Distributed Algorithms: A Case Study of the Java Memory Model

Matko Botinčan, Paola Glavan, and Davor Runje

University of Zagreb

Abstract. The goal of this paper is to give a mathematically precise specification of the Java memory model and discuss its interpretation in the ASM context. We have refactored the original specification in order to clearly stipulate how it conditions the behavior of the environment. We show how each thread in a multithreaded Java program can be seen as an ordinary interactive small-step algorithm, and, consequently, how the Java program gives rise to a distributed ordinary interactive small-step ASM. Due to rather relaxed conditions on the environment imposed by the Java memory model, runs of such ASM may, however, exhibit behavior that is impossible to be observed in sequentially consistent settings. We hope that notions of run and environment capturing this kind of behavior will serve as a helpful insight for the theory of distributed algorithms developed so far.

Keywords: Java memory model, distributed algorithms, abstract state machines

1 Introduction

In order to write correct and efficient multithreaded programs that deal with the shared memory, a programmer needs a precise notion of the shared memory semantics. The memory model [1, 2] of a multithreaded programming language specifies how the actions dealing with objects in a shared memory appear to execute to the programmer. Essentially, the memory model determines the values the programmer can expect from reads of a shared variable. Namely, due to transformations on a program code performed by any of the compiler, the running environment (i.e. the virtual machine) or the actual hardware the program is executed on, the actual outcomes of reads in a program may vary drastically from the one that could intuitively be expected from its program code. The memory model specifies possible outcomes and as such is indispensable for full understanding of multithreaded programs semantics.

We note that hardware and software transformations on a program code are in fact restricted in a way so that they maintain the program's intra-thread semantics — a thread, when run in isolation, should behave as if no transformations were performed at all. The problem, however, arises when multiple threads are run at the same time, since program transformations, although safe for single-threaded executions, may cause unexpected effects in multi-threaded settings. The memory model then serves as an essential aid for the programmer by providing her a guarantee on the worst-case possible scenario.

The Java language specification [9] and recent papers on the Java memory model (JMM) [13, 14] give a precise specification of the behavior of the shared memory for multithreaded Java programs. Nevertheless, it still lacks a rigorous mathematical treatment, and the formalization of the JMM (as well as memory models in general) attracts attention as a topic of growing interest [8, 15].

Following the original work on the Java memory model [12–14], we lay down the specification of the JMM in a mathematically precise way aiming to eliminate potential ambiguities present in the source. The original specification is refactored in order to clearly separate the notion of a run of a multithreaded Java program and the notion of an environment. It allows us to explicitly formulate how the memory model conditions the behavior of the environment, and what the memory model actually represents in a mathematical sense.

The key part of the JMM specification, however, still remains declarative in style and it is not evident how to effectively check it on a given environment of a run of a Java program. Also, we have to note that the specification in this paper does not include all items that are included in [12–14]. For ease of the presentation, we only consider actions that actually interact with the shared memory, thus we omit other actions, in particular, observable actions described in [12–14] (however, it would be relatively straightforward to include them). We also leave out semantics of final fields.

Later on, we show how each thread in a multithreaded Java program can be seen as an ordinary interactive small-step algorithm [4–6], and, consequently, how the Java program gives rise to a distributed ordinary interactive small-step ASM [10]. To the best of our knowledge, present ASM papers dealing with distributed algorithms assume that the communication between agents (that is performed either via shared memory or via message passing) is accomplished in a sequentially consistent manner. Due to rather relaxed conditions on the environment imposed by the JMM, runs of ASM corresponding to a multithreaded Java program may, however, exhibit behavior that is impossible to be observed in sequentially consistent settings. A particular instance of such bizarre behavior is receiving messages that have not yet been sent.

The main contributions of the paper are therefore twofold:

– We give a mathematically precise specification of the JMM in a way so that we can clearly stipulate how it conditions the behavior of the environment.
– By giving multithreaded Java programs an interpretation in the ASM context, we come to the notion of a run of a distributed algorithm that reveals counterintuitive behavior not attributed to distributed algorithms in the literature so far.

The rest of the paper is organized as follows. Section 2 introduces basic notions needed for formal treatment of multithreaded Java programs, gives the definition of a run of such programs, and deals with the notion of its justification. Different restrictions on justifications give rise to different memory models, which is the topic of Section 3. In Section 4, we put multithreaded Java programs into the ASM context and deal with the new notion of a distributed run that allows a more general behavior than when having sequential consistency. The final Section 5 gives concluding remarks.

2 Basic Notions

2.1 Preliminaries

Let $R \subseteq X \times X$ be a binary relation on a set X. Denote with $R^\tau := \{(y, x) \in X \times X \mid xRy\}$ the transposition of R, and let $I = \{(x, x) \mid x \in X\}$ be the identity relation. We say that R is reflexive if $I \subseteq R$, R is irreflexive if $R \subseteq I^c$, R is symmetric if $R \subseteq R^\tau$, R is antisymmetric if $R \cap R^\tau \subseteq I$ and R is transitive if $R^2 \subseteq R$. Further, R is a partial order if it is reflexive, antisymmetric and transitive relation; R is a strict partial order if it is irreflexive and transitive. R is a total order if it is a partial order such that $R \cup R^\tau = X \times X$; R is a strict total order if it is a strict partial order such that $R \cup R^\tau = (X \times X) \setminus I$.

