A Compositional Operational Semantics for $\textit{Java}_{MT}$[⋆]

August 11, 2003

Erika Ábrahám[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.*
In this paper we propose a class-based compositional operational semantics for multithreaded *Java* which provides a semantic characterization and a formal basis for further semantic investigations involving inheritance, subtyping, and full abstraction, and a compositional proof system.

From its inception, *Java* [10] attracted interest from the formal methods community: The widespread use of *Java* across platforms made the need for formal studies and verification support more urgent, the grown awareness and advances of formal methods for real-life applications and languages made it more acceptable, and last but not least the array of non-trivial language features made it challenging and interesting. Thus, *Java* offered a rich field for formal studies, ranging from formal semantics [14, 5] over bytecode verification and static analysis [13] to model checking [11].

In this paper we propose a class-based compositional operational semantics for multithreaded *Java* which provides a semantic characterization of the behavioral interface of a class and a formal basis for further semantic investigations involving inheritance, subtyping, and full abstraction.

Java offers concurrency in the form of threads integrated in its class-based object-oriented framework: The concurrent entities in the run-time system of *Java* consist of the different call-chains, the threads, which execute in parallel and which share the state space grouped into objects. Thus, concurrency arises in *Java* at two levels: sets of objects in parallel cooperate via method calls, and objects processing different operations at the same time on a shared state space, namely the states of objects.

⋆ 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). © All rights remain with the authors

At the syntactic level, a *Java* program consists of classes, and object-oriented features like inheritance are defined and understood basically in terms of classes. In this paper we provide a *compositional* semantic basis of this class-based view. It provides a generic description of the behavioral interface of a class which consists of the externally observable behavior of all of its instances. In other words, we show that the behavioral interface of a class in *Java* allows to abstract from the internal shared-variable concurrency, i.e., it allows to disentangle the two levels of concurrency.

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. The same access discipline was followed in [4] 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 paper, we recast the semantics of [4] to become compositional. 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.

1 The Programming Language *Java*$_{MT}$

In this section we describe the language *Java*$_{MT}$ (*"Multi-Threaded Java"*); the syntax corresponds to the one in [4]. We start with highlighting the features of *Java*$_{MT}$ and its relationship to full *Java*, before formally describing its abstract syntax.

1.1 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 $Java_{MT}$.

1.2 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. For each type, the corresponding value domain is equipped with a standard set of operators with typical element f.

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* variables and stack-allocated *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.

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. 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 a clean separation of local, object-internal behavior and a global one. For instance, it makes possible a proof system with separated verification conditions for interference freedom and cooperation. It should be clear that one can 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.

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

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

2 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 2.2 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 [4], where the behavior was given by a number of interacting threads or execution stacks, working on a global state.

2.1 States and Configurations

To give meaning to variables, we first fix the domains Val^t of the various types t. Thus $\mathit{Val}^{\mathsf{Int}}$ and $\mathit{Val}^{\mathsf{Bool}}$ denote the set of integers and booleans, $\mathit{Val}^{\mathsf{list}\, t}$ are finite sequences over values from Val^t, and $\mathit{Val}^{t_1 \times t_2}$ stands for the product $\mathit{Val}^{t_1} \times \mathit{Val}^{t_2}$. 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 $\mathit{Val}^{\mathsf{Object}}$ for $\bigcup_{c \in \mathcal{C}} \mathit{Val}^c$. For each class name c, $\mathit{nil}^c \notin \mathit{Val}^c$ 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 $\mathit{Val}^c_{\mathit{nil}}$ as $\mathit{Val}^c \mathbin{\dot\cup} \{\mathit{nil}^c\}$, and correspondingly for compound types. The set of all possible non-nil values $\bigcup_t \mathit{Val}^t$ is written as Val, and $\mathit{Val}_{\mathit{nil}}$ denotes $\bigcup_t \mathit{Val}^t_{\mathit{nil}}$.

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 $IVar \,\dot\cup\, \{\mathsf{this}\} \rightharpoonup Val_{nil}$ 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. The semantic function $[\![\_]\!]_{\mathcal{E}} : (\Sigma_{inst} \times \Sigma_{loc}) \rightarrow (Exp \rightharpoonup Val)$ is defined by induction on the structure of expressions: 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 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

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

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.

