Parallel Processing Letters, Vol. 13, No. 4 (2003) 721–733

World Scientific
www.worldscientific.com

COMPLEXITY OF VERIFYING JAVA SHARED MEMORY EXECUTION*

ALEX GONTMAKHER†, SERGEY POLYAKOV‡ and ASSAF SCHUSTER§
Computer Science Department
Technion — Israel Institute of Technology
Technion City, Haifa 32000, Israel

Received June 2003
Revised August 2003
Accepted by P. B. Gibbons

ABSTRACT

This paper studies the problem of testing shared memory Java implementations to determine whether the memory behavior they provide is consistent. The complexity of the task is analyzed. The problem is defined as that of analyzing memory access traces. The study showed that the problem is NP-complete, both in the general case and in some particular cases in which the number of memory operations per thread, the number of write operations per variable, and the number of variables are restricted.

Keywords: parallel processing, shared memory, testing, memory consistency, computational complexity, Java, Java Virtual Machine

1. Introduction

Programmers who write multithreaded programs for shared memory systems are interested in a high-level view of the memory system. There exist a number of standard shared memory models: Sequential Consistency, Coherence, PRAM Consistency, Processor Consistency, etc. They provide various kinds of tradeoffs between time efficiency and memory system flexibility, on the one hand, and the simplicity of writing a correct program on the other. Systems running Java must also provide some standard for shared memory behavior, in accordance with [4]. We call this standard *Java consistency*.

A memory system promising Java consistency may fail for several reasons:

*Research supported by the Bar-Nir Bergreen Software Technology Center of Excellence
†gsasha@cs.technion.ac.il
‡psergey@tx.technion.ac.il
§contact address; assaf@cs.technion.ac.il

(a) There may be errors in the compiler.
(b) There may be errors in the implementation of the Java Virtual Machine.
(c) There may be errors in the underlying operating system or hardware.

We are interested in testing the Java system in general, observing it from the higher level, which is called in [3] the *programmer view.* The programmer view focuses on how **use**, **assign**, **lock** and **unlock** operations interact. Our assumption is that **lock** and **unlock** operations are not used. Thus, the traces of **use** and **assign** operations, which are the only operations explicitly influenced by the programmer, can be used to test for Java consistency.

We use the Java shared memory model developed in [3]. This model is called the "non-operational characterization." This model was proved to be equivalent to that given in "The Java Language Specification," for the case without synchronization operations and prescient stores. We use the non-operational definition because the standard definition is very complicated and relies on a specific abstract machine as its underlying model.

Below we cite the non-operational definition of the Java programmer view, the case without prescient stores and synchronization operations, as it is given in [3]. This definition is called $\text{Java}_{\text{PS}}^{N}$.

A *serialization* S of a program P is a sequence containing all the operations in P. S is *legal* if each `READ X` operation o yields the result of the most recent `WRITE X` operation preceding o in S.

Let C be a set of order relations. If o_1 should precede o_2 according to C, we denote this $o_1 \xrightarrow{C} o_2$.

A *program order* for a certain thread is a sequence of all the operations performed by the thread, in the order in which they are to be issued.

The *Legal Serialization Consistency for* C, denoted $LS(C)$, is defined as follows:

LS(C): Execution H is $LS(C)$ if there is a *legal serialization* S of H such that
$o_1 \xrightarrow{C} o_2 \Rightarrow o_1 \xrightarrow{S} o_2$.

The *Causality*^{T} relation, denoted CR^T, is as follows:

Causality^T Relation: Let o_1 and o_2 be two operations performed by the same thread, where $o_1 \xrightarrow{po} o_2$ (o_1 precedes o_2 in the program order).Then, $o_1 \xrightarrow{CR^T} o_2$ if one of the following holds:
same variable, where o_1 and o_2 access the same variable, or
transistor rule, where o_1 is a `READ`$^{\checkmark}$ (`READ`$^{\checkmark}$ denotes reading a value written at another thread) and o_2 is a `WRITE`.

We define $\textbf{Java}_{\textbf{PS}}^{\textbf{N}}$ to be $LS(CR^T)$.

