Verifying Atomicity Specifications for Concurrent Object-Oriented Software using Model-Checking*

John Hatcliff, Robby, and Matthew B. Dwyer

Department of Computing and Information Sciences, Kansas State University **

Abstract. In recent work, Flanagan and Qadeer proposed *atomicity declarations* as a light-weight mechanism for specifying non-interference properties in concurrent programming languages such as Java, and they provided a type and effect system to verify atomicity properties. While verification of atomicity specifications via a static type system has several advantages (scalability, compositional checking), we show that verification via model-checking also has several advantages (fewer unchecked annotations, greater coverage of Java idioms, stronger verification). In particular, we show that by adapting the Bogor model-checker, we naturally address several properties that are difficult to check with a static type system.

1 Introduction

Reasoning about possible interleavings of instructions in concurrent object-oriented languages such as Java is difficult, and a variety of approaches including model checking [2, 4], static and dynamic [18] analyses and type systems [7] for race condition detection, and theorem-proving techniques based on Hoare-style logics [9] have been proposed for detecting errors due to unanticipated thread interleavings.

In recent work, Flanagan and Qadeer [10] noted that developers of concurrent programs often craft methods using various locking mechanisms with the goal of building method implementations that can be viewed as *atomic* in the sense that any interleaving of atomic method m's instructions should give the same result as executing m's instruction without any interleavings (i.e., in a single atomic step). They provide an annotation language for specifying the atomicity of methods, and a verification tool based on a type and effect system for verifying the correctness of a Java program annotated with atomicity specifications.

We believe that the atomicity system of Flanagan and Qadeer represents a very useful specification mechanism, in that it checks a light-weight and semantically simple property (e.g., as opposed to more expressive temporal specifications

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607) by Lockheed Martin, and by Intel Corporation.

** Manhattan KS, 66506, USA. {hatcliff,robby,dwyer}@cis.ksu.edu

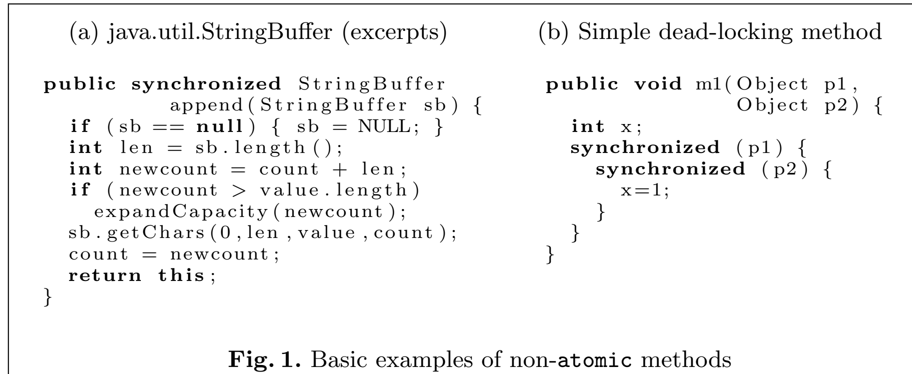

Fig. 1. Basic examples of non-`atomic` methods

that are harder to write) that can reveal a variety of programming errors associated with improper synchronization. For example, a common Java programming error is to assume that if a method is `synchronized`, then it is free from interference errors. However, Flanagan and Qadeer illustrate that this is not the case [10, p. 345] using the `java.util.StringBuffer` example of Figure 1(a). After `append` calls the method `sb.length` (which is also synchronized), a second thread could remove characters from `sb`. Thus, `length` becomes *stale* and no longer reflects the current length of `sb`, and so `getChars` is called with invalid arguments and throws a `StringIndexOutOfBoundsException`.

Inspired by the utility of the atomicity specifications, we investigated the extent to which these specifications could be checked effectively using model-checking. Clearly, checking via a type system can provide several advantages over model-checking: the type system approach is compositional – classes can be checked in isolation, a closing environment or test-harness need not be created, and computational complexity is lower which leads to better scalability.

Our efforts, which we report on in this paper, indicate that there are also several benefits in checking atomicity specifications using model checking when one employs a sophisticated software-dedicated model-checking engine such as Bogor [16] – an extensible software model checker that we have built as the core of the next generation of the Bandera tool set. Bogor provides state-of-the-art support for model-checking concurrent object-oriented languages including heap symmetry reductions, garbage collection, partial order reduction (POR) strategies based on static and dynamic (i.e., carried out during model-checking) escape analyses and locking disciplines, and sophisticated state compression algorithms that exploit object state sharing. Checking atomicity specifications with Bogor offers the following benefits.

- Due to the approximate and compositional nature of the type system, several forms of annotations are required for sufficient accuracy such as preconditions for each method stating which locks must be held upon entry to the method and declarations for each lock-protected object that indicates the lock that must be held when accessing that object. Much of this information can be derived automatically during Bogor's state-space exploration.

