Rollback Atomicity

Serdar Tasiran and Tayfun Elmas
Koc University, Istanbul, Turkey

Abstract. We introduce a new non-interference criterion for concurrent programs: rollback atomicity. Similarly to other definitions of atomicity, rollback atomicity of a given concurrent execution requires that there be a matching serial execution. Rollback atomicity differs from other definitions of atomicity in two key regards. First, it is formulated as a special case of view refinement. As such, it requires a correspondence between the states of a concurrent and a serial execution for each atomic block rather than only at quiescent states. Second, it designates a subset of shared variables as peripheral and has more relaxed requirements for peripheral variables than previous non-interference criteria.
In this paper, we provide the motivation for rollback atomicity. We formally define it and compare it with other notions of atomicity and non-interference criteria. We built a runtime checker for rollback atomicity integrated into the refinement checking tool, VYRD. This implementation was able to verify that concurrent executions of our motivating example are rollback atomic.

1 Introduction

Shared-memory multi-threaded programs are prone to concurrency-related bugs. In addition to program-specific correctness properties, several generic correctness and non-interference criteria have been studied for such programs, including freedom from race conditions and deadlocks. Such generic criteria provide a separation of concerns. By first verifying or trying to ensure that his program does not violate a generic correctness criterion, a programmer can then make certain assumptions and simplify his reasoning while ensuring other properties of the program. For instance, for Java programs, ensuring race freedom allows the programmer to assume sequential consistency for his program [9].

A higher-level correctness criterion that is frequently used is atomicity. Programmers designate blocks of their code *atomic* and would like to have certain mutual-exclusion and non-interference guarantees about such blocks. A programmer may ensure the atomicity of code blocks himself using synchronization constructs provided by the platform he is working on, or he may use a transactional memory implementation to ensure atomicity [8]. Several different variations of atomicity have been defined and investigated in the literature. In this study, we propose and investigate a new notion of atomicity called *rollback atomicity* that addresses certain limitations of existing definitions of atomicity.

Definitions of atomicity have the following general form:

Definition 1 (Atomicity). *A concurrent execution σ^{conc} of a program is* atomic *iff there exists an equivalent, serial execution σ^{ser} in which actions of each atomic block are executed consecutively.*

Rollback atomicity follows this template as well. Rollback atomicity requires a particular kind of match between the states of σ^{conc} and the witness execution σ^{ser} at certain points in each execution. These points approximately correspond to completion points of atomic blocks. A subset F of the shared data variables is designated by the user as the set of *focus variables*. The rest of the shared variables (the set P) are called *peripheral* variables. The valuation of focus variables in σ^{ser} right after an atomic block **A** completes is required to match a valuation obtained from σ^{conc} by (i) considering the program state at the point where **A** completes in σ^{conc} and (ii) by "rolling back" the effects of other atomic blocks **B** that commit later, i.e. appear later than **A** in σ^{ser}. Rollback atomicity is defined precisely in Section 3.

Different notions of atomicity in the literature differ in their formalization of the equivalence of executions that they use to interpret the Definition 1. Reduction and its variants [2, 12] are defined based on actions that are left-, right- and both-movers and actions that are non-movers. Reduction requires that it be possible to obtain σ^{ser} from σ^{conc} by swapping actions that commute for σ^{ser} and σ^{conc} to be equivalent. Conflict serializability (See e.g. [10, 12]) requires σ^{ser} to consist of the same accesses as in σ^{conc} and for the order of accesses to each variable to be the same in the two executions. View serializability [10, 12] is a more relaxed notion for atomicity. It requires that σ^{ser} consist of the same accesses as in σ^{conc}, that the final write to each variable in both executions be the same and that the write seen by each read be the same in both executions. Commit atomicity [4] requires that the order of the atomic blocks in σ^{ser} be the same as the order of occurrence of their commit points in σ^{conc}. All quiescent states, i.e., states in which no atomic block is in progress, including the final state of σ^{conc} are then required to match corresponding states of σ^{ser}. Rollback atomicity is a weaker requirement than reduction and conflict serializability, but is incomparable with view serializability and commit-atomicity. It provides more observability at more points along the execution and is therefore more stringent than the latter two criteria, but is more permissive and abstract than them in other regards.

In Section 2.1, we provide an example which we use to motivate and illustrate various aspects of rollback atomicity. In Section 2.2, we review existing non-interference criteria for concurrent programs and provide examples where they produce counterintuitive results. In some of these examples, existing criteria are too restrictive. In others, they miss errors when applied in the context of runtime verification because they only pose a requirement about the end states of executions. We propose the novel criterion of rollback atomicity to address some of these issues. In Section 3, we define rollback atomicity formally. Section 4 describes our runtime algorithm for checking whether a concurrent execution of a Java program is rollback atomic. Section 5 describes our implementation and presents preliminary results.

2 Motivation

2.1 Rollback Atomicity Example

In this example, several concurrent threads can each run the `send` method of a different `Msg` object (See Figures 1 and 2). The `Msg` objects that are to be sent wait in a queue called `toSendQueue`, thus, `toSendQueue` is shared among different threads. The static field `Msg.KBSentThisSec` and the pool of bytes to be sent, `SendPool` are also shared among threads. Each `Msg` object has a boolean field `sent` that indicates whether or not it has been copied into `SendPool`. The `send` method copies the contents of the message (a byte array) to `SendPool` byte by byte. In `SendPool`, each byte has a message identifier (shown by `msg1`, etc. in the leftmost box representing each message in Figure 1) and a sequence number. The programmer wants the modifications of the `sent` fields and the `toSendQueue` to be atomic. While the `sentPool` data structure is also a shared variable, since the network can already re-order messages, it is not necessary for the sequence of updates to `sentPool` by each `send` method to be atomic. The `KBytesSentThisSec` static field is shared (read and written to) by all threads. It is used for rate control and occasionally causes a `send` method to abort, but otherwise, in non-exceptionally-terminating executions of `send`, it does not affect the functionality of the method. This field is reset every second, and is incremented by all threads manipulating `Msg` objects. The user does not need the complete sequence of updates to the `KBytesSentThisSec` field within a single execution of `send` to be atomic. Also, the fact that, of two concurrent executions of `send`, one writes to `KBytesSentThisSec` while the other one reads the written value does not point to a true data dependency between the two executions of `send`. `KBytesSentThisSec` is simply there for rate control.

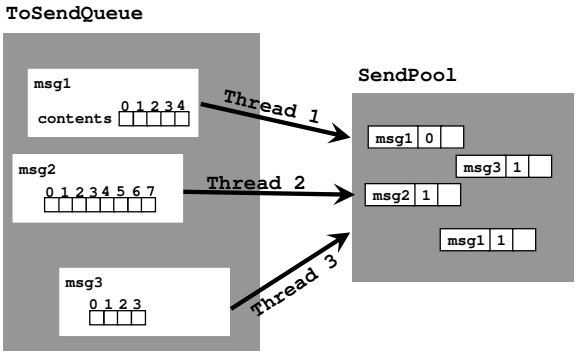

Fig. 1. Concurrent threads and objects in Example 1.

Consider two normally-terminating (i.e., without executing `abort()`) concurrent executions of `send` for `Msg` objects `msg1` and `msg2`. Consider the interleaving of actions by two threads (Figure 3) where an increment that `msg2.send()` being run by thread `T2` performs on `Msg.KBytesSentThisSec` occurs between two updates to the same static field by `msg1.send()` running on thread `T1`. Here, the increment of the static field `Msg.KBytesSentThisSec` is split into three ac-

```
 0: class Msg {
 1:   long msgId;                     /* @Focus  */
 2:   static long KBSentThisSec = 0; /* @Peripheral */
 3:   boolean sent = false;          /* @Focus  */
 4:   byte[] contents;               /* @Focus  */
 5:
 6:   static synchronized long getKBSentThisSecIncr() {
 7:     return ++KBSentThisSec;
 8:   }
 9:
10:   static synchronized long getKBSentThisSec() {
11:     return KBSentThisSec;
12:   }
13:
14:   synchronized atomic void send() {
15:
16:     if ( sent  || !toSendQueue.isIn(this))
17:       abort;  // Caller must retry
18:
19:     if (Msg.getKBSentThisSec() > MAX_RATE)
20:       abort;  // Caller must retry
21:
22:     int i = 0;
23:     while (i < contents.length) {
24:
25:       sendPool.insert(msgId, i, content[i]);
26:         if ( (++i % 1000) == 0 )
27:           if (Msg.getKBSentThisSecIncr() > MAX_RATE)
28:              abort; // Caller must retry
29:     }
30:
31:     sent = true;
32:     toSendQueue.remove(this);
33:   } //Commit point
34: }
```

Fig. 2. Example 1: Focus variables and rollback atomicity

tions (read-modify-write) to emphasize the read-write data dependencies between consecutive increments to this field. These dependencies are important when considering which re-orderings are allowed by different non-interference criteria.

Conflict serializability does not allow the reads and writes to `Msg.KBytesSentThisSec` by `msg1.send()` and `msg2.send()` to be re-ordered in order to obtain a witness serial execution. Likewise, view serializability does not allow such a re-ordering either, as it requires that the value of `Msg.KBytesSentThisSec` seen by each read (results of which are written to local variables `tmp1` and `tmp2`) to be the same in the concurrent and serial executions. Therefore, these two criteria declare such an execution unserializable.

Consider an interleaving of actions from threads in which a message is not sent during a period because `MAX_RATE` is exceeded. For such an execution, at

```
T1                                  T2
--                                  --
send 1000 Bytes of msg1
tmp1 = Msg.KBytesSentThisSec
tmp1 = tmp1 + 1
Msg.KBytesSentThisSec = tmp1
                                    send 1000 Bytes of msg2
                                    tmp2 = Msg.KBytesSentThisSec
                                    tmp2 = tmp2 + 1
                                    Msg.KBytesSentThisSec = tmp2
send 1000 Bytes of msg1
tmp1 = Msg.KBytesSentThisSec
tmp1 = tmp1 + 1
Msg.KBytesSentThisSec = tmp1
```

Fig. 3. A possible interleaving of actions in Example 1.

the end of the one-second period, the value of `Msg.KBytesSentThisSec` does not match the value of `Msg.KBytesSentThisSec` that would have been obtained at the end of any serial execution consisting of only the successful executions of `send`. Thus, such an execution would constitute a commit atomicity violation [4] as well.

However, this execution is consistent with the designer's intentions and is atomic in a certain sense. If only accesses to the shared variables `toSendQueue`, `contents` and `Msg.sent` are considered, these concurrent executions can be serialized in the order that `msg1` and `msg2` are removed from `toSendQueue`. Declaring `toSendQueue`, `contents` and `Msg.sent` as the set of our focus variables while designating the rest of the shared variables as peripheral variables, rollback atomicity provides us a way of expressing the requirement that `toSendQueue`, `contents` and `sent` be updated atomically by each execution of `send` while `Msg.KBytesSentThisSec` only have a consistent value that allows these executions to complete normally.

2.2 Limitations of Other Non-Interference Criteria

Formalizations of atomicity in the literature differ mainly in the notion of equivalence of serial and concurrent executions they use. This section presents a series of scenarios where previous non-interference criteria are too restrictive or permissive in the context of runtime verification.

Reduction and its variants ([2, 12]) are defined based on actions that are left-, right- and both-movers and actions that are non-movers. They require that it be possible to obtain σ^{ser} from σ^{conc} by swapping actions that commute. Reduction-based definitions have the advantage that it is possible to develop type and effect systems based on them that guarantee atomicity statically. Furthermore, at runtime, it is possible to efficiently check sufficient conditions for atomicity formulated based on whether the accesses within an atomic block adhere to a certain pattern. However, reduction by itself is too restrictive. For instance, acquiring and releasing a lock are right and left movers only. As a result, it is not even possible to re-order the two independent reads. Consider the execution fragment in Figure 4 and suppose that the atomic block in `T2` must be serialized first because of the reads and writes to `Z` (e.g. because of a conditional statement

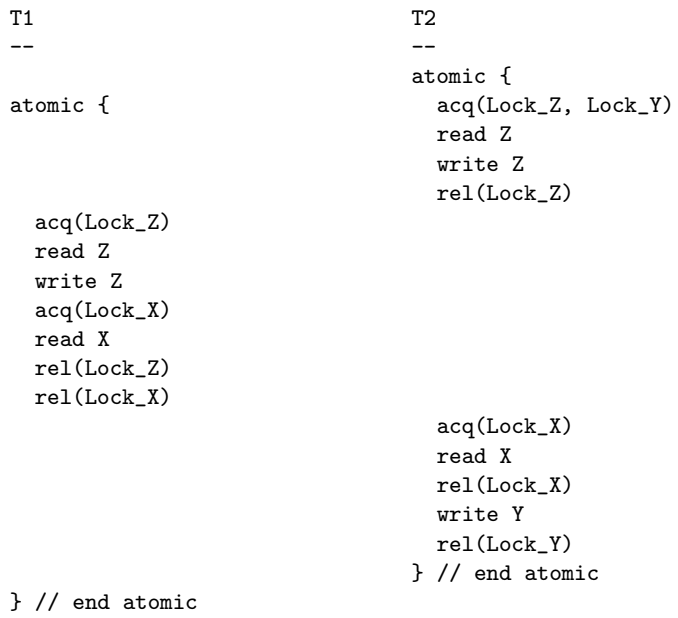

Fig. 4. Demonstrating restrictiveness of reduction. The vertical axis represents time.

in the original program). Intuitively, since the ordering of two lock-protected reads of `X` can be changed while preserving all thread-local and global assertions at any point in the execution it should be possible to commute all actions in `T2` so they occur before all actions of `T1` but reduction does not allow this.

Conflict serializability requires that σ^{ser} consist of the same accesses as in σ^{conc} and that the order of accesses to each variable remain the same. One can formulate a necessary and sufficient check for conflict serializability of an execution based on whether the graph of ordering dependencies between atomic blocks in the execution is acyclic [12]. While it can be checked efficiently, and is more permissive than reduction (e.g. it allows the two lock-protected reads in Figure 4 to be commuted) conflict serializability is still often regarded as too restrictive. For instance, in the execution in Figure 5, the first write to `X` by `T1` cannot be commuted after the accesses by `T2`although no read in this execution sees the first write to `X` by `T1`.

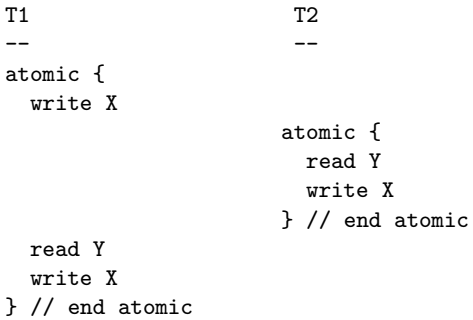

Fig. 5. Example demonstrating restrictiveness of conflict serializability.

Given that multi-processing and multi-threading is becoming more commonplace, one would like to define a notion of atomicity that is as relaxed as possible while still enabling the programmer to reason in a sequential manner about atomic blocks. View serializability is a more relaxed notion for atomicity. It requires that σ^{ser} consist of the same accesses as in σ^{conc}, that the final write to each variable in both executions be the same, and that the write seen by each read be the same in both executions. But there are scenarios in which view serializability is more permissive than desired, and other scenarios in which it is too restrictive. As explained in Section 2.1, in the example in Figure 2 view serializability is unnecessarily restrictive.

```
          T1                T2
        ------            ------
 1:     atomic {
 2:       X = 1           atomic {
 3:                         X = 2
 4:                         Y = 1
 5:                       }// end atomic
 6: ------------------------------------> read X, Y reveals error
 7:       Y = 2           atomic {
 8:     } // end atomic     Z = 0
 9: ------------------------------------> Rollback atomicity fails here
10:     atomic {
11:       X = 0
12:                         Y = 0
13:                       } // end atomic
14:     } // end atomic
```

Fig. 6. Demonstrating added observability provided by rollback atomicity

To see where view serializability is too permissive, consider the example σ^{conc} of a concurrent execution (Figure 6). At step 6 in the execution, the first atomic block in thread `T2` has completed its execution but the values of `X` and `Y` are not consistent with any serial execution. This is undesirable because, in another execution, actions by another thread could have been interleaved at this point and these actions could have observed these inconsistent values of `X` and `Y`. View serializability and commit atomicity (discussed in more detail below) are not able to detect this error. Because there are no reads in this execution, the view serializability requirement amounts to the final writes to each variable in a serial execution σ^{ser} being the same as the ones (`X = 0, Y = 0`) in Figure 6.

This example illustrates the fact that view serializability of a single execution may not provide enough observation points along the execution to reveal bugs. Requiring that all executions of a concurrent program be view serializable does not suffer from this observability limitation, but, for a single execution, view serializability may miss some obvious errors.

Purity[3] can be seen as a way of relaxing the requirements of reduction by (i) disregarding accesses by normally-terminating pure blocks and, in essence, allowing them to commute with all other accesses, (ii) allowing one to reason about

the atomicity of a more abstract version of the program in which reads within normally-terminating pure blocks can return non-deterministic values, and (iii) disregarding accesses to "unstable variables" such as performance counters which do not factor into the correctness requirement for the program and which may have races on them. If purity is used, the programmer, while performing sequential reasoning within an atomic block, must argue the correctness of a more abstract program where pure reads and accesses to unstable variables may return non-deterministic values. In the example in Figure 2, although it is not required for correctness for updates to `Msg.KBSentThisSec` to be atomic with the rest of the shared variables, the program will not work as intended for completely arbitrary values of `Msg.KBSentThisSec`. Therefore, making `Msg.KBSentThisSec` an unstable variable and marking the atomic block in Figure 2 will not give the desired semantics. The requirements on `Msg.KBSentThisSec` are more stringent than those on unstable variables.

Harris et al. [5] propose abstract nested transactions (ANT) as a mechanism to recover at runtime from a subset of benign conflicts between atomic blocks. ANT's still implement serializability but provide a mechanism for trying more than one witness serialization σ^{ser}. Furthermore, for the ANT recovery mechanism to be valid, the user is required to access each shared variable either only within or only outside an ANT. Therefore, even when ANT's are used, the example in Figure 2 would create a conflict that the runtime system cannot recover from.

In commit atomicity[4], the final state of a concurrent execution is required to match that of an execution in which atomic blocks are run one at a time, in the order of the occurrence of their commit points in the concurrent execution. Commit atomicity comes closest to the definition of atomicity proposed in this paper, however, it is too restrictive since it requires that the entire program states match at the end of the execution (which would include `Msg.KBytesSentThisSec` in the example in Figure 2.). Also, commit atomicity requires that a commit point (a program location) for each atomic block be determined and marked explicitly by the programmer. Rollback atomicity does not have this requirement.

All atomicity criteria discussed in this paper, including rollback atomicity, require the existence of a single σ^{ser} in which all the actions of *every* atomic block in the concurrent execution σ^{conc} occur consecutively. This has the important benefit that the programmer can reason about his program (or at least a well-defined subset of variables) as if each atomic block is executed sequentially and by considering thread interleavings only at the atomic block level. Causal atomicity, introduced in [1], is different in that, for a given concurrent execution σ^{conc}, it is not necessary to find a single execution σ^{ser}. Instead, to show causal atomicity for a given σ^{conc}, it suffices to produce a separate witness execution $\sigma^{conc}_{\mathbf{A}}$ for each atomic block $\mathbf{A}$. We believe this makes causal atomicity too relaxed a non-interference criterion and difficult to use as a programming abstraction.

In the next section, rollback atomicity is formulated as a special case of view refinement [13]. This formulation makes clear that rollback atomicity, in a way similar to linearizability [6] and commit-atomicity [4], does not require the serial execution to be obtained by commuting actions of the concurrent execution.

3 Rollback Atomicity

3.1 Preliminaries

We focus on well-synchronized Java programs whose executions are free of race conditions and thus sequentially consistent. In our model of programs and executions, *Tid* represents the set of thread identifiers and *Addr* represents the set of object identifiers. Objects created during executions of a Java program have a unique identifier from the set *Addr*. Each object has a finite collection of fields. *Field* represents the set of all fields. A *variable* is a pair (o, f) consisting of an object o and a field f. V denotes the set of variables. Each thread executes a sequence of actions. An execution of a program is a linearly-ordered finite sequence of actions obtained by interleaving the sequences of actions from each thread. Actions $read(o, f)$ and $write(o, f)$ represent a read of and a write to the field f of an object o respectively.

We suppose that the programmer has annotated certain code blocks as *atomic*. We make the assumption of *strong atomicity*, where each action executed outside an atomic block is interpreted to be an atomic block consisting of a single action. We partition the set of variables V into the sets of *focus* and *peripheral* variables: $F \cup P$. In our implementation described in Section 5 this is accomplished by annotating field definitions of a class with the comments `@Focus` or `@Peripheral`. All variables (o, f) where f is a field marked `@Focus` are in the set F. The rest of the variables are peripheral variables, i.e., in P.

Consider a concurrent execution σ^{conc} of a program with a set of atomic code blocks *AtBlk*. We give every execution of an atomic block that occurs in σ^{conc} a unique identifier from the set *XId*. Rollback atomicity of σ^{conc} requires the existence of a serial execution corresponding to σ^{conc} and satisfying certain conditions. In order to make these conditions precise, we make use of the commit order and the rollback function, explained below.

The *commit order* on *XId* is a total order defined as follows: $\alpha \leq_{cmt} \beta$ iff $\alpha = \beta$ or the atomic block execution with the identifier α occurs before β in σ^{ser}. The commit order is uniquely determined by σ^{ser}. We omit a reference to σ^{ser} while referring to the commit order in order to keep the notation simple when the serial execution σ^{ser} is clear from the context. If $\alpha \leq_{cmt} \beta$, we informally say that α *commits before* β and β *commits after* α.

The *rollback function RlBk* is defined for certain states along a concurrent execution – those reached right after an atomic block with identifier α performs its last write to a focus variable. Roughly speaking, *RlBk* has the effect of undoing actions by other atomic blocks that overlap with atomic block α in the concurrent execution but come after α in the commit order. *RlBk* is formally defined as follows. Let σ_α^{conc} be the state in σ^{conc} right after the atomic block execution with identifier α has performed the last write access to a variable in F. We denote with $\mathbf{s}_\alpha^{\mathbf{conc}} = proj(\sigma_\alpha^{conc}, F)$ be the projection of the state σ_α^{conc} onto the focus variables. Let σ^{ser} be the serial execution on which $\leq_{cmt}$ is defined. Then, for every focus variable (o, d), $RlBk(\mathbf{s}_\alpha^{\mathbf{conc}}, \sigma^{conc}, \sigma^{ser}, F)((o, d))$ is the value of the last write in σ^{conc} by an atomic block β such that $\beta \leq_{cmt} \alpha$ or the initial value of (o, d) if no such α exists. Note that if α has performed the last write to

(o, d) before $\mathbf{s}$, then $RlBk(\mathbf{s}_\alpha^{\mathbf{conc}}, \sigma^{conc}, \sigma^{ser}, F)((o, d))$ simply returns this value. Rollback atomicity makes use of the rollback function to formulate a refinement check associated with each atomic block and to ensure that the writes to each focus variable (o, d) occur in σ^{conc} occur in the commit order.

3.2 Rollback Atomicity Definition

We say that a concurrent execution σ^{conc} is *rollback atomic* iff there exists an execution σ^{ser} of the program that satisfies the following conditions:

(i) For each thread t, the projection of the two executions onto t, $proj(\sigma^{ser}, t)$ and $proj(\sigma^{conc}, t)$, consist of the same sequence of atomic blocks. We can therefore use the same identifier from XId to refer to corresponding occurrences of an atomic block execution by the same thread in σ^{ser} and σ^{conc}.

(ii) Let σ_α^{ser} denote the state of σ^{ser} right after the block with identifier α has completed executing. Let $\mathbf{s}_\alpha^{\mathbf{ser}} = proj(\sigma_\alpha^{ser}, F)$ be the projection of the state σ_α^{ser} onto the focus variables and let $\mathbf{s}_\alpha^{\mathbf{conc}}$ be as defined above. It must be the case that for each α, $RlBk(\mathbf{s}_\alpha^{\mathbf{conc}}, \sigma^{conc}, \sigma^{ser}, F)((o, d)) = \mathbf{s}_\alpha^{\mathbf{ser}}((o, d))$.

Rollback atomicity is a special case of view refinement and can be seen as a variant of linearizability [6] where part of the data structure state is projected away. Similarly to other non-interference criteria, rollback atomicity only requires the existence of some serial execution σ^{ser} or, equivalently, a commit order. The next section describes how we use the dependencies between actions to infer a commit order heuristically.

4 Checking Rollback Atomicity

We check runtime atomicity of executions by performing a view refinement check as described in [13]. The abstraction function that realizes the rollback atomicity check is *RlBk* as described above. The view refinement check requires that the order of atomic blocks in σ^{ser} be explicitly provided by the user. In order to allow more flexibility in the choice of this order, we attempt to infer it from causality relationships in the execution. We construct two graphs representing causality dependencies between accesses in order to infer this order: $\mathbf{CG}_F$ and $\mathbf{CG}_{F \cup P}$. The rules for constructing the two graphs are the same and explained below. The former graph is constructed only using accesses to F variables while the latter takes into account accesses to all shared variables.

A *causality graph* of an execution σ^{conc} is a directed graph $G = (V, E)$ where V contains a unique vertex for each atomic block in the execution σ^{conc} and a unique vertex for each individual read and write action occurring in σ^{conc}. E consists of the following edges:

(i) For each read action r and the write that it sees, $W(r)$ (see [9] for a formal definition), E contains an edge $(W(r), r)$ from the node representing $W(r)$ to the node representing r. If $W(r)$ (alternatively, r) is part of atomic block with identifier α then the edge starts (alternatively, ends) at the node representing the atomic block α instead. The case when both $W(r)$ and r are in the same atomic block is handled in (iii) below.

(ii) For each read action r and the next write action w to the same variable after r in the concurrent execution σ^{conc}, E contains an edge (r, w) from the node representing r to the node representing w. If r (or w) is part of an atomic block α, the edge starts (or ends) at the node representing atomic block α. The case when both $W(r)$ and r are in the same atomic block is handled below.

(iii) If an atomic block contains a write w to a variable (o, d) and subsequent reads r_1, r_2, ..., r_k of (o, d) with no write to (o, d) in the atomic block between w and $r_1, ..., r_k$, E contains the edges (w, r_1), (w, r_2), ...,(w, r_k).

(iv) For each pair of nodes α and β representing actions or atomic blocks ordered by program order, E contains an edge from α to β.

We incrementally update $\mathbf{CG}_F$ and $\mathbf{CG}_{F \cup P}$ as each access in the execution σ^{conc} is processed in order. We search for cycles in each graph after adding an edge that starts/ends at a node representing an atomic block [11]. At each such point, there are three possibilities:

(i) Neither $\mathbf{CG}_F$ nor $\mathbf{CG}_{F \cup P}$ have a cycle containing an atomic block. In this case, we obtain a commit order of atomic blocks by applying the algorithm in [11] to $\mathbf{CG}_{F \cup P}$. In this case the entire execution up to this point is conflict-serializable and it is not necessary to perform a rollback atomicity check.

(ii) $\mathbf{CG}_{F \cup P}$ has a cycle containing an atomic block but $\mathbf{CG}_F$ does not. In this case, we obtain a commit order by linearizing $\mathbf{CG}_F$ only.

(iii) $\mathbf{CG}_F$ and $\mathbf{CG}_{F \cup P}$ both have cycles. In this case, we take as the commit order the order of the last focus variable writes by atomic blocks.

When the refinement check fails, this means we were not able to obtain a serialized execution satisfying the requirements of rollback atomicity using the commit orders provided by (i)-(iii) above. In this case, the implementation could truly have undesired behavior, or it could be the case that we were not able to find the right commit order. If the latter is the case, however, the commit order conflicts with some causality dependencies between atomic blocks created by reads and writes to focus variables. In this case, the programmer can aid our atomicity check by explicitly providing commit point annotations which may produce a commit order that conflicts with the dependencies captured in the causality graphs above. The programmer may also want to revise the partition of variables into focus and peripheral variables.

If there is a read action r to which there is more than one causality edge from write actions in $\mathbf{CG}_F$, a warning is declared. This warning corresponds to a case where a read in an atomic block should have seen a write within the same atomic block, but, instead, has seen another write from another atomic block. While this does not necessarily correspond to a rollback atomicity violation (e.g. the two writes may have written the same value), we point this case out to the programmer as it is probably unintended. Programmers declare blocks atomic in order to be able to perform sequential reasoning. The sequential reasoning is limited to focus variables in the case of rollback atomicity. The scenario described corresponds to a case where this sequential assumption is broken.

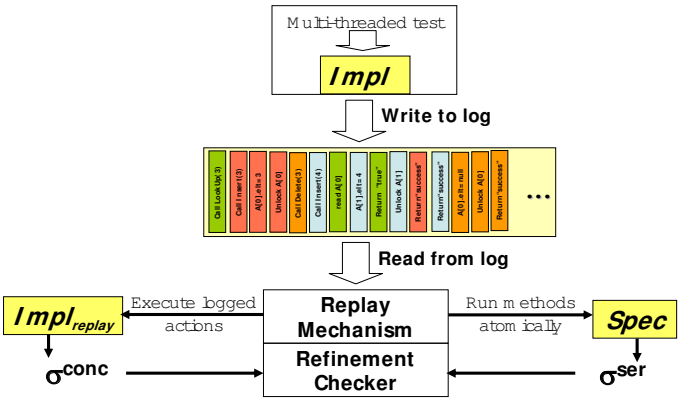

Fig. 7. Checking refinement using VYRD.

5 The Implementation

The runtime algorithm for checking rollback atomicity described in the previous section was implemented using the infrastructure built for the VYRD tool [13] which makes use of Java Pathfinder [7]. The block diagram depicting the refinement checking approach is reproduced in Figure 7. While the program runs and produces σ^{conc}, VYRD tracks the atomic blocks and accesses to the shared variables by these blocks throughout the execution. The beginning and the end of the atomic blocks are annotated by the method calls `Vyrd.beginAtomicBlock()` and `Vyrd.endAtomicBlock()`.

VYRD writes all the events during an execution (including instruction execution, object/thread creation and destruction) into a log file. In offline mode, after the execution completes, VYRD replays the execution using the log and detects the accesses to the focus variables and peripheral variables. The configuration file for VYRD includes the annotations for the focus variables.

Our current implementation only constructs the graph $\mathbf{CG}_F$ and computes a commit order corresponding to the order of the last writes to focus variables by atomic blocks. For atomic blocks that access no focus variables, their place in the commit order is chosen simply based on their order of occurrence in σ^{conc}. In addition, VYRD allows programmers to explicitly annotate commit points of atomic blocks by calling the special method `Vyrd.commit()` in the code. Using either method, VYRD is able to identify the commit point of the atomic block and perform the refinement check while replaying the execution from the log file. During the replay of the concurrent execution, on a separate instance of the program being verified, atomic blocks in the corresponding sequential execution are executed in the commit order. $\mathbf{s}_\alpha^{\mathbf{ser}}$ is computed on this separate copy.

VYRD provides a modular mechanism by which a separate software module observes log entries, computes a commit order and drives the execution of atomic blocks in σ^{ser}. Algorithms for incremental construction of and cycle detection for $\mathbf{CG}_F$ and $\mathbf{CG}_{F \cup P}$ will be a part of this external module that feeds the inferred commit order to VYRD. We have not implemented this feature yet.

Using the implementation outlined above, we were able to compute a commit order and verify rollback atomicity for the motivating example in Figure 2 for 10 concurrent threads and message lengths of 50.

6 Conclusion

We proposed rollback atomicity, a new notion of atomicity that we believe is a useful and natural non-interference criterion. We presented a runtime algorithm for checking rollback atomicity and reported on a preliminary implementation. Future work includes integrating cycle detection and the refinement checking implementation, trying other heuristics for inferring the commit order, and relaxing the strong atomicity assumption in the rollback atomicity definition in order to allow less stringent requirements on code blocks not marked atomic.

References

1. A. Farzan, P. Madhusudan. Causal Atomicity. In *Proc. Computer Aided Verification (CAV) 2006. Springer Lecture Notes in Computer Science*, Vol. 4144: 315–328.
2. C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In Proc. 31st ACM Symposium on Principles of Programming Languages. pp. 256–267, 2004.
3. C. Flanagan, S. Freund, and S. Qadeer. Exploiting Purity for Atomicity. In Proc. the Intl. Symposium on Software Testing and Analysis (ISSTA 2004).
4. C. Flanagan Verifying Commit-Atomicity Using Model Checking 11th Intl. SPIN Workshop, Barcelona, Spain, April 1-3, 2004. Springer Lecture Notes in Computer Science 2989, pp. 252–266.
5. T. Harris and S. Stipic Abstract Nested Transactions The 2nd ACM SIGPLAN Workshop on Transactional Computing, TRANSACT '07. Available at `http://www.cs.rochester.edu/meetings/TRANSACT07/papers/harris.pdf`
6. M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, Vol. 12 No. 3: pp. 463–492, 1990.
7. W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. Model Checking Programs Automated Software Engineering Journal. Vol. 10, No. 2, April 2003.
8. J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
9. J. Manson and W. Pugh and S. Adve. The Java Memory Model. In *Proc. POPL 05: Principles of Programming Languages.* ACM Press, pp. 378–391, 2005.
10. C. Papadimitriou The Serializability of Concurrent Database Updates *Journal of the ACM, 26 (4)*, ACM Press, pp. 631–653, 1979.
11. D. J. Pearce, P. H. Kelly, and C. Hankin. Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis. *Software Quality Control* Vol. 12, No. 4, Dec. 2004, 311-337.
12. L. Wang and S. D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs In *PPoPP '06: Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 137–146.
13. T. Elmas, S. Tasiran, and S. Qadeer. Vyrd: Verifying Concurrent Programs by Runtime Refinement-Violation detection. In *PLDI '05: Proc. 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 27–37.