Consider traces of memory access operations for all threads running in some program. A trace of a thread is called a sequence. Each sequence consists of

(READ/WRITE)(x_i, v_i) operations, where x_i is some variable and v_i is the value which is read/written by the operation. We define the problem of verifying Java$^N_{PS}$ consistency of a shared-memory execution (VJC$_{PS}$) as follows.

VERIFYING JAVA$^N_{PS}$ CONSISTENCY OF A SHARED MEMORY EXECUTION (VJC$_{PS}$)

INSTANCE: Input in the form of a set of tables, each one of the type:

Thread t_1	Thread t_2	$\cdots$	Thread t_m
$Op_{11}(x_{11}, v_{11})$	$Op_{21}(x_{21}, v_{21})$	$\cdots$	$Op_{m1}(x_{m1}, v_{m1})$
$Op_{12}(x_{12}, v_{12})$	$Op_{22}(x_{22}, v_{22})$	$\cdots$	$Op_{m2}(x_{m2}, v_{m2})$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$Op_{1n_1}(x_{1n_1}, v_{1n_1})$	$Op_{2n_2}(x_{2n_2}, v_{2n_2})$	$\cdots$	$Op_{mn_m}(x_{mn_m}, v_{mn_m})$

Sequences are given in columns. (Columns here are united in tables to reflect the grouping of sequences according to their meaning, as we shall group them in our proofs later in the text). t_k is the thread the sequence belongs to. Op_{ij} may be R or W (READ or WRITE), x_{ij} is the variable name, and v_{ij} is the variable value. The complexity parameter will be the length (number of characters) of some instance's input, including table start delimiters, column delimiters, and row delimiters.

QUESTION: Does a given collection of thread sequences correspond to some valid Java$^N_{PS}$ execution?

We prove that the problem is NP-complete in the general case and even under various restrictions.

2. Verifying Java Consistency

In this section we prove that the VJC$_{\mathbf{PS}}$ problem is NP-complete.

The VJC$_{\mathbf{PS}}$ problem is in NP. Indeed, given a schedule and a collection of thread sequences, we can check their correspondence for Java$^N_{PS}$ validity in cubic time. First, we test whether the schedule preserves CR^T for each thread. We can do this without exceeding cubic time, by successively checking each operation in the thread against all subsequent operations in the thread. There is a quadratic number of checks, and for each one we must scan the schedule to find the location of the two tested operations. Second, we check that the schedule does not contain violations of legal serialization. This can be done in linear time in one pass through the schedule, by simulating the operations.

2.1. The VJC_{PS} problem for short sequences

Theorem 1 *The VJC_{PS} problem, restricted to instances in which each sequence contains at most three memory operations and each variable occurs in at most four write operations, is NP-complete.*

Proof. We use the reduction from 3-Satisfiability(3SAT).

Consider a 3SAT instance $\mathcal{F}$ with n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. Let $S(v_i)$ be T when v_i appears in a given clause, and F when $\overline{v_i}$ appears. We denote an appearance of v_i as $[v_i, S(v_i)]$. We represent the value of $S(v_i)$ in clause j as $S_j(v_i)$.

Let us construct a $\mathrm{VJC_{PS}}$ problem instance $\mathcal{V}$, such that $\mathcal{F}$ can be reduced to $\mathcal{V}$.

In our model each variable v_i in the schedule may be assigned one of two values, T or F.

To simulate an assignment to v_i, we produce, for each variable v_i, the following operations, scheduled in three sequences, V_i^1, V_i^2, V_i^3:

Thread V_i^1	Thread V_i^2	Thread V_i^3
W(v_i, T)	W(v_i, F)	R(z_m, T) W(v_i, T) W(v_i, F)

Consider the order of two writes, W(v_i, T) in thread V_i^1 and W(v_i, F) in thread V_i^2. Correlate this order to the value assigned to v_i in $\mathcal{F}$: if W(v_i, T) is the first, assume that v_i is assigned T; otherwise, v_i is assigned F.