- The type system cannot handle some complex locking idioms since its static nature requires that objects and methods be statically associated with locks via annotations. Bogor's partial order reductions strategies were developed specifically to accommodate these more complex idioms – therefore Bogor can verify many methods that rely on more complex locking schemes for atomicity.
- The type system as presented in [10] requires several forms of unchecked annotations or assumptions about objects shared between threads, and Flanagan and Qadeer note that static escape analysis could be used to partially eliminate the need for these. In our previous work on using escape analysis to enable partial order reductions [5], we concluded that the *dynamic* escape analysis as implemented in Bogor performs dramatically better than static escape analysis for reasoning about potential interference in state-space exploration. Thus, Bogor provides a very effective solution for the unchecked annotations using previously implemented mechanisms.
- The type system [10] actually fails to enforce a key condition of Lipton's reduction framework, and thus will verify as atomic some methods whose interleaving can lead to deadlock. Bogor's atomicity verification (based on the ample set partial order reduction framework which provides reductions that preserve LTL_{-X} properties as well as deadlock conditions) correctly identifies methods that violates this condition, and thus avoids classifying these interfering methods as atomic.

Finally, Bogor's aggressive state-space optimizations enable atomicity checking without significant overhead; for most of the examples reported on in [10], for example, Bogor was able to complete the checks in under one second.

We *do not* conclude that the model-checking approach is necessarily better than the type system approach to checking atomicities. As noted earlier, a type system approach to checking has several advantages over the model-checking approach.

- The type system approach, though conservative, naturally guarantees complete coverage of all behaviors of a software unit. In contrast, model-checking a software unit requires a test harness that simulates interactions that the unit might have with a larger program context. During model-checking, the behaviors that are explored are exactly those generated by the test harness. It is usually difficult to guarantee that the test harness will drive the unit through all behaviors represented by any client context, and thus some behaviors that lead to atomicity violations may be missed if the test harness is not carefully constructed. Thus, checking atomicity specifications via model-checking is typically closer to debugging than the systematic guarantees offered by the type system approach. However, it is important to note that for complex programs unchecked type annotations are often required, consequently the type system will not cover the behavior that it assumes from the annotations.
- The type system approach scales better. In addition, by its very nature, the type system approach is compositional, which allows program units to easily

be checked in isolation. This enables incremental and modular checking of atomicity specifications.

We also do not conclude that *all* model checkers can effectively check atomicity specifications – using conventional model-checkers that do not provide direct support for heap-allocated data would be more difficult since information concerning locking or object-sharing is not directly represented.

Our conclusion is that type systems and model-checking are complementary approaches for checking the atomicity specification of Flanagan and Qadeer [10]. Moreover, in model-checkers such as Bogor and JPF that provide direct support for objects, checking for atomicity specifications should be included because such specifications are useful, and model-checking provides an effective verification mechanism for these specifications.

The rest of this paper is organized as follows. Section 2 provides a brief overview of Lipton's reduction theory, the atomicity type/effects system of Flanagan and Qadeer, and Bogor's enhancement of the ample set POR framework. Section 3 describes how Bogor's state-space exploration algorithm is augmented to check atomicity specifications. Section 4 presents experimental results drawn from the basic Java library examples used by Flanagan and Qadeer [10] as well as larger examples used in our previous work on partial order reductions [5]. Section 5 discusses related work, and Section 6 concludes.

2 Background

A *state transition system* [3] Σ is a quadruple (S, T, S_0, L) with a set of states S, a set of transitions T such that for each $\alpha \in T, \alpha \subseteq S \times S$, a set of initial states S_0, and a labeling function L that maps a state s to a set of primitive propositions that are true at s.

For the Java systems that we consider, each state s holds the stack frames for each thread (program counters and local variables for each stack frame), global variables (i.e., static class fields), and a representation of the heap. Intuitively, each $\alpha \in T$ represents a statement or step (e.g., execution of a bytecode) that can be taken by a particular thread t. In general, α is defined on multiple "input states", since the transition may be carried out, e.g., not only in a state s but also in another state s' that only differs from s in that it represents the result of another thread t' performing a transition on s.

For a transition $\alpha \in T$, we say that α is *enabled* in a state s if there is a state s' such that $\alpha(s, s')$ holds. Otherwise, α is disabled in s.

2.1 Lipton's reduction theory

Lipton introduced the theory of left/right movers to aid in proving properties about concurrent programs [14]. The motivation is that proofs can be made simpler if one is allowed to assume that a particular sequence of statements is indivisible, i.e., that the statements cannot be interleaved with statements from other threads. In order to conclude that a program P with a particular

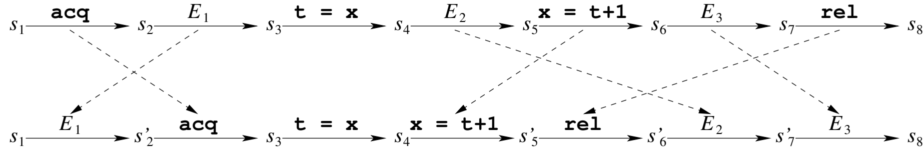

Fig. 2. Left/Right movers and atomic blocks

sequence of statements S is equivalent to a reduced program P/S where the statements S are executed in one indivisible transition, Lipton proposed the notion of *commuting transitions* in which particular program statements are identified as *left movers* or *right movers*. Intuitively, a transition α is a right (left) mover if whenever it is followed (preceded) by any other transition β of a different thread, then α and β can be swapped without changing the resulting state. Concretely, Lipton established that a lock acquire (e.g., such as those at the beginning of a Java synchronized block b) is a right mover, and the lock release at the end of b is a left mover. Any read or write to a variable/field that is properly protected by a lock is both a left and right mover.

