Formalising Java's Data Race Free Guarantee

David Aspinall and Jaroslav Ševčík

LFCS, School of Informatics, University of Edinburgh

Abstract. We formalise the *data race free* (DRF) guarantee provided by Java, as captured by the semi-formal Java Memory Model (JMM) [1] and published in the Java Language Specification [2]. The DRF guarantee says that all programs which are correctly synchronised (i.e., free of data races) can only have sequentially consistent behaviours. Such programs can be understood intuitively by programmers. Formalisation has achieved three aims. First, we made definitions and proofs precise, leading to a better understanding; our analysis found several hidden inconsistencies and missing details. Second, the formalisation lets us explore variations and investigate their impact in the proof with the aim of simplifying the model; we found that not all of the anticipated conditions in the JMM definition were actually necessary for the DRF guarantee. This allows us to suggest a quick fix to a recently discovered serious bug [3] without invalidating the DRF guarantee. Finally, the formal definition provides a basis to test concrete examples, and opens the way for future work on JMM-aware logics for concurrent programs.

1 Introduction

Today, most techniques for reasoning about shared-memory concurrent programs assume an interleaved semantics. But modern hardware architectures and compilers do not guarantee interleaved semantics [4], so there is a gap between languages with shared-memory concurrency and reasoning principles for them. Java aims to bridge this gap via a complex contract, called the *Java Memory Model* (JMM) [1, 2], which holds between the virtual machine and programmers. For programmers, the JMM claims that correctly synchronised (data race free) programs only have *sequentially consistent* behaviours; this is the DRF guarantee. For compiler writers implementing the virtual machine, the JMM constrains permissible code optimisations; most prominently, instruction reordering.

To see why the memory model contract is needed, consider the simple program below, with two threads sharing memory locations `x` and `y`:

initially: `x = y = 0`	
`1: r1 := x`	`3: r2 := y`
`2: y := 1`	`4: x := 1`

The contents of the local registers `r1` and `r2` at the end of the program depends on the order of serialisation of memory accesses. For example, the result `r1` = `0` and `r2` = `0` holds if the order of the statements is 1, 3, 2, 4. On the other

hand, the result `r1` = `r2` = `1` is not possible in any serialisation of the program, and most programmers would not expect such an outcome. But if a compiler or the low-level hardware reorders the independent instructions 1, 2 and 3, 4 in each thread, then the result `r1` = `r2` = `1` *is* possible, e.g., by executing the statements in the sequence 2, 4, 3, 1. This kind of reordering results from simple optimisations, such as write buffering to main memory or common subexpression elimination. To allow these optimisations, we need a *relaxed memory model* that provides weaker guarantees than global sequential consistency, but allows the programmer some kind of control to prevent unintended behaviours.

The Java Memory Model has already been revised after the initial version had serious flaws [5]. The current version is designed to provide:

1. *a promise for programmers*: sequential consistency must be sacrificed to allow optimisations, but it will still hold for data race free programs;
2. *a promise for compilers*: common hardware and software optimisations should be allowed as far as possible without violating the first requirement;
3. *no unintended information leakage*: even for programs with data races, values should not appear "out of thin air".

The first requirement gives us hope for intuitive understanding, and simple program logics, for data race free programs. Most programs should be correctly synchronised, as recommended by [6]. The third requirement makes a security promise for the "racy" programs: even though values may be undetermined, they should have an origin within the computation, and not, e.g., appear from other memory contents which may be sensitive. Manson et al [1] published proofs of two theorems. The first theorem, stating the promise for compilers, was recently falsified — Cenciarelli et al [3] found a counterexample. The second theorem, the DRF guarantee, is the topic of this paper.

Contribution. We describe a formalisation of definitions of the JMM and a proof of the DRF guarantee it provides. We believe that this is the first formalisation of a weak memory model. The exercise has provided significant clarifications of the official definition and proof, turning up considerable flaws. Our formalisation reveals a design space for changes that preserve the DRF guarantee, enabling us to suggest fixes for the recently discovered bug while preserving the key guarantee for programmers. When testing our fixes on the JMM test cases [7] we have discovered another flaw in the memory model and we offer a solution for it, as well. While we have intentionally stayed close to the informal definitions, we have made some simplifications. The most notable is that we require executions to be finite, because infinite executions cause some tricky problems without, we believe, illuminating the DRF proof (see Sect. 4 for further details).

Overview. The rest of this paper is organised as follows. In Sect. 2, we introduce the basic definitions of the JMM, as we have formalised them in Isabelle, together with some commentary to assist the reader. In Sect. 3 we give our proof of the DRF guarantee. This considerably expands the informal proof of [8]. Sect. 4 summarises our results, collecting together changes to the JMM to repair the

bugs, discussions of the differences between our formalisation and the official informal work, and the improvements we claim to have made. Sect. 5 concludes, mentioning related work on other formal methods applied to memory models, as well as our plans for future work building on the results here.

2 Formal Definitions for the JMM

The following definitions are from [2, 1], with minor differences described at the end of the section. We use abstract types $\mathcal{T}$ for thread identifiers, ranged over by t; $\mathcal{M}$ for synchronisation monitor identifiers, ranged over by m; $\mathcal{L}$ for variables (i.e., memory locations), ranged over by v (in examples, `x`, `y`, etc.); and $\mathcal{V}$ for values. The starting point is the notion of *action*.