An OR is simulated by having three writes of the same value to the same location; a read can be scheduled after either write. For each clause $C_j = [v_p, S(v_p)] \vee [v_q, S(v_q)] \vee [v_r, S(v_r)]$, we construct three sequences, C_j^1, C_j^2, C_j^3:

Thread C_j^1	Thread C_j^2	Thread C_j^3
R($v_p, S_j(v_p)$) R($v_p, \overline{S_j(v_p)}$) W(c_j, T)	R($v_q, S_j(v_q)$) R($v_q, \overline{S_j(v_q)}$) W(c_j, T)	R($v_r, S_j(v_r)$) R($v_r, \overline{S_j(v_r)}$) W(c_j, T)

Finally, we produce the following sequences, A^1, A^2,..., A^m:

Thread A^1	Thread A^2	...	Thread A^m
R(c_1,T) W(z_1,T)	R(c_2,T) R(z_1,T) W(z_2,T)	...	R(c_m,T) R(z_{m-1},T) W(z_m,T)

This simulates the AND part of the clauses. Note that z_m may be set to value T only if every c_j is set to T by some of the sequences C_j^1, C_j^2, C_j^3 (which simulate the OR clause j) in the group.

Note also that the first and second writes of T or F to some variable v_i will take place before the first use of this variable as a basis for writing T to some c_j.

The third and fourth writes to v_i must be scheduled after all AND sequences, because value T must be read from z_m in V_i^3 before this. Value T may be read from z_m in V_i^3 only after all AND sequences have ended.

The order of operations in the schedule is supported by the same variable and transistor rules (given in the definition of $\text{Java}_{\text{PS}}^N$ earlier in the article). Let us represent the k-th operation in sequence L as $L.k$. The precedence of $V_i^3.1$ to $V_i^3.2$ and $V_i^3.3$ is preserved by the transistor rule; the precedence of the first operation to the second in each C_j^{1-3} is preserved by the same variable rule; in C_j^{1-3} and in A^{2-m}, order relations from the first and the second operations to the third operation (in A^1: from the first operation to the second operation) are preserved by the transistor rule.

Lemma 1 *Let $\mathcal{F}$ be an instance of a 3SAT problem and let $\mathcal{V}$ be the instance of the VJC$_{PS}$ problem constructed as described above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. Then, there exists a satisfying assignment for $\mathcal{F}$: $T(v_1),\ldots,T(v_n)$, where $T(v_i) \in \{\text{T, F}\}$. We construct schedule ξ for $\mathcal{V}$ in which, for each i, the first scheduled write to v_i corresponds to the satisfying truth assignment. The schedule ξ is depicted in Figure 1.

The process of constructing schedule ξ from thread sequences representing $\mathcal{V}$ is illustrated in Figure 2.

The reader may verify that this is a legal schedule.

Conversely, suppose that $\mathcal{V}$ is a positive VJC$_{PS}$ instance. If S is a legal schedule for $\mathcal{V}$, then we take as a satisfying assignment the first value written to each v_i. Suppose, on the contrary, that $\mathcal{F}$ is not satisfied. Then there must exist a clause j in $\mathcal{F}$ that is not satisfied:

$$C_j = [v_p, S(v_p)] \vee [v_q, S(v_q)] \vee [v_r, S(v_r)].$$

STEP	OPERATIONS
1	$\mathrm{W}(v_1,T(v_1)),\ldots,\mathrm{W}(v_n,T(v_n))$ from V_i^1 or V_i^2;
2	for $j = 1,2,\ldots,m$, first operation of all sequences C_j^t whose first read is $\mathrm{R}(v_k,T(v_k))$ for some k;
3	$\mathrm{W}(v_1,\overline{T(v_1)}),\ldots,\mathrm{W}(v_n,\overline{T(v_n)})$ from V_i^1 or V_i^2;
4	for $j = 1,2,\ldots,m$, second and third operation of all sequences C_j^t whose first read was $\mathrm{R}(v_k,T(v_k))$ for some k, and first operation of all sequences C_j^t whose first read was $\mathrm{R}(v_k,\overline{T(v_k)})$ for some k;
5	consequently, $A^1,\ldots,A^m$;
6	for all i, first and second operations of all V_i^3;
7	for $j = 1,2,\ldots,m$, second operation of all sequences C_j^t whose first read was $\mathrm{R}(v_k,\overline{T(v_k)})$ for some k and $T(v_k) = \mathrm{T}$;
8	for all i, third operations of all V_i^3;
9	for $j = 1,2,\ldots,m$, second operation of all sequences C_j^t whose first read was $\mathrm{R}(v_k,\overline{T(v_k)})$ for some k and $T(v_k) = \mathrm{F}$;
10	for $j = 1,2,\ldots,m$, third operation of all sequences C_j^t whose first read was $\mathrm{R}(v_k,\overline{T(v_k)})$ for some k.