2.2 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 $\mathsf{callerobj}$ and id of types Object and $\mathsf{Object} \times \mathsf{Int}$, resp., which may not occur in programs. The value of $\mathsf{callerobj}$ 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 $new(\alpha)$.

[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.

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 2 and 3 define the transition relation $\longrightarrow^{l}$ between instance configurations and Table 4 between global configurations. For notational convenience, we will later simply write $\longrightarrow$ when leaving l unspecified.

We start with the rules for transitions of a single instance. Assignments to instance and local variables update the instance state, respectively the local state (cf. rules $\textsc{Ass}_{inst}$ and $\textsc{Ass}_{loc}$). Executing $u := \mathsf{new}^c$, as shown in rule 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.

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 2.[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 $isfree(\gamma_{loc}, id) = true$ 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.

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

[7] A rule similar to $\textsc{Call}_{out}$ not shown in the table takes care about the invocation $e_0.m(\vec{e})$ of methods without storing the return value.

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

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

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

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

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

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

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

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

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

Table 2. Operational semantics of an instance

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 from Table 2 is the semantics of self-calls and self-returns; they are handled separately in Table 3 as they are local to one instance. Consequently, the steps are all labeled with τ. The rule CALL$_{self}$ is the counter-piece for the pair CALL$_{out}$ and CALL$_{in}$ when caller and callee object match, and similar for returning. Note that the start-method can be invoked by a self-call (START$_{self}$ and START$_{self}^{skip}$). Nevertheless we do not need a corresponding rule for returning from a start-method, since it does not return to the caller.

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

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

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

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

Table 3. 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 4. 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 INTERLEAVE$_{\tau}$). 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; dually for when receiving a communication from the environment, the sender must not be resident 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 $\textsc{Sync}_{return}$). We have three rules for instantiation: Rule $\textsc{New}_{out}$ creates a new object without including it into the configuration. A new instance configuration in its initial state is added in rule $\textsc{New}_{in}$. Finally, synchronization for creator and created configurations is formulated 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.

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

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

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

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

$$\frac{fresh(\gamma, \alpha)}{\gamma \longrightarrow^{?new(\alpha)} \gamma \parallel \{(\sigma^{init}_{inst}[\mathsf{this} \mapsto \alpha], \emptyset)\}}\ \textsc{New}_{in}$$

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

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

Table 4. Parallel composition

We express by $\gamma \longrightarrow^{*}_{\pi} \gamma'$ that there is a computation leading from γ to γ', where π is the sequence of the transition labels of the given computation without the internal τ labels. We use ϵ for the empty label sequence. Let furthermore $\Pi(\gamma)$ be the set of all label sequences π for which $\gamma \longrightarrow_{\pi}^{*} \gamma'$.

The non-compositional semantics in [4] is defined by transitions between global configurations $\langle T, \sigma\rangle$, where σ is a global state of type $Val^c \rightharpoonup \Sigma_{inst}$ assigning an instance state to each existing object, and where T is a set containing the stacks of all executing threads. To relate the non-compositional semantics to the compositional one, let for a global configuration γ, $\sigma(\gamma)$ denote the global state with domain $dom(\sigma) = dom(\gamma)$ assigning to each existing object the same instance state as defined by γ, and let $T(\gamma)$ be the set containing all local configurations in γ, extended with the information, in which instance the execution takes place, where the local configurations of each single thread are grouped into a stack structure. Let $\Longrightarrow$ denote transitions of the non-compositional semantics, and $\Longrightarrow^*$ its transitive closure. The equivalence of the compositional and the non-compositional semantics is stated in the following lemma:

Lemma 1 (Equivalence).

$$\gamma \longrightarrow_{\epsilon}^{*} \gamma' \quad \textit{iff} \quad \langle T(\gamma), \sigma(\gamma)\rangle \Longrightarrow^{*} \langle T(\gamma'), \sigma(\gamma')\rangle .$$

An interesting property of the parallel composition semantics is, that is abstracts from the internal computation steps of an object. That means that we can replace in a global computation of a program the local computation of an object by another local computation, which generates the same label trace, i.e., which has the same external behavior. In this way, we abstract on the one hand from the internal non-determinism, caused by scheduling for interleaving between different threads within the same object. On the other hand, we abstract from the internal implementation of objects: A sufficient condition for the equivalence of the external behaviour of two objects is, that their methods have the same input-output behaviour. The presented compositional semantics can be seen as a first step towards a fully abstract compositional semantics for a class-based concurrent language.

