An Operational Happens-Before Memory Model

Yang Zhang and Xinyu Feng

University of Science and Technology of China

Abstract. Happens-before memory model (HMM) is used as the basis of Java memory model (JMM). Although HMM itself is simple, some complex axioms have to be introduced in JMM to prevent the causality loop, which causes absurd out-of-thin-air reads that may break the type safety and security guarantee of Java. The resulting JMM is complex and difficult to understand. It also has many anti-intuitive behaviors, as demonstrated by the "ugly examples" by Aspinall and Ševčík [3]. Furthermore, HMM (and JMM) specify only what execution traces are acceptable, but say nothing about how these traces are generated. This gap makes it difficult for static reasoning about programs.

In this paper we present OHMM, an operational variation of HMM. The model is specified by giving an operational semantics to a language running on an abstract machine designed to simulate HMM. Thanks to its generative nature, the model naturally prevents out-of-thin-air reads. On the other hand, it uses a novel replay mechanism to allow instructions to be executed multiple times, which can be used to model many useful speculations and optimization. The model is weaker than JMM for lockless programs, thus can accommodate more optimization, such as the reordering of independent memory accesses that is not valid in JMM. Program behaviors are more natural in this model than in JMM, and many of the anti-intuitive examples in JMM are no longer valid here. We hope OHMM can serve as the basis for new memory models for Java-like languages.

1 Introduction

A memory model of a programming language specifies how memory accesses are made during program execution. It serves as a contract between programmers and the language implementation. The most well-known model is the sequential consistency (SC) model proposed by Lamport[9]. It requires that one memory operation is executed at a time, and the operations issued from each thread are executed following their orders in the program (a.k.a. program-order).

However, the idealized SC model is too expensive to implement in practice, which prevents useful optimizations in hardware and compilers. The optimizations are designed to preserve behaviors of sequential programs, but may produce unexpected behaviors in a concurrent setting [4]. For instance, in the following program, we use x, y to represent shared (non-volatile) variables, and r for thread-local variables (registers). It is impossible to get the result $r_1 = r_2 = 0$ under the SC model, but the result could be produced if the compiler decides to flip lines 1 and 2 since they have no data dependency.

Example 1. Initially $x = y = 0$.

$$\begin{array}{ll|ll} 1: & x := 1; & 3: & y := 1; \\ 2: & r_1 := y; & 4: & r_2 := x; \end{array}$$

Result: $r_1 = r_2 = 0$?

Models allowing this kind of optimizations are called relaxed memory models. Many such models have been proposed for computer architectures to allow optimization in processors [1]. For programming languages, their memory models could be more complex since they have to reflect optimization both in compilers and in processors. In general, the memory model of programming languages should satisfy the following requirements:

– The model is usable by programmers. This means it should satisfy DRF-guarantee, which says Data-Race-Free programs have the same behaviors in this model as those in the SC model.
– The model cannot be too strong to prohibit important optimization techniques, especially those already used heavily in existing compilers. The weaker the model is, the more optimization it allows.
– Following the above two requirements, ideally the model should allow any behaviors of programs with races, and guarantee SC behaviors of DRF programs. However, for Java-like type safe and secure languages, we may want racy programs to be safe and secure too. This means the model cannot be too weak for racy programs. For instance, it should not produce out-of-thin-air values.

It is very challenging to define a memory model satisfying all the requirements. The common practice (including this work) is to make the set of rules defining the model as simple as possible, but at the same time being able to simulate the program behaviors under versatile optimization techniques. Sometimes the simulation of the behaviors has little to do with their cause in the real world, thus looks ad-hoc.

Java uses a happens-before memory model (HMM) as the basis for JMM. The basic HMM is very simple and weak, which satisfies the second requirement well, but its causality circle (which we will explain in Sec. 2) generates out-of-thin-air values and breaks the type-safety and security requirement. It also breaks the DRF-Guarantee. To avoid the causality circle, JMM introduces 9 axioms to further constrain the acceptable execution traces [12]. They are known as the most complex part of the model. The intuition behind these axioms is difficult to understand. Also, due to the non-generative nature of the model, the link between programs and their legal execution traces is missing, which means it is difficult to infer program behaviors in this model by looking at the code statically. Others also point out that JMM fails to permit some behaviors that should be allowed [5], and it has many anti-intuitive features, as demonstrated by the "ugly examples" by Aspinall and Ševčík [3].

In this paper we present OHMM, an operational variation of HMM. The model is specified by giving an operational semantics to a language running on an abstract machine. This operational approach shows how programs are executed line-by-line. It makes many hidden details in HMM (and JMM) explicit, such as register (or local variable) dependency and control dependency. The model satisfies the aforementioned three requirements. Its generative nature prevents the class of causality circle that generates out-of-thin-air values and breaks DRF-guarantee. On the other hand, it uses a novel replay mechanism to allow instructions to be executed multiple times, which can be used to simulate many useful speculation and optimization. Our model satisfies DRF-guarantee, but is weaker than JMM for programs with no locks or volatile variables. Many of the anti-intuitive examples in JMM [3] would not show in our model. We also prove the validity (semantics preservation) of a class of program transformations in our model, many of which are not valid in JMM.

We want to emphasize that OHMM is a new memory model that is not compliant with JMM, as we mentioned above. The main focus of this work is to explore the use of the replay mechanism to simulate speculation operationally, which makes the model weak enough but can naturally avoid out-of-thin-air values and some anti-intuitive features of JMM. To be focused, we formulate the model using a simple imperative language and ignore many language features of Java, such as object initialization, final fields and I/O. Although we hope the idea can serve as the basis for the next generation memory models for Java-like languages, the model in its current form is far from ready to serve as a replacement of JMM.

In the rest of this paper, we give an overview of HMM in Sec. 2. Then we introduce our abstract machine and OHMM informally in Sec. 3, and present the model formally as the operational semantics of the machine in Sec. 4. In Sec. 5 we show more examples to explain the model, with a detailed comparison with JMM at the end. We discuss related works and conclude in Sec. 6. In the appendix we define and prove the DRF-guarantee, and prove the semantics preservation of a class of program transformations in our model.

2 An Overview of HMM

In HMM, a program execution is modeled as a set of memory access events and some orders between them, including the program order and the synchronization order. The program order $\xrightarrow{po}$ is a total order among events executed by the same thread. It reflects the sequence of events generated by a sequential thread following the program text. The synchronization order $\xrightarrow{so}$ is a total order among all synchronization events, which are acquire/release of locks and read/write of volatile variables. $\xrightarrow{so}$ needs to be consistent with $\xrightarrow{po}$, that is, synchronization events from the same thread must be ordered in $\xrightarrow{so}$ the same way as in $\xrightarrow{po}$.

The synchronization-with order $\xrightarrow{sw}$ can be derived as a relation between a release of a lock and the next acquire event (following $\xrightarrow{so}$) of the same lock, or a write of a volatile variable and the next read (following $\xrightarrow{so}$) of the same variable. So we know $\xrightarrow{sw}$ is a partial order and a subset of $\xrightarrow{so}$. We demonstrate $\xrightarrow{po}$ and $\xrightarrow{sw}$ in Fig. 1.

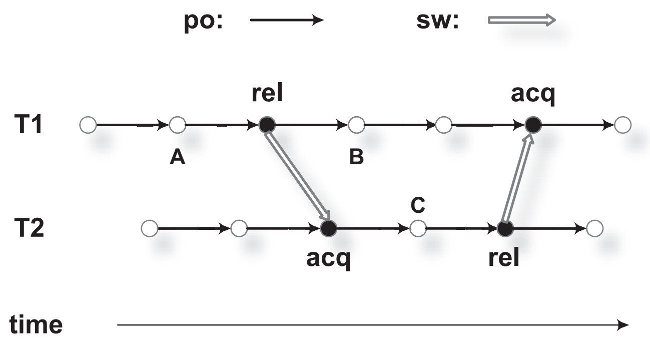

Fig. 1. Happens-Before Order

The happens-before order $\xrightarrow{hb}$ is then defined as the transitive closure of the union of $\xrightarrow{po}$ and $\xrightarrow{sw}$. In Fig. 1, we have $A \xrightarrow{hb} B$ and $A \xrightarrow{hb} C$, but not $B \xrightarrow{hb} C$ or $C \xrightarrow{hb} B$. In HMM, a read r can see the write w that immediately happens before it (that is, $w \xrightarrow{hb} r$ and $\neg\exists w'.\, w \xrightarrow{hb} w' \xrightarrow{hb} r$), or any write w that is not happens-before ordered with r ($\neg(w \xrightarrow{hb} r \vee r \xrightarrow{hb} w)$). We say a program is data-race-free (DRF) if, for every execution, a read can only see the write that immediately happens before it.

HMM is a very weak model. It allows the behavior in Example 1. Line 2 can get the value written by line 3 since they are not happens-before ordered. For the same reason line 4 can get the value written by line 1.

Problems with HMM. The obvious problem with HMM and other axiomatic memory models is that they only define what are acceptable executions. This is useful for dynamic testing, but not good for static reasoning because of the gap between programs and their executions. In other words, we cannot tell how program behaves in these models just by looking at the code.

A more serious problem is the causality circle. The happens-before order was originally introduced by Lamport [8] to describe the causality relations between actions in message-passing based distributed systems. For racy programs in the shared-memory model, the order fails to capture the de-facto causality between a write from one thread and the following read of it from another. The following examples by Manson et al. [12] show the problem.

