A Sound and Complete Shared-Variable Concurrency Model for Multi-Threaded Java Programs

F.S. de Boer

CWI, Amsterdam, Netherlands
F.S.de.Boer@cwi.nl

Abstract. In this paper we discuss an assertional proof method for multi-threaded Java programs. The method extends the proof theory for sequential Java programs with a generalization of the Owicki/Gries interference freedom test to threads in Java.

1 Introduction

We present a simple proof method which captures the main aspects of the multi-threaded flow of control in Java. In the object-oriented programming language Java instances of thread classes can be dynamically allocated and start their own thread of control. A thread class (defined as an extension of the public built-in Java class 'Thread') defines a run method and a call of the start method creates a new thread of computation initiated by the execution of the run method. The following Java syntax,

```
class MyThread extends Thread {
⋮
public void run() {
··· }
⋮
}
```

specifies a thread class 'MyThread' with a run method. The following code would then create a thread and start it running:

```
MyThread t = new MyThread();
t.start();
```

The thread executing this code continues its own execution, i.e., it does not wait for the start method to return. Operationally, a thread is described by a stack of calls generated by the run method. In this model of computation the different threads share the global object structure which consists of the values of the instances variables of the dynamically allocated objects and the static variables of the classes.

Our proof method consists of annotating each class definition of the given program with *assertions* which express certain global properties of the dynamically allocated program variables. Such an annotated class definition is *locally* correct if certain verification conditions hold which characterize the sequential flow of control within one thread. On the other hand, reasoning about the multi-threaded flow of control within an object involves a global *interference freedom test*. This test is modeled after the corresponding test in [13] for concurrent systems consisting of a statically fixed number of processes which interact via shared variables.

The main contribution of this paper is the generalization of the interference freedom test to *dynamic thread creation*. Furthermore, this paper also provides a formal justification of this generalization in terms of soundness and completeness proofs.

Related work To the best of our knowledge the only other (proven) sound and complete proof method for annotated multi-threaded Java programs is described in [1]. The proof method in [1] combines the Owicki&Gries method for shared variable concurrency with the proof method for Communicating Sequential Processes (CSP) as introduced in [4]. The latter proof method is used to reason about method calls in terms of message passing between objects. By restricting to Java programs that have no static variables and do not allow dereferencing, in [1] objects only interact via message passing. Consequently the interference freedom test in [1] only involves the local state of an object. In contrast, in this paper we extend the proof method for sequential Java programs, which is based on the standard proof theory of recursive procedures (see [2]), with a global interference freedom test. In other words, the main difference is that the proof method in [1] is based on a model of concurrent objects (along the lines of [8]) whereas the proof method in this paper is strictly thread-based. The model of concurrent objects integrates shared-variable concurrency and message passing, whereas the thread-based model integrates shared-variable concurrency with recursive method call. We think the latter integration more faithfully captures the semantics of the Java programming language.

2 Assertions

Assertions are used to annotate the control points of method bodies. In this paper we abstract from the syntax of assertions (we denote by $\neg$, $\wedge$, $\rightarrow$ the logical connectives of negation, conjunction and implication). An assertion P is evaluated in a configuration. A configuration γ consists of an object structure and a local context. For every existing object an object structure assigns values to its instance variables (static variables belong to the object representing their class). A local context assigns values to the the local variables of the method. The local variables of a method include its formal parameters. We view the keyword 'this' as an implicit local variable which denotes the current object. We denote by

$$\gamma \models P$$

that the configuration γ satisfies the assertion P. An assertion is valid, denoted by $\models P$, if $\gamma \models P$, for every configuration γ.

For a (sequential) statement S in Java, we denote by

$$WP(S, P)$$

the weakest precondition which guarantees that every terminating execution of S satisfies P. Formally, this weakest precondition is semantically defined in terms of a structural operational semantics for transitions

$$\langle S, \gamma \rangle \rightarrow \gamma'$$

where γ denotes an initial configuration and γ' denotes the resulting configuration of the execution of S. Note that the current object is given by the local context of the initial configuration. Given such an operational semantics, we have the following standard definition

$\gamma \models WP(S, P)$ if and only if $\gamma' \models P$, for $\langle S, \gamma \rangle \rightarrow \gamma'$

We refer to [6] for a weakest precondition calculus which formalizes aliasing and object creation at an abstraction level that coincides with that of the Java programming language.

