The Data Race Free Guarantee of the Java Memory Model

David Johnston dwtj@iastate.edu

A Process's Gobal Memory Address Space

It is an extremely useful abstraction.

But it's just an abstraction.

And nowadays it's arguably an illusion.

In Multithreaded Programs, The Illusion Breaks Down

In reality, data is shuffled between various memory heirarchy components.

Per-core caches can greatly increase performance.

But these caches can have inconsistent "views" of memory.

Hardware Memory Model

A system's memory model is a contract between the system and the programmer.

Informally, a memory model says how writes/stores on one processor are made visible to another processor.

Java Slogan: "Write Once Run Anywhere"

Java has been designed with strong portability guarantees between various hardware platforms.

Java supports multithreaded programming.

Java designers wanted to shield programmers from the subtleties hardware-specific memory models.

So they made the Java Memory Model (JMM). [MPA05][PAL04][MG04][Gos+18]

The Java Memory Model (JMM)

A contract between a Java programmer and a Java platform.

It says which executions of a program are legal.

It gives programmers an (somewhat) intuitive model of concurrent programming.

It constrains which program optimizations platform developers can implement.

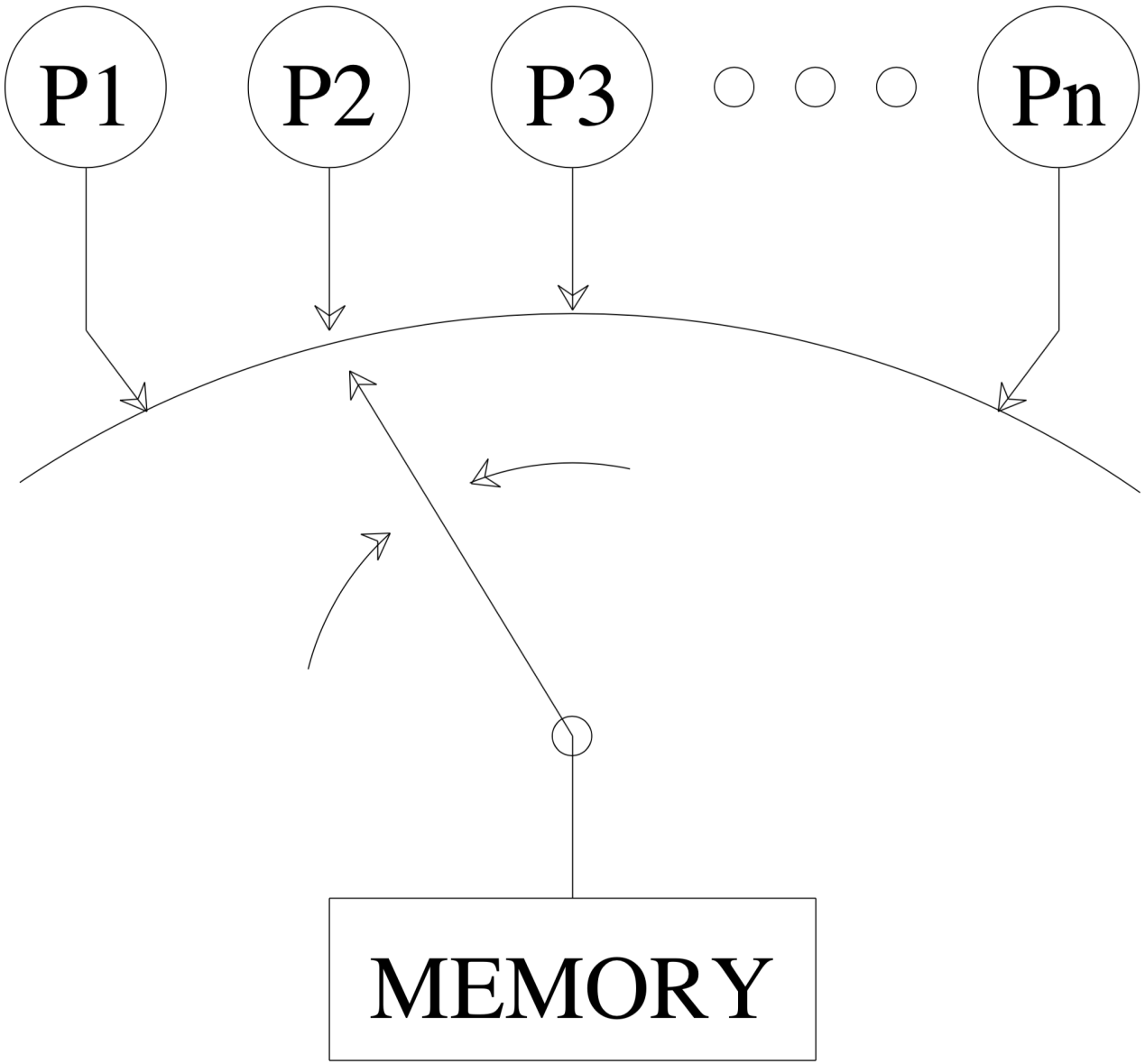
...

