Verifying Correct Usage of Atomic Blocks with Typestate

Nels E. Beckman Kevin Bierhoff Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
{nbeckman,kbierhof,jonathan.aldrich}@cs.cmu.edu

Abstract

The atomic block, a synchronization primitive provided to programmers in transactional memory systems, has the potential to greatly ease the development of concurrent software. However, atomic blocks can still be used incorrectly, and race conditions can still occur at the level of application logic. In this paper, we present a static analysis, formalized as a programming language and proven sound, that helps programmers use atomic blocks correctly. Using access permissions that describe how objects are aliased and modified, our system statically prevents race conditions up to the behavior specified using typestate annotations. We have implemented a prototype static analysis for the Java language based on our system and have used it to verify several realistic examples.

Categories and Subject Descriptors D.3.2 [*PROGRAMMING LANGUAGES*]: Concurrent, distributed, and parallel languages; F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Transactional memory, Typestate, Permissions

1. Introduction

It is now taken for granted in the field of computer science that the age of parallelism is upon us, and with good reason; with more and more of the transistors given to us by Moore's law going into an ever-increasing number of on-chip cores, we can no longer expect predictable increases in single-threaded performance. With this in mind, many researchers from the fields of computer science have begun investigating new techniques for the development of software which can actually take advantage of more cores.

Among the large number of recent proposals, transactional memory seems to have gained the greatest amount of traction. Transactional memory attempts to simplify the construction of concurrent applications that make use of shared memory. The phrase "Software Transactional Memory" (STM) describes an implementation of this concept in software [23]. STM is a subtle and complex topic, but at its core it provides programmers with a simple concurrency primitive, the atomic block. Code that is executed within an atomic block will execute sequentially, and as if no other threads were executing at the same time. The system is "transactional," because atomic blocks are usually implemented as transactions which provide atomicity, consistency and isolation to concurrent memory accesses.

However, as some of STM's greatest proponents will tell you, while atomic sections are a vast improvement over lock-based synchronization, they are far from perfect [14]. Synchronization needs beyond critical sections, such as those traditionally provided by *wait* and *notify*, are not handled cleanly by atomic sections, and as of yet it is unclear how I/O within transactions should be handled. Most importantly for the scope of this work, atomic sections by themselves do not provide correct synchronization, even when critical sections are the only form of concurrency needed, because they can still be used incorrectly.

As motivation, consider a hypothetical network chat application, used as a running example throughout this paper and partially shown in Figures 1 and 2. In this application, two threads, a GUI event thread and a network-monitoring thread, each modify one shared object of the Connection class. This class abstractly represents a connection between two hosts remote hosts. The GUI thread sends messages, and opens and closes the connection in response to user events, while the network-monitoring thread closes the connection in response to a remote closing.

The first example, Figure 1, shows that even if every access to thread-shared memory is performed inside of an atomic block, race-conditions at the level of program logic can still occur. In this example the `trySendMsg` method, fired in response to a GUI event, checks to see if the connection is active, and if so sends a message by calling the `send` method of the Connection class. Both the `isConnected` and

```
class Connection {
  ...
  void disconnect() { /* see fig. 2 */ }

  boolean isConnected() {
    atomic: {
      return (this.socket != null);
    }
  }

  void send(String msg) {
    atomic: {
      this.socket.write(msg);
      this.counter.increment();
    }
  }
  ...
}

class GUI {
  ...
  boolean trySendMsg(String msg) {
    if( this.myConnection.isConnected() ) {
      this.myConnection.send(msg);
      return true;
    }
    else {
      return false;
    }
  }
  ...
}
```

Figure 1. An example where a race condition could occur.

send methods of the Connection class are properly synchronized, reading and modifying thread-shared fields inside of atomic blocks. A race condition exists because the GUI thread relies on the connection remaining in the open state in between the call to isConnected and the call to send. If the network thread were to close the connection at this time, a null-pointer exception would occur once the GUI thread began executing the send method. Note that this type of thread interference is sometimes referred to as "external" interference [28], as it occurs external to the code that is executed by multiple threads.

While static race detectors and race-free type systems have been developed [6, 27] to ensure shared-memory access occurs inside regions of mutual exclusion, they are still vulnerable to the same problem. We will refer to this type of race condition as an "application-level" one, since the logic of the application expects a certain condition that could be violated by another thread.

```
class Connection {
  ...
  final Counter counter;

  Connection {
    this.socket = null;
    this.counter = new Counter();
  }

  void disconnect() {
    atomic: {
      this.socket.close();
      this.socket = null;
    }
    this.counter.reset();
  }
  ...
}
```

Figure 2. An example where object invariants might be violated.

The second example, shown in Figure 2, shows how improper use of atomic blocks can lead to violations of object invariants. The Connection class also privately keeps a counter to track the number of messages that have been sent during the lifetime of a connection. At the time of class creation this counter is initialized to zero, and each time a connection is disconnected, this counter is reset using the reset method. In fact, the Connection class has an invariant that its methods rely on: whenever a Connection object is not connected, the socket field will be null and the message counter will be reset. This helps to ensure that our message count will be accurate. We will assume that the reset method is implemented in such a way that all access to its member variables are done within atomic blocks.

Once again, even though access to shared variables is performed inside atomic blocks, race conditions can have adverse effects on the behavior of our program, this time invalidating our object invariant. Assume that the disconnect method is being executed by the network thread. If the GUI thread were call the connect method and begin sending messages using the send method precisely at a point in time where the network thread had exited the atomic block but had not yet reset the counter, we would lose count of each of those sent packets when the network thread eventually resets the counter. This type of interference from threads is sometimes referred to as "internal" interference [28], as it occurs internal to the thread-shared object.

This example is necessarily simplistic, and therefore it may seem strange that the developer of the disconnect method did not merely enclose the entire method body in an atomic block. In a more realistic example, there could be a large amount of other, thread-local computation occurring inside the body of this method. As with locks, developers often attempt to shrink the size of atomic regions in order to minimize performance overhead. The question then becomes, what is the minimum amount of code that must be executed inside of an atomic block in order to avoid race conditions and preserve the object's invariants. Answering this question correctly can be tricky.

In this paper, we describe a Java-like programming language whose type system statically prevents errors of these kinds. Up to the invariants that are specified by the programmer, this type system prevents race conditions and guarantees that invariants are reestablished at the end of method bodies, even in the face of concurrent access to an object and its fields. Our system uses typestate [8] specifications as the language of invariants, and object permissions [7] to approximate whether or not an object can be thread-shared. Our work builds upon recent work for verifying typestate of aliased objects [4].

The contributions of this paper are as follows:

- We have developed a programming language that begins to address the problem of improper atomic block usage. The type system of this language guarantees that no application-level race conditions can occur, and that specified invariants are preserved in concurrent programs.
- In this paper, we reinterpret access permissions, which we previously used as an alias-control mechanism, as an approximation of the thread-sharedness of a location in memory.
- We have proved soundness for a core subset of this language in the accompanying technical report [3].
- To our knowledge this is the first work that attempts to statically verify the proper placement of atomic blocks in object-oriented code.
- We have developed a prototype analysis for the Java language based on this type system and have used it to verify several realistic examples.

This paper proceeds as follows. In Section 2 we describe our technique at an informal level, using our chat program as a running example. By the end of this section, readers should understand the intuition behind our approach. Section 3 describes the formal language in greater detail. Section 4 describes our prototype implementation, as well as its use in verifying several real or realistic Java programs. In Section 5 we discuss the wealth of existing work in verification of concurrent software. In Section 6 we discuss how we would like to improve our technique, and in Section 7 we conclude.

2. Overview

At a high level, our approach is as follows:

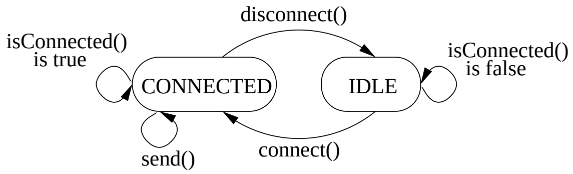
- We use typestate specifications on methods and classes to say which abstract state an object must be in before calling a method on it, and which states an object's fields must be in at the end of a method call. (In principle, other behavioral specifications would work as well.)
- Object references are annotated with access permissions which describe how an object pointed to by a reference is shared. Permissions were originally proposed as a means for guaranteeing the non-interference of threads. In previous work, we used interfering permissions to control aliasing. Now we reinterpret the same interfering permissions to describe how threads share objects.
- Finally, we track the state of objects as they flow through code, forgetting (or creating "havoc" on) anything that we know about the state of an object when we cannot determine statically that we are within an atomic section *and* the reference to that object indicates it may be modified by other threads.

In the next several sub-sections, we describe each part of the process in greater detail.

2.1 Typestate Specifications

Our approach uses typestate [8] as the language of behavioral specification. Specification is how the system knows which application-specific logic must be upheld in the face of concurrent access.

