

References

- [1] The Millipede Project Home Page. <http://www.cs.technion.ac.il/Labs/Millipede>.
- [2] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures (SPAA'93)*, 1993.
- [3] H. Attiya and J.L. Welch. Sequential Consistency versus Linearizability. In *ACM Transactions on Computer Systems*, May 1994.
- [4] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *COMPCON*, pages 528–537. IEEE, 1993.
- [5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th Int'l ACM Symp. on Computer Architecture*, pages 15–26, 1990.
- [6] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.
- [7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [8] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [9] P. W. Hutto and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In *Proc. of the 10th Int'l Conf. on Distributed Computing Systems (ICDCS-10)*, pages 302–311, May 1990.
- [10] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: a Bridge between Release Consistency and Entry Consistency. In *Proc. 8th ACM Symp. on Parallel Algorithms and Architectures*. ACM, June 1996.
- [11] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [12] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1996.
- [13] S.V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, pages 66–76, December 1996.
- [14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
- [15] A.S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

A.3 Constraints inside a thread

1. A **use** or **assign** of V is permitted only when dictated by the execution of T .
2. A **store** of V by T must intervene between an **assign** of V by T and a subsequent **load** of V by T .
3. An **assign** of V by T must intervene between a **load** or a **store** of V by T and a subsequent **store** of V by T .
4. A variable is said to be *new* when it is used by a thread for the first time, or when it is created by a thread. For each new variable, **assign** or **load** must be performed on it before any **use** or **store**.

A.4 Constraints between a thread and the main memory

1. Each **lock** and **unlock** is performed jointly by some thread and the main memory.
2. For every **load** performed by T on its working copy of V , there should be a corresponding preceding **read** by the main memory on the master copy of V .
3. For every **store** performed by T on its working copy of V , there must be a corresponding following **write** by the main memory.
4. Let A be a **load** or a **store** of V by T , and let P be the corresponding **read** or **write** by the main memory. Let B and Q be two other such operations by T and the main memory (on V), correspondingly. Now, if A precedes B , then P precedes Q .

A.5 Locks

1. A **lock** of L by T may occur only if for every other thread the number of preceding **unlocks** equals the number of preceding **locks**.
2. An **unlock** of L by T may occur only if the number of preceding **unlocks** of L by T is strictly less than the number of preceding **locks**.
3. **Locks** and **unlocks** of L are performed in some sequential order which is consistent with the program order of all the threads.
4. A **store** must intervene between an **assign** of V by T and a subsequent **unlock** of L by T , and the **write** which corresponds to the **store** must occur before the **unlock** by the main memory.
5. Between a **lock** of L by T and a subsequent **use** or **store** of V by T , an **assign** or **load** of V must appear. If what appears is a **load** then its corresponding **write** should appear before the **lock** by the main memory.

A.6 Volatile variables

1. A **use** of V by T is permitted only if the previous access to V by T was a **load**, and a **load** is permitted only if the next access to V by T is a **use**.
2. A **store** of V by T is permitted only if the previous access to V by T was an **assign**, and an **assign** is permitted only if the next access to V by T is a **store**.
3. Let A denote a **use** or an **assign** of V by T , and let F denote the corresponding **load** or **store**, and P denote the **read** or **write** corresponding to F . Similarly, let B denote a **use** or an **assign** of W by T , and let G denote the corresponding **load** or **store**, and Q denote the **read** or **write** corresponding to G . Then if A precedes B , then P must precede Q .

A The Java Memory Model Specification

For completeness we give in this section the operational Java memory model specification as defined in [7] (JLS). The specification consists of two parts: the set of operations in the model, and the set of constraints on those operations.

For simplicity, in the specifications below V and W always denote variables, T always denotes a thread, and L always denotes a lock.

A.1 Operations

A single Java thread issues a stream of **use**, **assign**, **lock** and **unlock** instructions, according to the program source code. The underlying Java implementation is then required to perform appropriate **load**, **store**, **read** and **write** instructions, according to the Java constraints. The semantics of these operations are as follows:

- A **use** operation transfers the value of the variable from the thread local copy into the execution engine.
- An **assign** operation transfers the value of the variable from the execution engine into the local copy.
- A **load** operation transfers the value of the variable from the main memory (which was read by the preceding **read** operation) to the local copy.
- A **store** operation transfers the value of the variable stored in the local copy of the thread to the main memory, to be written to the master copy of the variable by the **write** operation.
- A **read** operation transmits the value of the master copy of the variable to the thread for the use of a **load** operation.
- A **write** operation writes the value of the variable transferred by the **store** operation to the master copy in the main memory.

A.2 General constraints

1. The operations performed by any one thread are totally ordered. A **use** x in the order of a thread always uses the most recent value that was given to x by an **assign** or **load** operation in that order.¹
2. The operations performed by the main memory for any one variable are totally ordered. A **read** in the order of one of the variables always yields the value that was written by the last **write** in that order.² If there was no preceding **write** in the order then the value yielded by the **read** is some initialization value.
3. The operations performed by the main memory for any one lock are totally ordered.
4. It is not permitted for an instruction to follow itself.

¹The “register property” here follows implicitly from JLS [7].

²The semantics of the main memory, implying that a **read** from a master copy of a variable always returns the value stored there by the most recent **write** to the same variable, follows from several remarks in the JLS [7].

3.8 Volatile variables

As with regular variables, the JVM executes the bytecodes with no indication regarding the original order of operations in the source code. Thus the JVM must assume that the compiler preserves the order of `loads` and `stores`, and `uses` and `assigns`, according to the JLS constraints. Now, because for volatile variables the order of `uses` and `assigns` is equal to the order of `loads` and `stores` performed by the processor, the same reasoning as in Section 2.5 hold, and Volatile Consistency in Java' is equal to SC.

Note that the bytecodes do not contain the information which variable is volatile and which is not. This information is contained in the class headers, and can be accessed during the class loading time. Therefore, it is reasonable that the volatile variables and the regular ones will be mapped to different memory segments. This may ease the implementation in some cases, but should be done with care when the same object contains both volatile and regular variables.

3.9 Locks

Once again, in order to preserve the locks semantics, the JVM depends on the compiler. For `write` operations to terminate before the `unlock`, the corresponding `store` instructions must be placed by the compiler before the `unlock` instruction. The same holds for the `reads` and their corresponding `stores` and `locks`.

The requirements from the implementation are then, that if there is a `lock` instruction followed by a `load` instruction, then the main memory should preserve the same order of their corresponding `lock` and `read`, and if there is a `store` instruction followed by an `unlock` instruction, then their corresponding operations in the main memory should be performed in the same order.

Because no other instruction can bypass a `lock` or `unlock`, the order of `locks` and `unlocks` performed by the processor is compatible with its program order. As before, the tight coupling of the `lock/unlock` operations performed by the processors and the corresponding `lock/unlock` performed by main memory imply Linearizability. Note this does not necessarily require having a single memory agent: the needed consistency can be achieved by applying appropriate protocols between multiple memory agents.

4 Conclusions

In this work we gave non-operational characterizations of the Java memory behavior. In order for his program to work correctly on all implementations that comply with the standard Java definition, the Java programmer relies on memory behavior which is not stronger than the standard definition as given in the JLS [7]. We show that the programmer can rely on Coherence and Causality for regular variables, on Sequential Consistency for Volatile variables, and on at least Release Consistency when programming with the *synchronized* construct.

On the other hand, in order for his implementation to support all programs that comply with the standard Java specifications, the implementor of the Java Virtual Machine must make sure it is at least as strong as the original definition. We provide both the implementor and the programmer with two non-operational definitions that are a lot simpler to understand than the original, operational one, and can be implemented more freely.

We mention that this work was done in the scope of the Millipede project [1] with the goal of having Java implemented distributively in a correct and efficient way.

Let's examine the rules of Data Dependency:

1. A LOAD x, v that sees the result of a STORE x, v : Because JavaN presents a legal serialization, the STORE must precede the LOAD.
2. *Causality Relation*: By the definition of JavaN, if two operations are Causally related, then they appear in the same order in the serialization.
3. *The triangle rule of Data Dependency*: LOAD x, v o_2 that sees the result of a STORE x, v o_1 , while there is another STORE x, v o_3 that is Data Dependent on o_1 . Because of legal serialization, in order for o_2 to see the result of o_1 , o_3 cannot appear between o_1 and o_2 . It remains to show that o_3 cannot appear before o_1 . Suppose on the contrary that it does.

Let us call a *basic application of Data Dependency* a dependency which is obtained by one of the rules of Data Dependency which is not the rule of transitivity.

If o_3 precedes o_1 then there is a sequence of basic applications $o_1 = o'_1 \xrightarrow{DD} o'_2, o'_2 \xrightarrow{DD} o'_3, \dots, o'_{n-1} \xrightarrow{DD} o'_n = o_3$, where in at least one of the dependencies o'_{i-1} precedes o'_i . By the cases 1 and 2 above, the only basic application where this can happen is the triangle rule of Data Dependency: LOAD y, w o' that sees the result of a STORE y, w o'_i , while there is another STORE y, w o'_{i-1} that is Data Dependent on o'_i .

We thus arrived at the same situation from which we started, and can proceed with the same process over and over again. By the cases 1 and 2 above, this process cannot terminate. It can either form a loop of violated triangle rule applications or an infinite sequence of such. However, both cases are impossible as the Data Dependency rule is defined inductively, and thus such a sequence must terminate after a finite number of applications.

Direction 2. *JavaD is not weaker than JavaN.* Let H be a valid JavaD execution, we show that it is valid also under JavaN.

We must show that if H contains no cycles of Data Dependencies, then there is legal serialization that is consistent with the Causal Relation.

If there is no cycle of Data Dependencies, then a serialization which preserves the dependencies can be obtained, e.g., by applying a topological sort. From all possible serializations we choose the following: Given a STORE x, v w_1 and a LOAD x, v r_1 which sees the result of w_1 , and another STORE x, v w_2 to the same variable, then w_2 is placed after r_1 if there is Data Dependency from w_1 to w_2 , and before w_1 otherwise.

We need to show that such serialization exists in the partial topological order, and it is legal.

Assume that the required serialization does not exist among the possible topological orders. We use the same notation as above, and let's look at some w_2 that cannot be placed according to the requirements. There are two cases:

1. w_2 is Data Dependent on w_1 . w_1 is seen by r_1 , but w_2 cannot be placed after r_1 . This implies that there is a Data Dependency from w_2 to r_1 . However, $w_1 \xrightarrow{DD} w_2$ and $w_1 \xrightarrow{DD} r_1$ (the latter because r_1 sees the result of w_1), so by the rules of Data Dependency $w_2 \xrightarrow{DD} r_1$. This implies cycle of Data Dependencies $w_2 \xrightarrow{DD} r_1 \xrightarrow{DD} w_2$ — a contradiction.
2. There is no Data Dependency from w_1 to w_2 , but w_2 cannot be placed before w_1 . This implies that w_2 is Data Dependent on w_1 — a contradiction.

