Verifying Correct Usage of Atomic Blocks and Typestate

Nels E. Beckman Kevin Bierhoff Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
{nbeckman,kbierhof,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 intraprocedural static analysis, formalized as a type system and proven sound, that helps programmers use atomic blocks correctly. Using *access permissions*, which describe how objects are aliased and modified, our system statically prevents race conditions and enforces typestate properties in concurrent programs. 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*]: Mechanical Verification

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 in the field of computer science have begun investigating new techniques for the development of software that can actually take advantage of more cores.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

Among the large number of recent proposals, transactional memory (TM) 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. Most realizations of transactional memory provide 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 approach is "transactional," because atomic blocks are usually implemented as memory transactions which abort and retry in the event a thread witnesses an inconsistent view of memory.

However, as some of TM's greatest proponents will tell you, while atomic sections are a vast improvement over lock-based synchronization, they are far from perfect (Grossman 2007). Atomic sections by themselves do not guarantee correct synchronization, even when mutual exclusion is the only synchronization primitive needed, because they can still be used incorrectly. Even if every access to thread-shared memory is performed inside of an atomic block, race conditions at the level of program logic, or "high-level data races," (Artho et al. 2003) can still occur.

For the scope of this work, we consider how these race conditions can lead to misuse of object protocols. Our goal is to statically prevent races on the abstract state of an object, as well as violations of an object's concrete state invariants due to concurrent access.

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 a remote and local host. The GUI thread sends messages, and opens and closes the connection in response to local user events, while the network-monitoring thread closes the connection in response to a remote user event.

In Figure 1 the `trySendMsg` method, invoked 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 `send` methods of the Connection class are properly synchronized, reading

```
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.

and modifying thread-shared fields inside of atomic blocks. However, a race condition exists on the abstract state of the connection object. The GUI thread relies on the connection remaining in the connected state in between the call to `isConnected` and the call to `send`. If the network thread were to close the connection before the GUI thread's call to `send`, this call would be invalid and would cause a null-pointer exception.

The second example, shown in Figure 2, shows how misuse of atomic blocks can lead to violations of object invariants, thus leading to improper implementation of object protocols. 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 the 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 is done within atomic blocks.

```
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.

Once again, trouble occurs even though shared memory is accessed exclusively within atomic blocks. Concurrent access to the connection object has the potential to cause a violation of its invariants. Assume that the `disconnect` method is being executed by the network thread. If the GUI thread were to 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[1].

In this paper, we describe a Java-like programming language whose type system statically prevents misuse and incorrect implementation of object protocols in concurrent systems. 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 (Strom and Yemini 1986) specifications as the language of invariants, and object permissions (Boyland 2003) to approximate whether or not an object can be thread-shared. Our work builds upon recent work for verifying typestate of aliased objects (Bierhoff and Aldrich 2007).

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.

[1] While the race condition in this short illustrative example may seem unrealistic, it is more likely to occur in a situation when the method is longer and the programmer is motivated to make atomic blocks as short as possible to maximize concurrency.

The type system of this language guarantees that there are no race conditions on the abstract state of an object. If a method call requires the receiver object to be in some state, at run-time the object will be in that state. Furthermore, the specified invariants of these abstract states will be preserved, even in the face of concurrent access.

- 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. Our solution is an improvement over existing, lock-based approaches (Jacobs et al. 2005; Rodriguez et al. 2005) because it does not impose hierarchical restrictions on aliasing, and because our specifications are more compositional.
- We have proved soundness for a core subset of this language in the accompanying technical report (Beckman and Aldrich 2008).
- To our knowledge this is the first work that statically verifies 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.

Existing work on data race detection (Boyapati et al. 2002; Pratikakis et al. 2006; Engler and Ashcraft 2003) 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 (Jacobs et al. 2005; Jones 1983; Owicki and Gries 1976; Vaziri et al. 2006) 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.

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:

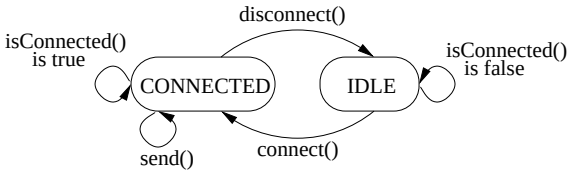
- We use typestate specifications on methods and classes to say which abstract state an object must be in before calling a method on it, and which concrete states an object's fields must be in at the end of a method call. (In principle, other behavioral specifications would work as well.)
- Object references 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 method bodies, discarding knowledge about the state of an object when the reference to that object indicates it may be modified by other threads *and* we cannot determine statically that we are within an atomic section.

