The Java Memory Model: a Formal Explanation [1]

M. Huisman[2] G. Petri[3]

INRIA Sophia Antipolis, France

Abstract

This paper discusses the new Java Memory Model (JMM), introduced for Java 1.5. The JMM specifies the allowed executions of multithreaded Java programs. The new JMM fixes some security problems of the previous memory model. In addition, it gives compiler builders the possibility to apply a wide range of singlethreaded compiler optimisations (something that was nearly impossible for the old memory model). For program developers, the JMM provides the following guarantee: if a program does not contain any data races, its allowed behaviours can be described with an interleaving semantics.
This paper motivates the definition of the JMM. It shows in particular the consequences of the wish to have the data race freeness guarantee and to forbid any *out of thin air* values to occur in an execution. The remainder of the paper then discusses a formalisation of the JMM in Coq. This formalisation has been used to prove the data race freeness guarantee. Given the complexity of the JMM definition, having a formalisation is necessary to investigate all aspects of the JMM.

Keywords: Java Memory Model, formalisation, Data-Race-Freeness Guarantee

1 Introduction

With the emergence of multiprocessor architectures, shared memory has shown to be a simple and comfortable communication model for parallel programming that is both intuitive for programmers, and close to the underlying machine model. However, the use of shared memory requires synchronisation mechanisms to keep the memory of the overall system up-to-date and coherent. Such synchronisation mechanisms have a big impact on performance; to avoid these, several relaxations of the consistency (or coherence) of the memory system have been proposed [2,15,13]. However, these relaxations might cause the program to have unexpected behaviours (from the programmer's point of view). In general, the more relaxed the memory system, the harder it is to reason about the programs executing on it.

[1] This work is partially funded by the IST programme of the EC, under the IST-FET-2005-015905 Mobius project, and the French national research organisation (ANR), under the ANR-06-SETIN-010 ParSec project.

[2] Email: Marieke.Huisman@inria.fr

[3] Email: Gustavo.Petri@inria.fr

A *memory model* defines all the possible outcomes of a multithreaded program running on a shared memory architecture that implements it. In essence, it is a specification of the possible values that read accesses on the memory are allowed to return [4], and thus specifies the multithreaded semantics of the platform.

Java is one of the few major programming languages with a precisely defined memory model [19]. Java's initial memory model allowed behaviours with security leaks [21], and in addition, it prevented almost all singlethreaded compiler optimisations. Therefore, since Java 1.5, a new memory model has been introduced, that fixes these defects. The Java Memory Model (JMM) has been designed with two goals in mind: *(i)* as many compiler optimisations as possible should be allowed, and *(ii)* the average programmer should not have to understand all the intricacies of the model. To achieve the second goal, the JMM provides the following Data Race Freeness (DRF) guarantee: if a program does not contain data races, its allowed behaviours can be described by an interleaving semantics.

To capture basic ordering and visibility requirements on memory operations, the JMM is based on the happens before order [16]. This order inspires the so-called *happens before model*, identifying the actions that necessarily have to happen before any other actions. In other words, this order specifies the updates on the memory that any read must see. Only executions that do not violate this order are allowed. As the guarantees that this model provides are very weak, it allows many singlethreaded compiler optimisations. However, the happens before model allows executions in which values are produced *out of thin air* [5], by using circular justifications of actions, as will be discussed in Section 2. To avoid such circular justifications, and to guarantee DRF, the current version of the JMM is much more complex than the happens before model.

Ample literature about the JMM and related models exists [19,18,22,6], but most descriptions are very dense. In particular, the motivation of the justification procedure that defines legal executions is not extensively explained, i.e., it is unclear why it is defined as it is. This paper tries to overcome this problem by presenting our understanding of the JMM.

In addition, it also presents a formalisation of the JMM in Coq [9]. Our formalisation proves that the DRF guarantee holds for the JMM (a hand-written proof for this is given in [19], which contains some (minor) mistakes). As future work we plan to make a link with the Bicolano formalisation of the Java Virtual Machine (JVM) [20]. This motivates the use of Coq. We also plan to investigate other properties of the JMM formally, in particular the out-of-thin-air (OoTA) guarantee (however, this requires first to state formally what it means to avoid OoTA values).

The rest of this paper is organised is follows. Section 2 explains and motivates the definition of the JMM. Next, Section 3 describes our formalisation. Finally Section 4 describes how we plan to use our JMM formalisation further.

[4] We will in general—in conformance with memory model terminology—simply talk about "the write that a read sees", instead of "the write that writes the value in the memory returned by the read", as in general we do not care about the value, but only about the write action itself.

[5] A value that cannot be deduced from the program.

