

An ASM-based Approach to Modeling Memory Models

Matko Botinčan, Paola Glavan, Davor Runje

(University of Zagreb, Croatia)

matko.botincan@math.hr, paola.glavan@gmail.com, davor.runje@fsb.hr)

Abstract: The main goal of this paper is to provide a general framework for modeling memory models of multithreaded Java-like programming languages and discuss its interpretation in the ASM context. The framework is organized in such way that we can clearly stipulate how the memory model conditions the behavior of the environment. It is shown how each thread in a multithreaded Java-like program can be seen as an ordinary interactive small-step algorithm, and, consequently, how such Java-like program gives rise to a distributed ordinary interactive small-step ASM. We also propose a general definition of a particular kind of distributed ordinary interactive small-step ASMs and the a notion of a distributed run. Depending on the conditions on the environment imposed by the memory model, runs of such distributed ASMs may exhibit behavior that is impossible to be observed under commonly assumed sequentially consistent settings.

Key Words: Java memory model, distributed algorithms, abstract state machines

Category: F.1.2, D.1.3

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 [Adve and Gharachorloo, 1996, Adve et al., 1999] 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 [Gosling et al., 2005] and recent papers on the Java memory model (JMM) [Manson et al., 2005b, Manson et al., 2005a] aim to give a precise specification of the behavior of the shared memory for multithreaded Java programs. The JMM has been designed to provide guarantees not only to programmers by ensuring sequentially consistent behavior of correctly synchronized (data race free) programs and promising that even in the presence of data races the values should not appear out of thin air, but also to compiler writers by allowing common compiler optimization techniques as long as they do not violate the previous guarantees. Nevertheless, the JMM still lacks a rigorous mathematical treatment, and the precise formalization of the JMM (as well as memory models in general) attracts attention as a topic of growing interest [Aspinall and Ševčík, 2007, Cenciarelli et al., 2007, Saraswat et al., 2007].

The main goal of this paper is to provide a general framework for modeling memory models of multithreaded Java-like programming languages. The framework is based on partial orders capturing intrinsic components of program's execution, while the interaction with the memory is seen as an interaction with the environment. Since the JMM is one of the first memory models connecting a high-level multithreaded language to such low-level concepts of execution, we tailor our approach so that it suits the formalization of the JMM. Nevertheless, we expect this approach to be suitable for a wider range of Java-like programming languages (e.g. in many aspects JMM serve as a basis for the new multithreaded C++ memory model [CPP, 2007]).

Therefore, we closely follow most of the notation used in the work on the Java memory model [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a] and lay down the definitions in a mathematically precise way aiming to eliminate potential ambiguities present in the source. The original specification of the JMM is refactored in order to clearly separate the notion of a run of a multithreaded Java-like 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 against a given environment of a run of a Java program. Also, we have to note that the JMM specification in this paper does not include all items that are included in [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a]. Namely, we only consider actions that actually interact with the shared memory, thus we omit other actions, in particular, observable actions described in [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a] (however, it would be relatively straightforward to include

them). We also leave out semantics of final fields.

In the final part of the paper we argue how each thread of a multithreaded Java-like program can be seen as an ordinary interactive small-step algorithm [Blass and Gurevich, 2007a, Blass and Gurevich, 2007b, Blass and Gurevich, 2006], and, consequently, how a multithreaded Java-like program gives rise to a particular kind of distributed ordinary interactive small-step ASM [Gurevich, 1995]. To the best of our knowledge, present ASM papers dealing with distributed algorithms assume that the communication between agents is accomplished in a sequentially consistent manner. Depending on the conditions on the environment imposed by the memory model, runs of a distributed ASM corresponding to a multithreaded Java-like 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 general framework for modeling memory models of multithreaded Java-like programming languages in a way so that we can clearly stipulate how the memory model conditions the behavior of the environment.
- By giving multithreaded Java-like programs an interpretation in the ASM context, we arrive to the notion of a particular kind of distributed ordinary interactive small-step ASMs and the corresponding notion of a distributed run that reveals counterintuitive behavior not attributed to distributed algorithms in the ASM literature so far.