Example 2. Initially $x = y = 0$.

```
1 :  r1 := x;           ||  4 :  r2 := y;
2 :  if (r1 ≠ 0)        ||  5 :  if (r2 ≠ 0)
3 :      y := 42;       ||  6 :      x := 42;
        Result: r1 = r2 = 42?
```

```
1 :  r1 := x;   ||  3 :  r2 := y;
2 :  y := r1;   ||  4 :  x := r2;
        Result: r1 = r2 = 42?
```

Both results in the above examples are allowed in HMM. The first example shows HMM does not have DRF-guarantee because the program is race-free. In SC semantics, the left thread could not access y, neither would the right thread access x. However, the result is possible in HMM because of a self-justifying causality circle: we speculate that line 3 will be executed, then line 4 gets 42, then we execute lines 6 and 1, which makes $r_1 = 42$ and justifies our speculation. The second example shows HMM allows out-of-thin-air read. The result is allowed for a similar reason.

To solve these problems, JMM introduces causality requirements to define valid executions [12]. As pointed out by many researchers, the resulting model is difficult to understand [2, 6], and is not completely satisfactory either [5, 3].

3 Informal Development of OHMM

In this section, we give a detailed semi-formal presentation of our memory model, including the language, the model of an abstract machine, and how code is executed on it. The model is formally defined as operational semantics of the machine in the next section.

3.1 The Language

$$
\begin{array}{rl}
(\mathit{Number}) & n \;\in\; \mathit{Integer} \\
(\mathit{NormVar}) & x, y, z, \ldots \\
(\mathit{VolVar}) & v, v_1, v_2, v_3, \ldots \\
(\mathit{Lock}) & l \;::=\; l_0 \mid l_1 \mid l_2 \mid \ldots \\
(\mathit{Reg}) & r \;::=\; r_0 \mid r_1 \mid r_2 \mid \ldots \\
(\mathit{Expr}) & E ::= r \mid n \mid op(E_1, ..., E_n) \\
(\mathit{Instr}) & \iota \;::=\; \iota_n \mid \iota_s
\end{array}
\qquad
\begin{array}{rrcl}
(\mathit{NonSyncI}) & \iota_n & ::= & x := r \mid r := x \\
 & & \mid & r := E \mid x := n \\
(\mathit{SyncI}) & \iota_s & ::= & v := r \mid r := v \mid v := n \\
 & & \mid & \textbf{lock}\ l \mid \textbf{unlock}\ l \\
(\mathit{Stmts}) & C & ::= & \iota \mid \textbf{skip} \mid C; C \\
 & & \mid & \textbf{if}\ r\ \textbf{then}\ C\ \textbf{else}\ C \\
 & & \mid & \textbf{while}\ r\ \textbf{do}\ C \\
(\mathit{ThrdID}) & \mathit{tid} & \in & \mathit{Nat} \\
(\mathit{Program}) & P & ::= & \mathit{tid}.C \mid \mathit{tid}.C \,\|\, P
\end{array}
$$

Fig. 2. Syntax of the language

Syntax of the language is presented in Fig. 2. A program P consists of one or more sequential threads. Each thread has a thread id *tid* and a statement C. A statement may be a primitive instruction ι, a **skip**, or composition of them. Primitive instructions are classified into synchronization instructions (ι_s) or normal ones (ι_n). Normal instructions include accesses of non-volatile variables or pure instructions ($r := E$). Accesses of volatile variable and acquire/release of locks are synchronization instructions. We use r to represent registers (thread-local variables), x, y and z for shared non-volatile variables, and v_1 and v_2 for volatile ones. An expression E is a mathematical operation over constants and registers. We say $r := E$ is pure since it does not access shared variables.

3.2 The Abstract Machine

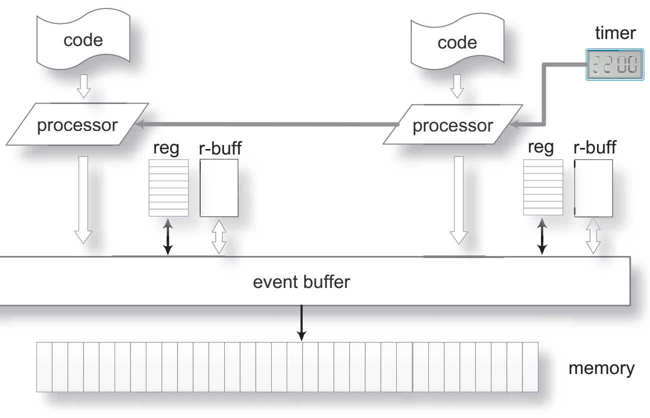

Fig. 3. Design of the abstract machine

We demonstrate the abstract machine model in Fig. 3. Comparing with the ideal SC model, ordering of memory reads and writes on this machine can be relaxed by three important constructs: the event buffer, the history-based memory, and the thread-local replay buffers ("r-buff" in Fig. 3). Processors run threads and issue events. Events are put in the event buffer, which allows us to relax the execution order between events with no data dependency. The history-based memory keeps all the historical values written to each normal variable, so there may be more than one values visible to each memory read. This further relaxes the model. We also allow a thread to execute an instruction multiple times by replaying the corresponding event. When an event is executed, it could be duplicated, put into the thread-local replay buffer, and executed a second time later. This allows us to simulate speculation or program analysis in compilers.

Here we want to emphasize that this is an abstract machine designed to simulate relaxed behaviors of programs only. We do not intend to use it to faithfully model real-world hardware or software optimization.

Events and Event Buffer. Each thread in the program P runs on a processor of the machine. Execution of the threads follows the standard interleaving semantics (as in SC model). However, when an instruction is executed, the effect does

$$
\begin{array}{rrcl}
(\mathit{Timer}) & t & \in & \mathit{Nat} \\
(\mathit{TStamp}) & ts & ::= & \langle \mathit{tid}, t\rangle \mid \textbf{init} \\
(\mathit{Event}) & e & ::= & \langle ts, \iota\rangle \\
(\mathit{EvtBuff}) & b, rb & ::= & \{e_0, \ldots, e_n\} \\
(\mathit{RegFile}) & \mathit{rf} & ::= & \{r_0 \rightsquigarrow n_0, \\
 & & & \ldots, r_k \rightsquigarrow n_k\} \\
(\mathit{ThrdQ}) & \mathit{TQ} & ::= & \{\mathit{tid}_0 \rightsquigarrow (\mathit{rf}_0, rb_0), \\
 & & & \ldots, \mathit{tid}_k \rightsquigarrow (\mathit{rf}_k, rb_k)\} \\
(\mathit{Viewed}) & \mu & ::= & \textbf{true} \mid \textbf{false} \\
(\mathit{WtOpr}) & wv & ::= & \langle ts, n, \mu\rangle
\end{array}
\qquad
\begin{array}{rrcl}
(\mathit{SyncAct}) & \mathit{syn} & ::= & \langle ts, \textbf{st}, v\rangle \mid \langle ts, \textbf{ld}, v\rangle \\
 & & & \mid \langle ts, \textbf{rel}, l\rangle \mid \langle ts, \textbf{acq}, l\rangle \\
(\mathit{HistOpr}) & o & ::= & wv \mid \mathit{syn} \\
(\mathit{History}) & h & ::= & \{o_0, \ldots, o_n\} \\
(\mathit{Mem}) & m & ::= & \{x \rightsquigarrow h_1, y \rightsquigarrow h_2, \ldots, \\
 & & & v_1 \rightsquigarrow n_1, v_2 \rightsquigarrow n_2, \ldots\} \\
(\mathit{LockSet}) & L & ::= & \{l_0 \rightsquigarrow \mathit{tid}_0, \\
 & & & \ldots, l_k \rightsquigarrow \mathit{tid}_k)\} \\
(\mathit{State}) & \sigma & ::= & (\mathit{TQ}, m, b, t, L)
\end{array}
$$

Fig. 4. Model of the abstract machine

not take place immediately. Instead, the processor issues a corresponding *event* and puts it into the global *event buffer*. As shown in Fig. 4, the event buffer b is modeled as a set of events. An event e is a pair $\langle ts, \iota\rangle$. It wraps the instruction ι with a timestamp ts recording when and by whom it is issued. A timestamp ts is a pair consisting of a thread id and a logical time t. The latter is a global counter shared by all processors (see Fig. 3). It increases when an event is put into the event buffer. Below we use the dot notation $ts.\mathit{tid}$ and $ts.t$ to refer to the first and second element of ts, respectively. There is also a special timestamp **init**, which represents the time when the machine configuration is initialized.

With timestamps, we could tell if two events are issued by the same thread, and, if yes, which one is issued earlier. We say $ts < ts'$ if they have the same thread id and the logical time of ts is smaller than ts'. **init** is smaller than all other timestamps. The formal definition is given in Fig. 5.

Thread Local Data and the Thread Queue. Each thread has a register file rf ("reg" in Fig. 3), which maps register names to integer values. It also has a local replay buffer rb, which will be explained later in Sec. 3.5. The thread queue TQ is defined as a mapping from thread id to its local state.

History-Based Memory. The shared memory maps variable names to values. We model volatile and non-volatile variables differently. A volatile memory cell contains the value stored at the variable only. For the non-volatile one, we keep all the historical write events. A write to this variable does not overwrite previous values. Instead, we put a new write action into the memory cell, which is a history h containing write or synchronization actions.