It remains to show that the elected serialization is legal. A LOAD x, v that sees the result of a STORE x, v is Data Dependent on it, and thus appears after the STORE in the serialization. Now, because of the choice of serialization, any other STORE x, v to the same variable would be placed either before our STORE x, v or after the LOAD x, v , thus the resulting serialization is legal. ■

3.6.1 JavaD vs. Processor Consistency (Revisited)

As mentioned in Section 2.3 (see there for precise definitions), Ahamad et. al. [2] defined two variants of PC, namely PCG and PCD. Since we show below that JavaD is equal to Java, we conclude that JavaD is incomparable to both PCG and PCD. Yet, in order to show the usage of the “constructive” JavaD definition we give here examples that can be shown to be JavaD and are known not to be PC (one for each variant).

	<i>Processor 1</i>	<i>Processor 2</i>	<i>Processor 3</i>	<i>Processor 4</i>
1	W X, 0		W Z, 0	
2	W X, 1		W Z, 1	
3	W Y, 1	R Y, 1	W V, 1	R V, 1
4		R Z, 0		R X, 0

Figure 9: An execution which is invalid for PCD (taken from [2]) and is valid for JavaD and JavaN.

Figure 9 gives an execution which is shown in [2] to be invalid for PCD. However, it is valid in JavaD, as we show by computing the Data Dependency relations. There are four relations of the first type: $1.1 \xrightarrow{DD} 4.4$, $3.1 \xrightarrow{DD} 2.4$, $1.3 \xrightarrow{DD} 2.3$, and $3.3 \xrightarrow{DD} 4.3$. There are two relations of the second type: $1.1 \xrightarrow{DD} 1.2$ and $3.1 \xrightarrow{DD} 3.2$. There are two relations of the third type (the triangle rule): $4.4 \xrightarrow{DD} 1.2$ and $2.4 \xrightarrow{DD} 3.2$. Hence there are no Data Dependency relations cycles and thus the execution is JavaD.

	<i>Processor 1</i>	<i>Processor 2</i>
1	W X, 0	W Y, 0
2	W X, 1	W Y, 1
3	W Z, 0	W Z, 1
4	R Y, 0	R X, 0

Figure 10: An execution which is invalid for PCG (taken from [2]) and is valid for JavaD and JavaN.

Figure 10 presents an execution which is shown in [2] to be invalid for PCG. However, it is valid for JavaD, as follows. There are two relations of the first type: $1.1 \xrightarrow{DD} 2.4$ and $2.1 \xrightarrow{DD} 1.4$. There are two relations of the second type: $1.1 \xrightarrow{DD} 1.2$ and $2.1 \xrightarrow{DD} 2.2$. There are two relations of the third type: $1.4 \xrightarrow{DD} 2.2$ and $2.4 \xrightarrow{DD} 1.2$. There are no Data Dependency cycles, hence the execution is JavaD.

3.7 JavaN is equivalent to JavaD

Theorem 8 *JavaN is equivalent to JavaD.*

Proof Direction 1. *JavaN is not weaker than JavaD.*

Let H be a valid JavaN execution, we show that it is valid also under JavaD. We need to show that all the Data Dependency constraints are preserved in the serialization, and thus conclude that there is no cycle of Data Dependencies.

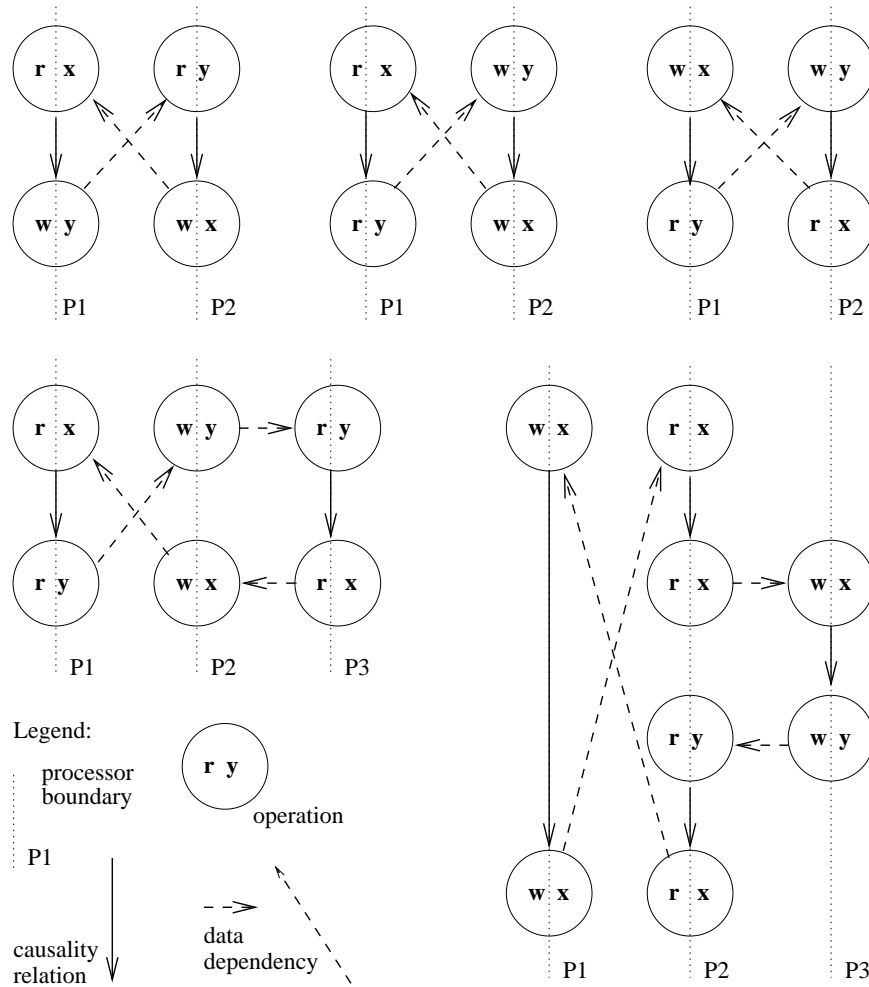


Figure 8: Some examples of Data Dependency cycles that are forbidden in JavaD.

Data Dependency: There are three types of Data Dependency. In the first, a LOAD x,v yields a value written by a WRITE x,v operation, and thus has a Data Dependency on that WRITE. Let the Causality Relation be a Data Dependency of the second type. (Note that if a dependency of the first type occurs between operations that belong to the same local history then this dependency also belongs to the second type). In the third type, it can be inferred from the execution that a LOAD yields an old value, and thus occurs before a WRITE. This is formalized as follows. We say that operation o_1 has a Data Dependency of the third type on operation o_2 when there is a sequence of Data Dependencies (of any type) leading from o_2 to o_1 , or when the following situation occurs: o_2 is a LOAD x,v which has a Data Dependency of the first type on a WRITE x,v , and o_1 is also a WRITE x,v' which is Data Dependent (second or third types) on WRITE x,v .

The definition of JavaD is as follows:

JavaD: Execution is JavaD if it does not contain a (directed) cycle of Data Dependencies.

Figure 8 shows some examples of Data Dependency cycles that are forbidden in JavaD.

construction, the `use` operations are inserted to \hat{S} with the same order as the loads, and since we assumed the order of the `uses` is switched we get a contradiction.

- Suppose the corresponding operations are `load x,v` and `assign x,w`, so that `use x,v` precedes `use x,w` in L , and `use x,v` succeeds `use x,w` in S . Since the `uses` are inserted in \hat{S} immediately after `load/assigns`, we conclude that in \hat{S} `assign x,w` precedes `load x,v`.

On the other hand, since `use x,v` precedes `use x,w` in L , by Constraint A.3.1 `load x,v` precedes `assign x,w` in T . Since `assign x,w` is inserted to \hat{S} in Stage 1 of the construction, we know that in \hat{S} there is either a `store x,w` or a `store x,w'` for which the corresponding `assign x,w'` succeeds the `assign x,w`. In any case, tracing the `store` back to T we conclude that it appears there after the `assign x,w`, and hence after the `load x,v`.

Clearly, in T there is a Causal Relation from the `load x,v` to the `store`, which is preserved in S' and \hat{S} , hence the `load x,v` precedes the `store` in \hat{S} . Now, in the construction of \hat{S} the `assign x,w` is inserted immediately before the `store` (or in a sequence of `assigns` which is inserted consecutively immediately before the `store`), hence it succeeds the `load x,v` in \hat{S} , which is a contradiction.

- Similarly, if the corresponding operations are `assign` and a `load`, with the `use` corresponding to the `assign` preceding that corresponding to the `load`, this implies on the one hand that the `assign` precedes the `load` in \hat{S} . On the other hand we find a matching `store` for the `assign` in \hat{S} , and then tracing it back to T and using Constraint A.3.3 we get that the `store` would appear before the `load` in T , and because of Causal Relation also in \hat{S} , hence by the construction the `assign` will also precede the `load`, a contradiction.

- *The first violating pair is an `assign x`, denoted a , which precedes a `use x`, denoted u , in L .*

If u was inserted because of a (they refer to the same value), then by the construction it would appear after a in \hat{S} , a contradiction.

If u was inserted because of some other `assign`, denoted a' , then for u to yield the value assigned to x by a' , a' must appear in L between a and u . We already saw that the `assigns` keep their program order in S , and since u is inserted after a' we conclude that u succeeds a in S , a contradiction.

If u was inserted because of some `load`, denoted l , then l must appear between a and u in T (Constraint A.3.1). But then, by Constraint A.3.2, a `store` must intervene between a and l . Because the `store` has Causal Relation to l it would precede l in \hat{S} as well. Therefore, by the construction a (that is inserted somewhere before the `store`) would precede u (that is inserted immediately after l) in \hat{S} . ■

3.6 A “Constructive” Non-Operational Definition: JavaD

We proceed to give a second non operational definition of Java' which we call JavaD. In contrast to JavaN, the decision whether a given execution is in JavaD is a simple, constructive process. Thus, the equivalence of Java', JavaN and JavaD, implies an “automatic” efficient verifying mechanism for the compliance of a given execution with the Java' constraints.

JavaD definition uses the notion of *Data Dependency*, denoted \xrightarrow{DD} . We are interested in the question: which LOADS see the results of which WRITES. We thus assume that the value yielded by a LOAD can identify a WRITE, e.g., by assuming that all WRITES assign different values.