Proviso: In the rest of the text, binary relations are often denoted with $\xrightarrow{\alpha}$, for some label α. If $\xrightarrow{\alpha}$ is a irreflexive relation, its transitive closure $(\xrightarrow{\alpha})^+$ is a strict partial order which we denote with $<_\alpha$. If $\xrightarrow{\alpha}$ is a antisymmetric relation, its reflexive and transitive closure $(\xrightarrow{\alpha})^*$ is a partial order which we denote by $\leq_\alpha$.

We say that partial orders $\leq_1$ and $\leq_2$ are *consistent* if for all x and y such that $x \leq_1 y$ and $y \leq_2 x$ it follows that $x = y$ (or, in other words, if $(\leq_1 \cup \leq_2)^*$ is a partial order). Strict partial orders $<_1$ and $<_2$ are said to be consistent if for all x and y, either $x \not<_1 y$ or $y \not<_2 x$ holds (or, equivalently, if $(<_1 \cup <_2)^+$ is a strict partial order).

R-chain is a subset of X that is totally ordered by R; a descending R-chain is a R^τ-chain. Relation R is well-founded if there are no infinite descending R-chains in X. If R is a strict partial order, we say that a is R-maximal element in X if $a \in X$ and there does not exist $b \in X$ such that aRb. The set of R-maximal elements in X is denoted by $\max_R(X)$. In cases when $\max_R(X)$ is a singleton set, we identify it with the unique element it contains.

If R is a (strict) partial order, then (X, R) (or, shortly, just X) is called a (strictly) partially ordered set. An initial segment of such X is a subset $Y \subseteq X$ ordered by R such that if $a \in Y$ and bRa, then $b \in Y$. We say that X satisfies the *finiteness* property if all its initial segments are finite.

2.2 Programs

Let $\mathcal{P}$ be a multithreaded Java program that spawns a set of threads $\mathsf{Threads}$. Each thread in $\mathsf{Threads}$ is assigned a program text $P \in \mathcal{P}$ consisting of a sequence of statements where each statement gives rise to one or more actions. Since we are interested in $\mathcal{P}$'s semantics with respect to the memory model, we only deal with actions that actually interact with the shared memory. These are *reads* and *writes* (as well as *volatile reads* and *volatile writes*) of shared variables, and *locks* and *unlocks* on shared synchronization objects (monitors) in $\mathsf{Monitors}$. Additionally, in order to deal with threads' life times we also consider actions for *creating* a thread and *joining* a thread, in which threads are referred by shared identifiers in $\mathsf{Threads}$.

The shared memory is abstractly seen as a set of locations $\mathsf{Locations}$, where each location can take on a value from a set of possible values Values. For simplicity, we additionally assume that $\mathsf{Locations} \subseteq \mathsf{Values}$, as well as that $\mathsf{Monitors} \subseteq \mathsf{Values}$ and $\mathsf{Threads} \subseteq \mathsf{Values}$. Also, the sets $\mathsf{Locations}$, $\mathsf{Monitors}$ and $\mathsf{Threads}$ are mutually disjoint.

2.3 Threads

Each thread $t \in \mathsf{Threads}$ is associated with a set of states S_t that the thread can possibly reside in. A state in S_t abstracts away the representation of t at a time instance (typically, it would include values of local variables, the call stack, the program counter, contents of registers, etc.). A t's state thus contains only a "local" snapshot of the thread; e.g. values of global variables in the shared memory are not part of t's state. The sequential intra-thread semantics of Java determines how the thread steps from one state to another. Nevertheless, since we only want to track the thread's interaction with the shared memory, we choose a level of abstraction such that steps are made just between the states at which actions interacting with the shared memory take place.

In order to capture the thread's ignorance of the shared memory behavior, we model the interaction of the thread with objects in the shared memory through a sequence of queries and replies. In each state, the thread issues a query from a set of potential queries $\mathsf{Queries}$ to the shared memory. The set $\mathsf{Queries}$ is the smallest set containing queries $\langle \texttt{read}, l\rangle$, $\langle \texttt{write}, l, v\rangle$, $\langle \texttt{volatile read}, l\rangle$, $\langle \texttt{volatile write}, l, v\rangle$, $\langle \texttt{lock}, m\rangle$, $\langle \texttt{unlock}, m\rangle$, $\langle \texttt{create thread}\rangle$, $\langle \texttt{join}, t\rangle$, for all $l \in \mathsf{Locations}$, $v \in \mathsf{Values}$, $m \in \mathsf{Monitors}$ and $t \in \mathsf{Threads}$. We write $\vdash^t_X q$ if t issues a query q in a state X.

All queries except the ones having `read` or `write` as their first component are called *synchronizing*. The queries starting with `volatile` are called *volatile*. We say that queries $\langle \texttt{read}, l\rangle$ and $\langle \texttt{volatile read}, l\rangle$ are *reading* l, queries $\langle \texttt{write}, l, v\rangle$ and $\langle \texttt{volatile write}, l, v\rangle$ are *writing* v to l, the query $\langle \texttt{lock}, m\rangle$ is *locking* m, the query $\langle \texttt{unlock}, m\rangle$ *unlocking* m, the query $\langle \texttt{create thread}\rangle$ is *creating a thread*,[1] and the query $\langle \texttt{join}, t\rangle$ is *joining a thread* t.

[1] Formally, the query $\langle \texttt{create thread}\rangle$ should have as a parameter a reference to a program text that the thread that is to be created will execute. Nevertheless, in our settings, the program text is immutable and fixed in advance, thus we omit this parameter for simplicity.

The shared memory replies to each issued query either with an element of Values, or with an automatic OK if the query requires no feedback. More precisely, each reading query is replied with a $v \in \mathsf{Values}$, each query creating a thread is replied with a $t \in \mathsf{Threads}$, and a query of any other kind is replied with an automatic OK. Let $\mathsf{Replies}$ stand for the union $\mathsf{Values} \cup \{\text{OK}\}$.

Possible runs of a thread t are given by a labeled transition system TS_t with a set of states S_t and transition relations $\xrightarrow[t]{(q,r)} \subseteq S_t \times S_t$, where $q \in \mathsf{Queries}$ and $r \in \mathsf{Replies}$. The labeled transition system represents the intra-thread semantics of the thread's program text at our chosen level of abstraction. If in a state X, the thread t issues a query q (i.e. if $\vdash^t_X q$ holds) and receives r as a reply to the query, let X' be the state the thread steps to according to the intra-thread semantics. We have in TS_t a (q,r)-labeled transition $X \xrightarrow[t]{(q,r)} X'$, for each possible reply $r \in \mathsf{Replies}$ and the corresponding successor state X'.

A run segment of a thread t is an alternating sequence $r = X_0\alpha_1X_1 \dots \alpha_nX_n$ of states and transition labels in TS_t such that $X_i \xrightarrow[t]{\alpha_i} X_{i+1}$, for all $0 \leq i < n$. If r begins with t's initial state and ends with the final state of t, we just call it a run of t. The projection of r to its transition labels $\alpha_1 \dots \alpha_n$ is called a trace segment of t in the former case, and just a trace of t in the latter case, respectively.

2.4 Run

A run of a multithreaded Java program $\mathcal{P}$ captures the inter-thread actions dealing with the shared memory that have been performed by any of the threads spawned by $\mathcal{P}$. It is important to note, however, that the run represents only $\mathcal{P}$'s perception of the execution process, and thus does not include any information on how the shared memory processed the queries issued to it and how it generated the replies. From $\mathcal{P}$'s point of view, the interaction with shared memory is seen just as an interaction with an environment that accepts queries and provides replies. Later on, we shall define different restrictions on such an environment in order to capture behaviors respecting different memory models.

Definition 1. *A run of $\mathcal{P}$ is a tuple $(\mathcal{A}, \Pi, \mathcal{T}, \sigma, \mathcal{E}, \xrightarrow{po})$ where:*

- *$\mathcal{A}$ is a set of* actions*;*
- *Π assigns a program text $\Pi(a) \in \mathcal{P}$ to every action $a \in \mathcal{A}$;*
- *$\mathcal{T}$ assigns a thread $\mathcal{T}(a) \in \mathsf{Threads}$ to every action $a \in \mathcal{A}$ such that actions of the same thread correspond to the same program text, i.e.:*

$$\textit{for all } a_1, a_2 \in \mathcal{A}, \quad \mathcal{T}(a_1) = \mathcal{T}(a_2) \implies \Pi(a_1) = \Pi(a_2);$$

- *σ assigns a state in $S_{\mathcal{T}(a)}$ to every action $a \in \mathcal{A}$;*
- *$\mathcal{E}$ assigns a tuple (q,r) to every action $a \in \mathcal{A}$, where q is a query such that $\vdash^{\mathcal{T}(a)}_{\sigma(a)} q$ and r is a reply with a value in $\mathsf{Replies}$;*

– *The* program order $\xrightarrow{po}$ *is an irreflexive binary relation on $\mathcal{A}$ representing the successor relation induced by the transition system of each thread,*[2] *i.e.: for all $a_1, a_2 \in \mathcal{A}$ such that $\sigma(a_i) = X_i$ and $\mathcal{E}(a_i) = (q_i, r_i)$ $(i = 1, 2)$ we have $a_1 \xrightarrow{po} a_2$ iff the following holds:*
  - *there exists $t \in \mathsf{Threads}$ such that $\mathcal{T}(a_1) = \mathcal{T}(a_2) = t$, and*
  - *$X_1 \xrightarrow[t]{(q_1,r_1)} X_2$ and there exists $X_3 \in S_t$ such that $X_2 \xrightarrow[t]{(q_2,r_2)} X_3$.*

Note that $\xrightarrow{po}$ satisfies the finiteness property and that $<_{po} = (\xrightarrow{po})^+$ is a strict total order on the set of actions belonging to the same thread.