A write action is a triple $\langle ts, n, \mu\rangle$. It records the timestamp and the written value n. The boolean flag μ records if this write has been seen by other threads (so it is initially **false**). It is used to replay write events, which we will explain in Sec. 3.5. Synchronization actions syn include acquire/release of locks and load/store of volatile memory cells.

$$ts_1 < ts_2 \overset{\text{def}}{=} ts_1 = \textbf{init} \land ts_2 \neq ts_1 \lor ts_1.tid = ts_2.tid \land ts_1.t < ts_2.t$$

$$UseR(E) \overset{\text{def}}{=} \begin{cases} \{r\} & \text{if } E = r \\ \emptyset & \text{if } E = n \\ \bigcup_{i\in[1..n]} UseR(E_i) & \\ \quad \text{if } E = op(E_1, \ldots, E_n) & \end{cases} \qquad UseR(\iota) \overset{\text{def}}{=} \begin{cases} \{r\} & \text{if } \iota = (\_ := r) \\ UseR(E) & \text{if } \iota = (\_ := E) \\ \emptyset & \text{otherwise} \end{cases}$$

$$UpdR(\iota) \overset{\text{def}}{=} \begin{cases} \{r\} & \text{if } \iota = (r := \_) \\ \emptyset & \text{otherwise} \end{cases} \qquad UpdR(rb) \overset{\text{def}}{=} \bigcup_{e\in rb} UpdR(e.\iota)$$

$$UseM(\iota) \overset{\text{def}}{=} \begin{cases} \{x\} & \text{if } \iota = (r := x) \\ \emptyset & \text{otherwise} \end{cases} \qquad UpdM(\iota) \overset{\text{def}}{=} \begin{cases} \{x\} & \text{if } \iota = (x := \_) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\begin{aligned} e_1 \overset{\mathbf{r}}{\leftarrow} e_2 &\overset{\text{def}}{=} e_1.ts < e_2.ts \land (UseR(e_1.\iota) \cap UpdR(e_2.\iota) \neq \emptyset \lor UseR(e_2.\iota) \cap UpdR(e_1.\iota) \neq \emptyset \\ &\qquad \lor UpdR(e_1.\iota) \cap UpdR(e_2.\iota) \neq \emptyset) \\ e_1 \overset{\mathbf{b}}{\leftarrow} e_2 &\overset{\text{def}}{=} e_1.ts < e_2.ts \land (e_2.\iota = (\textbf{unlock}\ \_) \lor e_2.\iota = (v := r) \lor e_1.\iota = (r := v)) \\ e_1 \overset{\mathbf{m}}{\longleftarrow} e_2 &\overset{\text{def}}{=} e_1.ts < e_2.ts \land UpdM(e_1.\iota) \cap UseM(e_2.\iota) \neq \emptyset \\ e_1 \overset{\mathbf{s}}{\leftarrow} e_2 &\overset{\text{def}}{=} e_1.ts.t < e_2.ts.t \land e_1.\iota \in SyncI \land e_2.\iota \in SyncI \\ e_1 \leftarrow e_2 &\overset{\text{def}}{=} (e_1 \overset{\mathbf{r}}{\leftarrow} e_2) \lor (e_1 \overset{\mathbf{b}}{\leftarrow} e_2) \lor (e_1 \overset{\mathbf{m}}{\longleftarrow} e_2) \lor (e_1 \overset{\mathbf{s}}{\leftarrow} e_2) \end{aligned}$$

$$readyR(ts, r, b) \overset{\text{def}}{=} \neg\exists(e \in b).\, e.ts < ts \land r \in UpdR(e.\iota)$$

Fig. 5. Dependency between events, and auxiliary definitions

The whole machine state σ consists of a thread queue TQ, memory m, event buffer b, timer t, and lock set L. L maps a lock to the id of the owner thread.

3.3 Execution Order of Events

Events in the event buffer do not have to be executed following the program order. They can be executed any time as long as the following dependency requirements are satisfied (see Fig. 5 for the formal definitions).

- Register dependency ($e_1 \overset{\mathbf{r}}{\leftarrow} e_2$). Event e_2 must wait for the execution of an earlier event e_1 if one of them reads or updates a register being updated by another. In Fig. 5, we use $UseR(\iota)$ and $UpdR(\iota)$ to represent the set of registers read or updated by ι respectively.
- Memory dependency ($e_1 \overset{\mathbf{m}}{\longleftarrow} e_2$). A read e_2 must wait for an earlier write e_1 if e_2 reads the variable being updated by e_1. In Fig. 5 we use $UseM(\iota)$ and $UpdM(\iota)$ to represent the set of non-volatile variables read or updated by ι.
- Barriers ($e_1 \overset{\mathbf{b}}{\leftarrow} e_2$). Memory accesses must wait for earlier lock-acquire events or read of volatile variables. Release of lock or write of volatile variables must wait for all earlier memory accesses.
- Synchronization order ($e_1 \overset{\mathbf{s}}{\leftarrow} e_2$). The execution of synchronization events (acquire/release of locks and read/write of volatile variables) must follow the

order in which they are put into the event buffer, no matter whether they are issued by the same thread or not. This order explains why we need the timer t to be global in σ.

It may look strange that $\_ \overset{\mathbf{m}}{\longleftarrow} \_$ does not ask write events to wait for earlier reads or writes. We will explain this below when we introduce the history-based memory and the execution of memory accesses.

With the event buffer, we allow the result shown in Example 1. The following events could be issued following the interleaving semantics.

$$\langle\langle tid_1, 0\rangle, x := 1\rangle, \quad \langle\langle tid_2, 1\rangle, y := 1\rangle \quad \langle\langle tid_2, 2\rangle, r_2 := x\rangle, \quad \langle\langle tid_1, 3\rangle, r_1 := y\rangle$$

We could get the result by executing events in the order of $2-3-0-1$, where the number refers to the logical time of the corresponding event.

In all the examples below, we follow the convention that the initial values of all memory cell are 0. That is, for all non-volatile variable x, we have $\langle \mathbf{init}, 0, \mathbf{true}\rangle \in m(x)$.

Example 3.

$$\begin{array}{ll|l} 1: & x := 1; & 3: \; r_1 := v_1; \\ 2: & v_1 := 1; & 4: \; \mathbf{if}\ (r_1) \;\; r_2 := x; \end{array}$$

Result: $r_1 = 1$ and $r_2 = 0$? Disallowed!

This is because the event e_2 generated from line 2 cannot be executed earlier than e_1 from line 1, since $e_1 \overset{\mathbf{b}}{\longleftarrow} e_2$. Similarly, line 4 depends on line 3. Therefore, line 1 must have been executed when line 3 reads value 1. This is actually a data-race-free program.

3.4 Histories and Memory Accesses

The reordering of events allows us to produce many relaxed behaviors already. However, it is not weak enough if we use a standard model of memory where each memory cell contains only the most recently written value. The following example shows why it is useful to keep the history of all memory updates.

Example 4 (Taken from [11].).

$$\begin{array}{ll|l} 1: & x := 1; & 3: \; x := 2; \\ 2: & r_1 := x; & 4: \; r_2 := x \end{array}$$

Result: $r_1 = 2$ and $r_2 = 1$?

The result is allowed in JMM, but cannot be produced by reordering since we cannot reorder events from the same thread due to dependency $\_ \overset{\mathbf{m}}{\longleftarrow} \_$. Below we introduce history into the model and explain how memory cells are accessed.

Writes of non-volatile variables. For a write $\langle ts, x := r\rangle$, we simply put the write action $\langle ts, n, \mathbf{false}\rangle$ into the history $m(x)$, where n is the value of r if it is ready to use (see $readyR(ts, r, b)$ in Fig. 5). The flag **false** means this write has not been seen by other threads. We will explain its use later.

$$AddSyn(m, syn) \stackrel{\text{def}}{=} \lambda x. \begin{cases} h \cup \{syn\} & \text{if } m(x) = h \\ n & \text{if } m(x) = n \end{cases}$$

$$o_1 \prec^{\text{po}} o_2 \stackrel{\text{def}}{=} o_1.ts < o_2.ts$$

$$o_1 \prec^{\text{sw}} o_2 \stackrel{\text{def}}{=} o_1.ts.t < o_2.ts.t$$
$$\wedge(\exists v.\, o_1 = \langle \_, \mathbf{st}, v\rangle \wedge o_2 = \langle \_, \mathbf{ld}, v\rangle \vee \exists l.\, o_1 = \langle \_, \mathbf{rel}, l\rangle \wedge o_2 = \langle \_, \mathbf{acq}, l\rangle)$$

$$o_1 \prec^{\text{hb}}_h o_2 \stackrel{\text{def}}{=} (o_1, o_2) \in ((\prec^{\text{po}} \cup \prec^{\text{sw}}) \cap (h \times h))^+$$

$$ts \prec^{\text{hb}}_h o \stackrel{\text{def}}{=} \exists n, \mu.\, \langle ts, n, \mu\rangle \prec^{\text{hb}}_{h \cup \{\langle ts,n,\mu\rangle\}} o \qquad o \prec^{\text{hb}}_h ts \stackrel{\text{def}}{=} \exists n, \mu.\, o \prec^{\text{hb}}_{h \cup \{\langle ts,n,\mu\rangle\}} \langle ts, n, \mu\rangle$$

$$visible(ts, wv, h) \stackrel{\text{def}}{=} wv \prec^{\text{hb}}_h ts \wedge \neg\exists wv'.\, wv \prec^{\text{hb}}_h wv' \wedge wv' \prec^{\text{hb}}_h ts \vee \neg(wv \prec^{\text{hb}}_h ts \vee ts \prec^{\text{hb}}_h wv)$$

Fig. 6. More auxiliary definitions

Reads and writes of volatile variables. We only keep the most recent value of a volatile variable. Reads get the value, and writes overwrite it. However, since accesses of volatile memory are synchronization operations, we record the actions ($\langle ts, \mathbf{st}, v\rangle$ or $\langle ts, \mathbf{ld}, v\rangle$) in the history of every non-volatile variable (see $AddSyn(m, syn)$ in Fig. 6 for the formal definition). Similarly, we also record every acquire and release of locks in every history.

Reads of non-volatile memory. To show how read of non-volatile memory works, we first define in Fig. 6 the happens-before order $\prec^{\text{hb}}_h$, which is the transitive closure of the union of program order $\prec^{\text{po}}$ and synchronizes-with order $\prec^{\text{sw}}$, with the extra requirement that only actions in h are ordered. Note the happens-before order, the program order and the synchronizes-with order here share the same name and intuition with those in HMM explained in Sec. 2, but the definitions are not identical. Then we overload $\prec^{\text{hb}}_h$ to represent that an action o in h happens before ts ($o \prec^{\text{hb}}_h ts$) or the inverse ($ts \prec^{\text{hb}}_h o$).