To illustrate the application of these ideas, we repeat the example given in [10]. Consider a method m that acquires a lock, reads a variable `x` protected by that lock, updates `x`, and then releases the lock. Suppose that the transitions of this method are interleaved with transitions E_1, E_2, and E_3 of other threads. Because the actions of the method m are movers (`acq` and `rel` are left and right movers, respectively, and the lock-protected assignment to `x` is both a left and right mover). Figure 2 implies that there exists an equivalent execution where the operations of m are not interleaved with operations of other threads. Thus, it is safe to reason about the method as executing in a single atomic step.

Although the original presentation of Lipton is rather informal, he does identify the property that is preserved when programs are reduced according to his theory: the set of final states of a program P equals the set of final states of the reduced program P/S; in particular P halts iff P/S halts. To achieve this property, Lipton imposes two restrictions on a set of statements S that are grouped together to form atomic statements [14, p. 719]. Restriction (R1) states that "if S is ever entered then it should be possible to eventually exit S"; Restriction (R2) states that "the effect of statements in S when together and separated must be the same." (R1) represents a fairly strong liveness requirement that can be violated if, e.g., S contributes to a deadlock or livelock, or fails to complete because it performs a Java `wait` and is never notified. (R2) is essentially stating an independence property for S: any interleavings between the statements of S do not effect the final store it produces.

2.2 Type and effect system for checking atomicity

The type system of Flanagan and Qadeer assigns an atomicity a to each expression of a program where a is one of the following values: `const` (the same value is always returned each time the expression is evaluated), `mover` (the expression both left and right commutes with operations of other threads), `atomic` (the expression can be viewed as a single atomic action), and `compound` (none of the previous properties hold).

To classify a field access with a `mover` atomicity, the type system needs to know that the field is protected by a lock. The type system requires the following program annotations to be attached to fields and methods to express this information.

- *field* `guarded_by` l: the lock represented by the lock expression l must be held whenever the field is read or written.
- *field* `write_guarded_by` l: the lock represented by the lock expression l must be held for writes, but not necessarily for reads.
- *method* `requires` $l_1, \cdots, l_n$: a method precondition that requires locks represented by lock expressions $l_1, \cdots, l_n$ to be held upon method entry; the type system verifies that these locks are held at each call-site of the method, and uses this assumption when checking the method body.

If neither guarded statement is present, the field can be read or written at any time (such accesses are classified as `atomic` since these correspond to a single JVM bytecode).

For soundness, the type system requires that each lock expression denote a fixed lock through the execution of the program. This is guaranteed in a conservative manner by ensuring that each lock expression has atomicity `const`; such expressions include references to immutable variables, accesses to final fields of `const` expressions, and calls to const methods. In particular, note that the `this` identifier of Java which refers to the receiver of the current method is `const`, and many classes in Java are synchronized by locking the receiver object.

The type system contains rules for composing statement atomicities which we do not repeat here, but we simply note that the pattern of statement atomicities required for an atomic method takes the form given by the regular expression $\texttt{mover}^*\texttt{atomic}^?\texttt{mover}^*$ (i.e., 0 or more movers followed by 0 or 1 atomic statements followed by 0 or more movers).

When considering the effectiveness of the type system for enforcing the restrictions (R1) and (R2) laid out by Lipton, there are several interesting things to note. There is no attempt by the type system to enforce the first condition (R1). This allows the method of Figure 1(b) to be assigned an `atomic` atomicity even though interleaving of its instructions can cause a deadlock when it is used in a context where a thread t_1 calls it with objects o_1, o_2 as the parameters and a thread t_2 calls it with objects o_2, o_1 as parameters. Indeed, it is difficult to imagine any type system or static analysis enforcing the condition (R1) without being very conservative (perhaps prohibitively so) or without resorting to various forms of unchecked annotations.

In many cases, the type system enforces (R2) in a very conservative way. It does not incorporate escape information which would allow accesses to fields of an object o that are not lock-protected but where o is only reachable from a single thread to be classified as `mover` rather than `atomic`. In addition, the restriction on lock expressions in annotations that only allows references from constant fields, means that more complex locking strategies are difficult to handle effectively in the type system.

2.3 Partial-Order Reductions for OO Languages

Lipton's theory for commuting transitions differs slightly from the conditions for commuting transitions used in most POR frameworks. For example, Lipton defines a transition α as a right-mover if it right commutes with *all* other transitions β from other threads. Commutativity properties in POR frameworks are often captured by a symmetric independence relation I on transitions such that $I(\alpha, \beta)$ implies **(a)** if $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha \in \mathit{enabled}(\beta(s))$, and **(b)** if $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$. The two notions of commutativity differ in the following ways. A right mover must right-commute with *all* transitions from other threads, while the independence relation allows a transition to commute with only *some* other transitions. In addition, the commutativity definitions in the Lipton theory are asymmetric whereas, the independence relation I is required to be symmetric. This fact, along with the condition (a) on I implies that a transition α that is a lock acquire cannot be independent of another transition β that acquires the same lock because α can disable β by acquiring the lock. However, the conditions of Lipton allow an acquire to be a right mover (see [14, p. 719] for details).