The rest of the paper is organized as follows. Section 2 deals with basic notions needed in our framework for modeling memory models. The taxonomy of several different memory models formalized within this framework is the topic of Section 3. In Section 4, we discuss relevant related work from the ASM field, analyze multithreaded Java-like programs in the ASM context and deal with the new notion of a distributed run of a distributed ordinary interactive small-step ASM. The final Section 5 gives concluding remarks.

2 Basic Notions

This section gives the basic definitions needed for formal treatment of multithreaded Java-like programs and deals with the notion of a run of such programs. Later on, it introduces the notion of an environment of such run and its justification.

2.1 Preliminaries

Let $R \subseteq X \times X$ be a binary relation on a set X . Denote with $R^r := \{(y, x) \in X \times X \mid xRy\}$ 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 $R \subseteq R^r$, R is antisymmetric if $R \cap R^r \subseteq I$ 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 $R \cup R^r = X \times X$; R is a strict total order if it is a strict partial order such that $R \cup R^r = (X \times X) \setminus I$.

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_{α} .

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^r -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-like program that spawns a set of threads **Threads**. Each thread $t \in \mathbf{Threads}$ is assigned a program text $\Pi(t) \in \mathcal{P}$ 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 **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 `Threads`.

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

2.3 Threads

Each thread $t \in \text{Threads}$ 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 the particular programming language the program \mathcal{P} has been written in 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 `Queries` to the shared memory. The set `Queries` is the smallest set containing queries $\langle \text{read}, l \rangle$, $\langle \text{write}, l, v \rangle$, $\langle \text{volatile read}, l \rangle$, $\langle \text{volatile write}, l, v \rangle$, $\langle \text{lock}, m \rangle$, $\langle \text{unlock}, m \rangle$, $\langle \text{create thread} \rangle$, $\langle \text{join}, t \rangle$, for all $l \in \text{Locations}$, $v \in \text{Values}$, $m \in \text{Monitors}$ and $t \in \text{Threads}$. We write $\vdash_X^t q$ 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 $\langle \text{read}, l \rangle$ and $\langle \text{volatile read}, l \rangle$ are *reading* l , queries $\langle \text{write}, l, v \rangle$ and $\langle \text{volatile write}, l, v \rangle$ are *writing* v to l , the query $\langle \text{lock}, m \rangle$ is *locking* m , the query $\langle \text{unlock}, m \rangle$ *unlocking* m , the query $\langle \text{create thread} \rangle$ is *creating a thread*,¹ and the query $\langle \text{join}, t \rangle$ is *joining a thread* t .

¹ Formally, the query $\langle \text{create thread} \rangle$ 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 $v \in \mathbf{Values}$, each query creating a thread is replied with a $t \in \mathbf{Threads}$, and a query of any other kind is replied with an automatic OK. Let **Replies** stand for the union $\mathbf{Values} \cup \{\text{OK}\}$.

Possible runs of a thread t are given by a labeled transition system \mathbf{TS}_t with a set of states S_t and transition relations $\xrightarrow[t]{(q,r)} \subseteq S_t \times S_t$, where $q \in \mathbf{Queries}$ and $r \in \mathbf{Replies}$. 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 $\vdash_X^t q$ 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 \mathbf{TS}_t a (q, r) -labeled transition $X \xrightarrow[t]{(q,r)} X'$, for each possible reply $r \in \mathbf{Replies}$ and the corresponding successor state X' .

A run segment of a thread t is an alternating sequence $r = X_0 \alpha_1 X_1 \dots \alpha_n X_n$ of states and transition labels in \mathbf{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-like 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 $(\mathcal{A}, \mathcal{H}, \mathcal{T}, \sigma, \mathcal{E}, \xrightarrow{po})$ where:

- \mathcal{A} is a set of *actions*;
- \mathcal{T} assigns a thread $\mathcal{T}(a) \in \mathbf{Threads}$ to every action $a \in \mathcal{A}$;
- 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 $\vdash_{\sigma(a)}^{\mathcal{T}(a)} q$ and r is a reply with a value in **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 $t \in \text{Threads}$ 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 [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a]. 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

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 [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a].

