INSTITUT FÜR INFORMATIK UND PRAKTISCHE MATHEMATIK

LEHRSTUHL FÜR SOFTWARETECHNOLOGIE

A Compositional Operational Semantics for *Java*$_{MT}$

Erika Ábrahám-Mumm
Frank S. de Boer
Willem-Paul de Roever
Martin Steffen

Bericht Nr. TR-ST-02-02
15. Mai 2002

CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

A Compositional Operational Semantics for $Java_{MT}$*

May 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] CWI Amsterdam, The Netherlands

Abstract. Besides the features of a class-based object-oriented language, *Java* integrates concurrency via its thread-classes, allowing for a *multithreaded* flow of control. The concurrency model includes *shared-variable* concurrency via instance variables, *coordination* via reentrant synchronization monitors, *synchronous message passing,* and dynamic *thread creation.*
This report contains a compositional version of the semantics of $Java_{MT}$ from [1].

1 Introduction

The semantical foundations of *Java* [4] have been thoroughly studied ever since the language gained widespread popularity (see e.g. [2, 7, 3]). The research concerning *Java*'s proof theory mainly concentrates' on various aspects of *sequential* sublanguages (see e.g. [5, 8, 6]). As a first step towards a compositional proof system, this paper presents a *compositional* operational semantics for *multithreaded Java* programs.

Concentrating on the issues of concurrency, we investigate an abstract programming language $Java_{MT}$[1], a subset of *Java*, featuring dynamic object creation, method invocation, object references with aliasing, and specifically concurrency. Threads, the units of concurrency, are created as instances of specific thread-classes and share the instance variables of objects.

As a mechanism of concurrency control, methods can be declared as *synchronized,* where synchronized methods within a single object are executed by different threads mutually exclusive. A call chain corresponding to the execution of a single thread can contain several invocations of synchronized methods within the same 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.

To support a clean interface between internal and external behavior, $Java_{MT}$ does not allow qualified references to instance variables. As a consequence,

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

shared-variable concurrency is caused by simultaneous execution within a single object, but not across object boundaries. The same access discipline was followed in [1] to obtain a modular proof system, cleanly separating verification conditions on the level of instances from those on a global level, dealing with object structures and communication.

Even if the proof system was split into a local and a global level in op. cit., the semantics was presented on a global level, only. In this note, we recast the semantics of [1] in a compositional way. This means, the operational semantics is described in two stages: first we define computations of a single instance, and afterwards specify rules for composing the behavior of sets of instances, where communication between different instances is synchronized by transition labels which uniquely identify the communication partners. The semantics serves as a stepping stone to a *compositional* proof-system.

2 The programming language $Java_{MT}$

In this section we describe the language $Java_{MT}$ (*"Multi-Threaded Java"*); the syntax corresponds to the one in [1]. We start with highlighting the features of $Java_{MT}$ and its relationship to full *Java*, before formally describing its abstract syntax.

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 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*. The execution of synchronized methods within a single object by different threads is mutually exclusive, whereas non-synchronized methods do not require such coordination. Note that recursive invocations of synchronized methods on the same object are allowed, as they are executed in a single call chain by the same thread. This corresponds to the notion of re-entrant monitors.

All programs are 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. 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$_{MT}$.

2.2 Abstract syntax

As *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 Int and Bool. Besides the built-in types for integers and booleans, 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. In other words, we work with a type-annotated abstract syntax where we omit the explicit mentioning of types when no confusion can arise.

For variables, we notationally distinguish between *instance* and *local* variables. Instance variables hold the state of an object and exist throughout the object's lifetime. We do not allow qualified references to instance variables in Java$_{MT}$, i.e., objects do not have direct access to instance variables of other objects. Local variables are stack-allocated. They play the role of formal parameters and variables of method definitions and exist only 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 instance and local variables. The identity of an object is stored in its class-typed constant $\mathsf{this} \notin Var$. The set Var^t contains 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 $u, u', u_1, \ldots$ as typical elements from $TVar$.

Besides using instance and local variables, side-effect free *expressions* $e \in Exp$ are built from this, nil, and from subexpressions using the given operators. We use Exp^t to denote the set of well-typed expressions of type t. The expression this is used for self-reference within an object, and nil is a constant representing an empty reference. Expressions with *side-effects* $sexp \in SExp$ contain clauses for *object creation* and *method invocation*. The expression new^c stands for the reference to a new instance of class c. An invocation of a method with name m on object e_0 with actual parameters $e_1, \ldots, e_n$ is written as $e_0.m(e_1, \ldots, e_n)$, where $\mathcal{M}$ is an infinite set of method names containing main, start, and run.