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

2.1 Typestate Specifications

Our approach uses typestate (Strom and Yemini 1986) as the language of behavioral specification. A specification tells the system 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 (DeLine and Fähndrich 2004). Our work, in particular, adapts the approach of Bierhoff and Aldrich (2007) for use in concurrent settings. In the approach proposed by Bierhoff and Aldrich, object states are tracked statically using linear logic predicates (Girard 1987) 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 "default," which are known ahead of time. "?" represents a lack of knowledge about the state of an object. "default," on the other hand, is the default state given to an object whose class defines no abstract states.

2.2 Access Permissions

Access permissions (Bierhoff and Aldrich 2007) 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 can be used 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 (Wadler 1990).
- 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 (Section 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.

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

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

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

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

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

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

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 the 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.

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 abstract states in which that object resides. When designing a class, a programmer has the ability to declare abstract states for a class. He can also decide that certain predicates over the fields of an object 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. Take, for example, the following invariant:

$$\mathsf{CONNECTED} := \mathsf{unique}(\texttt{counter}, \mathsf{COUNTING}) \otimes \mathsf{unique}(\texttt{socket}, \mathsf{default})$$

It specifies that when a connection is in the CONNECTED state, its counter field must be in the COUNTING state and its socket must be in the default state.

In order to allow methods to modify the fields of an object and still modularly verify that these invariants hold, we employ a packing/unpacking methodology (Barnett et al. 2004; DeLine and Fähndrich 2004).

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 inside of a method call, that object must first be "unpacked," which allows the invariants to be temporarily broken.

Packing itself is a concept, and the act of either packing or unpacking can be done explicitly by the programmer or can be left implicit. The formal system we present in Section 3 takes the former approach. The examples presented throughout this paper, on the other hand, are written in a Java-like language without pack and unpack expression. Figures 7 and 8, which walk through a verification example, illustrate where packing and unpacking implicitly occur. Similarly, our implementation (Section 4) does not require explicit annotations and instead infers them. It is important to note that an object must be unpacked before its fields can be written to or read from. Similarly, before a method returns, the receiver object must be packed, and within the method body the receiver must be packed before method calls. The latter is a requirement that ensures the receiver will be consistent in case of re-entrant calls.