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.

2.5 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:

- The write-seen function 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, $<_{po} \upharpoonright_{Dom(<_{so})} \subseteq <_{so}$;
 - 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 .

As we shall see later on, dealing with memory models will reduce to putting explicit or implicit constraints on the justification for an environment of a run. Its two components seems to provide enough information that is necessary for modeling common memory models encountered in practice. If, however, one would need to deal with more peculiar memory models requiring additional information for justification than those provided by the write-seen function and the synchronization order, one would just need to incorporate the additionally required components into the previous definition.

3 Memory Models

In this section, we give a framework for modeling memory models that is based on previously introduced concepts. In particular, we show how different constraints on justifications give rise to different kinds of memory models. However, before we dwell into the taxonomy of memory models, let us first introduce two auxiliary relations. Given a justification $(\Theta, \xrightarrow{so})$ for an environment $(\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ of a run, these two relations are defined as follows:

- The *synchronizes-with order* \xrightarrow{swo} is a restriction of $<_{so}$ that relates those pairs of synchronizing actions with release-acquire semantics. 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* $<_{hbo}$ is a strict partial order induced by the synchronizes-with order and the program order:

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

Therefore, the happens-before order establishes what is necessarily known about the order in which two actions have been executed. The actions which are not related by the happens-before order could have been executed in any possible relative order and we cannot presuppose any particular ordering between them.

3.1 Happens-before consistency

The write-seen function as given in the definition of the justification for an environment of a run is not a priori related to the happens-before order in any way. However, in order to make the write-seen function plausible at all, we have to require some minimum consistency constraints on it. In particular, the write-seen function should not assign to a read a write for which it is known that it has happened strictly after the read. Likewise, it should also respect the order between volatile reads and writes introduced with the synchronization order. This minimal set of requirements on a justification for an environment of a run is captured by the notion of happens-before³ 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 The name *happens-before* draws its roots from Lamport's paper [Lamport, 1978] dealing with total ordering a partially ordered set of events in a distributed system. This paper seems to be the first publication formally treating the notion of a relaxed memory model.

3.2 Sequential consistency

The most common consistency requirement on a memory that from the programmer’s perspective is also the easiest to deal with is the sequential consistency. It was first defined in [Lamport, 1979] as a property requiring that “. . . the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”. By transferring these requirements to our terminology we see that the sequential consistency is just a special case of the happens-before consistency in which all actions have to happen in a total order (namely, the execution order).⁴

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 Memory coherence

In the world of hardware memory models, a memory system is called coherent if for every possible execution and for any memory location, it is possible to construct a total order of all memory actions dealing with the location such that the memory operations of each thread occurs in the program order and the value returned by the read (load) is the value of the latest write (store) to the location in the total order [Collier, 1992, Dubois et al., 1988]. Since even in the most relaxed settings, the memory coherence requires properties that are captured by the notion of the happens-before consistency, transferring the notion of the memory coherence to the justification of an environment of a run gives rise to the following definition.

Definition 6. A justification $(\Theta, \xrightarrow{so})$ for an environment $\eta = (\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ of a run is *memory coherent* if:

- η is happens-before consistent; and

⁴ A careful reader might notice that the given notion of sequential consistency assumes atomicity, i.e. linearizability [Herlihy and Wing, 1987] of actions. Although this requirement is plausible for our purposes, in more general settings one would need to distinguish between the order of issuing write actions and the order that captures when they take place.

- for each location l , there exists a total order $<_{eo}^l$ on \mathcal{A} consistent with $<_{hbo}$ such that for each a such that a is reading l , $\Theta(a)$ is the $<_{eo}^l$ -maximal action in $\Xi(a)$, i.e. $\Theta(a) = \max_{<_{eo}^l}(\Xi(a))$.

