Assertional Reasoning about Data Races in Relaxed Memory Models

Beverly A. Sanders KyungHee Kim
University of Florida
{sanders,khkim}@cise.ufl.edu

Abstract

We describe the ideas behind a method to use assertional reasoning to statically show that all sequentially consistent executions of a concurrent program are free from data races.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness Proofs, Formal Methods; D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Languages, Theory, Verification

Keywords Memory Model, Sequential Consistency, Assertions, Data Race

1. Introduction

While virtually all approaches for reasoning about concurrent programs, both formal and informal, assume sequential consistency (SC) modern programming environments do not satisfy this assumption. Optimizations in both hardware and software (e.g. reordering independent operations, buffering writes) that can significantly speed up sequential computations while preserving their semantics are not necessarily benign in concurrent programs. The programmer is expected to insert sufficient synchronization to prevent data races and other undesirable nondeterministic behavior. What the synchronization actions are, how they constrain the program execution, and precisely what constitutes a data race depend on the particular environment and are determined by a memory model. Generally, memory models should satisfy the following Fundamental Property [SJMvP07]: "Programs whose SC executions have no races must have only SC executions". This means that we can soundly reason about concurrent programs using traditional methods which assume SC if we also prove that the program is free of data races[1].

In this paper, we briefly describe preliminary work on an assertional approach to proving that a concurrent program contains no data races. A more complete description can be found in [SK07]. We start with a language similar to Java byte code[BM05] and modify it to handle multithreading and more specifically to allow reasoning about data races. Then, we define weakest precondition semantics which can be used to construct *assertional proofs* of data-race freedom. The longer term goal is to provide tool support.

We consider memory models defined in terms of a happened-before relation between certain events in a program. The happened-before relation, denoted $\preceq$, is a transitive, irreflexive partial order on operations such that if $o_i \preceq o_j$, then the state of memory seen by o_j reflects the execution of o_i. In the Java Memory Model[MPA05], for example, there are happened-before edges between events occurring on a single thread according to the order indicated by the program text and between various pairs of actions occurring on different threads, such as:[2] releasing a particular monitor lock $\preceq$ a subsequent acquire of that lock; a write to a volatile variable $\preceq$ a subsequent read of that variable; and the action of starting a thread $\preceq$ the first action of the newly started thread. Two operations performed by different threads *conflict* if they access the same memory location and at least one is a write. A *data race* is a pair of conflicting operations *not* ordered by $\preceq$.

Reasoning about happened-before edges is difficult, and both programmers and static race detection tools typically try to ensure simpler conditions that imply data-race freedom rather than reasoning about data races using first principles based on the memory model. For example, if all accesses to a particular shared variable are guarded by the same lock[3] then that variable will not be involved in data races. However, even these simpler rules are not trivial to implement correctly and are not always applicable since there are important concurrent programming idioms that do not use locking in this way.

2. Assertional reasoning about data race freedom

We first describe a model of an object store that, for the purposes of this brief presentation, is a partial function from locations to values. In addition to the store, we maintain a happened-before map, which is a map from locations to sets of locations. For a thread represented by location th, and happened-before map hb, $hb(th)$ is the set of locations that the thread can access without causing a data race. The map is modified by *acquire*, *release*, and *invalidate* operations which are used to define the memory model related semantics of the programming language constructs. The following types are used: $thread : location$, $store : location \rightarrow values$, and $hbmap : location \rightarrow$ set of $location$.

[1] The term "race" is sometimes used to mean any kind of undesirable nondeterminism in a concurrent program. In this paper, we use the term "data race" in the strict sense to mean a situation that, according to the memory model, may cause the program to violate SC.

[2] See [MPA05] for a complete list.

[3] In Java, it is also required that there is a happened-before edge between the non-default initialization of a field in a constructor and the first access of the field by another thread. This is usually ensured by safe publication of the containing object rather than locking within the constructor.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
ACM 978-1-59593-960-9/08/0002.

- $read : location \rightarrow store \rightarrow value$
  Returns the value of the location in the given store.
- $write : location \rightarrow value \rightarrow store \rightarrow store$
  Updates the value stored at the location and returns a new store reflecting the update.
- $invalidate : thread \rightarrow location \rightarrow hbmap \rightarrow hbmap$
  $invalidate\ th\ v\ hb\ l\ \equiv$
  if $(l = th \vee l = v)$ then $hb(l)$ else $hb(l) \backslash \{v\}$.
- $acquire : thread \rightarrow location \rightarrow hbmap \rightarrow hbmap$
  $acquire\ th\ v\ hb\ l\ \equiv$
  if $(l = th)$ then $hb(l) \cup hb(v)$ else $hb(l)$.
- $release : thread \rightarrow location \rightarrow hbmap \rightarrow hbmap$
  $release\ th\ v\ hb\ l\ \equiv$
  if $(l = v)$ then $hb(l) \cup hb(th)$ else $hb(l)$.
- $new : thread \rightarrow classID \rightarrow (store * hbmap) \rightarrow ((location, classID) * (store * hbmap)$
  Allocates a new object of the given class type and returns its location. The happened-before map of each new location contains itself. The executing thread acquires all of the newly allocated locations.
- $norace : thread \rightarrow location \rightarrow boolean$
  $norace\ th\ v \equiv v \in hb(th)$
  If $norace\ th\ v$ holds, thread th can access v without causing a data race.

