Verification for Java's Monitor Concept

August 14, 2002

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

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

Abstract. Besides the features of a class-based object-oriented language, Java integrates concurrency via its thread-classes, allowing for a *multithreaded* flow of control. The concurrency model includes *shared-variable* concurrency via instance variables, *coordination* via reentrant synchronization monitors, *synchronous message passing,* and dynamic *thread creation.*
To reason about multithreaded programs, we introduce in this paper an *assertional proof method* for safety properties for Java$_{MT}$ ("*Multi-Threaded Java*"), a small concurrent sublanguage of Java, covering the mentioned concurrency issues as well as the object-based core of Java, i.e., object creation, side effects, and aliasing, but leaving aside inheritance and subtyping. We show soundness and relative completeness of the proof method.

Table of Contents

1 Introduction

The semantical foundations of *Java* [11] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [3, 19, 8]). The research concerning *Java*'s proof theory mainly concentrated on various aspects of *sequential* sublanguages (see e.g. [14, 22, 18]). Besides standard object-oriented features, *Java*integrates mulithreading and monitor synchronization.

This paper presents a proof system for *multithreaded* sublanguage of *Java*. The work extends [2] in that besides cuncrurrency the issues of *monitor synchronization* are captured. We introduce an abstract programming language $Java_{MT}$, a subset of *Java*, featuring dynamic object creation, method invocation, object references with aliasing, and specifically concurrency and *Java*'s monitor discipline.

The assertional proof system for verifying safety properties of $Java_{MT}$ is formulated in terms of *proof outlines* [17], i.e., of programs augmented by auxiliary variables and with Hoare-style assertions [10, 12] associated with every control point.

The behavior of a $Java_{MT}$ program results from the concurrent execution of method bodies, that can interact by shared-variable concurrency, synchronous message passing for method calls, and object creation. In order to capture these features in a modular way, the assertional logic and the proof system are formulated in two levels, a local and a global one. The local assertion language describes the internal object behavior. The global behavior, including the communication topology of the objects, is expressed in the global language. As in the Object Constraint Language (OCL) [23], properties of object-structures are described in terms of a navigation or dereferencing operator.

The execution of a single method body in isolation is captured by standard *local correctness* conditions.

To support a clean interface between internal and external 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, but not across object boundaries, and can therefore be handled on the local level, as well. A further healthy effect of disallowing references to external instance variables is that it reduces the potential of interference considerably, which means much less proof obligations generated by the proof system. The *interference freedom test* [17, 15] formulates the corresponding verification conditions. It has especially to accommodate for reentrant code and the specific synchronization mechanism.

Affecting more than one instance, synchronous message passing and object creation can be established locally only relative to assumptions about the communicated values. These assumptions are verified in the *cooperation test* on the global level. The communication can take place within a single object or between different objects. As these two cases cannot be distinguished syntactically, our cooperation test combines elements from similar rules used in [7] and in [15] for CSP.

Overview This paper is organized as follows. Section 2 defines the syntax and semantics of *Java$_{MT}$*. After introducing the assertion language in Section 3, the main Section 4.2 presents the proof system.

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

In this section we introduce the language *Java$_{MT}$* (*"Multi-Threaded Java"*). We start with highlighting the features of *Java$_{MT}$* and its relationship to full *Java*, before formally defining its abstract syntax and semantics.

2.1 Introduction

Java$_{MT}$ is a multithreaded sublanguage of *Java*. Programs, as in *Java*, are given by a collection of classes containing instance variable and method declarations. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation,* i.e., synchronous message passing. As we focus on a proof system for the concurrency aspects of *Java*, all classes in *Java$_{MT}$* are thread classes in the sense of *Java*: Each class contains a start-method that can be invoked only once for each object, resulting in a new thread of execution. The new thread starts to execute the start-method of the given object while the initiating thread continues its own execution.

As a mechanism of concurrency control, methods can be declared as *synchronized.* Each object has a *lock* which can be owned by at most one thread. Synchronized methods of an object can be invoked only by a thread which owns the lock of the object. If the thread does not own the lock, it has to wait until the lock gets free, i.e., till in the object all current synchronized method executions terminate. If a thread owns the lock of an object, it can recursively invoke several synchronized methods of that object. This corresponds to the notion of re-entrant monitors and eliminates the possibility that a single thread deadlocks itself on an object's synchronization barrier.

Besides mutual exclusion, via the lock-mechanism for synchronized methods, 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 give the lock free by invoking wait on the given object. The blocked thread can be reactivated by another thread via the object's notify method; the reactivated thread must re-apply for the lock before it may continue its execution. The method notifyAll, finally, generalizes notify in that it notifies all threads blocked on the object.

2.2 Abstract syntax

Similar to *Java*, the language *Java$_{MT}$* is strongly typed and supports class types and primitive, i.e., non-reference types. As built-in primitive types we restrict to integers and booleans, denoted by Int and Bool. Besides the built-in types, the set of user-definable types is given by a set of class names $\mathcal{C}$, with typical element c. Furthermore, the language allows pairs of type $t_1 \times t_2$ and sequences of type $\mathsf{list}\, t$. Side-effect expressions without a value, i.e., methods without a return value, will get the type Void. Thus the set of all types $\mathcal{T}$ with typical element t is given by the following abstract grammar:

$$t ::= \mathsf{Void} \mid \mathsf{Int} \mid \mathsf{Bool} \mid c \mid t \times t \mid \mathsf{list}\, t$$

For each type, the corresponding value domain is equipped with a standard set F of operators with typical element f. Each operator f has a unique type $t_1 \times \cdots \times t_n \to t$ and a fixed interpretation f, where constants are operators of zero arity. Apart from the standard repertoire of arithmetical and boolean operations, the set F of operators also contains operations on tuples and sequences like projection, concatenation, etc.

Since *Java$_{MT}$* is strongly typed, all program constructs of the abstract syntax —variables, expressions, statements, methods, classes— are silently assumed to be well-typed, i.e., each method invoked on an object must be supported by the object, the types of the formal and actual parameters of the invocation must match, etc. In other words, we work with a type-annotated abstract syntax where we omit the explicit mentioning of types when no confusion can arise. As the static relationships between classes are orthogonal to multithreading aspects, we ignore in *Java$_{MT}$* the issues of *inheritance*, and consequently subtyping, overriding, and late-binding. For simplicity, we neither allow method *overloading*, i.e., we require that each method name is assigned a unique list of formal parameter types and a return type. In short, being concerned with the verification of the run-time behavior, we assume a simple *monomorphic* type discipline for *Java$_{MT}$*.

For variables, we notationally distinguish between *instance* and *local* variables. Instance variables are always assumed to be private in *Java$_{MT}$*. They hold the state of an object and exist throughout the object's lifetime. Local variables are stack-allocated; they play the role of formal parameters and variables of method definitions and only exist during the execution of the method to which they belong. The set of variables $Var = IVar \,\dot\cup\, TVar$ with typical element y is given as the disjoint union of the instance and the local variables. Var^t denotes the set of all variables of type t, and correspondingly for $IVar^t$ and $TVar^t$. As we assume a monomorphic type discipline, $Var^t \cap Var^{t'} = \emptyset$ for distinct types t and t'. We use $x, x', x_1, \ldots$ as typical elements from $IVar$, and $u, v, u', v_1, \ldots$ as typical elements from $TVar$.

Basic constructs The syntax is summarized in the Tables 1 and 2. Besides using instance and local variables, *expressions* $exp \in Exp$ are built from this, nil, and from subexpressions using the given operators. We will use e as typical element for expressions, and $Exp^t_{m,c}$ to denote the set of well-typed expressions of type t in method $m \in \mathcal{M}$ of class $c \in \mathcal{C}$, where $\mathcal{M}$ is an infinite set of method names containing main, start, run, wait, notify, and notifyAll. The expression this is used for self-reference within an object, and nil is a constant representing an empty reference.

As *statements* $stm \in Stm$, we allow assignments, object creation via new, method invocation and standard control constructs like sequential composition, conditional statements, and iteration. Furthermore, we write ϵ for the empty statement. We refer by $Stm_{m,c}$ to the set of statements in method m of class c.

A *method* definition consists of a method name m, a list of formal parameters $u_1, \ldots, u_n$, and a method body $body_{m,c}$ of the form $stm; \mathsf{return}\ e_{ret}$. The set $Meth_c$ contains the methods of class c. To simplify the proof system we require that method bodies are terminated by a single return statement return e_{ret}, giving back the control and possibly a return value. Additionally, methods are decorated by a modifier *modif* distinguishing between *non-synchronized* and *synchronized* methods.[3] We use $sync(c, m)$ to state that method m in class c is synchronized. In the sequel we also refer to statements in the body of a synchronized method as being synchronized. A *class* is defined by its name c and its methods, whose names are assumed to be distinct.

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

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

A *program,* finally, is a collection of class definitions having different class names, where $class_{\mathsf{main}}$ is the entry point of the program execution. This class specifically contains a main-method $meth_{\mathsf{main}}$ without return value. We call its body, written as $body_{\mathsf{main}}$, the main statement of the program.

The set of *instance* variables $IVar_c$ of a class c is implicitly given by the set of all instance variables occurring in that class. Correspondingly for methods, the set of local variables $TVar_{m,c}$ of a method m in class c is given by the set of all local variables occurring in that method.

[3] Java does not have the "non-synchronized" modifier: methods are non-synchronized by default.

Thread coordination Besides the user-definable methods, we consider a number of predefined ones with fixed meaning (cf. Table 2). The methods run and start serve to launch a thread, where run contains the actual body of the thread, and start is a "wrapper" around run which allows the initiator of the thread to asynchronously continue after setting off the execution of the new thread. The methods wait, notify, and notifyAll can be used to block and reactivate threads at the object boundary, as informally described in Section 2.1. Java's `Thread` class additionally support methods for suspending, resuming, and stopping a thread, but they are deprecated and thus not considered here.

$$\begin{aligned}
meth_{\mathsf{run}} &::= modif\ \mathsf{run}()\{\ stm; \mathsf{return}\ \} \\
meth_{\mathsf{start}} &::= \mathsf{nsync\ start}()\{\ \mathsf{this.run}(); \mathsf{receive}; \mathsf{return}\ \} \\
meth_{\mathsf{wait}} &::= \mathsf{nsync\ wait}()\{\ ?\mathsf{signal}; \mathsf{return}_{getlock}\ \} \\
meth_{\mathsf{notify}} &::= \mathsf{nsync\ notify}()\{\ !\mathsf{signal}; \mathsf{return}\ \} \\
meth_{\mathsf{notifyAll}} &::= \mathsf{nsync\ notifyAll}()\{\ !\mathsf{signal\_all}; \mathsf{return}\ \} \\
meth_{\mathsf{main}} &::= \mathsf{nsync\ main}()\{\ stm; \mathsf{return}\ \}
\end{aligned}$$

Table 2. Java$_{MT}$ abstract syntax: predefined methods

Restrictions Besides the mentioned simplifications on the type system, we impose for technical reasons the following restrictions on the language: Statements of the form $u := e_0.m(\vec{e})$ are used as syntactic sugar for the composite statement $e_0.m(\vec{e}); \mathsf{receive}\ u$. We require that method invocation and object creation statements contain only local variables, i.e., that none of the expressions $e_0, \dots, e_n$ contains instance variables, and that formal parameters do not occur on the left-hand side of assignments. This restriction implies that during the execution of a method the values of the actual and formal parameters 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.

2.3 Semantics

In this section, we define the *operational semantics* of Java$_{MT}$, especially, the mechanisms of multithreading, dynamic object creation, method invocation, and coordination via synchronization. After introducing the semantic domains, we describe states and configurations in the following section. The operational semantics is presented in Section 2.3.2 by transitions between program configurations.

2.3.1 States and configurations To give semantics to the expressions, we first fix the domains Val^t of the various types t. Let Val^{Int} and Val^{Bool} denote the set of integers and booleans, $Val^{\mathsf{list}\,t}$ be finite sequences over values from Val^t, and let $Val^{t_1 \times t_2}$ stand for the product $Val^{t_1} \times Val^{t_2}$. For class names $c \in \mathcal{C}$, the set Val^c with typical elements $\alpha, \beta, \ldots$ denotes an infinite set of *object identifiers,* where the domains for different class names are assumed to be disjoint. For each class name c, $nil^c \notin Val^c$ represents the value of nil in the corresponding type. In general we will just write nil, when c is clear from the context. We define Val^c_{nil} as $Val^c \dot\cup \{nil^c\}$, and correspondingly for compound types. The set of all possible non-nil values $\bigcup_t Val^t$ is written as Val, and Val_{nil} denotes $\bigcup_t Val^t_{nil}$.

Let $Init : Var \to Val$ be a function assigning the initial value of type t to each variable $y \in Var^t$, i.e., nil, $false$, and 0 for class, boolean, and integer types, respectively, and analogously for compound types, where sequences are initially empty. We define $\mathsf{this} \notin Var$, such that the self-reference is not in the domain of $Init$.

The configuration of a program consists of all currently executing threads together with the set of existing objects and the values of their instance variables. Before formalizing the global configurations of a program, we define local states and local configurations. In the sequel we in general identify the occurrence of a statement in a program with the statement itself.

A *local state* $\tau \in \Sigma_{loc}$ of a thread holds the values of its local variables and is modeled as a partial function of type $TVar \rightharpoonup Val_{nil}$. For a class c and a method m of c we use the notation $\tau^{m,c}$ to refer to a local state of a thread executing method m of an instance of class c. The initial local state τ_{init} or $\tau^{m,c}_{init}$ assigns to each local variable u from $dom(\tau)$ the value $Init(u)$.

A *local configuration* (α, τ, stm) of a thread executing within an object $\alpha \neq nil$ specifies, in addition to its local state τ, its point of execution represented by the statement stm. A *thread configuration* ξ is a stack of local configurations $(\alpha_0, \tau_0, stm_0)(\alpha_1, \tau_1, stm_1) \ldots (\alpha_n, \tau_n, stm_n)$, representing the chain of method invocations of the given thread. We write $\xi \circ (\alpha, \tau, stm)$ for pushing a new local configuration onto the top of the stack.

The state of an object is characterized by its *instance state* $\sigma_{inst} \in \Sigma_{inst}$ of type $IVar \dot\cup \{\mathsf{this}\} \rightharpoonup Val_{nil}$ which assigns values to the self-reference this and to instance variables.[4] For a class c we write σ^c_{inst} to denote instance states assigning values to the instance variables of class c, i.e., σ^c_{inst} is of type $IVar_c \dot\cup \{\mathsf{this}\} \to Val_{nil}$. The initial instance state σ^{init}_{inst} or $\sigma^{c,init}_{inst}$ assigns a value from Val^c to this, and to each of its remaining instance variables x the value $Init(x)$. The semantics will maintain $\sigma^c_{inst}(\mathsf{this}) \in Val^c$ as invariant.

A *global state* $\sigma \in \Sigma$ stores for each currently *existing* object its instance state and is modeled as a partial function of type $(\bigcup_{c \in \mathcal{C}} Val^c) \rightharpoonup \Sigma_{inst}$. The set of existing objects of type c in a state σ is given by $dom^c(\sigma)$, and $dom^c_{nil}(\sigma)$ is defined by $dom^c(\sigma) \cup \{nil^c\}$. For the built-in types Int and Bool we define dom^t and dom^t_{nil}, independently of σ, as the set of pre-existing values Val^{Int}

[4] In Java, this is a "final" instance variable, which for instance implies, it cannot be assigned to.

and Val^{Bool}, respectively. For compound types, dom^t and dom^t_{nil} are defined correspondingly. We refer to the set $\bigcup_t dom^t$ by $dom(\sigma)$; $dom_{nil}(\sigma)$ denotes $\bigcup_t dom^t_{nil}$. The instance state of an object $\alpha \in dom(\sigma)$ is given by $\sigma(\alpha)$ with the invariant property $\sigma(\alpha)(\mathsf{this}) = \alpha$. We call an object $\alpha \in dom(\sigma)$ existing in σ, and we throughout require that, given a global state, no instance variable in any of the existing objects refers to a non-existing object, i.e., $\sigma(\alpha)(x) \in dom_{nil}(\sigma)$ for all $\alpha \in dom^c(\sigma)$. This will be an invariant of the operational semantics of the next section.

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. Analogously to the restriction on global states, we require that local configurations (α, τ, stm) in $\langle T, \sigma\rangle$ refer only to existing object identities, i.e., $\alpha \in dom(\sigma)$ and $\tau(u) \in dom_{nil}(\sigma)$ for all variables u from the domain of τ; again this will be an invariant of the operational semantics. In the following, we write $(\alpha, \tau, stm) \in T$ if there exists a local configuration (α, τ, stm) within one of the execution stacks of T.

Expressions $e \in Exp^t_{m,c}$ are evaluated with respect to an *instance local* state $(\sigma^c_{inst}, \tau^{m,c}) \in \Sigma_{inst} \times \Sigma_{loc}$. The semantic function $[\![\_]\!]_{\mathcal{E}} : (\Sigma_{inst} \times \Sigma_{loc}) \rightarrow (Exp \rightharpoonup Val_{nil})$ shown in Table 3 evaluates expressions containing variables from $dom(\sigma_{inst}) \cup dom(\tau)$ in the context of an instance local state (σ_{inst}, τ): Instance variables x and local variables u are evaluated to $\sigma_{inst}(x)$ and $\tau(u)$, respectively; this evaluates to $\sigma_{inst}(\mathsf{this})$, and nil has the *nil*-reference as value, where compound expressions are evaluated by homomorphic lifting.

$$\begin{aligned}
[\![x]\!]^{\sigma_{inst},\tau}_{\mathcal{E}} &= \sigma_{inst}(x)\\
[\![u]\!]^{\sigma_{inst},\tau}_{\mathcal{E}} &= \tau(u)\\
[\![\mathsf{this}]\!]^{\sigma_{inst},\tau}_{\mathcal{E}} &= \sigma_{inst}(\mathsf{this})\\
[\![\mathsf{nil}]\!]^{\sigma_{inst},\tau}_{\mathcal{E}} &= nil\\
[\![\mathsf{f}(e_1,\dots,e_n)]\!]^{\sigma_{inst},\tau}_{\mathcal{E}} &= f([\![e_1]\!]^{\sigma_{inst},\tau}_{\mathcal{E}},\dots,[\![e_n]\!]^{\sigma_{inst},\tau}_{\mathcal{E}})
\end{aligned}$$

Table 3. Expression evaluation

For a local state τ, we denote by $\tau[u \mapsto v]$ the local state which assigns the value v to u and agrees with τ on the values of all other variables. The semantic update $\sigma_{inst}[x \mapsto v]$ of instance states is defined analogously. Correspondingly for global states, $\sigma[\alpha.x \mapsto v]$ denotes the global state which results from σ by assigning v to the instance variable x of object α. We use these operators analogously for simultaneously setting the values of vectors of variables. We use $\tau[\vec{y} \mapsto \vec{v}]$ also for arbitrary variable sequences, where instance variables are untouched, i.e., $\tau[\vec{y} \mapsto \vec{v}]$ is defined by $\tau[\vec{u} \mapsto \vec{v}_u]$, where $\vec{u}$ is the sequence of the local variables in $\vec{y}$ and $\vec{v}_u$ the corresponding value sequence. Similarly, for instance states, $\sigma_{inst}[\vec{y} \mapsto \vec{v}]$ is defined by $\sigma_{inst}[\vec{x} \mapsto \vec{v}_x]$ where $\vec{x}$ is the sequence of the instance variables in $\vec{y}$ and $\vec{v}_x$ the corresponding value sequence. The semantics of $\sigma[\alpha.\vec{y} \mapsto \vec{v}]$ is analogous. Finally for global states, $\sigma[\alpha \mapsto \sigma^c_{inst}]$ equals σ except on α; note that in case $\alpha \notin dom^c(\sigma)$, the operation extends the set of existing objects by α, which has its instance state initialized to σ^c_{inst}.

2.3.2 Operational semantics The operational semantics of Java$_{MT}$ is given inductively by the rules of Tables 4 and 5 as transitions between global configurations. Before having a closer look at the semantical rules for the transition relation $\longrightarrow$, let us start by definining the starting point of a program. The initial configuration $\langle T_0, \sigma_0 \rangle$ of a program satisfies the following: $T_0 = \{(\alpha, \tau^{\mathsf{main},c}_{init}, body_{\mathsf{main}})\}$, where c is the main class, and $\alpha \in Val^c$. Moreover, $dom(\sigma_0) = \{\alpha\}$ and $\sigma_0(\alpha) = \sigma^{c,init}_{inst}[\mathsf{this} \mapsto \alpha]$.

In *Java*, the main method of a program is static. Since Java$_{MT}$ does not have static methods and variables, we define the initial configuration as having a single initial object in which an initial thread starts to execute the main-method.

We call a configuration $\langle T, \sigma \rangle$ of a program *reachable* iff there exists a computation $\langle T_0, \sigma_0 \rangle \longrightarrow^* \langle T, \sigma \rangle$ such that $\langle T_0, \sigma_0 \rangle$ is the initial configuration of the program and $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$. A local configuration $(\alpha, \tau, stm) \in T$ is *enabled* in $\langle T, \sigma \rangle$, if the statement stm can be executed at the current point, i.e., if there is a computation step $\langle T, \sigma \rangle \rightarrow \langle T', \sigma' \rangle$ executing stm in the local state τ and object α.

Assignments to instance or local variables update the corresponding state component, i.e., either the instance state or the local state (cf. rules $\textsc{Ass}_{inst}$ and $\textsc{Ass}_{loc}$). Object creation by $u := \mathsf{new}^c$ as shown in rule $\textsc{New}$ creates a new object of type c with a fresh identity stored in the local variable u, and initializes its instance variables. Invoking a method extends the call chain by a new local configuration (cf. rule $\textsc{Call}$). After initializing the local state and passing the parameters, the thread begins to execute the method body. The condition $sync(c, m) \rightarrow \neg owns(T, \beta)$ expresses that a synchronized method of an object can be invoked by a thread only if no other threads holds its lock, i.e., if the lock is free or if the executing thread already owns it.

Threads being blocked or waiting on an object, though, temporarily relinquish the lock. Formally, the wait set $wait(T, \alpha)$ of an object is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?\mathsf{signal}; stm)$. Analogously, we will need the set $notified(T, \alpha)$ of threads that have been notified and trying to get hold of the lock again: It is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, \mathsf{return}_{getlock})$.

With these definitions, the predicate $owns(\xi, \beta)$ is true iff there exists a $(\beta, \tau, stm) \in \xi$ with stm synchronized and $\xi \notin wait(T, \beta) \cup notified(T, \beta)$. The definition is used analogously for sets of threads. An invariant of the semantics is that at most one thread can own the lock of an object at a time, i.e., for all reachable $\langle T, \sigma \rangle$, for all ξ and ξ' in T and $\alpha \in dom^c(\sigma)$, $owns(\xi, \alpha)$ and $owns(\xi', \alpha)$ imply $\xi = \xi'$.

When returning from a method call (cf. rule RETURN), the callee evaluates its return expression and passes it to the caller which subsequently updates its local state. The method body terminates its execution and the caller can continue.

We elide the rules for the remaining sequential constructs —sequential composition, conditional statement, and iteration— as they are standard.

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

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

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

$$\frac{\begin{array}{c} m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\} \qquad \mathit{modif}\ m(\vec{u})\{\ \mathit{body}\ \} \in \mathit{Meth}_c \\ \beta = [\![e_0]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in \mathit{dom}^c(\sigma) \qquad \tau' = \tau_{init}^{m,c}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau}] \qquad \mathit{sync}(c,m) \to \neg \mathit{owns}(T,\beta) \end{array}}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, e_0.m(\vec{e}); \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathit{stm}) \circ (\beta, \tau', \mathit{body})\}, \sigma\rangle}\ \textsc{Call}$$

$$\frac{\tau'' = \tau[u_{ret} \mapsto [\![e_{ret}]\!]_{\mathcal{E}}^{\sigma(\beta),\tau'}]}{\langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau, \mathsf{receive}\ u_{ret}; \mathit{stm}) \circ (\beta, \tau', \mathsf{return}\ e_{ret})\}, \sigma\rangle \longrightarrow \langle T \,\dot\cup\, \{\xi \circ (\alpha, \tau'', \mathit{stm})\}, \sigma\rangle}\ \textsc{Return}$$

Table 4. Operational semantics (1)

The remaining rules of Table 5 handle *Java*$_{MT}$'s methods for thread manipulation. The start method brings a new thread into being (cf. rule CALL$_{start}$), thereby initializing the first activation record of a new stack. Only the first invocation of the start-method has this effect. This is captured by the predicate *started* which holds for a global configuration T and an instance α iff there exists a stack $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n) \in T$ such that $\alpha = \alpha_0$. Further invocations of the start-method are without effect (cf. rule CALL$^{skip}_{start}$).[5] A thread ends its lifespan by arriving at the end of its earliers local configuration (cf. rule RETURN$_{start}$), that is by returning from the initial invocation of the main-method or from a start-method.[6] Note that, since the initial thread begins its

[5] In *Java* an exception is thrown if the thread is already terminated.

[6] The worked-off local configuration (α, τ, ϵ) is kept in the global configuration to ensure that the thread of α cannot be started twice.

execution in the initial object, according to the definition of the *started* predicate, the start-method of the initial object cannot be invoked.

The remaining three methods offer typical monitor synchronisation mechanism at the object boundary, whose calls are descibed in rule $\textsc{Call}_{monitor}$. In all three cases it is necessary that the caller owns the lock of the object in question. If not, the caller will deadlock, as, once devoid of the lock, the caller stops and will never obtain it. In contrast, the successful call of synchronised methods as formalized by rule $\textsc{Call}$ of Table 4 depends contra-positively on the non-ownership of the lock by the rest of the program, which of course changes if another thread gives it free again. In Java, invoking a monitor method without owning the lock raises an exception, which terminates the culprit thread, but let the rest of the program continue. Insofar, our model is faithful with 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 α's *wait set* (cf. rule $\textsc{Call}_{monitor}$). In our formalization, this is indicated in that the thread is about to execute the statement ?signal after successful invokation of the wait method. Remember that according to the predicate *owns* the thread releases the lock thereby.

After having itself put on ice, the thread awaits notification to be reactivated by another thread who invokes the notify() method of the object. It is required that the notifier must own the lock of the object in question. The !signal-statement in the above method thus reactivates a thread waiting for notification on the given object (cf. rule $\textsc{Signal}$). It reactivates one of the blocked threads at least insofar as it is given the chance to re-apply for the lock: According to rule $\textsc{Return}_{wait}$, the receiver can continue after notification in executing $\mathsf{return}_{getlock}$ only if the lock is free. Note that the notifier does not automatically hand-over the lock to the one being notified but continues to own it. This behavior is know as *signal-and-continue* monitor discipline [5].

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

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.

To be able to argue about communication histories, represented as lists of objects, we add the type Object as the supertype of all classes into the assertion

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

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

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

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

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

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

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

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

Table 5. Operational semantics (2)

language. Note that we allow this type solely in the assertion language, but not in the programming language, thus preserving the assumption of monomorphism.

After fixing the syntax of the assertions in the next section, we define its semantics and provide basic substitution properties.

3.1 Syntax

In the language of assertions, we introduce a countably infinite set $LVar$ of well-typed *logical variables* with typical element z, where we assume that instance variables, local variables, and this are not in $LVar$. Logical variables are used for quantification in both the local and the global language. Besides that, they are used as free variables to represent local variables in the global assertion language: To express a local property on the global level, each local variable in a given local assertion will be replaced by a fresh logical variable.

Table 6 defines the syntax of the assertion language. *Local expressions* $exp_l \in LExp$ are expressions of the programming language possibly containing logical variables. The set $LExp^t_{m,c}$ consists of all local expressions of type t in method m of class c, where $LExp^t$ is defined by $\bigcup_{m,c} LExp^t_{m,c}$. In abuse of notation, we use e, $e' \ldots$ not only for program expressions of Table 1, but also for typical elements of local expressions. *Local assertions* $ass_l \in LAss$, with typical elements $p, p', q, \ldots$, are standard logical formulas over boolean local expressions; local assertions in method m of class c form the set $LAss_{m,c}$. We allow three forms of quantification over the logical variables: Unrestricted quantification $\exists z(p)$ is solely allowed for integer and boolean domains, i.e., z is required to be of type Int or Bool. For reference types c, this form of quantification is not allowed, as for those types, the existence of a value dynamically depends on the *global* state, something one cannot speak about on the local level, or more formally: Disallowing unrestricted quantification for object types ensures that the value of a local assertion indeed only depends on the values of the instance and local variables, but not on the global state. Nevertheless, one can assert the existence of objects on the local level satisfying a predicate, provided one is explicit about the set of objects to range over. Thus, the restricted quantifications $\exists z \in e(p)$ or $\exists z \sqsubseteq e(p)$ assert the existence of an element, respectively, the existence of a subsequence of a given sequence e, for which a property p holds.

Global expressions $exp_g \in GExp$, with typical elements $E, E', \ldots$, are constructed from logical variables, nil, operator expressions, and qualified references $E.x$ to instance variables x of objects E. We write $GExp^t$ for the set of global expressions of type t. *Global assertions* $ass_g \in GAss$, with typical elements $P, Q \ldots$, are logical formulas over boolean global expressions. Unlike the local language, the meaning of the global one is defined in the context of a global state. Thus unrestricted quantification is allowed for all types and is interpreted to range over the set of *existing* values, i.e., the set of values $dom_{nil}(\sigma)$ in a global configuration $\langle T, \sigma \rangle$.

$exp_l ::= z \mid x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp_l, \ldots, exp_l)$	$e \in LExp$	local expressions
$ass_l ::= exp_l \mid \neg ass_l \mid ass_l \wedge ass_l$		
$\mid\ \exists z(ass_l) \mid \exists z \in exp_l(ass_l) \mid \exists z \sqsubseteq exp_l(ass_l)$	$p \in LAss$	local assertions
$exp_g ::= z \mid \mathsf{nil} \mid \mathsf{f}(exp_g, \ldots, exp_g) \mid exp_g.x$	$E \in GExp$	global expressions
$ass_g ::= exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists z(ass_g)$	$P \in GAss$	global assertions

Table 6. Syntax of assertions

3.2 Semantics

Next, we define the interpretation of the assertion language. The semantics is fairly standard, except that we have to cater for dynamic object creation when interpreting quantification.

Expressions and assertions are interpreted relative to a logical environment $\omega \in \Omega$, a partial function of type $LVar \rightharpoonup Val_{nil}$, assigning values to logical variables. We denote by $\omega[\vec{z} \mapsto \vec{v}]$ the logical environment that assigns $v_i \in Val_{nil}$ to z_i, and agrees with ω on all other variables. Similarly to local and instance state updates, the occurrence of instance variables in $\vec{z}$ is without effect. For a logical environment ω and a global state σ we say that ω refers only to values existing in σ, if $\omega(z) \in dom_{nil}(\sigma)$ for all $z \in dom(\omega)$. This property matches with the definition of quantification which ranges only over existing values and *nil*, and with the fact that in reachable configurations local variables may refer only to existing values or to *nil*. Correspondingly for local states, we say that a local state τ refers only to values existing in σ, if $\tau(u) \in dom_{nil}(\sigma)$ for all $u \in dom(\tau)$.

The semantic function $[\![\_]\!]_{\mathcal{L}}$ of type $(\Omega \times \Sigma_{inst} \times \Sigma_{loc}) \rightarrow (LExp \cup LAss \rightharpoonup Val_{nil})$ evaluates local expressions and assertions in the context of a logical environment ω and an instance local state (σ_{inst}, τ) (cf. Table 7). The evaluation function is defined for expressions and assertions that contain only variables from $dom(\omega) \cup dom(\sigma_{inst}) \cup dom(\tau)$. The instance local state provides the context for giving meaning to programming language expressions as defined by the semantic function $[\![\_]\!]_{\mathcal{E}}$; the logical environment evaluates logical variables. An unrestricted quantification $\exists z(p)$ is evaluated to true in the logical environment ω and instance local state (σ_{inst}, τ) if and only if there exists a value $v \in Val^t$ such that p holds in the logical environment $\omega[z \mapsto v]$ and instance local state (σ_{inst}, τ), where for the type t of z only Int or Bool is allowed. The evaluation of a restricted quantification $\exists z \in e(p)$ with $z \in LVar^t$ and $e \in LExp^{\mathsf{list}\,t}$ is defined analogously, where the existence of an element in the sequence is required. An assertion $\exists z \sqsubseteq e(p)$ with $z \in LVar^{\mathsf{list}\,t}$ and $e \in LExp^{\mathsf{list}\,t}$ states the existence of a subsequence of e for which p holds. In the following we also write $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $[\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true$. By $\models_{\mathcal{L}} p$, we express that $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ holds for arbitrary logical environments, instance states, and local states.

Since *global* assertions do not contain local variables and non-qualified references to instance variables, the global assertional semantics does not refer to

$$\begin{array}{rcl}
[\![z]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \omega(z) \\
[\![x]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \sigma_{inst}(x) \\
[\![u]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \tau(u) \\
[\![\mathsf{this}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & \sigma_{inst}(\mathsf{this}) \\
[\![\mathsf{nil}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & nil \\
[\![\mathsf{f}(e_1,\ldots,e_n)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & f([\![e_1]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau},\ldots,[\![e_n]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}) \\
([\![\neg p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = false) \\
([\![p_1 \wedge p_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![p_1]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true \text{ and } [\![p_2]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) \\
([\![\exists z(p)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau} = true \text{ for some } v \in Val) \\
([\![\exists z{\in}e(p)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![z{\in}e \wedge p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau}{=}true \text{ for some } v \in Val_{nil}) \\
([\![\exists z{\sqsubseteq}e(p)]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true) & \text{iff} & ([\![z{\sqsubseteq}e \wedge p]\!]_{\mathcal{L}}^{\omega[z\mapsto v],\sigma_{inst},\tau}{=}true \text{ for some } v \in Val_{nil})
\end{array}$$

Table 7. Local evaluation

instance local states but to global states. The semantic function $[\![\_]\!]_{\mathcal{G}}$ of type $(\Omega \times \Sigma) \rightharpoonup (GExp \cup GAss \rightharpoonup Val_{nil})$, shown in Table 8, gives meaning to global expressions and assertions in the context of a global state σ and a logical environment ω. To be well-defined, ω is required to refer only to values existing in σ, and the expression respectively assertion may only contain free variables from $dom(\omega) \cup dom(\sigma)$. Logical variables, nil, and operator expressions are evaluated analogously to local assertions. The value of a global expression $E.x$ is given by the value of the instance variable x of the object referred to by the expression E. The evaluation of an expression $E.x$ is defined only if E refers to an object existing in σ. Note that when 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. The semantics of negation and conjunction is standard. A quantification $\exists z(P)$ evaluates to true in a logical environment ω and global state σ if and only if P evaluates to true in the logical environment $\omega[z \mapsto v]$ and global state σ, for some value $v \in dom_{nil}(\sigma)$. Note that quantification over objects ranges over the set of *existing* objects and *nil*, only.

$$\begin{array}{rcl}
[\![z]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \omega(z) \\
[\![\mathsf{nil}]\!]_{\mathcal{G}}^{\omega,\sigma} & = & nil \\
[\![\mathsf{f}(E_1,\ldots,E_n)]\!]_{\mathcal{G}}^{\omega,\sigma} & = & f([\![E_1]\!]_{\mathcal{G}}^{\omega,\sigma},\ldots,[\![E_n]\!]_{\mathcal{G}}^{\omega,\sigma}) \\
[\![E.x]\!]_{\mathcal{G}}^{\omega,\sigma} & = & \sigma([\![E]\!]_{\mathcal{G}}^{\omega,\sigma})(x) \\
([\![\neg P]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega,\sigma} = false) \\
([\![P_1 \wedge P_2]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P_1]\!]_{\mathcal{G}}^{\omega,\sigma} = true \text{ and } [\![P_2]\!]_{\mathcal{G}}^{\omega,\sigma} = true) \\
([\![\exists z(P)]\!]_{\mathcal{G}}^{\omega,\sigma} = true) & \text{iff} & ([\![P]\!]_{\mathcal{G}}^{\omega[z\mapsto v],\sigma} = true \text{ for some } val \in dom_{nil}(\sigma))
\end{array}$$

Table 8. Global evaluation

For a global state σ and a logical environment ω referring only to values existing in σ we write $\omega, \sigma \models_{\mathcal{G}} P$ when P is true in the context of ω and σ. We write $\models_{\mathcal{G}} P$ if P holds for arbitrary global states σ and logical environments ω referring only to values existing in σ.