A read issued at ts can get the value of any write wv visible in h, *i.e.*, $visible(ts, wv, h)$ holds. Defined in Fig. 6, the visibility requires that wv is the most recent write that happens before ts, or wv and ts are not happens-before ordered. If a write action $\langle ts, n, \_\rangle$ is seen by a read from a thread different from $ts.tid$, we mark the μ field of the write with **true**, so the write action in the history becomes $\langle ts, n, \mathbf{true}\rangle$.

Now we can see the result in Example 4 is allowed. We execute the writes at lines 1 and 3 first. The read at line 2 can see both writes. The write at line 1 happens before it, and there is no happens-before relation between this read and the write at line 3. Similarly both writes are visible to line 4.

Example 5. In the following sequential program, we could execute the second command first, without affecting the final result ($r_1 = 0$).

$$r_1 := x;\ x := 1;$$

This is because the read event has smaller timestamp than the write, since events are issued following the program order. Even if we execute the write first, the value 1 is not visible to the read, which only sees the initial value 0.

3.5 Replay of Events

Many compiler optimizations are based on results of program analysis. The resulting relaxed behaviors cannot be simulated by the machine we have so far.

Example 6 (Adapted from [12].).

```
1 :  r1 := x;             ||
2 :  r2 := r1;            ||  6 :  r4 := y;
3 :  r3 := (r1 == r2);    ||  7 :  x := r4;
4 :  if (r3)              ||
5 :       y := 42         ||
```

Result: $r_1 = r_2 = r_4 = 42$? It should be allowed since the compiler may realize the test at line 4 is always true and line 5 must be executed. Then line 5 can be executed before lines 1 and 2 since there is no dependency. In our machine, we must execute lines 1, 2, and 3 before 5 because of the register dependency, thus the result cannot be generated.

To simulate the program transformation made by the compiler, we notice that the compiler needs to first scan the first three lines before it decides the test at line 4 is always true, then it does the code transformation and reorders lines 1 and 2 with line 5. We could simulate the transformation by duplicating lines 1 and 2 and put the extra copy below line 5:

```
                               4 :   if (r3)
1 :   r1 := x;                 5 :       y := 42
2 :   r2 := r1;                1′ :  r1 := x;
3 :   r3 := (r1 == r2);        2′ :  r2 := r1;
```

The sequential behavior of the resulting thread is unchanged. We are using the first copy of lines 1 and 2 to simulate the static analysis pass, and the second copy (lines $1'$ and $2'$) for the real execution after reordering with line 5.

Based on this observation, we allow an event to be executed multiple times by putting it into a *thread-local replay buffer* when it is executed. Later we can move it back from the replay buffer to the event buffer, so that it can be executed a second time. Note that we do not change the timestamp of the duplicated event. Therefore, in Example 5, even if we duplicate the read event and replay it after the execution of the write, the value of r_1 can only be 0, for the same reason explained in Example 5. However, the following example shows unrestricted replay may change the sequential behavior of threads (thus break the DRF-guarantee) or produce out-of-thin-air values.

Example 7.

```
                    1 :  r1 := 1;        1 :  r1 := x;        || 4 :  r2 := x;
1 :  r1 := 1;       2 :  r2 := r1;       2 :  r1 := r1 + 1;   || 5 :  r2 := r2 + 1;
2 :  r1 := 2;       3 :  r1 := 2;        3 :  x := r1;        || 6 :  x := r2;
                                                              || 7 :  r3 := x;
```

(a) $r_1 = 1$? (b) $r_2 = 2$? (c) $r_3 = 3$?

In Example 7 (a), the result is allowed by replaying line 1 only. In (b), the result is allowed by replaying line 2 but not line 1. In (c), we can get the result by replaying line 1, 2 and 3 after we execute lines 1 to 6 sequentially. The result 3 is out-of-thin-air and should be disallowed.

To address this problem, we need to enforce two principles when replaying events. First the replay cannot change sequential behaviors of programs (to forbid Example 7 (a) and (b)). Second, a write could be replayed only if its value has not been seen by other threads at the time of replay. This is to forbid Example 7 (c). Technically, we need to follow the rules below:

– If event e reads or writes registers that are updated by an event e' in the replay buffer rb, e must be replayed too. The dependency is formulated as $(\mathit{UseR}(e.\iota) \cup \mathit{UpdR}(e.\iota)) \cap (\mathit{UpdR}(rb)) \neq \emptyset$, where $\mathit{UpdR}(rb)$ is the set of registers updated by events in rb, as defined in Fig. 5. In Example 7(a), if the event generated by line 1 is in rb when we execute line 2, we must also replay line 2. Therefore it is impossible to get $r_1 = 1$ at the end.
– If event e uses register r (*i.e.*, $r \in \mathit{UseR}(e.\iota)$), but the preceding event that sets the value of r is not in the replay buffer (*i.e.*, $r \notin \mathit{UpdR}(rb)$), we must *not* replay e. Otherwise, the replay of e may see updates of r by subsequent instructions, which changes sequential behavior of programs. In Example 7(b), if we replay line 2 but not line 1, the duplicate of line 2 may see the update of r_1 by line line 3. This rule prevents this from happening.
– If neither of the two conditions holds, we could execute e and decide nondeterministically whether to put it into the replay buffer rb or not.
– If both of the first two conditions hold, execution of e is stuck until one of them becomes false. Readers who want to see a formal definition of all these constraints could refer to the definition of $\mathit{Replay}(rb, e, rb')$ in Fig. 8.
– If a memory write is put into rb, it can be executed a second time only if the previous written value has not been seen by other threads, that is, the flag μ of the write action is **false**. In Example 7(c), we may execute line 3 (which writes 1), put it into rb, and then execute line 4 (which sees the written value 1). Then the duplicate of line 3 in rb cannot be executed again because the old write has been seen by a different thread (thus its flag μ becomes **true**). This prevents the out-of-thin-air result $r_3 = 3$. This rule explains the need of the flag μ in the write actions in history. Replaying a write overwrites the old write action in the history with the new one.

4 The Formal Model

We give a formal presentation of the memory model by giving the operational semantics of the language. Here we only show the most important part of the semantics. The complete definition is given in an extended version of this paper. We first define the execution context $\mathbb{P}$ of a program, which says we could pick any thread to execute at each step.

$$(\mathit{ThrdCtxt})\ \mathbb{P} ::= [\,] \mid \mathit{tid}.C \| \mathbb{P} \mid \mathbb{P} \| \mathit{tid}.C$$

$$\frac{C = \iota; C' \quad \iota \neq \mathbf{lock}\ \_ \quad e = \langle\langle i,t\rangle, \iota\rangle \quad b' = b \cup \{e\}}{(\mathbb{P}[i.C], (TQ, m, b, t, L)) \longmapsto (\mathbb{P}[i.C'], TQ, m, b', t{+}1, L)} \ (\textsc{issue})$$

$$\frac{C = \mathbf{lock}\ l; C' \quad l \notin \mathit{dom}(L) \quad L' = L\{l \rightsquigarrow i\} \quad m' = \mathit{AddSyn}(m, \langle\langle i,t\rangle, \mathbf{acq}, l\rangle)}{(\mathbb{P}[i.C], \langle TQ, m, b, t, L\rangle) \longmapsto (\mathbb{P}[i.C'], \langle TQ, m', b, t{+}1, L'\rangle)} \ (\textsc{lk})$$

$$\frac{C = (\mathbf{if}\ r\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2); C' \quad C'' = C_1; C' \quad \mathit{readyR}(\langle i,t\rangle, r, b) \quad TQ(i).\mathit{rf}(r) \neq 0}{(\mathbb{P}[i.C], \langle TQ, m, b, t, L\rangle) \longmapsto (\mathbb{P}[i.C''], \langle TQ, m, b, t{+}1, L\rangle)} \ (\textsc{if-t})$$

$$\frac{\begin{array}{c} TQ(i) = (\mathit{rf}, \mathit{rb}) \quad TQ' = TQ\{i \rightsquigarrow (\mathit{rf}', \mathit{rb}')\} \\ \langle(\mathit{rf}, \mathit{rb}), m, b, L\rangle \xrightarrow{i} \langle(\mathit{rf}', \mathit{rb}'), m', b', L'\rangle \end{array}}{(P, \langle TQ, m, b, t, L\rangle) \longmapsto (P, \langle TQ', m', b', t, L'\rangle)} \ (\textsc{evt})$$

$$\frac{TQ(i) = (\mathit{rf}, \mathit{rb}) \quad TQ' = TQ\{i \rightsquigarrow (\mathit{rf}, \emptyset)\}}{(P, \langle TQ, m, b, t, L\rangle) \longmapsto (P, \langle TQ', m, b \cup \mathit{rb}, t, L\rangle)} \ (\textsc{replay})$$

Fig. 7. Operational semantics: command to events

The execution of a program is shown in Fig. 7. For instructions other than **lock**, we wrap it with the thread id and the timer, and issue the event to the event buffer. The **lock** instruction is executed directly with no event issued. We use $f\{x \rightsquigarrow n\}$ to represent the update of the function f at the point x. After **lock**, the corresponding synchronization action is put into the history of every non-volatile variables. See Fig. 6 for the definition of *AddSyn*. The **lock** instruction will be blocked if the lock is owned by others. For the **if** command, it could be executed only if the value of the register r is ready. Recall the definition of *readyR* in Fig. 5. The IF-F rule and the rule for **while** are similar and omitted.

The EVT rule says the events in the event buffer could be executed in parallel with event issuance. Execution of an event issued by thread i is described in Fig. 8. The REPLAY rule says at any time we may choose to empty the replay buffer and move all the events in rb to the event buffer to execute them again.