In recent work [5], we described a partial order reduction framework for model-checking concurrent object-oriented languages that we have implemented in Bogor. This framework is based on the *ample set* approach of Peled [15] and detects independent transitions using locking information as inspired by the work of Stoller [19], reachability properties of the heap, and dynamic escape analysis. Independent transitions detected in our framework include accesses to fields that are always lock-protected or read-only, accesses to fields of objects that are only reachable through lock-protected objects, and accesses to *thread-local* objects, i.e., objects that are only reachable from a single thread at a given state s, etc. Because Bogor maintains an explicit representation of the heap, it is quite easy and efficient to check the heap properties necessary to detect the independence situations listed above. Independence properties for thread-local (non-escaping) objects are detected automatically by scanning the heap. Independence associated for read-only fields require a `read-only` annotation on a field, and a simple `lock-protected` annotation on a field that is protected by a lock (both of these annotations are checked for violations during model-checking). Note that our locking annotation is simpler than the annotation of Flanagan and Qadeer in that one does not need to give a static expression indicating the guarding lock. Dynamically scanning the heap during model checking allows independence to be detected in a variety of settings, e.g., when the lock protecting an object changes during program execution or that locking is not required in states where the object associated with the field is thread-local.

Figure 3 presents the depth-first search state-exploration algorithm that incorporates the ample set partial order reduction strategy [3, p. 143]. Space constraints do not allow a detailed explanation, but the intuition of the approach is a follows. At each state s with outgoing enabled transitions $\mathit{enabled}(s) = \{\alpha_1, \ldots \alpha_n, \beta_1, \ldots \beta_m\}$, $\mathit{ample}(s)$ returns a subset $\{\alpha_1, \ldots \alpha_n\}$ to expand. The selection of $\mathit{ample}(s)$ is constrained in such a way as to guarantee that the re-

```
 1  seen := {s_0}                      7      workSet := workSet \ {α}
 2  pushStack(s_0)                     8      s' := α(s)
 3  DFS(s_0)                           9      if s' ∉ seen then
                                      10         seen := seen ∪ {s'}
DFS(s)                                11         pushStack(s')
 4  workSet := ample(s)               12         DFS(s')
 5  while workSet ≠ ∅ do              13         popStack()
 6     let α ∈ workSet                end DFS
```

Fig. 3. Depth-first search with partial order reduction (DFS-POR)

duced system explored by Figure 3 satisfies the same set of LTL_{-X} formulas as the unreduced system (in which all enabled transitions from each state are expanded). Following [3, p. 158], our strategy for choosing $ample(s)$ is to find a thread t such that all of its transitions at s are independent of transitions from all other threads at s, and return that set $enabled(s,t)$ as the value of $ample(s)$ (technically, a few other conditions must be satisfied to preserve LTL_{-X} properties). If such a thread t cannot be found, $ample(s)$ is set to $enabled(s)$. We refer the reader to [5] for details.

3 Model-Checking Atomicity Specifications

We have presented two frameworks for describing commuting transitions: the Lipton mover framework, and the independence framework used in POR. We now describe how atomicity specifications can be checked by augmenting Figure 3 to classify transitions according to each of these frameworks. In effect, this yields two approaches for checking atomicity specifications using model checking: MC-mover which will classify transitions according to Lipton's framework and MC-ind which classifies transitions according to the independence framework. During our presentation, we will compare these approaches and contrast them with the Type-System checking approach of Flanagan and Qadeer.

In our atomicity checker, a method m or block of code can be annotated as `compound` (the default specification which every method trivially satisfies), `atomic`, or `mover`. Intuitively, a `mover` method is a special case of an `atomic` where all transitions are both left and right movers according to MC-mover or all independent according to MC-ind. For simplicity, we will only describe checking `atomic` methods since it is obvious that checking the patterns for `mover` will only require very minor changes.

In MC-ind, transitions can be classified uniquely as independent (I) or dependent (D) in Bogor by leveraging locking and heap structure information. In MC-mover, transitions can be classified as left-mover (L), right-mover (R), or atomic (A), and we allow a single transition to have both L and R classifications.

For an `atomic` method m, the goal of MC-ind checking is to guarantee that the transitions of m follow the pattern $\mathsf{I}^*\mathsf{D}^?\mathsf{I}^*$ every time method m is encountered during the state-space search. For each thread t, the checker uses an internal *region position* variable held in the state-vector with values (N,I,D) to mark where t's program counter (pc) is positioned with respect to the required transition pattern. N indicates t's pc is not in a block annotated as atomic, I indicates

```
...
2.1  M_ind := []
2.2  R⊗ := ∅
2.3  R×_ind := ∅
...
3.1  foreach ρ⊗ ∈ R⊗ do
3.2      signal ρ⊗ is definitely not atomic
3.3  foreach ρ× ∈ R×_ind do
3.4      signal ρ× is possibly not atomic
...
4.1  checkAtomicDeadlock(s, workSet)
...
6.1      t := thread(α)
6.2      σ_ind := getRegionPosition(t, M_ind)
...
8.1      updateIndAtomic(s, s′, α)
...
13.1     else checkAtomicCycle(s′, t)
13.2     M_ind := M_ind[t ↦ σ_ind]
...

checkAtomicCycle(s, t)
36 if isInStack(s) then
37     if region_atom?(s, t) ≠ ∅ then
38         R⊗ := R⊗ ∪ region_atom?(s, t)
end checkAtomicCycle

updateIndAtomic(s, s′, α)
39 t := thread(α)
40 σ := getRegionPosition(t, M_ind)
41 if region_atom?(s′, t) = ∅ then
42     if σ ≠ N then
43         M_ind := M_ind[t ↦ N]
44 elseif isWait(α) then
45     R⊗ := R⊗ ∪ region_atom?(s′, t)
46 elseif σ = N then
47     M_ind := M_ind[t ↦ I]
48 elseif σ = I ∧ ¬isIndependent(s, α) then
49     M_ind := M_ind[t ↦ D]
50 elseif σ = D ∧ ¬isIndependent(s, α) then
51     R×_ind := R×_ind ∪ region_atom?(s′, t)
end updateIndAtomic

checkAtomicDeadlock(s, workSet)
52 if workSet = ∅ then
53     foreach t ∈ threads(s) do
54         if region_atom?(s, t) ≠ ∅ then
55             R⊗ := R⊗ ∪ region_atom?(s, t)
56 elseif ∃θ ⊆ threads(s).
57                 circSync(θ) then
58     if ∃t ∈ θ.region_atom?(s, t) ≠ ∅ then
59         R⊗ := R⊗ ∪ region_atom?(s, t)
end checkAtomicDeadlock
```