Now it is easy to formally state the fact known in the hardware memory models world for a long time, namely, that the sequential consistency implies memory coherence.

Theorem 7. *If a justification $(\Theta, \xrightarrow{so})$ for an environment $\eta = (\mathcal{A}, \Pi, \mathcal{T}, \mathcal{E}, \xrightarrow{po})$ of a run is sequentially consistent, then it is memory coherent.*

3.4 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 [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a].

Requirements imposed by the Java memory model 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 [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a] 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 [Manson et al., 2005b].

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* step-wise 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 8. 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:⁵

- each action in \mathcal{C}_i corresponds to an action in \mathcal{A}_i , i.e. $\mathcal{C}_i \subseteq \mathcal{A}_i$;
- 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 $\Theta(a_r) \in \mathcal{C}_{i-1}$.

3.5 Definition of memory models

Now we can transfer the notions of consistency of justifications of an environment of a run to consistency of environments of a run in a natural way.

Definition 9. 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 of memory models.

⁵ Unlike in [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a], 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).

- Definition 10.** – Sequentially consistent memory model is the set $\{\eta \mid \eta \text{ is a sequentially consistent environment for some run}\}$;
- Happens-before memory model is the set $\{\eta \mid \eta \text{ is a happens-before consistent environment for some run}\}$
 - Java memory model is the set $\{\eta \mid \eta \text{ is a JMM consistent environment for some run}\}$.

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.

3.6 Examples

In order to clarify the difference between different memory models and to raise some intuition about them, we provide two simple examples.

Example 1.

$$\begin{aligned}
 I &: x == y == 0; \\
 (r1 := x; y := r1; \mid r2 := y; x := r2;) \\
 B &: r1 == r2 == 1
 \end{aligned}$$

i.e. locations x, y are initially set to 0 and two concurrent threads operate on shared memory locations x, y using thread local registers $r1, r2$. The question we would like to answer is whether the behavior $B : r1 == r2 == 1$ is allowed in different memory models.

The happens-before memory model allows only those read-write pairs $r, \theta(r)$ on the same location for which r does not happen-before $\theta(r)$, and there exists no other write action w between $\theta(r)$ and r . In this example, we can take the following values for θ : $\theta(\langle \mathbf{read}, x \rangle) = \langle \mathbf{write}, x, r2 \rangle$ and $\theta(\langle \mathbf{read}, y \rangle) = \langle \mathbf{write}, y, r1 \rangle$. Since there is no restriction on the values read from $r1$ and $r2$, if we take $\langle \mathbf{read}, r1 \rangle = 1$ and $\langle \mathbf{read}, r2 \rangle = 1$, we obtain the behavior $B : r1 == r2 == 1$ in question.

The sequentially consistent memory model requires the existence of a total order consistent with the happens-before order such that each read sees the most recent write. We cannot add this requirement to the previously chosen write-seen function since $\langle \mathbf{write}, x, r2 \rangle <_{hbo} \langle \mathbf{write}, y, r1 \rangle$ and $\langle \mathbf{write}, y, r1 \rangle <_{hbo} \langle \mathbf{write}, x, r2 \rangle$. Thus, the only possible sequentially consistent behavior is $r1 == r2 == 0$.

In Java memory model, one builds iteratively the set of committed actions, such that in each step committed actions respect the happens-before order and

the causality. In this example, the only possible initial actions that can be committed are the two initial writes $x := 0$ and $y := 0$ – they constitute the set \mathcal{C}_1 . Since all reads have to see writes that happen-before them, we have to commit actions $r1 := x$ and $r2 := y$ before the corresponding reads of $r1$ and $r2$ take place. We can take the set \mathcal{C}_2 to include both writes $r1 := x$ and $r2 := y$. At the end, the actions $y := r1$ and $x := r2$ get committed in \mathcal{C}_3 . This sequence of actions justifies the behavior $r1 == r2 == 0$. It also illustrates how the Java memory model prevents from the out-of-thin-air behavior $B : r1 == r2 == 1$ that is allowed by the happens-before memory model.