Figure 1: *Schedule* ξ *of thread sequences representing* $\mathcal{V}$

Since C_j is not satisfied,

the first $\mathrm{W}(v_p,\overline{S_j(v_p)})$ is scheduled before the first $\mathrm{W}(v_p,S_j(v_p))$,
the first $\mathrm{W}(v_q,\overline{S_j(v_q)})$ is scheduled before the first $\mathrm{W}(v_q,S_j(v_q))$,
the first $\mathrm{W}(v_r,\overline{S_j(v_r)})$ is scheduled before the first $\mathrm{W}(v_r,S_j(v_r))$.

Since $\mathcal{V}$ is a positive VJC_{PS} instance, there exists a legal serialization $LS(CR^T)$ for all thread sequences in $\mathcal{V}$. The third operation in A^m may be scheduled in $LS(CR^T)$ only after all operations in $A^1 \ldots A^{m-1}$ are scheduled. Thus, A^j would be scheduled before A^m. Before scheduling A^j, value T must be written to c_j in some of the sequences $C_j^1 \ldots C_j^3$. Thus, we have proved that some o=$\mathrm{W}(c_j,\mathrm{T})$ precedes the third operation in A^m and some sequence (not necessarily consecutive)

$$\begin{array}{c}\mathrm{R}(v_k,S_j(v_k))\\ \mathrm{R}(v_k,\overline{S_j(v_k)})\end{array},$$

where $k \in \{p,q,r\}$, must precede o. Since $\mathcal{V}$ is a positive VJC_{PS} instance having a legal serialization S which preserves the same variable rule (given in the definition of $\mathrm{Java}_{\mathrm{PS}}^N$ earlier in the article), some sequence (not necessarily consecutive)

$$\begin{array}{c}\mathrm{W}(v_k,S_j(v_k))\\ d{=}\mathrm{W}(v_k,\overline{S_j(v_k)})\end{array}$$