Assume the contrary, i.e., that the order in S of two **use/assign** instructions that are causally related in L , is switched. We call these instructions a *violating pair*.

Let's look at the first violating pair, i.e., the violating pair with the first of its instructions in S preceding all other instructions in any other violating pair. We claim that we can always choose the first pair so that the corresponding application of the Causality Relation is *basic*, namely, it is not by transitivity. The reason is that any application of transitivity and a corresponding violating pair p involves a sequence of basic applications, of which at least one is violating and is as early as p .

Note that the instructions inserted to \hat{S} in Stage 3 of the construction keep their original program order from L , hence they cannot violate the Causality Relation. Thus, none of the instructions in the violating pair could be inserted at that stage.

Let's check the rules of the Causality Relation for the other possibilities.

- *The first violating pair involves a use x , denoted u , which precedes an assign, denoted a (not necessarily referring to x) in L .* Suppose u is inserted to \hat{S} after a . Thus u cannot possibly correspond to a (same variable, same value), otherwise, since S is a legal serialization u must precede a in S . There are two possible sub-cases:

- u was inserted to \hat{S} according to some other **assign** or **load** instruction l , and so according to the construction it is inserted right after l , where a is inserted just before its corresponding **store** instruction s . Thus, in order for a to precede u , a must also precede l .

If l is an **assign x** then $l \stackrel{c}{\rightarrow} u$, and $u \stackrel{c}{\rightarrow} a$ (by assumption), thus by transitivity $l \stackrel{c}{\rightarrow} a$. We get another violation of the Causality Relation: between a and l . The **assign** instruction in this violation, l , precedes a , which contradicts the assumption that the first such violation was between u and a .

- If l is a **load x** then since u precedes a in L , by A.2.1 l should precede u in the timing T , and the **store** corresponding to a , denoted s , should succeed a in T . Thus, by transitivity, we get that l precedes s in T . Thus, $l \stackrel{c}{\rightarrow} s$, and therefore l precedes s in S' and in \hat{S} . Therefore, since by the construction, u immediately succeeds l and a immediately precedes s , u must precede a in S , a contradiction.

- *The first violating pair involves two assign x instructions.*

In Stage 1 of the construction, when an **assign** is inserted to \hat{S} then all the previous **assigns** that were not yet inserted join it, where their order in L is kept. Thus, at the end of Stage 1 all **assigns** (except for those left for Stage 3) were inserted to \hat{S} in their program order. This shows that all pairs of **assigns** in S are non violating.

- *The first violating pair involves two use x instructions.*

If both **uses** yield the same value w , then by the construction both are inserted after the same **load x,w** or **assign x,w** in Stage 2. Thus they keep the program order in S .

Suppose the **uses** return different values. We consider four cases:

- If the corresponding operations are both **assigns**, then by the arguments in the previous case we know that they keep the program order in S . However, since S is a legal serialization, a **use** returns the value of the most recent **assign/load**. Thus, since the order of **uses** is switched, so does the order of the corresponding **assign/loads**, a contradiction.
- If the corresponding operations are both **loads**, then by Constraint A.3.1 they must appear in T in the order of the **use** operations in L . These loads are also linked by a Causality Relation, and therefore they keep their order in S' and in \hat{S} . However, by the

We first show that in this construction, all the instructions in L are eventually placed into the serialization: In Stage 1 every `assign` in L gets inserted into \hat{S} , except for the ones at the end of the program order. In Stage 2 every `use` returns the value that was brought by some `load` or some `assign`, thus all of them (except possibly the ones at the end of L that do not have a corresponding `load` and that their corresponding `assign` was not inserted in Stage 1) get inserted into the serialization. Thus after Stages 1 and 2 all instructions in L were inserted into \hat{S} , except possibly for a suffix at the end of L . This suffix is inserted in Stage 3.

Denote by S the subsequence of `use/assign` operations in \hat{S} .

What remains to be shown is that S is a legal serialization of H , and that the order of `uses` and `assigns` is consistent with the Causality Relation.

S is a legal serialization of H

For the serialization S to be legal, we need to show that every `use` sees the result of the closest preceding `assign` in S . There are three ways in which `uses` are inserted to \hat{S} by the construction:

- *A use x, v is inserted after an assign x, v (Stage 2).* In this case, the `use` is inserted immediately after the `assign` as they both yield the same value, and no other `assign` can intervene between them. The same holds for the subsequence S .
- *A use x, v is inserted after a load x, v (Stage 2).* Once again, the `use` is inserted immediately after the `load`, and no other `assign` or `load` can intervene between them in \hat{S} . Since the serialization S' of `load/stores` was legal, the `load x, v` sees the result of the closest preceding `store x, v` in S' .

Consider the process of insertion of `uses` and `assigns` into \hat{S} . Since `assigns` are inserted immediately before the corresponding `load`, we conclude that no `assign x, v'` could be inserted between the `store x, v` and the `load x, v` . The `store x, v` certainly had a corresponding `assign x, v` (Constraint A.3.3). By the construction, the `assign x, v` was inserted immediately before the `store x, v` . We get a sequence of `assign-store-load-use` with no other `assign` instruction in between. Thus, in S , the `use` sees the result of the closest preceding `assign` in the serialization.

- *A use x that was inserted in Stage 3.* By Constraint A.2.1 a `use` instruction yields a value produced by some `load` or some `assign`. By the construction, all `uses` are inserted into \hat{S} during Stage 2, unless there is no corresponding `load` and the corresponding `assign` was not inserted in Stage 1. Thus the `uses` that are inserted in Stage 3 have their corresponding `assigns` preceding them in the program order of L , and these `assigns` are added to \hat{S} in Stage 3.

Furthermore, the sequence of `assigns` and `uses` added to \hat{S} in Stage 3 constitutes a legal serialization by itself; Otherwise there would be a `use x, v` with no preceding `assign x, v` , in which case there would necessarily be a corresponding `load x, v` , that would result in adding the `use x, v` to \hat{S} in Stage 2 and not in Stage 3. We conclude that the same sequence remains legal in S , and so the `use` sees the value given to x by the most recent `assign`.

S is consistent with the Causality Relation

We now show that the order of `use` and `assign` instructions is consistent with the Causality Relation. We will use the fact that the `loads` and `stores` were generated according to the Java constraints from the original program order of the `use/assigns`. We consider the relations defined in some arbitrary local history, L .

which processor or main memory are idle) by two subsequent `nops`. Finally, we add to each `read` and `write` operation a preceding `nop`.

The result is a timing T of all `u/a/l/s/r/w` instructions, in which the subsequences of the `use/assigns` in the local histories are the same as these in H . T complies with all the Java constraints: It is easy to verify that the constraints inside a thread (A.3) are preserved by the construction. The constraints between operations inside a thread and operations in the main memory (A.4) are preserved by the construction of T which maintains the timing relations from T' .

Direction 2. Java is at least as strong as JavaN. To prove this, we need to show that each execution that is valid under Java is also valid under JavaN. In other words, given Java execution H we need to show that there exists a legal serialization S of `use` and `assign` instructions which is consistent with the Causality Relation.

We construct the required serialization in the following way. First, we show a serialization of `load/store` instructions which is consistent with the causal relations between `load/stores`. Then, we will add to the `load/stores` the `use/assigns` (as in the given execution H), and will show that the added instructions form the required serialization.

Constructing the serialization S

Let T be an assignment of `l/s/r/w` operations and a timing for all instructions (those in H and the additional ones from the assignment) which complies with the Java constraints. Let T' be the `l/s/r/w` instructions in T and the timing induced on them. Let H' be an execution consisting of the `load/store` instructions in T . Since T complies with Java, T' complies with Java', thus H' is Java'. Since Java'=JavaN, H' is JavaN, i.e., there exists a legal serialization S' of the `load/store` instructions in H' , which is consistent with the Causality Relations between them.

Now, we extend S' to include also the `use/assign` operations from H . The resulting sequence is denoted \hat{S} . \hat{S} is initialized to S' , and then is extended by iterating on all local histories in H . For each variable x in the current local history L , its `use/assign` operations in L are inserted to \hat{S} in the following way.

The construction for local history L :

Stage 1 – Mapping assigns: We go through the operations in \hat{S} ; For each `store x,v` we add the corresponding `assign x,v` from L to \hat{S} , so that it immediately precedes the `store`. We also add all the `assigns` preceding `assign x,v` in L 's program order that were not yet added to \hat{S} , and write them just before `assign x,v` in the order in which they appear in L .

Stage 2 – Mapping uses: We now go through the operations in L ; For each `use x,v`, if there is a corresponding `load x,v` or `assign x,v` in \hat{S} , then we add the `use x,v` to \hat{S} immediately after the corresponding operation. Note that there may be several `use x,vs`. Note also that we use here the following assumption: Wlog there are no multiple `loads` and/or `assigns` to the same variable with the same value (otherwise we can add the instruction index to the actual value).

Stage 3 – Mapping leftovers: Finally, we add all the remaining `assigns` and `uses` to \hat{S} , so that their program order in L is preserved, and so that they reside at the end of the constructed sequence \hat{S} after all the previously inserted instructions.

4. For each variable x , there exists a total order of **read/write** operations, in which every **read** x yields the value of the latest **write** x,v (Constraint A.2.2).

Proof of 1. This follows by the construction: the **load** x,v is placed in T after its corresponding **store** x,v .

Proof of 2. Let's assume the contrary, i.e., that in T there is some **store** x,v that is placed after its corresponding **write** x,v . Suppose our **store** x,v is the first such situation in T . By the construction of T this can happen only if there is some other **load/store** instruction o which precedes our **store** x,v in the program order (of the corresponding local history), and that must be inserted after the **write** x,v . The only case when a **load/store** instruction is forced to appear after a main memory **write** instruction during the construction is when inserting a **load**, so we know that o is a **load** y,w .

Thus, we have the following situation: In the processor program order of H' , the **load** y,w appears before the **store** x,v , but the corresponding **write** x,v appears before the **read** y,w that is corresponding to the **load** y,w . But then we know that in H the **STORE** x,v is Causally related to the **LOAD** y,w , and thus in the JavaN serialization S , the **LOAD** y,w would appear before the **STORE** x,v . Therefore, in the timing T , the **read** y,w would appear before the **write** x,v — a contradiction.

Proof of 3. JavaN requires that all accesses to the same variable appear in the serialization S in the program order. By the construction of the Java' timing T , this order remains as is, and so this Java' requirement is satisfied.