All the event execution rules in Fig. 8 except the NO-WT-REPLAY rule require implicitly the premise $\mathit{Enabled}(b, e, i)$ defined on the top, which says the event e issued by thread i does not have any dependency with earlier events in b. Recall the dependency $e' \leftarrow e$ is defined in Fig. 6. The rules RD-V, WT-V and UNLK show the execution of synchronous events. We do not replay synchronization events, so rb is unchanged after each step. We use *AddSyn* (see Fig. 6) to insert the corresponding action into every history in memory.

The RD-SELF rule shows non-volatile read that sees a write from the same thread. The visibility $\mathit{visible}(\mathit{ts}, \mathit{wv}, h)$ is defined in Fig. 6. $\mathit{Replay}(\mathit{rb}, e, \mathit{rb}')$ defined on the top encodes the requirements for putting (or not putting) e into rb to get rb', which are explained in Sec. 3.5.

$$\mathit{Enabled}(b,e,i) \stackrel{\text{def}}{=} (e \in b) \wedge (e.ts.tid = i) \wedge \neg\exists e' \in b.\, e' \leftarrow e$$

$$\begin{aligned}\mathit{Replay}(rb,e,rb') \stackrel{\text{def}}{=}\ & ((\mathit{UseR}(e.\iota) \cup \mathit{UpdR}(e.\iota)) \cap \mathit{UpdR}(rb) = \emptyset) \wedge rb' = rb \\ & \vee(\mathit{UseR}(e.\iota) \subseteq \mathit{UpdR}(rb)) \wedge rb' = rb \uplus \{e\}\end{aligned}$$

$$\mathit{ModRef}(m,x,ts) \stackrel{\text{def}}{=} \lambda x'.\begin{cases} h \cup \{\langle ts, n, \mathbf{true}\rangle\} & \text{if } x'=x \wedge m(x)=h\uplus\{\langle ts,n,\_\rangle\} \\ m(x') & \text{if } x' \neq x \\ \mathit{undef} & \text{otherwise}\end{cases}$$

where $\uplus$ means the union of two disjoint sets.

$$\mathit{Add}(m,x,ts,n) \stackrel{\text{def}}{=} \lambda x'.\begin{cases} m(x) \cup \{\langle ts,n,\mathbf{false}\rangle\} & \text{if } x' = x \text{ and } \neg\exists n',\mu.\ \langle ts,n',\mu\rangle \in m(x) \\ h \cup \{\langle ts,n,\mathbf{false}\rangle\} & \text{if } x'=x \wedge m(x) = h\uplus\{\langle ts,n',\mathbf{false}\rangle\} \\ m(x') & \text{if } x' \neq x \\ \mathit{undef} & \text{otherwise}\end{cases}$$

$$\frac{\begin{array}{c} e = \langle ts, r := v\rangle \quad syn = \langle ts, \mathbf{ld}, v\rangle \quad m' = \mathit{AddSyn}(m, syn) \\ rf' = rf\{r \rightsquigarrow m(v)\} \quad r \notin \mathit{UpdR}(rb)\end{array}}{\langle (rf, rb), m, b, L\rangle \xrightarrow{i} \langle (rf', rb), m', b\backslash\{e\}, L\rangle}\ (\textsc{rd-v})$$

$$\frac{\begin{array}{c} e = \langle ts, v := r\rangle \quad syn = \langle ts, \mathbf{st}, v\rangle \quad m' = \mathit{AddSyn}(m, syn) \\ m'' = m'\{v \rightsquigarrow rf(r)\} \quad r \notin \mathit{UpdR}(rb)\end{array}}{\langle (rf, rb), m, b, L\rangle \xrightarrow{i} \langle (rf, rb), m'', b\backslash\{e\}, L\rangle}\ (\textsc{wt-v})$$

$$\frac{\begin{array}{c} e = \langle ts, \mathbf{unlock}\ l\rangle \quad L(l) = i \quad L' = L \setminus \{l\} \\ syn = \langle ts, \mathbf{rel}, l\rangle \quad m' = \mathit{AddSyn}(m, syn)\end{array}}{\langle (rf, rb), m, b, L\rangle \xrightarrow{i} \langle (rf, rb), m', b\backslash\{e\}, L'\rangle}\ (\textsc{unlk})$$

$$\frac{\begin{array}{c} e = \langle ts, r := x\rangle \quad \mathit{visible}(ts, \langle ts', n, \_\rangle, m(x)) \quad ts.tid = ts'.tid \\ rf' = rf\{r \rightsquigarrow n\} \quad \mathit{Replay}(rb, e, rb')\end{array}}{\langle (rf, rb), m, b, L\rangle \xrightarrow{i} \langle (rf', rb'), m, b\backslash\{e\}, L\rangle}\ (\textsc{rd-self})$$

$$\frac{\begin{array}{c} e = \langle ts, r := x\rangle \quad \mathit{visible}(ts, \langle ts', n, \_\rangle, m(x)) \quad ts.tid \neq ts'.tid \\ rf' = rf\{r \rightsquigarrow n\} \quad m' = \mathit{ModRef}(m, x, ts') \quad \mathit{Replay}(rb, e, rb')\end{array}}{\langle (rf, rb), m, b, L\rangle \xrightarrow{i} \langle (rf', rb'), m', b\backslash\{e\}, L\rangle}\ (\textsc{rd-other})$$

$$\frac{e = \langle ts, x := r\rangle \quad e \in b \quad \langle ts, \_, \mathbf{true}\rangle \in m(x)}{\langle (rf, rb), m, b, L\rangle \xrightarrow{i} \langle (rf, rb), m, b\backslash\{e\}, L\rangle}\ (\textsc{no-wt-replay})$$

$$\frac{e = \langle ts, x := r\rangle \quad rf(r) = n \quad m' = \mathit{Add}(m, x, ts, n) \quad \mathit{Replay}(rb, e, rb')}{\langle (rf, rb), m, b, L\rangle \xrightarrow{i} \langle (rf, rb'), m', b\backslash\{e\}, L\rangle}\ (\textsc{wt})$$

$$\frac{e = \langle ts, r := E\rangle \quad [\![E]\!]_{rf} = n \quad rf' = rf\{r \rightsquigarrow n\} \quad \mathit{Replay}(rb, e, rb')}{\langle (rf, rb), m, b, L\rangle \xrightarrow{i} \langle (rf', rb'), m, b\backslash\{e\}, L\rangle}\ (\textsc{pure})$$

Fig. 8. Execution of events

The RD-OTHER rule is for a read seeing a write from a different thread. In this case we mark the flag μ of the write action to **true** through $\mathit{ModRef}(m, x, ts)$ (defined on the top), thus we know the write has been seen by a different thread.

If we want to execute a write event and notice that there is already a write with the same timestamp in the history, we know this write must be a replay of the earlier write in history. The NO-WT-REPLAY says we must discard this write without executing it if the earlier write has been seen by a different thread. As explained in Sec. 3.5, this is necessary to avoid out-of-thin-air values. If there is no such write in history, we could execute the write following the next WT rule. Whether to put it into rb or not follows the constraint $\mathit{Replay}(rb, e, rb')$. The PURE rule is for the pure event $r := E$.

5 More Examples

In this section we show more examples of OHMM, and discuss the differences between OHMM and JMM. As in Sec. 3, we assume the initial values of all memory cells are 0 in the following examples.

Revisit of Example 2. The results due to the causality circle in HMM are not permitted in our model. For the first program, lines 2 and 3 cannot be executed earlier than line 1, which reads 0 and invalidates the test at line 2. Therefore line 3 cannot be executed. Similarly, line 4 can only read 0. We have proved that OHMM has DRF-guarantee, thus $r_1 = r_2 = 0$ is the only possible result. For the second program, since OHMM is a generative model, there is no way for r_1 and r_2 to have other values except 0.

Example 8 (Causality Test Case 5 [13]).

$$\begin{array}{l} 1: \; r_1 := x; \\ 2: \; y := r_1; \end{array} \quad \Bigg\| \quad \begin{array}{l} 3: \; r_2 := y; \\ 4: \; x := r_2; \end{array} \quad \Bigg\| \quad 5: \; z := 1; \quad \Bigg\| \quad \begin{array}{l} 6: \; r_3 := z; \\ 7: \; x := r_3; \end{array}$$

Result: $r_1 = r_2 = 1, r_3 = 0$?

The result is viewed to be out-of-thin-air and should be disallowed according to JMM. However, it is not as vicious as the results in Example 2 since the value 1 is indeed assigned to memory in the program. Whether it should be allowed or not has been very controversial in the JMM discussion mailing list. The result is allowed in our model.

We refer to the events generated by lines 1-7 as $e_1, e_2, \ldots e_7$. We execute e_5 first. Then execute e_6 and put it into the replay buffer. Let r_3 gets 1 from z. Next we execute e_7, e_1, e_2, e_3, e_4 in turn, and let r_1 and r_2 get 1. Finally, we remove e_6 from the replay buffer and execute it again. This time we let it read 0, the initial value of z. This is possible since both the initial value and the value 1 are visible by this read. The Causality Test Case 10 [13] is a similar controversial example forbidden in JMM but allowed in our model.

The next example shows the Causality Test Case 17 [13], whose result is claimed to be supported in JMM but it fails to do so due to a subtle bug [2].

Example 9 (Causality Test Case 17 [13]).

$$\left(\begin{array}{ll} 1: & r_1 := x; \\ 2: & r_2 := (r_1\,\texttt{!=}\,42); \\ 3: & \textbf{if}(r_2) \\ 4: & \quad x := 42; \end{array} \quad \begin{array}{ll} 5: & r_3 := x; \\ 6: & y := r_3; \end{array}\right) \Bigg\| \quad \begin{array}{ll} 7: & r_4 := y; \\ 8: & x := r_4; \end{array}$$

Result: $r_1 = r_3 = r_4 = 42$?

