Notes about Atomicity*

Venkatesh Prasad Ranganath
venkateshprasad.ranganath@gmail.com

April 27, 2006
revised: July 3, 2006

Recently there has been numerous efforts in detecting atomicity in programs. Most of these efforts have been focused on verification of atomicity by leveraging atomicity annotations either at compile-time [4, 3] or at runtime [2]. There have also been efforts focused on static [5] and dynamic [6] detection of atomicity violations without relying on annotations. Most of these efforts have been focused on method level atomicity.

In this informal notes, I will provide a non-exhaustive exposition about low-level atomicity and introduce three weaker notions of atomicity along with *sealing*, a special case of conditional atomicity. I will then express these notions in the static context in terms of program dependences. These notions can be trivially realized by leveraging the features of Indus,[1] a program analysis and transformation toolkit.

1 Atomicity

Quoting from [2],

> *A statement sequence s executed by thread t is **atomic** if its execution is not affected by and does not interfere with concurrently-executing threads.*

If the statement sequence forms the body of a method, then the method is *atomic*.

We can paraphrase the above notion in terms of reading and writing of shared data.

Definition 1 (Strong Atomicity) A non-singleton statement sequence[2] *s* executed by thread *t* is **strongly atomic** if its execution neither reads nor writes shared data (or only reads and writes thread local data).

In Figure 1, the statements of method `sqAtomic()` form an atomic sequence. To see why, suppose the system starts with `o`, an instance of `Number1`, and threads t_1 and t_2 ready to execute `sqAtomic()` and `set(4)` on `o`, respectively.

In any schedule of the above system, after the value of `i` is assigned to `m` in t_1, the execution of statements of `set` by thread t_2 does not affect the result of the computation

*If any of the contents of this notes has already been published by other authors, then please email me the details about such publications and I will be glad to update these notes.

[1] http://indus.projects.cis.ksu.edu

[2] A sequence with more than one element is referred to as *non-singleton* sequence.

```
1  class Number1 {
2    int i = 5;
3
4    int sqNonAtomic() {
5      int j = i;
6      int k = i;
7      int l = j * k;
8      return l;
9    }
10   int sqAtomic() {
11     int m = i;
12     int n = m * m;
13     return n;
14   }
15
16   void set(int p) {
17     i = p;
18   }
19 }
```

Figure 1: Atomicity Example

embodied in `sqAtomic()` in thread t_1, i.e. `n` will be equal to either `4*4=16` or `5*5=20` in `sqAtomic`. The key observation being that, after the execution of `m=i` in `sqAtomic()`, the data *read* and *written* by the statement sequence in `sqAtomic()` is local to the executing thread. By similar reasoning, the sequence `k=i; l=j*k;` in `sqNonAtomic` can be classified as being `atomic`.

In contrast, the statements of method `sqNonAtomic()` do not form an atomic sequence. To see why, in the above system, suppose that thread t_1 executes `sqNonAtomic()` instead of `sqAtomic()`.

In this case, if the execution of line 17 by thread t_2 is *not* interleaved between the execution of lines 5 and 6 by thread t_1, then the result of the computation embodied in `sqNonAtomic()` executed by thread t_1 will be identical across all schedules, i.e. `l` will be equal to either `16` or `25`. However, when line 17 executed by thread t_2 is interleaved between lines 5 and 6 executed by thread t_1, `l` will be equal to `5*4=20`. The key observation in being that, after executing `j=i`, the data *read* by the statement sequence in `sqNonAtomic` is not local to the executing thread.

2 First-read Atomicity

Suppose `Number1` class in Figure 1 is modified into class `Number2` as shown in Figure 2. In `sqAtomic` method, the value of `n` is assigned to the instance field `i`, hence, affecting any concurrent computation that is dependent on the value of `i`.

According to the earlier definition, `Number2.sqAtomic` method is non-atomic. The difference between `Number1.sqAtomic` and `Number2.sqAtomic` is that the assignment (write) `i=n` in the `Number2.sqAtomic` interferes with the concurrent execution of `Number2.sqNonAtomic`. However, this change does not affect the thread local behavior of `Number2.sqAtomic`.

In essence, independent of the interleavings in which the statement sequence in `Number2.sqAtomic` are executed, the resulting thread local state will be identical to that resulting by executing the sequence in a single step (serially). This new notion of atomicity based *solely on the absence of reading* shared data in the non-first statements of a statement sequence is referred to as *first-read atomicity*.

```
1  class Number2 {
2    int i = 5;
3
4    int sqNonAtomic() {
5      int j = i;
6      int k = i;
7      int l = j * k;
8      return l;
9    }
10   int sqAtomic() {
11     int m = i;
12     int n = m * m;
13     i = n;
14     return n;
15   }
16
17   void set(int p) {
18     i = p;
19   }
20 }
```

Figure 2: First-read Atomicity Example

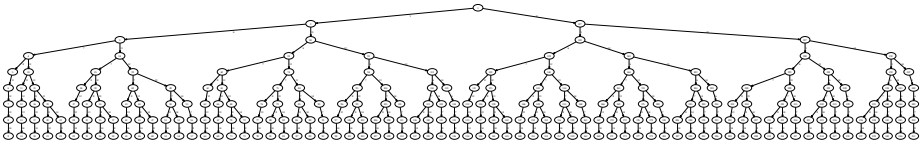

Figure 3: Atomicity based unreduced state space when `Number2.sqNonAtomic()` and `Number2.sqAtomic()` are executed concurrently.