Example 2. This example differs from Example 1 in that a write to y uses a constant value that does not need to be read beforehand.

$$\begin{aligned} I : & x == y == 0; \\ (r1 := x; y := 1; | r2 := y; x := r2;) \\ B : & r1 == r2 == 1 \end{aligned}$$

The happens-before memory model allows the behavior $B : r1 == r2 == 1$ by a similar argumentation as in Example 1, only now the read of y necessarily sees the value 1 written by $y := 1$. On the other hand, the sequentially consistent memory model still disallows the behavior in question, yet, it now allows the following two behaviors: $r1 == r2 == 0$ and $r1 == 0, r2 == 1$. For the case of the Java memory model, the behavior $B : r1 == r2 == 1$ can be justified by the following sequence of committed actions: C_1 containing $x := 0, y := 0$ and $y := 1$; C_2 containing $r1 := x$ and $r2 := y$; and C_3 containing $x := r2$.

4 An ASM View on the Memory Models

In order to make the presented taxonomy of memory models as intuitive as possible, and, in particular, to keep the Java memory model close to its original specification [Manson, 2004, Manson et al., 2005b, Manson et al., 2005a], we have tried to minimize explicit references to ASMs in our exposition so far. Nevertheless, all memory models that have been defined *are* ASM models. We aim to make this claim precise in the rest of this section.

The main contributions of the paper with respect to the ASMs can be summarized as follows:

- This is one of the first practical applications of the recently developed theory of interactive algorithms [Blass and Gurevich, 2007a, Blass and Gurevich, 2007b, Blass and Gurevich, 2006].
- We have extended the notion of distributed computation from the Lipari guide [Gurevich, 1995] in two directions:

1. The notion of a distributed run is defined with ordinary interactive small-step algorithms acting as agents. Such extension is natural and not particularly 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 have further generalized the notion of a distributed run by replacing the coherence condition by more general requirements on environment’s behavior such that different requirements correspond 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. For typical case studies so far see e.g. [Börger et al., 1995] or [Gurevich and Huggins, 1996].

Since our main focus when dealing with memory models was on the Java memory model, we briefly remind the reader of previously written publications that relate Java and ASM.

Börger and Schulte in [Börger and Schulte, 1998b, Börger and Schulte, 1998a], and Stärk, Schmid and Börger in [Stärk et al., 2001] define precise semantical description of the Java language using ASMs. They also verify that compilation of the Java language to the JVM bytecode 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 the current Java memory model.

Gurevich, Schulte and Wallace in [Gurevich et al., 2000] present the concurrent features of Java using imperative, ASM approach. They follow the earlier version of the Java language specification manual [Gosling et al., 2005]. 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 [Awhad and Wallace, 2003] formalize using ASMs the two memory models that have been proposed as replacements for the previously erroneous Java memory model. 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 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 [Blass and Gurevich, 2006]: 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 [Blass and Gurevich, 2006, Blass and Gurevich, 2007a, Blass and Gurevich, 2007b], i.e. they

1. proceed in discrete global steps;
 2. perform 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 program text $\Pi(t)$. 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 [Börger and Schulte, 1998b, Gurevich et al., 2000, Stärk et al., 2001].
- The finite set of labels A used for queries is:

$$A = \{ \text{read, write, volatile read, volatile write, lock, unlock,} \\ \text{create thread, join} \}.$$

- The causality relation $\emptyset \vdash_{\sigma(a)}^{\mathcal{T}(a)} q$ 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 a 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 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 the definitions 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 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 $\sigma(a)$: $\emptyset \vdash_{\sigma(a)}^{\mathcal{T}(a)} q$, and r is a reply with a value in 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 [Blass and Gurevich, 2006]: if $a \xrightarrow{po} a'$, $\emptyset \vdash_{\sigma(a)}^{\mathcal{T}(a)} q$, 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 [Gurevich, 1995], we do not have a global memory, and local states of threads (namely, ASM agents) are not local views of the global memory - i.e. they are not $\text{View}_a(S)$ reducts of a global state S to local agents’ vocabulary. Also, we implicitly assume that a program \mathcal{P} is a finite collection of finite program texts, and that there are only finitely many threads in each execution of \mathcal{P} .

