29, 24]) also concentrates on a sequential subpart of *Java*, translating the proof-theory into *PVS* and *Isabelle/HOL*.

The work [38, 37] use (a modification of) the *object constraint language* OCL as assertional language to annotate UML class diagrams and to generate proof conditions for *Java*-programs. The work [12] presents a model checking algorithm and its implementation in *Isabelle/HOL* to check type correctness of Java bytecode. In [41] a large subset of *JavaCard*, including exception handling, is formalized in *Isabelle/HOL*, and its soundness and completeness is shown within the theorem prover. The work in [2] presents a Hoare-style proof-system for a sequential object-oriented calculus [1]. Their language features heap-allocated objects (but no classes), side-effects and aliasing, and its type system supports subtyping. Furthermore, their language allows nested statically let-bound variables, which requires a more complex semantical treatment for variables based on closures, and ultimately renders their proof-system incomplete. Their assertion language is presented as an extension of the object calculus' language of type and analogously, the proof system extends the type derivation system. The close connection of types and specifications in the presentation is exploited in [40] for the generation of verification conditions.

Work on proof systems for parallel object-oriented languages or in particular the multithreading aspects of *Java* is more scarce. [15] presents a sound and complete proof system in weakest precondition formulation for a parallel object-based language, i.e., without inheritance and subtyping, and also without reentrant method calls. Later work [32] [17] [16] include more features, especially cater for Hoare logic for inheritance and subtyping.

A survey about *monitors* in general, including proof-rules for various monitor semantics, can be found in [13].

Future work As future work, we plan to extend $Java_{MT}$ by further constructs, like inheritance and subtyping. To deal with subtyping on the logical level requires a notion of behavioral subtyping [7]. An extension of the semantics and the proof theory to detect deadlocks and termination is also of interest.

Acknowledgments We thank Cees Pierik for fruitful discussions and suggestions, and furthermore Tim D'Avis for careful reading and commenting on an earlier version of this document.

References

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*, volume 1214 of *Lecture Notes in Computer Science*, 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-Mumm and F. de Boer. Proof-outlines for threads in Java. In C. Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 2000.
4. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's monitor concept: Soundness and completeness. Technical Report TR-ST-02-3, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 2002. to appear.
5. 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.
6. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. LNCS State-of-the-Art-Survey. Springer-Verlag, 1999.
7. P. America. A behavioural approach to subtyping in object-oriented programming languages. 443, Phillips Research Laboratories, January/April 1989.
8. P. America and F. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1993.
9. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.
10. 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.
11. 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.
12. D. Basin, S. Friedrich, and M. Gawkowski. Verified bytecode model checkers. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs'02)*, pages 47–66, August 2002.
13. P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, Mar. 1995.
14. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [6], pages 157–200.
15. F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–156. Springer-Verlag, 1999.
16. F. S. de Boer and C. Pierik. Computer-aided specification and verification of annotated object-oriented programs. In B. Jacobs and A. Rensink, editors, *Proceedings of the Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, volume 209, pages 163–177. Kluwer, 2002.
17. F. S. de Boer and C. Pierik. Towards an environment for the verification of annotated object-oriented programs. Technical report UU-CS-2003-002, Institute of Information and Computing Sciences, University of Utrecht, Jan. 2003.
18. C. C. de Figueiredo. A proof system for a sequential object-oriented language. Technical Report UMCS-95-1-1, University of Manchester, 1995.
19. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
20. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
21. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [22].

22. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science.* International Series in Computer Science. Prentice Hall, 1989.
23. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.
24. B. Jacobs, J. van den Berg, M. Huisman, M. van Barkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA) '98*, pages 329–340. ACM, 1998. in *SIGPLAN Notices.*
25. G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modelling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.
26. G. T. Leavens and W. E. Wheil. Reasoning about object-oriented programs that use subtypes. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA) '90 (Ottawa, Canada)*, pages 212–223. ACM, 1990. Extended Abstract.
27. G. T. Leavens and W. E. Wheil. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 1994. An expanded version appeared as Iowa State Unversity Report, 92-28d.
28. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
29. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.
30. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
31. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.
32. C. Pierik and F. S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. Technical report UU-CS-2003-010, Institute of Information and Computing Sciences, University of Utrecht, 2003.
33. A. Poetzsch-Heffter. A logic for the verification of object-oriented programs. In R. Berghammer and F. Simon, editors, *Proceedings of Programming Languages and Fundamentals of Programming*, pages 31–42. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Nov. 1997. Bericht Nr. 9717.
34. A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* Technische Universität München, Jan. 1997. Habilitationsschrift.
35. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W.-P. de Roever, editors, *Proceedings of PROCOMET '98.* International Federation for Information Processing (IFIP), Chapman & Hall, 1998.
36. 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.
37. 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*, volume 2029 of *Lecture Notes in Computer Science*, pages 300–316. Springer-Verlag, 2001.
38. B. Reus and M. Wirsing. A Hoare-logic for object-oriented programs. Technical report, LMU München, 2000.
39. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine.* Springer-Verlag, 2001.

40. 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.
41. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency – Practice and Experience*, 2001. to appear.
42. 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.
43. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml.* Object Technology Series. Addison-Wesley, 1999.