Definition 2 (First-Read Atomicity) A non-singleton statement sequence *s* executed by thread *t* is **first-read atomic** if none of the non-first statements in *s* **read** shared data.

2.1 Implications

To understand the implications in a dynamic context, let us consider a program that creates an instance `o` of `Number2` class (from Figure 2) and in which thread t_1 invokes `sqNonAtomic()` on `o` while thread t_2 invokes `sqAtomic()` on `o`.

If this program was being model checked using the notion of strong atomicity, then only lines 7 and 8 will form an atomic sequence. This will yield a state space as shown in Figure 3. On the other hand if the model checker used the notion of first-read atomicity, lines 6, 7, and 8 and lines 11, 12, 13, and 14 will form the atomic sequences. This will yield a state space as shown in Figure 4. Clearly, the latter state space contains lesser states, hence, it will be cheaper to model check.

However, the above approach fails when `sqAtomic` is extended by adding the statement `i=n+1;` immediately after `i=n;` as shown in Figure 5. By first-read atomicity, thread t_2 will only observe the value `4*4+1=17)` of `i` (line 14) and it will not observe the value `4*4=16` (line 13). The key observation being that first-read atomicity prohibits the interleaving of reads of shared data between consecutive write to the same shared data. Such data hiding may render the result of dynamic analysis such as model checking to be unsound.

In the context of testing concurrent software, given a statement sequence *s* executed by thread t_1, incoming influences (via concurrent writes of shared data by threads other

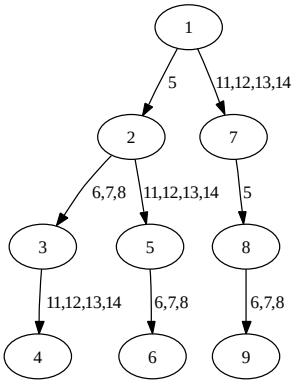

Figure 4: First-read atomicity based unreduced state space when `Number2.sqAtomic()` and `Number2.sqNonAtomic()` are executed concurrently by threads t_1 and t_2, respectively.

```
1  class Number3 {
2    int i = 5;
3
4    int sqNonAtomic() {
5      int j = i;
6      int k = i;
7      int l = j * k;
8      return l;
9    }
10   int sqAtomic() {
11     int m = i;
12     int n = m * m;
13     i = n;
14     i = n + 1;
15     return n;
16   }
17
18   void set(int p) {
19     i = p;
20   }
21 }
```

Figure 5: First-write Atomicity Example

than t_1) on s are relevant to test s but not the outgoing influence of s (via concurrent writes of shared data by s in t_1). Also, longer sequences imply fewer number of test cases. Hence, first-read atomicity is a good choice to trivially optimize specifying and testing concurrent software. However, this choice is *directional*, i.e. sequence s should be considered atomic (according to first-read atomicity) when specifying/testing only s.

3 First-write Atomicity

To address the shortcomings of first-read atomicity in dynamic context, we can introduce a new notion atomicity *based solely on the absence of writing* shared data in non-first statements of a statement sequence and this notion is referred to as *first-write atomicity*.

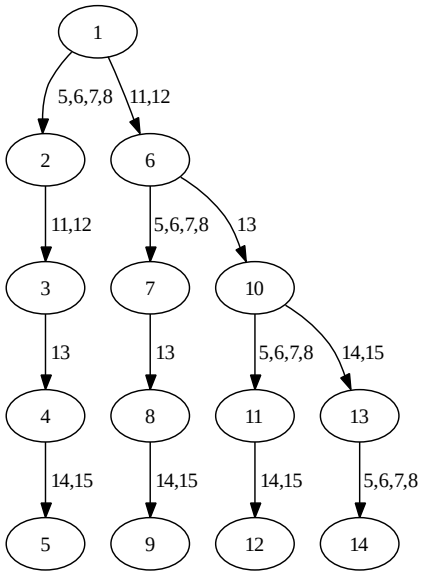

Figure 6: First-write atomicity based unreduced state space when `Number2.sqNonAtomic()` and `Number2.sqAtomic()` are executed concurrently.

Definition 3 (First-Write Atomicity) A non-singleton statement sequence *s* executed by thread *t* is **first-write atomic** if none of the non-first statements in *s* **write** shared data.