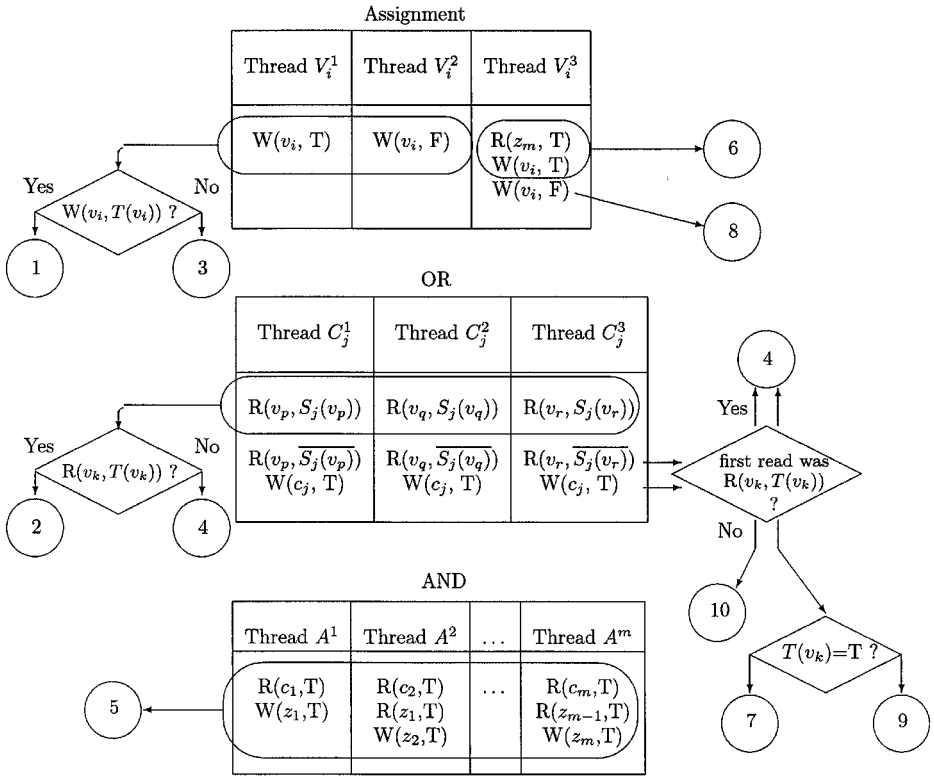


Figure 2: *Constructing schedule* ξ *from thread sequences representing* $\mathcal{V}$

must precede the third operation in A^m. There are two W(v_k, $\overline{S_j(v_k)}$) operations in our construction, the first in either V_k^1 or V_k^2, and the second in V_k^3. Since the first W(v_k, $S_j(v_k)$) succeeds the first W(v_k, $\overline{S_j(v_k)}$), d is in V_k^3. Therefore it succeeds R(z_m,T), and thus it also succeeds all of A^m. But it was shown that d precedes the third operation in A^m. Therefore, d must succeed itself, implying a cycle in S. This contradicts the legality of S □.

The above transformation can be carried out in polynomial time. Indeed, we have a 3SAT instance $\mathcal{F}$ with n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$. This instance may be described by a clause list. It is trivial to generate the variables list from the clause list in time $O(n * m) = O(m^2)$. To generate the VJC$_{\mathbf{PS}}$ instance $\mathcal{V}$ from $\mathcal{F}$, it is sufficient to generate thread groups simulating Assignment, OR and AND, as described previously. The first group is generated directly

from the variables list. Sequences V_i^1, V_i^2 and V_i^3 are produced for each variable v_i; this production requires no additional information other than v_i itself. Thus, this generation takes $O(n)$ time. Similarly, the second and third thread groups may be generated in time $O(m)$.

We conclude that Theorem 1 holds □.

2.2. The VJC_{PS} problem with two locations

Theorem 2 *The VJC_{PS} problem, restricted to instances in which there are only two variables and each sequence contains at most three memory operations, is NP-complete.*

Proof. The proof is similar to that of Theorem 1. We use the reduction from 3-Satisfiability(3SAT).

Consider a 3SAT instance $\mathcal{F}$ with n variables, $v_1, \ldots, v_n$, and m clauses, $C_1, \ldots, C_m$.

In our model we use two variables, a and b. Variable a is used to select a truth setting. It may be assigned some positive integer value. Let F = 0, T = 1.

To simulate an assignment to v_i, we produce, for each variable v_i, the following operations, scheduled in three sequences, V_i^1, V_i^2, V_i^3:

Thread V_i^1	Thread V_i^2	Thread V_i^3
W(a, $i*2$ + T)	W(a, $i*2$ + F)	R(b, T) W(a, $i*2$ + T) W(a, $i*2$ + F)

Consider the order of two writes, W(a, $i*2$ + T) in thread V_i^1 and W(a, $i*2$ + F) in thread V_i^2. Correlate this order to the value assigned to v_i in $\mathcal{F}$: if W(a, $i*2$ + T) was the first, assume that v_i is assigned T; otherwise, v_i is assigned F.