Definition 1 (Action). *An* action *is a memory-related operation; it is modelled by an abstract type $\mathcal{A}$ with the following properties: (1) Each action belongs to one thread, we will denote it by $T(a)$. (2) An action is one of the following* action kinds*:*

- volatile read *of* $v :: \mathcal{L}$,
- volatile write *to* $v :: \mathcal{L}$,
- lock *on monitor* $m :: \mathcal{M}$,
- unlock *on monitor* $m :: \mathcal{M}$,
- normal read *from* $v :: \mathcal{L}$,
- normal write *to* $v :: \mathcal{L}$.

An action kind also includes the associated variable or monitor. The volatile read, write, lock and unlock actions are called synchronisation actions*.*

In contrast to an interleaved semantics, in relaxed memory models there is no total ordering of all actions by time. Instead, the JMM imposes a total order, called *synchronisation order*, on all synchronisation actions, while allowing non-volatile reads and writes to be reordered to a certain degree. To capture the intra-thread ordering of actions, there is a total order, called *program order*, on all actions of each thread, which does not relate actions of different threads. These orders are gathered together in the definition of an *execution*. An execution allows non-deterministic behaviours because the order and visibility of memory operations may not be fully constrained.

Definition 2 (Execution). *An* execution *E is a tuple $E = \langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$, where*

- *$A \subseteq \mathcal{A}$ is a set of actions,*
- *P is a program, which is represented as a function that for each thread decides validity of a given sequence of action kinds with associated values if the action kind is a read or a write,*
- *the partial order $\leq_{po} \subseteq A \times A$ is the program order,*
- *the partial order $\leq_{so} \subseteq A \times A$ is the synchronisation order,*
- *$V :: \mathcal{A} \Rightarrow \mathcal{V}$ is a* value-written *function that assigns a value to each write from A, $V(a)$ is unspecified for non-write actions a,*

– $W :: \mathcal{A} \Rightarrow \mathcal{A}$ *is a* write-seen *function. It assigns a write to each read action from* A*, the* $W(r)$ *denotes the write seen by* r*, i.e. the value read by* r *is* $V(W(r))$*. The value of* $W(a)$ *for non-read actions* a *is unspecified.*

Following Lamport [9], an approximation of global time is now defined by the *happens-before* relation of an execution. An action a *happens-before* b, written $a \leq_{hb} b$, if either: (1) a and b are a *release-acquire* pair, such as an unlock of a monitor m and a lock of the same m, where a is ordered before b by the synchronisation order, or (2) a is before b in the program order, or (3) there is c such that $a \leq_{hb} c$ and $c \leq_{hb} b$.

Fig. 1 shows two simple cases of the happens-before relation. In the first execution, the write `x := 42` happens-before the read of x because the accesses to x are protected by the monitor m; the lock of m is ordered after the unlock of m by the synchronisation order in the given execution. Similarly, a volatile write and a volatile read from the same variable are a release-acquire pair. In the second execution, the happens-before includes two such pairs for the variable `v`. This means that the three instructions in thread 2 must happen in between the first two and last two instructions in thread 1.

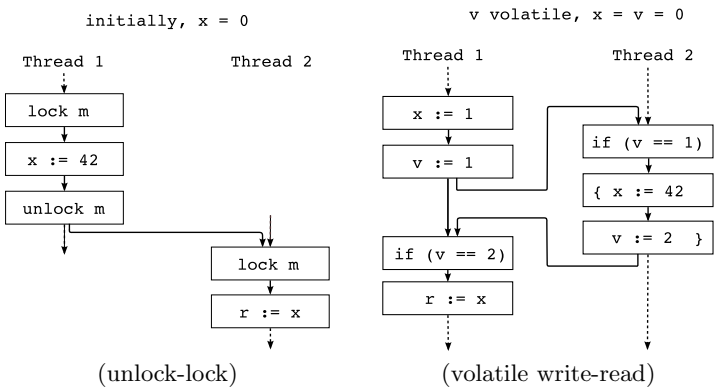


Fig. 1. Happens before by release-acquire pairs

The precise definition of the happens-before relation is provided via an auxiliary *synchronises-with* ordering. Each is derived from a given execution.

Definition 3 (Synchronizes-with). *In an execution with synchronisation order* $\leq_{so}$*, an action* a synchronises-with *an action* b *(written* $a <_{sw} b$*) if* $a \leq_{so} b$ *and* a *and* b *satisfy one of the following conditions:*

– a *is an unlock on monitor* m *and* b *is a lock on monitor* m*,*
– a *is a volatile write to* v *and* b *is a volatile read from* v*.*

Definition 4 (Happens-before). *The* happens-before *order of an execution is the transitive closure of the composition of its synchronises-with order and its program order, i.e.* $\leq_{hb} = (<_{sw} \cup \leq_{po})^{+}$*.*

To relate a (sequential) program to a sequence of actions performed by one thread we must define a notion of *sequential validity*. The next definition is new to our formalisation and was left implicit in [1], as was the notion of program itself. To be as abstract as possible, we consider programs as thread-indexed predicates on sequences of pairs of an action kind and a value. An action kind and value pair determines the effect of a memory operation (e.g. a read from a variable, or a write to a variable with a given value).