4.2 Multithreaded Java-like 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 [Gurevich, 1995] for a particular type of ASMs called the *distributed ASMs* (DASMs). DASM, as defined in [Gurevich, 1995], 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 [Gurevich, 1995] (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.

As argued in the previous section, each thread in a multithreaded Java-like program is an ordinary interactive small-step algorithm represented by an ordinary interactive small-step ASM that issues exactly one query in each of its states. Although not being covered by the original definition of a DASM, we consider it plausible to use such interactive ASMs for representing DASM's agents. In this way, each multithreaded Java-like program gives rise to such distributed single-query ordinary interactive small-step ASM. Therefore, we propose the following definition for this case of distributed ordinary interactive small-step ASMs.

Definition 11. A distributed single-query ordinary interactive small-step ASM \mathcal{D} is a tuple $(\mathcal{P}, \text{Agents}, \Pi, (S_t)_{t \in \text{Agents}}, (I_t)_{t \in \text{Agents}}, (\frac{(q,r)}{t} \rightarrow)_{t \in \text{Agents}}^{q \in \text{Queries}, r \in \text{Replies}})$ where:

- \mathcal{P} is a collection of (ordinary) ASM program texts using **Queries** as a finite set of labels for queries;
- **Agents** is a set of agents;
- $\Pi: \text{Agents} \rightarrow \mathcal{P}$ is a function that assigns each agent a program text in \mathcal{P} ;
- $(S_t)_{t \in \text{Agents}}$ is a disjoint collection of states S_t , for each agent $t \in \text{Agents}$, such that in each state the agent issues exactly one query;
- $(I_t)_{t \in \text{Agents}}$ is a disjoint collection of initial states $I_t \subseteq S_t$, for each agent $t \in \text{Agents}$;
- $(\frac{(q,r)}{t} \rightarrow)_{t \in \text{Agents}}$ is a collection of transition relations $\frac{(q,r)}{t} \rightarrow \subseteq S_t \times S_t$, where $q \in \text{Queries}$ and $r \in \text{Replies}$, representing the semantics of each agent t and the causality relation in a way that has been explained earlier in this section.

The original notion of a partially ordered run of a DASM from the Lipari guide [Gurevich, 1995], however, now does not suffice and the environment providing replies to queries will have to become 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 a rephraseal of definition of a run of a multithreaded Java-like program (Definition 1) in the style from [Gurevich, 1995].

Definition 12. A distributed run of a distributed single-query ordinary interactive small-step ASM \mathcal{D} is a tuple $(\mathcal{A}, \xrightarrow{po}, \mathcal{T}, \sigma, \mathcal{E})$ where:

- \mathcal{A} is a set of *actions*;
- \xrightarrow{po} is an irreflexive partial order on \mathcal{A} satisfying the finiteness property (we denote by $<_{po} = (\xrightarrow{po})^+$ the corresponding strict partial order);
- \mathcal{T} assigns an agent $\mathcal{T}(a) \in \mathbf{Agents}$ to every action $a \in \mathcal{A}$ such that for each agent $t \in \mathbf{Agents}$, the set $(\{a \in \mathcal{A} \mid \mathcal{T}(a) = t\})$ is totally ordered by $<_{po}$;
- $\mathcal{E}: \mathcal{A} \rightarrow \mathbf{Queries} \times \mathbf{Replies}$ assigns to each action $a \in \mathcal{A}$ a query-reply pair (q, r) such that $\vdash_{\sigma(a)}^{\mathcal{T}(a)} q$ and r is a reply to q ;
- $\sigma: \mathcal{A} \rightarrow \bigsqcup_{t \in \mathbf{Agents}} S_t$ assigns to each action $a \in \mathcal{A}$ a state such that for all $a_1, a_2 \in \mathcal{A}$ with $\mathcal{T}(a_1) = \mathcal{T}(a_2)$ and $\mathcal{E}(a_i) = (q_i, r_i)$ ($i = 1, 2$) we have $(a_1 \xrightarrow{po} a_2)$ iff $\sigma(a_1) \xrightarrow[t]{(q_1, r_1)} \sigma(a_2)$ and there exists $X \in S_t$ such that $\sigma(a_2) \xrightarrow[t]{(q_2, r_2)} X$.