An OR is simulated by having three writes to the same location of the same value; a read can be scheduled after either write.
For each clause $C_j = [v_p, S(v_p)] \vee [v_q, S(v_q)] \vee [v_r, S(v_r)]$, we have three sequences, C_j^1, C_j^2, C_j^3:

Thread C_j^1	Thread C_j^2	Thread C_j^3
R(a, $p*2+S_j(v_p)$) R(a, $p*2+\overline{S_j(v_p)}$) W(b, $j*2$+T)	R(a, $q*2+S_j(v_q)$) R(a, $q*2+\overline{S_j(v_q)}$) W(b, $j*2$+T)	R(a, $r*2+S_j(v_r)$) R(a, $r*2+\overline{S_j(v_r)}$) W(b, $j*2$+T)

Finally, we produce the following sequences, A^1, A^2,..., A^m:

Thread A^1	Thread A^2	...	Thread A^m
	R(b, $m*2+2$)	...	R(b, $m*3$)
R(b, 2+T)	R(b, 4+T)		R(b, $m*2$+T)
W(b, $m*2+2$)	W(b, $m*2+3$)		W(b, T)

This simulates the AND part of the clauses. Indeed, b may be set to value T only after it was previously set to every one of the values $j*2+T$ by some of the sequences C_j^1, C_j^2, C_j^3 (which simulate the OR clause j) in the group.

Note that the first and second write of $i*2$+T or $i*2$+F to a (these writes model an assignment to v_i) will take place prior to the first use of this variable as a basis for writing $j*2$+T to b in some C_j.

The third and fourth write of $i*2$+T or $i*2$+F to a would be scheduled after all AND sequences, because T must first be read from b in V_i^3. Value T may be read from b in V_i^3 only after all AND sequences have ended.

The order of operations in the schedule is supported by the same variable and transistor rule (these rules are given in the definition of $\text{Java}_{\text{PS}}^N$ earlier in the article).

Lemma 2 *Let $\mathcal{F}$ be an instance of a 3SAT problem and let $\mathcal{V}$ be the instance of the VJC$_{PS}$ problem constructed as described above. Then $\mathcal{V}$ is a positive instance if and only if $\mathcal{F}$ is satisfiable.*

Proof. Suppose $\mathcal{F}$ is satisfiable. Then, there exists a satisfying assignment for $\mathcal{F}$: $T(v_1),\ldots,T(v_n)$ where $T(v_i) \in \{\text{T}, \text{F}\}$. We construct the following schedule ξ for $\mathcal{V}$ in which, for each i, the first scheduled write of $i*2$+(one of {T,F}) to a corresponds to the satisfying truth assignment. This schedule is illustrated in Figure 3.

The reader may verify that this is a legal schedule.

Conversely, suppose $\mathcal{V}$ is a positive VJC$_{PS}$ instance. Assume that S is a legal schedule for $\mathcal{V}$. If the first write of the value $i*2$+T to a is completed before the first write of the value $i*2$+F to a, then the satisfying assignment for v_i is T, and otherwise the satisfying assignment for v_i is F. Suppose, contrary to our lemma, that some clause j is unsatisfied:

$$C_j = [v_p, S(v_p)] = \vee[v_q, S(v_q)] \vee [v_r, S(v_r)].$$

Since C_j is not satisfied,

the first W(a, $p*2+\overline{S_j(v_p)}$) is scheduled before the first W(a, $p*2+S_j(v_p)$),
the first W(a, $q*2+\overline{S_j(v_q)}$) is scheduled before the first W(a, $q*2+S_j(v_q)$),
the first W(a, $r*2+\overline{S_j(v_r)}$) is scheduled before the first W(a, $r*2+S_j(v_r)$).

Since $\mathcal{V}$ is a positive VJC$_{PS}$ instance, there exists a legal serialization $LS(CR^T)$ for all thread sequences in $\mathcal{V}$. A^m may be scheduled only after all $A^1 \ldots A^{m-1}$ have been scheduled. Thus, A^j would be scheduled before A^m. Before scheduling the last operation in A^j, the value $j*2$+T must be written to b in some sequence $C_j^1 \ldots C_j^3$. Thus, we have shown that some W(b, $j*2$+T) precedes the last operation in A^m, and some sequence (not necessarily consecutive)

