Verifying Correct Usage of Atomic Blocks Using Access Permissions

Nels E. Beckman
Carnegie Mellon University
nbeckman@cs.cmu.edu

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification

1. Introduction

While the number of cores in the CPUs of modern computers has been steadily increasing, improvements in the way we program concurrent applications have proceeded at a slower pace. One idea that seems to have gained a fair amount of traction is transactional memory (TM). Transactional memory is a means of simplifying mutual exclusion in shared-memory applications. While it is a complex topic, at its core, transactional memory provides a simple language primitive to programmers, the atomic block. Code executing inside of the atomic block will execute as if no other threads were executing at the same time.

```
class Connection {
  boolean isConnected() {
    atomic: {return this.socket!=null};
  }}
  ...
}
boolean trySendMsg(String msg, Connection c)
{
  if(c.isConnected()) {
    c.send(msg);
    return true;
  }
  return false;
}
```

Figure 1. An example where a race condition could occur at the level of program logic.

But as the code shown in Figure 1 illustrates, while transactional memory makes programming concurrent applications easier, it remains a difficult and error-prone task. This program is a short excerpt from a multi-threaded, networked application where many threads hold reference to a single `Connection` object, an abstraction of a host-to-host connection. Each thread may send messages, and if necessary, disconnect the connection. The important thing to note is that even if a programmer consistently accesses all thread-shared memory inside of an atomic block, race conditions can still occur. Here, in between the time when one thread checks that the connection is open and sends a message across that connection, another thread could disconnect the connection, setting the `socket` field of Connection to null, resulting in an eventual null-pointer dereference.

2. Proposed Thesis Plan

For my thesis, I plan to develop an analysis based on access permissions (Bierhoff and Aldrich 2007) to solve this sort of race condition.

Thesis Statement The goal of this thesis is to show that access permissions, which statically describe the aliasing behavior of program references in object-oriented programs, provide a good basis for the verification of the implementation and usage of object protocols in concurrent systems.

2.1 Approach

In work that already begun (Beckman et al. 2008), we discovered that by recasting access permissions (Bierhoff and Aldrich 2007), which were originally developed as a means of statically controlling aliasing patterns, we can soundly approximate thread-sharing in a concurrent context. This in turn allowed us to to verify object protocol implementation and usage in multi-threaded code. This verification helps developers ensure the absence of "application-level" race conditions; race conditions that exist at the level of program logic. This work uses the atomic block, a primitive provided by transactional memory systems, as its mutual exclusion primitive. We have developed a type system, proven sound, that formalizes our verification methodology. We have also developed a modular, branch sensitive data-flow analysis for the static verification of Java source code based on the formal rules of this system.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

While a full description of access permissions is outside the scope of this abstract, it is quite helpful to understand the five basic permissions.

Unique A unique reference points to an object that can only be read and modified by a single thread.

Full A full to a reference allows one thread to modify an object that several others can concurrently read.

Immutable An immutable reference points to an object that can be read by many threads but modified by none.

Share A share reference points to an object that can be concurrently read and modified by any thread in the system.

Pure A pure reference is a reference to an object that can be read by the given thread, but could be modified concurrently by other threads.

Software annotated with these permissions can be checked for consistency, at which point object protocol specifications can be modularly verified. Object protocols are abstract descriptions of the states that an object can be in, and determine the methods that can legally be called when objects are in a given state. The idea of statically checking object protocol conformance is known as “type-state” checking. As an example, consider the following type-state specification for the `send` and `isConnected` methods of Figure 1:

```
@Pure
@TrueIndicates("connected")
@FalseIndicates("disconnected")
boolean isConnected() { ... }

@Share(requires="connected",
       ensures="connected")
void send(String str) { ... }
```

The first specification states that, in order call the `isConnected` method, the caller must at least have a pure (non-modifying) permission to the receiver, but that object can be in any state. Depending on whether the method returns true or false, at the return the receiver will be known to be in either the connected or disconnected state. The second specification states that, in order to call the `send` method, the caller must have a share (modifying) permission to the receiver and that object must be known to be in the connected state. Putting it all together, the analysis itself works by tracking these states but discarding any known state information for objects whose permissions indicate they could potentially be modified by other threads (that is, pure and share permissions). At the moment, I have formalized and proven sound this analysis, and we are in the early stages of implementing a prototype checker.

2.2 Remaining Work and Proposed Timeline

In between my proposal date (Fall, 2008) and my ideal date of defense (Spring 2010), there are several major tasks that need to be accomplished. I need to complete a robust implementation of a checker for the system (3 months). Currently an implementation exists that works on smaller examples, but needs to be improved for the purposes of larger case studies. Turning the system of typing rules into a data-flow analysis is a non-trivial problem, and some work remains to be done in order to support the full specifications we describe. I plan to then evaluate this tool on six to eight minor case studies (3 months). The experience that I gain verifying these smaller programs I will then use to improve the design of the analysis and implementation (3 months). At the end of this phase, I will begin a series of two to four larger case studies (6 months) to fully evaluate the precision and overhead of our approach. Finally, I will write my thesis document and defend (4 months). In total, I plan on spending 19 months from proposal to defense.

3. Benefit of Doctoral Symposium

While I believe that my work thus far is a strong foundation for a thesis, I also believe that I am at somewhat of a crossroads regarding the structure of the rest of my thesis. I strongly believe that participation in the OOPSLA Doctoral Symposium will help me settle on a direction. The essence of my dilemma is the following: Should I spend significant effort evaluating my existing work on real programs or should I continue to supplement my analysis with features that could help demonstrate the important significance of software transactional memory to object-oriented verification?

In favor of the former approach is the importance of verifying the practicability of a theoretical approach. Given the questions I have already received from peers, it is not clear that the analysis as it stands is precise enough to verify real programs, nor that the sorts of behavioral properties it verifies (type-state) are all that common in concurrent programs. A large empirical study would help answer these questions as well as suggest practically motivated improvements. On the other hand, I believe there are specific technical means by which I can help to improve the quality of code programmers write by exploiting the interplay between verification and TM. For example, our analysis implies certain optimizations that could be used to reduce the overhead of TM implementations. Also, by extending our analysis, I believe that we can alleviate the loss of composability that normally comes with the introduction of blocking primitives into TM systems. A thesis that fully explored these and other features inspired by this interplay would be extremely interesting, but most likely less well-validated. I believe that by participating in OOPSLA’s Doctoral Symposium, I will receive the guidance necessary to choose a path forward.

References

Nels Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA*, 2008.

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.