It is allowed in our model. We can execute line 1, read 0, and replay it at the same time. Then we enter the branch of if statement and write 42 to the history of x. We execute the other instructions, get $r_3 = r_4 = 42$, and write another 42 to x at line 8. At last, we execute line 1 again. According to our semantics, it can read 42 from x, the one written by line 8 (not by line 4). Due to the same bug, JMM fails to support Test Cases 18-20 too, which are allowed in our model.

The next example is taken from Cenciarelli [5], also used by Aspinall and Ševčík [3] to show the ugly part of JMM.

Example 10. Result: $r_1 = r_3 = r_4 = 1$?

$$\begin{array}{ll} 1: & r_1 := z; \\ 2: & r_2 := (r_1\texttt{==}1); \\ 3: & \textbf{if}(r_2)\{ \\ 4: & \quad x := 1; \\ 5: & \quad y := 1;\} \\ 6: & \textbf{else}\{ \\ 7: & \quad y := 1; \\ 8: & \quad x := 1\} \end{array} \Bigg\| \quad \begin{array}{ll} 9: & r_3 := x; \\ 10: & r_4 := y; \\ 11: & r_5 := (r_3\texttt{==}1\texttt{\&\&}r_4\texttt{==}1); \\ 12: & \textbf{if}(r_5) \\ 13: & \quad z := 1; \end{array}$$

JMM disallows this result, but would allow it if we flip lines 4 and 5 (or lines 7 and 8), showing that reordering of independent instruction may introduce new behaviors, a bug in JMM. Our model allows the result no matter we swap the statements or not. We first execute line 1, read 0, and replay it at the same time. Then we execute other lines following the SC order, and let r_3 and r_4 get 1. Finally, we execute line 1 again, which reads 1, the value written by line 13.

Example 11 ("bait-and-switch" behaviors). Result: $r_1 = r_3 = r_4 = 1$?

$$\begin{array}{ll} 1: & r_1 := x; \\ 2: & r_2 := (r_1\texttt{==}0); \\ 3: & \textbf{if}(r_2) \\ 4: & \quad x := 1; \end{array} \Bigg\| \quad \begin{array}{ll} 5: & r_3 := x; \\ 6: & y := r_3; \end{array} \quad \Bigg\| \quad \begin{array}{ll} 7: & r_4 := y; \\ 8: & x := r_4; \end{array}$$

This example shows the bait-and-switch behavior ([12], Fig. 11) disallowed by JMM. However, it is allowed in JMM if we merge the first two threads by appending the second thread at the end of the first, showing a surprising fact that programs with less concurrency may have more behaviors than the more concurrent ones. In either case above the result is allowed in our model, which eliminates the surprise. We can execute the first thread first, and replay line 1. Then we execute the second and the third threads sequentially. Finally we execute line 1 a second time, which may read 1 written by line 8.

It seems harmless to allow this result. As Manson et al. [12] pointed out, disallowing this behavior in JMM is due to more of "taste and preference" than any concrete requirement. There is another similar example disallowed in JMM but allowed in our model ([12], Fig. 10).

The next example is also taken from Aspinall and Ševčík [3]. The result is not allowed in JMM. However, if we move line 7 into the following critical region or change the variable x into volatile, the result is allowed. This shows an anti-intuitive property of JMM: adding synchronization to a program may introduce more behaviors other than deadlock.

Example 12 (Roach Motel Semantics). Result: $r_1 = r_2 = r_4 = 1$?

		$7:\ \ r_1 := x;$	
		$8:\ \ \textbf{lock}\, l;$	
		$9:\ \ \ \ r_2 := z;$	
$1:\ \ \textbf{lock}\, l;$	$4:\ \ \textbf{lock}\, l;$	$10:\ \ \ \ r_3 := (r_1 \texttt{==} 2);$	
$2:\ \ \ \ x := 2;$	$5:\ \ \ \ x := 1;$	$11:\ \ \ \ \textbf{if}(r_3)$	$16:\ \ r_4 := y;$
$3:\ \ \textbf{unlock}\, l;$	$6:\ \ \textbf{unlock}\, l;$	$12:\ \ \ \ \ \ y := 1;$	$17:\ \ z := r_4;$
		$13:\ \ \ \ \textbf{else}$	
		$14:\ \ \ \ \ \ y := r_2;$	
		$15:\ \ \textbf{unlock}\, l;$	

The result is allowed in our model. We execute lines 1-3 first, then execute line 7 and replay it. We let line 7 read 2. Then execute lines 4 to 6 and 8 to 12 (since r_1 equals to 2, we can enter the first branch of the **if** statement), and replay line 9 in this process. Then execute lines 16 and 17, where we get $r_4 = 1$. Before executing line 15, we remove lines 7 and 9 from the replay buffer and execute them a second time. This time line 7 reads 1 (written by line 5), and line 9 reads 1 too (written by line 17). Thus we get $r_1 = r_2 = r_4 = 1$.

However, if we move line 7 into the critical region, we cannot get the result any more, because lines 7, 2 and 5 now have a total happens-before order. Replaying line 7 would not let it read a different value. Making x volatile makes the result impossible too, because we cannot replay line 7 any more, which is now a synchronization event.

Example 13. Result: $r_1 = r_2 = r_3 = 1$?

	$3:\ \ \textbf{lock}\, l;$	$7:\ \ \textbf{lock}\, l;$
$1:\ \ r_1 := y;$	$4:\ \ r_2 := x;$	$8:\ \ y := 1;$
$2:\ \ x := r_1;$	$5:\ \ z := 1;$	$9:\ \ r_3 := z;$
	$6:\ \ \textbf{unlock}\, l;$	$10:\ \ \textbf{unlock}\, l;$

This is another surprising example [3]. The result is possible in JMM, which seems to allow interleaving of critical regions protected by the same lock. Our model cannot produce this result since the interleaving of the last two threads is prohibited by the $\overset{\mathbf{b}}{\leftarrow}$ and $\overset{\mathbf{s}}{\leftarrow}$ dependency.

Comparison with JMM. As we have shown through the examples, our model does not subsume JMM, nor does the inverse hold. For programs with no locks or volatile variables, our model is weaker than JMM and supports a lot more behaviors disallowed in JMM. *Our model passes all the causality test cases, except those involving language features not supported here and the test cases 5 and 10.* We allow the questionable behaviors of 5 and 10, which are forbidden in JMM. Also, we allow the "bait-and-switch" behaviors forbidden in JMM, as shown in Example 11. These cases have been controversial in the JMM discussion mailing list. We believe it is harmless to support them.

For racy programs with locks and/or volatile variables, there are behaviors allowed in JMM but not in our model (see Examples 12 and 13). However, as Aspinall and Ševčík pointed out [3], these behaviors are very anti-intuitive and should be disallowed.

More about OHMM. In Appendix A, we prove that, like JMM, our model has DRF-Guarantee, which ensures DRF programs have SC behaviors only. In Appendix B, we prove the soundness of common program transformations in our model. Some of them are unsound in JMM. The corresponding proofs are given in the companion technical report [18]. We also give a mechanized formulation of the model in Coq. Proving the theorems and lemmas in Coq are ongoing work. An interpreter of the language is also provided [18], which could demonstrate the possible relaxed behaviors of a given program in OHMM.

6 Related Work and Conclusion

There has been much work on relaxed memory models. We only discuss the closely related work here.

Yang et al. [17] proposed Java thread semantics using their Uniform Memory Model (UMM), which is similar to ours in many aspects. Both are operational, and are defined based on an abstract machine. In UMM, the machine has thread-local instruction buffers, which play a similar role as our global event buffer and allow the reordering of events. UMM also has a global instruction buffer that keeps all previous memory writes, which is similar to our history-based memory. However, there is no replay mechanism in UMM. As a result, many important reordering, such as Example 6, cannot be supported. In addition to UMM, there was much work trying to fix an earlier version of JMM until Manson et al. [12] proposed the current JMM. As Manson et al. pointed out, most of the earlier work failed to support many important relaxed orderings.

Since the current JMM was proposed, there have been efforts to formalize it and prove its DRF-Guarantee [2, 6, 10]. Their formalizations are rigorous and thorough, and are helpful to understand JMM and to discover bugs in it. However, they all follow JMM's original axiomatic formulation instead of defining a different operational model.

Saraswat et al. [14] proposed a relaxed memory model, RAO. The model produces relaxed behaviors by means of program transformations. The set of transformations is carefully chosen to avoid out-of-thin-air behaviors. Like JMM, the RAO model forbids the causality test cases 5 and 10, and the controversial examples in Sec. 8 of Manson et al. [12], which are allowed in our model.

Cenciarelli et al. [5] formalized JMM using a semantic framework combining operational, denotational and axiomatic techniques. They use a configuration theory to specify the dependency of events. In the operational semantics, they allow events to be added to configurations before the corresponding code is executed. Later such prediction needs to be *fulfilled* by the execution. We do not do speculation directly. Instead, our replay mechanism allows us to run code multiple times. The result of the first time execution can be used as a speculation. Such a speculation is always valid and there is no need of justification.

Jagadeesan et al. [7] gave generative operational semantics for relaxed memory models. Their model also keeps all the write events in the memory, like our use of histories. Similar to Cenciarelli et al. [5], they support speculation directly by predicting memory values non-deterministically. The predication needs to be justified using an extra copy of the code running in a separate copy of state. As we explained above, this is different from our replay mechanism. Although the model supports many behaviors of lock-less programs that are disallowed in JMM, it disallows the controversial examples forbidden by JMM (*e.g.*, Example 11). These examples are allowed in our model, so it seems our replay mechanism is more relaxed than their support of speculation.

Sarkar et al. [15] gave a semantic model for Power multiprocessors. Their storage subsystem is similar to our memory and buffers. They support speculation by restarting an instruction before it is committed. However, writes are not send to the storage subsystem before their instructions are committed, which means threads cannot see the write from others until it is committed. This is different from the replay mechanism. The memory model of Power is stronger than our model and JMM.