Proof of 4. Since the main memory operations in T were injected according to the order of the corresponding instructions in the legal serialization S , the constraint follows. ■

3.5 Java Consistency is equal to JavaN

We are now ready to show that JavaN is also the memory behavior that can be assumed by the programmer. Recall that in contrast to Java' executions, in which a local history consists of a sequence of **load/stores**, executions in Java consist of sequences of **use/assigns**. Thus, the JavaN constraints in the proof of the theorem below correspond to sequences of **use/assigns** (where a **LOAD** and **STORE** in the constraint definition correspond to **use** and **assign**, respectively), and their relations to the **read/write** operations executed by the main memory.

In Theorem 6 we proved that Java', which constitutes the lower layer of Java, is equal to JavaN. We use this fact in proving that the upper layer, namely Java, is also equal to JavaN.

Theorem 7 *Java Consistency is equal to JavaN.*

Proof Direction 1. JavaN is at least as strong as Java. In order to prove this direction, we need to show that each execution which is valid under JavaN is also valid under Java. In other words, given an execution H that has a legal serialization of the **use/assigns** which is consistent with the Causality Relation, we construct an assignment of **l/s/r/w** operations and a timing T on all the instructions which is consistent with all the Java constraints.

Replace each **use** by a corresponding **load** and each **assign** by a corresponding **store**. Since the order of **load/stores** is the same as the order of their corresponding **use/assigns**, the resulting execution is JavaN. Since by Theorem 6 JavaN is equal to Java', this execution is also Java', so there exists an assignment of **read/writes** and a timing T' which complies with the Java' constraints.

We now extend this assignment and the timing T' with the **use/assign** instructions as follows: We replace each **load** x,v with a pair **load** x,v and a succeeding **use** x,v , and each **store** x,v with a pair **store** x,v and a preceding **assign** x,v . We also replace each **nop** in T' (time steps in

in \hat{H} , or the LOAD is after the STORE but there is another STORE x,w between them. Let's examine the cases:

1. o_1 is a STORE x,v and o_2 is a LOAD x,v but in \hat{H} o_2 precedes o_1 . Java' requires order of operations on every single variable in the main memory (Constraint A.2.2), and this order is preserved by the construction of \hat{H} . Thus, it is also the case that in H' the corresponding main memory operation `read x,v` precedes the corresponding main memory operation `write x,v` . By Constraint A.2.2 this cannot happen.
2. o_1 is a STORE x,v and o_2 is a LOAD x,v , and there is a STORE x,w operation o in \hat{H} between o_1 and o_2 . As above, H' implies a total order of main memory accesses to any single variable, and this order is preserved in the construction of \hat{H} . Thus, also in H' the `write x,w` instruction corresponding to o appears between o_1 and o_2 , which by Constraint A.2.2 implies a contradiction.

Direction 2: *JavaN is not weaker than Java'.*

Given an execution H which is valid under JavaN we show that H has an assignment of reads and writes to the main memory part and a timing which comply with the Java' constraints.

There exists a legal serialization S of H which preserves the Causal Relations in H . We construct the timing through the following steps:

1. We construct S' out of S by replacing each LOAD x,v by a `read x,v` and each STORE x,v by a `write x,v` .
2. We now extend S' to a sequence of operations T according to the following iterative process. We first let T consist of S' . Then, for every local history in H , for each of its LOAD instructions we insert a `load` instruction into T , and for each of its STORE instructions we insert a `store` into T . The local histories are handled in an arbitrary order, and the instructions of each local history are treated one by one according to the program order. The insertion of an instruction is as follows:
 - A `load x,v` is inserted immediately after both the last `load/store` instruction that was inserted, and its corresponding `read x,v` .
 - A `store` is inserted immediately after the last `load/store` instruction that was inserted.

By the construction, the instructions inserted into T for a certain local history of H preserve the program order of its corresponding instructions. Also, T contains corresponding main memory accesses, which thus may be viewed as an assignment of such operations to the main memory part of H . We denote this assignment by H' .

Since T is a serialization of all operations in H' we view it as a possible timing of H' , so that the i th instruction in T is executed at the i th time-step. If it is a `read/write` instruction then it is executed by the main memory. If it is a `load/store` instruction of which the original LOAD/STORE instruction in H belong to the local history of processor j then this instruction is performed by processor j .

We next show that the timing T complies with the Java' constraints. To this end, the following is required:

1. A `load x,v` is performed in T after its corresponding `read x,v` (Constraint A.4.2).
2. A `store x,v` is performed in T before its corresponding `write x,v` (Constraint A.4.3).
3. Main memory instructions which access a single variable and for which the corresponding `load/stores` appear in the same local history agree with the program order of that local history (Constraint A.4.4).

Figure 6 gives an example execution (taken from [15]) which is valid for Causal Consistency. This example is invalid for JavaN, as we later show that JavaN is equivalent to Java', and since Java' can be shown to be coherent by arguments similar to the proof given in Section 2.1. The fact that the example is invalid for JavaN is also easy to show directly from the definition of JavaN.

3.4 Java' is equal to JavaN

Theorem 6 *Java' is equivalent to JavaN.*

Proof Direction 1: *Java' is not weaker than JavaN.*

Given a Java' execution H we show that it has a legal serialization which preserves the Causality Relation, and thus H is also in JavaN.

Because H is in Java' it has an assignment of **reads** and **writes** to the main memory part and a timing which complies with the Java' constraints. We use such an assignment, and denote it by H' .

By Constraint A.2.4 the application of the Java' constraints to H' may not contain a cycle. Thus, the constraints induce a DAG on the set of operations in the execution. We thus obtain a serialization of H' (including the main memory part) which preserves the Java' constraints by using a topological sort. Consider the subsequence of this serialization \hat{H} which consists of the main memory operations only, and in which every **read** x,v (resp. **write** x,v) is replaced by the corresponding processor instruction **LOAD** x,v (resp. **STORE** x,v).

To prove that H is JavaN we now claim that \hat{H} is a legal serialization of H which preserves the Causality Relation (here we use the one-to-one correspondence between **read/write** and **load/store** operations).

Assume the otherwise. First assume that in \hat{H} there are two operations, o_1 and o_2 , so that there exists a Causality Relation from o_1 to o_2 , but o_2 precedes o_1 in \hat{H} . Let us call the first two cases of the Causality Relation (i.e., those that do not follow from transitivity) *basic*. Wlog we may assume that the Causal Relation between o_1 and o_2 is basic. (Otherwise, there is a sequence of operations $o'_1 = o_1, o'_2, \dots, o'_n = o_2$, such that each pair of o'_i, o'_{i+1} is Causally Related by one of the basic rules of Causality. Since o_2 precedes o_1 in the serialization, there must be at least one pair of operations o'_i, o'_{i+1} such that o'_i is Causally Related to o'_{i+1} by a basic rule, but o'_{i+1} precedes o'_i in \hat{H} .)

Consider the basic rules of the Causality Relation:

1. o_1 and o_2 are two accesses to the same variable that are performed by the same processor. In H' the **read/write** operations must appear in the same order as the corresponding **load/store** operations (Constraint A.4.4). Thus, the corresponding operations in the serialization \hat{H} must appear in the order that complies with Causality Relation requirements (which in H are the same as in H'), a contradiction.
2. o_1 is a **LOAD** x,v and o_2 is a **STORE** y,w performed by the same processor. The **LOAD** corresponds to a **load** instruction in H' , and the **STORE** corresponds to a **store** in H' , where the **load** precedes the **store** in the program order. But, by constraints A.4.2 and A.4.3, in H' the corresponding **read** should precede the **load**, and the corresponding **write** should succeed the **store**. By transitivity the **read** must precede the **write** in the topological order. By the construction this implies that the **LOAD** would precede the **STORE** in \hat{H} , a contradiction.

We now show that the serialization \hat{H} is legal. We assume the otherwise and get a contradiction. This can happen when a **LOAD** x,v that returns the result of some **STORE** x,v cannot see the result of the **STORE** according to the serialization \hat{H} . There are two cases: the **LOAD** is before the **STORE**

Java’: An execution consisting of **loads** and **stores** belongs to Java’ Consistency (or, Java’), when there is an assignment of **reads** and **writes** to the main memory part, so that there is a timing for the execution and the additional **read/writes** which complies with constraints from A.2 and A.4.

Note that Java’ cannot be seen as a subset of the Java constraints as Java also has additional **use/assign** instructions. The Java memory behavior defines the interaction of the **use/assigns** with the **read/writes**, where Java’ deals with the interaction of **load/stores** with the **read/writes**.

For the discussion in this section it is also important to note that the constraints in A.4 imply a one-to-one correspondence in Java’ between the **read/writes** and the **load/stores**.

3.3 JavaN

We now give a non-operational definition called JavaN, and then proceed in Sections 3.4 and 3.5 to show its equivalence to Java’ and Java, respectively.

We start with the definition of the *Causality Relation* denoted \xrightarrow{c} :

Causality Relation: Let o_1 and o_2 be two instructions performed by the same processor, where $o_1 \xrightarrow{po} o_2$. Then o_2 is *causally related* to o_1 ($o_1 \xrightarrow{c} o_2$) if one of the following holds:

1. o_1 and o_2 access the same variable, or
2. o_1 and o_2 access different variables, but o_1 is a LOAD and o_2 is a STORE, or
3. there exists instruction o' such that $o_1 \xrightarrow{c} o'$ and $o' \xrightarrow{c} o_2$.

The definition of JavaN is as follows:

JavaN: An execution H is JavaN if it has a legal serialization S that preserves the Causal Relation, i.e., if $o_1 \xrightarrow{c} o_2$ then $o_1 \xrightarrow{S} o_2$.

3.3.1 JavaN vs. Causal Consistency

Although JavaN is reminiscent of Causal Consistency [9, 15] it is incomparable to it. Basically, Causal Consistency requires that if two WRITES are causally related, they are necessarily seen by all other processors in the same order. In contrast, in JavaN if there are two READS that are not data dependent then they can see the WRITES in reverse order.

<i>Processor 1</i>	<i>Processor 2</i>	<i>Processor 3</i>
W X, 1	R Y, 1	R X, 1
R X, 1	R X, 0	R Y, 0
W Y, 1		

Figure 7: An execution which is invalid for Causal Consistency as in [9, 15] and is valid for JavaN.

Figure 7 gives an example which is invalid in Causal Consistency, since although the $w\ Y,1$ is causally dependent on the $w\ X,1$, Processors 2 and 3 see the writes in the reverse order. Here is a legal serialization for this same example which preserves the Causality Relation, thus showing that the example is JavaN.