Definition 5 (Sequential validity). *We say that a sequence s of action kind-value pairs is* sequentially valid *with respect to a thread t and a program P if $P(t, s)$ holds.*

The next definition places some sensible restriction on executions.

Definition 6 (Well-formed execution). *We say that an* execution $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ *is* well-formed *if*

1. *A is finite.*
2. *$\leq_{po}$ restricted on actions of one thread is a total order, $\leq_{po}$ does not relate actions of different threads.*
3. *$\leq_{so}$ is total on synchronisation actions of A.*
4. *$\leq_{so}$ is consistent with $\leq_{po}$.*
5. *W is properly typed: for every non-volatile read $r \in A$, $W(r)$ is a non-volatile write; for every volatile read $r \in A$, $W(r)$ is a volatile write.*
6. *Locking is proper: for all lock actions $l \in A$ on monitors m and all threads t different from the thread of l, the number of locks in t before l in $\leq_{so}$ is the same as the number of unlocks in t before l in $\leq_{so}$.*
7. *Program order is intra-thread consistent: for each thread t, the sequence of action kinds and values[1] of actions performed by t in the program order $\leq_{po}$ is sequentially valid with respect to P and t.*
8. *$\leq_{so}$ is consistent with W: for every volatile read r of a variable v we have $W(r) \leq_{so} r$ and for any volatile write w to v, either $w \leq_{so} W(r)$ or $r \leq_{so} w$.*
9. *$\leq_{hb}$ is consistent with W: for all reads r of v it holds that $r \not\leq_{hb} W(r)$ and there is no intervening write w to v, i.e. if $W(r) \leq_{hb} w \leq_{hb} r$ and w writes to v then[2] $W(r) = w$.*

2.1 Legal executions

The definition of *legal execution* constitutes the core of the Java Memory Model, but it appears convoluted at first sight.

[1] The *value* of an action a is $W(a)$ if a is a write, $V(W(a))$ if a is a read, or an arbitrary value otherwise.

[2] The Java Specification omits the part "$W(r) = w$", which is clearly wrong since happens-before is reflexive (perhaps counterintuitively!).

Definition 7 (Legal execution). *A well-formed execution* $\langle A, P, \leq_{po}, \leq_{so}, W, V\rangle$ *with happens before order* $\leq_{hb}$ *is* legal *if there is a finite sequence of sets of actions* C_i *and well-formed executions* $E_i = \langle A_i, P, \leq_{po_i}, \leq_{so_i}, W_i, V_i\rangle$ *with happens-before* $\leq_{hb_i}$ *and synchronises-with* $<_{sw_i}$ *such that* $C_0 = \emptyset$, $C_{i-1} \subseteq C_i$ *for all* $i > 0$, $\bigcup C_i = A$, *and for each* $i > 0$ *the following rules are satisfied:*

1. $C_i \subseteq A_i$.
2. $\leq_{hb_i} |_{C_i} = \leq_{hb} |_{C_i}$.
3. $\leq_{so_i} |_{C_i} = \leq_{so} |_{C_i}$.
4. $V_i|_{C_i} = V|_{C_i}$.
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$.
6. *For all reads* $r \in A_i - C_{i-1}$ *we have* $W_i(r) \leq_{hb_i} r$.
7. *For all reads* $r \in C_i - C_{i-1}$ *we have* $W_i(r) \in C_{i-1}$ *and* $W(r) \in C_{i-1}$.

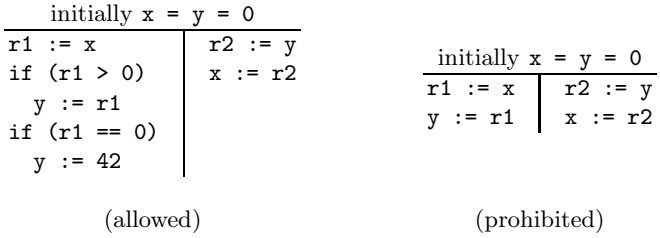
To get some intuition, we explain some of the behaviours that the memory model tries to rule in and out, and how they are argued about. The structure of reasoning is to consider the desired outcome, and then explain how it can (or cannot) arise by some (or any) execution. When considering a possibly-legal execution, we argue that it is allowed by repeatedly extending a committing sequence with actions which could happen in some speculated execution E_i. The aim is to commit the entire execution.

As a first example, consider the program on the first page and a desired execution in which the result is `r1` = `r2` = `1`. To denote actions, we use $W(x, v)$ for a write of value v to variable x and $R(x)$ for a read from x. We start with C_1 being two (implicit) initialisation actions which assign the default values: $C_1 = \{W(x, 0), W(y, 0)\}$. The two reads must each see the writes of value 1, which are in opposite threads. Since writes must be committed before the reads that see them (rule 7 of legality), we next commit the writes $C_2 = C_1 \cup \{W(x, 1), W(y, 1)\}$. Finally, we can commit the reads $C_3 = C_2 \cup \{R(x), R(y)\}$. Note that we do not need any speculation, we can justify this committing sequence using just one justifying execution—the one where the reads see the default writes and all the writes write 0.

Another example of a legal execution is for the first program in Fig. 2 where the read of x sees the write of x in the other thread and the read of y sees the first write of y in the first thread. We start with $C_1 = \{W(x, 0), W(y, 0)\}$. To commit the reads, we need to commit the writes with values 42 first (rules 4 and 7). As we cannot commit `x := r2` with the value 42 yet (rule 6 and intra-thread consistency), we commit $C_2 = C_1 \cup \{W(y, 42)\}$ and justify it by an execution performing the write `y:=42`. Committing the read `r2:=y` ($C_3 = C_2 \cup \{R(y)\}$) enables this read to see the write $W(y, 42)$ in E_4 (rule 5), hence we can commit the write to x with the value 42: $C_4 = C_3 \cup \{W(x, 42)\}$. Finally, committing the read `r1:=x` completes the justifying sequence. Note that in the final execution, the instruction performing the write of 42 to y is different from the one that was used to commit it!

In contrast, the out of thin air behaviour in the second program of Fig. 2 cannot be achieved by any legal execution. Let's assume it can and consider the committing sequence. The writes of 42 must be committed before the reads

(rules 4 and 7), hence when justifying the first committed of the two writes of 42 none of the reads is committed yet, thus both the reads must see the writes that happen-before them (rule 6). The only writes that happen-before the reads are the default writes of the value 0. As a result, it is not possible to justify the first write of the value 42.



(allowed) (prohibited)

Is it possible to get r1 = r2 = 42 at the end of an execution?

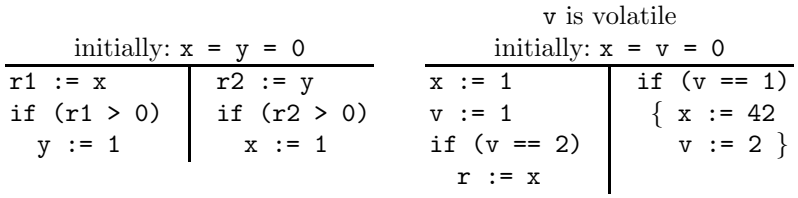
Fig. 2. Examples of legal and illegal executions.

Definition 8 (Sequential consistency). *An execution is* sequentially consistent *if there is a total order consistent with the execution's program order and synchronisation order such that every read in the execution sees the most recent write in the total order.*

Definition 9 (Conflict). *An execution has a* conflicting *pair of actions a and b if both access the same variable and either a and b are writes, or one of them is a read and the other one is a write.*

Definition 10 (DRF). *A program is* data race free *if in each sequentially consistent execution of the program, for each conflicting pair of actions a and b in the execution we have either* $a \leq_{hb} b$ *or* $b \leq_{hb} a$.

With this definition of data race freedom, the DRF guarantee is very strong. For example, using this definition, both the programs below are data race free.



The DRF guarantee ensures that in the first program [8], the reads of x and y will always see 0. In the second program, the DRF guarantees that the read r:=x can only see the write x:=42. Note that if v would not be volatile, the program is not data race free and the read of x could see the value 1 (but not 0). Section 4 contains further discussion on the above definitions.

2.2 Notes on Formalisation

In our formalisation, we define several notions slightly differently to the informal specification in [1]. We define actions as an abstract data type (Definition 1) instead of a tuple containing a unique action identifier, a thread identifier, an action kind and object involved in the action. Definition 2 (execution) differs from the one of [1] by omitting the synchronises-with and happens-before orders, which are derived in terms of the program and synchronisation orders, and by giving precise meaning to the notion of programs. Our definition of well-formed executions (Definition 6) requires the set of actions to be finite instead of just restricting the synchronisation order to an omega order. We will discuss the motivation for this change in Sect. 4. We omit action kinds irrelevant for the DRF guarantee proof – external actions and thread management actions, such as thread start, thread finish, thread spawn and thread join[3]. Compared to definition of legality in [1], we have removed rules 8 and 9 for legality as they are not important for the proof. Finally, we strengthen the notion of sequential consistency to respect mutual exclusivity of locks. See Sect. 4 for further details.

3 Proof of DRF guarantee

Definition 11 (Well-formed program). *We say that a* program P *is* well-formed *if for all threads t and action kind-value sequences s sequential validity of s with respect to t and P implies*

- *sequential validity of all prefixes u of s with respect to t and P (P is prefix-closed), and*
- *sequential validity of all sequences u obtained from s by changing the value of the last action-value pair to a different value with respect to t and P, provided that the last action in s is a read (P is final read agnostic).*

Lemma 1. *If $\leq_p$ is a partial order on A and $\leq_t$ is a total order on $S \subseteq A$, then $\leq_q = (\leq_p \cup \leq_t)^+$ is a partial order on A.*

Lemma 2. *For any well-formed execution of a data race free well-formed program, if each read sees a write that happens-before it, the execution is sequentially consistent.*

Proof. Using Lemma 1, $(\leq_{so} \cup \leq_{po})^+$ is a partial order. Let's take a topological sort $\leq_t$ on $(\leq_{so} \cup \leq_{po})^+$. Since $\leq_t$ is a total order on a finite set, it must be well-founded. We will prove the sequential consistency using well-founded induction on $\leq_t$.

Assume that we have a read r and all reads $x \leq_t r$ see the most recent write. We will show that r also sees the most recent write by contradiction.

Assume that r does not see the most recent write and let w be the most recent write to the variable read by r. Let A' be $\{x \mid x \leq_t r\}$. Then the execution

[3] However, we have formalised a version that contains all these actions.

$E' = \langle A', P, \leq_{po} |_{A' \times A'}, \leq_{so} |_{A' \times A'}, W[r \mapsto w], V \rangle$ is a well-formed execution (see Lemma 3). From the induction hypothesis, E' is sequentially consistent (all reads in E' see the most recent write). From the well-formedness of E it cannot be the case that $W(r) \leq_{hb} w \leq_{hb} r$, i.e. either $W(r)$ and w, or w and r are a conflicting pair of actions in both E and E'.

As a result, E' is a well-formed sequentially consistent execution with a conflict. This is a contradiction with data race freedom of P.

The proof of Lemma 2 is broadly similar to the informal proof in [1]. There is a minor difference in performing a topological sort on $(\leq_{so} \cup \leq_{po})^+$ instead of happens-before order. This guarantees consistency with the synchronisation order, which we use in the next lemma. A more significant difference is restricting the execution to all actions before r in $\leq_t$ order and updating $W(r)$ to see w, instead of constructing a sequentially consistent completion of the execution. The reason is that sequentially consistent completion might require to insert initialisation actions to the beginning of the execution, hence the original proof does not work as stated, because it assumes that the sequentially consistent completion only adds actions to the "end" of the execution (in terms of $\leq_t$). We discuss related issues in Sect. 4.

Lemma 3 (Cut-and-update). *Assume that P is a well-formed program, $\langle A, P, \leq_{po}, \leq_{so}, W, V \rangle$ is a well-formed execution, $\leq_{hb}$ its happens-before order, $\leq_t$ is a total order on A, $r \in A$ is a read action of variable v, and $w \in A$ is a write action to v such that*

- *$\leq_t$ is consistent with $\leq_{so}$ and $\leq_{hb}$,*
- *for every read $r \in A$ we have $W(r) \leq_{hb} r$,*
- *w is the most recent write to r in $\leq t$, i.e. $w \leq_t r$ and for all writes w' to variable v either $w' \leq_t w$ or $r \leq_t w'$,*

Let A' be $\{x \mid x \leq_t r\}$. Then the execution $\langle A', P, \leq_{po} |_{A' \times A'}, \leq_{so} |_{A' \times A'}, W[r \mapsto w], V \rangle$ is a well-formed execution.

Proof. It is straightforward (although tedious) to establish all the conditions from Def. 6 for well-formedness of $\langle A', P, \leq_{po} |_{A' \times A'}, \leq_{so} |_{A' \times A'}, W[r \mapsto w], V \rangle$.

The following is our equivalent of Theorem 3 from [1].

Theorem 1 (DRF guarantee). *Any legal execution E of a well-formed data race free program is sequentially consistent.*

Proof. From Lemma 2 it is sufficient to show that every read in E sees a write that happens-before it.

From the legality of E we have a committing sequence $\{C_i, E_i\}$ justifying E. We will show by induction on i that all reads in C_i see writes that happens-before them.

Base case is trivial ($C_0 = \emptyset$). For the induction step, let's assume that all reads $r \in C_{i-1}$ we have $W(r) \leq_{hb} r$. We will show that for any read $r \in C_i$ we get $W(r) \leq_{hb} r$.

Note that E_i is sequentially consistent: Let $\leq_{hb_i}$ be happens-before of E_i. Using the induction hypothesis with rules 2, 5 we obtain $W(r) \leq_{hb_i} r$ for all reads $r \in C_{i-1}$. Using rule 6, for all reads $r \in A_i - C_{i-1}$ we have $W(r) \leq_{hb_i} r$. Thus, E_i is sequentially consistent by Lemma 2.

Since the program is correctly synchronised, E_i is sequentially consistent, and r and $W(r)$ are accessing the same variable, r and $W(r)$ must be ordered by $\leq_{hb_i}$ in E_i and consequently in E (rules 2 and 7). From well-formedness of E we have $r \not\leq_{hb} W(r)$, hence it must be $W(r) \leq_{hb} r$. This proves the induction step.

Since each read has to appear in some C_i we have proved that all reads in E see writes that happen-before them, thus E is sequentially consistent.

Note that we use rule 2 for legality ($\leq_{hb_i} |_{C_i} = \leq_{hb} |_{C_i}$) only to compare $W(r)$ and r. Rules 1, 3, 4 and the first part of rule 7 for legality (i.e. $\forall r \in C_i - C_{i-1}. r$ is read $\Rightarrow W_i(r) \in C_{i-1}$) are not used in the proof at all. Also note that rule 1 is implied by rule 2.

4 Results of the Formalisation

Our formalisation has resulted in a clarification of the JMM definitions in several areas. We have also deviated from the JMM in a few places for simplification.

The most significant changes are restricting the executions to be *finite* and *omitting default initialisation* actions because of inconsistencies in the JMM definitions. Fixing these inconsistencies introduces strange artifacts in infinite executions, which is why we confine the executions to be finite. As a side product, this also simplifies proofs without sacrificing anything important, we believe.

We have improved the memory model in several ways. We have clarified the notion of *intra-thread consistency* and identified the precise interface with the sequential part of Java. The JMM does not describe a precise connection to the inter-thread Java specification, which makes it impossible to validate executions of very simple programs. Therefore, we have formalised a version of the JMM that strengthens the definition of well-formed execution by requiring executions to be *inter-thread consistent*. Moreover, we strengthen the DRF guarantee by considering a stronger definition of sequential consistency that requires existence of a total order consistent with program order *and synchronisation order*, making it respect mutual exclusion. In the proof itself, we fix small inconsistencies and eliminate proof by contradiction in favour of induction where possible, and we state and prove the technically challenging Cut-and-update lemma (Lemma 3).

Finally, we identify the requirements on legality that are necessary to prove the DRF guarantee. This allows us to formulate a weaker notion of legality that fixes the recently discovered bugs in the memory model.

In the following paragraphs we discuss each of the contributions in detail.

Fixing the Bugs by Weakening Legality. The formal proof of Theorem 1(DRF guarantee) revealed that Definition 7 (legality) is unnecessarily strong. We propose (and have checked) a weaker legality requirement by omitting the rule 3

($\leq_{so_i}|_{C_i} = \leq_{so}|_{C_i}$) and replacing rules 2 and 7 in Definition 7 by the following weaker ones[4]:

2. For all reads $r \in C_i$ we have $W(r) \leq_{hb} r \iff W(r) \leq_{hb_i} r$, and $r \not\leq_{hb_i} W(r)$,
7. For all reads $r \in C_i - C_{i-1}$ we have $W(r) \in C_{i-1}$.

This fixes the counterexample to Theorem 1 of [1] discovered by Cenciarelli et al [3]. We conjecture that this fix would also enable a correct proof to be given.

While manually testing the weaker notion of legality on causality tests [7], we have discovered a previously unreported problem of the existing JMM—it violates the causality tests 17–20. Our weaker legality (rule 7) validates these examples while properly validating the other test cases. For example, causality test case 20 says that the program below should allow the result `r1` = `r2` = `r3` = 42, because optimisations may produce this (see [7] for details):

initially `x = y = 0`

`T1`	`T2`	`T3`
`join T3`	`r2 := y`	`r3 := x`
`r1 := x`	`y := r2`	`if (r3 == 0)`
`y := r1`		`x : = 42`

However, it appears that this is not legal, because when committing the write `r1:=x`, the write of 42 to y must be already committed, hence the read of x must see a write to x with value 42. By rules 6 and 7 of legality, this write to x must be committed and it must happen-before the read. However, there is no such write. With the weaker legality we are not constrained by rule 7.

Inconsistency for Initialisation Actions. The Java Memory Model requires that

> The write of the default value (zero, false, or null) to each variable synchronises-with to the first action in every thread;

and defines synchronises-with as an order over synchronisation actions. This is inconsistent with the definition of synchronisation actions; ordinary writes are not synchronisation actions. Moreover, the JMM does not specify the thread associated with the initialisation actions and how they are validated with respect to intra-thread consistency.

There are several solution to this. The SC- memory model [10] suggests having a special initialisation thread and let its termination synchronise-with first actions of other threads. In our approach, we have tried to stay close to the JMM and we have defined a new kind of synchronisation action—an initialisation action. These actions belong to a special initialisation thread that contains only the initialisation actions for all variables accessed in the program. The initialisation actions behave like ordinary write actions except they synchronise-with the

[4] Note that we must keep rule 4 — it is needed to prohibit the out of thin air behaviours.

thread start action of each thread. Note that in both the SC- and our formalisation, there is no significant difference between treating initialisation actions differently from normal write actions[5].

Infinite Executions. The JMM does not require finiteness of well-formed executions. Instead, the synchronisation order must be an omega order. Although it is not stated explicitly in the JMM specification, the discussion of fairness in [8] implies a progress property that requires threads in finite executions to be either terminated or deadlocked. Combination of the progress with the initialisation actions introduces an unpleasant artifact. We would expect that the program

```
while(true) { new Object() { public volatile int f; }.f++; }
```

has just one infinite execution. However, an infinite number of variables `f` must be initialised. These initialisations must be ordered before the start of the thread performing the loop, which violates omega-ordering of the synchronisation order. As a result, this program does not have any well-formed execution in the current JMM. Similar issue also breaks the proof of Lemma 2 in [1], as noted in the paragraph following the proof given there. In our approach we do not require the finite executions to be finished[6]. This allows us to use the Cut-and-update lemma instead of the sequentially consistent completion used in [1].

Sequential Consistency. Sequentially consistency in the JMM requires existence of a total order consistent with program order such that each read sees the most recent write in that total order. This definition allows the program

initially `x = y = z = 0`

`r1 := y`	`lock m`	`lock m`
`x := r1`	`r2 := x`	`y := 1`
	`z := 1`	`r3 := z`
	`unlock m`	`unlock m`

to have a sequentially consistent execution resulting in `r1` = `r2` = `r3` = 1 , using the total order `lock m`, `lock m`, `y:=1`, `r1:=y`, `x:=r1`, `r2:=x`, `a:=1`, `r3:=a`, `unlock m`, `unlock m`. Taking the most recent writes as values of reads, this order surprisingly results into `r1` = `r2` = `r3` = `1`. One can check that this cannot happen in any interleaved execution respecting mutual exclusivity of locks.

To make sequential consistency compatible with the intuitive meaning of interleaved semantics, we have formalised sequential consistency by requiring existence of a total order consistent with *both* the program order and synchronisation order and proved the DRF guarantee for this case. Since our definition

[5] We actually have two versions of the formalisation—one with the initialisation actions and one without them.

[6] We believe that this does not affect the definition of observational equivalence, since observable behaviours are finite by the definition from [1]. We have not checked this formally.

of sequential consistency is stricter, the definition of data race free program becomes weaker[7], and consequently the DRF guarantee in the original setting directly follows from our DRF guarantee for stronger sequential consistency.

Requirements on Thread Management Actions. For thread management, the JMM introduces further actions for thread start, termination, spawn and join. In well-formed executions, the JMM requires certain actions to be ordered by the synchronises-with order, e.g., thread spawn must synchronise-with the corresponding thread start, thread finish synchronises-with thread join. However, it is also necessary to prohibit executions from other undesirable behaviours, such as running threads that have not been spawned.

To address this in our formalisation, we introduce a constant for application's main thread and we specify that (1) for any thread except the main thread there is a unique thread spawn action, (2) there is no thread spawn action for the main thread, (3) each thread has a unique thread start action that is ordered before all actions of that thread in the program order, (4) for each thread there is at most one thread terminate action, and there is no action ordered after the terminate action in the program order.

5 Conclusion

We have formally verified the guarantee of sequential consistency for data race free programs in the Java Memory Model. We believe that this is the first formalisation of a weak memory model and its properties; it has shown that the current JMM is still flawed, although perhaps not fatally so. The topic was an ideal target for formalisation: we started from a highly complex informal definition that is difficult to understand and for which it is hard to be certain about the claims made. Indeed, while we were working on the DRF proof (Theorem 3 of [1] and the first part of the JMM contract) Cenciarelli et al [3] meanwhile showed a counterexample to Theorem 1 of [1] that promised legality of reordering of independent statements (the second part of the contract). We discovered another bug while justifying causality test cases [7]. The third part of the contract, the out of thin air promise, was not proved or stated in [1], and is demonstrated only on small prohibited examples so far.

In keeping with the TPHOLs recommendation for good cross-community accessibility of papers, we have avoided theorem prover-specific syntax throughout. The formalisation occupies about 4000 lines in Isabelle/Isar [11], and takes under a minute to check. Our definitions build on the standard notion of relations in Isabelle/HOL as a two-place predicate together with a set representing the domain; nevertheless we had to establish some further properties required for the proof. To explore all definitions, we had to produce multiple versions of the scripts; current theorem prover technology still makes this harder than it ought

[7] In fact, both the definitions of data race free programs are equivalent.

to be. For those who are interested in the Isabelle specifics, we have made our formalisation available on the web.[8]

Related Work. In the computer architecture world, there is a long history of work on weak memory models. These models usually specify allowed memory operation reorderings, which either preserve sequential consistency for data race free programs, i.e. they provide the DRF guarantee [12], or provide mechanisms for enforcing ordering (memory barriers). For a detailed survey, see [4, 13].

Until recently, the programming languages community has mostly ignored the issues of memory models and relied on specifications of the underlying hardware (this is the case for, e.g., C, C++, and OCaml). But programmers write programs that execute incorrectly under memory models weaker then sequential consistency. For example, Peterson's mutual exclusion algorithm and double-checked locking are known to be incorrect for current hardware architectures [14, 15]. This has encouraged recent progress on adapting the work on hardware memory models to define and explain the memory models of programming languages [16, 17]. However, these techniques either violate the out of thin air property, or do not allow advanced optimisation techniques that remove dependencies. This was the reason for Java to adopt the memory model based on speculations [1, 18], which has spawned semantic investigation like [3] as well as our own study.

Likewise, little work has yet been done to formalise behaviours of hardware or software memory models in formal verification community. As far as we know all program logics and theorem prover based methodologies [19–21] for concurrent shared memory assume interleaved semantics, which implies sequential consistency. The DRF guarantee, formalised in this paper, ensures that these methods are correct for data race free programs. The data race freedom can be established inside the program logics themselves, or using dedicated type systems [22, 23]. A complete framework for thread-modular verification of Java programs, including the data race freedom and atomicity verification, is currently being developed in the Mobius project [24].

More work has been done to verify properties of small programs or data structures using state exploration techniques. Huynh and Roychoudhury [14] check properties of C# programs by examining all reachable states of a program, while accounting for reorderings allowed by the CLI specification [25]. Burckhardt et al [26] use a SAT solver to perform bounded tests on concurrent data structures in relaxed memory models. To explore proposals for replacing the early JMM [16, 27], Yang et al [28] built a flexible framework for modelling executions in memory models with complex reordering and visibility constraints. To our knowledge, the only formal verification tool for the current JMM is the data race analysis by Yang et al [29]. However, data race freedom is a relatively easy property to show—only sequentially consistent executions need to be examined. The declarative nature of the JMM and its speculative justifications make it very hard to explore all executions of the given program; the obligation to find an execution for each committed action quickly blows up the state space. Our the-

[8] Please see: `http://groups.inf.ed.ac.uk/request/jmmtheory/`

orem proving approach differs from state exploration in the typical way: we aim to verify general properties of the memory model rather than justify behaviour of individual programs automatically.

Future Work. The next direction for our analysis is to formalise the description and verification of the techniques used to justify optimisations in the allowed causality tests for the JMM [7]. We hope to begin this by formally checking Theorem 1 of [1] with our fix to legality.

The out of thin air guarantee is the main motivation for the JMM. Without this requirement, we could define the memory model easily by allowing all code transformations and executions that appear sequentially consistent for DRF programs [12] and leave the behaviour for programs that are not correctly synchronised unspecified. The notion of "out of thin air" is mostly example driven—we only have specific behaviours of concrete programs that should be prohibited and a general intuition that in two groups of threads, each group using disjoint sets of variables, there should not be any information leak from one group to the other, e.g., a reference allocated by a thread in the first group should not be visible to any read performed by a thread in the other group. We plan to state and prove these properties formally.

Although it is not described here, we have instantiated our framework with a simple sequential language that is sufficient to express simple examples, such as the ones in Fig. 2 or in [7]. However, the backwards style of reasoning in the JMM makes it difficult to show illegality of individual executions and we need to find better ways. Moreover, adding allocation to the language makes it impossible to verify validity of a thread trace separately, because we need to ensure freshness globally. To address this, our idea is to index locations by their allocating thread, but we have not explored the impact of this yet.

Finally, we hope that with a simplified and clear memory model, we will have a firmer basis for studying logics for concurrent programs which respect weaker memory models, which is our overall objective.

References

1. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, New York, NY, USA, ACM Press (2005) 378–391
2. Gosling, J., Joy, B., Steele, G., Bracha, G.: Memory Model. In: Java(TM) Language Specification, The (3rd Edition) (Java Series). Addison-Wesley Professional (2005) 557–573
3. Cenciarelli, P., Knapp, A., Sibilio, E.: The Java memory model: Operationally, denotationally, axiomatically. In: 16th ESOP. (2007)
4. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer **29**(12) (1996) 66–76
5. Pugh, W.: The Java memory model is fatally flawed. Concurrency - Practice and Experience **12**(6) (2000) 445–455
6. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley Professional (2005)

7. Pugh, W., Manson, J.: Java memory model causality test cases (2004) http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html.
8. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. Submitted to Special POPL Issue. (2005)
9. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
10. Adve, S.: The SC- memory model for Java (2004) http://www.cs.uiuc.edu/~sadve/jmm.
11. Wenzel, M.: The Isabelle/Isar reference manual (2005) http://isabelle.in.tum.de/doc/isar-ref.pdf.
12. Adve, S.V., Aggarwal, J.K.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. **4**(6) (1993) 613–624
13. Kawash, J.: Limitations and Capabilities of Weak Memory Consistency Systems. PhD thesis, The University of Calgary (2000)
14. Roychoudhury, A.: Formal reasoning about hardware and software memory models. In: ICFEM, London, UK, Springer-Verlag (2002) 423–434
15. Bacon, D., et al.: The "double-checked locking is broken" declaration (2001) http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.
16. Maessen, J.W., Shen, X.: Improving the Java memory model using CRF. In: OOPSLA, New York, NY, USA, ACM Press (2000) 1–12
17. Saraswat, V., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: ACM 2007 SIGPLAN Conference on Principles and Practice of Parallel Computing, ACM (2007)
18. Manson, J.: The Java memory model. PhD thesis, University of Maryland, College Park (2004)
19. Ábrahám, E., de Boer, F.S., de Roever, W.P., Steffen, M.: An assertion-based proof system for multithreaded Java. TCS **331**(2-3) (2005) 251–290
20. Flanagan, C., Freund, S.N., Qadeer, S., Seshia, S.A.: Modular verification of multithreaded programs. Theor. Comput. Sci. **338**(1-3) (2005) 153–183
21. Moore, J.S., Porter, G.: The apprentice challenge. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 193–216
22. Flanagan, C., Freund, S.N.: Type-based race detection for Java. In: PLDI, New York, NY, USA, ACM Press (2000) 219–232
23. Boyapati, C., Rinard, M.: A parameterized type system for race-free Java programs. In: OOPSLA, New York, NY, USA, ACM Press (2001) 56–69
24. Huisman, M., Grigore, R., Haack, C., Hurlin, C., Kiniry, J., Petri, G., Poll, E.: Report on thread-modular verification (2007) Mobius project deliverable D3.3. Available from http://mobius.inria.fr.
25. Microsoft: Standard ECMA-335 Common Language Infrastructure (CLI) (2005)
26. Burckhardt, S., Alur, R., Martin, M.M.K.: Bounded model checking of concurrent data types on relaxed memory models: A case study. In Ball, T., Jones, R.B., eds.: CAV. Volume 4144 of LNCS., Springer (2006) 489–502
27. Manson, J., Pugh, W.: Semantics of multithreaded Java. Technical Report CS-TR-4215, Dept. of Computer Science, University of Maryland, College Park (2001)
28. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an operational memory model specification framework with integrated model checking capability: Research articles. Concurr. Comput. : Pract. Exper. **17**(5-6) (2005) 465–487
29. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Memory-model-sensitive data race analysis. In Davies, J., Schulte, W., Barnett, M., eds.: ICFEM. Volume 3308 of LNCS., Springer (2004) 30–45