Conclusion. We propose OHMM, an operational happens-before memory model. To support speculation and analysis made by compiler optimizations, we introduce a novel replay mechanism, which allows us to run the code multiple times to simulate static analysis and the subsequent program reordering. We hope the vanilla state-transition-based operational formulation makes the model easy to understand by programmers. The model satisfies the three criteria we explain in Sec. 1. It has DRF-Guarantee, and prohibits the harmful out-of-thin-air behaviors in the naive happens-before model. On the other hand, it is reasonably weak. For programs with no locks, the model allows harmless behaviors prohibited in JMM and its variations such as Jagadeesan et al. [7]. It also rules out many of the anti-intuitive features of JMM, as discussed in Sec. 5.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] D. Aspinall and J. Ševčík. Formalising java's data race free guarantee. In *TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2007.

[3] D. Aspinall and J. Ševčík. Java memory model examples: Good, bad and ugly. In *VAMP 2007*, Sep 2007.

[4] H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI 2005*, pages 261–268. ACM, 2005.

[5] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2007.

[6] M. Huisman and G. Petri. The java memory model: a formal explanation. In *VAMP 2007*, 2007.

[7] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 307–326. Springer, 2010.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[10] A. Lochbihler. Java and the java memory model - a unified, machine-checked formalisation. In *ESOP 2012*, volume 7211 of *Lecture Notes in Computer Science*, pages 497–517. Springer, 2012.

[11] J. Manson. *The Java Memory Model*. PhD thesis, University of Maryland, College Park, 2004.

[12] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL 2005*, pages 378–391. ACM, 2005.

[13] W. Pugh. Java memory model causality test cases, 2004. `http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html`.

[14] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *PPoPP 2007*, pages 161–172. ACM, 2007.

[15] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

[16] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP 2008*, volume 5142 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 2008.

[17] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Proc. 2002 Joint ACM-ISCOPE Conf. on Java Grande 2002*, pages 192–201. ACM, 2002.

[18] Y. Zhang and X. Feng. An operational happens-before memory model (extended version). Technical report, USTC, 2012. `http://staff.ustc.edu.cn/~xyfeng/research/publications/OHMM.html`.

A DRF-Guarantee

DRF programs in OHMM behave the same as in the SC model. In this section, we give a brief overview of our formulation of data-race-freedom and our proof of the DRF-Guarantee. More details are shown in the extended version of this paper [18].

The DRF property needs to be defined in a SC model, therefore we first define a *strong* machine model that executes programs following the standard interleaving semantics. The machine configuration is defined in Fig. 9. We redefine the program state $\sigma_{\mathbf{s}}$, called a *strong state*, as a quadruple consisting of a thread queue, memory, timer and a lock set. The thread queue in this model maps a thread id to its local register file. The memory is standard, which maps a variable (non-volatile and volatile) to an integer value.

We omit the operational semantics of the strong machine, which is standard interleaving transition semantics with labels recording the event of individual steps. The labeled transition steps may form a sequential consistent execution event trace, a total order $<$ among events.

We say a program configuration $(P, \sigma_{\mathbf{s}})$ is DRF if, for every execution event trace, if there are *conflicting* memory access events e_1 and e_2 and $e_1 < e_2$, there must be a release (or a write of volatile) event e'_1 and a corresponding acquire of the same lock (or a read of the same volatile variable) e'_2 such that $e_1 < e'_1 < e'_2 < e_2$, e_1 and e'_1 are produced by the same thread, and so are e'_2 and e_2. Two memory accesses are conflicting if they are from different threads, access the same non-volatile variable, and at least one of them is a write.

Next we relate the machine state σ in our abstract weak machine with a strong state $\sigma_{\mathbf{s}}$. To distinguish the two, we also represent the former as $\sigma_{\mathbf{r}}$ (a relaxed state). The function $\mathit{Value}(\sigma_{\mathbf{r}}, x)$ gets the most recent written value of variable x:

$$\mathit{Value}(\sigma_{\mathbf{r}}, x) \stackrel{\text{def}}{=} \begin{cases} n & \text{if } \sigma_{\mathbf{r}}.m(x) = n \\ \mathit{wv}.n & \text{if } \sigma_{\mathbf{r}}.m(x) = h \wedge \mathit{wv} \in h, \\ & \quad \wedge(\forall \mathit{wv}', \mathit{wv}'' \in h. \mathit{wv}' \prec_h^{\text{hb}} \mathit{wv}'' \vee \mathit{wv}' \prec_h^{\text{hb}} \mathit{wv}'') \\ & \quad \wedge(\forall \mathit{wv}' \in h.\ \mathit{wv}' = \mathit{wv} \vee \mathit{wv}' \prec_h^{\text{hb}} \mathit{wv}) \end{cases}$$

Note, for non-volatile variable x, $\mathit{Value}(\sigma_{\mathbf{r}}, x)$ is defined only if the write actions in the history $\sigma_{\mathbf{r}}.m(x)$ are totally ordered by the happens-before relation. It is easy to prove that for DRF programs this requirement is always satisfied during the program execution.

$$\begin{array}{rrl} (\mathit{ThreadQ}) & \mathit{TQ} & ::= \{\mathit{tid}_0 \rightsquigarrow \mathit{rf}_0, \ldots, \mathit{tid}_k \rightsquigarrow \mathit{rf}_k\} \\ (\mathit{Mem}) & m & ::= \{x \rightsquigarrow n_0, y \rightsquigarrow n_1, z \rightsquigarrow n_3, \ldots, \quad v_0 \rightsquigarrow n', v_1 \rightsquigarrow n'', \ldots\} \\ (\mathit{State}) & \sigma_{\mathbf{s}} & ::= \langle \mathit{TQ}, m, t, L \rangle \end{array}$$

Fig. 9. The strong machine

We relate a relaxed state $\sigma_\mathbf{r}$ and a strong state $\sigma_\mathbf{s}$ below:

$$\begin{array}{lcl}
\sigma_\mathbf{r} \overset{T}{=} \sigma_\mathbf{s} & \overset{\text{def}}{=} & \sigma_\mathbf{r}.t = \sigma_\mathbf{s}.t \\
\sigma_\mathbf{r} \overset{R}{=} \sigma_\mathbf{s} & \overset{\text{def}}{=} & \forall i.\, \sigma_\mathbf{r}.TQ(i).rf = \sigma_\mathbf{s}.TQ(i) \\
\sigma_\mathbf{r} \overset{M}{=} \sigma_\mathbf{s} & \overset{\text{def}}{=} & \forall x.\, \mathit{Value}(\sigma_\mathbf{r}, x) = \sigma_\mathbf{s}.m(x) \\
\sigma_\mathbf{r} \overset{L}{=} \sigma_\mathbf{s} & \overset{\text{def}}{=} & \sigma_\mathbf{r}.L = \sigma_\mathbf{s}.L \\
\sigma_\mathbf{r} \overset{MRLT}{=\!=\!=\!=} \sigma_\mathbf{s} & \overset{\text{def}}{=} & \sigma_\mathbf{r} \overset{M}{=} \sigma_\mathbf{s} \wedge \sigma_\mathbf{r} \overset{R}{=} \sigma_\mathbf{s} \wedge \sigma_\mathbf{r} \overset{L}{=} \sigma_\mathbf{s} \wedge \sigma_\mathbf{r} \overset{T}{=} \sigma_\mathbf{s}
\end{array}$$

The relation says the timer, the register file and the lock status in the relaxed state must be the same as those in the strong state. For memory, although a history in the relaxed state contains multiple writes, they are always totally ordered for DRF programs, and only the most recent value can be read. Such a value needs to be the same with the single value in the strong state.

We define $\mathit{buff}(\sigma_\mathbf{r})$ as the union of the event buffer and all the thread-local replay buffers.

$$\mathit{buff}(\sigma_\mathbf{r}) \overset{\text{def}}{=} \{e \mid e \in \sigma_\mathbf{r}.b \vee \exists i.\, e \in \sigma_\mathbf{r}.TQ(i).rb\}$$

Theorem 1 (DRF-Guarantee). *For all P, σ_r and σ_s, if (P, σ_s) is DRF, $\sigma_r \overset{MRLT}{=\!=\!=\!=} \sigma_s$, and $\mathit{buff}(\sigma_r) = \emptyset$, then the following are true:*

- *If $(P, \sigma_r) \longmapsto^* (\mathbf{skip}, \sigma'_r)$ and $\mathit{buff}(\sigma'_r) = \emptyset$, then there exists σ'_s such that $(P, \sigma_s) \longrightarrow^* (\mathbf{skip}, \sigma'_s)$ and $\sigma'_r \overset{MRLT}{=\!=\!=\!=} \sigma'_s$.*
- *If $(P, \sigma_s) \longrightarrow^* (\mathbf{skip}, \sigma'_s)$, then there exists σ'_r such that $(P, \sigma_r) \longmapsto^* (\mathbf{skip}, \sigma'_r)$, $\mathit{buff}(\sigma'_r) = \emptyset$ and $\sigma'_r \overset{MRLT}{=\!=\!=\!=} \sigma'_s$.*

The theorem says, starting from related initial states, if a DRF program reaches a final state $\sigma'_\mathbf{r}$ in our relaxed semantics, it could also reach a strong state $\sigma_\mathbf{s}$ in SC semantics such that $\sigma'_\mathbf{r} \overset{MRLT}{=\!=\!=\!=} \sigma'_\mathbf{s}$; and the inverse is also true.

Proof of the second half of the theorem is trivial, since the issuance of events in our weak machine follows the interleaving semantics. We could simulate the strong machine by executing an event immediately after its issuance, and never replay it. To prove the first half, we use a decorated semantics of the weak machine, which adds labels to transition steps. The we can establish a weak simulation between the weak semantics and the strong one. The proof details are shown in the extended version [18].