R X, 0	W X, 1	R X, 1	R X, 1	R Y, 0	W Y, 1	R Y, 1
--------	--------	--------	--------	--------	--------	--------

3 Non-Operational Definitions for Java Consistency

In this section we present two non-operational definitions for Java Consistency, and show their equivalence to the original definition. The Section is organized as follows: First, in Section 3.1 we discuss some issues related to the bytecode production by the compiler and its execution on the JVM. Then, in Section 3.2 the implementor view of the Java memory behavior is presented, and we dub it Java'. Section 3.3 presents a new non-operational definition called JavaN. Section 3.4 gives the proof that JavaN is equal to Java'. Section 3.5 gives the proof that JavaN is also equal to Java. Section 3.6 presents a new definition which we call JavaD. Finally, we give in Section 3.7 a proof that JavaN is equivalent to JavaD.

3.1 Compliance to the Java Consistency Model

In the compilation stage of the Java source code, the order of `use` and `assign` instructions is determined by the compiler from the program code, and then `load` and `store` instructions are placed among them, as necessary. Note that the compiler may generate all the instructions at the same time, but the two stages structure is a more convenient way to examine the compilation process from a methodological point of view. The reason is that the placement of the `use` and `assign` instructions follows directly from the source code (Constraint A.3.1), whereas the placement of the `loads` and the `stores` is governed by the Java constraints according to the appearance of the `uses` and the `assigns`.

Note that during the execution stage the JVM can rely on the bytecodes only and has almost no information concerning the source code (some additional information is contained in the `.class` file, which specifies, in particular, which variables are volatile). Thus, if the compiler produces code that does not comply with the program, the JVM will not execute correctly. Therefore at the implementation level it can be assumed that the bytecodes produced by the compiler comply with the JLS constraints.

The same reasoning holds for the placement of `store` and `load` instructions by the compiler. There are cases in which it is possible to check the compliance of these instructions with the `assigns` and `uses`, but it seems infeasible to perform these checks during execution. For example, let's assume a Java method that begins with a `use` instruction of some variable (which is generally prohibited by Constraint A.3.4). This could happen, for example, because the compiler performed a global data flow analysis of the program and found out that every call to the method is preceded by an `assign` to the variable. Validating the code in this situation would require calculating again the global data flow of the program, which is obviously too expensive for run-time. Thus it can be assumed by the JVM implementor that the compiler produces correct bytecodes for `loads` and `stores`.

Since we conclude that the placement of `load` and `store` instructions should be left to the compiler, the responsibility that is left for the JVM is to execute the `u/a/l/s` instructions and map `load/store` instructions into `read` and `write` operations in the main memory.

3.2 Java'

Out of all the constraints in Appendix A, those in A.2 are general, and those in A.4 define the interaction of operations performed by the main memory with the instructions executed by the processors. The rest of the constraints in Appendix A are irrelevant for the implementation of the JVM memory manager (with respect to regular variables only). We thus refer to Constraints A.2 and A.4 as the definition of the JVM memory behavior, and call them: Java'.

Theorem 5 *The consistency model among volatile variables, namely, Volatile Consistency, is equal to Sequential Consistency.*

The two different directions are proven in the claims below.

Claim 5.1 *Volatile Consistency is not weaker than SC.*

Proof We show that there is a serialization of memory accesses to volatile variables, which is consistent with the program order of all the processors.

Consider the history of main memory accesses to volatile variables. From the constraints A.6.1 and A.6.2 there is a one-to-one correspondence between **reads** and **uses**, and between **writes** and **assigns**. Furthermore, from Constraint A.6.3, the order of the accesses is the same for the **uses** and **assigns** by a given processor, and for the **reads** and **writes** performed by the main memory on its behalf. Thus, if we show that there exists a serialization of all the main memory accesses to volatile variables, the same will be valid for the local accesses (which define the program order), and this will imply SC for accesses to volatile variables.

It can be verified that for volatile variables the following set of constraints subsume all others that are relevant:

1. There exists a total order of main memory accesses to any given variable, and in this order a **write** must precede **reads** that yield the written value (Constraint A.2.2).
2. There exists a total order of main memory accesses performed on behalf of any given processor, and this order is compatible with the program order of the processor (Constraint A.6.3).
3. There cannot be a set of dependencies between operations in which an operation is (transitively) dependent on itself (Constraint A.2.4).

The actual memory accesses in a given history involving only accesses to volatile variables comply with all the constraints. Given an execution H , we construct a directed graph $G_H = (V, E)$, as follows. The set of vertices V consists of the main memory accesses in H . There exist an arc $e \in E$, $e = (a \rightarrow b)$, $a, b \in V$, iff a precedes b according to some constraint. There are no circular dependency chains, hence G_H has no loops (is a DAG). Now, in order to show that H is sequentially consistent, define the required serialization as some topological order over G_H . By the definition of G_H the serialization is valid, as it takes into account all the data dependencies, and is consistent with the program orders of all the processors.

We conclude that the interaction between volatile variables is at least as strong as Sequential Consistency. ■

We remark that the claim remains valid even when H contains accesses to both volatile and regular variables, as the accesses to regular variables can only add constraints for the serialization of volatile variables, thus making their consistency model equal or stronger than Volatile Consistency.

Claim 5.2 *SC is not weaker than Volatile Consistency.*

Proof Consider a sequentially consistent history H . It is consistent with the program orders of all the processors, therefore all the main memory accesses are in the same order as the **use** and **assign** operations. This implies that H does not violate the volatile variables constraints, and is thus valid under Java, even when all variables are volatile. ■

the variables are updated, so a simple minded simulation of Java on ScC may result in an incorrect execution.

Entry Consistency (EC) [4] is a model defined similar to RC, except that the locks are associated with variables. In contrast to Java where the unlock operation implies synchronization of all the updates done by the processor during the synchronized code section, in EC when the lock on some variable is released only the value of this variable is made consistent between the processors. Thus, a simple minded simulation of an EC program on the Java memory model (by mapping EC locks to Java locks) may incur redundant updates not intended in the original program.

Release Consistency: In the original notation of RC, the operation corresponding to `lock` in Java is called `ACQUIRE`, and the operation corresponding to `unlock` is called `RELEASE`, but here we will rename them after Java as `lock` and `unlock`. The requirements for RC are [5]:

1. Before an ordinary access to a shared variable is performed, all previous `LOCKS` done by the processor must have already completed.
2. Before an `UNLOCK` is allowed to be performed, all previous `READS` and `WRITES` done by the processor must have already completed.
3. The `LOCK` and `UNLOCK` accesses must be processor consistent, with the corresponding memory model denoted RC_{PC} , or sequentially consistent, in which case the corresponding memory model is denoted RC_{SC} [13].

Proof To prove the theorem we must show that any execution possible under Java is also possible under RC, or, dually, that the Java constraints imply those of RC.

The first condition states that a `LOCK` succeeds in RC only if all the previous allocations of the lock (initiated by the `LOCK` operations) are freed by matching `UNLOCK` operations. This is obviously preserved in Java as required by Constraint A.5.1.

The second condition is also satisfied: all the `uses` (regarded as `READS` in the RC definition) which appear before the `unlock` instruction have their corresponding `READ` operations performed before the `unlock` instruction is issued, and thus before the `unlock` operation is performed by the main memory. The `write` operations (corresponding to the `assigns`) which appear before the `unlock` instruction are performed before the `unlock` in the main memory, according to the Constraint A.5.4.

The third condition is also satisfied. The Constraint A.4.1 forces the main memory agent to participate in all the operations on locks. Because Java defines a single main memory agent, this implies that only one such operation can be performed at a time, hence the consistency between locks is in fact *Linearizability* [8]. By [2] and [3] Linearizability is strictly stronger than Processor Consistency or Sequential Consistency, whichever is required for the RC model. Note that this result remains valid even when the actual implementation employs several main memory agents, as the memory behavior will remain unchanged. ■

2.5 Volatile Variables

Sometimes there are several variables in the program that are heavily used for transferring data between processors. In this case it is not convenient and not efficient to use locks for each access to these variables, since (as explained above) locks preserve Linearizability between them, and this requirement may not always be essential. This is where *volatile* variables are useful, as defined by the constraints in Appendix A.6.

We examine the behavior of code employing volatile variables only, and dub the corresponding memory model *Volatile Consistency*.

We now find serializations for each processor. For Processor 1, the required serialization is 2.2, 1.1, 1.2. It does not violate the semi-causality relation (because 1.1 is before 1.2, as is required by the \xrightarrow{s}), and there is only one WRITE operation for each variable, so the second part of the condition is satisfied trivially. The same can be shown for Processor 2.

We have shown that Java is neither weaker than PCD, nor stronger. We thus conclude that Java is incomparable with PCD. ■

2.4 Locks

There are two purposes for the use of locks in the Java language: first, to synchronize the flow of control between processors, and second, to synchronize the views of memory between different processors. The corresponding constraints are given in Appendix A.5.

In the bytecodes the locks are implemented by `lock` and `unlock` instructions. However, in the Java programming language there are no explicit `lock` and `unlock` operations. Instead, sequences of instructions may be marked as *synchronized*, implying the `lock` instruction at the beginning of the sequence, and `unlock` when it terminates. Thus, `lock` and `unlock` operations in Java always appear in pairs. They are put in the bytecodes by the compiler according to the synchronized sections in the source code.

Although the Java language defines correspondence between objects and locks, there is no such correspondence on the level of JVM bytecodes. So, `lock` and `unlock` operations, synchronize all the variables, and not only the ones stored in the object whose method was called.

We compare Java to Release Consistency (RC) [5] that is defined by distinguishing between two classes of operations: regular and special. Special operations are: `ACQUIRE` — the same as `lock` in Java, and `RELEASE` — the same as `unlock` in Java. Java is different from the Release Consistency in associating a lock with each object, while in the Release Consistency there is only one global lock.

Another difference between Java and RC is that the latter does not specify how the updates of variables propagate from one processor to another when no processor enters or leaves a synchronized code section. In particular, it is valid to implement RC with no updates whatsoever, except at synchronization points. In contrast, in Java the updates still follow the constraints as discussed throughout this section. From this point of view *Java is stronger than RC* as it enforces more constraints on the JVM execution.

The following theorem shows how code that is written for RC can perform correctly on the Java memory model. In other words, it shows that the programmer can rely on at least Release Consistency when writing in Java.

Theorem 4 *Java can simulate RC by mapping the special operations one to one: `ACQUIRE` to `lock` and `RELEASE` to `unlock`.*

Before we provide the formal definition of RC and the proof of the theorem we proceed to compare Java to other related models.