Besides the mentioned simplifications on the type system, we impose for technical reasons the following restrictions: We require that method invocation and object creation statements contain only local variables, i.e., that none of the expressions $e_0, \ldots, e_n$ 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.

Statements $stm \in Stm$ are built from side-effect expressions and assignments of the form $x := e$, $u := e$, and $u := sexp$ by using standard control constructs like sequential composition, conditional statements, and iteration, to form composite statements. Especially, we will use ϵ to denote the empty statement.

A *method* definition $modif\ m(u_1, \ldots, u_n)\{\ stm; rexp\ \} \in Meth$ 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 $stm; rexp$. 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, either giving back a value using $\mathsf{return}\ e$, or not, written as return. 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* $c\{meth_1 \ldots meth_n\, meth_{\mathsf{start}}\, meth_{\mathsf{run}}\}$ is defined by its name c and its methods, whose names are assumed to be distinct. As mentioned earlier, all classes in Java$_{MT}$ are thread classes; all classes contain a start-method $meth_{\mathsf{start}}$ and a run-method $meth_{\mathsf{run}}$ without return values. A *program* $\langle class_1 \ldots class_n\, class_{\mathsf{main}}\rangle$, 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 contains 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.

The syntax is summarized in Table 1.

---

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

exp	$::=$	$x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp, \ldots, exp)$	$e \in Exp$	expressions
$sexp$	$::=$	$\mathsf{new}^c \mid exp.m(exp, \ldots, exp)$	$sexp \in SExp$	side-effect exp
stm	$::=$	$sexp \mid x := exp \mid u := exp \mid u := sexp$		
	$\mid$	$\epsilon \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm$		
	$\mid$	$\mathsf{while}\ exp\ \mathsf{do}\ stm \ldots$	$stm \in Stm$	statements
$modif$	$::=$	$\mathsf{nsync} \mid \mathsf{sync}$		modifiers
$rexp$	$::=$	$\mathsf{return} \mid \mathsf{return}\ exp$		
$meth$	$::=$	$modif\ m(u, \ldots, u)\{\ stm; rexp\}$	$meth \in Meth$	methods
$meth_{\mathsf{run}}$	$::=$	$modif\ \mathsf{run}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{run}} \in Meth$	run-meth.
$meth_{\mathsf{start}}$	$::=$	$\mathsf{nsync}\ \mathsf{start}()\{\ \mathsf{this.run}(); \mathsf{return}\ \}$	$meth_{\mathsf{start}} \in Meth$	start-meth.
$meth_{\mathsf{main}}$	$::=$	$\mathsf{nsync}\ \mathsf{main}()\{\ stm; \mathsf{return}\ \}$	$meth_{\mathsf{main}} \in Meth$	main-meth.
$class$	$::=$	$c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{\mathsf{start}}\}$	$class \in Class$	class defn's
$class_{\mathsf{main}}$	$::=$	$c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{\mathsf{start}}\ meth_{\mathsf{main}}\}$	$class_{\mathsf{main}} \in Class$	main-class
$prog$	$::=$	$\langle class \ldots class\ class_{\mathsf{main}} \rangle$		programs

Table 1. $Java_{MT}$ abstract syntax

3 Semantics

Next, we define compositionally 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 3.2 by labeled transitions between program configurations. The semantics is given in two levels. Transitions on the local level describe the behavior of a single instance, where we distinguish self-calls from non-self-calls. The combined behavior of collections of instances is formulated on the global level, where different objects communicate by label synchronization. The semantics described here is equivalent to the one presented *non-compositionally* in [1], where the behavior was given by a number of interacting threads or execution stacks, working on a global state.

3.1 States and configurations

To give meaning to variables, we first fix the domains Val^t of the various types t. Thus Val^{Int} and Val^{Bool} denote the set of integers and booleans, $Val^{\mathsf{list}\,t}$ are finite sequences over values from Val^t, and $Val^{t_1 \times t_2}$ stands 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. We will write Val^{Object} for $\bigcup_{c \in \mathcal{C}} Val^c$. 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 $\bigcup_t Val^t_{nil}$.

The configuration of a program is characterized by the configurations of all existing instances, where in each instance, a number threads may be executing, each with its own local state and all sharing the instance state.