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, at the beginning of the `send` method of the Connection class, seen in Figure 4, we know that the receiver object (`this`) is CONNECTED. Af-

```
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, default) ⊗ share(this, IDLE) ⊸
        share(this, CONNECTED)
  {
    atomic: {
      this.socket = new Socket(addr);
      this.counter.startCounting();
    }
  }

  void send(String msg) :
    immutable(msg, default) ⊗ 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, default)

  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, where 1 means, "requires no permission."

ter the receiver is unpacked, we know that the counter field is in the COUNTING state, but the receiver is no longer known to be in the CONNECTED state since the invariants for that state 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 an object is packed, either to the same state or to a different state, it is at the point of packing that we are required to prove the invariant of that state.

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 reference to 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 works by discarding state information for each reference that passes through code that may not be executing atomically and whose permission indicates the referred object might be modified by another thread. For objects referenced by local variables, our analysis discards state information for references of pure and share permission. For objects referenced by object fields, there are additional concerns.

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 only accessible through local variables in the current thread's stack, and therefore could not be accessed by any other threads.
- The object is referred to by a 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.

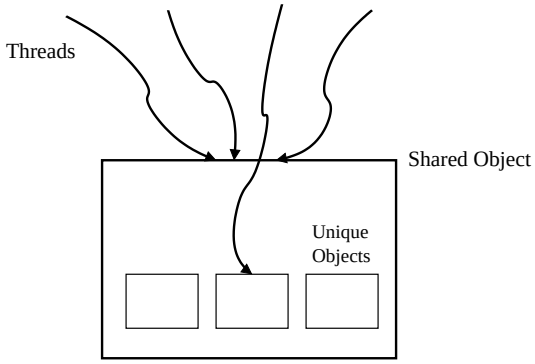


Figure 6. Unique and full fields within a thread-shared object have necessarily been unpacked within a transaction. The single thread inside is free to modify at will.

Finally, we require that all static member variables are read or written to inside of atomic blocks. Our formal system (Section 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).
- 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 (Moore and Grossman 2008). 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.

At the beginning of the method, we have a unique permission to the receiver, and this receiver is in the default 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. The unpacked predicate is technical device that is used to ensure that a.) objects are packed before method calls and method returns, and that b.) a given object cannot be unpacked twice before it is packed, which could have the effect of duplicating permissions. Here, unpacking also gives us a share permission to the `myConnection` field, which is in some unknown state ("?"). 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 discards all known state information about pure and share permissions. Intuitively, this process simulates the possible interleavings of other threads executing at this point in the program. 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

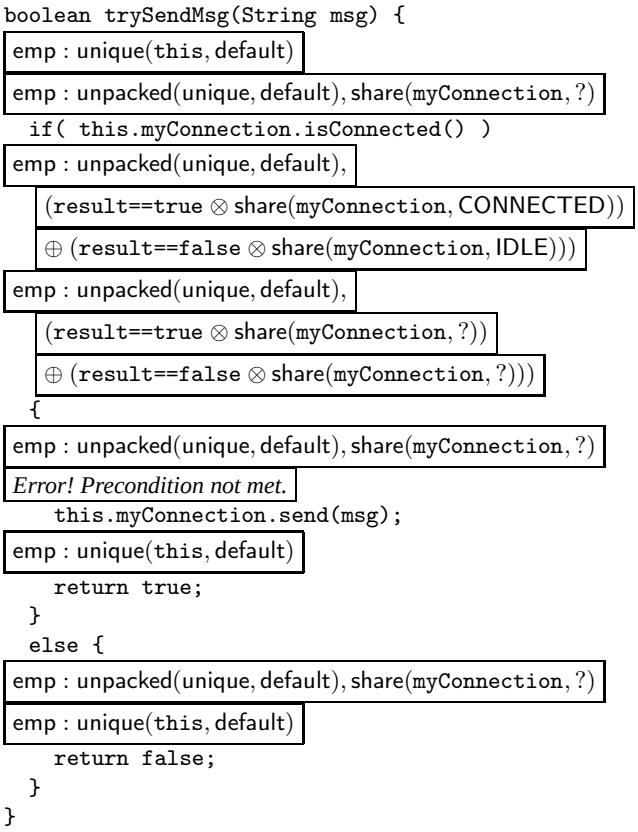


Figure 7. Verification of the `trySendMsg` method of the GUI class from Figure 1. Immediately after the conditional expression, two versions of the context are shown in order to illustrate the effect of 'forgetting.'

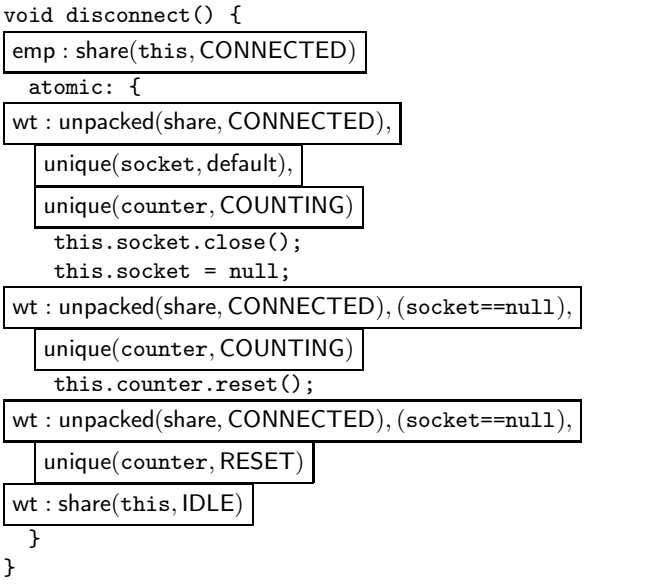


Figure 8. Verification of the *corrected* `disconnect` method.