The notion of a run as given in Definition 1 does not directly correspond to the notion of the execution from [12–14]. On the one hand, our definition additionally incorporates states of each thread since they are inherent to the intra-thread semantics of a program, and, consequently, to the definition of the program order. On the other hand, we do not include the synchronization order, the write-seen function as well as their dependents, since their actual purpose, as we shall see later on, is to justify well-formedness of a particular run.

Nevertheless, the state parts of a run are intrinsic to programs' threads and the environment acts without knowing what state a particular thread resides in. In order to clearly separate the "environmental" part of the run, we introduce the notion of an environment of a run by simply ignoring the state parts.

Definition 2. *Given a run $\varrho = (\mathcal{A}, \Pi, \mathcal{T}, \sigma, \mathcal{E}, \xrightarrow{po})$ of $\mathcal{P}$, the environment of ϱ is the tuple $(\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$.*

Note, however, that since the environment does not have an insight into the threads' states, any partial order on the set of actions could potentially be a program order of an environment (formally we could define it by inspecting the order in which the environment has received actions from each thread separately and then taking the union). Yet, we do not deal with this unnecessary generality here, since we assume that an environment is defined for a particular run only, and, thereby respects the program order of the run by the definition.

Given an action $a \in \mathcal{A}$, if the query in $\mathcal{E}(a)$ is reading, writing, joining thread, locking or unlocking x, we say that the action a is also *reading, writing, joining thread, locking or unlocking x*, respectively. If a is reading from l and its reply is v, we say that a is *reading v from l*. If the query in $\mathcal{E}(a)$ is creating a thread and the reply in it is t, we say that a is *creating a thread t*. If the query in $\mathcal{E}(a)$ is volatile or synchronizing, we also say that the action a is *volatile* or *synchronizing*, respectively.

[2] The name "program order" may probably be confusing, since the program order as it has been defined is not an order. However, we have decided to keep this name (as well as the names of relations introduced further) in order to be consistent with the nomenclature in [12–14].

2.5 Justification for an environment of a run

The definition of a run of a program $\mathcal{P}$ does not include enough information to see from it how $\mathcal{P}$ has actually been executed by the system. The run only provides what has been perceived by each of $\mathcal{P}$'s threads. From a given run of $\mathcal{P}$, not only it is not possible to reconstruct the actual execution flow of $\mathcal{P}$, but it is also not evident whether such run is feasible at all, i.e. whether there actually exists an environment that would support the run. Here the notion of environment should be understood in a broader sense than as in Definition 2 — it includes everything outside the scope of a single thread, e.g. the shared memory behavior, the thread scheduler, etc.

Certain extrinsic components of the environment, however, need to be known in order to reason about runs of $\mathcal{P}$, and, more particularly, about environments of runs of $\mathcal{P}$. For memory model issues, it is deemed necessary to know the total order in which all synchronization actions appear and to know which write actions caused the values the read actions see. When knowing this information, we can say that we can *justify* a particular run of $\mathcal{P}$. The items constituting the justification, however, do not deal with threads' states, but only with the actions the threads have performed, thus, the notion of justification is defined for an environment of a run.

Definition 3. *A* justification *for an environment* $(\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ *of a run is a pair* $(\Theta, \xrightarrow{so})$ *with the following properties:*

- Θ *assigns to each (volatile) read action in* $\mathcal{A}$ *a (volatile) write action in* $\mathcal{A}$ *such that if* $\Theta(a_r) = a_w$*, for some action* a_w *writing* v *to* l*, then the action* a_r *is reading* v *from* l*.*
- *The* synchronization order $\xrightarrow{so}$ *is an irreflexive well-founded relation on the set of synchronizing actions in* $\mathcal{A}$ *such that:*
  - $<_{so}$ *is a strict total order that is consistent with* $<_{po}$*;* *in particular,* $<_{po} |_{Dom(<_{so})} \subseteq <_{so}$*;*
  - *for each action* a *such that* a *is locking* m*, the number of actions* a' *such that* $\mathcal{T}(a') \neq \mathcal{T}(a)$*,* $a' <_{so} a$ *and* a' *is locking* l *equals to the number of actions* a'' *such that* $\mathcal{T}(a'') \neq \mathcal{T}(a)$*,* $a'' <_{so} a$ *and* a'' *is unlocking* l*.*

3 Memory Models

Given a justification $(\Theta, \xrightarrow{so})$ for an environment $(\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ of a run, we define the following two orders:

- The *synchronizes-with order* $\xrightarrow{swo}$ is a restriction of $<_{so}$ that relates roughly those pairs of actions that are concerned with the same object. It is defined as the smallest binary relation over the set of synchronizing actions in $\mathcal{A}$ such that:
  - if a_1 is unlocking l, a_2 is locking l, $a_1 <_{so} a_2$, then $a_1 \xrightarrow{swo} a_2$;
  - if volatile a_1 is writing l, volatile a_2 is reading l, and $a_1 <_{so} a_2$, then $a_1 \xrightarrow{swo} a_2$;

  - if a_1 is creating a thread t and a_2 is the first action performed by t, then $a_1 \xrightarrow{swo} a_2$;
  - if a_2 is joining the thread t and a_1 is the last action performed by t, then $a_1 \xrightarrow{swo} a_2$.
- *Happens-before order* $\xrightarrow{hbo}$ is a strict partial order induced by the synchronizes-with order and the program order:

$$\xrightarrow{hbo} = (\xrightarrow{po} \cup \xrightarrow{swo})^+.$$

3.1 Happens-before consistency

The weakest requirement on a justification for an environment of a run is the happens-before consistency.

Definition 4. *A justification* $(\Theta, \xrightarrow{so})$ *for an environment* $\eta = (\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ *of a run is* happens-before consistent *if:*

- *for each volatile action* a *such that* a *is reading* l *we have:*
  - *it is not the case that* $a <_{so} \Theta(a)$*; and*
  - *there does not exist an volatile action* a' *such that* a' *is writing to* l *and* $\Theta(a) <_{so} a' <_{so} a$.
- *for each action* a *such that* a *is reading* l *we have:*
  - *it is not the case that* $a <_{hbo} \Theta(a)$*; and*
  - *there does not exist an action* a' *such that* a' *is writing to* l *and* $\Theta(a) <_{hbo} a' <_{hbo} a$.

Let us denote with $\Xi(a_r)$ the set of writing actions that a reading action $a_r \in \mathcal{A}$ is allowed to observe taking into account the happens-before order. Namely, if the action a_r is reading l, then $\Xi(a_r)$ comprise all $<_{hbo}$-maximal actions writing to l that do not happen-before after a_r, i.e.:

$$\Xi(a_r) = \max_{<_{hbo}}(\{a_w \mid a_w \text{ is writing to } l \text{ and } \neg(a_r <_{hbo} a_w)\})$$

The second requirement that the happens-before consistency places on an environment η can now be rephrased by requiring that for each action a such that a is reading l, η needs to satisfy $\Theta(a) \in \Xi(a)$.

3.2 Sequential consistency

Sequential consistency is just a special case of the happens-before consistency in which all actions occur in a total order (the execution order).

Definition 5. *A justification* $(\Theta, \xrightarrow{so})$ *for an environment* $\eta = (\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ *of a run is* sequentially consistent *if:*

- η *is happens-before consistent; and*
- *there exists a total order* $<_{eo}$ *(the execution order) on* $\mathcal{A}$ *consistent with* $<_{hbo}$ *such that for each read action* a, $\Theta(a)$ *is* $<_{eo}$*-maximal action in* $\Xi(a)$*, i.e.* $\Theta(a) = \max_{<_{eo}}(\Xi(a))$.

3.3 JMM consistency

Although easiest to deal with, sequential consistency gives rise to a memory model that is too strong to be used as a memory model for Java. It requires that the total order all actions appear in has to respect the program order, and thus forbids standard compiler or processor optimizations. On the other hand, happens-before consistency gives rise to an overly weak memory model that, although providing undoubtedly necessary guarantees, allows undesirable behaviors such as appearance of out of thin air values [12–14].

Requirements imposed by JMM lie somewhere strictly between happens-before consistency and sequential consistency. Its crucial addition to happens-before consistency is elimination of self-justifying speculative writes, so called *causal loops*. The approach to causality taken by authors of [12–14] is based on reasoning that an action should be allowed to happen if its occurrence is not dependent on an action reading a value from a data race. Namely, it is perfectly legal that in the actual execution of a program a write action can occur earlier than it appears in the program order. That write action, however, must have been able to appear in the execution without requiring that some read action gets its value via a data race. Yet, as it turns out, such causality requirement is not easy to formally define. Due to page limitation we lay out the definition capturing this notion without much explanation. For the actual motivation and the intuition behind its conditions we refer the reader to a thorough discussion in [13].

Justifying an environment $(\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ of a run with a JMM consistent justification is an iterative process. Sets of actions from $\mathcal{A}$ are *committed* stepwise so that in each step there exists a happens-before consistent justification comprising (at least) committed actions which additionally respects the causality introduced by steps.