Typestate specifications allow programmers to develop abstract protocols describing a method or class' behavior. The abstractions take the form of state-machines, an abstraction with which most programmers are familiar. As an example, the developer of a file class might specify that a file can be in either the open or closed states, and that data can only be read from that file when it is in the open state. For a more relevant example, consider the following specification of the Connection class:



This indicates that a connection can abstractly be idle or connected. Calling the `connect` method will take an object from the idle state to the connected state, while the reverse holds for the `disconnect` method. The sending of messages can only occur while the object is in the connected state, but sending a message does not affect the object's state. Finally, we can dynamically test whether or not we are in the connected state by calling `isConnected`.

Existing work has been done in statically verifying that an object's behavior will conform to its typestate specification at run-time [8]. Our work, in particular, adapts the approach of Bierhoff and Aldrich for use in concurrent settings [4]. In the approach proposed by Bierhoff and Aldrich, object states are tracked statically using linear logic predicates [12] which treat object state information as a resource that can be consumed and transformed. Methods that transform the state of an object will consume its old state, and return a new state, and the type of the reference to that object will reflect its new state in subsequent lines of code.

Usually state names are defined by an application, however, this paper mentions two special states, ? and alive, which are known ahead of time. ? represents a lack of knowledge about the state of an object. alive, on the other hand, is the default state given to an object whose class defines no abstract states.

One of the main limitations of so-called "data race" analyses and type systems [6, 27] is that their definition of race condition is relatively restricted; the unsychronized reading and writing of a shared location in memory. It is precisely because these race detectors do not know what a given application intends to accomplish that they cannot detect "application-level" race conditions. Behavioral specifications, of which typestate is just one form, give analysis tools exactly this information.

2.2 Access Permissions

Access permissions [4] are a means of associating object references with (a) the state of the object referenced and (b) the ways in which that object can be aliased. This is important because statically tracking the state of an object in the face of unrestricted aliasing is undecidable. In this section we will show how access permissions can approximate information on whether or not an object is thread-shared, and why this is a sound approximation.

The access permissions system that we use has five different permission types, each one describing whether or not the object is aliased, whether the given reference is allowed to modify the object, and whether other references to the object, if they exist, are allowed to modify the object. These permissions are named as follows:

- unique permission to an object indicates that this reference is the sole reference to an object in the program. This is the same as a linear reference in other type-systems [32].
- full permissions are exclusive read/write references that can coexist with any number of read-only references.
- immutable permissions are associated with references that point to immutable objects. Any number of these references can point to the same object, but no reference may have modifying access.
- pure permissions are read-only permissions to objects that may be modified through other references.
- share permissions are associated with references that can read and write objects that can also be read and modified by any number of other references in the system. This is the least restrictive permission, and is effectively the default in languages like Java.

The access permissions are arranged in a partial order and can be *split* in order to create other permissions to the same object. This is necessary because when an object constructor is called, a single unique reference is returned, but we may want to then create multiple references to distribute to different parts of the program. These splitting rules are described in Figure 3. In the formal language, it is the responsibility of the linear logic proof judgment to automatically determine when and how permissions should be split into other permissions. If several expressions in a method require different permissions to the same reference, the implementation of this judgment must solve these constraints by splitting the permission in an appropriate way. In our implementation (§ 4), this is done with a constraint solver.

An example access permission is shown below:

$$\mathsf{unique}(\texttt{counter}, \mathsf{RESET})$$

This permission tells us that the counter field points to an object that can only be reached via this field, and therefore this reference has exclusive read/write access. Furthermore, it is known at this point that the counter is in the "RESET" abstract state.

$$\frac{k = \mathsf{share}|\mathsf{pure}|\mathsf{immutable}}{k(r,s) \Rrightarrow k(r,s) \otimes k(r,s)}\ \textsc{S-Sym}$$

$$\frac{k = \mathsf{full}|\mathsf{share}|\mathsf{pure}|\mathsf{immutable}}{\mathsf{unique}(r,s) \Rrightarrow k(r,s)}\ \textsc{S-Unique}$$

$$\frac{k = \mathsf{share}|\mathsf{pure}|\mathsf{immutable}}{\mathsf{full}(r,s) \Rrightarrow k(r,s)}\ \textsc{S-Full}$$

$$\frac{}{\mathsf{immutable}(r,s) \Rrightarrow \mathsf{pure}(r,s)}\ \textsc{S-Imm}$$

$$\frac{k = \mathsf{full}|\mathsf{share}}{k(r,s) \Rrightarrow k(r,s) \otimes \mathsf{pure}(r,s)}\ \textsc{S-ASym}$$

$$\frac{\Gamma; \Delta \vdash P' \quad P' \Rrightarrow P}{\Gamma; \Delta \vdash P}\ \textsc{Subst}$$

Figure 3. Permission splitting rules

2.2.1 Method Specifications

Now that we have seen access permissions, we can string them together with linear logic connectives to create specifications. The $\multimap$ connective is used to specify method pre and post-conditions. Predicates on left-hand side form the method pre-condition, and those on the right-hand side form the post-condition. Predicates in the pre-condition are consumed and cannot be reused unless explicitly returned by the post-condition. Linear conjunction ($\otimes$) is used when we wish to say that multiple objects must be in specific states at the same time, and linear disjunction ($\oplus$) is used when one of several state predicates may be true. We have annotated the methods of the Connection class with behavioral annotations in Figure 4. For example, the `isConnected` method is described in the following manner: If the method is called when the receiver is a shared object in an unknown state, after the method completes the receiver object will either be in the CONNECTED state, signified by a return value of true, or the receiver will be in the IDLE state, signified by a return value of false. Other methods are annotated similarly.

```
class Connection {
  boolean isConnected() :
    share(this, ?) ⊸
        (result == true ⊗ share(this, CONNECTED)) ⊕
        (result == false ⊗ share(this, IDLE))
  {
    atomic: {
      return (this.socket != null);
    }
  }

  void connect(String addr) :
    immutable(addr, alive) ⊗ share(this, IDLE) ⊸
        share(this, CONNECTED)
  {
    atomic: {
      this.socket = new Socket(addr);
      this.counter.startCounting();
    }
  }

  void send(String msg) :
    immutable(msg, alive) ⊗ share(this, CONNECTED) ⊸
        share(this, CONNECTED)
  {
    atomic: {
      this.socket.write(msg);
      this.counter.increment();
    }
  }

  void disconnect() :
    share(this, CONNECTED) ⊸ share(this, IDLE)
  {
    atomic: {
      this.socket.close();
      this.socket = null;
    }
    this.counter.reset();
  }

  // ... continued
}
```

Figure 4. Method specifications and implementations for the Connection class. The class definition is continued in Figure 5.

```
class Connection {
  // ... from above

  states IDLE, CONNECTED;

  IDLE := unique(counter, RESET) ⊗
          socket == null
  CONNECTED := unique(counter, COUNTING) ⊗
          unique(socket, alive)

  private final Counter counter;
  private Socket socket;

  Connection() :
    1 ⊸ unique(this, IDLE)
  {
    this.socket = null;
    this.counter = new Counter();
  }
}
```

Figure 5. State, invariant and constructor specifications for the Connection class.

2.2.2 State Invariants and Packing

The same access permissions can be used to annotate classes with invariant predicates. In our system, an object's invariants are tied to the state the object is in. When designing a class, a programmer has the ability to declare abstract states for a class, and he can also decide that certain predicates describing the condition of an object's fields must hold true whenever the object is in one of those states. These predicates are called state invariants. In Figure 5, we have annotated the `Connection` class with state invariants, predicates that should hold true when that connection is either open or closed. For example, when a connection is in CONNECTED state, its counter field must be in the COUNTING state, and its socket must be in the alive state.

In order for methods to modify the fields of an object and still modularly verify that these invariants hold, we employ a packing/unpacking methodology [8].

Unpacking is a means of statically delineating the portions of code during which object invariants are not expected to hold. Normally, objects are "packed," meaning that their state invariants hold. However, in order to read or modify the fields of an object, that object must first be unpacked. At the point of unpacking, we are allowed to assume the information about the fields of the unpacked object that is implied by the state invariant of the object that is being unpacked. For example, in the `send` method in of the Connection class seen in Figure 4, we cannot assume that field `counter` is in the COUNTING state until the receiver object (`this`) is unpacked. While the receiver is unpacked, it cannot be treated as being in the CONNECTED state, since the invariants may not hold. Our formal system tracks this information using a separate access permission, unpacked(share, CONNECTED), which tells us what state the receiver was in before unpacking.

When it comes time to pack an object, either to the same state or to a different state, it is at the point of packing that we must be able to prove the state invariant implied by that state. Because the state invariants of an object should hold at all times when a method is not currently being called, our system requires that an object is packed before the method returns. Additionally, packing must occur before a method call, so that the receiver will be consistent in case of reentrant calls. While in general any object can be packed or unpacked, our formal system allows only the receiver object to be unpacked. This means that access to fields of objects is restricted to methods of that object's class. Finally, note that our formal system requires packing and unpacking to be made explicit in the source code. This is necessary for the purposes of the type-safety proof. Our checker (§ 4) does not have this requirement and instead infers when and to/from what state an object must be packed/unpacked.

