Verifying Protocols in Concurrent Software Using Atomic Blocks and Alias Control

Nels E. Beckman

December 2008

1 Introduction

In object-oriented programs, it is often the case that object types define usage protocols as an implicit part of their interface. These protocols define legal orderings of method calls, and must be respected at run-time by code that uses objects of that type. Unfortunately, these protocols are implicit and are not checked as part of the compilation process. Statically verifying the correct usage of object protocols would be beneficial, because the violation of protocols at run-time can result in undefined behavior, or in the throwing of exceptions. But it's not an easy task. Statically tracking the state of heap-based objects has proven difficult because of aliasing, specifically the ability for one object's state to be modified through multiple program references.

All of this is true for single-threaded programs. The verification of object protocol usage in concurrent programs is an even more challenging task. In that case, multiple threads may simultaneously refer to a single object whose protocol must be obeyed. But given the current state of the computer world, this is an important goal.

Currently the computer industry is in the midst of a transition, from a world in which single-CPU machines and single-threaded programs are the norm, to a world in which multi-core machines and concurrent programs are standard. If more and more developers are starting to write concurrent object-oriented programs, ensuring conformance to object protocols, already one of the most challenging parts of using a library or framework, will become a high priority.

Already, researchers in the fields of programming languages and computer engineering have taken notice of the influx of multi-core machines, and have begun exploring programming language features that will make the development of concurrent software less burdensome. This thesis describes how we can use one of these features, the atomic block, to simplify verification of object protocols in concurrent programs.

1.1 Example: A Concurrent Queue

In order to motivate the verification of object protocols in concurrent systems, let us examine a blocking queue class used as part of the Axyl-Lucene program. Axyl-Lucene[1] is an open-source program written in Java that is designed to be a server wrapper around the Apache Lucene text search program. In order to improve responsiveness, Axyl-Lucene is designed as a multi-threaded application, and includes a thread-safe blocking queue class[2]. This class is designed to be used concurrently by one producer thread, who puts items into the queue, and multiple consumer threads that consume those items. Furthermore, this queue defines a protocol, which is described in Figure 1.

[1] packages.ubuntu.com/dapper/web/axyl-lucene

[2] Thanks to Allen Holub.

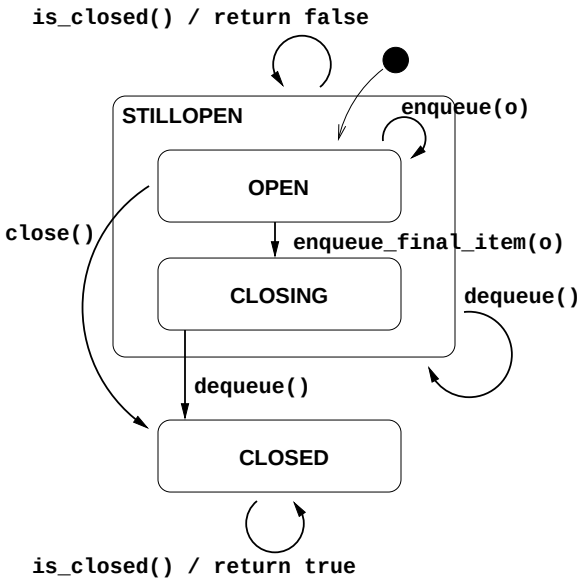

Figure 1: A simplified depiction of the protocol defined by the `Blocking_queue` class.

As the state diagram indicates, a queue starts out in an abstract state, "OPEN," during which time new elements can be added to the queue and removed by consumers. At some point in time, the producer will decide to stop putting items into the queue. The producer may decide to immediately close the queue by calling the `close` method. This puts the queue into the "CLOSED" abstract state, and ensures that items will be neither inserted nor removed from the queue. The producer may also decide that it wants to close the queue, but that the elements that are still in the queue should be retrievable by the consumers. To do this, the producer will call the `enqueue_final_item` method, which puts the queue into the "CLOSING" abstract state. During this period, and until the last item has been removed, the producer can add no more new items, but the consumers can continue to call `dequeue` to retrieve the remaining items. Finally, when `dequeue` is called with only one item remaining in the queue, the queue transitions to the "CLOSED" abstract state. If this protocol is ever disobeyed at run-time, a `Closed` exception will be thrown.

There are two things to point out about the description of the Queue's protocol. The first is the nature of the `dequeue` method, which is used in two different places in the state diagram. Normally, `dequeue` removes an element from the queue and returns the queue to the state it as in before the call. (In this sense, this diagram represents an under-specification, since all it indicates is that the queue will go back to the "STILLOPEN" state.) However, `dequeue` can also transition the queue from the "CLOSING" state to the "CLOSED" state. The second thing to notice is the `is_closed` method. If the consumer thread cannot call `dequeue` when the queue is closed, it must have some way of knowing if the queue is closed or not! Therefore, the `is_closed` method can be called at run-time to determine whether or not the queue is in the closed state. It does not alter the current state of the queue, but its return value, `true` or `false` provides information at run-time about the queue's current abstract state. We will refer to these sorts of methods as dynamic state tests.

1.2 Using the Concurrent Queue (Incorrectly)

There are two ways in which protocols of thread-shared objects can be abused that we want our static analysis to catch. First, as clients use objects that define protocols, they may inadvertently create race conditions on the abstract state of those objects. Second, in the implementation of those protocols, the objects

```
final Blocking_queue queue = new Blocking_queue();

(new Thread() {
  @Override
  public void run() {
    while( !queue.is_closed() )
      System.out.println("Got object: " + queue.dequeue());
  }}).start();

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

Figure 2: A usage of the `Blocking_queue` class which contains a race on the abstract state of the queue.

themselves may not atomically transition from one state to another, which may cause other threads to see fields of that object in an inconsistent state. In an attempt to illustrate the first point, Figure 2 shows a client of the `Blocking_queue` class using the class to share information between two threads. Unfortunately, and even though the implementation `Blocking_queue` uses correct synchronization, there is a race condition on the abstract state of the queue. In between the consumer thread's call to `is_closed` and its call to `dequeue`, it is possible for the producer to close the queue, causing the consumer's call to `dequeue` to throw a run-time exception. The author of this class alludes to this fact in comments, saying the `is_closed` method, "is inherently unreliable in a multithreaded situation" and that to achieve correct behavior the client must, "must synchronize on the queue." While the comments are helpful, because the protocol is a very real part of the object's interface it would be nice to ensure it is used correctly at compile-time.

It is also important to verify that state transitions for thread-shared objects are performed atomically. Figure 3 shows an implementation of the `close` method of the `Blocking_queue` class that does not atomically transition from the current state to the closed state. (Note that this is *not* the actual implementation used in the Axyl-Lucene project, but is used for illustrative purposes.) A design invariant of the queue is that when it is closed, the `elements` field, which holds a list, is to be null-ed out, and the `closed` field must be set to true. If the queue is thread-shared, it must transition atomically to the closed state otherwise there is the risk of a null pointer dereference. The implementation of the `is_closed` method only checks the `closed` field, and the `dequeue` method dereferences `elements` without checking whether or not it is null. Therefore, two threads racing on the queue, one to close and one to dequeue, could inadvertently cause a null pointer dereference even if the consumer atomically checked for the open state and dequeued.

1.3 Tracking Abstract States Without Alias Control

Our goal is to track the abstract states of objects as they flow through methods in a program, and we want to do this in a modular way, so that once a method has been analyzed it never needs to be analyzed again. Suppose we were to implement this analysis in a naive way. First, we will examine the `run` method of the consumer thread from Figure 2. At the outset, we would have no idea which abstract state the queue is in. Immediately after the condition expression in the while loop, our analysis would know that the queue was "STILLOPEN." Unfortunately, the `queue` reference points to an object that has escaped from its allocation context. Since we'd like our analysis to be sound, we'll have to assume that this or any other escaped object

```
boolean is_closed() {
  atomic: {
    return is_closed;
  }
}

Object dequeue() {
  atomic: {
    if( elements.size() > 0 ) {
    ...
  }
}

void close() {
  atomic: { elements = null; }
  atomic: { closed = true; }
}
```

Figure 3: An implementation of the `close` method that does not atomically transition to the closed state.

could potentially be modified by any other number of threads, as we have no information to tell our analysis otherwise. Therefore, our analysis will discard information about the abstract states of objects that have escaped. This will at least allow our naive analysis to find the race that we previously identified.

Of course, if we go back and try to verify the producer code, we find that our decision to throw out state information for every escaping object is much too imprecise. The `Blocking_queue` constructor creates the queue in the "OPEN" state, and both the `enqueue` and `close` methods require that the queue be in the "OPEN" state as a pre-condition. If our analysis discards the fact that the queue is in the "OPEN" state, after queue construction because a reference escapes to the anonymous class, then our analysis will also flag both calls as errors, since their pre-conditions can not be met. This is too bad, because our program is designed in such a way that only the producer will change the abstract state of the queue, and therefore the producer should never have to assume that another thread might change the queue's state. In order for our analysis to know this without performing a global analysis, it will need the programmer to provide information about the patterns of program sharing.

When checking that an object atomically transitions from one state to another, a similar problem arises. Without any information about whether or not the receiver of an analyzed method is intended to be shared amongst threads, we must conservatively assume that it will be. This in turn would cause nearly every analyzed method body to perform its work inside of an atomic block. With modern implementations of transactional memory, this would lead to quite a high overhead, and it something that we'd definitely like to avoid.

1.4 Current Approaches

Existing work on data race detection [10, 34, 14] does a good job of ensuring that access to thread-shared memory is protected by locks or other mutual exclusion primitives, but it does not prevent a program's threads from interleaving in ways that destroy application invariants.

Preventing thread interleavings that destroy program invariants is an important goal, because invariants allow programmers to reason about the behavior of their programs. Toward this goal, several earlier

works [26, 27, 33, 41] attempt to statically prevent or prove impossible thread interactions that might invalidate invariants. Compared to these approaches, our work allows for a larger variety of thread-sharing patterns, and additionally helps to ensure the proper use of object protocols, an abstraction of object state that forms an implicit but unchecked interface in many object-oriented programs.

1.5 Proposed Approach

This thesis proposes the following approach to modular verification of concurrent object protocol use: First, object references in method and type signatures will be annotated by the programmer using a technology called access permissions [5]. Access permissions are a static description of a.) whether or not a referenced object can be referred to by other program references and b.) whether or not those other references, if they exist, can be used to modify the state of that object. Access permissions also track the current abstract state of the object to which the reference refers. Then our analysis can track the abstract state of each reference as it flows through the body of a method, checking the required state indicated by pre-conditions at method call sites. At any point in the program where the access permission associated with a reference indicates that another modifying reference to that same object exists, the analysis will discard information about the abstract state of the object. This will be done unless the code is inside the lexical scope of an atomic block, the mutual exclusion primitive given to us by transactional memory systems.

Finally, because current implementations of atomic blocks are associated with a large overhead, we will use access permissions as a means of compile-time optimization. Atomic blocks are often implemented using transactional memory, which must synchronize and log each memory access, since it may later be found to be a conflicting one. Since access permissions provide us with static information about how memory will be read and written by different threads, we will use this information to remove unnecessary synchronization and logging.

2 This Thesis

In an ideal world, programmers would know before running a program if the object protocols they use and define are always used correctly and implemented consistently, even in the face of concurrent access. This thesis makes progress towards that goal.

2.1 Thesis Statement

Access permissions, which statically describe the aliasing behavior of program references in object-oriented programs, provide a good basis for the verification of the implementation and usage of object protocols in concurrent systems, allowing us to verify real programs and provide optimizations of the underlying runtime system.

2.2 Hypotheses

We can break the thesis statement down into more concrete, and measurable hypotheses.

2.2.1 Hypothesis: Formalization

We can develop and formalize an analysis that will guarantee a concurrent program does not violate the object protocols that it defines and prove that the system will not produce false negatives.

Validation This hypothesis will be validated by developing and formalizing a type system and operational semantics based on our permission system and proving the type system sound with respect to its semantics. The proof essentially says that no object in a program will even be required to be in some abstract state that at runtime it will not actually be in.

2.2.2 Hypothesis: Specification Coverage

Our specification system can be used to specify the behavior and implementation of object protocols in real concurrent, object-oriented programs.

Validation In order to validate this hypothesis, I will specify the behavior of object protocols in 6-8 small and 2-4 large concurrent Java programs, collected from open source projects. I will classify small programs as being from zero to 1000 lines of source, and expect the larger case studies to be from 5000 to 30,000 lines of source. During this process I will note and report recurring and interesting patterns of protocols that cannot be specified.

2.2.3 Hypothesis: Specification Burden

Our approach requires fewer and less complex specifications than comparable automated behavioral analyses.

Validation On the same suite of small and large programs, I will record the number of annotations required per line and compare that with the reported numbers from similar concurrent behavioral verification techniques, specifically Spec$^{\#}$ and JML. In order to compare their relative complexities, I will measure, as a proxy, the number of program states mentioned per annotation. Both Spec$^{\#}$ and JML boast large feature sets, including many features that have no analog in my approach. Therefore, for comparison purposes, I will use numbers reported by these approaches from concurrency-specific evaluations [26, 35].

2.2.4 Hypothesis: Analysis Precision

Our analysis will report a relatively low number of false positives, on the order of the number of false-positives reported by comparable automated behavioral analyses.

Validation In order to validate this hypothesis, I will build an automated static analysis for Java that will check the specifications on the suite of case studies. Then I will compare the number of false positives reported per line of source with that of similar concurrent behavioral verification techniques, specifically Spec$^{\#}$ and JML.

2.2.5 Hypothesis: Mutual Exclusion Requirements

In order for a program to be verified, it should not require a great deal more or much "wider" critical sections than is strictly necessary for functional correctness.

Validation While this hypothesis may prove difficult to evaluate in an objective manner, I will attempt to observe and report on the number of times that my analysis forced me to add an atomic block or increase the size of an atomic block, assuming that the original programs were synchronized correctly.

2.2.6 Hypothesis: Optimization

Because access permissions describe aliasing behaviors, permission annotations can be used to optimize transactional memory, improving its performance.

Validation In order to validate this hypothesis, I will modify a source-to-source implementation of transactional memory for Java to remove unnecessary synchronization and logging based on the access permission annotations. Again, using the same suite of verified programs, I will compare performance with and without the optimization.

3 Proposed Approach

Our approach, at a high level, is to specify and verify object protocols in concurrent programs using an alias control technology known as access permissions [5]. Access permission specifications will be provided by the programmer at method boundaries in the form of pre and post-conditions in order that our analysis can tell locally the ways in which parameters and the receiver can be aliased. These permissions, along with the current abstract state of each reference, will be tracked as they flow through method bodies, as new aliases are created, and as other methods, with their own associated pre and post-conditions, are called. At any point during the method when the permission on a reference indicates that the object to which it points could be concurrently modified, we will discard known state information about that reference. State information will *not* be discarded, however, if the code is within the lexical scope of an atomic block. State invariants, which tie the abstract state of an object to the concrete or abstract states of its fields, will also be verified in a similar manner, but with the modular invariant verification mechanism known as packing/unpacking. Finally, these same permissions, which could for example indicate that a certain object will never be read by another thread, will be used to remove unnecessary synchronization and logging in an optimized source-to-source implementation of software transactional memory.

3.1 Specification Using Access Permissions

Access permissions [5] are predicates that are associated with program references. They are useful because they soundly, statically and locally describe the aliasing behavior of the references with which they are associated. An access permission can answer three important questions about the reference with which they are associated:

1. Is the object to which this reference points referenced by any other references in the program?
2. Can this reference be used to modify the object to which it points?
3. Can other references to this object, if they exist, be used to modify this object?

There are five permissions, each of which answers these questions in a different way:

Unique permissions are the only existing permission to the object to which they refer. They can be used to read and modify.

Full permissions are associated with references that can read and modify the object to which they point, but can exist simultaneously with other, read-only permissions.

Immutable permissions are associated with references that will only be used to read an object. Other references may exist to that same object, but they cannot be used to modify either.

Pure permissions are associated with references that can only be used to read an object. Other references to that same object may exist, and they may be used to modify the object.

Share permissions are associated with references that can be used to read and modify the object to which they point. Any other number of read-only or modifying references to the same object may exist simultaneously.

Multiple different permission types can be associated with different references that at run-time will point to the same object. This feature allows verified programs to use interesting and common patterns of aliasing. For example, in a "producer/consumer" pattern, one thread might use a full permission on a reference to write to a shared object while, simultaneously, several other threads use pure permissions associated with other program references to read from that same object.

In order to statically track the state of an object that a reference points to, we will also associate current state of an object with our access permission. This gives us an access permission that looks like the following, where k is the permission, r is the reference name and s is the current state:

$$k(r)\ \texttt{in}\ s$$
$$e.g.,\ \mathsf{full}(blocking\_queue)\ \texttt{in}\ \mathsf{OPEN}$$

At the moment of object creation, a single unique, permission will be created for the new object, and will be associated with the `this` reference inside the constructor body. But from here, in order to allow the programmer to actually create aliases to that object, or in order to call a method on the object that requires a permission other than unique in its pre-condition, we must "split" the permission. Splitting is the process by which we soundly create one or many new permissions from an old permission of a different type. In order to preserve the meaning of each permission, only certain permission types can be split into other permission types. Figure 4 shows the legal splitting operations. Note that when a permission is split into one or many new permissions, the old permission is destroyed in the process.

Finally, before we can specify methods, we will need connectives to combine our permissions, and to describe the pre and post-conditions. For the underlying logic of our system we will use linear logic [18], which is a logic of resources. This will help ensure that permissions are not inadvertently, and hence unsoundly, duplicated. The decidable multiplicative, additive linear logic provides us with linear implication ($\multimap$), additive conjunction ($\otimes$), internal choice ($\oplus$) and external choice ($\&$). Now, in a series of examples, we will string these permissions together in order to form complete method specifications.

The specification of the constructor says that, given nothing at all, it will return a unique permission to the newly allocated queue that is in the OPEN state.

Blocking_queue() : $\mathbf{1} \multimap \mathsf{unique}(\mathbf{this})\ \texttt{in}\ \mathsf{OPEN}$

The `enqueue` method requires that the queue be in the open state, and the caller must have full permission to the queue. In addition, the caller must pass in a shared permission to the enqueued element. This permission is not returned, but the permission to the queue itself is, and it remains in the OPEN state.

void enqueue(Object obj) : $\mathsf{full}(\mathbf{this})\ \texttt{in}\ \mathsf{OPEN} \otimes \mathsf{share}(obj) \multimap \mathsf{full}(\mathbf{this})\ \texttt{in}\ \mathsf{OPEN}$

The `enqueue_last_item` method has probably the most interesting specification. It requires a full permission to the queue in the OPEN state and a share permission to the object, just like the `enqueue` method. However, it returns only a pure permission, and that permission is in the CLOSED state. In effect, what is happening is that the full, modifying permission is being transferred from the producer side to the consumer side. The last consumer, who is in charge of closing the queue, will need this permission in order to modify the abstract state of the queue.

$$\frac{k = \mathsf{share}|\mathsf{pure}|\mathsf{immutable}}{k(r)\ \mathtt{in}\ s \Rrightarrow k(r)\ \mathtt{in}\ s \otimes k(r)\ \mathtt{in}\ s}\ \text{S-SYM}$$

$$\frac{k = \mathsf{full}|\mathsf{share}|\mathsf{pure}|\mathsf{immutable}}{\mathsf{unique}(r)\ \mathtt{in}\ s \Rrightarrow k(r)\ \mathtt{in}\ s}\ \text{S-UNIQUE}$$

$$\frac{k = \mathsf{share}|\mathsf{pure}|\mathsf{immutable}}{\mathsf{full}(r)\ \mathtt{in}\ s \Rrightarrow k(r)\ \mathtt{in}\ s}\ \text{S-FULL}$$

$$\frac{}{\mathsf{immutable}(r)\ \mathtt{in}\ s \Rrightarrow \mathsf{pure}(r)\ \mathtt{in}\ s}\ \text{S-IMM}$$

$$\frac{k = \mathsf{full}|\mathsf{share}}{k(r)\ \mathtt{in}\ s \Rrightarrow k(r)\ \mathtt{in}\ s \otimes \mathsf{pure}(r)\ \mathtt{in}\ s}\ \text{S-ASYM}$$

$$\frac{\Gamma; \Delta \vdash P' \quad P' \Rrightarrow P}{\Gamma; \Delta \vdash P}\ \text{SUBST}$$

Figure 4: The splitting operation defines which permission types can legally be converted into other permission types. The Subst rule explains that if we need to prove a fact but cannot with the current permission, we can attempt to split the permission to get the correct one.

void enqueue_last_item(Object obj) : $\mathsf{full}(\textbf{this})\ \mathtt{in}\ \mathsf{OPEN} \otimes \mathsf{share}(obj) \multimap \mathsf{pure}(\textbf{this})\ \mathtt{in}\ \mathsf{CLOSING}$

The `dequeue` method needs only a pure permission to the queue. This is good, because we will be giving this permission to all of the consumers. The queue must be somewhere in the STILLOPEN state. In return, the caller will receive a share permission to the item dequeued. It will also receive the pure permission back, but since the call could potentially close the queue, it will be in an unknown state. While this specification captures the behavior of the dequeue method at an intuitive level, in Section 3.3 we will have to revise it somewhat in order to account for the fact that the `dequeue` method actually modifies the underlying structure of the queue.

Object dequeue() : $\mathsf{pure}(\textbf{this})\ \mathtt{in}\ \mathsf{STILLOPEN} \multimap \mathsf{share}(result) \otimes \mathsf{pure}(\textbf{this})$

The `is_closed` method is a dynamic state test, which clients can use at run-time to query the abstract state of the object. It requires pure permission to the queue. It returns two implications, and the caller can choose to eliminate one of them. If the result is true, the caller can obtain a pure permission to the queue in the CLOSED state. Otherwise, the caller can obtain a pure permission to the queue in the STILLOPEN state.

boolean is_closed() : $\mathsf{pure}(\textbf{this}) \multimap (result = \textbf{true} \multimap \mathsf{pure}(\textbf{this})\ \mathtt{in}\ \mathsf{CLOSED})\ \&$
$(result = \textbf{false} \multimap \mathsf{pure}(\textbf{this})\ \mathtt{in}\ \mathsf{STILLOPEN})$

Finally, the `close` method requires a full permission to the queue in the STILLOPEN state. This ensures that either the producer or the consumer (from within the dequeue method) can call it. It returns the full permission to the queue, now in the CLOSED state.

void close() : full(**this**) in STILLOPEN $\multimap$ full(**this**) in CLOSED

3.2 Verification Using Access Permissions

Conceptually, the verification of a program that uses object protocols consists of client-side verification, in which code that uses objects with protocols is verified, and provider-side verification, in which we verify that the implementation of the methods of an object which define a protocol is correct. In practice, this distinction may be blurred, since an object may define a protocol, but also in turn depend on its fields which themselves define protocols.

The OOPSLA 2008 paper [4] and accompanying technical report [3] form the most complete description of this work. In particular, in order to prove soundness, this system is formalized as a type system for a Java-like language with an associated operational semantics. In this section we will describe the verification process without the benefit of this formalization, due to space constraints. Please see these works for full details.

Figure 5 illustrates the verification of the producer code originally presented in Figure 2. At each program point, it shows the state of the linear context, which holds the currently known permissions for references in the program. The program differs slightly from the one originally presented in that we have pulled the inner class up to the top level. The most important thing to note about this example is that it correctly verifies. Unlike in our naive verification strategy, back in Section 1.3 because the producer thread retains full permission, an exclusive modifying permission, to the queue, we are never forced to discard its abstract state, and the pre-conditions of each method are satisfied.