Lemma 2 (Compositionality). *Assume* $\Pi(\gamma_1) = \Pi(\gamma_2)$. *Then*

$$\gamma_1 \parallel \gamma \longrightarrow_{\pi}^{*} \gamma_1' \parallel \gamma' \quad \textit{iff} \quad \gamma_2 \parallel \gamma \longrightarrow_{\pi}^{*} \gamma_2' \parallel \gamma' .$$

It is worthwhile to point out what makes full abstractness fail for the semantics as given implicitly here as sets of labeled traces (even if we haven't bothered to spell out the reference semantics and the notion of observability against which to gauge the trace semantics). First, the semantics contains enough information to be compositional, but is unnecessarily detailed, for instance with respect to the object identifiers. Secondly, not all traces are actually realizable by a configuration, the most obvious reason being that calls and returns must be "parenthetic".

3 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 [4]. 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 [4]. We consider this work as an important step towards a compositional proof-system for *Java*$_{MT}$.

A formal semantics covering all of the *Java*-language, i.e., including multithreaded execution, and its virtual machine is given in [14] in terms of abstract state machines. Similarly, [15] presents a semantics based on ASMs. A structural operational semantics of multithreaded *Java*, based on events, can be found in [7, 6]. The work is closer to *Java*'s memory model than our interleaving semantics in distinguishing between the main memory and the local memories of the threads.

Apart from studies focusing on (sublanguages of) *Java*, there exists a vast body of calculi combining objects and concurrency, often in the tradition of the object calculi of [1] or of process calculi. We only mention here the concurrent object calculus [8][9], as it offers a combination of multithreading concurrency, objects, thread creation and aliasing similar to what is featured in *Java*. For the concurrent object calculus, [12] present a fully-abstract trace semantics for may-testing. One crucial difference to our setting, which influences also the notion of external behavior, is that those languages are *object*-based, and not based on classes.

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*. In connection with *Java*$_{MT}$'s Hoare style proof theory, for instance explored in [2], this work on compositional semantics is of interest as well, because at the heart of the completeness proof for the *modular* proof system lies an augmentation with auxiliary variables capturing the *compositional* semantics. Besides that, on the more practical side, the Hoare-style proof theory of *Java*$_{MT}$ is being implemented in a verification tool [3] which generates verification conditions for the *PVS* theorem prover. Ultimately, we consider the compositional semantics as an important step towards a compositional proof theory.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. Inductive proof-outlines for monitors in Java. In U. Nestmann and P. Stevens, editors, *FMOODS '03*, Lecture Notes in Computer Science. Springer-Verlag, Nov. 2003. To appear.

A longer version appeared as Software Technologie technical report TR-ST-03-1, April 2003.

3. E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. A tool-supported proof system for monitors in Java. In *Proceedings of the FMCO 2002*, 2002. To appear.
4. E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In M. Nielsen and U. H. Engberg, editors, *Proceedings of FoSSaCS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 4–20. Springer-Verlag, Apr. 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.
5. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS State-of-the-Art-Survey.* Springer-Verlag, 1999.
6. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From sequential to multi-threaded Java: An event-based operational semantics. In M. Johnson, editor, *Proceedings of AMAST '97*, volume 1349 of *Lecture Notes in Computer Science*, pages 75–90. Springer-Verlag, Dec. 1997.
7. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In Alves-Foss [5], pages 157–200.
8. P. Di Blasio and K. Fisher. A concurrent object calculus. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*, pages 655–670. Springer-Verlag, 1996. An extended version appeared as Stanford University Technical Note STAN-CS-TN-96-36, 1996.
9. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science Publishers, 1998.
10. J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
11. K. Havelund. *Java Pathfinder User Manual.* NASA, Aug. 1999. NASA Ames Technical Report, available at `http://ase.arc.nasa.gov/havelund/jpf.html`.
12. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02.* IEEE, Computer Society Press, July 2002.
13. X. Leroy. Java bytecode verification: An overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV '01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.
14. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.
15. C. Wallace. The semantics of the Java programming language. Technical Report CSE-TR-355-97, University of Michigan, 1997.