2.2.3 Access Permissions as Thread-Sharing

In order to determine when the state of an object could potentially be changed by another thread, we need to know which objects are shared across threads. In our system, we use access permissions as an approximation of this information. If a reference is annotated with a permission that indicates the referred object can be reached via other references, we assume that those references are held by other threads, and all consequences that this might imply.

This is a sound, if potentially imprecise, approximation because in order for a new thread to be spawned, a new thread object must be created, with the relevant object references passed to that thread's constructor. Alternatively, as in our formalization (see Section 3), if threads can be spawned by calling a method on an object, objects that must be used by both spawning thread and the spawnee must be passed to this method. In our system, the only means by which an object can be passed to a method or constructor and still be held by the caller is by splitting that permission to one of the potentially-shared permissions. We now reexamine our access permissions in the context of thread sharing:

- unique permissions are permissions to objects that only one thread has access to at a given time. These objects can be passed from one thread to another in a linear manner.
- full permissions are permissions to objects that only one thread can modify, but many threads can read. The thread with full permission can rely on the fact that no other threads can change the state of the object.
- immutable permissions are permissions to objects that will only ever be read. All threads can rely on this object never changing state.
- pure permissions are reading permissions to objects that another thread could potentially modify. Unless inside an atomic block, a thread with a pure permission must assume that the object's state could change at any moment.
- share permissions are modifying permissions to objects that could potentially be modified by a number of other threads. Again, unless inside an atomic block, we must assume that the object's state could change at any moment.

Given access permissions in this light, our analysis will work by forgetting state information whenever we are not inside a transaction and an object might be modified by another thread. For local variables our analysis can retain state information when the variable's permission is not pure or share. Fields of an object, however, are a slightly different story.

Unpacking an object may give us access to the fields of that object, and those fields often may have permissions that we have said cannot be modified by other threads. But if the object that is being unpacked has pure or share permissions, then multiple threads could read these "safe" objects by traversing through the thread-shared reference. Therefore, in order to reestablish the condition that all unique and full fields of an object could not be modified concurrently by another thread, we require that the unpacking of a pure, share, or full object be done within a transaction. Now, regardless of whether a variable is a field or local variable, our analysis only needs to forget state information if the permission on the variable is pure or share.

The soundness of this technique boils down to this intuition. If a method has access to a unique (or full) permission, one of the following two cases must be true:

- The object referred to is a new object that exists only on the stack of the current thread, and therefore could not be accessed by any other threads.
- The object is referred to by the field of another object. Since thread-shared objects cannot be unpacked outside of an atomic block, if the referring object is thread-shared we must already be inside of one. This situation is shown pictorially in Figure 6.

Finally, we require that all static member variables are read or written to inside of atomic blocks. Our formal system (§ 3) has no notion of static member variables and therefore does not enforce this requirement. Our implementation, on the other hand, does.

In summary, the following additions are required to make access permissions function as a sound approximation of thread-sharing:

- We immediately forget state information about references whose access permission indicates that the referred object could be modified by other threads (pure and share).

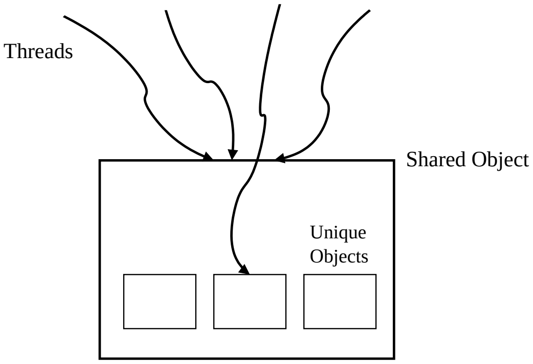


Figure 6. Unique and full fields within a thread-shared object have necessarily been unpacked within a transaction. The single thread inside is free to modify at will.

- We require that share, pure, and full references are only unpacked inside of atomic blocks. This ensures that we have exclusive access to the fields of that object. This is required for full permissions only because our system uses weak transactional semantics, and is done for the benefit of the other, pure references to the same object.
- All static fields must be read from and written to inside of atomic blocks.

One of the nice aspects of this methodology is that there is no additional annotation burden over and above the permission annotations. If you are already using them to track typestate in a single-threaded application, no additional annotations, with the exception of atomic blocks, are necessary if you decide to make that application concurrent.

2.3 Tracking Transactions

In order to track whether or not a given line of code must be executing within an atomic block, we use a simple type and effect system recently formalized [25]. Atomic blocks are dynamically scoped. At run-time, a statement within a method body could very well be executing within a transaction, even if the method itself never explicitly opened an atomic block. This is because any methods called within an atomic block will execute within the same transaction. This also means that if we use a modular analysis, it may be impossible to tell if a method body is inside of an atomic block.

This intuition corresponds to three effect values in our system: Expressions type-checked with the wt effect are known to definitely be executing within a transaction. Statements inside of an atomic block are type-checked in this manner. Expressions type-checked with the ot effect are known to be executing outside of a transaction. Because of the dynamic nature of an atomic block only the single, top-level expression is type-checked with this effect. You might also imagine type-checking the main method of a Java program in this way. Finally, the emp effect indicates that the type-system cannot be sure one way or the other. Method bodies are type-checked with this effect since they could potentially be called within an atomic block. The tracking of transactions is treated more formally in Section 3.

2.4 Examples Revisited

Now that we have seen typestate specifications, access permissions and we can statically track whether or not code is executing inside of a transaction, we can revisit our original examples and see where these examples would fail to check. In Figure 7 we have taken the original trySendMsg method from Figure 1 and annotated it with the typestate and permission information that is known statically at each line of the method, as well as the “in-transaction” effect that the line is currently being checked under. The transaction effect is always emp in this example, since no atomic blocks are ever entered. It may also be helpful to refer to Figure 4 which shows the method specifications.

```
boolean trySendMsg(String msg) {
emp : unique(this, alive)
emp : unpacked(unique, alive), share(myConnection, ?)
  if( this.myConnection.isConnected() )
emp : unpacked(...),
    (result==true ⊗ share(myConnection, CONNECTED)) ⊕
    (result==false ⊗ share(myConnection, IDLE)))
emp : unpacked(...),
    (result==true ⊗ share(this, ?)) ⊕
    (result==false ⊗ share(myConnection, ?)))
  {
emp : unpacked(...), share(myConnection, ?)
Error! Precondition not met.
    this.myConnection.send(msg);
emp : unique(this, alive)
    return true;
  }
  else {
emp : unpacked(...), share(myConnection, ?)
emp : unique(this, alive)
    return false;
  }
}
```

Figure 7. Verification of the trySendMsg method of the GUI class from Figure 1

At the onset of the method, we have a unique permission to the receiver, and this receiver is in the alive state, as no states were defined for the GUI class. In order to access fields of the receiver, the receiver is immediately unpacked, introducing an unpacked predicate which prevents double-unpacking. Unpacking also gives us a share permission to the myConnection field, which is in some unknown state.

```
void disconnect() {
  atomic: {
wt : share(this, CONNECTED)
wt : unpacked(share, CONNECTED),
    (unique(socket, alive))
    unique(counter, COUNTING))
    this.socket.close();
    this.socket = null;
wt : unpacked(...), (socket==null),
    unique(counter, COUNTING)
    this.counter.reset();
wt : unpacked(...), (socket==null),
    unique(counter, RESET)
wt : share(this, IDLE)
  }
}
```

Figure 8. Verification of the *corrected* disconnect method.

This is enough to satisfy the pre-condition for the dynamic state test isConnected, which consumes the original permission to the field, and returns a predicate indicating that if the return value is true we will know that the connection is open, and the reverse if the return value is false. It is at this point that the analysis "plays havoc," or forgets all known state information about pure and share permissions. When the analysis arrives at the true branch of the conditional, it knows that the result of the method call must have been true, and therefore can reduce the predicate describing myConnection. Unfortunately, because of the havoc that was played on our knowledge, we now cannot fulfill the precondition of the send method, and and error is signaled. Before each method return the receiver is packed to the post-condition.

The object invariant example from Figure 2 proceeds in a similar manner. In Figure 8 we successfully verify a version of the disconnect method that we have corrected by pulling the call to reset into the atomic block. Initially we begin with the method pre-condition, which we unpack. Unpacking gives us the knowledge that we have a unique permission to both the socket and the counter fields of the receiver, and that the counter is in the counting state.

One may wonder why we are not forced to forget that the receiver is in the connected state in between the pre-condition and the entry into the atomic block. Actually, we know here that this method can only be called from inside a atomic block. The only way to ever satisfy its precondition is to establish the fact inside of a transaction (or in turn to get that information from the calling method's pre-condition). For this reason, forgetting actually only occurs inside of method bodies. This point is discussed in more detail when the P-METH rule is discussed in Section 3.

From this point forward, we are checking inside the wt effect, and therefore are not required to forget the state of share or pure permissions. The socket field is assigned null, and this fact is recorded in our resource context. Then the reset method is called on the counter field. While we have not given the full specification for this method, the specification can be paraphrased as, "given a unique pointer to a counting counter, the method will return a unique pointer to a reset counter." Finally we have enough facts to pack the receiver to the post-condition. A similar reasoning, again discussed in Section 3, applies to why we do not have to forget share and pure permissions in between the last statement and establishment of the post-condition.

Both Figure 7 and Figure 8 elide certain details. In reality, and in order to ensure that recursive method calls see objects in consistent states, we are required to pack before method calls. This can be done for both examples, but was not shown for presentation reasons. Also, some permissions were shortened or ignored (e.g., the immutable permission to the msg parameter in trySendMsg) for space reasons.

In the introduction we say that race conditions are prevented up to the program behavior that is specified, and now hopefully it is clear why. Only those method behaviors and class invariants that can be expressed in terms of typestate, and that are actually annotated by the programmer will be guaranteed in the face of concurrency.

3. Language

We have formalized our proposed analysis as a core, Java-like language. We chose a language-based approach so that our proof could model threads and their non-determinism at run-time. In this section we will present this formal language. The syntax of this language is given in Figure 9.

Our formal language builds heavily upon two existing systems in the literature. We will point out the major differences. Our system of access permissions reuses many of the pieces developed by Bierhoff and Aldrich [4], but leave out some of the more advanced features, like state dimensions and sub-typing in order to focus on concurrency. Our implementation does inherit these features.

Much of the formalism regarding transactional memory, threads and their operational semantics was adapted from Moore and Grossman [25]. In particular we use their Weak language, a language that provides weak atomicity and does not explicitly model transaction roll-back, as a starting point.

Expressions are type-checked using the following judgment: $\Gamma; \Delta; \mathcal{E} \vdash^{C} e : \exists x : T.P$. This judgment says, "given a list of variable types that can be used many times, Γ, and a list of consumable predicates that can be used only once, Δ, and an effect describing whether or not we are known stat-

$$\begin{array}{rrcl}
\textit{program} & PG & ::= & \langle \overline{CL}, e\rangle \\
\textit{class decls.} & CL & ::= & \texttt{class}\ C\ \{\ \overline{F}\ I\ \overline{N}\ \overline{M}\} \\
\textit{field decls.} & F & ::= & f : T \\
\textit{methods} & M & ::= & T\ m(\overline{T\ x}) : \mathit{MS} = e \\
\textit{terms} & t & ::= & x, y, z \mid o \\
 & & \mid & \texttt{true} \mid \texttt{false} \mid t_1\ \texttt{or}\ t_2 \\
 & & \mid & t_1\ \texttt{and}\ t_2 \mid \texttt{not}\ t \\
\textit{expressions} & e & ::= & t \mid t.f \mid f := t \\
 & & \mid & \texttt{new}\ C(\bar{t}) \mid t_o.m(\bar{t}) \\
 & & \mid & \texttt{if}(t, e_1, e_2) \\
 & & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
 & & \mid & \texttt{spawn}\ (t_o.m(\bar{t})) \mid \texttt{atomic}\ e \\
 & & \mid & \texttt{unpack}(k, s)\ \texttt{in}\ e \mid \texttt{pack to}(s)\ \texttt{in}\ e \\
\textit{values} & v & ::= & o \mid \texttt{true} \mid \texttt{false} \\
\textit{references} & r & ::= & x \mid o \mid o.f \\
\textit{types} & T & ::= & C \mid \texttt{bool} \\
\textit{atomic} & \mathcal{E} & ::= & \mathsf{wt} \mid \mathsf{ot} \mid \mathsf{emp} \\
\textit{valid contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : T \mid \Gamma, q \\
\textit{linear contexts} & \Delta & ::= & \cdot \mid \Delta, P
\end{array}$$

$$\begin{array}{rlrlrl}
\textit{classes} & C & \textit{fields} & f & \textit{variables} & x \\
\textit{objects} & o & \textit{methods} & m & \textit{states} & s
\end{array}$$

Figure 9. Language Syntax. Permission syntax (P, MS, N, and I) defined in Figure 10.

$$\begin{array}{rrcl}
\textit{permissions} & p & ::= & k(r, \$) \mid \mathsf{unpacked}(k, \$) \\
 & \$ & ::= & s \mid ? \\
\textit{facts} & q & ::= & t = \texttt{true} \mid t = \texttt{false} \\
\textit{predicates} & P & ::= & p \mid q \mid P_1 \otimes P_2 \mid P_1 \oplus P_2 \\
 & & \mid & 1 \mid 0 \mid \top \\
\textit{method specs} & \mathit{MS} & ::= & P \multimap E \\
\textit{expr types} & E & ::= & \exists x : T.P \\
\textit{state inv.} & N & ::= & s = P \\
\textit{initial state} & I & ::= & \texttt{initially}\ \langle s\rangle \\
 & k & ::= & \mathsf{full} \mid \mathsf{pure} \mid \mathsf{share} \\
 & & \mid & \mathsf{immutable} \mid \mathsf{unique}
\end{array}$$

Figure 10. Permission syntax

ically to be within a transaction, $\mathcal{E}$, the expression e being executed within receiver class C has type T and produces a new permission P. This permission may contain existentially bound variables. Note that for clarity of presentation the receiver class annotation is left off unless it is needed in a typing rule.

The existential type of an expression is somewhat unusual and therefore deserves further mention. The reason a permission can contain existentially bound variables is because there are times when our system tracks the permissions of an object to which no reference points. For instance after the first expression in a let binding is evaluated, its result (if of a class type) represents an object even before it is bound to a variable. Similarly, after a field has been reassigned, the permission to the object it used to point to still exists and can be assigned to another reference. Giving expressions existential types allows us to later assign the object and make its permission consistent with the new reference.

We use a decidable fragment of linear logic, the multiplicative additive fragment (MALL), as our language of behavioral specification [24]. Throughout the typing rules, we will use the standard linear logic proof judgment, $\Gamma; \Delta \vdash P$, extensively. This judgment can be read as, "in the context of some typing information and a list of consumable resources, the predicate P can be proven true." The syntax for the permissions themselves are given in Figure 10.

The declarative nature of the linear logic judgment can make the reading of typing rules somewhat confusing. Often, this judgment appears to synthesize permissions "out of the blue," for example, in the P-TERM rule. Similarly, several typing rules divide the linear context in a seemingly arbitrary manner, written as (Δ, Δ'). In reality, the linear logic judgment is working like a constraint solver. In a typing derivation, different rules restrict the permissions or the context in various ways, and it is the job of the implementation to find a rearrangement of permissions that satisfies all of these constraints. The same judgment is also allowed to split permission types (Figure 3), and can therefore legally try even more possible rearrangements.

The most important new additions to our type system are the judgments shown in Figure 11. Rather than dispatch directly to the linear logic proof-judgment, the typing rules first dispatch to the "atomic-aware" version of this judgment, $\Gamma; \Delta \vdash_{\mathcal{E}} P$. It is the job of this judgment to ensure that predicates that must be proven do not depend on a share or pure reference being in a particular state, unless we know statically that we are inside an atomic block. In order to maintain this invariant, it is occasionally necessary to actively "forget" the state of an object pointed to by a share or pure permission. For example, in the typing rule for a method call, P-CALL, we sometimes must forget state information for potentially thread-shared permissions in the post-condition. It is acceptable for a method's post-condition to include share and pure permissions since that method may be called within an atomic block, but if that is not the case, these permissions must not be relied upon. The forget judgment, whose action is predicated upon $\mathcal{E}$, performs this work.

The typing rules themselves are given in Figure 12. Here we discuss each rule in turn.

- P-ATOMIC: The rule for typing atomic blocks types the sub-expression under the wt effect, since it is trivially known that this expression must be inside an atomic block. Because the atomic block itself may or may not be used inside of another atomic block (nesting atomic blocks is legal) we must use the forget$_{\mathcal{E}}$ judgment on the resulting permission.
- P-LET: In order to prove that a let expression is well-typed, we rely on e_1 being well-typed. Like the standard

$$\frac{}{\mathsf{forget}_{\mathsf{wt}}(P) = P} \qquad \frac{\mathcal{E} \neq \mathsf{wt} \quad \mathsf{forget}(P) = P'}{\mathsf{forget}_{\mathcal{E}}(P) = P'}$$

$$\frac{k = \mathsf{immutable}|\mathsf{unique}|\mathsf{full}}{\mathsf{forget}(k(r,\$)) = k(r,\$)} \qquad \frac{k = \mathsf{pure}|\mathsf{share}}{\mathsf{forget}(k(r,\$)) = k(r,?)}$$

$$\frac{\mathsf{forget}(P_1) = P_1' \quad \mathsf{forget}(P_2) = P_2' \quad \mathsf{op} = \otimes|\oplus}{\mathsf{forget}(P_1 \; \mathsf{op} \; P_2) = P_1' \; \mathsf{op} \; P_2'}$$

$$\frac{P = q|1|0|\top}{\mathsf{forget}(P) = P} \qquad \frac{\Gamma; \Delta \vdash P}{\Gamma; \Delta \vdash_{\mathsf{wt}} P}$$

$$\frac{\begin{array}{c}\mathcal{E} = \mathsf{ot}|\mathsf{emp} \quad \Gamma; \Delta \vdash P \\ k(r,\$) \notin \Delta, \; \textit{where } k = \mathsf{pure}|\mathsf{share} \textit{ and } \$ = s\end{array}}{\Gamma; \Delta \vdash_{\mathcal{E}} P}$$

$$\frac{}{k(r,s) \notin \cdot} \qquad \frac{k(r,s) \notin P \quad k(r,s) \notin \Delta}{k(r,s) \notin \Delta, P}$$

$$\frac{}{k(r,s) \notin k(r,?)} \qquad \frac{(k \neq k' | r \neq r' | s \neq \$')}{k(r,s) \notin k'(r',\$')}$$

$$\frac{k(r,s) \notin P_1 \quad k(r,s) \notin P_2 \quad \mathsf{op} = \otimes|\oplus}{k(r,s) \notin P_1 \; \mathsf{op} \; P_2} \qquad \frac{P = q|1|0|\top}{k(r,s) \notin P}$$

Figure 11. Forgetting and atomic-aware linear judgment

let rule, we then type e_2 assuming x has e_1's type. The somewhat unusual premise $\Gamma; \Delta', P \vdash_{\mathcal{E}} P'$ does not actively forget state information, which is done in other rules, rather it reestablishes for the purposes of the soundness proof that we do not know anything we should not about the state of pure and share permissions.

- P-CALL: This rule describes method calls. We retain the original restriction of Bierhoff and Aldrich's system that the receiver object must be in a packed state by noting that we could always pack to some intermediate state in the event of recursive calls. Since the post-condition could potentially contain state information about shared objects, we again use the $\mathsf{forget}_{\mathcal{E}}$ judgment. The notation $[t/x]P$ signifies capture-avoiding substitution and is used throughout. It means, "replace x with t in P, alpha-converting if necessary."
- P-SPAWN: In our language thread spawns are very similar to method calls. We require that threads be spawned at the outermost program expression, enforced by requiring the ot effect. This restriction can be relaxed by using one of the more permissive languages proposed by Moore and Grossman [25]. In some ways this rule is the most interesting because it formalizes our notion of aliased objects as an approximation of thread-shared objects. This rule returns no permissions to the calling context (signified by the 1 permission). Unlike synchronous method calls that can temporarily "borrow" an unshared writing permission and then return it to the calling context, this restriction requires the calling context to either give up its own writing permission permanently, or use permission splitting rules to create two shared permissions, one for the caller and one for the new thread.
- P-METH: Method bodies are actually type-checked twice. Because we do no know statically whether or not a method will be executing within a transaction, we type-check method once with the emp effect, which establishes that the method is legal outside of a transaction. Then the method is type-checked a second time with the wt effect in order to verify that it meets its specification. This behavior is essential to typing examples such as the `trySendMsg` method in Figure 1, where state information about share or pure references is used in subsequent lines of code. It is the responsibility of the P-CALL rule, to not allow these sorts of methods to be called, nor their post-conditions to be relied upon outside of transactions. Note also that the post-condition that is actually proved is $P_r \otimes \top$. The linear logic we use does not allow for unused linear resources. Therefore, if there are extra permissions created during the course of the method body, those permissions can legally be ignored by using them to prove $\top$.
- P-UNPACK-WT: The unpack expression is broken into two rules. As discussed in Section 2, our system requires that share, pure and full permissions be unpacked within an atomic block. Therefore, if the unpack expression is type-checked under the wt effect, k is allowed to be a permission of any type. This is in contrast to the P-UNPACK rule which requires $k = \mathsf{immutable}|\mathsf{unique}$. First off, in order to unpack an object we must prove that the receiver object is in the state that we claim. This is done using the linear proof judgment, $\Gamma; \Delta \vdash_{\mathsf{wt}} k(this, s)$. Since we divided the linear context into two, this will also prevent the sub-expression from relying on this fact, since the state invariant will be in flux. Then, the sub-expression can be typed with information about the object's fields implied by the state invariant, $\mathsf{inv}_C(s,k)$. This judgment, shown in Figure 13, has two roles. It will look up state invariant predicate for state s from the class definition, and it will also "down-grade" writing permissions if necessary. Down-grading is necessary when a read-only permission (immutable or pure) is being unpacked. During this process, we temporarily change writing permissions on that object's fields to read-only permissions. This is performed by the dg predicate, also seen in Figure 13. The sub-expression is also given $\mathsf{unpacked}(k,s)$, which implies that the receiver is temporarily unpacked.
- P-UNPACK: This rule is similar, but occurs when not inside a transaction. We are limited to unpacking unique and immutable permissions.

- P-ASSIGN: When we assign a value to a field, the only sort of effect allowed in the calculus, we must first prove that the value has some permission and that it is the same type as the ith field of class C to which we are assigning. The next premise says that we can prove the field currently has some permission and that the receiver is unpacked. The unpacked permission must be a modifying permission. The resulting permission of the entire expression is the permission to the field's old value, suitable for assignment to another field, as well as permission to the field's new value and the unpack predicate.
- P-NEW: In order to instantiate a new object, we must be able to prove the state invariant for the initial state of that object. This is done by looking up the state invariant P for the initial state, and proving it when treating the permissions to the constructor arguments as fields of the object. These permissions are consumed, and the result is a unique permission to the object in the initial state.
- P-TERM: Individual terms are given a permission and a type by type-checking the term, proving some permission P from the linear context and then pulling the term itself out of the permission, resulting in an existentially bound one.
- P-IF: The conditional expression binds a boolean term in both the branch expressions. Each branch is type-checked with the knowledge that the term is either true or false. The resulting permission for the entire expression is a disjunction, since the permission from either branch could be produced.
- P-FIELD: A field read proves some permission P which contains permissions for f_i and existentially binds it so that it can be assigned to another reference.
- P-PROG: A program type-checks if all of its classes are well-formed and the single, top-level expression type-checks outside of a transaction.
- P-CLASS: A class declaration is well-formed if its parts are well-formed.
- P-FDECL: The well-formedness rule for field declarations is somewhat informal, as are the remaining well-formedness rules. This rule states that a field declaration is well-formed if its name is unique inside the current class, and if it type is either a boolean or one of the declared class types.
- P-CTR: A declaration of the initial state is well-formed if the state is mentions is actually one defined in the current class.
- P-SINV: A state invariant declaration is well-formed if three conditions hold. The state name must be unique within the current class. Any references mentioned in access permissions inside P must be fields of the current class. Finally, invariants describing share and pure permissions to fields cannot mention specific state information.

Dynamic semantics for our language are given in the accompanying technical report [3]. These rules are extremely similar to those of the Weak language [25]. They differ primarily in that there are additional technical requirements for the firing of rules, necessary for our proof of soundness. While the formal operational semantics of this language must actively maintain information regarding the states and permissions of each object, the language itself does not actually change the run-time behavior of a Java-like language with weak atomicity, and requires none of our typing information to be present at run-time.

In the technical report, we state that this core language is sound. Informally soundness means the following:

1. Well-typed thread pools either consist exclusively of evaluated threads, or can take an evaluation step. There are two sub-cases for individual threads:
   (a) No single thread in the thread pool is executing inside of an atomic region, and therefore any arbitrary thread in the thread pool must be able to take a step.
   (b) Exactly one thread in the thread pool is executing inside of an atomic region, and therefore that thread must be able to take a step.
2. Any thread pool that is well-typed and can take an evaluation step must step to a well-typed thread pool. The burden of proof for this fact is delegated to individual threads which must in turn step to a well-typed expression.

The most important part of maintaining a well-typed thread pool is maintaining a well-typed heap and per-thread stacks. This well-typedness restricts how many threads can know the definite state of objects in the system. For instance, in a well-typed thread pool, at most one thread can have definite knowledge about the state of a share or pure object at any given time. Since we must reestablish well-typedness after each step, we know that this invariant holds.

Because well-typed threads can always step, it is never the case that the running system arrives at a evaluation step where an object should be in one state but instead is in another.

4. Implementation and Examples

We have begun investigating the applicability of our approach by annotating several real and realistic programs and verifying them with a prototype checker. In this section we briefly describe the checker as well as the examples that we have verified thus far.

4.1 Prototype Checker

We have extended a static typestate checker [5] to check the rules described in this paper in Java language programs. This checker is a modular, branch-sensitive data-flow analy-

$$\dfrac{\begin{array}{c}\Gamma;\Delta;\textsf{wt}\vdash e:\exists x:T.P\\ \textsf{forget}_{\mathcal{E}}(P)=P'\end{array}}{\Gamma;\Delta;\mathcal{E}\vdash\texttt{atomic}\ (e):\exists x:T.P'}\ \textsc{P-Atomic}\qquad\qquad \dfrac{\begin{array}{c}\Gamma;\Delta;\mathcal{E}\vdash e_1;\exists x:T.P\\ \Gamma;\Delta',P\vdash_{\mathcal{E}}P'\quad(\Gamma,x:T);P';\mathcal{E}\vdash e_2:E\end{array}}{\Gamma;(\Delta,\Delta');\mathcal{E}\vdash\texttt{let}\ x=e_1\ \texttt{in}\ e_2:E}\ \textsc{P-Let}$$

$$\dfrac{\begin{array}{c}\Gamma\vdash t_o:C_o\quad\Gamma\vdash\overline{t:T}\quad\Gamma;\Delta\vdash_{\mathcal{E}}[t_o/\mathit{this}][\overline{t}/\overline{x}]P\\ \textsf{mtype}(m,C_o)=\forall\overline{x:T}.P\multimap\exists\mathit{result}:T.P_r\quad\textsf{unpacked}(k',\$')\notin\Delta\\ \textsf{forget}_{\mathcal{E}}(P_r)=P_r'\end{array}}{\Gamma;\Delta;\mathcal{E}\vdash t_o.m(\overline{t}):\exists\mathit{result}:T.[t_o/\mathit{this}][\overline{t}/\overline{x}]P_r'}\ \textsc{P-Call}$$

$$\dfrac{\begin{array}{c}\Gamma\vdash t_o:C_o\quad\Gamma\vdash\overline{t:T}\quad\Gamma;\Delta\vdash_{\textsf{ot}}[t_o/\mathit{this}][\overline{t}/\overline{x}]P\\ \textsf{mtype}(m,C_o)=\forall\overline{x:T}.P\multimap E\quad\textsf{unpacked}(k',\$')\notin(\Delta,\Delta')\end{array}}{\Gamma;\Delta;\textsf{ot}\vdash\texttt{spawn}\ (t_o.m(\overline{t})):\exists\_:\texttt{bool}.1}\ \textsc{P-Spawn}$$

$$\dfrac{\begin{array}{c}(\overline{x:T},\texttt{this}:C);P;\textsf{emp}\vdash_C e:E'\\ (\overline{x:T},\texttt{this}:C);P;\textsf{wt}\vdash_C e:\exists\mathit{result}:T_r.P_r\otimes\top\quad E=\exists\mathit{result}:T_r.P_r\\ E=\textsf{forget}_{\textsf{emp}}(E')\end{array}}{T_r m(\overline{Tx}):P\multimap E=e\ \textsf{ok in}\ C}\ \textsc{P-Meth}$$

$$\dfrac{\begin{array}{c}\Gamma;\Delta\vdash^C_{\textsf{wt}}k(this,\$)\quad\textsf{unpacked}(k',\$')\notin(\Delta,\Delta')\\ \Gamma;(\Delta',\textsf{inv}_C(\$,k),\textsf{unpacked}(k,\$));\textsf{wt}\vdash^C e:E\end{array}}{\Gamma;(\Delta,\Delta');\textsf{wt}\vdash^C\texttt{unpack}(k,\$)\ \texttt{in}\ e:E}\ \textsc{P-Unpack-Wt}$$

$$\dfrac{\begin{array}{c}\mathcal{E}\neq\textsf{wt}\quad k=\textsf{immutable}|\textsf{unique}\\ \Gamma;\Delta\vdash^C_{\mathcal{E}}k(this,\$)\quad\textsf{unpacked}(k',\$')\notin(\Delta,\Delta')\\ \Gamma;(\Delta',\textsf{inv}_C(\$,k),\textsf{unpacked}(k,\$));\mathcal{E}\vdash^C e:E\end{array}}{\Gamma;(\Delta,\Delta');\mathcal{E}\vdash^C\texttt{unpack}(k,\$)\ \texttt{in}\ e:E}\ \textsc{P-Unpack}$$

$$\dfrac{\begin{array}{c}\textsf{forget}_{\mathcal{E}}(k(this,s))=k(this,\$)\quad\Gamma;(\Delta',k(this,\$));\mathcal{E};\vdash^C e:E\\ \textsf{localFields}(C)=\overline{f:T}\quad\textit{no fields in}\ \Delta'\\ \textsf{readonly}(k)\ \textit{implies}\ \$'=s\quad\Gamma;\Delta\vdash_{\mathcal{E}}\textsf{inv}_C(s,k)\otimes\textsf{unpacked}(k,\$')\end{array}}{\Gamma;(\Delta,\Delta');\mathcal{E}\vdash^C\texttt{pack to}\ s\ \texttt{in}\ e:E}\ \textsc{P-Pack}$$

$$\dfrac{\begin{array}{c}\Gamma;\Delta;\mathcal{E}\vdash t:\exists x:T_i.P\quad\Gamma;\Delta'\vdash^C_{\mathcal{E}}[f_i/x']P'\otimes p\\ \textsf{localFields}(C)=\overline{f:T}\quad p=\textsf{unpacked}(k,s)\quad\textsf{writes}(k)\end{array}}{\Gamma;(\Delta,\Delta');\mathcal{E}\vdash^C f_i:=t:\exists x':T_i.P'\otimes[f_i/x]P\otimes p}\ \textsc{P-Assign}$$

$$\dfrac{\Gamma\vdash\overline{t:T}\quad\textsf{init}(C)=\langle\exists\overline{f:T}.P,s\rangle\quad\Gamma;\Delta\vdash_{\mathcal{E}}[\overline{t/f}]P}{\Gamma;\Delta;\mathcal{E}\vdash\texttt{new}\ C(\overline{t}):\exists x:C.\textsf{unique}(x,s)}\ \textsc{P-New}\qquad\qquad\dfrac{\Gamma\vdash t:T\quad\Gamma;\Delta\vdash_{\mathcal{E}}P}{\Gamma;\Delta;\mathcal{E}\vdash t:\exists x:T.[x/t]P}\ \textsc{P-Term}$$

$$\dfrac{\begin{array}{r}(\Gamma,t=\texttt{true});\Delta;\mathcal{E}\vdash\exists x:T.P_1\\ \Gamma\vdash t:\texttt{bool}\quad(\Gamma,t=\texttt{false});\Delta;\mathcal{E}\vdash\exists x:T.P_2\end{array}}{\Gamma;\Delta;\mathcal{E}\vdash\texttt{if}(t,e_1,e_2):\exists x:T.P_1\oplus P_2}\ \textsc{P-If}\qquad\qquad\dfrac{\textsf{localFields}(C)=\overline{f:T}\quad\Gamma;\Delta\vdash_{\mathcal{E}}P}{\Gamma;\Delta;\mathcal{E}\vdash^C f_i:\exists x:T_i[x/f_i]P}\ \textsc{P-Field}$$

$$\dfrac{\overline{CL}\ \textsf{ok}\quad\cdot;\cdot;\textsf{ot}\vdash e:E}{\langle\overline{CL},e\rangle:E}\ \textsc{P-Prog}\quad\dfrac{\overline{F}\ \textsf{ok in}\ C\ldots\overline{M}\ \textsf{ok in}\ C}{\texttt{class}\ C\ \{\ \overline{F}\ I\ \overline{N}\ \overline{M}\}\ \textsf{ok}}\ \textsc{P-Class}\quad\dfrac{f_i\ \textit{unique}\quad T_i\in\overline{CL}\cup\{\texttt{bool}\}}{\overline{f:T}\ \textsf{ok in}\ C}\ \textsc{P-FDecl}$$

$$\dfrac{\texttt{class}\ C\{\ldots s=P\ldots\}\in\overline{CL}}{\texttt{initially}\langle s\rangle\ \textsf{ok in}\ C}\ \textsc{P-Ctr}\qquad\qquad\dfrac{\begin{array}{c}s_i\ \textit{unique}\quad r\in P_i\supset r\in\overline{F}\in C\\ r(k,\$)\in P_i\ \textit{where}\ k=\textsf{share}|\textsf{pure}\supset\$=?\end{array}}{\overline{s=P}\ \textsf{ok in}\ C}\ \textsc{P-SInv}$$

Figure 12. Typing Rules. Helper judgments (localFields, init, mtype, inv, and writes) defined in Figure 13.

sis that uses specialized Java annotations as behavioral and access specifications. For example, the `disconnect` method of the Connection class from Figure 2 is annotated with the following specification:

```
@Share(requires="CONNECTED", ensures="IDLE")
```

This indicates the method requires a share permission to the receiver which must be in the connected state, and will return that same permission but with the receiver in the idle

$$\frac{\texttt{class}\ C\ \{\ldots s = P \ldots\} \in \overline{CL}}{\mathsf{inv}_C(s) = P} \qquad \frac{\mathsf{inv}_C(s) = P \quad \mathsf{dg}(P, k) = P'}{\mathsf{inv}_C(s, k) = P'} \qquad \frac{}{\mathsf{inv}_C(?, k) = 1}$$

$$\frac{\mathsf{dg}(P_1, k) = P_1' \quad \mathsf{dg}(P_2, k) = P_2' \quad \mathsf{op} = \otimes|\oplus}{\mathsf{dg}(P_1\ \mathsf{op}\ P_2, k) = P_1'\ \mathsf{op}\ P_2'} \qquad \frac{(k, k') \circlearrowleft k''}{\mathsf{dg}(k(r, \$), k') = k''(r, \$)}$$

$$\frac{k' \neq \mathsf{pure}|\mathsf{immutable}}{(k, k') \circlearrowleft k} \qquad \frac{k = \mathsf{unique}|\mathsf{full}|\mathsf{immutable} \quad k' = \mathsf{pure}|\mathsf{immutable}}{(k, k') \circlearrowleft \mathsf{immutable}}$$

$$\frac{k = \mathsf{share}|\mathsf{pure} \quad k' = \mathsf{pure}|\mathsf{immutable}}{(k, k') \circlearrowleft \mathsf{pure}} \qquad \frac{\texttt{class}\ C\{\ldots \overline{F} \ldots\} \in \overline{CL}}{\mathsf{localFields}(C) = \overline{F}}$$

$$\frac{\texttt{class}\ C\{\ldots \overline{M} \ldots\} \in \overline{CL} \quad T_r\ m(\overline{T\ x}) : P \multimap \exists \mathit{result} : T_r.P' \in \overline{M}}{\mathsf{mtype}(m, C) = \forall \overline{x : T}.P \multimap \exists \mathit{result} : T_r.P'}$$

$$\frac{\texttt{class}\ C\{\ldots \texttt{initially}\langle s \rangle \ldots\} \quad \mathsf{inv}_C(s) = P}{\mathsf{init}(C) = \langle \exists \overline{f : T}.P, s \rangle}$$

$$\frac{}{\mathsf{writes}(\mathsf{unique})} \quad \frac{}{\mathsf{writes}(\mathsf{full})} \quad \frac{}{\mathsf{writes}(\mathsf{share})} \quad \frac{}{\mathsf{readonly}(\mathsf{pure})} \quad \frac{}{\mathsf{readonly}(\mathsf{immutable})}$$

Figure 13. Helper judgments

state. Similar annotations exist for state invariants. Because of our desire to use existing, Java-based tools, we use Java's labeled statement with the label value "atomic" to delineate atomic blocks. For example:

```
atomic: { /* code that will
            execute atomically */ }
```

This legal Java code allows us to get around our inability to annotate arbitrary blocks using Java's annotation facility. We have modified AtomJava [19], a tool which provides atomicity via source-to-source translation, to use labeled statements as atomic blocks so that our examples can be run.

While the formal language presented in this paper requires the programmer to explicitly pack and unpack the receiver, our checker does not. Before method calls and method returns, the checker automatically attempts to pack the receiver to some reasonable state. If one state does not permit permission constraints to be satisfied, other states are tried until a good one can be found or no more states are available. Unpacking is also done automatically before field reads and writes.

Our checker does allow some of the more advanced features of Bierhoff and Aldrich's system [4] that were not discussed in this work. For instance, it supports fractional permissions which allow shared permissions to be joined back together to produce unique ones. It also allows a developer to create more complex state hierarchies.

As this time our checker does not recognize full linear logic specifications, and accepts only a limited sub-set, although enough to specify all of the examples in this paper. Finally, reading from or writing to static fields requires being within an atomic block, since in general, even if a static field is the only field to point to a particular object many threads can access it simultaneously.

4.2 Verified Examples

In addition to a corrected version of the running example from Figures 5 and 4, we have used our implementation to verify several other examples.

JGroups Application In this example, we annotated the JChannel class of the JGroups open source library and verified that a demo application was using it correctly. JGroups [21] is an open-source library for use by developers of multi-cast network applications. The JChannel class is a thread-safe channel abstraction that allows a host to connect and send messages to a group of other hosts. This particular class seemed to be a good candidate for specification because its original developers actually provided the same specification in the form of source-code comments:

The FSM for a channel is roughly as follows: a channel is created (unconnected). The channel is connected to a group (connected). Messages can now be sent and received. The channel is disconnected from the group (unconnected). The channel could now be connected to a different group again. The channel is closed (closed).

Therefore formally specifying and statically checking that this class is used in accordance with its informal specification seemed appropriate. After specifying this class, we ran our analysis on the CausalDemo class. This demo, provided with JGroups, creates multiple threads, one of which is responsible for closing the channel. This client was successfully verified.

Reservation Manager Reservation Manager is a multi-threaded application of our own design. It is meant to be similar in architecture to a vacation reservation system. In it, various threads acting on behalf of clients attempt to reserve bus or plane tickets. This application requires client threads to atomically check for seat availability and make a reservation. This application has some interesting object invariants. For example, once an bus itinerary has been issued to a passenger, he can upgrade to a plane flight, as long as the demand for bus tickets is high enough. Once an itinerary has been issued, it must at all times represent either a valid bus or plane trip. At the same time, a daemon thread will occasionally send a (simulated) email describing an itinerary to each itinerary holder, therefore it is important that any upgrades happen atomically. We have successfully verified this entire application.

Request Processor Request Processor is another multi-threaded application of our own design, partially shown in Figure 14. This program is meant to be similar in spirit to a server application where processes are received and farmed off to other threads for handling. Upon initialization, the RequestProcessor creates a request pipe object which acts as an intermediary between the request processor, which receives the requests, and the request handlers which handle them. This program is notable because each side of the producer/consumer architecture has a different permission to the shared object. The RequestProcessor has a full permission while the handlers themselves have only pure permissions.

Full source for all of the examples in this paper can be found at: *www.cs.cmu.edu/~nbeckman/research/atomicver/.*

In the future we hope to improve the quality of our checker, and verify larger and more realistic examples. Our experiences with these smaller examples, however, lead us to believe that this is a feasible goal.

```
class RequestProcessor {
  states IDLE, RUNNING;

  IDLE := full(requestPipe, closed)
  RUNNING := full(requestPipe, opened)

  RequestPipe requestPipe = new RequestPipe();

  void start() :
    unique(this, IDLE) ⊸ unique(this, RUNNING)
  {
    this.requestPipe.open();
    // Handler(rp) : pure(rp, ?) ⊸ 1
    (new Thread(new
           Handler(this.requestPipe))).start();
    (new Thread(new
           Handler(this.requestPipe))).start();
    return;
  }

  void send(String str) :
    unique(this, RUNNING) ⊗ immutable(str, alive) ⊸
        unique(this, RUNNING)
  {
    this.requestPipe.send(str);
    return;
  }

  void stop() :
    unique(this, RUNNING) ⊸ unique(this, IDLE)
  {
    this.requestPipe.close();
    return;
  }
}
```

Figure 14. RequestProcessor, an example of a server-like program where class invariants depend on thread-shared objects.

5. Related Work

5.1 Verifying Behavior of Concurrent Programs.

The work that most closely resembles our own was developed as part of the Spec$^{\#}$ Project. Jacobs et al. [20] 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 heritage in research methodology [2]. Nonetheless, we believe our work to be different in several important ways. First, they use ownership as their underlying methodology, which imposes some hierarchical restrictions on the architecture of an application. On the other hand, their system allows more expressive specifications, as behaviors can be specified in first-order predicate logic, rather than typestate. While we believe our approach would neatly accommodate more expressive specifications which we plan to investigate as part of future work, typestate provides a simple, programmer-understandable abstraction of application behavior. This system does have a proof of soundness but provides neither formal typing rules nor a formal semantics.

Their system also is restrictive in the types of objects that can be mentioned in object 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 14 where the invariant of the RequestProcessor class depends on the thread-shared RequestPipe object, 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 [28]. 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 fails to terminate about half of the time.

Calvin-R [11] allows the specification and verification of method annotations in concurrent software and also has a more expressive specification language. This work differs in three significant ways. First, the programmer is required to specify a locking discipline (e.g., variable x is protected by lock y). Incorrect specification could lead to the technique's false verification of a program. Second, this system does not allow for the specification of object invariants. Finally, this system does not offer a solution to the problem of tracking aliased objects.

Harris and Peyton Jones [16] introduce a mechanism for STM Haskell that ensures a given invariant will not be violated during a given execution of a program. However, this is a dynamic technique that cannot guarantee conformance for all executions.

Our reasoning for permissions is somewhat similar to assume/guarantee reasoning [22]. Our full permissions *assume* that other references in the program will not modify an object in memory. pure permissions, on the other hand, assume nothing, but *guarantee* that they will not modify the location.

5.2 Race Detection.

There has been much work in the automated prevention of race conditions.

Dynamic race detection tools have been explored [30, 33], but they cannot guarantee absence of race conditions across all program runs.

Model-checking approaches have also been explored [17, 31]. These work by abstractly exploring possible thread interleavings in order to find ones where a there is no ordering on a read and write to the same memory location. These approaches must deal with a very large state space since there is a large number of potential thread interleavings. They are also not modular.

Closer in style to our approach is the large number of static, flow-based race detection tools and race-free type systems [6, 13, 15, 27, 9]. These approaches work by statically correlating an object (or any piece of memory) with the lock that protects it. Each varies in the level of expressiveness (for type-systems) and numbers of false-positives (for static analyses) it provides. Some works use ownership types [6, 13] while others use regions [15], both means of controlling aliasing so that modular analysis can be performed. Locksmith [27] and RacerX [9], on the other hand, each must perform some form of inter-procedural analysis, unlike our approach.

The fundamental difference between all of these static race detectors and our approach is that our approach attempts to prevent "application-level" race conditions, that is race conditions that could lead to a violation in program invariants, which these approaches attempt to prevent data races. Data race freedom, that is the absence of unordered reads and writes to the same piece of memory, is absolutely a hard and important problem. The Java memory model, for instance, provides sequential consistency only in the absence of data races. However, we believe that from a programmer's perspective, the absence of data races is just the tip of the iceberg. Once a program is data-race free, a programmer must ensure that atomicity is provided when it is necessary to ensure program correctness. It is the introduction of behavioral annotations that distinguish our approach from these approaches. Finally, we focus our approach on atomic blocks and STM, while these techniques attack lock-based synchronization.