Scope Consistency (ScC) [10] is similar to RC, except that it employs multiple locks (associated with the so-called *consistency scopes*). Once a lock is released, the propagation of updates from one processor to another is done only when the other processor acquires the same lock. Thus, the memory updates in ScC are a subset of those in RC. Since we prove that any program valid for RC can execute correctly under Java, we conclude that any program valid for ScC can also run under Java. We remark, however, that Java is not equal to ScC (even with respect to the special operations only), since its multiple locks semantics are different: In Java, when a lock is released, all

serialization 2.1, 1.1, 1.2, and for Processor 2 the serialization 1.2, 2.1, 2.2. It is easy to see that these orders are legal in PCG and that the READS yield the required results. ■

2.3.2 Java vs. PCD

The definition of the PCD consistency in [2] is as follows:

PCD: We first define several notions:

- *Weak order relation* between two operations of the same processor: We say that o_1 *weakly precedes* o_2 , denoted $o_1 \xrightarrow{wpo} o_2$, if $o_1 \xrightarrow{po} o_2$ and either
 1. o_1 and o_2 are operations on the same variable, or
 2. o_1 and o_2 are both reads or both writes, or
 3. o_1 is a read and o_2 is a write, or
 4. (transitivity) there is another operation o' , such that $o_1 \xrightarrow{wpo} o' \xrightarrow{wpo} o_2$.
- *Weak writes-before*. $o_1 \xrightarrow{wwb} o_2$ iff $o_1 = W X,V$, $o_2 = R Y,U$, and there is another operation $o' = W Y,U$ such that $o_1 \xrightarrow{wpo} o'$.
- *Weak reads-before*. $o_1 \xrightarrow{wrb} o_2$ iff $o_1 = R X,V$, $o_2 = W Y,U$, and there is another operation $o' = W X,V'$, such that $o_1 \xrightarrow{S_x} o'$ and $o' \xrightarrow{wpo} o_2$, where $\xrightarrow{S_x}$ is some legal serialization for all operations on the variable x .
- *Semi-causality*. Semi-causality (denoted \xrightarrow{s}) is the transitive closure of the weak order, the weak writes-before, and the weak reads-before relations.

Now, a history H is PCD if:

1. H is coherent, i.e., for every variable x there exists a legal serialization S_x of $H|x$ that is consistent with all the processors views.
2. For each processor p , there is a legal serialization S_p of H_{p+w} such that
 - (a) if o_1 and o_2 are two operations in H_{p+w} and $o_1 \xrightarrow{s} o_2$, then $o_1 \xrightarrow{S_p} o_2$;
 - (b) for each variable x , if there are two write operations o_1 and o_2 to x , then these operations appear in the same order in S_x and S_p .

Claim 3.2 *Java is incomparable to PCD.*

Proof In order to show the claim we use the example from Figure 3 once again. We have seen that it is not valid under Java. In order to show that it is valid under PCD we check the execution against the stages of the PCD definition, verifying that none of them is violated.

To satisfy Condition 1 of PCD: The execution is coherent, as was shown by Claim 1.2.

To satisfy Condition 2 of PCD: First, we calculate the semi-causality relation for this execution.

1. *Program order relation*. There are two dependencies according to it: $1.1 \xrightarrow{po} 1.2$, and $2.1 \xrightarrow{po} 2.2$.
2. *Weak order relation*. Because in both processors a READ appears before WRITE, the dependencies remain the same: $1.1 \xrightarrow{wpo} 1.2$ and $2.1 \xrightarrow{wpo} 2.2$.
3. *Weak writes-before and Weak reads-before relation*. We can see from the definitions that no two operations are in any of these relations.
4. Finally, *semi-causality relation*. It is the transitive closure of the relations above, and thus is equal to the program order. So, the semi-causality relations are: $1.1 \xrightarrow{s} 1.2$ and $2.1 \xrightarrow{s} 2.2$.

Claim 2.2 *PRAM is not stronger than Java.*

Proof Figure 6 shows an example for an execution which is valid for PRAM and is invalid for Java. The reason for the difference, as was shown in the previous section, is that the Java constraints require Coherence for every single variable, while PRAM does not.

<i>Processor 1</i>	<i>Processor 2</i>	<i>Processor 3</i>	<i>Processor 4</i>
W X, 1	W X, 2	R X, 1	R X, 2
		R X, 2	R X, 1

Figure 6: A valid execution for PRAM which is invalid for Java.

Each one of Processors 3 and 4 sees an order consistent with program orders of both writing processors (1 and 2). Because PRAM does not require the views of Processor 3 and Processor 4 to be correlated, there is no contradiction, and thus the execution is valid under PRAM. However, the conflict in the views of Processors 3 and 4 implies that there is no legal serialization which preserves program order for both, therefore the execution is not coherent, and (as was shown in Section 2.1) is thus invalid in Java. ■

2.3 Java vs. Processor Consistency (PC)

[2] defines two variants of PC: PCD and PCG, of which the corresponding non-operational definitions are taken from [5] and [6] respectively. Both variants are shown to be stronger than PRAM, and since we have seen an example indicating that Java is not stronger than PRAM, we conclude also that Java is not stronger than either PCD or PCG. The following theorem answers the question whether Java is strictly weaker than any of them.

Theorem 3 *Java is incomparable to both PCG and PCD.*

2.3.1 Java vs. PCG

The definition of PCG, according to [2] is the following:

PCG: For each processor p there is a legal serialization S_p of H_{p+w} such that:

1. If o_1 and o_2 are two operations in H_{p+w} and $o_1 \xrightarrow{po} o_2$, then $o_1 \xrightarrow{S_p} o_2$.
2. For each variable x , if there are two WRITE operations to x then they appear in the same order in the serializations of all the processors.

Claim 3.1 *Java is incomparable to PCG.*

Proof To demonstrate this it remains to show that Java is not weaker than PCG, i.e., demonstrate an execution which is valid for PCG and is invalid for Java. We will use the same example given above for Coherence in Figure 3. We have already shown that this execution is invalid for Java. Now, we will show that it is valid for PCG.

The second condition of the PCG definition is trivially satisfied as there is only one WRITE operation to each variable. In order to satisfy the first condition we use for Processor 1 the

<i>Processor 1</i>	<i>Processor 2</i>
WRITE X, 1	READ Y, 1
WRITE X, 2	READ X, 1
WRITE Y, 1	READ X, 2

Figure 4: An execution valid for Java but invalid for PRAM.

From the program order of Processor 1, the operation `WRITE X,2` precedes `WRITE Y,1`. However, Processor 2 sees the change of x to 2 after it sees the result of the `WRITE` to y , hence it sees `WRITE X,2` after `WRITE Y,1`. Therefore, the execution is invalid under PRAM.

Now, let us show how this could happen in Java. As stated before, the operation `READ X,1` is **use** in the Java definition. The **stores** done by Processor 1 to x and to y are independent, and so it is possible for the processor to perform a **store** into y before it performs a **store** into x . Thus, the instructions actually executed by Processor 1 (left to right) could be as follows:

a x,1,
 s x,1,
 a x,2,
 a y,1,
 s y,1,
 s x,2

A possible local execution of Processor 2 could be:

l y,1,
 u y,1,
 l x,1,
 u x,1,
 l x,2,
 u x,2

Now, coupled with the main memory agent, this can produce the timing shown in Figure 5.

<i>Processor 1</i>	<i>Processor 2</i>	<i>Main Memory</i>
1. a x, 1		
2. s x, 1		
3. a x, 2		w x, 1 ; 1.2
4. a y, 1		
5. s y, 1		w y, 1 ; 1.5
6. s x, 2		r y, 1 ; 2.1
	1. l y, 1	r x, 1 ; 2.3
	2. u y, 1	w x, 2 ; 1.6
	3. l x, 1	r x, 2 ; 2.5
	4. u x, 1	
	5. l x, 2	
	6. u x, 2	

Figure 5: A possible timing for a Java execution implementing the program from Figure 4. The comment after each operation in the main memory column tells which **load** or **store** instruction initiated this operation; For instance: " ; 1.2" after the operation `w x,1` indicates that this operation was provoked by instruction 2 in Processor 1 (`s x,1`).

Since the values yield by the **use** operations in Processor 2 are the same as these obtained by the `READ` operations in Figure 4, we conclude that this scenario is possible under the Java constraints. ■

	Processor 1	Processor 2
1	READ X, 1	READ Y, 1
2	WRITE Y, 1	WRITE X, 1

Figure 3: Execution valid for Coherence and invalid for Java. As explained in Section 1.3, for Java READ denotes `use` and WRITE denotes `assign`.

`use` to see the result of `load`. Similarly, the operation `store y` by Processor 1 should appear after `assign y`. Because of the program order (Constraint A.2.1 and the explanation at the beginning of A.1), `use x` must complete before `assign y`, and thus `load x` appears before `store y`. Now, because of Constraints A.4.2 and A.4.3, the `read x` by the main memory on behalf of Processor 1 should appear before `load x`, and `write y` should appear after `store y`. By transitivity, `read x` performed by the main memory for Processor 1 is before `write y`. For the same reasons, `read y` by the main memory on behalf of Processor 2 is before `write x`.

Let us denote the memory accesses by the indices of their corresponding instructions, e.g., the operation `read x` performed by the main memory will be denoted 1.1. In order to produce the required results, the interleaving of memory accesses must have 2.2 before 1.1, and 1.2 before 2.1. However, together with the constraints discussed in the previous paragraph, this induces a dependency loop ($1.1 \rightarrow 1.2 \rightarrow 2.1 \rightarrow 2.2 \rightarrow 1.1$), in which the instruction 1.1 follows itself. This is prohibited by Constraint A.2.4. Thus, this scenario is impossible as a Java execution. ■

We remark here that the additional restriction on the memory behavior which is imposed by Java and not by Coherence is the “Causality relation”, which basically (and informally) forces writes that follow local reads to keep this order in the view of all processors. This section and the following one will give examples, explanations, and formal definitions on the way Java enforces this restriction.

2.2 Java vs. PRAM Consistency

The definition of the PRAM consistency is as follows [2]:

PRAM: History H is PRAM if for each processor p there is a legal serialization S_p of H_{p+w} , such that if o_1 and o_2 are two operations in H_{p+w} , and $o_1 \xrightarrow{po} o_2$, then $o_1 \xrightarrow{S_p} o_2$.

Theorem 2 *Java is incomparable (neither stronger nor weaker) to PRAM.*

To demonstrate this we show two example executions, one that is valid under Java but is invalid under PRAM, and another which is valid for PRAM but not for Java.