Informally:

If a program is data race free (a.k.a. "correctly synchronized"), then the program's execution will be sequentially consistent.

Programmer Intuition of Sequentially Consistent Memory

Taken from [Adv93]



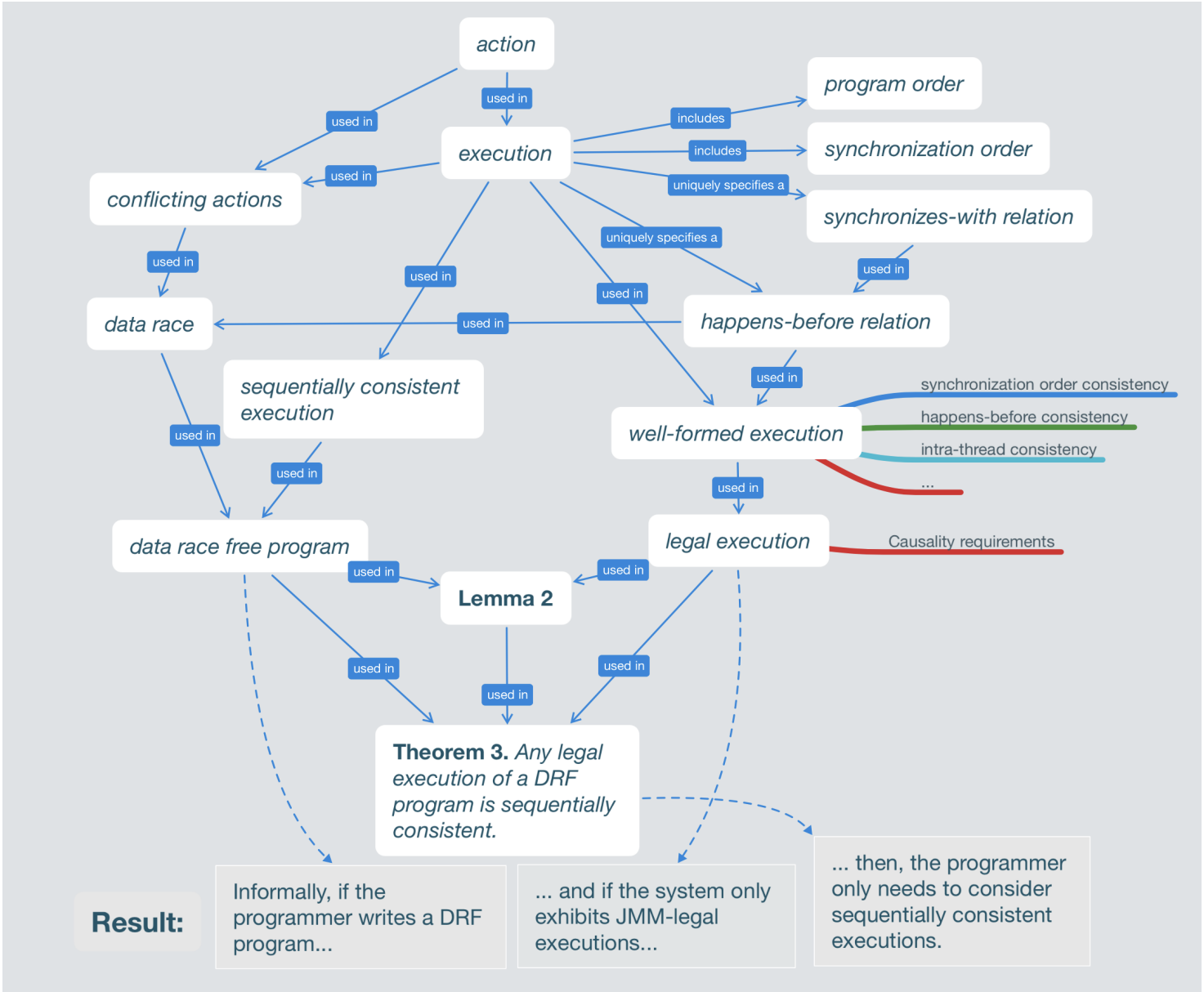
Before we can formally define what we mean by sequential consistency, we need to pin down what we mean by execution.

To precisely state and prove the aforementioned data race free guarantee, we need a bunch of definitions.

We need a roadmap.

JMM Definitions

A Roadmap to the JMM's Data Race Free Guarantee



My Primary Sources

JSR 133, drafted Pugh et al. [PAL04]. Motivation and standards draft.

Java Language Specification, Java SE 11 Edition [Gos+18]. The official standard.

A POPL'05 paper by Manson, Pugh, and Adve [MPA05]: Motivations! Formalisms! Proofs!

A Journal Paper by Aspinall and Ševčík [AŠ07]: Tweaked, fixed, and formalized the DRF guarantee with a proof assistant, Isabelle. We primarily follow this formalism.

Def. Action

A memory related operation belonging to a particular thread; kinds:

- volatile variable[1] read
- volatile variable write
- monitor lock
- monitor unlock
- normal variable read
- normal variable write

[1] The term variable is used throughout this discussion, but variable really means a Java object or a memory location.

Def. Execution

An execution is a 6-tuple $E = (A, P, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V)$, where

A, a set of actions

P, a program

$\overset{po}{\rightarrow}$, a partial order over A called the *program order*.

$\overset{so}{\rightarrow}$, a partial order over A called the *synchronization order*.

W, the write-seen function between actions.

V, the value-written function from actions to values.

Key: The JMM primarily exists to distinguish between legal/illegal executions.

Def. Synchronizes-With Order, $\overset{sw}{\rightarrow}$

Given an execution E, we can use its synchronization order, $\overset{so}{\rightarrow}$, to derive $\overset{sw}{\rightarrow}$

It is a more fine-grained partial order. It relates only those actions which directly synchronize with one another.

Def. Let a and b be actions in E such that $a \overset{so}{\rightarrow} b$.

If a unlocks a monitor which b locks, or

if a is a write and b is a read of volatile variable v,

then $a \overset{sw}{\rightarrow} b$.

Def. Happens-Before Order, $\overset{hb}{\rightarrow}$

We really only use $\overset{sw}{\rightarrow}$ as part of this next (very important) definition.

The happens-before order is also derived from an execution E.

Def. $\overset{hb}{\rightarrow}$ is the transitive closure of the union of $\overset{sw}{\rightarrow}$ and $\overset{po}{\rightarrow}$.

Defining Legal Executions

We can return to the JMM's primary responsibility: distinguishing legal from illegal executions.

There are quite a few conditions required of a legal execution.

These conditions are specified in two definitions:

- Well-formed execution (the "simple stuff")
- Legal execution (the "harder stuff")

As we will see, legality includes well-formedness.

Def. Well-Formed Execution (informal)

The "simple stuff"

We say that an execution $E = (A, P, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V)$ is well-formed if all nine of these conditions hold:

A is finite.

For each thread t, restricting $\overset{po}{\rightarrow}$ to t is a total order.

$\overset{so}{\rightarrow}$ is a total order over synch (i.e. monitor/volatile) actions

$\overset{so}{\rightarrow}$ is consistent with $\overset{po}{\rightarrow}$.

Reads to a particular variable only see writes to that variable.

Proper locking on each monitor.

$\overset{po}{\rightarrow}$ is intra-thread consistent.

$\overset{so}{\rightarrow}$ is synchronization-order consistent.

$\overset{hb}{\rightarrow}$ is happens-before consistent.

Def. Intra-Thread Consistency

For Non-JMM Dynamic Semantics

Intra-thread consistency means that for each thread t, given all of the values that it read, the non-JMM parts of the Java language specification allow for the behavior observed in $\xrightarrow{po}| \ t$.

Def. Synchronization-Order Consistency

For Volatile Reads

For every volatile read r,

$r \overset{so}{\not\rightarrow} W(r)$, and

no write covers $W(r)$ in $\overset{so}{\rightarrow}$, (i.e. $\nexists w : W(r) \overset{so}{\rightarrow} w \overset{so}{\rightarrow} r$).

Def. Happens-Before Consistency

For All Reads

This is similar to synchronization order consistency but w.r.t. all reads and the happens-before order.

For every read r,

$$r \overset{hb}{\not\rightarrow} W(r)$$

$$\nexists w : W(r) \overset{hb}{\rightarrow} w \overset{hb}{\rightarrow} r.$$

Notice that this doesn't say that a read must see a value which preceeds it in the $\overset{hb}{\rightarrow}$. It just says that a read can't see writes covered in $\overset{hb}{\rightarrow}$.

Def. Legal Execution (Informal)

An execution is legal if it is well-formed and if there exists a sequence of increasing sets of *committed actions* with *justifying executions*.

The definition is pretty technical.

Motivation: Well-formed executions can have some very strange behaviors. In particular, "out-of-thin-air" values (i.e. values that have never been written to a variable) can be read.

Def. Sequentially Consistent Execution

An execution is sequentially consistent if there is a exists a total order, $\xrightarrow{t}$, such that:

$\xrightarrow{t}$ is consistent with $\xrightarrow{po}$

$\xrightarrow{t}$ is consistent with $\xrightarrow{so}$

For any read r of variable v, the execution reads the most recent write to v in $\xrightarrow{t}$.

Def. Conflicting Actions

Two actions on the same variable in execution E are conflicting if at least one of them is a write.

Def. Data Race

Two conflicting actions a and b form a *data race* if neither $a \xrightarrow{hb} b$ nor $b \xrightarrow{hb} a$.

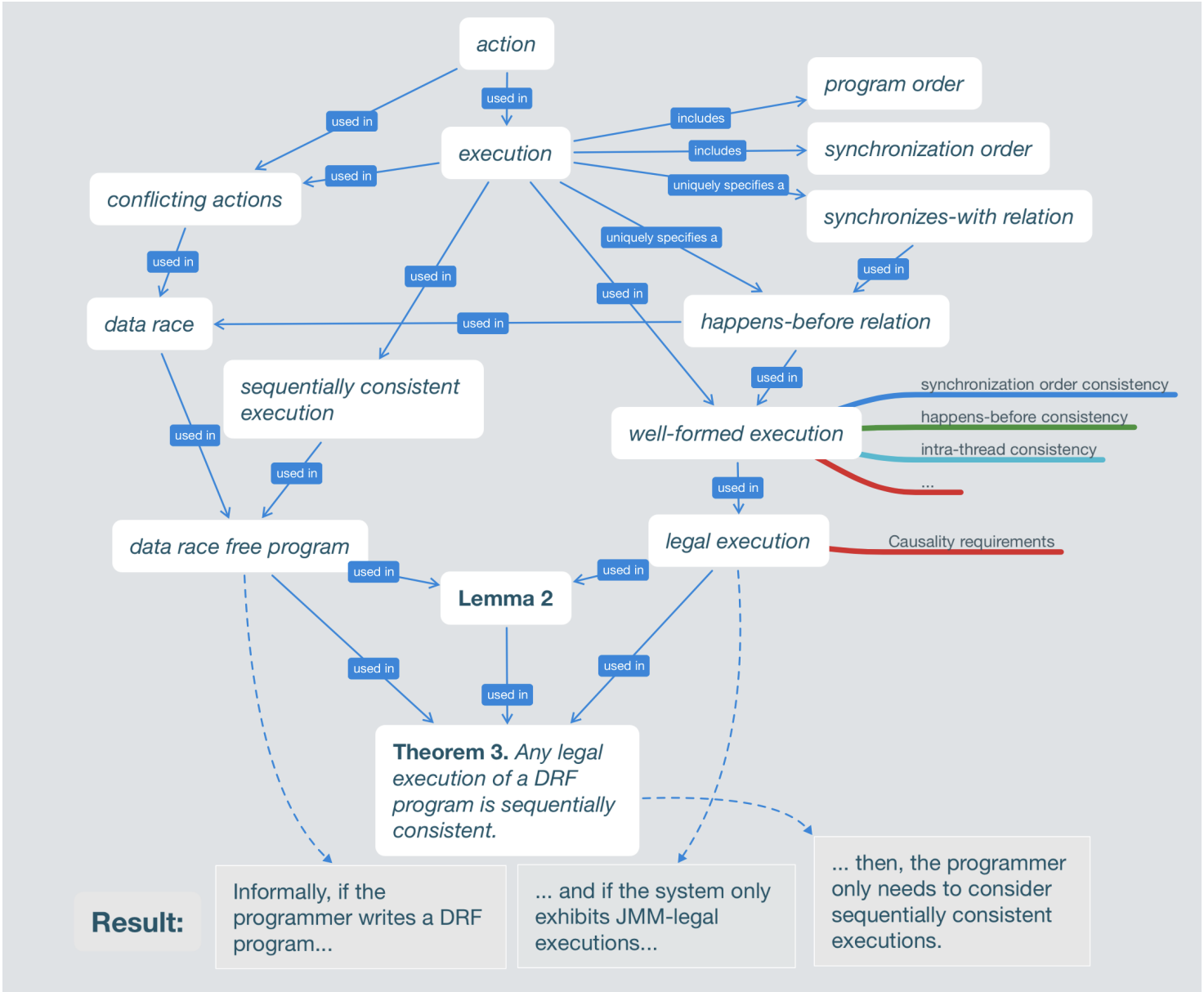
Def. Data Race Free (DRF) Program

Now we finally get to the point.

A program is data-race-free if in all sequentially consistent executions, there are no data races.

JMM Definitions

A Roadmap to the JMM's Data Race Free Guarantee



Theorem 3: JMM's DRF Guarantee

Any legal execution of a data-race-free program is sequentially consistent.

References I

S. V. Adve and K. Gharachorloo. "Shared memory consistency models: a tutorial". In: *Computer* 29.12 (Dec. 1996), pp. 66–76. ISSN: 0018-9162. DOI: 10.1109/2.546611.

S. V. Adve and M. D. Hill. "A unified formalization of four shared-memory models". In: *IEEE Transactions on Parallel and Distributed Systems* 4.6 (June 1993), pp. 613–624. ISSN: 1045-9219. DOI: 10.1109/71.242161.

Sarita Vikram Adve. "Designing memory consistency models for shared-memory multiprocessors". PhD thesis. University of Wisconsin-Madison, Dec. 1, 1993.

References II

David Aspinall and Jaroslav Ševčík. "Formalising Java's Data Race Free Guarantee". In: *Theorem Proving in Higher Order Logics*. Ed. by Klaus Schneider and Jens Brandt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 22–37. ISBN: 978-3-540-74591-4.

Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons, 2004.

Hans-J. Boehm and Sarita V. Adve. "Foundations of the C++ Concurrency Memory Model". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 68–78. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375591. URL: http://doi.acm.org/10.1145/1375581.1375591.

References III

James Gosling et al. *The Java® Language Specification. Java SE 11 Edition*. Oracle, Aug. 21, 2018. URL: https://docs.oracle.com/javase/specs/jls/se11/html/index.html (visited on 12/03/2018).

Jeremy Manson and Brian Goetz. *JSR 133 (Java Memory Model) FAQ*. Feb. 1, 2004. URL: https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html.

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. Long Beach, California, USA: ACM, 2005, pp. 378–391. ISBN: 1-58113-830-X. DOI: 10.1145/1040305.1040336. URL: http://doi.acm.org/10.1145/1040305.1040336.

References IV

William Pugh, Sarita Adve, and Doug Lea. *JSR-133: Java^{TM} Memory Model and Thread Specification*. Research rep. Aug. 24, 2004.