Atomicity checkers [10, 29, 18], on the other hand, do attempt to provide a programmer with atomicity. Given a method that has been annotated by the programmer as needing to be atomic, these approaches will guarantee that the method body uses locks in such a way that it will execute atomically, as if no other threads had been interleaved. Our approach actually helps a programmer determine what code needs to be atomic, and we delegate the task of ensuring atomicity to the underlying STM system. One could imagine using these techniques in a complementary manner. Interestingly, the type system proposed by Sasturkar et al. [29] also includes a *read-only* type, which is similar in spirit to our immutable permission.

6. Future Work

We are currently pursuing a number of future courses of research. While our work is an attempt to advance the work of Bierhoff and Aldrich [4] to the world of concurrent software, we first wanted to study the problems of concurrency in relative isolation. Therefore, we have not included many of the more advanced features of that system into the work presented here. These features, like fractional permissions and support for sub-typing and inheritance, would make our system even more expressive, and we plan to reintroduce them into our system. This should be relatively straightforward.

Additionally, we are attempting to determine what sorts of access permissions might be more useful in a thread-shared context. At the moment, permissions that are thread-shared, and permissions that are merely aliased locally are not distinguishable, and we would like to tease them apart. For instance, we would like to have a thread-local version of the share permission that would not require synchronization.

We have also begun developing an implementation of software transactional memory that uses these same permission annotations as a means of improving run-time performance by eliminating unnecessary synchronization and logging. While the implementation is complete, we have only performed preliminary experiments and have not yet established the efficacy of our technique.

Finally, we would like to see a greater usage of STM for the purposes of static verification by those in the object-oriented language community. Currently, most existing flow analyses and verification tools are unsound in the face of concurrency, and those that are not impose a great annotation burden on the programmer, in addition to any burden imposed by the single-threaded version of the analysis. In this work we were able to prove our concurrent language sound, thanks in part to the clean dynamic semantics of atomic blocks. If we were to extend Dan Grossman's Garbage Collection/STM analogy [14], we would say the following: In the same way garbage collection allows proofs of type safety that would difficult or impossible in a language with explicit memory allocation and reclamation, transactional memory will allow proofs of type safety for multi-threaded languages, when doing the same with lock-based synchronization would be difficult or impossible. The performance of STM implementations continues to improve [1], and we believe this will also help to encourage the adaptation of static analyses for use in concurrent programs.

7. Conclusion

In this paper we described a static analysis, formalized as a programming language, that can help to ensure the proper usage of atomic blocks. The atomic block, provided by transactional memory implementations, is a simple concurrency primitive, when compared with locks, but can still be used incorrectly. Our type system ensures that, up to the method and object behavioral specifications, race conditions will not occur and object invariants will be preserved. We believe this is the first work to attempt to statically ensure the correct usage of transactional memory in object-oriented languages. This language uses access permissions, a means of denoting the manner in which objects may be aliased, as an approximation for whether or not objects are thread-shared, which in turn helps determine whether or not code must be inside of an atomic block. We use typestate as our language of specification, and track transactions using a simple type-and-effect system. We have proved this language sound in our accompanying technical report [3]. Finally, we have created a prototype static analysis for the Java programming language based on the system described in this paper. We have used it to verify several realistic concurrent programs.

Acknowledgments

The authors would like to acknowledge the PLAID group and Todd Millstein for their helpful comments.

References

[1] Adl-Tabatabai, A., Lewis, B. T., Menon, V. S., Murphy, B. R., Saha, B., Shpeisman, T. Compiler and Runtime Optimizations for Efficient Software Transactional Memory. In *PLDI '06*, Ottawa, Canada. June, 2006.

[2] Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., Schulte, W. Verification of Object-Oriented Programs with Invariants. In *Journal of Object Technology*, 3(6). June, 2004.

[3] Beckman, N. E., Aldrich, J. Verifying Correct Usage of Atomic Blocks with Typestate: Technical Companion. `www.cs.cmu.edu/~nbeckman/oopsla08/proof.pdf`. Accessible with username, `"oopsla2008"` and password, `"IReview"`.

[4] Bierhoff, K., Aldrich, J. Modular Typestate Checking of Aliased Objects. In *OOPSLA '07*, Montreal, Canada. October, 2007.

[5] Bierhoff, K. Aldrich, J. PLURAL: Checking Protocol Compliance under Aliasing. Demo in *ICSE 2008 Companion*, Leipzig, Germany. May, 2008. To appear.

[6] Boyapati, C., Lee, R., Rinard, M. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA '02*, Seattle, WA. November, 2002.

[7] Boyland, J., Checking Interference with Fractional Permissions. In *International Symposium on Static Analysis*, San Diego, CA. June, 2003.

[8] DeLine, R., Fähndrich, M. Typestates for Objects. In *ECOOP '04*, Oslo, Norway. June, 2004.

[9] Engler, D., Ashcraft, K. Racerx: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP '03*, Bolton Landing, New York. October, 2003.

[10] Flanagan, C., Qadeer, S. A Type and Effect System for Atomicity. ACM In *PLDI '03*, San Diego, CA. June, 2003.

[11] Freund, S. N., Qadeer, S. Checking Concise Specifications for Multi-threaded Software. *Journal of Object Technology*, 3(6). June, 2004.

[12] Girard, J. Y., Linear Logic. In *Theoretical Computer Science*, 50, 1987.

[13] Greenhouse, A., Scherlis, W. L. Assuring and Evolving Concurrent Programs: Annotations and Policy. In *ICSE '02*, Orlando, FL. May, 2002.

[14] Grossman, D. The Transactional Memory / Garbage Collection Analogy. In *OOPSLA '07*, Montreal, Canada. October, 2007.

[15] Grossman, D. Type-Safe Multithreading in Cyclone. *TLDI '03*, New Orleans, LA. January, 2003.

[16] Harris, T., Peyton Jones, S. Transactional Memory With Data Invariants. In *TRANSACT ‘06*, Ottawa, Canada. June, 2006.

[17] Henzinger, T. A., Jhala, R., Majumdar, R. Race Checking by Context Inference. In *PLDI ‘04*, Washington, D.C. June, 2004.

[18] Hicks, M., Foster, J. S., Pratikakis, P. Lock Inference for Atomic Sections. *TRANSACT ‘06*. Ottawa, Canada. June 2006.

[19] Hindman, B., Grossman, D. Atomicity via Source-to-Source Translaction. Workshop on Memory Systems Performance and Correctness, San Jose, CA. October, 2006.

[20] Jacobs, B., Rustan, K., Leino, M., Piessens, F., Schulte, W. Safe Concurrency for Aggregate Objects with Invariants. In *Conference on Software Engineering and Formal Methods*, Koblenz, Germany. September, 2005.

[21] JGroups Website `www.jgroups.org`

[22] Jones, C. B., Tentative Steps Toward a Development Method for Interfering Programs. In *TOPLAS* 5(4), 1983.

[23] Herlihy, M., Luchangco, V., Moir, M., Scherer, W. Software Transactional Memory for Dynamic-Sized Data Structures. In *PODC ‘03*, Boston, MA. July, 2003.

[24] Lincoln, P., Scedrov, A. First-Order Linear Logic Without Modalities is NEXPTIME-Hard. In *Theoretical Computer Science*, 135, 1994.

[25] Moore, K., Grossman, D. High-Level Small-Step Operational Semantics for Transactions. In *POPL ‘08*, San Francisco, CA. January, 2008.

[26] Oaks, S., Wong, H. Java Threads, Third Edition, page 164. O’Reilly, Sebastopal, CA. 2004.

[27] Pratikakis, P., Foster, J. S., Hicks, M. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI ‘06*, Ottawa, Canada. June 2006.

[28] Rodriguez, E., Dwyer, M., Flanagan, C., Hatcliff, J., Leavens, G. T., Robby. Extending Sequential Specification Techniques for Modular Specification and Verification of Multithreaded Programs. In *ECOOP ‘05*, Glasglow, Scotland. July, 2005.

[29] Sasturkar, A., Agarwal, R., Wang, L., Stoller, S. D. Automated Type-Based Analysis of Data Races and Atomicity. In *PPoPP ‘05*, Chicago, IL. June, 2005.

[30] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *ACM Transactions on Computer Systems* 15(4), 1997.

[31] Stoller, S. D. Model-Checking Multi-Threaded Distributed Java Programs. In *SPIN ‘00*, Stanford, CA. August, 2000.

[32] Wadler, P. Linear Types Can Change the World! In *Working Conference on Programming Concepts and Methods*, Sea of Galilee, Israel. 1990.

[33] Yu, Y., Rodeheffer, T., Chen, Wei. Racetrack: Efficient Detection of Data Race Conditions Via Adaptive Tracking. In *SOSP ‘05*, Brighton, United Kingdom. October, 2005.