B Program Transformations in OHMM

Following Ševčík and Aspinall [16], we study the validity of some simple program transformations in OHMM. As shown in Fig. 10, we take the same set of transformations considered in Ševčík and Aspinall [16], except the trace-preserving transformations and the external action reordering transformation. We omit the former because we do not use a trace semantics here. The latter is omitted because it does not apply in our language, which does not produce external

Transformation	SC	JMM	JMM-Alt	OHMM
Reordering normal memory accesses	×	×	√	√
Redundant read after read elimination	√	×	×	√
Redundant read after write elimination	√	√	√	√
Irrelevant read elimination	√	√	√	√
Irrelevant read introduction	√	×	×	√
Redundant write before write elimination	√	√	√	√
Redundant write after read elimination	√	×	×	√
Roach-motel reordering	×(√ for locks)	×	×	√

Fig. 10. Validity of transformations

events. Figure 10 also shows the validity of the transformations in SC, JMM, JMM-Alt [2] and OHMM. The result for the first three models are taken directly from [16]. The result under OHMM is proved here. All the transformations are valid in our model, which means OHMM can accommodate more optimizations (Note the transformations studied here are defined more syntactically in Figs. 11 and 12 than those in [2], which are semantically defined and could be more general than ours).

B.1 Transformations

We divide the transformations into two classes: elimination and reordering. Irrelevant read introduction does not belong to either of them, but it can be defined as the inverse of irrelevant read elimination (rule E-IR below). The set of eliminations we consider are defined in Fig. 11, including:

– Elimination of a read following a read from the same non-volatile variable (rule E-RAR).
– Elimination of a read following a write to the same non-volatile variable (rule E-RAW).
– Elimination of a write following a read with the same value from the same non-volatile variable (rule E-WAR).
– Elimination of a write preceding a write to the same non-volatile variable (rule E-WBW).
– Elimination of a read that whose value is not used(rule E-IR).

The reordering transformations are defined in Fig.12, including:

– Reordering of independent non-conflicting non-volatile memory accesses (rules R-RR, R-WW, R-WR and R-RW);
– Reordering an lock statement and a preceding normal instruction (rule RoachMotel-L).
– Reordering an Unlock statement and a following normal instruction (rule RoachMotel-U).

Note that in these rules, r_1 and r_2 (x_1 and x_2) do not necessarily refer to different registers (variables), unless explicitly required in the premise.

$$(\mathit{SeqContext}) \qquad \mathbb{E} ::= [\,] \mid E; C$$

$$\frac{}{C \xrightarrow{\text{E}} C} \qquad \frac{\mathit{tid}.C \xrightarrow{\text{E}} \mathit{tid}.C'}{\mathbb{P}[\mathit{tid}.C] \xrightarrow{\text{E}} \mathbb{P}[\mathit{tid}.C']} \qquad \frac{C \xrightarrow{\text{E}} C'}{\mathit{tid}.\mathbb{E}[C] \xrightarrow{\text{E}} \mathit{tid}.\mathbb{E}[C']}$$

$$\frac{}{r_1 := x; r_2 := x; \xrightarrow{\text{E}} r_1 := x; r_2 := r_1;}\ (\text{E-RAR})$$

$$\frac{}{x := r_1; r_2 := x; \xrightarrow{\text{E}} x := r_1; r_2 := r_1;}\ (\text{E-RAW})$$

$$\frac{}{r := x; x := r; \xrightarrow{\text{E}} r := x;}\ (\text{E-WAR})$$

$$\frac{}{x := r_1; x := r_2; \xrightarrow{\text{E}} x := r_2;}\ (\text{E-WBW})$$

$$\frac{r \notin \mathit{UseR}(E)}{r := x; r := E; \xrightarrow{\text{E}} r := E;}\ (\text{E-IR})$$

Fig. 11. Syntactic Elimination

$$\frac{}{C \xrightarrow{\text{S}} C} \qquad \frac{\mathit{tid}.C \xrightarrow{\text{S}} \mathit{tid}.C'}{\mathbb{P}[\mathit{tid}.C] \xrightarrow{\text{S}} \mathbb{P}[\mathit{tid}.C']} \qquad \frac{C \xrightarrow{\text{S}} C'}{\mathit{tid}.\mathbb{E}[C] \xrightarrow{\text{S}} \mathit{tid}.\mathbb{E}[C']}$$

$$\frac{r_1 \neq r_2}{r_1 := x_1; r_2 := x_2; \xrightarrow{\text{S}} r_2 := x_2; r_1 := x_1;}\ (\text{R-RR})$$

$$\frac{x_1 \neq x_2}{x_1 := r_1; x_2 := r_2; \xrightarrow{\text{S}} x_2 := r_2; x_1 := r_1;}\ (\text{R-WW})$$

$$\frac{x_1 \neq x_2 \quad r_1 \neq r_2}{x_1 := r_1; r_2 := x_2; \xrightarrow{\text{S}} r_2 := x_2; x_1 := r_1;}\ (\text{R-WR})$$

$$\frac{x_1 \neq x_2 \quad r_1 \neq r_2}{r_1 := x_1; x_2 := r_2; \xrightarrow{\text{S}} x_2 := r_2; r_1 = x_1;}\ (\text{R-RW})$$

$$\frac{}{\iota_n; \textbf{lock}\, l; \xrightarrow{\text{S}} \textbf{lock}\, l; \iota_n;}\ (\text{RoachMotel-L})$$

$$\frac{}{\textbf{unlock}\, l; \iota_n; \xrightarrow{\text{S}} \iota_n; \textbf{unlock}\, l;}\ (\text{RoachMotel-U})$$

Fig. 12. Syntactic Reordering

B.2 Validity of Transformations

The validity of a transformation says that any behavior of the target program (the one produced by the transformation) is a behavior of the original one, *i.e.*, the target program does not produce new behaviors. The transformations should be valid in any context. However, it is possible that the original program may have more behaviors than the target in some transformations, as shown in the following example.

Example 14 (E-WAR).

$$\begin{array}{l} r_1 := x; \\ x := r_1; \end{array} \Bigg\| \begin{array}{l} r_2 := x; \\ x := 2; \end{array} \quad \xrightarrow{\text{E}} \quad r_1 := x; \Bigg\| \begin{array}{l} r_2 := x; \\ x := 2; \end{array}$$

The transformation is done by applying the E-WAR rule over the left thread. In the source program r_2 can read the value 2 (written by $x := r_1$), which is not possible in the target, where the only write operation is $x := 2$, and r_2 can not read this value since this write happens after it in program order.

When we compare behaviors of programs, we compare their final states reached from the same initial state. Below we define $\sigma \overset{obsv}{=\!=\!=} \sigma'$, an observational equality between σ and σ'.

$$\begin{array}{lcl}
\sigma \overset{r}{=\!=} \sigma' & \overset{\text{def}}{=} & \sigma.TQ.rf = \sigma'.TQ.rf \\
\sigma \overset{l}{=\!=} \sigma' & \overset{\text{def}}{=} & \sigma.L = \sigma'.L \\
\sigma \overset{b}{=\!=} \sigma' & \overset{\text{def}}{=} & buff(\sigma) = buff(\sigma') = \emptyset \\
\sigma_{\mathbf{r}} \overset{m}{=\!=\!=} \sigma' & \overset{\text{def}}{=} & dom(\sigma.m) = dom(\sigma'.m) \wedge \forall x \in dom(\sigma.m). \\
 & & \quad \sigma.m(x) = \sigma'.m(x) \\
 & & \quad\quad \vee \forall i.\ visibleV(i, \sigma.t, \sigma.m(x)) = visibleV(i, \sigma'.t, \sigma'.m(x))
\end{array}$$

$$\sigma \overset{obsv}{=\!=\!=} \sigma' \overset{\text{def}}{=} \sigma \overset{r}{=\!=} \sigma' \wedge \sigma \overset{b}{=\!=} \sigma' \wedge \sigma \overset{m}{=\!=\!=} \sigma' \wedge \sigma \overset{l}{=\!=} \sigma'$$

$$\text{where } \ visibleV(i, t, h) \overset{\text{def}}{=} \{wv.v \mid wv \in h.visible(\langle i, t\rangle, wv, h)\}$$

Note here we do not require σ and σ' to be literally the same, which is unnecessarily strong for the history-based memory. We say two states have the same memory (*i.e.*, $\sigma \overset{m}{=\!=\!=} \sigma'$) if any subsequent read could see the same set of values (recall the definition of $visible(ts, wv, h)$ in Fig. 6).

We use $P \xrightarrow{\text{T}} P'$ to represent a transformation from P to P' through either elimination or reordering, *i.e.*, $\xrightarrow{\text{T}} \overset{\text{def}}{=} \xrightarrow{\text{E}} \cup \xrightarrow{\text{S}}$. Then $\xrightarrow{\text{T}}{}^*$ represents the reflexive transitive closure of $\xrightarrow{\text{T}}$.

Theorem 2 (Validity of Transformation). *For all P, σ and P', if $P \xrightarrow{\text{T}}{}^* P'$, $(P', \sigma) \longmapsto^* (\mathbf{skip}, \sigma')$, and $buff(\sigma') = \emptyset$, then there exists σ'' such that $(P, \sigma) \longmapsto^* (\mathbf{skip}, \sigma'')$ and $\sigma' \overset{obsv}{=\!=\!=} \sigma''$.*

The theorem says, staring from the same initial states, if a transformed program reaches a final state σ', then the original program could reach a final state σ'' such that σ' and σ'' are observationally equal.

The theorem follows the validity of each individual transformation, which is proved by defining proper simulation relations between the target and the original programs. By the transitivity of $\overset{obsv}{=\!=}$, we know arbitrary combinations of individual transformation are also valid. Details of the proof are given in the TR [18].