A *local state* $\eta \in \Sigma_{loc}$ of a thread holds the values of its local variables and is modeled as a partial function of type $TVar \rightharpoonup Val_{nil}$. We denote by η^{init} local states which assign to each class-typed local variable of type c' from their domain the value of $nil^{c'}$, to each boolean variable the value *false*, and to each integer variable the value 0. Pairs are initialized correspondingly; sequences are initially empty. A *local configuration* (η, stm) of a thread specifies, in addition to its local state, its point of execution represented by the statement *stm*.

The state of an object is characterized by its *instance state* $\sigma_{inst} \in \Sigma_{inst}$ of type $IVar \mathbin{\dot\cup} \{\mathsf{this}\} \rightharpoonup Val_{nil}$ which assigns values to its instance variables; we require that $\mathsf{this} \in dom(\sigma_{inst})$ and that $\sigma_{inst}(\mathsf{this}) \in Val^{\mathsf{Object}}$. [4] The initial instance state σ_{inst}^{init} assigns to each variable from its domain of type c', Bool, and Int the initial values $nil^{c'}$, *false*, and 0, respectively. Pairs are initialized correspondingly; sequences are initially empty. An *instance configuration* $(\sigma_{inst}, \gamma_{loc})$ consists of an instance state paired with a finite set γ_{loc} of local configurations of the threads currently executing within the instance.

Finally, a *global configuration* γ specifies a finite set of instance configurations.

Given a global configuration γ, we can use the values for the self-references this in the instance states to define what it means for an object to exist in γ. So let the set of *existing* objects of type c defined as $dom^c(\gamma) = \{\alpha \in Val^c \mid \exists(\sigma_{inst}, \gamma_{loc}) \in \gamma \,.\, \sigma_{inst}(\mathsf{this}) = \alpha\}$; the set $dom^c_{nil}(\gamma)$ is given by $dom^c(\gamma) \cup \{nil^c\}$. For the set of objects $\bigcup_c dom^c(\gamma)$ we write $dom^{\mathsf{Object}}(\gamma)$, and correspondingly for $dom^{\mathsf{Object}}_{nil}(\gamma)$. For the built-in types Int and Bool we define dom^t and dom^t_{nil}, independently of γ, as the set of pre-existing values Val^{Int} and Val^{Bool}, respectively. For compound types, dom^t and dom^t_{nil} are defined correspondingly. We refer to the set $\bigcup_t dom^t(\gamma)$ by $dom(\gamma)$; $dom_{nil}(\gamma)$ denotes $\bigcup_t dom^t_{nil}(\gamma)$.

Expressions $e \in Exp$ are evaluated with respect to an *instance local* state $(\sigma_{inst}, \eta) \in \Sigma_{inst} \times \Sigma_{loc}$, where the instance state defines the identity and values of the instance variables of the object $\sigma_{inst}(\mathsf{this})$ in which the expression is evaluated, and η gives values to the local variables. This means, the semantic function $[\![\_]\!]_{\mathcal{E}} : (\Sigma_{inst} \times \Sigma_{loc}) \to (Exp \rightharpoonup Val)$ shown in Table 2 evaluates in the context of an instance local state (σ_{inst}, η) all expressions containing only variables from $dom(\sigma_{inst}) \cup dom(\eta)$: Instance variables x and local variables u are evaluated to $\sigma_{inst}(x)$ and $\eta(u)$, respectively. The value of this refers to the object in which the expression is evaluated, the value of nil is given by the empty reference *nil*. Finally, the evaluation of compound expressions is defined by homomorphic lifting.

For a local state η, a local variable $u \in dom(\eta)$ of type t, and a value $v \in Val^t_{nil}$, we denote by $\eta[u \mapsto v]$ the local state which assigns v to u and agrees

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

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

Table 2. Expression evaluation

with η on the values of all other variables . The semantic update $\sigma_{inst}[x \mapsto v]$ of instance states is defined analogously. We use these operators analogously for setting the values of a sequence of variables. We use $\eta[\vec{y} \mapsto \vec{v}]$ also for arbitrary variable sequences, where instance variables are untouched, i.e., $\eta[\vec{y} \mapsto \vec{v}]$ is defined by $\eta[\vec{u} \mapsto \vec{v}_u]$, 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.

3.2 Operational semantics

Computation steps of a program are represented by labeled transitions between global configurations. The operational semantics is given in two stages: first we describe the behavior of a single instance and afterwards the combined behavior of sets of instances, both as labeled transition system between instance configurations, respectively between global configurations.