```
final Blocking_queue queue = new Blocking_queue();
{ unique(queue) in OPEN }
(new ConsumerThread(queue)).start();
{ full(queue) in OPEN }
for(int i=0;i<5;i++) {
   queue.enqueue("Object " + i);
   { full(queue) in OPEN }
}
{ full(queue) in OPEN }
queue.close();
{ full(queue) in CLOSED }
```

Figure 5: Verification of the producer thread. The definition, specification and verification of the consumer thread is shown in Figure 6.

The verification of the consumer thread, is another story; it fails to correctly verify, which is a good thing. The code for the ConsumerThread class, as well as its verification, appear in Figure 6. To verify this class, we must use a few features of the provider-side verification mechanism. These features will be discussed in more detail in the next section. However, note that the ConsumerThread class has an invariant for the state alive, the only state that it ever inhabits. This invariant says that the consumer thread will always have a pure permission to the queue, a reference to which is stores in the `queue` field. This permission initially comes from the constructor, whose specification dictates that it consumes a pure permission to the given field which it does not return.

In the `run` method, initially there is a unique permission to the receiver in the context, as given by the method pre-condition. In order to call the `is_closed` method, the receiver must be unpacked. This

```
final class ConsumerThread {
  invariant alive: pure(queue)
  Blocking_queue queue;

  ConsumerThread(Blocking_queue q) : pure(q) ⊸ unique(this) {
    this.queue = q;
  }
  public void start() : unique(this) ⊸ 1 {
    super.start();
  }

  public void run() : unique(this) ⊸ unique(this) {
    { unique(this) }
    while(!queue.is_closed()) {
      { unpacked(this,alive), ⊗ pure(queue) }
      System.out.println("Got object: " + queue.dequeue()); // Error! Not in STILLOPEN.
    }
  }
}
```

Figure 6: The specification and verification of the ConsumerThread class.

feature of provider-side verification will be discussed in the next section, but essentially it is a sound means of providing a method with the facts implied by a state invariant. Here this means we get the pure permission to the queue. This pure permission is used to call the `is_closed` method, and while we can eliminate the returned implication inside the true branch of the loop, the analysis immediately discards the implied state of the queue. This is done because the method's permission to the queue, pure, implies that another thread could be concurrently modifying the queue. Our analysis will always discard this information unless we are inside an atomic block. Because this information was discarded, the `dequeue` method's precondition cannot be satisfied, and an error is signaled.

Finally, we present a corrected version of the `run` method in Figure 7. In this implementation, we use an atomic block to eliminate the race on the abstract state of the queue object. This time, in the else branch of the conditional, we are able to retain the fact that the queue is in the STILLOPEN state. While it is true that other threads may have modifying permission to the queue, that thread could not concurrently modify the queue because we are inside of an atomic block, and the semantics of atomic blocks prevent this.

3.2.1 Provider-Side Verification

We touched on provider-side verification a bit in the previous section, but in this section we will discuss it in more detail, and in particular we will explain how our analysis prevents non-atomic state transitions for thread-shared objects. The goal of provider-side verification is to ensure that a given method actually performs the state transition its specification claims. In order to do this, our analysis allows the abstract states that a class defines to be associated with an arbitrary predicate over the fields of the receiver object. For example, in Figure 6, we used the invariant pure$(queue)$ for the alive state, which tells us that whenever the thread is in the alive state, (i.e., *always*), it must have a pure permission to the `queue` field. (Note that state invariants are private to the implementing class. To clients, abstract states are indeed abstract.)