Claim 2.1 *Java is not stronger than PRAM.*

Proof Figure 4 shows an execution which is valid for Java and is invalid for PRAM, thus suggesting that Java is not stronger than PRAM. The reason for the difference between Java and PRAM in this example is that the Java constraints draw little connection between operations on different variables, whereas PRAM requires that program order is preserved for all operations in the processor (On the other hand we already know that Java imposes some connections between operations on different variables, as it is stronger than Coherence).

there can be no additional `store` until the `store-block` terminates, which, by definition of a `store-block`, happens when the next instruction in the input is not `use` or `store`. Thus, when the block terminates, then the next instruction is either `assign` or `load` implying the beginning of either a new `load-block` or a new `store-block`, accordingly.

Next, we notice that each block has only one associated `load` or `store` instruction. Because of constraints A.4.2 and A.4.3, there is exactly one operation by the main memory corresponding to each block. Because of constraint A.2.2, those memory accesses are totally ordered. Now, we construct a global order of the blocks by placing them one after another in the order of their corresponding main memory accesses. This induces a serialization of the instructions, which, as we show below, can serve as the required serialization of $H|x$. Note that in the construction, because of Constraint A.4.4, the order of operations performed by any individual processor does not change.

Although there are other (`load` and `store`) operations in the constructed order, only the order of the `assign` and `use` instructions is of interest. In order to show that this is indeed a legal serialization, it remains to show that the results seen by `use` operations in the original execution are the same as in the constructed order.

Consider the last `assign` or `load` operation, op , that appears before a `use` in the program order of some processor. No operation which appears between op and the `use` can change the value set by op , so this value is yielded by the `use`. It is easy to see from the definition of the expression, that op appears in the same block as the `use`. Thus, in the constructed serialization no operation by any other processor may intervene between op and the `use`. Let us look at the possibilities:

- op is `assign`. In the serialization constructed above, the result of this `assign` is seen by the `use`. Since in the original Java execution, the value provided by this `assign` is seen by the `use`, we conclude that the `use` sees the same value in the original execution and in our serialization.
- op is `load`. In the original execution, the value brought by `load` was read by a corresponding preceding `read` operation (Constraint A.4.2), which, in turn, sees the result written by the closest preceding `write`. The `write` corresponds to a `store` instruction by some processor (Constraint A.4.3). In the order of memory accesses in the original execution this `write` is the closest to our `read`, and thus, in the serialization, it resides in the closest `store-block` to the `load-block` that contains our `load` and `use` instructions. Since there are no `assigns` in the blocks that can be placed between these two blocks, the last `assign` performed by the `store-block` is seen by our `use` in the serialization. Now, in the original execution, the last `assign` of the `store-block` provides the value to be written by the `store` of that block, and its value is seen by the `use`. We conclude that both in the original execution, and in our serialization, the `use` sees the value provided by the same `assign`.

■

Claim 1.2 *Java is strictly stronger than Coherence.*

Proof To demonstrate this we show an execution which is valid for Coherence and is invalid for Java.

Consider the execution in Figure 3. In order to show it is coherent, set the order of accesses in the serialization for variable x as 2.2, 1.1, and the order of accesses to y as 1.2, 2.1. Clearly these are legal serializations which preserve program order.

For Java, however, this execution is impossible. We must show that no assignment of `load/store` and `read/write` operations that is consistent with the Java constraints can yield the same results as in the example. The `load x` operation by Processor 1 should appear before the `use x` for the

2 Java Consistency and Conventional Models

In this section we compare Java Consistency to some other models that appear in the literature. The section is organized as follows: Section 2.1 shows that Java is coherent. Section 2.2 shows Java is incomparable to PRAM Consistency. Section 2.3 shows Java is incomparable to both versions of Processor Consistency. Section 2.4 shows Java can perform Release Consistency programs by using locks. Section 2.5 shows the consistency of volatile variables is sequential.

In addition, we show in the next Section that Java is incomparable to Causality (This is deferred to Section 3.3.1 for reasons of readability).

2.1 Java vs. Coherence

The definition of Coherence, as in [2], is:

Coherence: A history H is said to be *coherent* if for each variable x , there is a legal serialization S_x of $H|x$ such that if o_1 and o_2 are two operations in $H|x$ and $o_1 \xrightarrow{po} o_2$ then $o_1 \xrightarrow{S_x} o_2$. A consistency model is said to be *coherent* if every history under it is coherent, and the corresponding memory model is called *Coherence*.

Theorem 1 *Java Consistency is stronger than Coherence.*

To demonstrate this, we will show two things: first, that Java is not weaker than Coherence, i.e., that any execution that is valid for Java is also valid for Coherence, and second, that Java is not equivalent to Coherence, i.e., that there exists an execution that is valid for Coherence but is not valid for Java.

Lemma 1.1 *Java is coherent, i.e., for each execution history H and a variable x there is a global serialization of $H|x$ which is consistent with the views of all the processors.*

Proof Consider the set of operations $H|x$ on some variable x in some Java execution H . In order to show the required serialization, we divide the sequence of operations specified by each processor into blocks, and then arrange the blocks in a global order between all the processors. The division is specified by the following regular expression.

```
Order = (load-block | store-block)*
load-block = load (use)*
store-block = assign (use | assign)* store (use)*
```

First, we show that the expression covers all possible executions. We need to show that when a block terminates, the next operation in the sequence will be `assign` or `load`. A new block begins at the beginning of a sequence or when the previous block terminates. If the execution is finite, a block may terminate prematurely.

Constraint A.3.4 requires that the first instruction in the sequence for any variable is `load` or `assign`, so the first block in the beginning of the sequence can correctly start.

If the current block is a `load-block`, then any number of `uses` is going to the same block. No `store` instruction can appear because of constraint A.3.3. The appearance of `load` or `assign` will start a new block, and it is well-defined (`load-block` or `store-block`, respectively). Therefore, a `load-block` terminates correctly and is followed by another block.

If the current block is `store-block`, then because of Constraint A.3.2, after the `assign` no `load` can appear before at least one `store` instruction. Once the `store` appears, because of A.3.3

These operations can be seen as executing in different layers. The `use` and `assign` operations follow immediately from the source code of the Java program. We say that the *program order* of a thread is the order of `use` and `assign` instructions it issues. These instructions are generated by the compiler from the source code according to the semantics of the Java language. They are local to the processor, and thus their generation is independent of the constraints on the memory behavior, and does not interfere with the memory consistency specification. The `load` and `store` instructions are inserted to the bytecode by the compiler according to the constraints between `uses/assigns` and `loads/stores`. These constraints constitute the *upper layer*.

When the program is executed on a JVM, the `load` and `store` instructions in the bytecode initiate the execution of `read` and `write` instructions which are performed in the main memory. The set of constraints that govern the relation between the `loads/stores` and the `reads/writes` constitutes the *lower layer*. This set of constraints will be denoted as Java' below (see Section 3).

Since the programmer explicitly influences only the `use` and `assign` instructions, we are interested in how these instructions can be seen by other processors. This will be referred to as the "programmer view" of Java. The `use` and `assign` instructions are local, so a local `use` sees the result of a local `assign`. However, the result of the `assign` is only seen by remote `uses` (which access the same variable) at other processors when it is propagated to them by a sequence of `store-write-read-load` operations. A local `use` instruction is typically not seen by remote processors since it produces nothing that can be propagated.

All the constraints are quoted in the Appendix A, and will be referred below by their section number and index within the section. For example, *Constraint A.2.1* means Constraint 1 in Section A.2 of the Appendix.

1.3 Java Execution, Java Consistency and Notation

The standard notation in the literature denotes the operations that are executed locally by a processor as `READ` and `WRITE`. Since in the original Java notation from JLS these operations are denoted as `use` and `assign`, respectively, and since `read` and `write` are used there for operations done by the main memory, we chose to present Java executions by means of sequences of `uses` and `assigns`.

A *timing* for a Java execution determines for each operation the time-step in which it is performed, where each operation is assumed to take a single time-step.

Java Consistency: An execution consisting of `uses` and `assigns` belongs to Java Consistency (or, Java), when there is an assignment of `loads` and `stores` to the processors parts, and an assignment of `reads` and `writes` to the main memory part, so that there is a timing for the execution and the additional `load/store/read/writes` which complies with the Java constraints.

As mentioned above, for presenting execution examples for both Java and other models a `READ` instruction refers to either `use` or `READ`, depending on whether we consider Java or another model, respectively. Similarly, `WRITE` corresponds to either `assign` or `WRITE`.

For brevity, we sometimes denote `use` as `u`, `assign` as `a`, `load` as `l`, `store` as `s`, `read` as `r` and `write` as `w`.

Consistency A is *strictly stronger* than B if $H_A \subset H_B$. Dually, definition A is stronger than definition B if it has more restrictions on the set of possible variable updates.

All READS and WRITES operate on individual variables. The variables are assumed to be of some built-in atomic type in the machine architecture. For instance, *objects*, which are regarded as variables in some object-oriented languages, are not considered variables here, since an object may consist of many atomic variables.

One important model is the Sequential Consistency (SC) [11].

Sequential Consistency: A history H is said to be *sequentially consistent* if there is a legal serialization S of H such that if o_1 and o_2 are two operations in H and $o_1 \xrightarrow{po} o_2$ then $o_1 \xrightarrow{S_x} o_2$. In other words, there is a serialization of H which is consistent with the views of all the threads. A memory model is called Sequential Consistency when every execution under this model is sequentially consistent.

1.2 The Java Virtual Machine (JVM)

As mentioned above, the memory behavior in the Java Language Specification (which we call JLS, see Appendix A.1 and Chapter 17 in [7]) defines Java by specifying an abstract memory system, AMS, a set of operations, and the constraints that are imposed on them. The machine consists of a *main memory agent* (for brevity, *main memory*) and *threads* (here referred to as *processors*, see below), each of which has its own local memory. *Variables* are stored in the main memory, and their values are available for computation by a thread only after they are explicitly brought to its local memory. In some situations, they are also written back from the thread local memory to the main memory. Each one of the threads is executed by a *thread engine*, which can be implemented as a separate CPU, a thread provided by the operating system, or some other mechanism. For the sake of consistency with previous works, we use in the rest of the paper the term *processor* which is the term used in the definition of most conventional consistency models. Thus, when talking about Java, the term *processor* refers to the notion of a thread.

The operations are divided into four classes:

Operations local to the thread engine. The operations are `use` and `assign`. `use` loads the local copy of a variable into the engine for some calculation, and `assign` writes the result of the calculation into the local copy of the result variable. There are no individual `use` and `assign` instructions in the bytecode, but the operations specified by the bytecodes use several of them during the execution. For example, `x=y+z` in the Java source code implies bytecode instruction `add`, which involves `use y`, `use z`, and finally, `assign x`.

Operations between the thread and the main memory. There are two of them: `load` and `store`. `load` transfers the value of a given variable from the main memory to the local copy, and `store` transfers it back. These operations are represented by explicit instructions in the bytecode (their corresponding JVM mnemonics are `get field` and `set field` [14]).