In this paper, we denote by $P\sigma$ the result of applying the substitution σ to the assertion P. An application of a substitution

$$[e_1/u_1, \ldots, e_n/u_n]$$

simply consists of replacing simultaneously every occurrence of the *local* variable u_i by the corresponding expression e_i.

3 Proof-outlines

A proof-outline is a correctly annotated program. An annotation of a multi-threaded Java program associates with every sub-statement S (appearing in a method body) a precondition $Pre(S)$ and a postcondition $Post(S)$. Validation of verification conditions establish the correctness of an annotated program. We first discuss the verification condition which establishes that assertions are interference free. Then we discuss the verification conditions which establish that assertions specify correctly the sequential control flow.

Interference freedom test

In order to characterize the interference between different threads we assume that each method has a distinguished local variable 'thread' which is used to identify the executing thread. A thread itself is uniquely identified by the initial object executing its run method (in Java calling the start method twice on an object throws the exception ' IllegalThreadStateException').

We define an assertion P to be invariant over the execution of a statement S by a *different* thread if the following verification condition holds:

$$\models (P \wedge Pre(S) \wedge \text{thread!} = \text{thread}') \rightarrow WP(S, P)$$

For notational convenience, we implicitly assume that the local variables of P and $Pre(S)$ are named apart by 'priming' the local variables of P. Note that these local variables include 'this' and 'thread', which are thus renamed in P by 'this$'$' and 'thread$'$'. Furthermore, it is important to note that an unqualified instance variable x of the class of the current object denoted by 'this', is transformed by this renaming into 'this$'.x$'.

The above verification condition models the situation that the execution of the thread denoted by the fresh local variable 'thread$'$' in the object denoted by the fresh local variable 'this$'$' is interleaved by the execution of the statement S by the thread denoted by the distinguished local variable 'thread' in the object denoted by the distinguished local variable 'this'. That we are dealing with two *different* threads is simply described by the disequality thread! = thread$'$.

Example 1. As a (very) simple example, given a boolean instance variable 'b', the assertion 'thread.b' is invariant over the execution of an assignment 'thread.b=false' by *another* thread. This is captured by the valid verification condition

$$\models (\text{thread}'.\text{b} \wedge \text{thread!} = \text{thread}') \rightarrow WP(\text{thread.b=false}, \text{thread}'.\text{b})$$

Example 2. In example 7 we introduce an instance variable 'lock' to reason about synchronized methods. This variable stores the identity of the thread that owns the lock of the object. That is, for every synchronized method we have the invariant

thread==lock

With this additional information any annotation of a synchronized method is trivially interference free:

$$\models (\text{thread}' == \text{lock} \wedge \text{thread} == \text{lock} \wedge \text{thread}! = \text{thread}') \rightarrow \text{false}$$

Local correctness

An annotated program is locally correct if the verification conditions hold which characterize the sequential flow of control within one thread. We have the standard verification conditions which characterize control structures like sequential composition, choice, and iteration constructs.

Method invocation and return Without loss of generality, we restrict discussion to the verification conditions for method calls

$$x = e_0.m(e_1, \ldots, e_n)$$

where x is an instance variable or a local variable, and $e_0, e_1, \ldots, e_n$ are expressions without side-effect and which are not affected by the call itself. Furthermore, we assume that the formal parameters of the method m are read-only. Given such a call we denote by σ the (simultaneous) substitution

$$[e_0, e_1, \ldots, e_n/\text{this}, u_1, \ldots, u_n]$$

This substitution describes the context switch which consists of passing control to the callee, modeled by substituting every occurrence of 'this' by e_0, and initializing the formal parameters $u_1, \ldots, u_n$ of the method m, modeled by substituting every local variable u_i by e_i, $i = 1, \ldots, n$.

Let S denote the body of the method m. We have the following verification condition for the precondition P of the call $x = e_0.m(e_1, \ldots, e_n)$:

$$\models P \rightarrow \mathit{Pre}(S)\sigma$$

Here we assume that the local variables of the precondition $\mathit{Pre}(S)$ of the method body, excluding the formal parameters of the method and the local variables 'thread' and 'this', are named apart from those in P. Note that the distinguished local variable 'thread' thus may occur both in the precondition P of the caller and the precondition $\mathit{Pre}(S)$ of the callee. We do not need to distinguish these different occurrences because the local variable 'thread' in both preconditions denotes the *same* thread executing the method call.