Definition 6. *A justification* $(\Theta, \xrightarrow{so})$ *for an environment* $\eta = (\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ *of a run is* JMM consistent *if there exists a sequence of actions* $(\mathcal{C}_i)_{i\geq 0}$ *(called committed actions) such that:*

- $\mathcal{C}_0 = \emptyset$;
- *for each* $i \geq 0$, *we have* $\mathcal{C}_i \subset \mathcal{C}_{i+1}$;
- $\mathcal{A} = \bigcup_{i\geq 0} \mathcal{C}_i$;
- $(\mathcal{C}_i)_{i\geq 0}$ *is a finite sequence iff* $\mathcal{A}$ *is finite;*

and a sequence of happens-before consistent justifications $(\Theta_i, \xrightarrow{so_i})_{i\geq 1}$ *of environments* $\eta_i = (\mathcal{A}_i, \Pi_i, \mathcal{T}_i, \mathcal{E}_i, \xrightarrow{po_i})_{i\geq 1}$ *satisfying the following conditions:*[3]

- *each action in* $\mathcal{C}_i$ *corresponds to an action in* $\mathcal{A}_i$, *i.e.* $\mathcal{C}_i \subseteq \mathcal{A}_i$;

[3] Unlike in [12–14], our definition of JMM consistency includes neither the condition for so called sufficient synchronizes-with relation (since it is not needed for the JMM specification), nor the condition for external actions (since we only deal with actions that interact with the shared memory).

– *actions in $\mathcal{C}_i$ are ordered in η_i by the same synchronization order and happens-before order as in η, i.e. $\xrightarrow{so_i}|_{\mathcal{C}_i} = \xrightarrow{so}|_{\mathcal{C}_i}$ and $\xrightarrow{hbo_i}|_{\mathcal{C}_i} = \xrightarrow{hbo}|_{\mathcal{C}_i}$;*
– *read actions in $\mathcal{C}_{i-1}$ need to see the same write actions in η_i as in η, i.e. $\Theta_i|_{\mathcal{C}_{i-1}} = \Theta|_{\mathcal{C}_{i-1}}$;*
– *each read action $a_r \in \mathcal{A} \setminus \mathcal{C}_{i-1}$ sees a write action that happens-before it, i.e. $\Theta_i(a_r) <_{hbo_i} a_r$;*
– *each read action $a_r \in \mathcal{C}_i \setminus \mathcal{C}_{i-1}$ sees (not necessarily the same) writes from $\mathcal{C}_{i-1}$ in η_i and η, i.e. $\Theta_i(a_r) \in \mathcal{C}_{i-1}$ and $\Theta(a_r) \in \mathcal{C}_{i-1}$.*

3.4 Definition of memory models

We transfer the notions of consistency of justifications to consistency of environments in a natural way.

Definition 7. *An environment of a run is* sequentially *(*happens-before*,* JMM*)* consistent *if there exists a sequentially (happens-before, JMM) consistent justification for it.*

At last, we define a memory model as a set of all environments (each corresponding to some run) that satisfy the consistency property of interest. In particular, we have the following definition.

Definition 8 (Memory models).

– *Sequentially consistent memory model is the set $\{\eta \mid \eta$ is a sequentially consistent environment for some run$\}$;*
– *Happens-before memory model model is the set $\{\eta \mid \eta$ is a happens-before consistent environment for some run$\}$*
– *Java memory model is the set $\{\eta \mid \eta$ is a JMM consistent environment for some run$\}$.*

The memory model can thus be seen as a *contract* that gives to a program a guarantee on behavior of its environment — a behavior of the environment that is in accordance with the memory model will be permitted, while all undesired behaviors will be disallowed.

4 Relations to Abstract State Machines

In order to keep the model as close as possible to the original model [12–14], we minimized explicit references to ASMs in our exposition so far. However, the model we just defined *is* an ASM model. We will make this claim precise in the current section.

The main contributions of the paper with respect to the ASMs are the following:

– This is one of the first practical applications of the recently developed theory of interactive algorithms [4–6].

– We have extended the notion of distributed computation from the Lipari guide [10] in two directions:
  1. The notion of distributed run is defined with ordinary interactive small-step algorithms acting as agents. Such extension is natural and not particulary difficult. One can find an implicit justification for it in the Lipari guide:

     ”We do not suppose that agents are deterministic or do only a bounded amount of work at each step.”
  2. We further generalized the new notion of a distributed run by replacing the coherence condition by more general requirements on environment behavior such that different requirements corresponds to different memory models. As a consequence of this generalization, a distributed run does not have to be sequentially consistent. This is the first model not requiring sequential consistency in the ASM literature known to us.

We briefly remind the reader of previously written papers and a book that relate Java and ASM.

Börger and Schulte in [7], and Stärk, Schmid and Börger in the book [16] define precise semantical description of the Java language using ASMs. They also verify that compilation of the Java language to the JVM code is correct, by describing a hierarchy of natural sublanguages from Java to JVM and using refinement techniques in order to relate the models. The paper covers the language features for concurrency, but its focus is on the high-level language aspects. The former Java language specification contained a (erroneous) memory model definition (although not explicitly referred to as such in the text), however, the paper does not deal with issues related to this memory model.