Note that in this definition of a distributed run we have excluded the coherence condition present in the definition of a partially ordered run of a DASM from [Gurevich, 1995]. Namely, the coherence condition puts additional restriction on σ that would imply its sequentially consistent behavior. Although on the level of intra-agent semantics this kind of behavior might be desirable, it would also impose an implicit restriction on the answer function provided by the environment. Since here we are only interested in what is happening on the inter-agent level and do not want to a priori restrict the behavior of the answer function, we have excluded the coherence condition from Definition 12 and decided to express all additional requirements as constraints on a justification of an environment of a distributed run (the notions of a justification and an environment of a distributed run are defined exactly the same as in Definitions 2 and 3, respectively).

Now, in order to express the coherence condition as a consistency requirement on a justification of an environment of a run, we just need to take an appropriate set of actions that would include information about agents' states in each of its actions. 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 from a run, while the run satisfies the direct syntactic translation of the coherence condition from [Gurevich, 1995]. Therefore, by accepting this way of phrasing, we can say that the distributed run

from Definition 12 indeed generalizes the original notion of a partially ordered run of a DASM from [Gurevich, 1995].

5 Conclusions and Future Work

Two main issues that we have dealt with in this paper are a general framework for modeling memory models of multithreaded Java-like programming languages and an analysis of its implications when multithreaded Java-like programs are put into the ASM context. We have proposed a definition of a particular kind of distributed ordinary interactive small-step ASMs (namely, the ones that issue exactly one query in each state) and a new notion of a distributed run.

There exists a vast amount of related research dealing with modeling concurrent systems in sequentially consistent settings (e.g. consider TLA, various process algebras such as CSP or CCS, etc). However, since for concurrent processes they typically employ interleaving semantics, it is hard, if not impossible, to reason about more relaxed memory models inside of them. On the other hand, the approach we have followed in this paper allowed us to explicitly separate the interaction with the shared memory from the rest of the system and then reason about it in full generality.

When dealing with memory models our main focus was on the JMM. The crucial problem with the JMM specification is that it is not at all clear how to check its declarative formulation effectively. Namely, it only specifies the conditions that a JMM-consistent environment of a run of a Java-like program needs to satisfy, however, it does not give rise to an algorithm for checking them. The decidability of the general case has not yet been determined, however, even the most simple case of verifying JMM requirements for programs with at most two threads, at most one shared variable and involving no synchronization actions is known to be NP-complete [Polyakov and Schuster, 2006]. It is the subject of our future work to investigate algorithmic properties and computational complexity of verifying JMM requirements for other classes of programs.

Finally, the interpretation of multithreaded Java programs as distributed ordinary interactive small-step ASMs may be considered fairly straightforward. Nevertheless, the 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 proposed notions of a distributed single-query ordinary interactive small-step ASM, a distributed run, as well as an environment of a run and its justification will give a helpful insight into what perhaps should be taken into account in a further development of this theory. In particular, we would like to investigate generalizations of proposed notions when (ordinary) interactive small-step ASMs are put into the place of agents of a distributed ASM.