To be able to synchronize communicating partners in the parallel composition semantics, we have to identify local configurations being in caller-callee relationship. To do so, we extend the local state domains with the variables callerobj and id of types Object and Object $\times$ Int, resp., which may not occur in programs. The value of callerobj stores the identity of the caller object in the local state of the callee. We identify a local configuration by the thread to which it belongs together with its position in the thread's call chain. Thus the first component of id identifies the executing thread via the object in which it has begun its execution and the second component stores the position of the local configuration in the call chain of the thread. Note that this identification is unique, since at most one thread can begin its execution in a single object.

Using these identities, we define the predicates $callee(\alpha, n) = (\alpha, n+1)$ and $caller(\alpha, n+1) = (\alpha, n)$, for all $n \geq 0$. For the first local configuration in a call chain we define $caller(\alpha, 0) = (nil, 0)$. With the above identification mechanism we can express that two local configurations belong to the same thread using the predicate $samethread((\alpha_1, n_1), (\alpha_2, n_2))$ iff $\alpha_1 = \alpha_2$. That a local configuration occurs earlier than another in the call chain of a single thread, is captured by $(\alpha_1, n_1) < (\alpha_2, n_2)$ iff $\alpha_1 = \alpha_2$ and $n_1 < n_2$.

As *synchronization labels*, we distinguish $\beta!m(\alpha, id, \vec{v})$ and $\beta?m(\alpha, id, \vec{v})$ for sending and receiving method calls, respectively, where method m of the callee object β is invoked with actual parameters $\vec{v}$, and where the local configuration of the caller executing in the object α is identified by id. In analogy, we use $\alpha!(\beta, id, v)$ and $\alpha?(\beta, id, v)$ for sending, resp. receiving the value v exchanged when returning to α from a method of β executed in the local configuration identified by id. For methods without a return value the value v is omitted. For a terminating thread, the caller object to which the control returns is given by the value nil. Though the fact that a thread terminates is captured by the label $nil!(\beta, id)$.[5] Creating a new instance α is indicated by the label $new(\alpha)$. Finally, we use τ to label internal steps. We write Lab for the set of labels with l as typical element. Furthermore, we will use l_{com} as typical element for labels representing communication, i.e., different from τ and all object creation labels $new(\alpha)$, and write $sender(l_{com})$ and $receiver(l_{com})$ to denote the sender, respectively the receiver, of the message, as fixed in l_{com}.

Now, Tables 3 and 4 define the transition relation $\longrightarrow^l$ between instance configurations and Table 5 between global configurations. For notational convenience, we will later simple write $\longrightarrow$ when leaving l unspecified.

We start with the rules for transitions for one instance. Assignments to instance and local variables update the instance state, respectively the local state (cf. $\textsc{Ass}_{inst}$ and $\textsc{Ass}_{loc}$). Executing $u := \mathsf{new}^c$, as shown in rule $\textsc{New}$,[6] has no local effect except that it stores the new object's identity in the local variable u. The creation of the new object itself and the initialization of its instance variables is dealt with at the global level. The predicate *fresh* expresses that a given configuration does not refer to an object identity, i.e., that the object identity is fresh in the given context. Formally, for an object α and a value $v \in Val_{nil}$ we define:

$$fresh(v,\alpha) = \begin{cases} false & \text{if } v = \alpha \\ true & \text{if } v \neq \alpha \wedge v \in Val^{\mathsf{Bool}} \cup Val^{\mathsf{Int}} \cup Val^{\mathsf{Object}}_{nil} \\ fresh(v_1,\alpha) \wedge fresh(v_2,\alpha) & \text{if } v = (v_1,v_2) \in \bigcup_{t_1,t_2} Val^{t_1 \times t_2}_{nil} \\ \forall v_i \in v.\, fresh(v_i,\alpha) & \text{if } v \in \bigcup_t Val^{\mathsf{list}\,t}_{nil}\,. \end{cases}$$

[5] A thread of a well-typed program cannot return a value when terminating, since the start-method is of type Void. Therefore, v is left out of the label. Note also, that a terminating thread will send as id the value $(\beta, 0)$, since terminating means, popping-off from topmost frame with depth 0 from the call-stack.

[6] The statement new^c is handled similarly but without changing the local state.

$$\frac{}{(\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, x := e; stm)\}) \longrightarrow^{\tau} (\sigma_{inst}[x \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma_{inst},\eta}], \gamma_{loc} \dot\cup \{(\eta, stm)\})} \text{ ASS}_{inst}$$

$$\frac{}{(\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, u := e; stm)\}) \longrightarrow^{\tau} (\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta[u \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma_{inst},\eta}], stm)\})} \text{ ASS}_{loc}$$

$$\frac{fresh((\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, u := \mathsf{new}; stm)\}), \alpha)}{(\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, u := \mathsf{new}; stm)\}) \longrightarrow^{new(\alpha)} (\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta[u \mapsto \alpha], stm)\})} \text{ NEW}$$

$$\frac{\begin{array}{c}[\![e_0]\!]_{\mathcal{E}}^{\sigma_{inst},\eta} = \beta \neq nil \qquad \sigma_{inst}(\mathsf{this}) = \alpha \neq \beta \qquad id = \eta(\mathsf{id}) \qquad \vec{v} = [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\eta} \\ \text{if } m = \mathsf{start} \text{ then } stm' = stm \text{ else } stm' = \mathsf{return}?u; stm \text{ fi}\end{array}}{(\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, u := e_0.m(\vec{e}); stm)\}) \longrightarrow^{\beta!m(\alpha,id,\vec{v})} (\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, stm')\})} \text{ CALL}_{out}$$

$$\frac{\begin{array}{c}\sigma_{inst}(\mathsf{this}) = \beta \in Val^{c} \qquad \beta \neq \alpha \in Val^{\mathsf{Object}} \qquad modif\ m(\vec{u})\{\ body\ \} \in Meth_c \\ sync(c, m) \rightarrow isfree(\gamma_{loc}, id) \\ (m {\neq} \mathsf{start}) \rightarrow \gamma'_{loc} = \{(\eta^{init}[\vec{u} \mapsto \vec{v}][\mathsf{id} \mapsto callee(id)][\mathsf{callerobj} \mapsto \alpha], body)\} \\ (m{=}\mathsf{start} \wedge started(\gamma_{loc}, \beta)) \rightarrow \gamma'_{loc} = \emptyset \\ (m{=}\mathsf{start} \wedge \neg started(\gamma_{loc}, \beta)) \rightarrow \gamma'_{loc} = \{(\eta^{init}[\mathsf{id} \mapsto (\beta, 0)][\mathsf{callerobj} \mapsto nil], body)\}\end{array}}{(\sigma_{inst}, \gamma_{loc}) \longrightarrow^{\beta?m(\alpha,id,\vec{v})} (\sigma_{inst}, \gamma_{loc} \dot\cup \gamma'_{loc})} \text{ CALL}_{in}$$

$$\frac{\sigma_{inst}(\mathsf{this}) = \beta \neq \alpha = \eta(\mathsf{callerobj}) \qquad id = \eta(\mathsf{id}) \neq (\beta, 0) \qquad v = \eta(u)}{(\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, \mathsf{return}\ u)\}) \longrightarrow^{\alpha!(\beta,id,v)} (\sigma_{inst}, \gamma_{loc})} \text{ RETURN}_{out}$$

$$\frac{\sigma_{inst}(\mathsf{this}) = \alpha \neq \beta \in Val^{\mathsf{Object}} \qquad id = \eta(\mathsf{id})}{(\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, \mathsf{return}?u; stm)\}) \longrightarrow^{\alpha?(\beta,callee(id),v)} (\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta[u \mapsto v], stm)\})} \text{ RETURN}_{in}$$

$$\frac{\sigma_{inst}(\mathsf{this}) = \beta \qquad \eta(\mathsf{callerobj}) = nil}{(\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, \mathsf{return})\}) \longrightarrow^{nil!(\beta,id)} (\sigma_{inst}, \gamma_{loc} \dot\cup \{(\eta, \epsilon)\})} \text{ TERMINATE}$$

Table 3. Operational semantics of an instance

For local states η, instance configurations $(\sigma_{inst}, \gamma_{loc})$, and global configurations γ, the predicate *fresh* is defined by

$$\begin{aligned}
\mathit{fresh}(\eta, \alpha) &\iff \forall u \in \mathit{dom}(\eta).\mathit{fresh}(\eta(u), \alpha) \\
\mathit{fresh}(\sigma_{inst}, \alpha) &\iff \forall x \in \mathit{dom}(\sigma_{inst}).\mathit{fresh}(\sigma_{inst}(x), \alpha) \\
\mathit{fresh}((\sigma_{inst}, \gamma_{loc}), \alpha) &\iff \mathit{fresh}(\sigma_{inst}, \alpha) \wedge \forall(\eta, \mathit{stm}) \in \gamma_{loc}.\mathit{fresh}(\eta, \alpha)) \\
\mathit{fresh}(\gamma, \alpha) &\iff \forall(\sigma_{inst}, \gamma_{loc}) \in \gamma.\mathit{fresh}((\sigma_{inst}, \gamma_{loc}), \alpha).
\end{aligned}$$

Objects communicate by method calls, i.e., method invocation and the corresponding returning of the result. For both types of communication, an instance can play the role of the sender or of the receiver, and the transitions carry appropriate labels to define the composed behavior. For method invocation, the caller determines the callee object and evaluates the method arguments locally. When receiving a method invocation, the callee object creates a new local configuration to evaluate the body. The identity of the caller object and the caller local configuration is communicated together with the actual parameter values via the synchronizing label to the callee object as show in the CALL-rules of Table 3.[7] The handing-back of the return value, using the caller/callee identification, is described in the two RETURN-rules of the same table.