Hence, a dynamic analysis such as model checking can consider each first-write atomic sequences as a single step. In the code in Figure 5, lines 5, 6, 7, and 8, lines 11 and 12, and lines 14 and 15 will form the first-write atomic sequences.

Interestingly, the first-write atomicity *incorrectly* forces `j` and `k` in `sqNonAtomic` to take on the same value of `i` even when line 19 of `set(int)` can be interleaved between line 5 and line 6 in a concurrent scenario. The reason being that first-write atomicity does not allow writes of shared data to be interleaved between consecutive reads of the same shared data.

4 Weak Atomicity

The above shortcomings is addressed by *weak-atomicity*.

Definition 4 (Weak Atomicity) A non-singleton statement sequence *s* executed by thread *t* is **weakly atomic** if none of the non-first statements in *s* read or write shared data.

In contrast with atomicity, the weakly atomic sequences in various examples are given below.

- Figure 1: line 5, line 6, 7, and 8, lines 11, 12, and 13.
- Figure 2: line 5, line 6, 7, and 8, lines 11 and 12, and lines 13 and 14.
- Figure 5: line 5, line 6, 7, and 8, lines 11 and 12, line 13, line 14 and 15.

5 Locks

If we model lock acquisition as a blocking data read and lock release as a data write, then the above notions can be trivially extended to handle sequences containing lock acquiring and releasing statements.

Locking Patterns Most efforts pertaining to atomicity consider locking patterns to combine access of shared data into atomic regions. For example, if every access to field `f` of an object `o` in a program is protected by a lock `l`, then one could safely conclude that access to `o.f` can be perceived as *thread-local* and hence the access can be merged into the neighbouring atomic region. This optimization becomes obvious when atomicity is expressed in terms of program dependences.

6 Program Dependences

The above notions can be expressed in terms of program dependences.

- A statement sequence *s* is *atomic* if none of statements participate in interference or ready dependences.
- A statement sequence *s* is *first-read atomic* if none of non-first statements participate as dependents in interference or ready dependences.
- A statement sequence *s* is *first-write atomic* if none of non-first statements participate as dependees in interference or ready dependences.
- A statement sequence *s* is *weakly atomic* if none of non-first statement participate in interference or ready dependences.

Results from dependence analyses (as provided in Indus) can be used to construct atomic sequences according to the above definitions.

Locking Patterns As mentioned in the previous section, locking patterns can be leveraged to enlarge atomic regions. In case of program dependences, locking patterns enable us to prune away spurious interference dependences, hence, trivially contributing to the enlargment of atomic regions.

7 Sealing in Java

In dynamic analysis such as model checking, it is optimal if we could execute a large sequence of statements as a single step. To do so, none of the non-first statements in the sequence should operate (read/write) on shared data. Proving such absence via static analysis is expensive. Instead, we could take a simple hybrid approach[3].

To give a concrete example, consider model checking Java programs. A flow-insensitive static analysis can detect

1. if the statements enclosed in the dynamic scope[4] of the method do not access (read/write) global data[5] and

2. if no shared data is accessed (read/written) in any of the possible access paths leading from the parameters and receiver of a method.

If condition (1) and (2) are satisfied, then the method is marked as *atomic*. If condition (1) is satisfied and condition (2) is not satisfied, then the method is marked as *sealed* and the possible violating access paths are recorded. If condition (1) is not satisfied, then the method is marked as *open*.

During model checking, the model checker refers to the analysis to check if a method is atomic. If so, the model checker executes the method as a single transition. If not, the model checker checks if the method is sealed. If so, it executes the method as a single transition if it can verify the possible violating access paths do not access shared data. If the method is open, then the model checker executes each statement as a single transition.

In scenarios where global data is absent, condition (1) is always true and no method will be marked as open by the static analysis. Indus currently supports the detection of atomic and sealed method. However, it does not (yet) provide support to extract the possible violating access paths.

Other applications Similar to model checking, in applications such as online program optimizations such as JIT compilation. Sealing information can be used to generate minimal atomicity conditions offline and these conditions can be leveraged online to improve performance by removing unnecessary synchronization and in turn reducing runtime overheads such as scheduling costs (reduced interleavings) and memory system costs (reducing memory barriers).

Further, the same conditions can be used at runtime to check if a method aggressively annotated as atomic is indeed atomic (even before executing the method).

[3] This approach is inspired by dynamic heap-reachability based escape analysis implemented in Bogor [1]

[4] Dynamic scope of a method is the static scope of the method along with the dynamic scope of the methods invoked in the method.

[5] Data reachable from static fields in the system are referred to as *global data*.

References

[1] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.

[2] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of Symposium on Principles of programming languages (POPL)*, 2004.

[3] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003.

[4] C. Flanagan and S. Qadeer. Types For Atomicity. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in Languages Design and Implementation (TLDI)*, pages 1–12, 2003.

[5] C. von Praun and T. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 115–128, San Diego, CA, USA, Jun 2003.

[6] L. Wang and S. D. Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, February 2006.