2 The Java Memory Model

The JMM, as previously mentioned, has two main goals. The first goal is: to allow as many compiler optimisations as possible. This is limited by the second goal: to make the task of multithreaded programming easier for the programmer. It is a well known fact that multithreaded programming is a hard task, and weak memory models further complicate this, in the sense that they add non-determinism to programs (caused by unexpected interleaving of actions). To achieve programmability, the approach of the JMM is that of data race free memory models [3,10,11]; i.e., the DRF guarantee, which reads:

> *Data Race Freeness*: Correctly synchronised programs have sequentially consistent semantics.

The notion of sequential consistency was first defined by Lamport [17]:

> "... the result 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."

This definition has several consequences. First, the sequence of operations determines that there exists a total order which is consistent with the program order (the order dictated by the sequence of instructions in each thread), and where each read of a memory location sees the last value written to that location. This implies that sequential consistent executions can be described by an *interleaving semantics*. Another important consequence is that the execution has to have a result as if it was executed in a total order, but the actual order of the execution does not need to be total. Thus, compiler optimisations and parallel execution of instructions are allowed, provided that they can be serialised. This makes sequential consistency a very attractive model for concurrency. While sequential consistency has a clear semantics, it is difficult for a compiler to determine statically whether it can or not rearrange intructions or allow parallel execution of operations preserving sequentially consistent semantics. Therefore, many (but not all) common compiler optimisations for sequential code are prevented by this semantics. For that reason, weaker semantics are proposed for programs that are not correctly synchronised (i.e., that contain data races).

A program is said to be correctly synchronised if its sequential consistent executions are free of data races. A data race occurs when two or more threads concurrently access a common memory location, where at least one of the accesses updates the memory. Usually, the presence of data races is considered a bug in the program. In general the result of such races cannot be predicted, thus the programmer must take care to avoid them. In addition, the architecture must provide guarantees to rule out data races. However, notice that sometimes data races are intended, so-called *benign data races*; even though programs with this kind of races are not common.

An important distinction between weak and strong models, is that the former distinguishes between normal memory operations and synchronisation operations, while the second does not. The basic idea is that synchronisation operations induce visibility and ordering restrictions on the other operations in the execution. Fur-

thermore, synchronisation operations in Java are intended to have a sequentially consistent semantics [19], i.e., there exists a total order of synchronisation operations, which is consistent with the program order, and where each synchronisation operation is aware of all the previous synchronisation operations in that order. The presence of such a total order guarantees that all processors see synchronisation operations in the same order (which is not the case for normal memory accesses). This order is called the *synchronisation order* (`so`). Lock and unlock operations on monitors and read and writes of volatile variables, are Java's main synchronisation operations. The order of actions issued by single threads is called the *program order* (`po`). Thus, `po` only relates actions of the same thread, while `so` can relate synchronisation actions of different threads.

The `so` relates synchronisation actions on different locations (either variables or monitors). This is, in general, too restrictive to define which are the minimal visibility conditions for actions. For example, read actions on different volatile variables are necessarily related by `so`, since these are synchronisation actions and `so` is total, but these need not impose restrictions among the threads involved. Therefore a weaker order, derived from the `so`, is defined; namely the *synchronises-with order* (`sw`). This is a per location restriction of the `so`, i.e., it only relates actions on the same location. The intuition of `sw` pairs is that they impose synchronisation of the memory between the intervening threads, in the sense that actions that happen before a synchronised write need to be visible by any other thread that can see that write via a volatile read that variable. The same intuition applies for unlock and lock actions. Thus, only unlock and volatile write actions appear in the source of a `sw` link while only volatile reads and unlock appear in the sink of the link. More precisely, `sw` links relate every volatile write with every subsequent (w.r.t. `so`) volatile read on the same variable, and every unlock with every subsequent (w.r.t. `so`) lock on the same monitor.

The `sw` and the `po` orders allow us to define what constitutes a data race in the JMM, captured by the happens before (`hb`) order. The `hb` order is formally defined as the transitive closure of the union of the `po` and `sw` orders, i.e., it extends the dependency of the `po` between different threads through `sw` edges. We say that two normal actions are conflicting if they operate on the same memory location and at least one of them is an update. An important note is that `so` and `po` must be consistent in JMM executions (i.e., synchronisation operations performed by a single thread must appear in `so` in the same order as they appear in `po`). This guarantees that `hb` is a partial order. A more operational intuition of these orders is that `hb` represents minimal visibility conditions, where `sw` links impose memory synchronisations between threads, and the actions of a single thread are aware of all previous actions (in `po`) by that thread. A *data race* in the JMM occurs when there are two (or more) conflicting actions not related by the `hb` order.

Thus, we can restate the DRF guarantee as:

> If every sequentially consistent execution of a program is free of data races, these are all the executions allowed for that program.