Fig. 4. Additions to DFS-POR for MC-ind

t's pc is in an atomic block, but has not yet executed a D transition, and D indicates t's pc is in an atomic block, and has already executed a D transition. Checking using MC-mover looks for the pattern $\mathsf{R}^*\mathsf{A}^?\mathsf{L}^*$ with region position values (N,R,L) where R indicates that t's pc is in an atomic block but has not yet executed an A or L transition, and L indicates t has executed an A (i.e., t's pc is moving into or through the L region of the pattern).

In addition to checking that m conforms to the required pattern of transitions, the checker will also flag as "unverified" any `atomic` or `mover` method where a thread that cannot eventually move past the end of that method. We have two approaches for implementing this. The first uses the standard approach of Spin [12] to detect invalid end-states (completions of execution where a thread has not moved past the end of an atomic method) as well as nested depth-first search to look for non-progress cycles (indicating live-locks). Nested depth-first search makes this approach more expensive, so we implemented a cheaper scheme that looks directly for cycles in a lock-holding graph and can catch many common cases.

Figure 4 presents the MC-ind algorithm. Due to space constraints, we have elided the parts of the algorithm (using ...) that do not change from the basic DFS algorithm presented in Figure 3. Line numbers $x.y$ of Figure 4 are inserted after the line number x of Figure 3.

The input of the algorithms is a set of regions $\mathcal{R}_{atom?}$ where each region represents a method or block of code that is annotated as `atomic`. In this presentation, we represent a region $\rho \in \mathcal{R}_{atom?}$ as a non-empty finite sequence of transitions $[\alpha_1, ..., \alpha_n]$ drawn from the same thread. For simplicity, we assume that regions do not overlap with each other, and there are no loop edges across

...
2.1 $\mathcal{M}_{mov} := []$
...
2.3 $\mathcal{R}^{\times}_{mov} := \emptyset$
...
3.3 **foreach** $\rho^{\times} \in \mathcal{R}^{\times}_{mov}$ **do**
...
6.2 $\quad \sigma_{mov} := \mathit{getRegionPosition}(t, \mathcal{M}_{mov})$
...
8.1 $\quad \mathit{updateMovAtomic}(s, s', \alpha)$
...
13.2 $\quad \mathcal{M}_{mov} := \mathcal{M}_{mov}[t \mapsto \sigma_{mov}]$
...

updateMovAtomic(s, s', α)
60 $t := \mathit{thread}(\alpha)$
61 $\sigma := \mathit{getRegionPosition}(t, \mathcal{M}_{mov})$
62 **if** $\mathit{region}_{atom?}(s', t) = \emptyset$ **then**
63 $\quad$ **if** $\sigma \neq N$ **then**
64 $\quad\quad \mathcal{M}_{mov} := \mathcal{M}_{mov}[t \mapsto N]$
65 **elseif** $\mathit{isWait}(\alpha)$ **then**
66 $\quad \mathcal{R}^{\otimes} := \mathcal{R}^{\otimes} \cup \mathit{region}_{atom?}(s', t)$
67 **elseif** $\sigma = N$ **then**
68 $\quad \mathcal{M}_{mov} := \mathcal{M}_{mov}[t \mapsto R]$
69 **elseif** $\sigma = R \wedge \neg \mathit{isRightMover}(\alpha)$ **then**
70 $\quad \mathcal{M}_{mov} := \mathcal{M}_{mov}[t \mapsto L]$
71 **elseif** $\sigma = L \wedge \neg \mathit{isLeftMover}(\alpha)$ **then**
72 $\quad \mathcal{R}^{\times}_{mov} := \mathcal{R}^{\times}_{mov} \cup \mathit{region}_{atom?}(s', t)$
end *updateMovAtomic*

Fig. 5. Additions to DFS-POR for MC-mover

regions.[1] Given a thread t and a state s, we define the function $\mathit{region}_{atom?}$ to return a singleton set containing the region $\rho \in \mathcal{R}_{atom?}$ if one of t's transition at state s is in the sequence of ρ. Otherwise, it returns the empty set.

Region positions are maintained using the position table $\mathcal{M}_{ind}$ (line 2.1) that maps each thread t to its position status $\{N, I, D\}$. Given a thread t and a position table $\mathcal{M}$, we define the function *getRegionPosition* to return N if $t \notin \mathit{dom}(\mathcal{M})$; otherwise, it returns $\mathcal{M}(t)$.