can reduce the predicate describing `myConnection`. Unfortunately, because we discarded knowledge of the abstract state of the `myConnection` field, the pre-condition of the `send` method cannot be fulfilled, and an 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 inside the atomic block. 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. The rules of our system allow state information for all permissions to flow from pre-conditions into the first expression of a method body, and from the last expression of a method body out to the post-condition. If this first expression is inside an atomic block, then no state information is discarded for any permission type. This works because, at the calling context for a method, if we were able to establish the pre-condition for a share or pure reference, this implies that either it was established inside of an atomic block, or split from a stronger permission (unique, full or immutable) that did not need to be inside of an atomic block anyway. This feature allows methods to be used in a larger number of permission contexts. This point is discussed in more detail when the P-METH rule is discussed in Section 3.

Inside the atomic block, we check under 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 counter that is COUNTING, the method will return a unique pointer to a counter that is RESET." Finally we have enough facts to pack the receiver to the IDLE state, which satisfies the post-condition.

Both Figure 7 and Figure 8 elide certain details. In order to ensure that re-entrant method calls see objects in consistent states, we are required to pack before method calls when object re-entrancy is possible. 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 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 runtime. In this section we will present this formal language. The syntax of this language is given in Figure 9.

program	PG	$::=$	$\langle \overline{CL}, e \rangle$
class decls.	CL	$::=$	$\texttt{class}\ C\ \{\ \overline{F}\ I\ \overline{N}\ \overline{M}\}$
field decls.	F	$::=$	$f : T$
methods	M	$::=$	$T\ m(\overline{T\ x}) : MS = e$
terms	t	$::=$	$x \mid o$
		$\mid$	$\texttt{true} \mid \texttt{false} \mid t_1\ \texttt{or}\ t_2$
		$\mid$	$t_1\ \texttt{and}\ t_2 \mid \texttt{not}\ t$
expressions	e	$::=$	$t \mid t.f \mid f := t$
		$\mid$	$\texttt{new}\ C(\overline{t}) \mid t_o.m(\overline{t})$
		$\mid$	$\texttt{if}(t, e_1, e_2)$
		$\mid$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$
		$\mid$	$\texttt{spawn}\ (t_o.m(\overline{t})) \mid \texttt{atomic}\ e$
		$\mid$	$\texttt{unpack}(k, S)\ \texttt{in}\ e$
		$\mid$	$\texttt{pack to}(S)\ \texttt{in}\ e$
values	v	$::=$	$o \mid \texttt{true} \mid \texttt{false}$
references	r	$::=$	$x \mid o \mid o.f$
types	T	$::=$	$C \mid \texttt{bool}$
permissions	p	$::=$	$k(r, S) \mid \mathsf{unpacked}(k, S)$
states	S	$::=$	$s \mid ?$
facts	q	$::=$	$t = \texttt{true} \mid t = \texttt{false}$
predicates	P	$::=$	$p \mid q \mid P_1 \otimes P_2 \mid P_1 \oplus P_2$
		$\mid$	$1 \mid 0 \mid \top$
method specs	MS	$::=$	$P \multimap E$
expr types	E	$::=$	$\exists x : T.P$
state inv.	N	$::=$	$s = P$
initial state	I	$::=$	$\texttt{initially}\ \langle s \rangle$
	k	$::=$	$\mathsf{full} \mid \mathsf{pure} \mid \mathsf{share}$
		$\mid$	$\mathsf{immutable} \mid \mathsf{unique}$
atomic	$\mathcal{E}$	$::=$	$\mathsf{wt} \mid \mathsf{ot} \mid \mathsf{emp}$
valid contexts	Γ	$::=$	$\cdot \mid \Gamma, x : T \mid \Gamma, q$
linear contexts	Δ	$::=$	$\cdot \mid \Delta, P$

classes	C	*fields*	f	*variables*	x, y, z
objects	o	*methods*	m	*states*	s

Figure 9. Language and Permission syntax.

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 (2007), 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. Our formalization is influenced by Boyland (2003) and Zhao's (2007) work on fractional permissions but we give fractions a different semantics (full, pure, and share are not part of their work).

Much of the formalism regarding transactional memory, threads and their operational semantics was adapted from Moore and Grossman (2008). 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: $\Gamma; \Delta; \mathcal{E} \vdash^C e : \exists x : T.P$. The rules defining the judgment are the first twelve rules in Figure 11. 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 statically to be within an atomic block, $\mathcal{E}$, 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, while normally a permission is associated with a reference, there are times when our system tracks the permissions of an object to which no reference points. For instance, after the first subexpression of a let binding is evaluated, the result (if of a class type) is an object, and before it is bound to a variable, the available permission to this object must be tracked. Similarly, after a field has been reassigned, the permission to the object to which it previously referred still exists and can be reassigned to another reference. In rule P-ASSIGN, one can see this process occurring in the resulting permission $[f_i/x]P$, where the field to which object is assigned, f_i, is being substituted in for the bound variable x. Thus, giving expressions existential types allows us to keep consistent object permissions and the references that point to those objects.

The last six rules, beginning with P-METH, describe general well-formedness rules, rather than the expression typing judgment.

We use a decidable fragment of linear logic, the multiplicative additive fragment (MALL), as our language of behavioral specification (Lincoln and Scedrov 1994). 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 also given in Figure 9.

The declarative nature of the linear logic judgment can make for typing rules that appear to come up with permissions from almost no information. See, for example, the $\Gamma; \Delta \vdash_{\mathcal{E}} P$ premise of 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 works more 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 the type system are the judgments shown in Figure 10. Rather than dispatch directly to the linear logic proof-judgment, the typing rules first dispatch to the "atomic-aware" version of this judgment, $\Gamma; \Delta \vdash_{\mathcal{E}} P$, which is distinguished by the $\mathcal{E}$ subscript. It is the job of this judgment to ensure that predicates that must be proven do not depend on permissions of share or pure type being in a known abstract state, unless it is known statically to be within 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. The forget judgment, whose action is also predicated upon $\mathcal{E}$, accomplishes this deliberate loss of information. For example, in the typing rule for a method call, P-CALL (Figure 11), we sometimes must forget state information for potentially thread-shared permissions in the post-condition of a method's contract. It is acceptable for a method's post-condition to include share and pure permissions since that method could be called within an atomic block, but if that is not the case, these permissions must not be relied upon.

The typing rules themselves are given in Figure 11. 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 $\mathsf{forget}_{\mathcal{E}}$ 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 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 (2008). 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-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

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

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

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

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

$$\frac{\mathcal{E} = \mathsf{ot}|\mathsf{emp} \quad \Gamma; \Delta \vdash P \quad (k(r,S) \in \Delta) \supset (S = ?) \text{ where } k = \mathsf{pure}|\mathsf{share}}{\Gamma; \Delta \vdash_{\mathcal{E}} P}$$

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

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

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

Figure 10. Forgetting and atomic-aware linear judgment

$$\frac{\Gamma;\Delta;\mathsf{wt}\vdash e:\exists x:T.P \quad \mathsf{forget}_{\mathcal{E}}(P)=P'}{\Gamma;\Delta;\mathcal{E}\vdash \mathtt{atomic}\ (e):\exists x:T.P'}\ \text{P-ATOMIC}$$

$$\frac{\Gamma;\Delta;\mathcal{E}\vdash e_1;\exists x:T.P \quad \Gamma;\Delta',P\vdash_{\mathcal{E}} P' \quad (\Gamma,x:T);P';\mathcal{E}\vdash e_2:E}{\Gamma;(\Delta,\Delta');\mathcal{E}\vdash \mathtt{let}\ x=e_1\ \mathtt{in}\ e_2:E}\ \text{P-LET}$$

$$\frac{\Gamma\vdash t_o:C_o \quad \Gamma\vdash \overline{t:T} \quad \Gamma;\Delta\vdash_{\mathcal{E}}[t_o/\mathit{this}][\overline{t}/\overline{x}]P \quad \mathsf{mtype}(m,C_o)=\forall\overline{x:T}.P\multimap\exists \mathit{result}:T.P_r \quad \mathsf{unpacked}(k',S')\notin\Delta \quad \mathsf{forget}_{\mathcal{E}}(P_r)=P_r'}{\Gamma;\Delta;\mathcal{E}\vdash t_o.m(\overline{t}):\exists \mathit{result}:T.[t_o/\mathit{this}][\overline{t}/\overline{x}]P_r'}\ \text{P-CALL}$$

$$\frac{\Gamma\vdash t_o:C_o \quad \Gamma\vdash \overline{t:T} \quad \Gamma;\Delta\vdash_{\mathsf{ot}}[t_o/\mathit{this}][\overline{t}/\overline{x}]P \quad \mathsf{mtype}(m,C_o)=\forall\overline{x:T}.P\multimap E \quad \mathsf{unpacked}(k',S')\notin\Delta}{\Gamma;\Delta;\mathsf{ot}\vdash \mathtt{spawn}\ (t_o.m(\overline{t})):\exists\_:\mathtt{bool}.1}\ \text{P-SPAWN}$$

$$\frac{\Gamma;\Delta\vdash^{C}_{\mathsf{wt}} k(\mathit{this},S) \quad \mathsf{unpacked}(k',S')\notin(\Delta,\Delta') \quad \Gamma;(\Delta',\mathsf{inv}_C(S,k),\mathsf{unpacked}(k,S));\mathsf{wt}\vdash^{C} e:E}{\Gamma;(\Delta,\Delta');\mathsf{wt}\vdash^{C}\mathtt{unpack}(k,S)\ \mathtt{in}\ e:E}\ \text{P-UNPACK-WT}$$

$$\frac{\mathcal{E}\neq\mathsf{wt} \quad k=\mathsf{immutable}|\mathsf{unique} \quad \Gamma;\Delta\vdash^{C}_{\mathcal{E}} k(\mathit{this},S) \quad \mathsf{unpacked}(k',S')\notin(\Delta,\Delta') \quad \Gamma;(\Delta',\mathsf{inv}_C(S,k),\mathsf{unpacked}(k,S));\mathcal{E}\vdash^{C} e:E}{\Gamma;(\Delta,\Delta');\mathcal{E}\vdash^{C}\mathtt{unpack}(k,S)\ \mathtt{in}\ e:E}\ \text{P-UNPACK}$$

$$\frac{\Gamma;\Delta\vdash_{\mathcal{E}}\mathsf{inv}_C(S,k)\otimes\mathsf{unpacked}(k,S') \quad \Gamma;(\Delta',k(\mathit{this},S''));\mathcal{E};\vdash^{C} e:E \quad \mathsf{forget}_{\mathcal{E}}(k(\mathit{this},S))=k(\mathit{this},S'') \quad \mathsf{readonly}(k)\ \mathit{implies}\ S'=S \quad \mathit{no\ fields\ in}\ \Delta'}{\Gamma;(\Delta,\Delta');\mathcal{E}\vdash^{C}\mathtt{pack\ to}\ S\ \mathtt{in}\ e:E}\ \text{P-PACK}$$

$$\frac{\Gamma;\Delta;\mathcal{E}\vdash t:\exists x:T_i.P \quad \Gamma;\Delta'\vdash^{C}_{\mathcal{E}}[f_i/x']P'\otimes p \quad \mathsf{localFields}(C)=\overline{f:T} \quad p=\mathsf{unpacked}(k,s) \quad \mathsf{writes}(k)}{\Gamma;(\Delta,\Delta');\mathcal{E}\vdash^{C} f_i:=t:\exists x':T_i.P'\otimes[f_i/x]P\otimes p}\ \text{P-ASSIGN}$$

$$\frac{\Gamma\vdash\overline{t:T} \quad \mathsf{init}(C)=\langle\exists\overline{f:T}.P,s\rangle \quad \Gamma;\Delta\vdash_{\mathcal{E}}[\overline{t/f}]P}{\Gamma;\Delta;\mathcal{E}\vdash\mathtt{new}\ C(\overline{t}):\exists x:C.\mathsf{unique}(x,s)}\ \text{P-NEW}$$

$$\frac{\Gamma\vdash t:T \quad \Gamma;\Delta\vdash_{\mathcal{E}}P}{\Gamma;\Delta;\mathcal{E}\vdash t:\exists x:T.[x/t]P}\ \text{P-TERM}$$

$$\frac{\Gamma\vdash t:\mathtt{bool} \quad (\Gamma,t=\mathtt{true});\Delta;\mathcal{E}\vdash\exists x:T.P_1 \quad (\Gamma,t=\mathtt{false});\Delta;\mathcal{E}\vdash\exists x:T.P_2}{\Gamma;\Delta;\mathcal{E}\vdash\mathtt{if}(t,e_1,e_2):\exists x:T.P_1\oplus P_2}\ \text{P-IF}$$

$$\frac{\mathsf{localFields}(C)=\overline{f:T} \quad \Gamma;\Delta\vdash_{\mathcal{E}}P}{\Gamma;\Delta;\mathcal{E}\vdash^{C} f_i:\exists x:T_i[x/f_i]P}\ \text{P-FIELD}$$

$$\frac{(\overline{x:T},\mathtt{this}:C);P;\mathsf{emp}\vdash^{C} e:E' \quad (\overline{x:T},\mathtt{this}:C);P;\mathsf{wt}\vdash^{C} e:\exists \mathit{result}:T_r.P_r\otimes\top \quad E=\exists \mathit{result}:T_r.P_r \quad E=\mathsf{forget}_{\mathsf{emp}}(E')}{T_r\ m(\overline{Tx}):P\multimap E=e\ \mathsf{ok\ in}\ C}\ \text{P-METH}$$