STEP	OPERATIONS AND OPERATION SEQUENCES
1	for all $i = 1,2,\ldots,n$,
	a W(a, $i*2+T(v_i)$) from V_i^1 or V_i^2;
	b then for $j = 1,2,\ldots,m$, first operations of all sequences C_j^t which are R(a, $i*2+T(v_i)$);
2	for all $i = 1,2,\ldots,n$,
	a W(a, $i*2+\overline{T(v_i)}$) from V_i^1 or V_i^2;
	b then for $j = 1,2,\ldots,m$, first operations of all sequences C_j^t which are R(a, $i*2+\overline{T(v_i)}$);
	c then for $j = 1,2,\ldots,m$, second operations of all sequences C_j^t whose first read was R(a, $i*2+T(v_i)$);
3	consequently for all $j = 1,2,\ldots,m$,
	a if $j \neq 1$, R(b, $m*2+j$), the first operation of A^j;
	b for all t such that the first operation of C_j^t reads some $T(v_k)$, W(b, $j*2$+T), the third operation of C_j^t;
	c then R(b, $j*2$+T): if $j \neq 1$, it is the first, otherwise the second operation of A^j;
	d then W(b, $m*2+j+1$): if $j \neq 1$, it is the second, otherwise the third operation of A^j;
4	for all $i = 1,2,\ldots,n$,
	a W(a, $i*2$+T) from V_i^3;
	b then for $j = 1,2,\ldots,m$, second operations of all sequences C_j^t whose first read was $\overline{T(v_i)}$ and it was R(a, $i*2$+F));
5	for all $i = 1,2,\ldots,n$,
	a W(a, $i*2$+F) from V_i^3;
	b then for $j = 1,2,\ldots,m$, second operations of all sequences C_j^t whose first read was $\overline{T(v_i)}$ and it was R(a, $i*2$+T));
6	for $j = 1,2,\ldots,m$,
	third operations of all sequences C_j^t whose first read was $\overline{T(v_i)}$.

Here simulation is finished and dummy processing starts →

Figure 3: *Schedule* ξ

$$\begin{array}{c} \mathrm{R}(a,\ k*2+S_j(v_k)) \\ \mathrm{R}(a,\ k*2+\overline{S_j(v_k)}) \end{array},$$

where $k \in \{p,q,r\}$, must precede this W(b, $j*2$+T). Since $\mathcal{V}$ is a positive VJC$_{PS}$ instance having legal serialization, some sequence (not necessarily consecutive)

$$\begin{array}{c} \mathrm{W}(a,\ k*2+S_j(v_k)) \\ \mathrm{W}(a,\ k*2+\overline{S_j(v_k)}) \end{array}$$

would precede the last operation in A^m. Denote the second operation in this sequence d. There are two W(a, $k*2+\overline{S_j(v_k)}$) operations in our construction, the first in either V_k^1 or V_k^2, and the second in V_k^3. Since the first W(a, $k*2+S_j(v_k)$) succeeds the first W(a, $k*2+\overline{S_j(v_k)}$), d is in V_k^3. Consequently, d succeeds R(b, T), and thus it succeeds A^m as well. According to the above reasoning, d also precedes the last operation in A^m. Thus, d succeeds itself, creating a cycle in S. This contradicts the legality of S □.

The above transformation may be done in polynomial time. The proof of this statement is the same as for Theorem 1. We conclude that Theorem 2 holds □.

2.3. Conclusions

This paper provides a formal study of the complexity of testing the correctness of a shared memory execution in the Java system, using neither prescient stores nor synchronization operations. The study showed that the problem is NP-complete, both in the general case and when it is restricted to instances

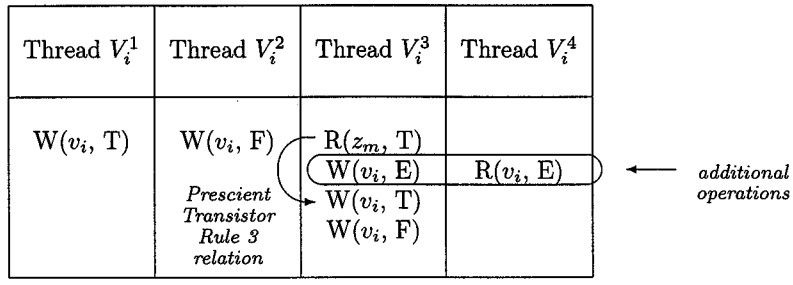
- in which each sequence contains at most three memory operations and each variable occurs in at most four write operations;
- in which there are only two variables and each sequence contains at most three memory operations.