Note that transition classifications are necessarily conservative since the analysis must be safe, and so a method may actually be atomic, yet fail to be classified as such because of imprecision in the Bogor independence detection algorithm. Yet there are some cases where we can tell that there is definitely an atomicity violation (e.g., if a transition executes a wait lock operation, or a deadlock or cycle within the region is found). We wish to distinguish these cases when reporting to the user, and we use the set $\mathcal{R}^{\otimes} \subseteq \mathcal{R}_{atom?}$ to hold the regions that *definitely* violate their atomicity annotations (line 2.1), so we use the set $\mathcal{R}^{\times}_{ind} \subseteq \mathcal{R}_{atom?}$ to hold the regions that *possibly* violate their atomicity annotations as determined by transition classifications (lines 2.2-2.3).

The algorithm proceeds as follows (for liveness issues, we include an explanation of our less expensive checks, since the strategy using the nested depth-first search is well-documented [12]). Once the ample set is constructed for the current state (line 4), we check if there exists a thread such that it is in one of its atomic regions and it is in a (fully/partially) deadlock state (line 4.1 and lines 52-59). If a deadlock exists, then the atomic region cannot really be atomic (given a set of threads θ, we define the function *circSync* to return *True* if there is cycle in the lock-holding/acquiring relation for threads of θ). Once a transition α is selected from the work set, then we save the region position of the thread t that is executing α (lines 6.1-6.2) so that we can restore the position marker when backtracking later on (line 13.2). After α is executed, we update the region position of t (line 8.1). If the next state s' has been seen before, we check if s' is in the stack. If it is, then there is a cycle (non-terminating loop) in the state space.

[1] Our actual implementation handles all of Java, and allows e.g., atomic methods to call other atomic methods, etc.

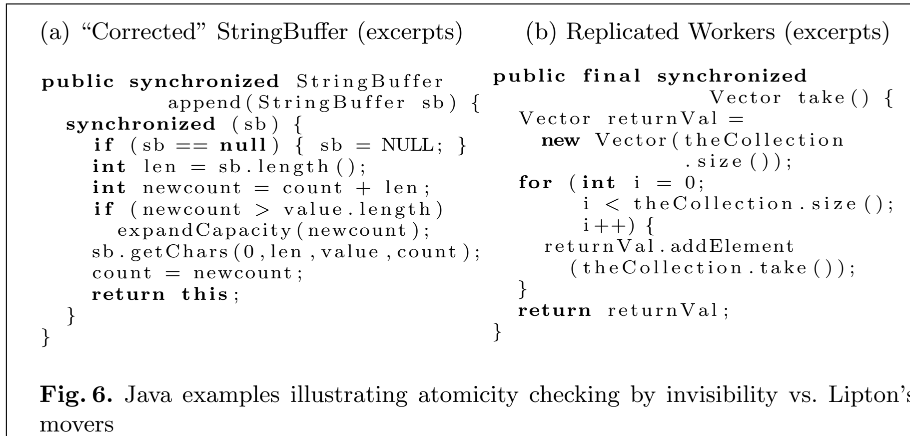

Fig. 6. Java examples illustrating atomicity checking by invisibility vs. Lipton's movers

Thus, if t is in the same atomic region at state s', then that region definitely cannot be atomic (line 13.1 and lines 36-38).

The region position is updated as follows (*updateIndAtomic*). If t is not in one of its atomic regions at state s', and if t's status at state s was other than N, then the status is updated to N (lines 41-43). If α is a wait transition, then the annotated atomic region where t is at state s' definitely cannot be atomic (line 44-45). If t's status was N, then we can infer that t has just entered one of its atomic regions. Thus, the status is updated to I (lines 46-47). If t's status was I and α is not an independent transition at state s, then t's status is updated to D (lines 48-49). Lastly, if t's status was D and α is not an independent transition, then we can infer that we have seen two non-independent transitions inside the atomic region. Thus, the required atomicity pattern has been violated (lines 50-51). The checking algorithm for MC-mover is similar and is presented in Figure 5.

Note that considering both styles of transition classification with MC-ind and MC-mover rather than some hybrid approach allows us to highlight the subtle differences between the frameworks. Also, using the approaches as they currently exist allows us to appeal to previous correctness results for both frameworks instead of proving the correctness of a combined approach. Taking MC-ind as an example, if the algorithm confirms that a method m is atomic with respect to a given test harness (i.e., environment that closes the system and provides the calls into the classes under analysis), then, for that particular test harness, we can conclude that m always completes, and previously established correctness results for our notion of independence [5] allow us to conclude that there is no interference with the statements of the methods.

3.1 Assessing MC-ind vs. MC-mover

Recall the `StringBuffer` example of Figure 1(a) which was not atomic because threads could interfere with the variable `sb` between a sequence of reads. Figure 6(a) attempts to solve this problem by locking the `sb` variable. With this modification, Type-System would verify that this method is atomic. However, in a manner similar to Figure 1(b), this method can cause a deadlock if `append` is

```
public abstract class RWVSN {
  protected Vector waitingWriterMonitors_ = new Vector();
  ...
  protected synchronized void notifyWriter() {
    if (waitingWriterMonitors_.size() > 0) {
      Object oldest = waitingWriterMonitors_.firstElement();
      waitingWriterMonitors_.removeElementAt(0);
      synchronized(oldest) { oldest.notify(); }
      ++activeWriters_;
}}}
```

Fig. 7. A readers/writers example (excerpts)

called from two different threads with the receiver object and method parameter reversed in the calls. Using a test harness with two threads that call append with arguments in opposing orders, both MC-ind and MC-mover would detect that the method is non-atomic.