Gurevich, Schulte and Wallace in [11] present the concurrent features of Java using imperative, ASM approach. They follow the earlier version of the Java Language specification manual [9]. The paper covers all aspects of Java threads and synchronization, gradually adding the details to the model and obtains a lower-level concurrency model. The paper presents a clear understanding of the Java concurrency model and discusses its consequences.

Awhad and Wallace in [3] formalize using ASMs the two memory models that have been proposed as replacements for the previously erroneous JMM. They develop a unified representation of them and relate the proposed Java memory models to the so called Location Consistency memory model and to each other.

4.1 Threads as ordinary interactive small-step algorithms

Execution of multithreaded Java programs can be viewed as a distributed computation, where threads are concurrently running sequential agents communicating through a shared global memory. Communication between threads, and between a thread and the virtual machine, is ordinary in the sense of [6]: thread issues a query in a state, waits for a reply, and then computes the next state using no information from the environment other than the reply it received. Existence of a bound on the amount of work performed between two queries is not important for modeling of memory models, but we claim there is only so much a thread can do without reading or writing a value to the shared memory. It is easy to see that threads satisfy postulates from [4–6], i.e. they

1. proceed in discrete global steps;
2. preform only a bounded amount of work in each step;
3. use only such information from the environment as can be regarded as answers to queries; and
4. never complete a step until all queries from that step have been answered,

and therefore *are* ordinary interactive small-step algorithms. More precisely,

– The (abstract) vocabulary of a thread t is determined by its associated Java program $\Pi(a)$, where a is any of its action. States of a thread can be represented by first order structures of a fixed vocabulary. A sceptical reader is referred to a wide experience of modeling Java with ASMs [7, 11, 16].
– The finite set of labels Λ used for queries is:

$$\Lambda = \{\, \texttt{read}, \texttt{write}, \texttt{volatile read}, \texttt{volatile write}, \texttt{lock}, \texttt{unlock}, \\ \texttt{create thread}, \texttt{join}\}.$$

– The causality relation $\emptyset \vdash^{\mathcal{T}(a)}_{\sigma(a)} q$ of a thread t, where a ranges over all actions of t, was already made explicit. It is a special kind of a causality relation: in each state, exactly one query is issued. This is simply our design choice determined by our primary interest — the study of memory models — but it is not the only one. E.g. we could make a single step of a thread for each statement in the original Java program. Appropriate causality relations would potentially be more complex, reflecting numerous reads and writes to shared memory in a single step at such level of abstraction. Although this generalization is straightforward, it would complicate our definition with details unnecessary for our purposes.
– The communication among threads and the environment's interactions with threads, are modeled by sequences of query-reply pairs. Query-reply pairs model the the interaction of threads with objects in shared memory. The answer function α, mapping a query q to a reply r, represents the interaction with an environment and is defined as follows: $\mathcal{E}$ assigns (q, r) to every action $a \in \mathcal{A}$, where q is a query issued in action a in accordance with the causality relation of $\mathcal{T}(a)$ in state $\sigma(a)$: $\emptyset \vdash^{\mathcal{T}(a)}_{\sigma(a)} q$, and r is a reply with a value in $\mathsf{Replies}$.
– Transitions between states are modeled by TS_t: for each possible query-reply pair (q, r) and X, the (q, r)-labeled transition $X \xrightarrow[t]{(q,r)} X'$ gives the corresponding successor state X'. This is nothing more than rephrasing the standard definition of a transition function from [6]: if $a \xrightarrow{po} a'$, $\emptyset \vdash^{\mathcal{T}(a)}_{\sigma(a)} q$, α is a function assigning r to q, and $\sigma(a) \xrightarrow[t]{(q,r)} \sigma(a')$, then $\sigma(a') = \tau_{\mathcal{T}(a)}(\sigma(a'), \alpha)$.

Note that, contrary to the Lipari guide [10], we do not have a global memory, and local states of our threads (ASM agents) are not local views of the global memory - i.e. they are not $View_a(S)$ reducts of global state S to local agents' vocabulary. Also, by implicit assumption of the Java language specification, program $\mathcal{P}$ is finite, and also there are only finitely many threads in each execution of a multithreaded Java program.

4.2 Multithreaded Java programs as distributed ordinary interactive small-step ASMs

The notion of a distributed run in the ASM context has first been defined in the Lipari guide [10] for a particular type of ASMs called the *distributed ASMs* (DASMs). DASM, as defined in [10], consists of a finite set of single-agent programs, each executing at its own pace and eventually communicating through the global memory. The definition also includes the vocabulary comprising vocabularies of each agent's program, and the corresponding set of states. The definition, however, does not presuppose that an agent has to be represented by a particular kind of ASM — any one known at the time of the writing of [10] (namely, either a non-interactive small-step or a wide-step ASM) is allowed to be taken. One can also reason about sequential, quasi-sequential or, most generally, partially-ordered runs of such DASMs.