We will now see how the `hb` order serves to formally define the basics of the JMM. To start we describe a simpler model, the so-called *happens before memory*

model (HBMM) [19]. This uses the `hb` order to define which writes a certain read is allowed to see. In the HBMM a normal read `r` can see a write `w` on the same variable provided that[6]: *(i)* $\neg$ (`r` $\overset{hb}{\rightarrow}$ `w`), and *(ii)* for all writes `w'` on the same (normal) variable as `r`, $\neg$ (`w` $\overset{hb}{\rightarrow}$ `w'` $\overset{hb}{\rightarrow}$ `r`); in other words, a normal read can see any write on the same variable that is not related to it by `hb` (thus, forming a data race), or the write on the same variable that immediately precedes it in `hb`.

As mentioned before, volatile variables are meant to have a sequential consistent semantics. Therefore, for volatile variables the HBMM demands each volatile read to see the most recent volatile write on the same variable in the `so`.

Interestingly, the HBMM does not guarantee the DRF property desired for the JMM, as shown in Figure 1 (from [1,19]). In particular, Figure 1(a) shows how a value can be produced out of thin air (an OoTA value). The authors of the JMM decided that such behaviours should be avoided for all programs, thus, also programs with data races should avoid OoTA, one example of this kind of programs is given in Figure 1(b). We do not discuss other examples that motivate the JMM requirements[7], but it should be clear that avoiding OoTA values is one of the most important contributions of the JMM, and it is also the source of much of the complexity of its formal definition.

We will focus on the example in Figure 1(a) to see how the behaviour depicted could happen in the HBMM. First, notice that this program is correctly synchronised, because in every sequentially consistent execution none of the guarded writes is executed, therefore only the default writes are allowed to be seen by both reads. Now we will see that through speculative reasoning a compiler could determine that under the HBMM a behaviour producing an OoTA value is legal. Imagine that a compiler speculates that one of the writes could happen, if afterwards can justify it to happen it could optimise the program to make it happen always. Hence, assuming that any of the writes could happen we can justify such behaviour as follows: *(i)* assume first (w.l.o.g.) that the write of `42` to `x` in Thread 2 could happen, *(ii)* we conclude that the read of `x` in Thread 1 could see a value of `42`. *(iii)* This validates the guard, and justifies the write of 42 to `y` in Thread 1. *(iv)* With a similar reasoning as before the read of `y` in Thread 2 could see that write (reading a `42`) which validates the guard in Thread 2, and justifies the first write of `x` to happen, closing the circular justification cycle (we can see that *(i)* justifies *(i)* at the end). A similar kind of reasoning is applied to Figure 1(b) (but in that example it is more clear that the value `42` appears out of nothing for a programmer). This is the kind of behaviour that the authors of the JMM have named OoTA reads (usually associated to the self-justification of actions) and the actual JMM is designed as a restriction of the HBMM that disallows this kind of circular justifications. To achieve this, the JMM defines a special committing procedure that we explain below.

2.1 Formal Definitions

This subsection serves as a summary where we present the core definitions of the JMM [19], and in addition we give our intuitive understanding of these definitions.

[6] `r` $\overset{hb}{\rightarrow}$ `w` stands for: action `r` is ordered before action `w` in the `hb` order.

[7] The motivating examples can be found in [19]

x == y == 0

Thread 1	Thread 2

r1 := x;
if (r1 != 0)
y := 42;
	r2 := y;
	if (r2 != 0)
	x := 42;

Disallowed: r1 = r2 = 42

(a) *Race Free*

x == y == 0

Thread 1	Thread 2
r1 := x; y := r2;	r2 := y; x := r2;

Disallowed: r1 = r2 = 42

(b) Not Race Free

Fig. 1. Out of Thin Air Examples

The building blocks of the formal definition of the JMM are actions, executions, and a committing procedure to validate actions, which in turn validates complete executions.

Actions Formally, a JMM action is defined as a tuple `<t, k, v, u>` which contains; `t`, the thread identifier of the thread issuing the action; `k` the kind of the action, which can be one of `Read`, `Write`, `Volatile Read`, `Volatile Write`, `Lock`, `Unlock`[8]; `v` the variable involved; and `u` a unique identifier.

Executions With the notion of actions, an execution is defined as a tuple `<P, A, po, so, W, V, sw, hb>` containing: a program `P`; a set of actions `A`; the program order `po`; the synchronisation order `so`; the write seen function, `W`, that for each read action `r` (either normal or volatile) in `A` returns the write action `w` (also in `A`) seen by that read; the value written function, `V`, that for each write `w` (either normal or volatile) in `A` returns the value written `v` by that write; the synchronises-with order, `sw`; and the happens before order, `hb`. Notice that `sw` and `hb` are derived from the `po` and `so` orders.