Some interesting questions remain open. First, what about the model with prescient stores? In [3], the model with prescient stores, Java^N, is defined as $LS(CR_{PS}^T)$, where CR_{PS}^T is defined as follows:

> **Causality$_{PS}^T$ Relation**: Let o_1 and o_2 be two operations performed by the same thread p, such that $o_1 \xrightarrow{po} o_2$ (o_1 precedes o_2 in the program order).Then, $o_1 \xrightarrow{CR_{PS}^T} o_2$ if one of the following holds:
> **Same Variable:** o_1 and o_2 access the same variable.
> **Prescient Transistor Rule 1:** the program order of p includes the sub-sequence
> o_1 = READ$^{\swarrow}$X, READ Y,V, READ$^{\swarrow}$Y,W, o_2 = WRITE Y.
> **Prescient Transistor Rule 2:** the program order of p includes the sub-sequence
> o_1 = READ$^{\swarrow}$X, WRITE Y,V, READ$^{\swarrow}$Y,W, o_2 = WRITE Y.
> **Prescient Transistor Rule 3:** the program order of p includes the sub-sequence
> o_1 = READ$^{\swarrow}$X, WRITE$_{\searrow}$Y, o_2 = WRITE Y.
> Here READ$^{\swarrow}$X means read a value written at another thread, and WRITE$_{\searrow}$X means write a value seen at another thread.

The *transistor rule* may be simply converted to *Prescient Transistor Rule 3* by inserting an artificial additional WRITE$_{\searrow}$ in the construction for reduction from 3SAT.

For example, sequences for simulating assignments from Theorem 1 (see page 4) may be transformed to the following:



We thus suppose that testing this model is NP-complete. However, we suppose that additional simplicity restrictions, which we may apply to this model without breaking NP-completeness, would be weaker: in the example above, a new sequence and new operations were added.

The second issue is synchronization. Verifying memory consistency for the model with synchronization operations may not be simpler than verifying it for the model with prescient stores. In fact, it may be harder. This is because, for traces without synchronization operations, the two models are equivalent. The memory model that corresponds to the code which employs volatile variables only is proved to be sequentially consistent [3]. Thus, verifying memory consistency for this model may be no less complex than verifying sequential consistency (described in [2]).

The third issue is the designing of an appropriate verification algorithm. There are some reasons to suppose that the following is not impossible:

(a) The $\text{Java}_{\text{PS}}^{N}$ model may be verifiable in polynomial time while restricting the number of threads.
(b) The model with synchronization operations may be verifiable in polynomial time while restricting both the number of threads and the number of volatile variables (this may be acceptable for real systems).

In this paper we have used some ideas and constructions from the study of the Sequential Consistency model in [2], where it was proved that the problem of verifying Sequential Consistency for shared memory executions is NP-complete while restricted to instances in which

(a) each sequence contains at most two memory operations and each variable occurs in at most two write operations;
(b) there are only two variables;
(c) there are only three processors.

In [2], the authors mentioned that proving the third case is harder than proving the first and second, because "for a small fixed number of processors ... we do not have as much freedom to schedule operations in an arbitrary order." Therefore, for this case the authors have made a reduction from POSITIVE ONE-IN-THREE 3SAT

[1] instead of the usual 3SAT. Our attempts to prove NP-completeness for VJC_{PS} in a similar way have failed. Proving NP-completeness for VJC_{PS} seems to be a more difficult task than proving NP-completeness for the problem of verifying sequential consistency. This is because in Java not all order relations are preserved in the schedule, and this complicates reduction.

For volatile variables, the Java model was found to be sequentially consistent[3]. However, it was not proved in [2] that verifying sequential consistency is NP-complete while both the number of processors and variables are restricted.

References

[1] M. R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[2] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, August 1997.

[3] A. Gontmakher and A. Schuster. Non-operational characterizations for Java Memory Behavior. *ACM Transactions On Computer Systems (TOCS)*, 18(4):333–386, November 2000.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[5] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition, 1999.