$$\frac{\overline{CL}\ \mathsf{ok} \quad \cdot;\cdot;\mathsf{ot}\vdash e:E}{\langle\overline{CL},e\rangle:E}\ \text{P-PROG} \qquad \frac{\overline{F}\ \mathsf{ok\ in}\ C\ldots\overline{M}\ \mathsf{ok\ in}\ C}{\mathtt{class}\ C\ \{\ \overline{F}\ I\ \overline{N}\ \overline{M}\}\ \mathsf{ok}}\ \text{P-CLASS} \qquad \frac{f_i\ \mathit{unique} \quad T_i\in\overline{CL}\cup\{\mathtt{bool}\}}{\overline{f:T}\ \mathsf{ok\ in}\ C}\ \text{P-FDECL}$$

$$\frac{\mathtt{class}\ C\{\ldots s=P\ldots\}\in\overline{CL}}{\mathtt{initially}\langle s\rangle\ \mathsf{ok\ in}\ C}\ \text{P-CTR} \qquad \frac{s_i\ \mathit{unique} \quad r\in P_i\supset r\in\overline{F}\in C \quad r(k,S)\in P_i\ \mathit{where}\ k=\mathsf{share}|\mathsf{pure}\supset S=?}{\overline{s=P}\ \mathsf{ok\ in}\ C}\ \text{P-SINV}$$

Figure 11. Typing Rules. Helper judgments (localFields, init, mtype, inv, and writes) defined in Figure 12.

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, $\Gamma;\Delta\vdash_{\mathsf{wt}} k(\mathit{this},S)$. Since we divided the linear context into two, this will also prevent the sub-expression from relying on this fact, as the invariant for state S may not hold. 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 12, has two roles. It will look up that 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 12. The sub-expression is also given unpacked(k, S), which signifies that the receiver is temporarily unpacked.

- P-UNPACK: This rule is similar to P-UNPACK-WT, but occurs when not inside a transaction. We are limited to unpacking unique and immutable permissions.
- P-PACK: In order to pack, we treat the linear context as if it has been split in two. With the first part, Δ, we must be able to prove all of the invariants of the state S of class C to which the programmer wants to pack. These invariant permissions are retrieved with the inv_C function. In our small calculus, only the object receiver of a method call can be packed and unpacked, so there is no need to specify which object is to be packed. We must also be able to show that the receiver has already been unpacked by producing the unpacked predicate. Then, we combine the remainder of the linear context, Δ', and the information that `this` has been packed to state S'' to prove that the subexpression e has type E. S'' is S passed through the forget function. If k, the permission with which the reference was unpacked, is a read-only permission, then the state from which the object was unpacked S' must match S: A read-only permission should not be used to change the abstract state of an object. Finally, the requirement that there are no fields in Δ' ensures that fields can only be read when their object is unpacked.
- 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-METH: Method bodies are actually type-checked twice. Because we do not 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-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 it 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.

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

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

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

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

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

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

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

Figure 12. Helper judgments. Note that the dg' function is a helper function for dg that operates directly on permission kinds.

Dynamic semantics for our language are given in the accompanying technical report (Beckman and Aldrich 2008). These rules are extremely similar to those of the Weak language (Moore and Grossman 2008). 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 prove 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 (Bierhoff and Aldrich 2008) to check the rules described in this paper in Java language programs. This checker is a modular, branch-sensitive data-flow analysis 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 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, as follows:

```
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 (Hindman and Grossman 2006), 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 the Bierhoff and Aldrich (2007) system that were not discussed in this work. For instance, it supports fractional permissions which allow multiple share permissions to be joined together to reconstruct a unique one. 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 4 and 5, we have used our implementation to verify several other examples[2].

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[3] is an open-source library for use by developers of multicast 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 provided a finite state machine (FSM) based specification in the 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 13. 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.

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.

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. (2005) have

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

[3] www.jgroups.org

```
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, default) ⊸
        unique(this, RUNNING)
  {
    this.requestPipe.send(str);
    return;
  }

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

Figure 13. RequestProcessor, an example of a server-like program where class invariants depend on thread-shared objects.

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 (Barnett et al. 2004). 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. 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 abstraction of object state and of effects on that object. 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 13 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 (Rodriguez et al. 2005). 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.

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 (1976), Concurrent Separation Logic (O'Hearn 2007), and Rely-Guarantee Logic (Jones 1983). All three allow you 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. This characteristic is quite similar to unpacking of state invariants in our system which, for references of full, share, and pure permission, must be performed inside of an atomic block. Concurrent Separation Logic furthermore allows one to reason modularly about heap memory that cannot be thread-shared, and does so in a manner that is similar to our unique permission. Overall it lacks the flexibility of our permissions, which allow a larger variety of thread-sharing patterns.

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. However, the Rely-Guarantee 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. All three logics are pen and paper-based techniques and are not, as described in these works, automated analyses.

Calvin-R (Freund and Qadeer 2003) 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 (1975) 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.

In recent work, Vaziri et al. (2006) 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 (2006) 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.

5.2 Race Detection.

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

Dynamic race detectors (Savage et al. 1997; Yu et al. 2005) 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 (Henzinger et al. 2004; Stoller 2000). 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 (Boyapati et al. 2002; Greenhouse and Scherlis 2002; Grossman 2003; Pratikakis et al. 2006; Engler and Ashcraft 2003) 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 (Flanagan and Qadeer 2003; Sasturkar et al. 2005; Hicks et al. 2006) 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 (Lipton 1975). 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.

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 (2007) 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. We believe these features are orthogonal and can be added without difficulty.

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 TM for the purposes of static verification. Currently, most existing flow analyses and verification tools are unsound in the face of concurrency, and those that are not impose a great an-

notation 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 (2007), we would say the following: In the same way garbage collection allows proofs of program properties that would be difficult or impossible in a language with explicit memory allocation and reclamation, transactional memory will allow proofs of program properties for multi-threaded languages, when doing the same with lock-based synchronization would be difficult or impossible. The performance of TM implementations continues to improve (Adl-Tabatabai et al. 2006), 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 an intraprocedural static analysis, formalized as a type system, 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 (Beckman and Aldrich 2008). 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 also like to acknowledge the PLAID group, Todd Millstein and John Boyland for their helpful comments. Additionally, we are very grateful for the detailed feedback we received from the anonymous reviewers.

The authors would like to acknowledge the sponsors who helped fund this work. This work was supported by a University of Coimbra Joint Research Collaboration Initiative, DARPA grant #HR00110710019, Army Research Office grant #DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems", the Department of Defense, and the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation. The first author is supported by a National Science Foundation Graduate Research Fellowship (#DGE0234630).

References

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 *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37. ACM Press, 2006.

Cyrille Artho, Klaus Havelund, and Armin Biere. High level data races. In *VVEIS '03: Proceedings of the Workshop on Verification and Validation of Enterprise Information Systems*, pages 82–93, April 2003.

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.

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`.

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 301–320. ACM Press, 2007.

Kevin Bierhoff and Jonathan Aldrich. Plural: Checking protocol compliance under aliasing. In *Companion Proceedings of ICSE-30*, pages 971–972. ACM Press, May 2008.

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.

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.

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

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.

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.

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

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

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.

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.

Dan Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 695–706. ACM Press, 2007.

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.

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.

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.

Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91. ACM Press, 2006.

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.

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

Patrick Lincoln and Andre Scedrov. First-order linear logic without modalities is NEXPTIME-hard. *Theor. Comput. Sci.*, 135(1): 139–153, 1994.

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

Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 51–62. ACM Press, 2008.

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

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

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.

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.

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.

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.

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.

Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

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.

Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

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.

Yang Zhao. *Checking Interference with Fractional Permissions*. PhD thesis, University of Wisconsin-Milwaukee, August 2007.