

# Safe Fine-Grained Locking for Aggregate Objects

*Bart Jacobs      Frank Piessens      Wolfram Schulte*

*Report CW 444, April 2006*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Safe Fine-Grained Locking for Aggregate Objects

*Bart Jacobs      Frank Piessens      Wolfram Schulte*

*Report CW444, April 2006*

Department of Computer Science, K.U.Leuven

## **Abstract**

Programmers have difficulty writing correct multithreaded code, not to mention code that scales well. One way to approach this problem is by offering a transaction construct, and leaving it to the compiler and run-time system to implement efficient synchronization. However, automatically generating efficient synchronization code is an open research problem. In this paper, rather than attempting to generate synchronization code automatically, we propose a programming methodology for explicit fine-grained locking of aggregate objects. We also propose a method for run-time checking, and a method for sound modular static verification of the safety of programs written according to the methodology. The system prevents deadlocks. The system is an extension of our methodology for safe concurrency for aggregate objects with invariants, which is in turn based on the Spec#/Boogie methodology for aggregate objects with invariants. This paper is a preliminary result of our investigation into programming methodologies for safe and efficient concurrency in object-oriented languages.

# Safe Fine-Grained Locking for Aggregate Objects

Bart Jacobs<sup>1</sup> Frank Piessens<sup>1</sup> Wolfram Schulte<sup>2</sup>

<sup>1</sup>Dept. CS, K.U.Leuven, Belgium {bartj,frank}@cs.kuleuven.be  
<sup>2</sup>Microsoft Research, Redmond, USA schulte@microsoft.com

## Abstract

Programmers have difficulty writing correct multithreaded code, not to mention code that scales well. One way to approach this problem is by offering a transaction construct, and leaving it to the compiler and run-time system to implement efficient synchronization. However, automatically generating efficient synchronization code is an open research problem. In this paper, rather than attempting to generate synchronization code automatically, we propose a programming methodology for explicit fine-grained locking of aggregate objects. We also propose a method for run-time checking, and a method for sound modular static verification of the safety of programs written according to the methodology. The system prevents deadlocks. The system is an extension of our methodology for safe concurrency for aggregate objects with invariants, which is in turn based on the Spec#/Boogie methodology for aggregate objects with invariants. This paper is a preliminary result of our investigation into programming methodologies for safe and efficient concurrency in object-oriented languages.

## 1 Introduction

In earlier work [2], we proposed a programming methodology that enables run-time checking and modular static verification of the data-race-freedom and data consistency of programs that use mutual exclusion locks to protect multi-object abstractions in an object-oriented programming language such as Java or C#. The multi-object abstractions are defined using an ownership system that allows the program to manipulate the ownership graph at run time. While the methodology is simple and flexible, it limits parallelism since no two threads can access the same object structure concurrently.

In this paper, we conservatively extend this programming methodology to increase parallelism through programmer-specified fine-grained locking within a single object structure; this way, multiple threads may access the same object structure concurrently, provided they do not write to overlapping sub-structures.

We illustrate our approach with a running example (see Fig. 1). The example consists of an Account class, a Bank class, and a main program. The main program creates a Bank object and adds four Account objects. It then starts two threads. The first thread performs a transfer from account 0 to account 1; *concurrently*, the second thread performs a transfer from account 2 to account 3. The *Transfer* method first orders the accounts for deadlock prevention. It then locks them and performs the transfer. Note that this preserves the Bank object’s consistency.

The program in Fig. 1 is written according to the proposed methodology and uses the accompanying extensions to the syntax and the semantics of the programming language. The methodology provides the following benefits for this program:

- **Thread-Safety** The methodology guarantees the absence of unexpected interference between threads. No operation performed by a thread influences or is influenced by other threads, except for the **acquire.finegrained** operation, which causes a thread to gain control of an object that may previously have been manipulated by other threads. Specifically, the methodology guarantees serializability of transactions.

- **Protection of Aggregate Objects** Objects in the program may be grouped into aggregates, and invariants may be declared over these aggregates. In the example, the four Account objects form an aggregate with the Bank object; this is achieved by declaring the sequence of Account objects as being *owned* by the Bank object. This in turn allows the programmer to express the invariant that the total balance of all accounts of the bank is equal to the value of the *totalBalance* field. The composition of aggregates may change at run time.
- **Increased Parallelism** Even though the `acquire_finegrained` operation is conceptually equivalent to acquiring a mutual exclusion lock, annotations in the program enable a more efficient implementation. In the example, the `write_indirectly` annotation causes the *Transfer* method to acquire a read lock on the Bank object instead of a write lock, which allows both threads to access the Bank object in parallel. Only the Account objects involved in the transfer are locked for writing.
- **Deadlock Prevention** The program may dynamically define a partial order among the objects in an aggregate, using the special *childOrdinals* field. The methodology enforces that locks are acquired according to this partial order.
- **Modular Static Verification** The methodology includes a method for sound modular static verification of compliance of the program with the rules of the methodology.
- **Run-Time Monitoring** As an alternative to static verification, the methodology also supports instrumentation of a program with run-time monitoring code that detects violations of the methodology and aborts the program before it performs a potentially unsafe operation.

The rest of the paper is structured as follows: in Section 2, we summarize the original coarse-grained locking methodology. In Section 3, we propose a regime for fine-grained locking of aggregates that ensures data-race-freedom but not serializability. In Section 4, we gain serializability by introducing a notion of transactions. In Section 5, we add deadlock prevention, and in Section 6, we ensure genericity of modules with respect to their contexts. Finally, in Section 7, we modularize the system to arrive at the final programming methodology. We end the paper with a short discussion and a conclusion.

## 2 Mutual Exclusion for Aggregate Objects

The methodology proposed in this paper is a refinement of a methodology for coarse-grained locking of aggregate objects that we proposed in earlier work [2]. In this section, we summarize the coarse-grained methodology, using the running example of Fig. 1. The following sections build on this section to arrive at the final system.

A program in the coarse-grained methodology can be thought of as manipulating the heap using just the following seven kinds of operations: object creation, field read, field write, connect, disconnect, share, acquire, and release. All of these operations work on special per object fields. The owner field captures the ownership relationship, the shared field describes whether an object is local to a thread or shared between threads, and the writer field denotes the thread that currently holds the mutual exclusion lock on the object.

Objects created by a thread are considered local to that thread (i.e. their *owner* field points to the creating thread), until the thread explicitly shares them using a share operation. The acquire and release operations, which acquire and release the write lock of an object, may be applied only to shared objects. The connect and disconnect operations manipulate an *ownership graph* whose nodes are the objects in the heap. Through the ownership graph, the program defines (potentially nested) aggregates of objects, each consisting of a root object and its transitively owned objects.

Fig. 2 depicts the first several heap operations that occur during an execution of the program in Fig. 1. Each thread has a local heap (inner circle) containing the thread’s unshared aggregates. A thread’s *writable*

```

class Account
{ int balance; void Deposit(int amount) { write (this) { this.balance += amount; } } }

class Bank {
  [Owned] Seq<Account> accounts;
  int totalBalance;
  invariant forall{int i in (0:this.accounts.Length);
    this.childOrdinals[this.accounts[i]] == i};
  invariant this.totalBalance == Sum{Account a in this.accounts; a.balance};

  int AddNewAccount() {
    write (this) {
      Account account = new Account();
      int id = this.accounts.Length;
      this.childOrdinals[account] = id;
      this.accounts.Add(account);
      return id;
    }
  }

  void Transfer(int fromId, int toId, int amount) {
    write_indirectly (this) {
      if (toId < fromId) {
        int temp = fromId; fromId = toId; toId = temp; amount = -amount;
      }

      lockW_child(this, this.accounts[fromId]);
      lockW_child(this, this.accounts[toId]);

      this.accounts[fromId].Deposit(-amount);
      this.accounts[toId].Deposit(amount);
    }
  }
}

class Test {
  static void Main() {
    Bank bank = new Bank();
    for (int i = 0; i < 4; i++) bank.AddNewAccount();

    share(bank);

    new Thread(delegate {
      transaction { acquire_finegrained (bank) { bank.Transfer(0, 1, 1000); } }
    }).Start();
    new Thread(delegate {
      transaction { acquire_finegrained (bank) { bank.Transfer(2, 3, 2000); } }
    }).Start();
  }
}

```

Figure 1: Bank class with fine-grained locking and deadlock prevention



Per-object fields: *owner, shared, writer*  
(*shared* is a *volatile* field; i.e., accessing it has acquire-release semantics)

$$t.Roots \stackrel{\text{def}}{=} \{o \mid o.owner = t \vee o.writer = t\} \quad t.Writable \stackrel{\text{def}}{=} \{p \mid o \text{ owns }^* p \wedge o \in t.Roots\}$$

$$(o \text{ owns } p) \stackrel{\text{def}}{=} (p.owner = o) \quad t.Readable \stackrel{\text{def}}{=} t.Writable;$$