Different threads execute synchronized methods mutually exclusive on a given object. So in case of a synchronized method, the invocation is successful only if the lock is currently free, or the invocation is executed by a thread that already possesses the lock. Whether a local configuration of a thread identified by *id* is allowed to execute a method call on an instance with local configuration set γ_{loc}, is formalized by the predicate *isfree*, defined as $\mathit{isfree}(\gamma_{loc}, id) = \mathit{true}$ iff all local configurations in γ_{loc} with a synchronized statement have an identity less or equal *id*.

The start-method is special in two respects. First, it can effectively be invoked only once on each object; further invocations of the start-method are without effect. [8] Secondly, its invocation gives rise to a new call chain, i.e., a new thread of execution. Hence, the identity of the local state is initialized to $(\beta, 0)$. The local variable callerobj is set to *nil*, since after termination of the start-method the whole thread terminates; thus the control will not be given back to the caller. Returning from a start-method or from the initial invocation of the main-method is handled by the rule TERMINATE, where the local configuration remains in the local configuration set with an empty statement, representing the terminated thread.

Left-out in Table 3 is the semantics of self-calls and self-returns; they are handled separately in Table 4 as they are local to one instance. Consequently, the steps are all labeled with τ. The rule $\text{CALL}_{\mathit{self}}$ is the counter-piece for the

[7] A rule similar to CALL_{out} not shown in the table takes care about the invocation $e_0.m(\vec{e})$ of methods without storing the return value. Note that for methods without formal parameters $\vec{e}$ and $\vec{v}$ are empty.

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

pair $\textsc{Call}_{out}$ and $\textsc{Call}_{in}$ when caller and callee object match, and similar for returning. Note that the start-method can be invoked by a self-call ($\textsc{Start}_{self}$ and $\textsc{Start}^{skip}_{self}$). Nevertheless we do not need a corresponding rule for returning from a start-method, since it does not return to the caller.

$$\frac{\begin{array}{c}[\![e_0]\!]_{\mathcal{E}}^{\sigma_{inst},\eta} = \alpha = \sigma_{inst}(\mathsf{this}) \in Val^c \qquad id = \eta(\mathsf{id}) \\ modif\ m(\vec{u})\{\ body\ \} \in Meth_c \qquad m \neq \mathsf{start} \qquad sync(c,m) \rightarrow isfree(\gamma_{loc}, id) \\ \eta' = \eta^{init}[\vec{u} \mapsto [\![\vec{e}]\!]_{\mathcal{E}}^{\sigma_{inst},\eta}][\mathsf{id} \mapsto callee(id)][\mathsf{callerobj} \mapsto \alpha]\end{array}}{(\sigma_{inst}, \gamma_{loc}\ \dot{\cup}\ \{(\eta, u := e_0.m(\vec{e}); stm)\}) \longrightarrow^{\tau} (\sigma_{inst}, \gamma_{loc}\ \dot{\cup}\ \{(\eta, \mathsf{return}?u; stm), (\eta', body)\})}\ \textsc{Call}_{self}$$

$$\frac{\begin{array}{c}\sigma_{inst}(\mathsf{this}) = \eta'(\mathsf{callerobj}) \qquad \eta(\mathsf{id}) = callee(\eta'(\mathsf{id})) \\ \eta'' = \eta'[u \mapsto [\![e]\!]_{\mathcal{E}}^{\sigma_{inst},\eta}]\end{array}}{(\sigma_{inst}, \gamma_{loc}\ \dot{\cup}\ \{(\eta, \mathsf{return}\ e), (\eta', \mathsf{return}?u; stm)\}) \longrightarrow^{\tau} (\sigma_{inst}, \gamma_{loc}\ \dot{\cup}\ \{(\eta'', stm)\})}\ \textsc{Return}_{self}$$