References

- [CPP, 2007] (2007). Threads and memory model for C++. <http://www.hpl.hp.com/personal/Hans.Boehm/c++mm/>.
- [Adve and Gharachorloo, 1996] Adve, S. V. and Gharachorloo, K. (1996). Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76.
- [Adve et al., 1999] Adve, S. V., Pai, V. S., and Ranganathan, P. (1999). 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.
- [Aspinall and Ševčík, 2007] Aspinall, D. and Ševčík, J. (2007). Formalising java’s data race free guarantee. In *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007), Kaiserslautern, Germany, September 10-13, 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 22–37. Springer.
- [Awhad and Wallace, 2003] Awhad, V. and Wallace, C. (2003). 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.
- [Blass and Gurevich, 2006] Blass, A. and Gurevich, Y. (2006). Ordinary interactive small-step algorithms, I. *ACM Transactions on Computational Logic*, 7(2):363–419.
- [Blass and Gurevich, 2007a] Blass, A. and Gurevich, Y. (2007a). Ordinary interactive small-step algorithms, II. *ACM Transactions on Computational Logic*, 8(3).
- [Blass and Gurevich, 2007b] Blass, A. and Gurevich, Y. (2007b). Ordinary interactive small-step algorithms, III. *ACM Transactions on Computational Logic*, 8(3).
- [Börger et al., 1995] Börger, E., Gurevich, Y., and Rosenzweig, D. (1995). The bakery algorithm: Yet another specification and verification. In Börger, E., editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press.
- [Börger and Schulte, 1998a] Börger, E. and Schulte, W. (1998a). Defining the Java Virtual Machine as platform for provably correct Java compilation. In Brim, L., Gruska, J., and Zlatuska, J., editors, *23rd International Symposium on Mathematical Foundations of Computer Science (MFCS’98), Brno, Czech Republic*, volume 1450 of *Lecture Notes in Computer Science*, pages 17–35. Springer-Verlag.
- [Börger and Schulte, 1998b] Börger, E. and Schulte, W. (1998b). Programmer friendly modular definition of the semantics of Java. In Alves-Foss, J., editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer-Verlag.
- [Cenciarelli et al., 2007] Cenciarelli, P., Knapp, A., and Sibilio, E. (2007). The Java memory model: Operationally, denotationally, axiomatically. In *Proceedings of the 16th European Symposium on Programming (ESOP’07)*.
- [Collier, 1992] Collier, W. (1992). *Reasoning About Parallel Architectures*. Prentice-Hall.
- [Dubois et al., 1988] Dubois, M., Scheurich, C., and Briggs, F. A. (1988). Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21.
- [Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification, Third Edition*. Addison-Wesley.
- [Gurevich, 1995] Gurevich, Y. (1995). Evolving algebras 1993: Lipari Guide. In Börger, E., editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press.
- [Gurevich and Huggins, 1996] Gurevich, Y. and Huggins, J. (1996). The railroad crossing problem: An experiment with instantaneous actions and immediate reactions. In *Proc. CSL’95 (Computer Science Logic)*, volume 1092 of *Lecture Notes in Computer Science*, pages 266–290. Springer-Verlag.
- [Gurevich et al., 2000] Gurevich, Y., Schulte, W., and Wallace, C. (2000). Investigating Java concurrency using Abstract State Machines. In Gurevich, Y., Kutter, P., Odersky, M., and Thiele, L., editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 151–176. Springer-Verlag.

- [Herlihy and Wing, 1987] Herlihy, M. P. and Wing, J. M. (1987). Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 13–26, New York, NY, USA. ACM Press.
- [Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.
- [Lamport, 1979] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transaction on Computers*, 28(9):690–691.
- [Manson, 2004] Manson, J. (2004). *The Java memory model*. PhD thesis, University of Maryland, College Park.
- [Manson et al., 2005a] Manson, J., Pugh, W., and Adve, S. V. (2005a). 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.
- [Manson et al., 2005b] Manson, J., Pugh, W., and Adve, S. V. (2005b). The Java memory model (expanded version). Draft journal paper.
- [Polyakov and Schuster, 2006] Polyakov, S. and Schuster, A. (2006). Verification of the java causality requirements. In *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, volume 3875 of *Lecture Notes in Computer Science*, pages 224–246. Springer.
- [Saraswat et al., 2007] Saraswat, V. A., Jagadeesan, R., Michael, M., and von Praun, C. (2007). 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.
- [Stärk et al., 2001] Stärk, R. F., Schmid, J., and Börger, E. (2001). *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag.