A Causality-Based Runtime Check for (Rollback) Atomicity

Serdar Tasiran Tayfun Elmas
Koc University, Istanbul, Turkey
{stasiran,telmas}@ku.edu.tr

Atomicity is used widely for expressing interference-freedom requirements between code blocks in executions of concurrent programs. In this study, we propose a new notion of atomicity called "rollback atomicity".

Commonly used definitions of atomicity have the following form: *A concurrent execution σ^{conc} of a program is* atomic *iff there exists an equivalent, serialized execution σ^{ser} in which every* `atomic` *block by thread* t *is executed with no interruption by actions of other threads.* Rollback atomicity fits into this template as well. In rollback atomicity, we require a particular kind of match between the states of σ^{conc} and the witness execution σ^{ser} at certain points in each execution approximately corresponding to completion points of atomic blocks. A subset F of the shared data variables is designated by the user as the "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 $\mathbf{A}$ completes is required to match a valuation obtained from σ^{conc} by (i) considering the program state at the point where $\mathbf{A}$ completes in σ^{conc} and (ii) by rolling back the effects of other atomic blocks $\mathbf{B}$ that commit later, i.e. appear later than $\mathbf{A}$ in σ^{ser}.

Formalizations of atomicity in the literature differ in the notion for the equivalence of executions that they use to interpret the definition above. Reduction and its variants ([2, 5]) 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. 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. 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. In commit atomicity[3], 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. 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, but is more permissive in other regards. We highlight some of the differences in Figure 1 where we provide an example in which view-serializability as well as reduction and conflict-serializability are unnecessarily restrictive.

In this example, several concurrent threads can each run the `send` method of a different `Msg` object. 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 it has been sent or not. The `send` method copies the contents of the message (a byte array) to a pool byte by byte where each byte has a message id 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 this field within a single execution of `send` to be atomic. Also, the read-write dependencies between concurrent executions of `send` caused by this field are not really significant. They do not really point to a data dependency between the atomic blocks.

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

Figure 1. Example 1: Focus variables and rollback atomicity

Consider two normally-terminating concurrent executions of `send` for `Msg` objects `m1` and `m2`. Suppose that an increment that `m1.send()` performs on `Msg.KBytesSentThisSec` is interleaved between two updates to the same static field by `m2.send()` running on another thread. Conflict serializability does not allow the reads and updates to `Msg.KBytesSentThisSec` by `m1.send()` and `m2.send()` to be re-ordered. Likewise, view-serializability does not allow such a re-ordering either, as it requires that the value of `Msg.KBytesSentThisSec` seen by each increment operation to be the same in the concurrent and serial executions. Therefore, these two criteria declare such an execution unserializable. It is possible to construct other interleavings for which commit atomicity would flag a warning because the value of `Msg.KBytesSentThisSec` at the end of a serial execution does not match that at the end of a concurrent one. However, in terms of the other shared variables (`toSendQueue`, `contents` and `Msg.sent`) these concurrent executions can be serialized in the order that `m1` and `m2` are removed from `toSendQueue`. Declaring this latter set of variables as our focus variables while designating the rest of the shared variables as peripheral variables, rollback atomicity gives 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.

In the full paper, we provide examples where view serializability and commit atomicity of a single execution may not provide enough observation points along the execution to reveal bugs. In these examples, rollback atomicity provides the necessary early warning.

1. Rollback Atomicity

We focus only on well-synchronized Java programs whose executions are free of race conditions and thus sequentially consistent. We suppose that the programmer has annotated certain code blocks as atomic. We work with *strong atomicity*, where all other actions modifying or reading focus variables are considered to be atomic blocks as well. Our definition is based on a partition of the set of shared variables into focus and peripheral variables: $F \cup P$.

Consider a concurrent execution σ^{conc} of a program with a set of atomic code blocks $AtBlk$. Let us suppose that a every execution of an atomic block that occurs in σ^{conc} is given a unique id from the set XId. We say that σ^{conc} is *rollback atomic* iff there exists an execution σ^{ser} of the program with the following properties

- 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 for each thread id t. Exploiting this fact, we use the same id from XId to refer to corresponding occurrences of an atomic block execution by the same thread in σ^{ser} and σ^{conc}. Let us define the *commit order* on XId as follows: $\alpha \leq_{cmt} \beta$ iff $\alpha = \beta$ or α occurs before β in σ^{conc}. If $\alpha \leq_{cmt} \beta$, we say that α *commits before* β.
- Let σ^{ser}_α denote the state of σ^{ser} right after the block with id α has completed executing. Let $\mathbf{s}^{\mathbf{conc}}_\alpha = proj(\sigma^{ser}_\alpha, F)$ be the projection of the state σ^{ser}_α onto the focus variables. Let σ^{conc}_α be the state in σ^{conc} right after atomic block execution with id α has performed the last write access to a variable in F. Let $\mathbf{s}^{\mathbf{ser}}_\alpha = proj(\sigma^{conc}_\alpha, F)$ be the projection of the state σ^{conc}_α onto the focus variables. Let $RlBk(\mathbf{s}^{\mathbf{ser}}_\alpha)$ be obtained from $\mathbf{s}^{\mathbf{ser}}_\alpha$ as follows:
  - Let $v \in F$. If v was last written by a transaction $\beta \leq_{cmt} \alpha$ in σ^{conc} then $RlBk(\mathbf{s}^{\mathbf{ser}}_\alpha)(v) = \mathbf{s}^{\mathbf{ser}}_\alpha(v)$.
  - Otherwise, if v was last modified before σ^{conc}_α by an atomic block that commits before α, then find the most recent write ω to v in σ^{conc} by a block that has committed before α. $RlBk(\mathbf{s}^{\mathbf{ser}}_\alpha)(v)$ is assigned the value written by this most recent committed write. If no such write exists, the value of v is set to its initial value.

  We require that for each α, $RlBk(\mathbf{s}^{\mathbf{ser}}_\alpha)(v) = \mathbf{s}^{\mathbf{conc}}_\alpha(v)$.

2. Checking Rollback Atomicity

Using the infrastructure built for the Vyrd tool [6] we track the accesses to the shared variables throughout the execution. We perform a view refinement check as described in [6] where the abstraction function is given by $RlBk$ 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 instead try to infer it from causality relationships. We construct two graphs of 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. The former is constructed only using accesses to F variables while the latter uses accesses to all shared variables.

In this directed graph, called the causality graph, each `atomic` block in the execution and each individual read and write action correspond to a unique node. The graph has the following sets of edges:

- For each read action r and the write that it sees, $W(r)$, an edge from the node representing $W(r)$ to the node representing r. If any of these actions happen inside some atomic block, the edge starts/ends at the node representing that atomic block.
- For each read action r and the write action w to the same variable that happens immediately after r, there is an edge from the node representing r to the node representing w. If any of these actions happen inside some atomic block, the edge starts/ends at the node representing that atomic block.
- Within each atomic block, if the block contains a write to and a subsequent read of the same variable, there is an edge to each read action from the last write to the same variable in the same atomic block.
- For each pair of nodes α and β representing actions or atomic blocks ordered by program order, there is an edge from α to β.

We update $\mathbf{CG}_F$ and $\mathbf{CG}_{F\cup P}$ as we processes each access in order, by iteratively adding nodes and edges. We search for cycles in each graph after adding an edge that starts/ends at a node representing an `atomic` block [4]. At each such point, there are three possibilities:

- 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 [4] to $\mathbf{CG}_{F\cup P}$. In this case the entire execution is conflict-serializable and it is not necessary to perform a rollback atomicity check.
- $\mathbf{CG}_{F\cup P}$ has a cycle containing an atomic block but $\mathbf{CG}_F$ does not. In this case, we obtain a witness order using $\mathbf{CG}_F$ only.
- $\mathbf{CG}_F$ 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.

If there is a read action r to which there is more than one causality edge from write actions in $\mathbf{CG}_F$, an error is declared. This latter warning captures a form of interference in the atomic block that will be discussed in the full paper.

In case our algorithm declares a warning, we were not able to obtain a serialized execution satisfying the rollback atomicity check. 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 ordering of the atomic blocks. If the latter is the case, however, the witness ordering conflicts certain causality dependencies between focus variables. The programmer can aid our atomicity check by explicitly providing commit point annotations.

References

[1] F. Chen and G. Roşu. Predicting Concurrency Errors at Runtime using Sliced Causality. Technical Report: UIUCDCS-R-2005-2660. Department of Computer Science, University of Illinois at Urbana-Champaign. 2005.

[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 Verifying Commit-Atomicity Using Model Checking Model Checking Software. 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings. Lecture Notes in Computer Science 2989, pp. 252–266.

[4] D. J. Pearce, P. H. Kelly, and C. Hankin. Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis. *Software Quality Control* 12, 4 (Dec. 2004), 311-337.

[5] 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.

[6] T. Elmas, S. Tasiran, and S. Qadeer. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN Conference on Programming language Design and Implementation*, pp. 27–37.