$$\frac{\begin{array}{c}[\![e_0]\!]_{\mathcal{E}}^{\sigma_{inst},\eta} = \beta = \sigma_{inst}(\mathsf{this}) \in Val^c \qquad \neg started(\gamma_{loc}\ \dot{\cup}\ \{(\eta, e_0.\mathsf{start}(); stm)\}, \beta) \\ \eta' = \eta^{init}[\mathsf{id} \mapsto (\beta, 0)][\mathsf{callerobj} \mapsto nil]\end{array}}{(\sigma_{inst}, \gamma_{loc}\ \dot{\cup}\ \{(\eta, e_0.\mathsf{start}(); stm)\}) \longrightarrow^{\tau} (\sigma_{inst}, \gamma_{loc}\ \dot{\cup}\ \{(\eta, stm), (\eta', body_{\mathsf{start},c})\})}\ \textsc{Start}_{self}$$

$$\frac{[\![e_0]\!]_{\mathcal{E}}^{\sigma_{inst},\eta} = \beta = \sigma_{inst}(\mathsf{this}) \qquad started(\gamma_{loc}\ \dot{\cup}\ \{(\eta, e_0.\mathsf{start}(); stm)\}, \beta)}{(\sigma_{inst}, \gamma_{loc}\ \dot{\cup}\ \{(\eta, e_0.\mathsf{start}(); stm)\}) \longrightarrow^{\tau} (\sigma_{inst}, \gamma_{loc}\ \dot{\cup}\ \{(\eta, stm)\})}\ \textsc{Start}^{skip}_{self}$$

Table 4. Operational semantics of an instance (2)

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

As for the composed behavior of more than one object, the rules are displayed in Table 5. Two instances can perform their steps interleaved, when not forced to synchronize. A component can proceed by an internal step independently of other instances (cf. rule $\textsc{Interleave}_{\tau}$) and similarly for instantiation steps (cr. rule $\textsc{Interleave}_{new}$). For communication, the sender object proceeds on its own by communicating to the environment, if the receiver of the message is not contained in the system (cf. rule $\textsc{Interleave}_{comm}$). If both communication partners are existing within the system, they synchronize on the common label (rule $\textsc{Sync}_{call}$ and $\textsc{Sync}_{return}$). A new instance configuration in its initial state finally is added in rule $\textsc{Sync}_{New}$.

Since global configurations are defined as sets, parallel composition means set union, and thus $\|$ is symmetric and associative. To maintain uniqueness of object identities in global configurations, we throughout assume in writing $\gamma_1 \parallel \gamma_2$ that $dom^{\mathsf{Object}}(\gamma_1)$ and $dom^{\mathsf{Object}}(\gamma_2)$ are disjoint.

$$\frac{(\sigma_{inst}, \gamma_{loc}) \longrightarrow^{l} (\sigma'_{inst}, \gamma'_{loc})}{\{(\sigma_{inst}, \gamma_{loc})\} \longrightarrow^{l} \{(\sigma'_{inst}, \gamma'_{loc})\}}\ \textsc{Base} \qquad \frac{\gamma_1 \longrightarrow^{\tau} \gamma'_1}{\gamma_1 \parallel \gamma_2 \longrightarrow^{\tau} \gamma'_1 \parallel \gamma_2}\ \textsc{Interleave}_{\tau}$$

$$\frac{\gamma_1 \longrightarrow^{new(\alpha)} \gamma'_1 \qquad \mathit{fresh}(\gamma_2, \alpha)}{\gamma_1 \parallel \gamma_2 \longrightarrow^{new(\alpha)} \gamma'_1 \parallel \gamma_2}\ \textsc{Interleave}_{new}$$

$$\frac{\gamma_1 \longrightarrow^{l_{com}} \gamma'_1 \qquad receiver(l_{com}) \notin dom(\gamma_2) \wedge sender(l_{com}) \notin dom(\gamma_2)}{\gamma_1 \parallel \gamma_2 \longrightarrow^{l_{com}} \gamma'_1 \parallel \gamma_2}\ \textsc{Interleave}_{comm}$$

$$\frac{\gamma_1 \longrightarrow^{\beta!m(\alpha,id,\vec{v})} \gamma'_1 \qquad \gamma_2 \longrightarrow^{\beta?m(\alpha,id,\vec{v})} \gamma'_2}{\gamma_1 \parallel \gamma_2 \longrightarrow^{\tau} \gamma'_1 \parallel \gamma'_2}\ \textsc{Sync}_{call}$$

$$\frac{\gamma_1 \longrightarrow^{\alpha!(\beta,id,v)} \gamma'_1 \qquad \gamma_2 \longrightarrow^{\alpha?(\beta,id,v)} \gamma'_2}{\gamma_1 \parallel \gamma_2 \longrightarrow^{\tau} \gamma'_1 \parallel \gamma'_2}\ \textsc{Sync}_{return}$$

$$\frac{\gamma \longrightarrow^{new(\alpha)} \gamma' \qquad \mathit{fresh}(\gamma, \alpha)}{\gamma \longrightarrow^{\tau} \gamma' \parallel \{(\sigma^{init}_{inst}[\mathsf{this} \mapsto \alpha], \emptyset)\}}\ \textsc{Sync}_{new}$$