Well-formedness of Executions The *well-formedness* conditions express basic requirements for valid JMM executions, e.g., *(i)* reads see only writes on the same variable, and volatile reads and writes are issued on volatile variables; *(ii)* `po` and `so` are consistent orders (which guarantees that the `hb` order is a partial order); *(iii)* the restrictions of the HBMM for normal variables as well as for volatile variables apply; *(iv)* `so` respects mutual exclusion, i.e., there is a one at a time semantics for locks; *(v)* the `so` order is of type less or equal to Omega[9]; and *(vi)* executions obey intrathread consistency. This last requirement means that the actions that each thread generates should be those that the singlethreaded semantics of Java (described in the JLS, not taking the JMM into account) would generate, provided that each read `r` of a shared variable returns a value of `V(W(r))`. This is the only link between the singlethreaded semantics and the JMM.

[8] Other secondary actions like thread start are not here.

[9] This means that there are no infinitely decreasing sequences of elements.

Every Java execution should be well-formed and furthermore, in the commitment procedure only well-formed executions are used to justify actions.

Causality Requirements The justification of an execution procedes by committing sets of actions, until every action of the execution is committed. To commit an action we must find an execution that justifies it, obeying certain conditions discussed below. Once an action is committed, it remains committed for the rest of the justification procedure. Thus the justification of an execution is a sequence of pairs of executions and commitment sets, satisfying certain rules. An execution is allowed by the JMM only if such a sequence can be found.

Below we will present the causality rules as stated in [19] with an explanation of our interpretation for each. We will use the restriction notation, both for relations and for functions (domain restriction). Thus for a set S, a relation R, and a function F we define:

- $R|_S = \{(x,y)|x,y \in S \wedge (x,y) \in R\}$ and
- $F|_S(x) = F(x)$ if $x \in S$ and undefined otherwise.

Following [19], we will use C_i to denote the i^{th} commitment set in the sequence, and $E_i = < P, A_i, \overset{po_i}{\rightarrow}, \overset{so_i}{\rightarrow}, W_i, V_i, \overset{sw_i}{\rightarrow}, \overset{hb_i}{\rightarrow} >$ to denote the i^{th} execution. The first pair of the justification sequence contains an empty commitment set, and any execution where every read sees a write that happens before it. The commitment set of the last justification must contains all the actions of the justified execution. It is interesting to note that the last justifying executions needs not to be the same as the execution being justified (this can be seen in rule 5, that only applies to the previous commitment as we shall see). These requirements are captured by the following JMM rules.

- $C_0 = \varnothing$
- $C_i \subset C_{i+1}$
- $A = \bigcup(C_0, C_1, C_2, ...)$

The main idea to prevent the behaviours depicted in Figure 1 is to disallow circular justification of actions. The committing procedure guarantees this by disallowing reads not already committed to see writes that do not happen before them. Moreover, for a read `r` to be able to see a write `w` that is not `hb` related to it, `w` must be committed before `r`, thus, it must be able to happen regardless of whether `r` sees it.

After presenting the rules we show how they disallow the executions in Figure 1.

The first three rules are simple: they require all the committed actions to be present in the justifying execution and the `hb` and `so` orders of the justifying and justified execution to coincide on the committed actions.

1 $C_i \subseteq A_i$

2 $\overset{hb_i}{\rightarrow}|_{C_i} = \overset{hb}{\rightarrow}|_{C_i}$

3 $\overset{so_i}{\rightarrow}|_{C_i} = \overset{so}{\rightarrow}|_{C_i}$

The next rule only refers to write actions (the domain of the `V` function) that are

committed; it simply says that whenever a write is committed, the value written cannot change in future justifying executions.

4 $V_i\ |_{C_i} = V|_{C_i}$

For the purpose of presentation we show rule 6 first, followed by rule 5. The main purpose of the causality requirements is to avoid the self justification of actions as in the example of Figure 1(b). A simple way to achieve this would be to require that all reads see only writes that happen before them, motivated by the fact that `hb` contains no cycles as it is a partial order. Requiring this would preclude circular justification of writes, but unfortunately it would also prevent any kind of data race, which is not the intention. To allow data races in the execution the JMM restricts them to the committed actions. For this purpose the next rule requires all uncommitted reads to see writes that happen before them. Furthermore, reads currently being committed must see writes that happen before them.

6 $\forall r \in Reads(A - C_{i-1}) : W_i(r) \overset{hb_i}{\rightarrow} r$

The following rule refers to reads committed in the previous justification (if there were any). It says that when a read is not yet committed, it can see a write different from the one it sees in the justified execution. Moreover, when a read is being committed it can read a different write, but in the following justification step it must read the write of the justified execution. Recall from the previous rule that reads being committed must see writes that happen before them. If rule 5 was given on the current commitment set C_i, instead of the previous one C_{i-1}, all reads in the final execution should see writes that happen before them, completely disallowing the presence of data races. To avoid this the rule is expressed over reads committed in the previous execution (which could be involved in data races, since they are already committed). In general, to commit data races in the JMM, the sequence starts by committing the writes involved in the data race. Clearly, for any read involved in a data race two writes must be committed as required by rule 7 in conjunction with rule 6; one that is not ordered by `hb` in the final execution (that forms the data race), and one that happens before (required by rule 6). We commit the read seeing the write that happens before it; and then, in the following commitment step we make it see the write that corresponds to the justified execution (i.e., the write involved in the data race) as required by the following rule.

5 $W_i\ |_{C_{i-1}} = W|_{C_{i-1}}$

The following rule states that whenever a read is committed, both the write that it reads in the execution being justified (E) and in the justifying execution (E_i) must be already committed. As mentioned when explaining rule 5, when committing a read action `r`, it can see a different value in the justifying execution than what it sees in the execution justified. But, to guarantee that the write that `r` actually sees does not depend on the return value of `r`, we must show that that write can be committed without assuming that `r` saw it. Therefore, part (b) of this rule requires the write that `r` finally sees to be already committed. Furthermore, part (a) requires the write that `r` sees while being committed (and which happens before it, by rule 6) to be also committed. This guarantees that the read happens independently from the write it eventually sees.

7 (a) $\forall r \in Reads(C_i - C_{i-1}) : W_i(r) \in C_{i-1}$
(b) $\forall r \in Reads(C_i - C_{i-1}) : W(r) \in C_{i-1}$

Rules 2, 5, 6, and 7 are the most important rules to dissallow circular justifications of actions. Rule 7 forces that whenever a read is committed, the writes it sees in the justifications and the justified execution must already be committed, and rule 6 mandates that non-committed reads can only see writes that happen before them. Thus, this prevents circular read-write sequences in justifying executions. Rule 2 restricts `hb` links that have been committed to remain committed (and to be coherent with the justified execution), avoiding possible circularities (since `hb` is a partial order). Finally rule 5 allows read-write data races to be justified, provided these are not circularly justified.

The next rule only constraints synchronisation actions. The basic idea for this rule is that once synchronisation links have been used to commit actions those links must (transitively) remain. The `ssw` links are defined to be the `sw` links present in the transitive reduction of the `hb` order that are not in `po`. The intuition behind these links is that they extend the `hb` relation across threads. Following rule 6, a read `r` is allowed to see a write `w` that happens before it without being committed. But if after committing some actions that depend on such a read, the `sw` link that extended the `hb` relation among them was not required to be present in following executions, the `hb` link could also disappear. Therefore the actions committed based on this assumption would not be correct anymore. A more operational intuition is that if an action is justified, assuming that a synchronisation of the memory was issued between two threads, then that synchronisation should be performed [10].

8 $\forall x, y \in A_i, z \in (C_i - C_{i-1}) : x \overset{ssw_i}{\rightarrow} y \overset{hb_i}{\rightarrow} z \Rightarrow (\forall j \geq i : x \overset{sw_j}{\rightarrow} y)$

The last rule requires *external* actions, which are actions observable outside of an execution, to be committed before any action that happens after them. Typical examples of external actions are printing values, or communications with other processes. External actions are used to define the observable behaviour of executions. In turn this is used to define the allowed optimisations: optimisations are only allowed if the observable behaviour of the program is the same.

9 $\forall x, y \in A_i : External(x) \wedge y \in C_i \wedge x \overset{hb_i}{\rightarrow} y \Rightarrow x \in C_i$

Let us see now how the rules prevent the executions discussed in Figure 1. For the first example (Figure 1(a)) in order to get both reads to see the writes of `42` in the other thread, we need to have both writes performed (i.e., committed). Moreover, by rule 7, they have to be committed before any of the reads is allowed to see them in a committing step. So we know that apart from the default writes to `x` and `y`, the first actions that have to be committed are the writes of `42`. In addition, following rule 6, the reads in both threads are only allowed to see default writes (the only writes that happen before them in the execution). Now, we argue that it is impossible to find a justifying execution where both reads see the default writes and any of the writes of `42` happen, as this would be a violation of the intrathread

[10] Note that with this description we are trying to explain the motivation of the rule as we understand it. This does not mean that we think that the rule achieves it purpose as stated. In fact, we think that the statement of the rule is not correct, though the intuition behind it should be as described.

consistency requirement of well-formed executions. Therefore there is no execution that makes it possible to commit a write of `42`, and thus the result is prevented. For the second example (in Figure 1(b)) the reasoning is quite similar. The writes must be committed first, and there is no intra-thread consistent execution where the reads see writes that happen before them, that allows a write of `42` to appear. Therefore, the result is forbidden.

2.2 Some Problems of the Java Memory Model

During our study and formalisation of the JMM we and others realised that there are several problems with its definition. This section summarises the most important issues.

First of all, the formal definition of the model is hard to understand and to use. This is partly due to the lack of motivation for the rules that validate the executions, and partly to the non-constructiveness of the committing procedure itself. The latter makes it hard to reason about the possible interactions between the rules. The lack of motivation for the rules is reflected by the fact that there is no formal definition of the out-of-thin-air guarantee, and no proofs are given to show that the rules actually enforce it.

On a more concrete side, some inconsistencies have been found; we will briefly mention some of these. First, Cenciarelli et al. [6] found a counter example to the claim in Theorem 1 in [19], which says that every two adjacent instructions that are independent can be reordered. In addition, Aspinall and Sevcik [4] have pointed out that the Omega type restriction to the `so` order prevents executions with infinitely many volatile instance variables, since the initialisation of all of these must come before the first synchronisation action of any thread. Thus infinitely many actions have to be ordered before the first synchronisation action of any thread, breaking the Omega type restriction. As we shall see, this has an impact on our formalisation.

With respect to the OoTA property, we still find it hard to argue whether some reads constitute an OoTA violation or not. Furthermore, Aspinall and Sevcik [4] have shown that many of the executions required to be accepted by the JMM [8] are not allowed. We think that the definition of OoTA needs to be clarified and revised.

3 Formalisation

The issues exposed above indicate that there is a real need for a formalisation of the JMM. In addition, another motivation for this formalisation emerges as a requirement for the Mobius project. The Mobius project aims at developing a proof carrying code (PCC) infrastructure for the specification and verification of mobile Java programs. For this purpose, program logics for Java source and bytecode are developed within the project. This bytecode logic is proven correct w.r.t. Bicolano, a Coq formalisation of the Java bytecode operational semantics [20]. We have extended the Bicolano semantics to an interleaving semantics, called BicolanoMT, augmented with multithreaded instructions. This semantics will be used to prove the correctness of the Mobius program logic for correctly synchronised multithreaded

Java programs. To motivate that is sufficient to prove soundness w.r.t. an interleaving semantics we plan to prove that the BicolanoMT semantics is in a one-to-one correspondence with the executions allowed by the JMM for correctly synchronised programs. This motivates a formally mechanised specification of the JMM and in particular a proof of the DRF guarantee.

As expected, some of the problems mentioned above appeared while formalising the model and influenced our formalisation. We will present a general overview of the model and we will explain the most important differences between our formalisation and the official JMM definitions [19]. For our formalisation we used the Coq proof assistant, which will allow us to integrate our model with Bicolano.

3.1 Brief Description of the Formalisation

To be able to prove the DRF guarantee mechanically we formalised the JMM as close as possible to its definition [19]. With this we expected to avoid problems derived from misunderstandings or biased interpretations. We did not have to formalise the whole JMM specification to prove the DRF guarantee. Therefore, we limited our formalisation to the needed definitions, except for the causality rules, that are completely formalised. However, some of them are currently not used in any proof. The kind of actions we formalised only contain volatile and normal reads and writes, lock and unlock actions; currently we do not include thread start, thread termination and other actions that are not important to prove the DRF guarantee. However, we believe that these could be easily added without changing the current proofs, as we plan to do.

Further, we have defined programs, values, variables, thread identifiers and unique identifiers as abstract data types, and we have axiomatised their interactions (e.g., two different actions have different unique identifiers). The data types for actions and executions are axiomatised using the Coq module system, in a way that is very similar to their description in the JMM paper, except for little technical details (e.g., the `sw` order is not part of the execution data type, since it can be derived from the `po` and `so` orders, which are parameters of the execution).

An interesting challenge is the specification of the commitment procedure. For this we used an abstract function that, given a program, an execution and a proof that the execution belongs to the allowed executions for that program, returns a list containing the sequence of pairs of commitment set and commitment execution, that justify the execution.

The use of Coq lists to represent the commitment procedure limits it to be finite, but in the JMM definition justifications could be infinite. This choice was made only to simplify the first version of the proofs as we can use induction to prove facts about finite lists. To cover the full semantics of the JMM we plan to replace current lists for streams which allow for infinite sequences, and where the proofs given by induction must be replaced for proofs by coinduction. We expect that this will not have any significant impact on the current proofs. In the rest of the formalisation, we did not assume finiteness.

The (causality) rules are axiomatised using a justifying list, as discussed above. To give a feeling what the formalisation looks like, here is the formalisation rule 2

of the causality requirements in [19].

```
Definition req2 :=
   ∀ E_exec_P j x y, In (justification E_exec_P) j →
         eIn (comm j) x → eIn (comm j) y →
         hb E x y ↔ hb (exec j) x y.
```

In this definition the function `justification` is the abstract function that returns the list containing the pairs of commitment set and committing execution. The bound variable `j` stands for a committing pair, and the predicate `In (justification E_exec_P) j` states that `j` is in the committing sequence of the execution `E`. The functions `comm` and `exec` extract from a committing pair the corresponding set and execution, respectively. Thus, the rule says that if both actions `x` and `y` are committed in `j`, they are related by `hb` identically in the justifying execution of `j` and in the justified execution.

In the following we present the statement of the main theorems and the assumptions we needed for their proofs. For each assumption, we justify how we plan to prove them in future versions of the formalisation.