Operations performed by the main memory. They are: `read` and `write`. These operations are initiated by the main memory as results of `loads` and `stores`. `read` actually reads the value that is yielded by the `load` instruction, and `write` stores the value brought by the `store` instruction.

Locking operations. They are: `lock` and `unlock`. They serve for synchronization of memory and program control flow. There are explicit `lock` and `unlock` operations in the bytecode, and they are performed in tight coupling with the main memory agent.

views are in fact the same.

The JVM can be implemented as a true interpreter, which executes the instructions one-by-one, or as an optimizing *Just In Time compiler* (JIT), which compiles the bytecodes to native machine code, and then executes it. In both cases, regardless of possible optimizations, we are interested in the exhibited memory behavior. Hence the internals of the implementation, and in particular whether it is an interpreter or a JIT, do not make any difference for the results presented in this paper.

The rest of the paper is organized as follows. In the rest of this section we give definitions and explain the Java memory model. In Section 2 we compare Java to some conventional consistency models that appear in the literature. Section 3 provides the new non-operational definitions of Java, called JavaN and JavaD. Section 4 give our conclusions. For completeness, we also provide in the Appendix A the original list of constraints from [7] for the implementation of Java on top of the AMS.

1.1 Definitions and Conventions

There are three types of instructions in this paper. We distinguish them by the font used to write the instructions, as follows.

Java code – a typewriter font will be used: `use`, `assign`, `load`, `store`, `read`, `write`.

Code in other memory models – a small caps font will be used: READ, WRITE.

Code for both Java and other models – once again, a small caps font will be used.

In our examples we always assume that the variables are initialized to 0. The instructions in the examples are indexed by the processor name and the instruction index in the processor local program, so instruction i in the program of processor j is denoted as $j.i$.

We follow the definitions presented in [2]. A *local history* of a processor p , denoted H_p , is a sequence of READ and WRITE operations, denoted o_1, o_2, \dots , that are performed by p . READ x, v denotes a read operation, where the source variable is x and the operation yields v as its result. WRITE x, v denotes a write operation, where the destination variable is x , and the value to be written is v . The order of operations in H_p defines the *program order*, po , and $o_1 \xrightarrow{po} o_2$ denotes that instruction o_1 appears before o_2 in the program order. A *global history*, or just *history* H , is a collection of all local histories for the execution. In the sequel, we sometimes call history: *execution*, and use these terms interchangeably.

A *serialization* S of the history H is a linear sequence containing all the operations of H . We denote $o_1 \xrightarrow{S} o_2$ when o_1 appears before o_2 in the serialization S . A serialization is *legal* if each READ operation returns the result of the latest WRITE to the corresponding variable.

For a given history H and a processor p , denote as H_{p+w} the partial history consisting of all the operations of p and all the WRITE operations of other processors. Denote also $H|x$ the partial history of H consisting only of operations on the variable x .

A *memory consistency model* (or simply, *consistency*) for a system of processors which use a collection of variables is a set of constraints (or, a governing protocol) on the way the variables are modified, and/or the way these modifications are seen by the processors. A dual definition for the consistency would specify the set of executions that are possible according to the constraints. Throughout the paper we introduce consistency models in both ways. Loosely speaking, the first way corresponds to operational definitions, while the second corresponds to non-operational ones.

We say that consistency A is *stronger* than consistency B if, for any program P , the set of histories possible under A , H_A , is contained in the set of histories possible under B , H_B ($H_A \subseteq H_B$).

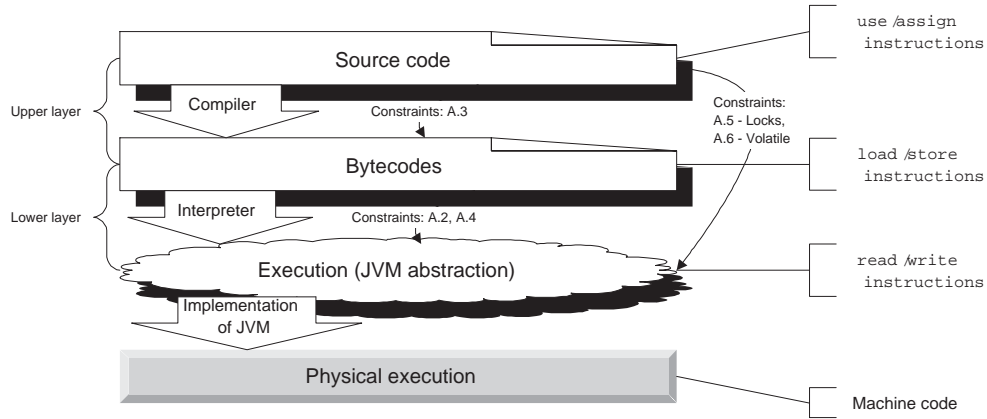


Figure 1: The layered structure of the Java programming paradigm.

Note that the requirements on the memory specifications provided for the programmer are slightly different from those used by the implementor. The programmer needs to know that if the program is correct under the model specified, then it is correct under the actual implementation. Thus, the model provided to the programmer should be weaker than or equal to the standard Java Consistency. For the implementor, if the program executes correctly under the original Java model then it must also perform correctly under the implementation according to our definition. Thus, from the implementor point of view, the actual implementation must be stronger than or equal to the original Java memory consistency model.

Figure 2 schematically depicts the programmer and implementor views. At different levels different types of the AMS operations are added to the execution (the process is explained in Section 1.2 below). Note that the original operational specification defines the relation between different levels of operations in the same processor, whereas the non-operational definition gives the relations between the views of different processors at the same level.

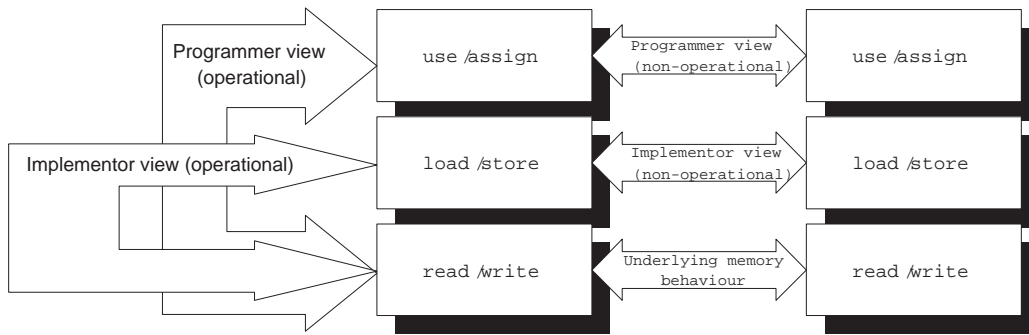


Figure 2: Programmer and implementor views in Java: Operational vs. Non-Operational.

As it turns out, the new definitions that we present apply to both the programmer view and the implementor view (assuming the compiler of the source code produces correct bytecodes). Somewhat surprisingly (and hard to predict from the original definition), this shows that these

1 Introduction

One of the more interesting and useful features of Java is its built-in concurrency support in terms of multithreading, a feature that can be exploited for several purposes [12]. In this paper we are interested in using multithreading for writing parallel code, so that different threads may execute using different processors or different machines. Since the threads use shared memory, executing on a distributed environment requires that the system provides distributed shared memory (DSM). Typically, in order to execute efficiently in parallel on such an environment the conditions on the coherency of the shared memory are weakened, thus preventing “implicit” communication that is sometimes provoked by the use of the shared memory.

A Java system consists of a compiler that, given the source code, produces bytecodes that are platform-independent and are thus absolutely portable, and an interpreter, called the *Java Virtual Machine* (JVM), that executes the bytecodes on the platform of choice. To preserve portability, the JVM must be able to execute the bytecodes produced by a common compiler, without any modifications. The compiler thus must comply with the standard definition of Java, as given in the Java Language Specification book [7][Chapter 17] (JLS).

Although the JLS provides a standard definition for the Java memory model, the description is given in terms of an implementation on some specific abstract memory system (AMS). The definition in JLS consists of a set of constraints binding the program code with the actual execution on the AMS (see Appendix A). All the constraints given are operational, defining how possible AMS executions for a given thread can be produced from its program code. Since Java allows for some weakening of the consistency of the shared memory, JLS actually describes the way multiple copies of the same data are kept coherent using the AMS (which implicitly defines the strength of the derived coherency on any other memory system as well).

The specification of the Java Consistency on the AMS consists of two main layers: the upper one, which defines the relation of the program code with the Java bytecodes, and the lower layer, which defines the relation between the bytecodes and the actual execution sequences. This layered structure is illustrated in Figure 1. Thus, the resulting consistency model, as the relationship between the program code and the execution in the processor, and further the relations between executions on different processors, is rather complicated. Obviously, the definition given in JLS is inconvenient for both the Java programmer and for the JVM implementor, who do not have the AMS in hand.

In this work we are interested in providing useful non-operational characterizations of the Java memory model (Java Consistency, or simply Java). Non-operational models specify just how strongly should the shared memory be kept consistent across processors in a distributed environment. They attempt to keep the specification independent of a specific – even an abstract – machine, and thus are “cleaner”, are easier to implement, and are easier to understand, than the operational models.

We compare Java to existing memory models for which implementation algorithms and programming methodologies have already been developed. All the conventional consistency models that we refer to are given in the literature in terms of non-operational definitions. Comparing them to Java may assist in the selection of those programs and algorithms which can be adapted to Java even though they may be using other memory models.

We then present two non-operational definitions and show that they match exactly the original definition. These definitions are relatively simple, and are useful in guiding programmers how to use the shared memory efficiently, and in the design of efficient virtual machines in a distributed environments.

Java Consistency = Causality + Coherency

Non-Operational Characterizations for the Java Memory Behavior

Alex Gontmakher

Assaf Schuster

Computer Science Department, Technion
{gsasha, assaf}@cs.technion.ac.il

Abstract

We provide non-operational, declarative characterizations of the Java memory coherency model (Java Consistency, or simply Java). The work is based on the operational, imperative definition of the Java memory consistency as given in the Java Language Specification [7].

We first compare the memory behavior to that of some previously studied models, proving that Java is incomparable to PRAM Consistency and to both versions of Processor Consistency. We show that a programmer can rely on Coherence for regular variables, Sequential Consistency for volatile variables, and Release Consistency when locks are employed.

We then present two declarative, non-operational definitions, which essentially show that Java Consistency is some combination variant of Causality and Coherency.

Acknowledgments: We thank Jan Groth, Ayal Itskovitz, and Avi Mendelson, who took part in the initial discussions that motivated this work.