Table 5. Parallel composition

We conclude with the definition of *initial* and *reachable* configurations. We call a configuration γ' *reachable from the configuration* γ iff there exists a computation $\gamma \longrightarrow^* \gamma'$, where $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$; we write $reach(\gamma)$ for the set of global configurations reachable from γ. A configuration γ of a program is *reachable*, if $\gamma \in reach(\gamma_0)$, where γ_0 is the initial configuration $(\sigma_{inst}^{init}, \{(\eta^{init}[\mathsf{id} \mapsto (\alpha, 0)][\mathsf{callerobj} \mapsto nil], body_{\mathsf{main}})\})$ with main class c and $\alpha \in Val^c$.

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 that an initial thread starts to execute its main-method. Note that according to the definition of the *started* predicate, the start-method of the initial object cannot be invoked.

4 Conclusion

The paper presented a compositional operational semantics of $Java_{MT}$, where the semantics of a system is described compositionally from the behavior of its instances. The semantics coincides with the one presented non-compositionally in [1]. The formalization presented here, i.e., the thread-identification mechanism, the design of the operational rules, the information added to the labels etc., was inspired by the modular proof-system of [1]. We consider this work as an important step towards a compositional proof-system for $Java_{MT}$.

As further work, we plan to extend $Java_{MT}$ by further constructs, especially adding further synchronization primitives for monitor synchronization such as wait and notify, but also extending the language in the direction of "object-orientedness", adding inheritance, subtyping, and other concepts featured in *Java*.

References

1. E. Ábrahám-Mumm, F. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In M. Nielsen and U. H. Engberg, editors, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS'02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, Apr. 2002.
2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS State-of-the-Art-Survey. Springer-Verlag, 1999.
3. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [2].
4. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
5. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.
6. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.

7. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.
8. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. submitted for publication, 2002.

A Notation

typ. element	symbol	definition	explanation
		Semantics	
α	Val^c		object identities
	Val^c_{nil}	$Val^c \dot\cup \{nil^c\}$	identities or *nil*
v	Val^t		values of type t
	Val	$\bigcup_t Val^t$	all values (except *nil*'s)
	Val_{nil}	$\bigcup_t Val^t_{nil}$	all values
η	Σ_{loc}	$TVar \rightharpoonup Val_{nil}$	local state
(τ, stm)		$\Sigma_{loc} \times Stm$	local configuration
σ_{inst}	Σ_{inst}	$IVar \dot\cup \{\mathsf{this}\} \rightharpoonup Val_{nil}$,	instance state
$\gamma_{inst} = (\sigma_{inst}, \gamma_{loc})$	Γ_{inst}	$\Sigma_{inst} \times 2^{\Sigma_{loc} \times Stm}$	instance configuration
γ	Γ	$2^{\Gamma_{inst}}$	global configuration
$dom(\gamma)$		2^{Val}	existing values in γ
$dom^c(\gamma)$		2^{Val^c}	existing instances of class c
(σ_{inst}, η)		$\Sigma_{inst} \times \Sigma_{loc}$	instance local state
$[\![\_]\!]_{\mathcal{E}}$		$(\Sigma_{inst} \times \Sigma_{loc}) \to Exp \rightharpoonup Val_{nil}$	local evaluation function
id		$Val^{\mathsf{Object}} \times \mathsf{Int}$	originating object × depth
$started$		$2^{\Sigma_{loc} \times Stm} \times Val^c \to \mathsf{Bool}$	
$fresh$		$2^{\Sigma_{loc} \times Stm} \times Val^c \to \mathsf{Bool}$	
$callee$		$Val^{\mathsf{Object}} \times \mathsf{Int} \to Val^{\mathsf{Object}} \times \mathsf{Int}$	"$+1$"
$caller$		$Val^{\mathsf{Object}} \times \mathsf{Int} \rightharpoonup Val^{\mathsf{Object}} \times \mathsf{Int}$	"-1"
l	Lab		labels