A partially ordered run of a DASM consists of a partially ordered set of moves (representing actions performed by each of the agents) satisfying the finiteness property, and such that the moves of any single agent are totally ordered. Further, there exists a function ς assigning a state to the empty set of moves and to each finite initial segment of moves. The function ς has to satisfy the *coherence condition* requiring that if x is a maximal element in a finite initial segment X of the set of moves and $Y = X \setminus \{x\}$, then x is a move of the agent it is assigned to, and the state $\varsigma(X)$ is obtained from the state $\varsigma(Y)$ by performing x at $\varsigma(Y)$.

The coherence condition asserts that actions in a partially ordered run that do not affect one another may be ordered in any possible way, while those that do (e.g. due to performing updates to the same location or one performing an update dependent on a condition that is changed by another) have to be totally ordered with respect to one another. In particular, the coherence condition implies that the view on the global memory in each of the actions in a partially ordered run is sequentially consistent.

Each thread in a multithreaded Java program is an ordinary interactive small-step algorithm represented by an ordinary interactive small-step ASM. Although not covered by the original definition of a DASM, we consider it plausible to use such interactive ASMs for representing DASM's agents. The exposition in [4] explaining relationship between query templates and external function symbols might serve as a preliminary justification. A more elaborate treatment of this issue will, however, have to be provided elsewhere.

In this way, each multithreaded Java program gives rise to a distributed ordinary interactive small-step ASM. Since by the requirements of the Java memory model we do not have a sequentially consistent global memory any more, the original notion of a partially ordered run of a DASM from [10] does not suffice. As explained earlier, in order to capture the shared memory behavior, the communication between agents is modeled by query-reply pairs, thus the environment providing replies to queries becomes an essential part of the run.

The definition of a run of a distributed ordinary interactive small-step ASM (henceforth called a distributed run) that we propose is the definition of a run of a multithreaded Java program as stated in Definition 1. It is not difficult to rephrase Definition 1 by using the terminology of [10], and one would come out with a definition that requires all but the last of the conditions in a definition of a partially ordered run of a DASM from [10] to be satisfied. The coherence condition, on the other hand, has to be expressed as a consistency requirement on a justification of an environment of a run.

Viewed from the perspective that the coherence condition reflects sequential consistency of the shared memory, this additional requirement is indeed a restriction of the environment. In this particular case, a justification of an environment of a run that is consistent with a coherence condition would be defined as a justification that does not introduce any new components in addition to those of a run, and satisfies the consistency requirement placed on the run. Therefore, by accepting this way of phrasing, our distributed run generalizes the original notion of a partially ordered run of a DASM from [10].

5 Conclusions and Future Work

Two main issues that we have dealt with in this paper are a mathematically precise specification of the Java memory model and analysis of its implications when multithreaded Java programs are put into the ASM context.

The crucial problem with the JMM specification is that its declarative formulation is hard to check effectively. Namely, we only specify conditions that a JMM-consistent environment of a run of a Java program needs to satisfy, however, we do not provide any practical algorithm for checking them. It is the subject of our future work to try to simplify the original definition as much as possible and to deal with its algorithmic applications.

The interpretation of multithreaded Java programs as distributed ordinary interactive small-step ASMs may be considered fairly straightforward. Nevertheless, the potential behavior of such ASMs is not captured by the theory of distributed algorithms developed so far, and thus, in our opinion, deserves further attention. We hope that the notions of an environment and a run introduced in this paper will give a helpful insight into considerations that will possibly have to be taken into account in some more general theory of distributed algorithms that yet has to be developed.

References

1. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

2. Sarita V. Adve, Vijay S. Pai, and Parthasarathy Ranganathan. Recent advances in memory consistency models for hardware shared-memory systems. *Proceedings of the IEEE, special issue on distributed shared-memory*, 87(3):445–455, March 1999.
3. Varsha Awhad and Charles Wallace. A unified formal specification and analysis of the new java memory models. In *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 166–185. Springer, 2003.
4. Andreas Blass and Yuri Gurevich. Ordinary interactive small-step algorithms, II. To appear in *ACM Transactions on Computational Logic*.
5. Andreas Blass and Yuri Gurevich. Ordinary interactive small-step algorithms, III. To appear in *ACM Transactions on Computational Logic*.
6. Andreas Blass and Yuri Gurevich. Ordinary interactive small-step algorithms, I. *ACM Transactions on Computational Logic*, 7(2):363–419, 2006.
7. E. Börger and W. Schulte. Programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
8. Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, 2007.
9. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
10. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
11. Y. Gurevich, W. Schulte, and C. Wallace. Investigating Java concurrency using Abstract State Machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 151–176. Springer-Verlag, 2000.
12. Jeremy Manson. *The Java memory model*. PhD thesis, University of Maryland, College Park, 2004.
13. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model (expanded version). Submitted to ACM Transactions on Programming Languages and Systems.
14. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 378–391. ACM Press, 2005.
15. Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'07)*, pages 161–172. ACM Press, 2007.
16. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.