Now consider Figure 6(b), which is taken from the replicated worker framework presented in [6]. In this method, there is a series of lock acquires/releases on the vectors referenced by `theCollection` and `returnVal` when the `size()` and `take()` methods are called. Thus, using the mover transition classification, the transitions have the form ..R..L..R..L which means that neither Type-System nor MC-mover can confirm that this method is atomic. However, due to the way that the data structures are set up, the vector referenced by the `theCollection` field is always dominated by the receiver object's lock (and thus protected by it). In addition, our dynamic escape analysis can detect that the `returnVal` local variable refers to an object that is thread-local at every point it is accessed during the method. Using our independence classification, the transitions (even the acquire/releases) have the form I^+, and therefore MC-ind can verify that the method is indeed atomic.

Thus, there are some cases where MC-ind can detect atomicity but MC-mover cannot (in the case above, due to the fact that the dynamic escape analysis detects that the lock acquires are not actually needed to obtain non-interference *for that method*). Similarly, there are cases when MC-mover can detect atomicity but MC-ind cannot. This typically occurs when there is a method with nested lock acquire/release which has a pattern RR..LL in MC-mover but DD..II in MC-ind. Since both methods are conservative but safe with respect to the supplied test harness, even if only one method concludes that a method is `atomic`, then we can safely conclude that the method is atomic with respect to the supplied test harness. We take advantage of this in an alternate implementation that runs both methods simultaneously. Specifically, we weave line 2.1, 2.3, 4.3, 8.1, and 13.2 of Figure 4 and Figure 5 and replace line 3.3 as follows:

3.3 **foreach** $\rho^\times \in \mathcal{R}^\times_{ind} \cap \mathcal{R}^\times_{mov}$ **do**

That is, only if both algorithms agree that a region is possibly not atomic do we report that it is *possibly* not atomic. As will be discussed in the experiment section, the combination of the algorithms is precise enough to determine the methods in Figure 6 are atomic.

The approach can be further improved by noting that methods that fail MC-mover due to patterns like ..acq l_1..rel l_1..acq l_2..rel l_2.. (i.e., R..L..R..L) often

Example			(I)	(L)	(C)	(W)
Original String-buffer	Trans: 383	**check**	9	9	9	9
Threads: 3	States 9	**noise**	0	0	0	0
Locations: 175	Time: .26	**error**	1	1	1	1
Deadlock String-buffer	Trans: 99	**check**	9	9	9	9
Threads: 3	States: 4	**noise**	0	0	0	0
Locations: 183	Time: .13	**error**	1	1	1	1
Readers Writers	Trans: 127337	**check**	23	23	23	26
Threads: 5	States: 2504	**noise**	3	3	3	0
Locations: 314	Time: 19	**error**	0	0	0	0
Replicated Workers	Trans: 230894	**check**	38	38	39	39
Threads: 4	States: 4477	**noise**	1	1	0	0
Locations: 509	Time: 63.88	**error**	0	0	0	0

Table 1. Experiment Data

are still atomic because l_1 and l_2 are from objects o_1 and o_2 that are either thread-local or already dominated by (and thus protected by) another lock. Figure 7 presents a readers/writers example[2] that illustrates this case (note the sequence of acquire/release pairs represented by the method calls on the `waitingWriterMonitors_` and the synchronization on `oldest`). However, the read-only `waitingWriterMonitors_` field points to a `Vector` object that is lock dominated (hence protected) by the receiver `RWVSN` object.

Based on this observation, we can modify the existing mover classification (in which an acquire l is never considered a left mover except when it actually represents a re-acquire of l) to take into account thread-locality and lock domination (protection) information accumulated by Bogor:

– In the right mover region of an atomic region, a left mover is now defined to be a lock release when the lock being released is non-thread-local or not protected by some locks held by the current executing thread.
– In the left mover region of an atomic region, a right mover is now defined to be a lock acquire when the lock being acquired is non-thread-local or not protected by some locks held by the current executing thread.

That is, we avoid changing the region position information when encountering lock acquires/releases that do not actually play a role in protecting an object. This change allows MC-mover to correctly identify the method of Figure 7 as atomic since the lock operations on `waitingWriterMonitors_` do not change the the region position (i.e., one has the pattern R^+ up to the point where the lock on `object` is released, and the remaining increment can be classified as L).

Note that Type-System cannot establish that this method is atomic because its rule for `synchronize` l does not recognize the fact that some other lock may already by protecting l, nor does it take into account thread-locality information.

4 Experimental Results

Figure 1 presents the results of running the examples that we have described previously on an Opteron 1.8 GHz (32-bit mode) with maximum heap of 1 Gb using the Java 2 Platform (we have also run almost all of the examples from [10] with results similar to what is shown above). In all runs, we used all

[2] `http://gee.cs.oswego.edu/dl/cpj/classes/RWVSN.java`

reductions available in Bogor described in [5] with the addition of the read-only reduction [19] The **(I)**, **(L)**, **(C)**, and **(W)** denote the MC-ind, MC-mover, the combination of MC-ind and MC-mover and and the combination that uses the modified definition of left/right movers presented at the end of the previous section. For each example, we give the number of threads and the locations or control points of the example that are executed. Furthermore, we give the number of transitions and states, and time (in seconds) needed to run each test case. The maximum memory consumption of the examples is 2.69 Mb memory, which is for the replicated workers example, and the minimum memory consumption is .8 Mb for the String-buffer example. For each example, **check** is the number of atomic methods that are verified, **noise** is the number of atomic methods that cannot be verified due to imprecision of the algorithm, and **error** is the number of specified atomic methods that definitely cannot be atomic.