The verification conditions defined in the next section involve the following substitution operations: The standard capture-avoiding substitution $p[\vec{e}/\vec{y}]$ replaces in the local assertion p all occurrences of the given distinct variables $\vec{y}$ by the local expressions $\vec{e}$. We apply the substitution also to local expressions. The following lemma expresses the standard property of the above substitution, relating it to state-update. The relation between substitution and update formulated in the lemma asserts that $p[\vec{e}/\vec{y}]$ is the *weakest precondition* of p wrt. to the assignment. The lemma will be used for proving invariance of local assertions under assignments.

Lemma 1 (Local substitution). *For arbitrary logical environments ω and instance local states (σ_{inst}, τ) we have*

$$\begin{array}{rcl} [\![e'[\vec{e}/\vec{y}]]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} & = & [\![e']\!]_{\mathcal{L}}^{\omega,\sigma_{inst}[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}],\tau[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}]}\ , \textit{ and} \\ \omega,\sigma_{inst},\tau \models_{\mathcal{L}} p[\vec{e}/\vec{y}] & \textit{iff} & \omega,\sigma_{inst}[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}],\tau[\vec{y}\mapsto[\![\vec{e}]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau}] \models_{\mathcal{L}} p\,. \end{array}$$

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 [4]:

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

The substitution is extended to global assertions homomorphically. We use this substitution to express that a property defined in the global assertion language is invariant under assignments. We will also use the substitution $P[\vec{E}/z.\vec{y}]$ for arbitrary variable sequences $\vec{y}$ possibly containing logical variables, whose semantics is defined by the simultaneous substitutions $P[\vec{E}_x/z.\vec{x}]$ and $[\vec{E}_u/\vec{u}]$, 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 $\vec{E}$; if only logical variables are substituted, we simply write $P[\vec{E}/\vec{u}]$. That the substitution accurately catches the semantical update, and thus represents the weakest precondition relation, is expressed by the following lemma:

Lemma 2 (Global substitution). *For arbitrary global states σ and logical environments ω referring only to values existing in σ we have*

$$\begin{array}{rcl} [\![E'[\vec{E}/z.\vec{y}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = & [\![E']\!]_{\mathcal{G}}^{\omega',\sigma'}\ , \textit{and} \\ \omega,\sigma \models_{\mathcal{G}} P[\vec{E}/z.\vec{y}] & \textit{iff} & \omega',\sigma' \models_{\mathcal{G}} P\ , \end{array}$$

where $\omega' = \omega[\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]$ *and* $\sigma' = \sigma[[\![z]\!]_{\mathcal{G}}^{\omega,\sigma}.\vec{y} \mapsto [\![\vec{E}]\!]_{\mathcal{G}}^{\omega,\sigma}]$.

To express a local property p in the global assertion language, we define the substitution $p[z/\mathsf{this}]$ by simultaneously replacing in p all occurrences of the self-reference this by the logical variable z, which is assumed to occur neither in p nor in $\vec{E}$. For notational convenience we view the local variables occurring in the global assertion $p[z/\mathsf{this}]$ as logical variables. Formally, these local variables are replaced by fresh logical variables. We will write $P(z)$ for $p[z/\mathsf{this}]$, and similarly for expressions. The main cases of the substitution are defined as follows:

$$\begin{aligned}
\mathsf{this}[z/\mathsf{this}] &= z \\
x[z/\mathsf{this}] &= z.x \\
u[z/\mathsf{this}] &= u \\
(\exists z'(p))[z/\mathsf{this}] &= \exists z'(p[z/\mathsf{this}]) \\
(\exists z' \in e(p))[z/\mathsf{this}] &= \exists z'((z' \in e[z/\mathsf{this}]) \wedge p[z/\mathsf{this}]) \\
(\exists z' \sqsubseteq e(p))[z/\mathsf{this}] &= \exists z'((z' \sqsubseteq e[z/\mathsf{this}]) \wedge p[z/\mathsf{this}]) \ ,
\end{aligned}$$

where $z \neq z'$ in the cases for existential quantification. The substitution replaces all occurrences of the self-reference this by z, and transforms all occurrences of instance variables x into qualified references $z.x$. For unrestricted quantifications $(\exists z'(p))[z/\mathsf{this}]$ the substitution applies to the assertion p. Local restricted quantifications are transformed into global unrestricted ones where the relations $\in$ and $\sqsubseteq$ are expressed at the global level as operators.

This substitution will be used to combine properties of instance local states on the global level. The substitution $[z/\mathsf{this}]$ preserves the meaning of local assertions, provided the meaning of this and the local variables $\vec{u}$ is matchingly represented by ω:

Lemma 3 (Lifting substitution). *Let σ be a global state, ω and τ a logical environment and local state, both referring only to values existing in σ. Let furthermore e and p be a local expression and a local assertion containing local variables $\vec{u}$. If $\tau(\vec{u}) = \omega(\vec{u})$ and z a fresh logical variable, then*

$$\begin{array}{rcl}
[\![e[z/\mathsf{this}]]\!]_{\mathcal{G}}^{\omega,\sigma} & = & [\![e]\!]_{\mathcal{L}}^{\omega,\sigma(\omega(z)),\tau} \ , \ \textit{and} \\
\omega,\sigma \models_{\mathcal{G}} p[z/\mathsf{this}] & \textit{iff} & \omega,\sigma(\omega(z)),\tau \models_{\mathcal{L}} p \ .
\end{array}$$

4 The proof system

This section presents the assertional proof system for reasoning about Java$_{MT}$ programs, formulated in terms of *proof outlines* [17, 9], i.e., where Hoare-style pre- and postconditions [10, 12] are associated with each control point. The proof system has to accommodate for dynamic object creation, shared-variable concurrency, aliasing, method invocation, synchronization, and especially the monitor concept.

The following section defines how to augment and annotate programs into proof outlines, before Section 4.2 describes the proof method.

4.1 Proof outlines

The definition of a complete proof system requires that we can formulate the transition semantics of $Java_{MT}$ in the assertion language. As the assertion language can reason about the local and global states, only, we have to augment the program with fresh auxiliary variables to represent information about the control points and stack structures within the local and global states.

Formally, assignments $y := e$ of the program can be extended to multiple assignments $y, \vec{y}_{aux} := e, \vec{e}_{aux}$ by inserting additional assignments to distinct auxiliary variables $\vec{y}_{aux}$. Besides the above extension of already occurring assignments, additional multiple assignments to auxiliary variables can be inserted at any point of the program.

An important point of the proof system is the identification of the communicating objects and threads. Roughly speaking, the local state of the execution of a method must represent information about the caller object to distinguish self-calls from others. Additionally, information about its thread membership and its position within the call stack is needed to detect local configurations in caller-callee relationship and reentrant calls.

A local configuration is identified by the object in which it executes together with a unique object-internal identifier. The uniqueness of the local identifier conf is assured by the auxiliary instance variable counter, which is incremented for each new local configuration. The callee receives the "return address" as additional auxiliary formal parameter caller, given by the caller object together with the identity of the calling local configuration.

We identify a thread by the object in which it has begun its execution, i.e., by the self-reference of the deepest local configuration in the thread's stack. This identification is unique since the start-method of an object can be invoked only once, i.e., at most one thread can begin its execution in a single object. The caller thread identity is handed over to the callee in the formal parameter caller_thread, where the callee keeps his own identification in the local variable thread. Their values agree for all method calls except for starting a new thread.

Besides auxiliary variables for thread identification to keep track of the control and information flow between local configurations, we need to capture mutual exclusion and the monitor discipline. The instance variable lock of type $\mathsf{Object} \times \mathsf{Int} + \mathsf{free}$ and initial value free stores the identity of the thread who owns the lock, if any, together with the number of reentrant synchronized calls in the call chain. The instance variables wait and notified of type $2^{\mathsf{Object} \times \mathsf{Int}}$ and initial value $\emptyset$ are the analogues of the *wait* and *notified*-predicates of the semantics and store the threads waiting at the monitor, respectively those being notified. Besides the thread identity, the number of reentrant synchronized calls is stored. In other words, the wait and notified sets remember the old lock-value prior to suspension which is restored when the thread becomes active again. To be able to identify the receiver thread in a !signal-communication we additionally use a local auxiliary variable partner of type 2^{Object} for the notify method. The boolean instance variable started, finally, remembers whether the object's start-method has already been invoked. All auxiliary variables are initialized as usual, except the started-variable of the initial object which gets the value true.

The mentioned additional auxiliary variables are used in the program as follows. The formal parameter list of each method is extended by the auxiliary parameters caller of type $\mathsf{Object} \times \mathsf{Int}$ for the return address and caller_thread of type Object for the calling thread. At the beginning of its body, each method sets $\mathsf{thread} := \mathsf{caller\_thread}$ in an auxiliary assignment except the main and the start method which use $\mathsf{thread} := \mathsf{this}$, instead. Additionally, the beginning of each method body increases counter by one, after its value is remembered in the variable conf. The start-method is specific in setting started to true. In case of synchronized methods, additionally the assigment $\mathsf{lock} := \mathsf{inc}(\mathsf{lock})$ is added, where the semantics of incrementing the lock is given as follows: $[\![\mathsf{inc}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ is $(\tau(\mathsf{thread}), 0)$ for $\sigma_{inst}(\mathsf{lock}) = \mathit{free}$, and $(\alpha, n+1)$ for $\sigma_{inst}(\mathsf{lock}) = (\alpha, n)$. Inversely, decrementing is given by $[\![\mathsf{dec}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ is free, when $\sigma_{inst}(\mathsf{lock})$ equals $(\alpha, 0)$ or *free*, and (α, n) when it equals $(\alpha, n+1)$. This operator is used at the end of synchronized method bodies to decrement the lock-value. Note that the augmentation assures that the decrement operator is applied only in configurations where the value of lock is not *free*. We included this case only for a total function definition.

Matching with the method definition, each method call sends the values of $(\mathsf{this}, \mathsf{conf})$ and thread in addition to the original parameters. The main-method is initially executed with the parameters $(\mathit{nil}, 0)$ and α_0, where α_0 is the initial object.

Invoking the wait-method gives the lock free which is represented by the assignment $\mathsf{lock} := \mathsf{free}$ of the callee. The old lock-value is put into wait, and moved to notified upon signalling. Finally, the notified thread can continue if it gets the lock again, mirrored by the assignments $\mathsf{notified} := \mathsf{notified} \backslash \mathsf{get}(\mathsf{notified}, \mathsf{thread})$ and $\mathsf{lock} := \mathsf{get}(\mathsf{notified}, \mathsf{thread})$ executed when returning from the wait-method. Given a thread identity α, the *get* function retrieves the corresponding stored lock value (α, n) from the set. Note that it is an invariant of the semantics that the association is unique.

The above auxiliary assignments make observations about the control flow available in the state space. To allow *simultaneous* observation, we enclose the observed statement and the auxiliary assignment in a *bracketed section*. The effect of assignments can be observed via multiple assginments, thus we use bracketed sections only for communication and object creation. Introduced only for the sake of verification, they do not influence the control flow and are executed atomically without interleaving with other threads. Atomic execution of a bracketed section, thought, does not mean that the actions inside are executed simultanously. Uniformely, the observation follows the observed actions, such that the received values in a communication can be recorded. To uniformly refer to communication statements together with their observations, we will sometimes write $\langle ?m(\vec{u}); \vec{y} := \vec{e} \rangle$ to indicate that the assignment observes the reception of a method call.

Note that while object creation statements are enclosed in bracketed sections which allow their observation, we do not introduce a specific augmentation as for the communication statements.