Example 3. Consider the precondition

account.lock==thread

of a call

newbalance=account.add(amount)

of a synchronized method. This precondition can be obtained from the precondition

$$\text{lock} == \text{thread}$$

of the body of the method 'add' simply by replacing the (implicit) local variable 'this' by the expression 'account', which transforms the expression 'lock' into the expression 'account.lock'

Furthermore, we have the following verification condition for the postcondition Q of a call $x = e_0.m(e_1, \ldots, e_n)$:

$$\models Post(S)\sigma \rightarrow WP(x = \text{return}, Q)$$

As above, we assume that the local variables of the postcondition $Post(S)$ of the method body, excluding the formal parameters of the method and the local variables 'thread' and 'this', are named apart from those in Q. Note that since the formal parameters are read-only and the actual parameters are not affected by the call itself, we can apply the subsitution σ modeling the context switch and parameter passing. The distinguished local variable 'return' is used to store temporarily the return value. That is, the precondition P and the postcondition Q of a return statement are validated by the verification condition

$$\models P \rightarrow Q[e/\text{return}]$$

where e denotes the return value.

Example 4. Consider the postcondition

$$\text{lock} == \text{thread} \wedge \text{return} == \text{balance} + \text{u}$$

of the body S of the synchronized method 'add' ('u' denotes its formal parameter). Applying the context switch and parameter passing of the call

newbalance=account.add(amount)

we obtain the assertion

$$\text{account.lock} == \text{thread} \wedge \text{return} == \text{account.balance} + \text{amount}$$

This assertion clearly implies the weakest precondition of the assignment 'newbalance=return' and the postcondition

$$\text{account.lock} == \text{thread} \wedge \text{newbalance} == \text{account.balance} + \text{amount}$$

of the call.

Auxiliary variables

In general to prove the correctness of a program we need auxiliary variables which are used to describe certain properties of the flow of control.

*Example 5 (***Mutual exclusion).** Consider the run method defined by

```
run(){
sem.acquire();S;sem.release() }
```

where sem is a static binary semaphore (initialized to 1). In order to prove that no two threads are executing the critical section S in the body of the run method we introduce a static variable 'in' which stores the set of threads that are in their critical section (it is initialized to the empty set). We extend the run method as follows:

```
run(){
[sem.acquire();in.add(thread)];S;[sem.release();in.remove(thread)] }
```

The brackets are used to indicate statements which are assumed to be executed atomically, that is without interleaving. Note that without loss of generality we can indeed assume that between acquiring (or releasing) the semaphore and the corresponding update of the auxiliary variable no other threads are interleaved.

Mutual exclusion then can be expressed by the assertion Mutex defined by

$$\text{in.size} == \text{sem} \wedge 0 <= \text{sem} \wedge \text{sem} <= 1$$

Note that in the assertion language, the static variable 'sem' is simply an integer variable, which takes the values 0 or 1.

The assertion Mutex is introduced as an invariant of the run method which annotates all its interleaving points, that is, the start and end of the body of the run method itself, and the start and end of the critical section S.

For the proof of the local correctness of the annotation we use the following (standard) characterization of the weakest precondition of a postcondition Q (of the operations for acquiring and releasing the semaphore):

$$WP(\text{sem.acquire}(), Q) = (\text{sem} == 1 \rightarrow Q[0/\text{sem}])$$

and

$$WP(\text{sem.release}(), Q) = (\text{sem} == 0 \rightarrow Q[1/\text{sem}])$$

Local correctness of the invariant Mutex then is expressed by the verification conditions

$$\models \text{Mutex} \rightarrow WP(\text{sem.acquire}(); \text{in.add}(\text{thread}), \text{Mutex})$$

and

$$\models \text{Mutex} \rightarrow WP(\text{sem.release}(); \text{in.remove}(\text{thread}), \text{Mutex})$$

Next we note that these local verification conditions which establish Mutex as an invariant of the run method, (trivially) imply the verification conditions for the interference freedom test:

$$\models (\text{Mutex} \wedge \text{thread!} = \text{thread}') \rightarrow WP(\text{sem.acquire}(); \text{in.add}(\text{thread}), \text{Mutex})$$

and

$$\models (\text{Mutex} \wedge \text{thread!} = \text{thread}') \rightarrow WP(\text{sem.release}(); \text{in.remove}(\text{thread}), \text{Mutex})$$

In other words, using a local invariant like Mutex makes the inteference freedom test *redundant*.

Auxiliary variables are also used to describe the semantics of built-in mechanisms in Java. Below we describe the semantics for starting a thread, the execution of synchronized methods, and the semantics of the synchronization mechanism of wait and notify methods.

Start method In order to describe the specific semantics of the start method, we assume that each thread class has a boolean auxiliary instance variable 'Alive' which indicates that the start method of the object has been called and its run method has not yet terminated. Otherwise it is false. It is initialized to 'false' by the constructor method. Note that in Java starting a running thread throws an exception. Since, for technical convenience only, this paper restricts to invariance properties of normal executions of multi-threaded Java programs, we can describe the semantics of the start method simply by the code

if !e.Alive {
e.Alive=true;e.start }
else { abort }

Correspondingly, we append the body of a run method by the assignment 'Alive=false'. We have the following (standard) verification condition of the 'abort' statement: For *arbitrary* postcondition Q

$$\models \text{false} \rightarrow Q$$

Note that this verification condition validates *any* postcondition.

The precondition of a call e.start of the start method is validated like the precondition of an ordinary call, as described above. The postcondition Q of the call e.start is simply validated by the verification condition

$$\models P \rightarrow Q$$

where P denotes its precondition.

Example 6. Clearly we can validate by means of the above verification condition for method calls, for every run method, the precondition

$$\text{thread} = \text{this} \wedge \text{this.Alive}$$

Consequently, every local assertion of a run method is trivially invariant over any local assignment of any run method: For example, the local assertion 'b', where 'b' is an instance variable, is invariant over an assignment b=false in any run method, because

$$\models (\text{this}'.\text{b} \wedge \text{thread}' = \text{this}' \wedge \text{thread} = \text{this} \wedge \text{thread}! = \text{thread}') \rightarrow WP(\text{b} = \text{false}, \text{this}'.\text{b})$$

trivially holds (note that the antecedent implies this! = this$'$).

*Example 7 (***Synchronized methods).** In order to describe the specific semantics of synchronized methods in Java, we introduce an auxiliary (instance) variable 'lock' which belongs to the class of the method and which stores the identity of the thread owning the lock. Since a thread releases the lock of an object only when it has finished executing its synchronized methods the thread has called on the object, we also need an auxiliary (instance) variable 'count' which belongs to the class of the thread and which denotes the number of called synchronized methods in the thread. Every method invocation $e_0!m(e_1, \ldots, e_n)$ involving a synchronized method m is prefixed with an *await* statement

await e_0.lock==thread || e_0.lock==null{
e_0.lock=thread;thread.count++ }

The boolean condition states that either the thread already owns the lock or the lock is not yet initialized (i.e., is 'free').

On the other hand, every synchronized method ends with the execution of the await statement

await true {
thread.count-=1; if thread.count==0 { lock=null}}

We extend our notion of proof outlines with the following standard verification condition for await statements

$$\models (P \wedge b) \rightarrow WP(S, Q)$$

where P and Q denote the precondition and the postcondition of the await statement, b denotes its boolean condition and S denotes its main body.

Since the evaluation of the boolean guard of an *await*-statement and the execution of its body are assumed to be atomic we only need to apply the interference freedom test to the pre- and postcondition of the *await*-statement itself.

*Example 8 (***Wait and notify).** A thread which owns the lock of an object can release it by calling the wait method on the object. It has to wait until another thread owning the lock calls the 'notify' or 'notifyAll' method on this object. In order to describe the semantics of this mechanism we denote by 'wait' an auxiliary instance variable of the object which is used to store the set of objects waiting for its lock. The semantics of a call

e.wait()

then is described by the following statement

if lock==thread{
u=e;u.lock=null;u.wait.add(thread) }
else { abort };
await u.lock==null & !u.wait.contains(thread) {
u.lock=thread}

Here 'u' is a 'fresh' local variable used to keep the identity of the object. This statement first checks whether the thread owns the lock. If so, the thread simply releases the lock and is added to the set of waiting threads. If the thread does not own the lock the execution is aborted because we only consider normal executions (e.g., we abstract from exceptions). The await statement waits for the lock to be free and for the thread to be removed from the set of waiting threads. A call

e.notifyAll()

of the 'notifyAll' method is modeled by the statement

if lock==thread{
e.wait.clear()}
else { abort }

which removes all waiting threads (in case the executing thread owns the lock). In order to model a call

e.notify()

which involves an *arbitrary* choice of the thread to be notified, we introduce an (abstract) set operation 'removeAny()' which removes an arbitrary element from a set. We then can model the above call by the statement

if lock==thread{
e.wait.removeAny() }
else { abort }

Given a precondition P, a postcondition Q of a statement

e.wait.removeAny()

is validated by the verification condition

$$\models (P \wedge e.\text{wait.contains(any)}) \rightarrow WP(e.\text{wait.remove(any)}, Q)$$

By definition of the validity of assertions the 'fresh' local variable 'any' is here implicitly universally quantified.

In general, auxiliary variables can be introduced as local variables, instance variables and static variables. Assignments to auxiliary variables can be introduced which are side-effect free (e.g., assignments which do not involve methods calls or object creation) and which do not affect the flow of control of the given program. It is important to note that we also allow auxiliary variables as additional formal parameters of method definitions. Such auxiliary variables can be used to reason about invariance properties of method calls.

*Example 9 (***Faculty function).** Consider for example the following recursive method for computing the faculty function.

fac() {
if x>0 { x-=1;this.fac();x++;y=y*x } else { y=1 }}

Here 'x' and 'y' are instance variables. Upon termination 'y' stores the faculty of the value stored by 'x'. In order to prove that the value of 'x' upon termination equals its old value, we introduce as auxiliary variable a formal parameter u and extend the method by

fac(u) {
if x>0 { x-=1;this.fac(u-1);x++;y=y*x } else { y=1 }}

We then can express the above invariance property by introducing the assertion 'u==x' both as precondition and the postcondition of the method body. This specification of the method body can be validated by introducing 'u==x+1' as the precondition and the postcondition of the recursive call. We have the following trivial verification conditions for method invocation and return

$\models$u==x+1→u-1==x and $\models$u-1==x→u==x+1

where the assertion 'u-1==x' results from replacing the formal parameter 'u' in 'u==x' by the actual parameter 'u-1'.

4 Soundness and completeness

In this section we sketch soundness and completeness proofs. These proofs are based on a formal semantics of multi-threaded Java programs. This semantics is described in terms of a structural operational semantics which defines a transition relation on global states. A global state Θ of a program consists of a set of threads and an object structure which specifies for every existing object the values of its instance variables. Operationally, a thread is a stack of closures, i.e., pairs (S, τ) consisting of a statement S and a local context τ specifying the values of the local variables of S. For any two closures (S, τ) and (S', τ') belonging to the same thread we have that

$$\tau(\text{thread}) = \tau'(\text{thread})$$

because the local variable 'thread' denotes the initial object (executing its run method). That is, for the bottom closure (S_0, τ_0) of a thread we have that

$$\tau_0(\text{thread}) = \tau_0(\text{this})$$

The thread itself is executing the closure on top of the call stack, which is also called its active closure. All other closures represent pending calls. The details of the definition of the global transition relation

$$\Theta \to \Theta'$$

which represents the execution of an atomic statement by one thread in Θ resulting in the global state Θ', are straightforward and omitted (see also [1]).

For notational convenience only, we assume throughout this section that every interleaving point of the given program is uniquely labeled. Such labels we denote by $l, l', \ldots$. By

$$l : S : l'$$

we denote a statement S with its start and end labeled by l and l', or the label l itself (': $S : l'$' thus being optional). A label on its own marks the termination of a method body. The assertion annotating an interleaving point l we denote by $@l$.

Soundness

Let π be an annotated program. A global state Θ satisfies an annotated program π, denoted by

$$\Theta \models \pi$$

if for every thread in the global state Θ with active closure $(l : S : l', \tau)$, we have

$$\gamma \models @l$$

where γ denotes the configuration consisting of the global object structure of Θ and the local context τ. Roughly, a global state satisfies an annotated program if every thread satisfies the assertion annotating the statement of its active closure. We can now state the following theorem.

Theorem 1 (Soundness). *For any correctly annotated program π (possibly extended with auxiliary variables),*

$\Theta \models \pi$ and $\Theta \to \Theta'$ implies $\Theta' \models \pi$

Roughly, this theorem states the invariance of the assertions of a correctly annotated program. The proof involves a straightforward but tedious case analysis of the computation step.

Completeness

Conversely, we show completeness by proving the correctness of an extended program annotated with so-called reachability predicates. These predicates are introduced in [3] and [12] and adapted to (extended) multi-threaded Java programs as follows: Given a program we define for every interleaving point l the predicate $@l$ by

> $\gamma \models @l$
> if there exists a reachable global state Θ that realizes the object structure of γ and that contains a thread with an active closure $(l : S : l', \tau)$, where τ is the local context of γ.

A global state Θ is reachable if there exists a partial computation

$$\Theta_0 \rightarrow^* \Theta$$

starting from a fixed initial global state Θ_0. Here $\rightarrow^*$ denotes the reflexive, transitive closure of $\rightarrow$.

Using the encoding techniques of [14] it can be shown that the above reachability predicates can be expressed in the assertion language. Of particular interest to note here is that *pure* methods,. i.e., methods that do not affect the program state, in assertions greatly facilitates such an encoding.

By a straightforward, though tedious, induction on the length of the computation we can prove that a program annotated with the above reachability predicates is locally correct. The main case of interest is a proof of the verification condition

$$\models (@l)\sigma \rightarrow WP(x = \text{return}, @l')$$

for validating the postcondition of a method call $x = e_0.m(e_1, \ldots, e_n)$. The label l marks the end of the method body of m and l' the termination of the call. The context switch and parameter passing are modeled by the substitution σ (as described above). In order to validate this verification condition we extend every method definition with an additional formal parameter which stores the local context of the caller and an additional parameter for passing the label identifying the call. The local context of the caller, i.e., the values of its local variables, are stored in an array. These additional formal parameters we denote by 'con' and 'lab' (run methods contain these variables as local variables). In order to initialize the local context of the callee (to be passed in subsequent calls), we add to each method the following initialization:

> Objects [ ] mycontext;
> mycontext=new Objects[n+1];
> mycontext[0]=u_1;
> $\vdots$
> mycontext[n-1]=u_n;
> mycontext[n]=con;
> mycontext[n+1]=lab;

Here $u_1, \ldots, u_n$ are the formal parameters of the method (as specified by the given program). A call $x = e_0.m(e_1, \ldots, e_n)$ is extended by

$$x = e_0.m(e_1, \ldots, e_n, \text{mycontext}, l')$$

(the label l' marks its termination). Note that in Java arrays are objects, e.g., the actual parameter 'mycontext' is an object which refers to an array.

The additional parameters ensure that the predicate $(@l)\sigma$ indeed describes the return of the method m to the given call (σ is also extended with these new parameters). To see this, let

$$\gamma \models (@l)\sigma$$

By the usual substitution lemma of the logic underlying the assertion language, this is equivalent to

$$\gamma \models WP(\bar{u} = \bar{e}, @l)$$

where $\bar{u} = \bar{e}$ denotes the sequence of assignments corresponding to the substitution σ. Let

$$\langle \bar{u} = \bar{e}, \gamma \rangle \rightarrow \gamma'$$

that is, γ' is the resulting configuration of the execution of the statement $\bar{u} = \bar{e}$ in γ. It follows that

$$\gamma' \models @l$$

Note that the local context τ' of the configuration γ' in fact denotes the result of switching the context from the caller back to the callee.

By the above definition of the reachability predicates it follows that there exists a partial computation

$$\Theta_0 \rightarrow^* \Theta'$$

that realizes the object structure of γ' (which equals that of γ) and that contains an active closure (l, τ') which marks the termination of the body of m.

From

$$\langle \bar{u} = \bar{e}, \gamma \rangle \rightarrow \gamma'$$

it follows immediately that

$$\tau'(\text{con}) = \tau(\text{mycontext}) \text{ and } \tau'(\text{lab}) = l'$$

So we know that this invocation of m has been called by the given call statement. More specifically, we know that Θ' contains a thread

$$\cdots (x = \text{return}; l' : S : l'', \tau)(l, \tau')$$

Let

$$\Theta' \rightarrow \Theta$$

be the computation step which models the context switch from callee to caller. That is, the closure (l, τ') is removed from the above call stack. Since Θ realizes the object structure of γ we conclude that

$$\gamma \models WP(x = \text{return}, @l')$$

Remains to show that the reachability predicates are interference free. More specifically, we have to show that for any interleaving points l and l', with l' marking the start of an atomic statement S, we have

$$\models (@l' \wedge @l \wedge \text{thread!} = \text{thread}') \rightarrow WP(S, @l')$$

Roughly, this verification condition states that if one thread reaches l' and if another thread reaches l, then l' is still reachable after the execution of the statement S. This follows trivially if there exists one computation where both threads reach l' and l at the same time. However, in general this is not the case, e.g., the reachability of l' may require a *scheduling* of the threads which is incompatible with the reachability of l.

*Example 10 (***Scheduling).** Consider a thread class with the following method

run() {
if race() $\{l_1 : S_1\}$ else $\{l_2 : S_2\}\}$

The labels l_1 and l_2 denote the start of the 'then' and the 'else' branch, respectively. The synchronized method 'race' is defined by

race() {
u=b;
if b==true { b=false };
return u }

where 'u' is a local variable and 'b' is a static variable. which is initially true. Let the main method of the program initialize 'b' to 'true' and then simply create two instances of the thread class and start their run methods. Let τ be a local context such $\tau(\text{thread})$ and $\tau(\text{thread}')$ are two different instances of the thread class. Let $t = \tau(\text{thread})$ or $t = \tau(\text{thread}')$. Clearly, in both cases there exists a reachable global state Θ in which 'b=false' holds and which contains the active closure $(l_1 : S_1, \tau')$, where $\tau'(\text{thread}) = t$. But there exists no reachable global state in which *both* threads are at l_1 at the same time.

Therefore we introduce a static auxiliary variable 'sched' which records the scheduling of the threads. We introduce this variable as a vector of objects in the class containing the main method. Every read or write operation which involves access to the global object structure is extended with an update which adds the identity of the executing thread.

Example 11. Returning to the above example, we note that this additional scheduling information implies that

$$\models (@l_1' \wedge @l_1 \wedge \text{thread!} = \text{thread}') \rightarrow \text{false}$$

(the predicate $@l_1'$ refers to the thread denoted by the fresh local variable 'thread$'$'). Note that $@l_1'$ implies that 'sched' stores the thread denoted by 'thread$'$' first, whereas $@l_1$ stores the thread denoted by the distinguished local variable 'thread' first.

Note that the interleaving of the local computations of the threads, i.e., the computations which only access the local context of the active closures and which do not access the global object structures (the static variables and the instance variables of the existing objects), does not affect the global computation. More specifically, the variable 'sched' enforces the following confluence property of the global transition relation.

Lemma 1 (Confluence). *Let π be a multi-threaded Java program extended with the auxiliary variable 'sched' for recording the scheduling of threads, as described above. Furthermore, let the object structures of the global states Θ and Θ' assign the same value to the variable 'sched'. It follows that if*

$$\Theta_0 \rightarrow^* \Theta \text{ and } \Theta_0 \rightarrow^* \Theta'$$

then there exists a global state Θ'' such that

$$\Theta \rightarrow^* \Theta'' \text{ and } \Theta' \rightarrow^* \Theta''$$

Furthermore, these partial compotations only consist of local compouations steps which do not involve (read or write) access to the global object structure.

We can now prove the following theorem which states that the reachability predicates are interference free.

Theorem 2. *For any labeled statements $l : S$ and $l' : S'$ of a program extended with the auxiliary variable 'sched' we have*

$$\models (@l' \wedge @l \wedge \text{thread}'! = \text{thread}) \rightarrow WP(S, @l')$$

Proof. Let

$$\gamma \models @l' \wedge @l \wedge \text{thread}! = \text{thread}'$$

By definition of the reachability predicates $@l'$ and $@l$ there exists partial computations

$$\Theta_0 \rightarrow^* \Theta \text{ and } \Theta_0 \rightarrow^* \Theta'$$

starting from a fixed initial global state Θ_0, such that $(l : S, \tau)$ is the active closure of the thread $\tau(\text{thread})$, whereas the $(l' : S', \tau')$ is the active closure of $\tau(\text{thread}')$. Here τ denotes the local context of the configuration γ and $\tau'(u) = \tau(u')$, for every local variable (remember that primed local variables are introduced in order to avoid name clashes between the local variables of $@l$ and $@l'$). Furthermore, the global object structure of γ is realized in both the global states Θ and Θ'. The auxiliary variable 'sched' thus has the same value in the global object structures of Θ and Θ'. By the above lemma, there exists a global state Θ'' which can be reached from both Θ and Θ' by local computations only. But then we can also backtrack the local computation steps of the two threads (denoted by $\tau(\text{thread})$ and $\tau(\text{thread}')$) and obtain a reachable global state in which $\tau(\text{thread})$ is about to execute S and $\tau(\text{thread}')$ the statement S'. Clearly, the thread denoted by $\tau(\text{thread}')$ is still about to execute S' in the global state which results from the execution of S by $\tau(\text{thread})$. It follows by definition of the reachability predicates that

$$\gamma' \models @l'$$

where γ' consists of the object structure resulting from the execution of S' (by $\tau(\text{thread})$) and the initial local context τ' (of $\tau(\text{thread}')$).

5 Conclusion and future work

In this paper we presented a sound and complete proof method for multi-threaded Java programs. The proof method distinguishes a local level which is based on a Hoare logic for the sequential flow of control of (recursive) method calls within one thread and a global level which deals with interference between threads. The formal justification of the proof method is based on a formal semantics of Java programs annotated with assertions.

The proof method incorporates the use of auxiliary variables. These variables are used to capture specific aspects of the flow of control. Of particular interest is their use introduced in this paper as additional formal parameters to describe the sequential flow of control of

(recursive) method calls within one thread. This use allows a complete characterization of method calls in a multi-threading context. More specifically, in this paper we introduced such a characterization in terms of the reachability predicates instead of the strongest postcondition as is used in the seminal completeness proof of Gorelick ([9]) for recursive procedure calls in a sequential context (see also [2]).

In general, auxiliary variables can be used to extend the proof method in a systematic manner to other mechanisms like synchronized methods, wait and notify methods, and further details of the underlying memory model as described in [10].

Future work The main challenge is integrated tool support for the annotation of multi-threaded Java programs with assertions (as provided by [11]), the automatic generation of the verification conditions and (semi)automated validation of these conditions using theorem proving (as provided by [7, 5]).

References

1. E. Abraham, F.S. de Boer, W.P. de Roever, and M. Steffen. An assertion-based proof system for mutithreaded Java. Theoretical Computer Science, Vol. 331, 2005.
2. K.R. Apt: Ten years of Hoare logic: a survey — part I. ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, October 1981, pp. 431–483.
3. K.R. Apt. Formal justification of a proof system for Communicating Sequential Processes. Journal of the ACM, Vol. 30, No. 1, January 1983, pp. 197–216.
4. K.R. Apt, N. Francez and W. P. de Roever. A proof system for Communicating Sequential Processes. ACM Transactions on Programming Languages and Systems, 2:359-385, 1980.
5. B. Beckert, R. Hhnle, P. H. Schmitt (Eds.). Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer-Verlag, 2007.
6. F.S. de Boer. A WP-calculus for OO. Proceedings of Foundations of Software Science and Computation Structures, FOSSACS'99, LNCS Vol. 1578, 1999.
7. The Extended Static Checker for Java (ESC/Java).
   URL: http://secure.ucd.ie/products/opensource/ESCJava2.
8. R.T. Gerth and W.-P. de Roever. Proving monitors revisited: A first step towards verifying object oriented systems. Fundamenta informaticae IX, North-Holland, p. 371-400, 1986.
9. G. A. Gorelick . A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Department of Computer Science, University of Toronto, 1975.
10. J. Manson, W. Pugh and S.V. Adve. The Java memory model. Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005. ACM 2005.
11. The Java Modeling Language (JML).
    URL of the JML home page: http://www.cs.iastate.edu/ leavens/JML.
12. S. Owicki. A consistent and complete deductive system for the verification of parallel programs. Proceedings of the eighth annual ACM symposium on Theory of computing. ACM Press, 1976.
13. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. Acta Informatika, 6:319-340, 1976.
14. J.V. Tucker and J.I. Zucker: Program Correctness over Abstract Data Types, with Error-State Semantics. CWI Monograph Series, Vol. 6, Centre for Mathematics and Computer Science/North-Holland, 1988.