The DRF proof sketched in [19] is separated in a lemma and the main theorem. We have followed their reasoning as close as possible. For both proofs we assume we have a correctly synchronised program P and an execution `E` of the program `P`. That is stated in our Coq formalisation as follows:

```
Hypothesis p_well_synch:  correctly_synchronized P.
Hypothesis E_exec_P: Program_Executions P E.
```

The definition of the `correctly_synchronized` predicate states formally that all sequentially consistent executions of the program `P` are free of data races. The `Program_Executions` predicate states that `E` belongs to the set of allowed executions for `P`, which is axiomatically defined in the model.

The lemma (Lemma 2 in [19]) takes as hypothesis that read actions only see writes that happen before them. This hypothesis is later proved in the theorem, which concludes the DRF proof using the lemma. This hypothesis is stated in our formalisation by the following definition.

```
Definition reads_see_hb (E:Exec):
   ∀ r act_r r_read, hb E (W E r act_r r_read) r.
```

To understand the definition above it is important to note that the `W` function takes four parameters. The first parameter `E` is the execution, the second parameter `r` is the read action whose write we are trying to obtain; the other two parameters are related to our choice for modelling partial functions in Coq. Our approach was that of "preconditions for partial functions" as in [5], where a proof must be given showing that the argument belongs to the domain of the function to apply it. In this particular case, the parameters `act_r` and `r_read` are proofs that the action `r` is an action of the execution `E` and that `r` is a read action, respectively.

Now we focus on additional assumptions we had to make, as these where left implicit in the original proofs of [19]. We expect to be able to prove each of these assumptions we added, and we will explain how to do it in each case. Both proofs of the lemma and the main theorem go by absurdity. Assuming that there is some

non sequentially consistent execution of the correctly synchronised program P, a sequentially consistent execution that contains a data race can be constructed, contradicting the correct synchronisation hypothesis of P. To show a sequentially consistent execution with a data race a topological sort of the hb order is used, which is described in our formalisation as:

```
Definition to := top_sort (hb E) (hb_partial_order E).
```

The top_sort predicate is the axiomatisation of the particular sort needed. With to in hand, the proof of the lemma proceeds by identifying the first occurrence of an intromission of a write w between a read r and the write that it sees (W E r act_r r_read) in that to order. To guarantee that such a read exists we need to assume that to is well founded, and then the proof goes by well founded induction. This is not proved in the JMM paper. This is in fact true if we prove that the po is well founded and we assume that so has type Omega or less (as we said before, default actions should be revised to do guarantee this for so). The proof of this is not currently formalised, but it should be proved using these hypothesis. We plan to formalise it in following versions of our model. A simpler approach could be that of Aspinall and Sevcik [4], where they assume that the executions are finite, where well foundedness of to is trivial (every finite order is well founded).

```
Hypothesis wf_to:  well_founded to.
```

Finally, one problem that we encountered formalising the JMM is that the papers proof assumes that for any sequentially consistent "prefix" of an execution, there must exist a sequentially consistent continuation for it. This fact is not entirely true, since new default writes of variables could be needed (or discarded) as we take a different branch of the program. As default writes synchronise with the first action of any thread, actions should be added at the beginning (before any thread start action) in the to. To overcome this problem some revision on the definition of default write actions should be proposed. There are several alternatives to fix this problem; e.g., developing a different mechanism for initialising variables, or a different formulation of the sequential continuation property (e.g., subsegment starting from the first thread start action). We postponed this decision for later revisions of the formalisation and assume it correct as currently stated in the JMM.

With this assumption we have to show that there is always a sequential consistent continuation. To exhibit such continuation an interleaving of the instructions of the threads from the program point where they last executed should complete a sequential consistent continuation of the execution. The difficulty appears when giving such continuations at the level of the JMM formalisation, where nothing is said about instructions and programs, but only about actions. For this reason we added that fact as a hypothesis and expect to prove it when we establish a proper link between programs and executions for the JMM.

```
Hypothesis seq_continuation:
   Program_Executions P E → first_w_r_intromission to r w →
   ∃ E', Program_Executions P E' ∧
         eq_exec_complete_upto lower_r E E' ∧
         (∀ acr_r', W E' r act_r' r_read = w) ∧
```

```
sequential_consistent E'.
```

The intuition of the definition above is that; given an execution `E` and the first occurrence of a read write race, such that the write `w` is the last write in `to` to the same variable as the read `r` ordered before `r` in to, and such that `r` does not see that write; then, there should be another execution of `P`, `E'` where the read `r` sees the last write in the `to`, `w`, and that `E'` is equal to `E` when restricted to the actions that come before `r` in `to`. The details of the definition are not simple, but the intuition is; it only says that there is a sequential consistent continuation of the prefix.

We expect to be able to prove this very intuitive result, but a link between the program instructions and the actual actions should be developed first. Several authors [6,4,22] agree that there is no good link between the semantics of Java as defined in the JLS [12] (disregarding the JMM itself) and the JMM (as in JSR-133 [14]). Currently this link is given at a very high level by the requirement of intra-thread consistency of the well-formedness requirements. Different approaches have been taken, which include an approach towards operational semantics proposed by Cenciarelli et al. [6], and a function that verifies the validity the semantics of each thread, proposed by Aspinall and Sevcik [4]. We plan to make this link for correctly synchronised programs only, using the BicolanoMT interleaving semantics.

With all hypotheses mentioned in context, the Coq formalisation of the theorems from the JMM paper are simple.

```
Theorem Lemma2:  ∀ E, reads_see_hb E → sequential_consistent E.
Theorem DRF: sequential_consistent E.
```

Summing up, to finish our formalisation, a proof that the `to` has order type Omega or less should be developed; and a link between actions and instructions should be given for correctly synchronised programs. For these proofs the initialisation of actions need to be fixed. We plan to do this in a following version.

The current version of the model occupies 3000 lines in Coq, and has been of great use to help understanding the model, as well as to identify problems in the definitions and underspecified assumptions. It is in our plans to use the formalisation to further explore facts about the JMM, for example the OoTA property.

4 Conclusions & Future Work

In this paper we have presented a formalisation of the Java Memory Model, using the Coq theorem prover, and we have formally proved the DRF guarantee, as claimed by the authors of the JMM. Because of the different requirements for the JMM (allowing many compiler optimisations, avoiding security problems caused by out-of-thin-air values, and easy programmability), the formal definition is fairly complex, and needs a formalisation to reason about it. Also, as part of the formalisation process, we have gained a good insight in the motivation of the rules that define the validity of executions, and we have tried to convey our insights, hopefully being more intuitive than the description of the rules as presented in [19].

In the near future, we plan to exploit the DRF guarantee, by linking the JMM formalisation with BicolanoMT. We will define a mapping from BicolanoMT programs to the program model, as used for the JMM. Then we will prove that programs

that are correctly synchronised in BicolanoMT semantics, are also correctly synchronised according to JMM semantics. Moreover, for correctly synchronised programs, BicolanoMT executions are in a one to one relationship with JMM executions.

We also plan to study the OoTA guarantee more precisely. The authors of the JMM claim that it does not allow any out-of-thin-air values to be produced, but they do not give a formal definition. We would like to give a more formal definition for this property, and then verify whether the JMM actually respects it.

In the more long term, we also would like to study modularity of the JMM: if a part of a program is data race free, can the behaviour of that part then described with an interleaving semantics. This property is essential to support compositional verification. We are also interested in studying whether there are so-called benign data races, i.e., data races that still result in sequentially consistent executions. Finally, we are also interested in proving formally that a compiler optimisation is allowed by the JMM, by showing that the program transformation does not change the set of legal executions.

Acknowledgement

We would like to thank Jaroslav Sevcik and David Aspinall whose discussions helped to improve our understanding of the model. We also thank Clement Hurlin and the reviewers that contributed to the presentation of this work.

References

[1] S. V. Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 1993.
[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
[3] S.V. Adve and M.D. Hill. Weak ordering - a new definition. In *International Symposium on Computer Architecture*, pages 2–14, 1990.
[4] D. Aspinall and J. Ševčík. Formalising Java's data-race-free guarantee. Technical report, LFCS, School of Informatics, University of Edimburgh, 2007. To appear.
[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
[6] P. Cenciarelli, A. Knapp, and E. Sibilo. The Java memory model: Operationally, denotationally, axiomatically. *European Symposium on Programming*, 2007.
[7] Mobius Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from http://mobius.inria.fr, 2006.
[8] The JSR 133 Consortium. The Java memory model causality test cases, 2004. http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html.
[9] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, mars 2004. http://coq.inria.fr/doc/main.html.
[10] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.
[11] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories (extended abstract). In *ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, 1991.
[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, third edition*. The Java Series. Addison-Wesley Publishing Company, 2005.
[13] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part I: Definitions and comparisons. Technical Report 98/612/03, The University of Calgary, 1998.
[14] JSR-133: Java Memory Model and Thread Specification, 2004.
[15] J. Kawash. *Limitations and capabilities of weak memory consistency systems*. PhD thesis, The University of Calgary, 2000.
[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Computer*, 28(9):690–691, 1979.
[18] J. Manson. *The Java Memory Model*. PhD thesis, Faculty of the Graduate School of the University of Maryland, 2004.

[19] J. Manson, W. Pugh, and S.V. Adve. The Java memory model. In *Principles of Programming Languages*, pages 378–391, 2005.
[20] D. Pichardie. Bicolano – Byte Code Language in Coq. http://mobius.inia.fr/bicolano. Summary appears in [7], 2006.
[21] William Pugh. The java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.
[22] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–172, 2007.