The points between communication and observation in bracketed sections, and those at the beginning and at the end of whole methods, are no control points; we will call them *auxiliary points.*

Very generally, the specific auxiliary variables are needed to make the global predicates of the semantics expressible in the assertion language. That the variables and the predicates match will be shown in Section 5.1 as part of the soundness. The crucial predicate for object creation, the freshness-proviso, is already expressible in the global assertion language by existential quantification over existing objects. Therefore we do not need to prescribe a specific augmentation.

Note that in case of self-communication the effect of sender and receiver observations, executed in the same object, depends on the execution order. To assure determinacy, we make the following arrangement: for communication, after parameter passing, first the sender and than the receiver observation is executed. For signalling, the observation of the notifier is followed by the observations of the notified threads, which may contain assignments to local variables only.

In [2] we allowed that in a self-communication both the caller and the callee may change the instance state. This complicated the proof system, especially the interference freedom test. In this work we avoid this complication in that we require that the observation of the caller side in a self-communication may not change the values of instance variables. Formally, each assignment to instance variables in the bracketed section of a method invocation $e_0.m(\vec{e})$ or its following receive statement must have the form $x := \mathsf{if}\ e_0 = \mathsf{this}\,\mathsf{then}\ x\,\mathsf{else}\ e\,\mathsf{fi}$. Invoking the start-method by a self-call when the thread is already started is specific in that the caller is the only active entity (cf. rule $\textsc{Call}^{skip}_{start}$). In this case, it has to be the caller that updates the instance state; the corresponding oberservation has the form $x := \mathsf{if}\ e_0 = \mathsf{this} \wedge \neg\mathsf{started}\,\mathsf{then}\ x\,\mathsf{else}\ e\,\mathsf{fi}$. The values of the introduced auxiliary variables are changed only in the bracketed sections as described above.

To specify invariant properties of the system, the augmented programs are *annotated* by attaching local assertions to each control and auxiliary point. Besides that, for each class c, the annotation defines a local assertion I_c called *class invariant* that expresses invariant properties of instances of the class.[7] Finally, the *global invariant* $GI \in GAss$ specifies properties of communication between objects. As such, it should be invariant under object-internal computation. For that reason, we require that the values of instance variables occurring in the global invariant may be changed only in the bracketed sections of object creation, and in those of communication statements, i.e., sending or receiving method call and return. Note that the global invariant is not affected by the monitor signalling mechanism since this is object-internal communication.

[7] Note that the notion of class invariant commonly used for sequential object-oriented languages differs from our notion: In a sequential setting, it is sufficient that class invariant holds initially and is preserved by whole method calls, but not necessarily in between.

Definition 1 (Annotation, proof outline). *An* annotation *of an augmented program associates with each control and auxiliary point a local assertion $p \in LAss$. Furthermore, it assigns to each class c a* class invariant $I_c \in LAss$ *which may refer only to the instance variables of c. We require that $pre(body_{m,c}) = post(body_{m,c}) = I_c$. Finally, the program is assigned a* global invariant $GI \in GAss$. *We require that in the annotation no free logical variables occur, and that for all qualified references $E.x$ in GI with $E \in GExp^c$, all assignments to x in class c occur in bracketed sections of communication or object creation statements. An augmented and annotated program prog, denoted by prog′, is called a* proof outline.

For annotated programs, we use the standard notation $\{p\}\ stm\ \{q\}$ to express that p and q are the *pre-* and *postconditions* of *stm*, i.e., the assertions in front of and after *stm*, and write $pre(stm)$ and $post(stm)$ to refer to them.

The use of the specific auxiliary variables for the observation of communication is illustrated below. Figure 1 handles ordinary synchronized method calls, except the predefined start- and monitor-methods. Non-synchronized methods are treated analogously except that they do not change the lock value. The start method (cf. Figure 2) additionally handles thread creation.

```
...
{$p_1$}⟨e0.m(this,conf,thread,e){$p_2$}; $\vec{y}_1 := \vec{e}_1$⟩{$p_3$};
⟨receive $u_{ret}$;{$p_4$}; $\vec{y}_4 := \vec{e}_4$⟩;{$p_5$}
...

{$I_c$} sync m (caller,caller_thread,u) { {$q_2$}
        ⟨conf := counter, counter := counter + 1, thread := caller_thread,
         lock := inc(lock), $\vec{y}_2 := \vec{e}_2$⟩; {$q_3$}
        ... {$q_4$}
        ⟨return $e_{ret}$; {$q_5$}lock := dec(lock), $\vec{y}_3 := \vec{e}_3$⟩
} {$I_c$}
```

Fig. 1. Method call

The communication pattern of the wait- and notify-methods is observed as shown in Figure 3.

4.2 Proof system

The proof system formalizes a number of *verification conditions* which inductively ensure that for each *reachable* configuration $\langle T, \sigma\rangle$ and for each local configuration (α, τ, stm) in T the precondition of the statement *stm* is satisfied

```
$\{I_c\}$ nsync start (caller,caller_thread) { $\{q_2\}$
       $\langle$conf := counter, counter := counter + 1, thread := this,
        started := true, $\vec{y}_2 := \vec{e}_2\rangle$; $\{q_3\}$
       ... $\{q_4\}$
       $\langle$return; $\{q_5\}\vec{y}_3 := \vec{e}_3\rangle$
} $\{I_c\}$
```

Fig. 2. Start

```
$\{I_c\}$ nsync wait (caller,caller_thread) { $\{q_2\}$
       $\langle$conf := counter, counter := counter + 1, thread := caller_thread,
        wait := wait $\cup$ {lock}, lock := free, $\vec{y}_2' := \vec{e}_2'\rangle\{q_3\}$
       $\langle$?signal; $\{q_4\}$; $\vec{y}' := \vec{e}'\rangle\{q_5\}$;
       $\langle$return$_{getlock}$; $\{q_6\}$lock := get(notified,thread),
        notified := notified \ get(notified,thread), $\vec{y}_3' := \vec{e}_3'\rangle$
} $\{I_c\}$


$\{I_c\}$ nsync notify (caller,caller_thread) { $\{p_2\}$
       $\langle$conf := counter, counter := counter + 1, thread := caller_thread,
       $\vec{y}_2 := \vec{e}_2\rangle\{p_3\}$
       $\langle$!signal;$\{p_4\}$ notified := notified $\cup$ get(wait,partner),
        wait := wait \ get(wait,partner), $\vec{y} := \vec{e}\rangle\{p_5\}$
       $\langle$return; $\{p_6\}\vec{y}_3 := \vec{e}_3\rangle$} $\{I_c\}$

$\{I_c\}$ nsync notifyAll (caller,caller_thread) { $\{r_2\}$
       $\langle$conf := counter, counter := counter + 1, thread := caller_thread,
       $\vec{y}_2'' := \vec{e}_2''\rangle\{r_3\}$
       $\langle$!signal_all; $\{r_4\}$notified := notified $\cup$ wait,
        wait := $\emptyset$, $\vec{y}'' := \vec{e}''\rangle\{r_5\}$
       $\langle$return; $\{r_6\}\vec{y}_3'' := \vec{e}_3''\rangle$} $\{I_c\}$
```

Fig. 3. Signalling

and the class invariants and the global invariant hold. To cover concurrency and communication, the verification conditions are grouped, as usual, into initial conditions, local correctness conditions, an interference freedom test, and a cooperation test.

A proof outline is *initially correct*, if the precondition of the main statement and the global invariant are satisfied in the initial configuration. *Local correctness* ensures that local properties of a thread are invariant under its own execution. This invariance can be guaranteed by local correctness conditions only if no communication or object creation takes place, since their effect depends on the communicated values and cannot be determined locally. They will be analyzed in the *cooperation test* whose conditions are formalized in the global language. The invariance of local properties of a thread that currently executes in a given object can also be influenced by other threads executing in the same object which possibly change the instance state. The corresponding verification conditions are formalized in the *interference freedom test.*

Our proof method is *modular* in the sense that it allows for separate interference freedom and cooperation tests. This modularity, which in practice simplifies correctness proofs considerably, is obtained by disallowing the assignment of the result of communication and object creation to instance variables. Clearly, such assignments can be avoided by additional assignments to fresh local variables and thus at the expense of new interleaving points.

Before specifying the verification conditions for a proof outline, we first fix some auxiliary functions and notations. Let Init be a syntactical operator with interpretation *Init* (cf. page 8). Given $IVar_c$ as the set of instance variables of class c and $z \in LVar^c$, then $\mathsf{InitState}(z)$ denotes the global assertion $z \neq \mathsf{nil} \wedge \bigwedge_{x \in IVar_c} z.x = \mathsf{Init}(x)$, expressing that the object denoted by z is in its initial instance state.

For readability, in the following definitions we will use the notation $p \circ f$ with $f = [\vec{e}/\vec{y}]$ for the substitution $p[\vec{e}/\vec{y}]$; we use a similar notation for global assertions. Note that the substitution binds stronger than the logical operators $\wedge$ and $\rightarrow$.

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

4.2.1 Initial correctness A proof outline is *initially correct*, if the precondition of the main statement is satisfied by the initial instance and local states after the execution of the observation at the beginning of the main-method. Furthermore, the global invariant must be satisfied by the initial global state.

Definition 2 (Initial correctness). *A proof outline is* initially correct, *if*

$$\models_{\mathcal{L}} pre(stm) \circ f_{obs} \circ f_{init} \ , \qquad (1)$$

$$\models_{\mathcal{G}} \forall z(\mathsf{InitState}(z) \wedge \forall z'(z' = \mathsf{nil} \vee z = z') \rightarrow GI \circ f^z_{obs} \circ f^z_{init}) \ , \qquad (2)$$

where $body_{\mathsf{main}} = \langle ?\mathsf{main}(\vec{u}); \vec{y}_2 := \vec{e}_2\rangle; stm$ *is the body and* $\vec{y}$ *the local and instance variables of the main-method,* z *is of the type of the main class, and* $z' \in LVar^{\mathsf{Object}}$*. Furthermore,*

$$\begin{aligned} f_{init} &= [\mathsf{nil}, (\mathsf{this}, 0)/\mathsf{callerobj}, \mathsf{id}][\mathsf{true}/\mathsf{started}][\mathsf{Init}(\vec{y})/\vec{y}] \qquad & f_{obs} &= [\vec{e}_2/\vec{y}_2] \\ f^z_{init} &= [\mathsf{nil}, (z, 0)/\mathsf{callerobj}, \mathsf{id}][\mathsf{true}/z.\mathsf{started}][\mathsf{Init}(\vec{y})/z.\vec{y}] \qquad & f^z_{obs} &= [\vec{E}_2(z)/z.\vec{y}_2]\,. \end{aligned}$$

4.2.2 Local correctness A proof outline is *locally correct*, if the usual verification conditions [6] for standard sequential constructs hold. Especially, the precondition of an ordinary assignment, as given in the proof-outline, must imply its postcondition after the execution of the assignment (cf. Equation (3)). Besides that, local correctness requires that all assertions of a class imply the class invariant.

Definition 3 (Local correctness). *A proof outline is* locally correct*, if for all assignments* $\{p_1\}\vec{y} := \vec{e}\{p_2\}$ *outside bracketed sections,*

$$\models_{\mathcal{L}} p_1 \rightarrow p_2 \circ f_{ass}\ , \tag{3}$$

with $f_{ass} = [\vec{e}/\vec{y}]$*, and for all assertions* p *in class* c *with class invariant* I_c*,*

$$\models_{\mathcal{L}} p \rightarrow I_c\ . \tag{4}$$

Note that we have no local verification conditions for assignments in bracketed sections. The postconditions of such statements express *assumptions* about the communicated values. These assumptions will be verified in the *cooperation test.*

Other threads concurrently executing in the same object may influence or *interfere with* the invariance of the local assertions. This is covered in the interference freedom test.

4.2.3 The interference freedom test For inductivity, it must be shown that local assertions are invariant under computation steps in which they are not involved. The resulting proof obligations constitute the interference freedom test. Since we disallow qualified reference to instance variables in Java$_{MT}$, we only have to deal with the invariance under execution within the *same* object. Affecting only local variables, communication and object creation do not change the state of the executing objects. Thus we only have to take assignments into account. In the following let p and $\vec{y} := \vec{e}$ be an assertion and an assignment occurring in the same class. Using the specific auxiliary variables, we next formalize the conditions when p has to be invariant under the assignment, namely if $\vec{y} := \vec{e}$ is executed independently of p.

If they belong to the *same* thread, the only assertions endangered are those at control points waiting for a return value earlier in the current execution stack. In other words, an assignment belonging to a *reentrant* code segment can affect the precondition of a receive statement whose execution is suspended earlier in the same call chain. The assignment belonging to the *matching* return statement, however, need not be considered. To express this kind of interference, we define $\mathsf{waits\_for\_ret}(p, \vec{y} := \vec{e})$ by

- $\mathsf{conf}' < \mathsf{conf}$, if p is the precondition of a receive statement and $\vec{y} := \vec{e}$ is not in the bracketed section of a return statement;
- $\mathsf{conf}' < \mathsf{conf(caller)}$ if p is the precondition of a receive statement and $\vec{y} := \vec{e}$ is in the bracketed section of a return statement and where conf(caller) is the second component of the caller-value;
- false, otherwise.

If the assertion and the assignment belong to *different* threads, the assignment cannot interfere with p if they belong to a *matching* communication pair. Note that observations of ?signal do not change the instance state, and thus interference freedom is trivial. So we only have to exclude matching communication where the assignment belongs to the sending part. In other words, the precondition of a ?signal statement needs not to be shown invariant under the observation of a matching !signal or !signal_all. To capture this condition, we define the assertion $\mathsf{gets\_signalled}(p, \vec{y} := \vec{e})$ as follows.

- $\mathsf{partner} = \xi'$, if p is the precondition of a ?signal statement and the assignment is from the bracketed section of a !signal;
- true, if p is the precondition of a ?signal statement and the assignment is in the bracketed section of a !signal_all;
- and false otherwise.

Collecting the above two cases, we define for assertions p at control points and assignments $\vec{y} := \vec{e}$ in the same class:

$$\begin{aligned}\mathsf{interleavable}(p, \vec{y} := \vec{e}) := \mathsf{thread} = \mathsf{thread}' &\rightarrow \mathsf{waits\_for\_ret}(p, \vec{y} := \vec{e}) \wedge \\ \mathsf{thread} \neq \mathsf{thread}' &\rightarrow \neg\mathsf{gets\_signalled}(p, \vec{y} := \vec{e}) \ .\end{aligned}$$

Using this predicate the interference freedom test is formulated below. Remember that q' denotes q with all local variables u replaced by some fresh local variables u'.

Definition 4 (Interference freedom). *A proof outline is* interference free, *if for all classes c, all assignments $\{p\}\vec{y} := \vec{e}$ and assertions q at control points in c,*

$$\models_{\mathcal{L}} p \wedge q' \wedge \mathsf{interleavable}(q, \vec{y} := \vec{e}) \rightarrow q' \circ f_{ass} \ , \tag{5}$$

with $f_{ass} = [\vec{e}/\vec{y}]$.

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 preconditions of the involved bracketed sections imply their postconditions after the common step. We start with the cooperation test for notification (cf. Figure 3).

Since signalling takes place within a single object, the cooperation test is formulated in the local assertion language. The effects of the observations are represented as before by substitutions. Note that the order in which the syntactic substitutions are applied is reverse compared with the order in which the corresponding assignments update the state. In addition, we substitute the variable partner to fix the identity of the receiving thread. Also broadcast signalling via !signal_all can be handled pairwise for each of the receivers together with the sender, as we require that the receiver side may not change the instance state. If there is no thread to be signalled, the corresponding observation is skipped.

The soundness of the proof system requires, that the precondition of an observation, used in the interference freedom test, describes the state in which the assignment is executed. Therefore, we additionally have to show that these assertions are valid after communication.

Definition 5 (Cooperation test: Notification). *A proof outline satisfies the* cooperation test for notification, *if the following conditions hold:*

1. SIGNAL *For all classes c and statements* $\{p_1\}\langle\text{!signal}; \{p_2\}\vec{y}_1 := \vec{e}_1\rangle\{p_3\}$ *and* $\{q_1\}\langle\text{?signal}; \{q_2\}\vec{y}_2 := \vec{e}_2\rangle\{q_3\}$ *in c,*
$$\models_{\mathcal{L}} p_1 \wedge q_1' \quad \rightarrow \quad (p_2 \wedge q_2') \circ f_{comm} \wedge (p_3 \wedge q_3') \circ f_{obs2} \circ f_{obs1} \circ f_{comm}\ , \qquad (6)$$
where $f_{comm} = [\{\text{thread}'\}/\text{partner}]$, $f_{obs1} = [\vec{e}_1/\vec{y}_1]$, *and* $f_{obs2} = [\vec{e}_2'/\vec{y}_2']$.
2. SIGNALALL *For statements* $\{p_1\}\langle\text{!signal\_all}; \{p_2\}\vec{y}_1 := \vec{e}_1\rangle\{p_3\}$ *the above condition holds with* f_{comm} *is the identity function.*
3. SIGNAL$_{skip}$ *For statements* $\{p_1\}\langle\text{!signal}; \{p_2\}\vec{y}_1 := \vec{e}_1\rangle\{p_3\}$ *Equation(6) must hold with the additional antecedent* wait $= \emptyset$, *where* $q_1 = q_2 = q_3 =$ true, $f_{comm} = [\emptyset/\text{partner}]$, *and* f_{obs2} *is the identity function. For* !signal_all, *the same is required with* f_{comm} *as the identity function.*

We continue with the cooperation test for communication; for the corresponding augmentation see Figure 1. Since different objects may be involved, the corresponding cooperation test is formulated in the global assertion language. Consequently the local properties and expressions are expressed in the global language using the lifting substitution. To avoid name clashes between local variables of the partners, we rename those of the callee. Besides ensuring invariance of the global invariant over bracketed sections, it specifies conditions under which the local properties of the communicating partners are preserved.

In the following definition, the logical variable z denotes the object calling a method and z' refers to the callee. In the global state prior to the communication, we can assume that the global invariant and the preconditions of the communicating statements hold. In the case of method invocation, the precondition of the callee is given by its class invariant, as defined in the annotation. Dually for the postcondition for a bracketed section of a return. That the two statements indeed represent communicating partners and especially that the communication is enabled is captured in the assertion communicating, which depends on the type of communication. For instance, in case of synchronized method invocation, the lock of the callee object has to be free, or owned by the caller, which is expressed by $z'.\text{lock} = \text{free} \vee \text{thread}(z'.\text{lock}) = \text{thread}$, where thread is the caller-thread and $\mathit{thread}(\alpha, n) = \alpha$.

Similarly to notification, the cooperation test assures under the above assumptions, that the postconditions of the communicating bracketed sections are valid after communication and observation. Furthermore, the preconditions of the observations must hold immediately after the communication. As before, communication and state updates of the observations are captured by substitutions. Remember that method invocation also hands over the return address, which allows to determine matching communication pairs for return. . Note that the actual parameters do not contain *instance* variables, i.e., their interpretation does not change during the execution of the method body. Therefore, the actual parameters can be used not only to logically capture the conditions at the entry of the method body, but at the exit of the method body, as well.

The global invariant GI is not allowed to refer to instance variables whose values are changed outside bracketed sections. Consequently, it will be automatically invariant under the execution of statements outside bracketed sections. For the bracketed sections, however, the invariance must be shown as part of the cooperation test. The global invariant must hold after both communication and the accompanying observations have been performed.

Invoking the start-method of an object whose thread is already started does not have communication effects. Neither has returning from a start-method or from the first execution of the main-method.

Let again p' denote the assertion p with every local variable u replaced by a fresh one u', and similarly for expressions. As already mentioned, we use the shortcuts $P(z)$ for $p[z/\mathsf{this}]$, $Q'(z')$ for $q'[z'/\mathsf{this}]$, and similarly for expressions, where local variables are viewed on the global level as logical ones.

Definition 6 (Cooperation test: Communication). *A proof outline satisfies the* cooperation test for communication, *if*

$$\begin{aligned}\models_{\mathcal{G}}\ & GI \wedge P_1(z) \wedge Q'_1(z') \wedge \mathsf{communicating} \wedge z \neq \mathsf{nil} \wedge z' \neq \mathsf{nil} \rightarrow \\ & (P_2(z) \wedge Q'_2(z')) \circ f_{comm} \wedge (GI \wedge P_3(z) \wedge Q'_3(z')) \circ f_{obs2} \circ f_{obs1} \circ f_{comm}\end{aligned} \tag{7}$$

holds for distinct fresh logical variables $z \in LVar^c$ and $z' \in LVar^{c'}$, in the following cases:

1. *(a)* CALL: *For all statements $\{p_1\}\langle e_0.m(\vec{e}); \{p_2\}\vec{y}_1 := \vec{e}_1\rangle\{p_3\}$ in class c with $e_0 \in Exp_c^{c'}$, where method $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ of c' is synchronized with body $\{q_1\}\langle ?m(\vec{u}); \{q_2\}\vec{y}_2 := \vec{e}_2\rangle; \{q_3\}stm$ and local variables $\vec{v}$ except the formal parameters. The assertion* communicating *is given by $E_0(z) = z' \wedge (z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread})$. Furthermore, $f_{comm} = [\vec{E}(z), \mathsf{Init}(\vec{v})/\vec{u}', \vec{v}']$, $f_{obs1} = [\vec{E}_1(z)/z.\vec{y}_1]$, $f_{obs2} = [\vec{E}'_2(z')/z'.\vec{y}'_2]$. If m is not synchronized, $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ in* communicating *is dropped.*
   (b) CALL$_{monitor}$: *For $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$,* communicating *given by $E_0(z) = z' \wedge \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$.*
   (c) CALL$_{start}$: *For $m = \mathsf{start}$,* communicating *given by $E_0(z) = z' \wedge \neg z'.\mathsf{started}$.*

(d) $\text{CALL}^{skip}_{start}$: *For* $m = \mathsf{start}$, *additionally, (7) must hold with* $\mathsf{communicating}$ *given by* $E_0(z) = z' \wedge z'.\mathsf{started}$, $q_2 = q_3 = \mathsf{true}$, *and* f_{comm} *and* f_{obs2} *are the identity functions.*

2. (a) RETURN: *For all* $\langle e_0.m(\vec{e}); \vec{y}_1 := \vec{e}_1\rangle; \{p_1\}\langle \mathsf{receive}\, u_{ret}; \{p_2\}\vec{y}_4 := \vec{e}_4\rangle\{p_3\}$ *occurring in* c *with* $e_0 \in Exp^{c'}_c$, *such that method* m *of* c' *has the return statement* $\{q_1\}\langle \mathsf{return}\; e_{ret}; \{q_2\}\vec{y}_3 := \vec{e}_3\rangle\{q_3\}$, *and formal parameter list* $\vec{u}$, *Equation (7) must hold with* $\mathsf{communicating}$ *given by* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z)$, *and where* $f_{comm} = [E'_{ret}(z')/u_{ret}]$, $f_{obs1} = [\vec{E}'_3(z')/z'.\vec{y}'_3]$, *and* $f_{obs2} = [\vec{E}_4(z)/z.\vec{y}_4]$.

(b) RETURN_{wait}: *For* $\{q_1\}\langle \mathsf{return}_{getlock}; \{q_2\}\vec{y}_3 := \vec{e}_3\rangle\{q_3\}$ *in a wait method,* $\mathsf{communicating}$ *is* $E_0(z) = z' \wedge \vec{u}' = \vec{E}(z) \wedge z'.\mathsf{lock} = \mathsf{free}$.