```
public void run() : unique(this) ⊸ unique(this) {
  { unique(this) }
  while(true) { Object item;
    atomic: {
      if(queue.is_closed())
        { unpacked(this,alive), ⊗ pure(queue) in CLOSED }
        return;
      else
        { unpacked(this,alive), ⊗ pure(queue) in STILLOPEN }
        System.out.println("Got object: " + queue.dequeue());
    }
  }
}
```

Figure 7: Verification of a corrected version of the `run` method from Figure 6 that uses an atomic block.

In order to verify that a state transition is correctly implemented, we must be able to establish the truth of the invariant of the post-condition state before the method returns. Fortunately, we can use the invariant for the incoming state in order to help us prove it. In Section 1.2 we mentioned how, in our implementation of the `Blocking_queue`, an invariant existed; when the queue is closed, the `closed` field should have the value `true`, and the `elements` field should have the value `null`. We can specify this requirement with the following state invariant:

$$\textbf{invariant}\ \mathsf{CLOSED}: \ elements = \textbf{null} \otimes closed = \textbf{true}$$

In order to verify the implementation of the `closed` method, we must ensure that this invariant is true before the method returns. In order to do this, we use a methodology known as *packing* [2, 13]. The packing methodology allows for the modular verification of class invariants. This methodology requires us to unpack an object at any point when its invariants may not hold. While unpacked, we cannot return from the method, and we cannot use the permission to the unpacked object. However, we can use the state invariants over the fields of the object associated with the abstract state the object was in when it was unpacked. Finally, and in order to avoid the unsound duplication of permissions, an object cannot be unpacked if it is already unpacked.

In order to ensure that thread-shared objects cannot be observed in an inconsistent state, our analysis adds an additional restriction; the unpacking of any reference associated with full, pure or share permission must be performed entirely within the scope of an atomic block. The original motivation behind this restriction was the observation that the section of code where an object is unpacked corresponds with the section of code where an object's invariants might temporarily not hold. When an object is in an inconsistent state, and that object is thread-shared, other threads should not be allowed to observe its state, and the atomic block ensures just that. Given this intuition, one may wonder why this restriction is necessary for references of pure permission, since it cannot be used to modify the object. A pure permission can still unpack invariants from an abstract state, and it expects those facts to hold. Therefore, unpacking within an atomic block ensures that the facts that are unpacked will not be modified concurrently.

In Figure 8 we demonstrate provider-side verification on the example from Figure 3. Before the fields of the queue can be reassigned, the receiver must be unpacked. This is tracked with the **unpacked** predicate. This also introduces any predicates associated with the OPEN state into the context. Then `elements` is reassigned. At this point, the atomic block closes. Assuming that the OPEN state requires the `elements`

```
void close() : full(this) in OPEN ⊸ full(this) in CLOSED {
  { full(this) in OPEN }
  atomic: {
    {unpacked(this,OPEN), ... OPEN state invariants ... }
    elements = null;
    {unpacked(this,OPEN), elements == null, ... }
  } // Error! Atomic block closed while receiver unpacked!
  atomic: {
    closed = true;
  }
}
```

Figure 8: Verification of the `close` method fails because the atomic block ends before the receiver can be packed, and it is associated with full permission.

field to be non-null, this code will not verify; there are not enough facts in the context to pack the receiver to any state, and it is associated with a full permission, which requires it to be packed outside of an atomic block. Therefore an error is signaled. The important point is that a potentially thread-shared object was about to become visible to other threads in a concrete state that did not correspond to any abstract state. If the programmer instead had extended the atomic block across both assignments, this method would verify.

3.3 State Dimensions and State Hierarchies

Sometimes it is necessary for objects to define more complex protocols than those I have shown here, and for those we may need to make use of state dimensions and state hierarchies. A full description of either is outside of the scope of this document, as neither is a contribution of this thesis. They are described in full detail elsewhere [6]. However, adding both state dimensions and state hierarchies to this work greatly enriches the system, and therefore it is important to describe in some detail.

State Dimensions allows one object to define several orthogonal protocols. For example, an object that models a wristwatch may have two protocols: one for an alarm, which can be on or off, and another for a light, which independently can be on or off. On the implementation side, state dimensions are conceptually similar to data groups [30]. Each orthogonal protocol defines its state invariants over a subset of the object's fields, where each set is disjoint from one another. From the client's perspective, it is possible to have permission to just one dimension of an object if it defines multiple protocols. Similarly, a client could hold references of different permission types to different dimensions of an object.

In fact, and as hinted in Section 3.1, we need state dimensions to verify the original implementation of the `Blocking_queue` class. While only one thread (the producer) will modify the fields of the queue associated with the protocol, both producer and consumer threads modify the underlying linked list, producers to insert items and consumers to remove them. Therefore, to verify the full blocking queue example, we map the `elements` field, which points to the underlying linked list, into a separate dimension, "STRUCTURE" which defines no interesting protocol.[3] Both producer threads and consumer threads have a share permission to this dimension. As a result, each method signature changes slightly. The full queue specification is shown in the appendix. Here, for example, is the proper specification[4] of the `dequeue` method:

[3]Note that this makes the CLOSED state invariant we defined in Section 3.2.1 very hard to verify, since the `elements` field and `closed` field are mapped to two different dimensions. Fortunately, this invariant does not exist in the actual implementation, and was created just for presentation purposes.

[4]The *fr* notation indicates that this is a permission to the object frame, giving us permission to read fields of the object, as

Object dequeue() : share(**this**!fr, STRUCTURE) $\otimes$ pure(**this**!fr, PROTOCOL) in OPEN $\multimap$
share(**this**!fr, STRUCTURE) $\otimes$ pure(**this**!fr, PROTOCOL)

State hierarchies allow abstract states to be refined into multiple sub-states. This allows subclasses to define more specific abstract states while still preserving behavioral subtyping on overridden methods. In the example in Figure 1, the "OPEN" and "CLOSING" states refine the "STILLOPEN" state. In fact, in the implementation of our concurrent typestate analysis, every object has one super-state, "alive" that it always occupies, and which all other states implicitly refine. When we discard knowledge about the state of an object, for instance, because of the possibility of concurrent modification, we really are just dropping the state to the "alive" state. Finally, a state S_g can be statically *guaranteed*, which tells a weak permission such as pure that, while the modifying permission can transition the object at will to any sub-state, it can never leave the S_g state.

Programmers familiar with UML [8] will recognize both state dimensions and state hierarchies, both of which are features of the state diagram notion.

3.4 Permission-Based Optimization

Access permissions can also be used to optimize the performance of software transactional memory implementations, since they statically describe the aliasing behaviors of a program. Many software transactional memory systems are implemented in an optimistic fashion, where transactions execute and roll back only if it is later determined they observed an inconsistent memory state. In order to implement this behavior, STM systems keep read and write sets which track which pieces of memory were read from and written to by which transactions, and what the original value of an object was at the beginning of a transaction.

The fastest current approaches [1] "open" an object for writing (or reading) before fields of that object are written to (or read from). A thread opens an object for writing by setting its transaction as the owner of the object (using an atomic test and set), and copying the initial value of the object to a write log. Before the transaction commits, a "version" number associated with the written object will be incremented. A thread opens an object for reading by checking to see if that object is owned (using an atomic test) and if not, adding the current version number for the read object to the transaction read set. In either case, if the object is already owned by another thread, the opening thread must defer to contention management to solve the dispute.

Since access permissions tell us statically which objects will be modified and which will be shared, we can remove synchronization and logging operations that will never be needed. The general principles of our optimization are as follows:

- Objects of immutable permission will never be opened for reading, since no thread will change their value.
- When writing to the fields of a unique object, it is not necessary to "open" that object for writing (a process that requires synchronization) since no other thread can concurrently access the object. However, it is necessary to log the initial value of the object as the transaction still may be rolled back.
- Neither objects of unique nor full permission ever need to be opened for reading.
- We would like the above three rules to always be sound. However, because unique and full permissions can be reached through fields of other thread-shared objects, we require that any share, full, or pure object be opened for writing before any method is called on a unique or full field of that object.

opposed to a virtual permission, which gives us the right to call virtual methods on an object.

Because of the last restriction, it is possible that our optimization will decrease performance in some circumstances, since it must open objects for writing that might not otherwise be opened. We expect that our optimization will improve performance by reducing overhead in applications where objects of unique and immutable permission are frequently modified inside of transactions, but where threads most commonly access disjoint regions of memory.

We can use state dimensions to decrease the granularity of locking below that of the object level. At the moment, most STM systems perform conflict detection on either the object or the word level. Conflict detection on the word level can often impose too much overhead, while the object level has the potential to create increased contention. State dimensions divide an object into groups of fields that will be accessed as part of the same logical operations, and therefore make a good candidate for a granularity of conflict detection that is finer than the object level, yet coarser than the word level.

4 Related Work

In this section, we discuss related work. While there are a large number of competing approaches, we feel that our work is significantly different, and provides a substantial contribution. The most important contribution over existing work is that of the full/pure permission, which allows one object to depend on knowledge about another, thread-shared but unencapsulated object. As an example, consider the `Thread_pool` class, shown in Figure 9, that also comes from the Axyl-Lucene project. This thread pool also defines an open-closed protocol. Interestingly, the invariants of the open and closed abstract state depend on the abstract state of the underlying blocking queue (e.g., when the pool is in the open state, the queue must be in the open state). Our approach allows this type of invariant, even though the thread pool is shared with several instances of the `Pooled_thread` class. As we will show, this sort of invariant cannot be expressed in other approaches.

```
public class Thread_pool {
  states alive = { open, closed }

  private final Blocking_queue queue;
  private boolean has_closed;

  invariant open: has_closed = false ⊗
        share(queue,STRUCTURE) ⊗ full(queue,PROTOCOL) in open;
  invariant closed: has_closed = true ⊗
        share(queue,STRUCTURE) ⊗ full(queue,PROTOCOL) in closed;

  private class Pooled_thread {
    ...
  }

  ...
}
```

Figure 9: The Thread_pool class, also from the Axyl-Lucene project, defines a state invariant that depends on the abstract state of a thread-shared blocking queue.

4.1 Automated Verification of Concurrent Programs

The work that most closely resembles our own was developed as part of the Spec$^{\#}$ Project. Jacobs et al. [26] have also created a system that will preserve object invariants even in the face of concurrency. Moreover, our system uses a very similar unpacking methodology which comes from a shared research heritage [2]. Nonetheless, we believe our work to be different in several important ways. First, they use ownership as their underlying means of alias-control, which imposes some hierarchical restrictions on the architecture of an application. Our approach allows for ownership transfer between threads (e.g., the `enqueue_final_object` method in the blocking queue. See Appendix A.). On the other hand, their system allows more expressive specifications, as behaviors can be specified in first-order predicate logic, rather than typestate. This system does have a proof of soundness but provides neither formal typing rules nor a formal semantics.

As mentioned, their approach cannot mention thread-shared objects in invariants. Once an object becomes thread-shared, a process which must be signified by the "share" annotation, it can no longer be mentioned in another object's invariant. Therefore, examples like the one shown in Figure 9 cannot be verified.

Finally, our system uses atomic blocks while the Jacobs approach is based on locks. While this may seem like a minor detail, it actually provides our system with nice benefits. In their approach, in order to determine whether it is the responsibility of the client or provider to ensure proper synchronization, there is a notion of *client-side locking* versus *provider-side locking*. Methods using client-side locking can provide more information-laden post-conditions, while provider-side locking methods cannot. Because atomic blocks are a composable primitive, it is sufficient in our system to create one method with a full post-condition. This method can then be type-checked correctly in atomic and non-atomic contexts.

Some related work has also been done within the context of the JML project [35]. This work is mainly focused on introducing new specifications useful for those who would like to verify lock-based, concurrent object-oriented programs. Some of the specifications can be automatically verified, however due to the fact that this verification is done with a model-checker, verification failed to terminate on about half of their examples.

Joshi and Sen [28] attempt to solve the same problem, using the same motivation. Their solution is a dynamic analysis which has significantly lower burden on the programmer, but can miss some races and can infer inaccurate typestate properties.

In recent work, Vaziri et al. [41] have proposed a system to help programmers preserve the consistency of objects with a feature called *atomic sets*. In this approach, programmers specify that certain fields of an object are related, and must be modified atomically. An interprocedural static analysis then infers code locations where synchronization is required. While a promising approach, it does not allow verification of functional properties of code, such as the correct usage of object protocols.

Finally, Harris and Jones [21] introduce a mechanism for STM Haskell that ensures a data invariants will not be violated during a given execution of a program. However, this is a dynamic technique that cannot guarantee conformance for all executions.

4.2 Logics for Concurrency

There are a number of popular logics for concurrency, which can be used to prove important properties of concurrent programs. These logics include the logic of Owicki and Gries [33], Concurrent Separation Logic [32], and Rely-Guarantee Logic [27]. All three allow programmers to specify invariants over thread-shared, mutable data in simple imperative languages. Owicki-Gries and Concurrent Separation Logic are similar, differing in the expressive power of the logics they each use. In these systems, one associates both a lock and an invariant with a piece of thread-shared data. Upon entering a critical section, the invariants

over thread-shared data are revealed. These invariants can be used to prove other propositions, but must be reestablished before the end of the critical section.

Concurrent separation logic has become extremely popular recently. Concurrent separation logic essentially has our unique and immutable permissions, as well as a "critical-protected" permission that we do not. This permission cannot be aliased and can only be accessed inside of a critical region. (In fact, it can only be referenced by multiple threads because a thread spawn is a nested syntactic construct in the imperative language they use.) As has our work, separation logic has even been extended [9] with fractional permissions [11], which allow weak permissions to be tracked and recombined to form stronger ones. However, separation logic is usually used to reason about low level properties of programs, rather than abstractions of state, and additionally does not have the flexibility of the full/pure permission pair.

In the Rely-Guarantee approach, a thread must specify invariants which describe how it will not interfere with particular conditions required by other threads. Simultaneously a thread must specify the non-interference conditions that it requires of other threads. When a program is correct, the rely and guarantee specifications of each thread weave together to form a global proof of correctness. Rely-Guarantee is great for reasoning about lock-free concurrent algorithms. However, the approach suffers because system specifications must be written in a global manner. A thread states not only its pre and post conditions, but also which invariants of other threads it promises to not invalidate. These invariants could have nothing to do with the memory that it modifies. Work on combining the two approaches [15, 40] still contains many of their original limitations. All three logics are pen and paper-based techniques and are not, as described in these works, automated analyses.

Calvin-R [17] is an automation of the Rely-Guarantee concept, where the rely and guarantee predicate for every thread is a conjunction of *access predicates*, describing which locks must be held when accessing shared variables. Calvin-R uses this information, along with the Lipton [31] theory of reduction, to prove method behavioral specifications. Calvin-R must assume that every method could be called concurrently, and therefore variables must always be accessed in accordance with their access predicate. Whereas in our system, a unique permission to the receiver of a method call says that the object cannot be thread-shared for the duration of that call, and therefore fields do not require protected access. Also, this work does not mention the effect that aliasing might have on the validity of access predicates, but presumably something must be done to ensure soundness.

4.3 Race and Atomicity Checkers

There has been much work in the automated prevention of data races.

Dynamic race detectors [37, 42] check for unordered reads and writes to the same location in memory at execution time by instrumenting program code. Model-checking approaches have also been explored [23, 39]. These work by abstractly exploring possible thread interleavings in order to find ones in which there is no ordering on a read and write to the same memory location. There have also been a number of static analyses and type systems for data race prevention [10, 19, 20, 34, 14] as well, each making trade-offs in the number of false-positives and the complexity of annotations required.

The fundamental difference between each of these race detection approaches and our approach is the presence or absence of behavioral specifications. None of the other approaches require behavioral specifications, and therefore can check only an implicit specification; that the program should contain no data races. In our system, typestate specifications, which describe the intended program behavior, allows us to prevent more semantically meaningful race conditions.

Atomicity checkers [16, 36, 24] help programmers achieve atomicity using locks, but can only ensure the atomicity that the programmer deems necessary. Given a specification of a piece of code that must execute as if atomic and specifications relating locks to the memory that they protect, an atomicity checker will tell the programmer whether or not locks are used correctly, according to the theory of reduction [31].

Once again, because atomicity checkers do not require behavioral specifications, they do not tell the program which sections of code must execute atomically in order to ensure program correctness.

4.4 Optimization of STM

Since it has been noted that overhead is a major barrier to adoption for transactional memory systems, many researchers have worked to eliminate unnecessary overhead in transactional systems. Our approach is primarily different in the number of annotations we require; almost every other approach is either dynamic, or a completely automated approach requiring no programmer annotations. We make the argument that, because we giving programmers both a partial guarantee of functional correctness and static optimization, our annotations are somewhat more palatable. Moreover, because our approach uses programmer provided annotations, we can do with an intra-procedural analysis optimizations that in past have been done with a whole program analysis.

Several basic optimizations work by identifying immutable objects [1]. For example, final fields of objects cannot be modified, and certain well-known types, such as strings, can never be modified. For these objects, no logging is ever necessary. Other approaches work by dynamically (or statically) identifying objects that are allocated inside of a transaction [1] ([22]) or are never shared with other threads [38].

The work of Shpeisman and others [38] also proposes a, "Not Accessed Inside Transaction" analysis, that removes reading and/or writing barriers for references that can never be accessed inside of a transaction. This is done using a whole-program analysis which depends on a whole-program alias analysis. (Our analysis essentially requires programmers to annotate this alias information.) Interestingly, the authors complain about the fact that the thread object itself must be treated as a thread-shared object, since it can additionally be accessed by the thread that spawned it. They claim that they see lower performance because what are in fact thread-local objects are often stored as fields of the thread object, and their analysis must assume them to be thread-shared. In our optimization, programmers can annotate the thread `start` method as consuming the permission to the thread object itself, allowing us to optimize accesses to the newly spawned thread object.

Finally, the workhorse of static optimization of concurrency is the escape analysis [7, 12], which identifies objects which cannot outlive their allocation context. This in turn implies that the objects cannot be shared with other threads. While an escape analysis can cheaply eliminate unnecessary synchronization, it often fails to identify objects that are not thread-shared, but still manage to escape their allocation context.

5 Research Plan

5.1 Time-line

Below is an estimated time-line for the completion of my thesis work. This time-line begins in Fall 2008. The total estimated time is 17 months.

Some work towards this thesis has already been completed. I formalized this analysis as a type system for a core, Java-like language with threads and atomic blocks. This language was proven sound with respect to an operational semantics that models multi-threading and shared memory. This work is described in detail elsewhere [4, 3]. I implemented this analysis as NIMBY[5], a intra-procedural, dataflow analysis for the Java language. We have also implemented AtomicPower, which is the realization of optimizations described in Section 3.4. This work was performed in cooperation with master's student Yoon Phil Kim [29]. We took AtomJava [25], a source-to-source STM implementation and modified it to use an optimistic read/pessimistic write approach, which in practice performs quite well [1]. This modified version was then further

[5]For, "Not in My Back Yard!"

Estimated Time Required	Description
3 months	Specification and verification of six to eight small, but real/realistic concurrent Java programs, for the purpose of identifying patterns and deficiencies in my analysis. These programs will also be used as benchmarks for the evaluation of the optimization.
5 months	Feedback phase, during which I will improve the theory and implementation based on the programs the case studies from the proceeding phase. The goal of this phase is to use the knowledge gained from these case studies in order to make the final phase of case studies a successful one.
5 months	Specification and verification of two to four larger, real concurrent Java programs. During this phase, the expressiveness, coverage and rate of false positives for my approach will be evaluated.
4 months	Writing and defense.

extended with our optimizations. Finally, I have already begun specifying, verifying, and benchmarking the six to eight small applications using the NIMBY checker.

The feedback phase described in the time-line is meant to help improve my analysis based on my experience verifying actual programs. There is no doubt that my analysis will produce false positives. We'd like to reduce them. In order to do so, I will see what program patterns frequently occur that my analysis is not able to handle, and find a way to add those features. For example, some early experience verifying concurrent programs shows that many programs use a, "one reference per thread" pattern. In this pattern, each thread will have no more than one reference to a thread-shared object. It can then treat this reference as unique when inside of an atomic block, which due to our effect system allows for higher precision across method calls. The result of this feedback phase will be the theory and the implementation necessary to verify two to four larger, real concurrent Java programs.

5.2 Risks

In this section I list the major risks to the successful completion of my thesis work and the steps I intend to take to mitigate each of those risks.

5.2.1 Risk: Lack of Protocol-Based, Concurrent Programs

One risk is that there will not be any, or at least not many concurrent, Object-Oriented programs that use protocols. This is a risk because it would indicate that the problem I am trying to solve is not a particularly important one. It would also make it difficult to choose enough real programs for the intended series of case studies.

Mitigation I plan to mitigate this risk by immediately beginning to identify potential examples. This process involves searching open-source code bases for concurrent programs that also use some kind of object protocol, and then investigating whether or not they would likely be susceptible to my approach. Initially this seemed like a larger risk, but as of this writing I have already identified and investigated 12 concurrent, open-source Java programs that I believe can be verified by my approach. These programs vary in size from 171 to 30,000 lines of source, with most being less than 2000 lines. I am also aware of a large number of other potential candidates that I have not yet been able to investigate.

5.2.2 Risk: Programs Cannot be Proven

Another risk is that the programs that I do find simply cannot be verified by my approach. In other words, the automated analysis would produce too many false-positives to be useful.

Mitigation I am mitigating this risk by explicitly including a feedback phase into my research plan. After the initial collection, specification, and verification of six to eight concurrent programs, I will use the knowledge that I have gained to expand the theory of the analysis and improve the implementation. Then, the results of this feedback phase would be used for another series of case studies, this time on two to four larger programs.

5.2.3 Risk: Poor Optimization Performance

As with any optimization, there is a risk that a good theoretical idea simply fails to improve performance in practice, for any number of practical reasons. While our optimizations seem like they will improve performance, our early results have been good but not outstanding.

Mitigation While the optimizations represent a relatively small contribution of this thesis, I would still like the optimizations to be successful. To this end, I will attempt to identify the behaviors of programs that will be improved by our optimizations, and start early in collecting examples of this sort.

6 Conclusion

In conclusion, I propose for a thesis a verification system that requires programmers to annotate program references with an alias control mechanism known as access permissions. Given these annotations, we can automatically verify the correct use of object protocols in concurrent, object-oriented programs. This is possible because access permissions tell us which program references will be used to modify references, and whether or not they are aliased. This in turn helps us determine whether or not an object might be thread-shared. All this is done with the help of the atomic block, a mutual exclusion primitive provided by transactional memory systems whose semantics is easy to formalize. Finally, as an added benefit for the programmer's annotation effort, we will use access permission annotations in order to reduce the overhead of an implementation of software transactional memory by removing unnecessary synchronization and logging.

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *The 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37. ACM Press, 2006.

[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology Special Issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs*, 3(6):27–56, June 2004.

[3] Nels E. Beckman and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate: Technical companion. Technical Report CMU-ISR-08-126, Carnegie Mellon University, 2008. `http://reports-archive.adm.cs.cmu.edu/anon/isr2008/CMU-ISR-08-126.pdf`.

[4] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *The 2008 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 2008.

[5] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *The 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 301–320. ACM Press, 2007.

[6] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 217–226. ACM Press, 2005.

[7] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *The 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999.

[8] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[9] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

[10] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230. ACM Press, 2002.

[11] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[12] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *The 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999.

[13] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP '04: European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

[14] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252. ACM Press, 2003.

[15] Xinyu Feng. Local rely-guarantee reasoning. In *ACM Conference on Principles of Programming Languages*. ACM Press, 2009.

[16] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003.

[17] Stephen Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[18] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

[19] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 453–463. ACM Press, 2002.

[20] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25. ACM Press, 2003.

[21] Tim Harris and Simon Peyton Jones. Transactional memory with data invariants. In *TRANSACT '06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[22] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[23] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13. ACM Press, 2004.

[24] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *TRANSACT '06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[25] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *The 2006 workshop on Memory system performance and correctness*, pages 82–91. ACM Press, 2006.

[26] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[28] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded java programs. *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 288–296, Sept. 2008.

[29] Yoon Phil Kim. Permission-based optimization for effecient software transactional memory. Master's thesis, Carnegie Mellon University, 2008.

[30] K. Rustan M. Leino. Data groups: specifying the modification of extended state. In *The 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 144–153. ACM Press, 1998.

[31] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[32] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[33] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[34] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331. ACM Press, 2006.

[35] Edwin Rodriguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP '05: Object-Oriented Programming 19th European Conference*, pages 551–576, 2005.

[36] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94. ACM Press, 2005.

[37] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[38] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. *SIGPLAN Notices*, 42(6):78–88, 2007.

[39] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 224–244, London, UK, 2000. Springer-Verlag.

[40] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *In 18th CONCUR*. Springer, 2007.

[41] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 334–345. ACM, 2006.

[42] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234. ACM Press, 2005.

Appendices

A Full Specification of the Blocking_queue

In this appendix we provide the full specification of the `Blocking_queue` class. Note that the STILLOPEN state does not appear, as its presence was unnecessary for correct verification of the queue.

final class Blocking_queue {
  states STRUCTURE = { structurestate }
  states PROTOCOL = { open, closed }

```
  private LinkedList elements; in STRUCTURE;
  private boolean closed; in PROTOCOL;
  private boolean reject_enqueue_requests; in PROTOCOL;
  private int waiting_threads; in STRUCTURE;

  invariant STRUCTURE: share(elements) ⊗ reject_enqueue_requests ⊸ full(this,PROTOCOL)
  invariant closed: closed = true;
  invariant open: closed = false;

  public Blocking_queue() : 1 ⊸ unique(this!fr) in open, structurestate

  public void enqueue(Object o) :
    share(o) ⊗ full(this, PROTOCOL) in open ⊗ share(this, STRUCTURE) ⊸
    full(this, PROTOCOL) in open ⊗ share(this, STRUCTURE)

  public void enqueue_final_item(Object o) :
    share(o) ⊗ full(this, PROTOCOL) in open ⊗ share(this, STRUCTURE) ⊸ 1

  public Object dequeue() :
    pure(this!fr, PROTOCOL) in open ⊗ share(this!fr, STRUCTURE) ⊸
    pure(this!fr, PROTOCOL) ⊗ share(this!fr, STRUCTURE) ⊗ share(result)

  public boolean is_empty() : pure(this, STRUCTURE) ⊸ pure(this, STRUCTURE)

  boolean is_closed() : pure(this) ⊸
    (result = true ⊸ pure(this!fr, PROTOCOL) in closed) &
    (result = false ⊸ pure(this!fr, PROTOCOL) in open)

  public void close() : full(this, PROTOCOL) in open ⊸ full(this, PROTOCOL) in closed
}
```