For Original String-buffer, we correctly detect the atomicity violation of `append` as reported by [10], and for Deadlock String-buffer, we correctly detect the atomicity violation due to deadlock that is not detected by [10]. Note, however that this relied on constructing a test harness that happened to expose the deadlocking schedule. Note that the Readers/Writers and the Replicated Workers example contains Java synchronized collection library such as the `java.lang.Vector` class that are also indirectly verified for atomicity via calls from the larger examples. In addition, we verify the `notifyReaders`, `notifyWriter`, `afterRead`, and `afterWrite` methods of the Readers Writers example as atomic as well as various methods for synchronization in the Replicated Workers.

The timing numbers indicate the model-checking approach is feasible for unit-testing. Fewer annotations are required – we do not require the pre-condition annotations of Type-System that indicate the locks that are held when a method is called, and although we do require annotations stating that fields are lock protected, our approach infers the protecting objects instead of having them stated directly. Flanagan and Qadeer note that they run all their experiments with an unchecked annotation that allows their checker to assume that an object does not escape its constructor. Such a property is automatically checked in Bogor's dynamic escape analysis.

We have already noted that to apply our model-checking approach, the user must construct a test harness (environment) that generates appropriate coverage of the system's execution paths. It is possible that the model-checking approach can fail to detect errors because the behavior of the environment is not sufficiently rich. Also note that the model-checking approaches will likely check a method m many times during the course of verification whereas the type system only needs to make one pass. Of course the benefit is an increase in the precision due to customization of reasoning to the invocation contexts.

5 Related Work

We have already made extensive comparisons with Flanagan and Qadeer's type system [10], which inspired the work presented in this paper. Their type system

is based on the earlier type system for detecting race conditions [7]. Flanagan and Freund developed Atomizer [8], a runtime verification tool to check atomicity specifications that uses a variant of the Eraser algorithm to detect the lock-sets that protect shared variables and thread-local variables similar to [5]. However, in their algorithm, shared variables cannot become unshared later on, and they also failed to enforce Lipton's R1 condition. Wang and Stoller [21] also developed a similar runtime tool for checking atomicity. One notable feature of their tool is that given a trace, the tool permutes the ordering of events to find atomicity violations. Both of these run-time checking approaches scale better than our model-checking approach because they do not store states and they do not consider every possible interleaving. Of course, this means that they are also more likely to fail to detect atomicity violations since many feasible paths are not examined.

Our previous work on partial order reductions [5] combined escape analysis with lock-based reduction strategies formalized earlier by Stoller [19] and implemented in JPF [2]. Stoller and Cohen have recently developed a theory of reductions using abstract algebra [20], and their framework addresses many of the issues related to independence and commutativity discussed here.

Other interesting efforts on software model-checking such as [1, 11] have not considered many of the issues addressed here, since those projects have focused on automated abstraction of sequential C programs and have not included concurrency nor dynamically created objects.

Finally, model-checkers such as Spin [13] include the keyword `atomic`. However, this represents a directive to the model-checker to group transitions together without any interleaving rather than a specification to be verified against an implementation in which a developer has tried to achieve non-interference using synchronization mechanisms.

6 Conclusion

Flanagan and Qadeer have argued convincingly that atomicity specifications and associated verification mechanisms are useful in the context of concurrent programming. We believe that our work demonstrates that model-checking using state-of-the-art software model-checkers like Bogor provides an effective means of checking atomicity specifications. The key enabling factors are (1) Bogor's partial order reduction strategies based on dynamically accumulated locking and object escape information that is more difficult to obtain in static type systems, and (2) model-checking enables the verification process to easily enforce the "must complete" requirement from Lipton's theory which was not enforced in the type system of Flanagan and Qadeer (and indeed, would be difficult to enforce in any type system without resorting to very conservative approximations). An extended version of this paper and more information about examples and experimental results can be found on the Bogor web site [17].

References

1. T. Ball and S. Rajamani. Bebop: a symbolic model-checker for boolean programs. SPIN 2000, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
2. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
4. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. ICSE, 2000.
5. M. B. Dwyer, J. Hatcliff, V. Ranganath, and Robby. Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. Technical Report TR2003-1, SAnToS Laboratory, Kansas State University, 2003.
6. M. B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency : Practice and Experience*, 9(11):1293–1310, Nov. 1997.
7. C. Flanagan and S. N. Freund. Type-based race detection for Java. PLDI, 2000.
8. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of POPL*, 2003. (to appear)
9. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. ESOP, 2000.
10. C. Flanagan and S. Qadeer. A type and effect system for atomicity. PLDI, 2003.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In T. Ball and S. Rajamani, editors, *Proceedings of 10th International SPIN Workshop*, LNCS 2648, pages 235–239. Springer-Verlag, 2003.
12. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.
13. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
14. R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), 1975.
15. D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. Dill, editor, *Proceedings of the 1994 Workshop on Computer-Aided Verification (LNCS 818)*, pages 377–390. Springer, 1994.
16. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
17. Bogor Website. `http://bogor.projects.cis.ksu.edu`, 2003.
18. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
19. S. Stoller. Model-checking multi-threaded distributed Java programs. In *International Journal on Software Tools for Technology Transfer*. Springer-Verlag, 2002.
20. S. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In H. Garavel and J. Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS 2619)*, 2003.
21. L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proceedings of the Workshop on Runtime Verification*, volume 89.2 of ENTCS, 2003.