(c) RETURN_{start}: *For* $\{q_1\}\langle \mathsf{return}; \{q_2\}\vec{y}_3 := \vec{e}_3\rangle\{q_3\}$ *occuring in the main method or in a start method,* $p_1 = p_2 = p_3 = \mathsf{true}$, $\mathsf{communicating} = (\mathsf{id}' = (z', 0))$, *and furthermore* f_{comm} *and* f_{obs2} *the identity function.*

Besides method calls and return, the cooperation test needs to handle object creation, taking care of the preservation of the global invariant, the postcondition of the new-statement and its bracketed section, and the new object's class invariant. We can assume that the precondition of the object creation statement and the global invariant hold in the configuration prior to the instantiation. The extension of the global state with a freshly created object is formulated in a *strongest postcondition* style, i.e., it is required to hold immediately *after* the instantiation. We use existential quantification to refer to the old value: z' of type $LVar^{\mathsf{list\,Object}}$ represents the existing objects prior to the extension. Moreover, that the created object's identity stored in u is fresh and that the new instance is properly initialized is captured by the global assertion $\mathsf{Fresh}(z', u)$ defined as $\mathsf{InitState}(u) \wedge u \notin z' \wedge \forall v(v \in z' \vee v = u)$, where $\mathsf{InitState}(u)$ is as defined above. To express that an assertion refers to the set of existing objects *before* the new-statement, we need to *restrict* any existential quantification to range over objects from z', only. So let P be a global assertion and $z' \in LVar^{\mathsf{list\,Object}}$ a logical variable not occurring in P. Then $P \downarrow z'$ is the global assertion P with all quantifications $\exists z(P')$ replaced by $\exists z(\mathsf{obj}(z) \subseteq z' \wedge P')$, where $obj(v)$ denotes the set of objects occurring in the value v, formally

$$obj(v) = \begin{cases} \emptyset & \text{if } v \in Val^{\mathsf{Bool}} \cup Val^{\mathsf{Int}} \\ \{v\} & \text{if } v \in \bigcup_c Val^c_{nil} \\ obj(v_1) \cup obj(v_2) & \text{if } v = (v_1, v_2) \in \bigcup_{t_1,t_2} Val^{t_1 \times t_2}_{nil} \\ \bigcup_{v_i \in v} obj(v_i) & \text{if } v \in \bigcup_t Val^{\mathsf{list}\, t}_{nil} . \end{cases}$$

The following lemma formulates the basic property of the projection operator:

Lemma 4. *Assume a global state* σ, *an extension* $\sigma' = \sigma[\alpha \mapsto \sigma^{c,init}_{inst}]$ *for some* $\alpha \in Val^c$, $\alpha \notin dom(\sigma)$, *and a logical environment* ω *referring only to values existing in* σ. *Let* v *be the sequence consisting of all elements of* $\bigcup_c dom^c_{nil}(\sigma)$. *Then for all global assertions* P *and logical variables* $z' \in LVar^{\mathsf{list\,Object}}$ *not occurring in* P,

$$\omega, \sigma \models_{\mathcal{G}} P \textit{ iff } \omega[z' \mapsto v], \sigma' \models_{\mathcal{G}} P \downarrow z'.$$

Thus the predicates $GI \downarrow z'$ and $\exists u(pre(u := \mathsf{new}^c)[z/\mathsf{this}]) \downarrow z'$ express that the global invariant and the precondition of the object creation statement hold for the old value of u prior to the creation of the new object. This leads to the following definition of the cooperation test for object creation:

Definition 7 (Cooperation test: Instantiation). *A proof outline satisfies the* cooperation test for object creation*, if for all classes c' and statements $\{p_1\}\langle u := \mathsf{new}^c; \{p_2\}\vec{y} := \vec{e}\rangle\{p_3\}$ in c':*

$$\models_{\mathcal{G}} z \neq \mathsf{nil} \wedge \exists z'\Big(\mathsf{Fresh}(z', u) \wedge (GI \wedge \exists u(P_1(z))) \downarrow z'\Big) \rightarrow \\ P_2(z) \wedge I_c(u) \wedge (GI \wedge P_3(z)) \circ f_{obs} \ , \tag{8}$$

with $z \in LVar^{c'}$ and $z' \in LVar^{\mathsf{list\,Object}}$ fresh , $\vec{E}(z) = \vec{e}[z/\mathsf{this}]$, and where $f_{obs} = [\vec{E}(z)/z.\vec{y}]$.

5 Soundness and completeness

This section contains soundness and completeness of the proof method of Section 4.2. Given a program together with its annotation, the proof system stipulates a number of induction conditions for the various types of assertions and program constructs. *Soundness* for the inductive method means that for a proof outline satisfying the verification conditions, all configurations reachable in the operational semantics satisfy the given assertions, *completeness* conversely means that if a program does satisfy an annotation, this is provable. For convenience, let us introduce the following notations. Given a program *prog*, we will write φ_{prog} or just φ for its annotation, and write $prog \models \varphi$, if *prog* satisfies all requirements stated in the assertions:

Definition 8. *Given a program prog with annotation φ, then $prog \models \varphi$ iff for all reachable configurations $\langle T, \sigma\rangle$ of prog, for all $(\alpha, \tau, stm) \in T$, and for all logical environments ω referring only to values existing in σ:*

1. $\omega, \sigma(\alpha), \tau \models_{\mathcal{L}} pre(stm)$, and
2. $\omega, \sigma \models_{\mathcal{G}} GI$.

Furthermore, for all classes c, objects $\beta \in dom^c(\sigma)$, and local states τ':

3. $\omega, \sigma(\beta), \tau' \models_{\mathcal{L}} I_c$.

For proof outlines, we write $prog' \vdash \varphi'$ iff $prog'$ satisfies the verification conditions of the proof system.

5.1 Soundness

Soundness, as mentioned, means that all reachable configurations do satisfy their assertions for an annotated program that has been verified using the proof conditions. Soundness of the method is proved by a straightforward, albeit rather tedious induction on the computation steps.

Before embarking upon the soundness formulation and its proof, we need to clarify the connection between the original program and proof outline, i.e., the one decorated with assertions, extended by auxiliary variables and sprinkled with bracketed sections. The transformation is done for the sake of verification, only, and as far as the un-augmented portion of the states and the configurations is concerned, the behavior of the original and the transformed program are the same.

To make the connection between original program and the proof outline one precise, we define a projection operation $\downarrow prog$, that jettisons all additions of the transformation. So let $prog'$ be a proof outline for $prog$, and $\langle T', \sigma' \rangle$ a global configuration of $prog'$. Then $\sigma' \downarrow prog$ is defined by removing all auxiliary instance variables from the instance state domains. For the set of thread configurations $T' \downarrow prog$ is given by restricting the domains of the local states to non-auxiliary variables and removing all annotations, augmentations, and bracketed sections. The following lemma expresses that the transformation does not change the behavior of programs:

Lemma 5. *Let $prog'$ be a proof outline for program $prog$. Then $\langle T, \sigma \rangle$ is a reachable configuration of $prog$ iff there exists a reachable configuration $\langle T', \sigma' \rangle$ of $prog'$ with $\langle T' \downarrow prog, \sigma' \downarrow prog \rangle = \langle T, \sigma \rangle$.*

The augmentation introduced a number of specific auxiliary variables that reflect the predicates used in the semantics. That the semantics is faithfully represented by the variables is formulated in the following lemmas.

Lemma 6 (Identification). *Let $\langle T, \sigma \rangle$ be a reachable configuration of a proof outline program. Then*

1. *for all stacks ξ and ξ' in T and for all local configurations $(\alpha, \tau, stm) \in \xi$ and $(\alpha', \tau', stm') \in \xi'$ we have $\tau(\mathsf{thread}) = \tau'(\mathsf{thread})$ iff $\xi = \xi'$, and*
2. *for each stack $(\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$ in T and indices $i < j$,*
   (a) $\tau_i(\mathsf{thread}) = \alpha_0$;
   (b) $\alpha_i = \alpha_j$ implies $\tau_i(\mathsf{conf}) < \tau_j(\mathsf{conf})$,
   (c) $\tau_j(\mathsf{caller}) = (\alpha_{j-1}, \tau_{j-1}(\mathsf{conf}))$, and
   (d) $\tau_j(\mathsf{caller\_thread}) = \tau_{j-1}(\mathsf{thread})$.

Lemma 7 (Wait, Notify). *For all reachable $\langle T, \sigma \rangle$ and $\alpha \in dom(\sigma)$ we have*

1. *$\xi = (\alpha_0, \tau_0, stm_0) \circ \xi' \in wait(T, \alpha)$ iff $(\alpha_0, n) \in \sigma(\alpha)(\mathsf{wait})$ where $n = |\{(\alpha_i, \tau_i, stm_i) \in \xi \mid stm_i \text{ synchr.}\}| - 1$,*
2. *$\xi = (\alpha_0, \tau_0, stm_0) \circ \xi' \in notified(T, \alpha)$ iff $(\alpha_0, n) \in \sigma(\alpha)(\mathsf{notified})$ where $n = |\{(\alpha_i, \tau_i, stm_i) \in \xi \mid stm_i \text{ synchr.}\}| - 1$.*

Lemma 8 (Lock). *Let $\langle T, \sigma \rangle$ be a reachable configuration of a proof outline, $\alpha \in dom(\sigma)$ and $\xi = (\alpha_0, \tau_0, stm_0) \circ \xi' \in T$. Then*

1. *$\neg owns(T, \alpha)$ iff $\sigma(\alpha)(\mathsf{lock}) = free$;*

2. $owns(\xi, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{lock}) = (\alpha_0, n)$ *with* $n = |\{(\alpha, \tau, stm) \in \xi \mid stm\ synchr.\}| - 1$

Lemma 9 (Started). *For all reachable configurations* $\langle T, \sigma\rangle$ *of a transformed program and all objects* $\alpha \in dom(\sigma)$*, we have* $started(T, \alpha)$ *iff* $\sigma(\alpha)(\mathsf{started})$.

Let *prog* be a program with annotation φ, and $prog'$ a a corresponding proof outline with annotation φ'. Let GI' be the global invariant of φ', I'_c denote its class invariants, and for an assertion p of φ let p' denote the assertion of φ' associated with the same control point. We write $\models \varphi' \to \varphi$ iff $\models_{\mathcal{G}} GI' \to GI$, $\models_{\mathcal{L}} I'_c \to I_c$ for all classes c, and $\models_{\mathcal{L}} p' \to p$, for all assertions p of φ associated with some control point. To give meaning to the auxiliary variables, the above implications are evaluated in the context of states of the augmented program. The following theorem states the soundness of the proof method.

Theorem 1 (Soundness). *Given a proof outline* $prog'$ *with annotation* $\varphi_{prog'}$.

$$\text{If} \quad prog' \vdash \varphi_{prog'} \quad \text{then} \quad prog' \models \varphi_{prog'} \ .$$

The soundness proof is basically an induction on the length of computation, simultaneous on all three parts from Definition 8. Theorem 1 is formulated for reachability of augmented programs. With the help of Lemma 5, we immediately get:

Corollary 1. *If* $prog' \vdash \varphi_{prog'}$ *and* $\models \varphi_{prog'} \to \varphi_{prog}$*, then* $prog \models \varphi_{prog}$.

5.2 Completeness

Next we conversely show that if a program satisfies the requirements asserted in its proof outline, then this is indeed provable, i.e., then there exists a proof outline which can be shown to hold and which implies the given one:

$$\forall prog.\ prog \models \varphi_{prog} \ \Rightarrow\ \exists prog'.\ prog' \vdash \varphi_{prog'} \ \wedge\ \models \varphi_{prog'} \to \varphi_{prog} \ .$$

Given a program satisfying an annotation $prog \models \varphi_{prog}$, the consequent can be uniformly shown, i.e., independently of the given assertional part φ_{prog}, by instantiating $\varphi_{prog'}$ to the strongest annotation still provable, thereby discharging the last clause $\models \varphi_{prog'} \to \varphi_{prog}$. Since the strongest annotation still satisfied by the program corresponds to reachability, the key to completeness is to

1. augment each program with enough information, to be able to
2. express reachability in the annotation, i.e., annotate the program such that a configuration satisfies its local and global assertions exactly if reachable (see Definition 10 below), and finally
3. to show that this augmentation indeed satisfies the verification conditions.

We begin with the augmentation, using the transformation from Section 4.1 as starting point, where the programs are equipped with bracketed sections and augmented with specific auxiliary variables.

Now to make visible within a configuration whether or not it is reachable, the standard way is to add information into the states about the way it has been reached, i.e., the *history* of the computation leading to the configuration. It is recorded in history variables, containing enough information to distinguish reachable from unreachable configurations.

The assertional language is split into a local and a global level, and likewise the proof-system is tailored to separate local proof obligations from global ones to obtain a modular proof system. The history will be recorded in instance variables, and thus each instance can keep track only of its own past. To mirror the split into a local and a global level in the proof system, the history per instance is recorded separately for *internal* and *external* behavior. The sequence of internal state changes local to that instance are recorded in the *local* history and the external behavior in the *communication* history.

The communication history keeps information about the kind of communication, the communicated values, and the identity (both object and local configuration identities) of the communication partners involved. For the kind of communication, we distinguish as cases object creation, ingoing and outgoing method calls, and likewise ingoing and outgoing communication for the return value. We use the set of constants $\{\mathsf{new}, \mathsf{call}, \mathsf{called}, \mathsf{return}, \mathsf{receive}\}$ for this purpose. Note in passing that the information stored in the communication history matches exactly the information needed to decorate the transitions in order to obtain a compositional variant of the operational semantics of Section 2.3.2. See [1] for such a compositional semantics.

To facilitate reasoning, we introduce an additional auxiliary local variable loc, which stores the current control point of the execution of a thread. Given a function which assigns to all control points unique location labels, we extend each assignment with the update $\mathsf{loc} := l$, where l is the label of the control point after the given occurrence of the assignment. Also bracketed sections which do not contain assignments are extended with the update. We write $l \equiv stm$ if l represents the control point in front of stm in a method body $stm'; stm$.

Definition 9 (Augmentation with histories). *Each class is further extended by two auxiliary instance variables* h_{inst} *and* h_{comm}*, both initialized to the empty sequence. They are updated as follows:*

1. *Each assignment* $\vec{y} := \vec{e}$ *outside bracketed sections or in bracketed sections of* $?m(\vec{u})$, $\mathsf{return}\ e_{ret}$, $\mathsf{!signal}$, $\mathsf{!signal\_all}$, *and* new *in a class c is extended with*

$$\mathsf{h}_{inst} := \mathsf{h}_{inst} \circ ((\vec{x}, \vec{u})[\vec{e}/\vec{y}]) ,$$

where $\vec{x}$ *are the instance variables of class c containing also* h_{comm} *but without* h_{inst}*, and* $\vec{u}$ *are the local variables. Bracketed sections of* $e_0.m(\vec{e})$ *and its subsequent* $\mathsf{receive}\, u_{ret}$ *statement with auxiliary assignment* $\vec{y} := \vec{e}$ *get* *extended with the assignment*

$$\mathsf{h}_{inst} := \mathsf{if}\ e_0 = \mathsf{this}\,\mathsf{then}\ \mathsf{h}_{inst}\ \mathsf{else}\ \mathsf{h}_{inst} \circ ((\vec{x}, \vec{u})[\vec{e}/\vec{y}])\ ,$$

instead.

2. *Every bracketed section of* $?m(\vec{u})$, return e_{ret}, *and* new *is extended with*

$$\mathsf{h}_{comm} := \mathsf{h}_{comm} \circ ((\mathsf{kind}, \mathsf{sender}, \mathsf{receiver}, \mathsf{values})[\vec{e}/\vec{y}])\ .$$

Bracketed sections of $e_0.m(\vec{e})$ *and its subsequent receive statement update* h_{comm} *to*

$$\mathsf{if}\ e_0 = \mathsf{this}\,\mathsf{then}\ \mathsf{h}_{comm}\,\mathsf{else}\ \mathsf{h}_{comm} \circ ((\mathsf{kind}, \mathsf{sender}, \mathsf{receiver}, \mathsf{values})[\vec{e}/\vec{y}])\ .$$

The value of kind *is from the set* $\{\mathsf{new}^c, !m, ?m, !\,\mathsf{return}, ?\,\mathsf{return}\}$ *for bracketed sections creating an object of type c, sending* $!m$ *or receiving* $?m$ *a method call,* $!\,\mathsf{return}$ *for returning, and* $?\,\mathsf{return}$ *for receiving a return value. The value of* sender *is given by the sender object and sender thread's identity, i.e.,* (this, thread) *for method invocation, return, and object creation,* (e_0, thread) *for receive, and* (callerobj, caller_thread) *for bracketed sections of* $?m(\vec{u})$*, where* callerobj *is the first component of the variable* caller*. Conversely, the value of* receiver *is* (this, thread) *for* $?m(\vec{u})$ *and receive,* (e_0, thread) *for the invocation of methods different from* start*,* (e_0, e_0) *for the invocation of* start*,* (callerobj, caller_thread) *for* return*, and* (u, nil) *for object creation* $u := \mathsf{new}^c$*. Finally,* values *are the actual and formal parameter lists for method invocation, and the return value for return and receive statements; for object creation,* values *is empty.*

In the update of the history variable h_{inst}, the expression $(\vec{x}, \vec{u})[\vec{e}/\vec{y}]$ identifies the active thread by the local variables thread and conf, and specifies its instance local state after the execution of the assignment. Note that especially the values of the auxiliary variables introduced in the augmentation are recorded in the history h_{inst}. In the following we will also write (σ_{inst}, τ) when referring to elements of h_{inst}.

Next we introduce the annotation for the augmented program. In the following let $\omega \in \Omega$, $\sigma_{inst} \in \Sigma_{inst}$, and $\tau \in \Sigma_{loc}$ with $\alpha = \sigma_{inst}(\mathsf{this})$.

Definition 10 (Reachability annotation).

1. $\omega, \sigma \models_{\mathcal{G}} GI$ *iff there exists a reachable* $\langle T, \sigma' \rangle$ *such that* $dom(\sigma) = dom(\sigma')$*, and for all* $\alpha \in dom(\sigma)$, $\sigma(\alpha)(\mathsf{h}_{comm}) = \sigma'(\alpha)(\mathsf{h}_{comm})$.
2. *For each class c, let* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} I_c$ *iff there exists a reachable* $\langle T, \sigma \rangle$ *such that* $\sigma(\alpha) = \sigma_{inst}$.
3. (a) *For assertions at control points,* $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} pre(stm)$ *iff there is a reachable* $\langle T, \sigma \rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$ *and* $(\alpha, \tau, stm; stm') \in T$.
   (b) $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} post(e_0.m(\vec{u}))$ *iff there is a reachable* $\langle T, \sigma \rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$*, and a local configuration* $(\alpha, \tau, \langle e_0.m(\vec{u}); \vec{y} := \vec{e} \rangle; stm) \in T$ *which is enabled to execute. The postconditions of* return e_{ret}, !signal_all*, and* ?signal *statements are analogous.*

(c) $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} post(!\mathsf{signal})$ *iff there is a reachable* $\langle T, \sigma\rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$*, and a local configuration* $(\alpha, \tau', stm) \in T$ *which is enabled to signal the thread of* β *with* $\tau = \tau'[\mathsf{partner} \mapsto (\beta, n)]$.

(d) $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} post(\mathsf{receive}\, u_{ret})$ *iff there is a reachable* $\langle T, \sigma\rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$*, and a local configuration* $(\alpha, \tau', \langle \mathsf{receive}\, u_{ret}; \vec{y} := \vec{e}\rangle; stm) \in T$ *with* $\tau = \tau'[u_{ret} \mapsto v_{ret}]$ *which is enabled to receive the value* v_{ret}.

(e) $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} post(u := \mathsf{new})$ *iff there is a reachable* $\langle T, \sigma\rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$*, and a local configuration* $(\alpha, \tau', \langle u := \mathsf{new}; \vec{y} := \vec{e}\rangle; stm) \in T$ *with* $\tau = \tau'[u \mapsto \beta]$ *which is enabled to create the object* β.

(f) $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} post(?m(\vec{u}))$ *iff there is a reachable* $\langle T, \sigma\rangle$ *with* $\sigma(\alpha) = \sigma_{inst}$*, in that the invocation of method* m *of* α *is enabled with parameters* $\tau(\vec{u})$*; furthermore,* $\tau(\vec{v}) = \mathsf{Init}(\vec{v})$.

(g) $pre(?m(\vec{u})) = post(body_{m,c}) = I_c$ *for each class* c *and method* m *of* c.

It can be shown that these assertions are expressible in the assertion language [21]. The augmented program together with the above annotation build a proof outline that we denote by *prog'*.

What remains to be shown for completeness is that the proof-outline *prog'* indeed satisfies the verification conditions of the proof system. Initial and local correctness are straightforward.

Completeness for the interference freedom test and the cooperation test are more complex, since, unlike initial and local correctness, the verification conditions in these cases mention more than one local configuration in their respective antecedents. Now, the reachability assertions of *prog'* guarantee that, when satisfied by an instance local state, there *exists* a reachable global configuration responsible for the satisfaction. So a crucial step in the completeness proof for interference freedom and the cooperation test is to show that individual reachability of two local configurations implies that they are reachable in a *common* computation. This is also the key property for the history variables: they record enough information such that they allow to uniquely determine the way a configuration has been reached; in the case of instance history, uniqueness of course, only as far as the chosen instance is concerned. This property is stated formally in the following local merging lemma.

Lemma 10 (Local merging lemma). *Let* $\langle T_1, \sigma_1\rangle$ *and* $\langle T_2, \sigma_2\rangle$ *be two reachable global configurations of prog' and* $(\alpha, \tau, stm) \in T_1$ *with* $\alpha \in dom(\sigma_1) \cap dom(\sigma_2)$*. Then* $\sigma_1(\alpha)(\mathsf{h}_{inst}) = \sigma_2(\alpha)(\mathsf{h}_{inst})$ *implies* $(\alpha, \tau, stm) \in T_2$.

For completeness of the cooperation test, connecting two possibly different instances, we need an analogous property for the communication histories. Arguing on the global level, the cooperation test can assume that two control points are individually reachable but agreeing on the communication histories of the objects. This information must be enough to ensure common reachability. Such a common computation can be constructed, since the internal computations of different objects are independent from each other, i.e., in a global computation, the local behavior of an object is interchangeable, as long as the external behavior does not change. This leads to the following lemma:

Lemma 11 (Global merging lemma). *Let $\langle T_1, \sigma_1\rangle$ and $\langle T_2, \sigma_2\rangle$ be two reachable global configurations of prog′ and $\alpha \in dom(\sigma_1) \cap dom(\sigma_2)$ with $\sigma_1(\alpha)(\mathsf{h}_{comm}) = \sigma_2(\alpha)(\mathsf{h}_{comm})$. Then there exists a reachable configuration $\langle T, \sigma\rangle$ with $dom(\sigma) = dom(\sigma_2)$, $\sigma(\alpha) = \sigma_1(\alpha)$, and $\sigma(\beta) = \sigma_2(\beta)$ for all $\beta \in dom(\sigma_2)\backslash\{\alpha\}$.*

Note that together with the local merging lemma this implies that all local configurations in $\langle T_1, \sigma_1\rangle$ executing in α and all local configurations in $\langle T_2, \sigma_2\rangle$ executing in $\beta \neq \alpha$ are contained in the commonly reached configuration $\langle T, \sigma\rangle$.

This brings us to the last result of the paper:

Theorem 2 (Completeness). *Given a program prog, the proof outline prog′ satisfies the verification conditions of the proof system from Section 4.2.*

References

1. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Compositional operational semantics of Java$_{MT}$. Technical Report TR-ST-02-2, Lehrstuhl für Software-Technologie, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2002.
2. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In Nielsen and Engberg [16], pages 4–20. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.
3. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS State-of-the-Art-Survey. Springer-Verlag, 1999.
4. P. America and F. de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1993.
5. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.
6. 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.
7. 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.
8. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [3].
9. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods.* Cambridge University Press, 2001.
10. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
11. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969. Also in [13].
13. C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science.* International Series in Computer Science. Prentice Hall, 1989.
14. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.

15. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
16. M. Nielsen and U. H. Engberg, editors. *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS'02)*, volume 2303 of *Lecture Notes in Computer Science*. Springer-Verlag, Apr. 2002.
17. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
18. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In Swierstra [20], pages 162–176.
19. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.
20. S. Swierstra, editor. *Proceedings of the 8th European Symposium on Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*. Springer, 1999.
21. J. V. Tucker and J. I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, volume 6 of *CWI Monograph Series*. North-Holland, 1988.
22. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. submitted for publication, 2002.
23. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml*. Object Technology Series. Addison-Wesley, 1999.