A configuration contains an object store, a happened-before map, and a local state map that maps threads to their local states. The local state of a thread is a stack of frames, one for each method invocation. Each frame contains a local variable array, an expression stack, and a program counter. For brevity, we informally describe the semantics only of the operations with the most significant interaction with the memory model. In the following, $\$th$, $\$sto$, and $\$hb$ refer to the current thread, store, and happened-before map, respectively. We omit descriptions of changes to the program counter and stack.

- *GetField* takes a class name and field name as parameters, and obtains the target object's location from the stack. From these, the location of v, the field to be read, can be obtained. If v corresponds to a volatile field, the interaction with the store can be modeled as $acquire\ \$th\ v\ \$hb$; $read\ \$sto\ v$. If the v is not volatile, then the operation is modeled by $assert\ norace(\$th, v)$; $read\ \$sto\ v$.
- *PutField* takes a class name and field name as parameters and obtains the target object's location and the value to be written from the stack. If the location to be written is volatile, the interaction with the store can be modeled as $write\ \$sto\ v\ val$; $release\ \$th\ v\ \$hb$. If v is not volatile, then the interaction is modeled by $assert\ norace(\$th, v)$; $write\ \$sto\ v\ val$; $invalidate\ \$th\ v\ \$hb$.
- *Starting a new thread* by calling its start method causes a new frame to be created, the new thread's state to be changed to runnable, and the local state map to be updated. The memory effects are modeled by $release\ \$th\ th_n$, where th_n is the new thread.

Weakest precondition semantics for individual operations are fairly straightforward extensions of those given in [BM05] with the happened-before map included as a ghost variable. In addition, a standard non-interference proof [OG76] is needed. $(norace\ \$th\ v)$ is part of the precondition of non-volatile PutField or GetField operations since $\{P \wedge Q\} assert\ P \{Q\}$. If the proofs of individual threads and the non-interference proofs go through, the program is data-race free.

```
class X
{ int w; int z; volatile boolean b;

   void main()
   { X x = new X; x.start(); x.w = 1; x.b = true; }

   void run(){if (b) z = w; }
}
```

Figure 1. Two thread example: Thread 0 executes main, Thread 1 executes run

Thread 0	Thread 1
New "X"	Pushv 0
Pop 1	$\{ b \Rightarrow w \in hb(b)\}$
Pushv 1	GetField ("X","b")
InvokeVirtual ("X","start")	$\{b \Rightarrow (norace\ th1\ w)\ \}$
Pushc 1	BranchIfTrue 4
Pushv 1	Return
$\{\neg b\}$	4: Pushv 0
PutField ("X","w")	$\{ b \wedge (norace\ th1\ w)\}$
Pushc true	GetField ("X","w")
Pushv 1	Pushv 0
PutField ("X","b")	PutField ("X","z")
$\{b \wedge w \in hb(b)\}$	$\{true\}$

Figure 2. Bytecode with assertions for the program from Fig. 1

3. Example

Source code for a simple example is shown is Fig. 1 and its translation into bytecode in Fig. 2. The interesting assertions are indicated, while the transformations to deal with hb have been omitted. Using our weakest precondition semantics, the proof outlines for each individual thread have been shown to be valid and interference-free. Thus this program is free of data races, a property that depends the fact that b is volatile. This example captures the essence of an important concurrent programming idiom (which is not handled by most static race detection tools) where a volatile variable is used to indicate that a non-volatile variable will no longer be modified, and thus can be read without synchronization.

References

[BM05] F. Y. Bannwart and P. Müller. A logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[OG76] Susan Owicki and David Gries. Verifying Properties of Parallel Programs: an Axiomatic Approach. *Commun. ACM*, 19(5):279–285, 1976.

[SJMvP07] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A Theory of Memory Models. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–172, New York, NY, USA, 2007. ACM Press.

[SK07] Beverly A. Sanders and KyungHee Kim. Assertional Reasoning about Data Races in Relaxed Memory Models. Technical report, Department of Computer and Information Science and Engineering, University of Florida, 2007.