$x := \mathbf{new} C; \equiv$ $x \leftarrow \mathbf{new} C;$ $x.owner \leftarrow \mathbf{tid};$	$\mathbf{connect} p \text{ to } o; \equiv$ $\mathbf{assert} o \in \mathbf{tid}.Roots;$ $\mathbf{assert} p.owner = \mathbf{tid};$ $p.owner \leftarrow o;$	$\mathbf{share} o; \equiv$ $\mathbf{assert} o.owner = \mathbf{tid};$ $o.owner \leftarrow \mathbf{null};$ $o.shared \leftarrow \mathbf{true};$
$x := o.f; \equiv$ $\mathbf{assert} o \in \mathbf{tid}.Readable;$ $x \leftarrow o.f;$	$\mathbf{disconnect} p \text{ from } o; \equiv$ $\mathbf{assert} o \in \mathbf{tid}.Writable;$ $\mathbf{assert} p.owner = o;$ $p.owner \leftarrow \mathbf{tid};$	$\mathbf{lockW} o; \equiv$ $\mathbf{assert} o.shared;$ $\mathbf{lockW}_0 o;$ $o.writer \leftarrow \mathbf{tid};$
$o.f := x; \equiv$ $\mathbf{assert} o \in \mathbf{tid}.Writable;$ $o.f \leftarrow x;$	$\mathbf{unlockW} o; \equiv$ $\mathbf{assert} o.writer = \mathbf{tid};$ $o.writer \leftarrow \mathbf{null};$ $\mathbf{unlockW}_0 o;$	

Figure 3: The mutual exclusion regime of Section 2. The platform primitives are denoted using a subscript 0. **tid** denotes the current thread.

Per-object fields: *owner, shared, writer, readers*

$$t.Readable \stackrel{\text{def}}{=} t.Writable \cup \{o \mid t \in o.readers\} \quad CanLock(o) \stackrel{\text{def}}{=} (o.shared \vee \mathbf{tid} \in o.owner.readers)$$

$x := \mathbf{new} C; \equiv$ $x := o.f; \equiv$ $o.f := x; \equiv$	$\mathbf{lockW} o; \equiv$ $\mathbf{assert} CanLock(o);$ $\mathbf{lockW}_0 o;$ $o.writer \leftarrow \mathbf{tid};$	$\mathbf{unlockW} o; \equiv$ $\mathbf{assert} o.writer = \mathbf{tid};$ $o.writer \leftarrow \mathbf{null};$ $\mathbf{unlockW}_0 o;$
$\mathbf{connect} p \text{ to } o; \equiv$ $\mathbf{disconnect} p \text{ from } o; \equiv$ $\mathbf{share} o; \equiv$ <i>as in Fig. 3</i>	$\mathbf{lockR} o; \equiv$ $\mathbf{assert} CanLock(o);$ $\mathbf{lockR}_0 o;$ $o.readers.Add(\mathbf{tid});$	$\mathbf{unlockR} o; \equiv$ $\mathbf{assert} o \in \mathbf{tid}.Readable;$ $\mathbf{assert} (\forall p \mid o \text{ owns } p \bullet p \notin \mathbf{tid}.Readable);$ $o.readers.Remove(\mathbf{tid});$ $\mathbf{unlockR}_0 o;$

Figure 4: The fine-grained locking system

Per-object fields: *owner, shared, writer, readers*  
 Per-thread fields: *locks, txDepth*

<pre> x := new C; ≡ x := o.f; ≡ o.f := x; ≡ connect p to o; ≡ disconnect p from o; ≡ share o; ≡   as in Fig. 3 </pre>	<pre> lockW o; ≡   assert 0 &lt; tid.txDepth;   lockW<sub>(Fig. 4)</sub> o;   tid.locks.Push(o);  lockR o; ≡   assert 0 &lt; tid.txDepth;   lockR<sub>(Fig. 4)</sub> o;   tid.locks.Push(o); </pre>	<pre> transaction { S } ≡   tid.txDepth++;   S   tid.txDepth--;   if (tid.txDepth = 0)     commit;   where     commit; ≡       while (o ← tid.locks.Pop())         if (o.writer = tid)           unlockW<sub>(Fig. 4)</sub> o;         else           unlockR<sub>(Fig. 4)</sub> o; </pre>
---	---	--

Figure 5: The transaction system. Notice that this system has no separate **unlockW** or **unlockR** commands.

are in fact more like atomic blocks than like full-fledged database transactions). The system guarantees that a sequence of consecutive operations performed by a thread during a transaction will appear to be atomic to other threads. In other words, each execution that contains such a sequence is equivalent to one where the operations are not interleaved with operations performed by other threads.

The system does not allow a thread to separately release read or write locks; rather, upon acquiring a read or write lock, the lock is added to the per-thread *locks* field, which represents the set of locks of the current transaction. All of these locks are released at the end of the transaction, using the auxiliary **commit** operation.

The transaction system is shown formally in Fig. 5.

**Claim 3** *Each compliant execution is serializable.*

## 5 Deadlock Prevention

The system of the previous section does not prevent deadlocks. In this section, we add provisions that guarantee deadlock-freedom.

As is conventional, we achieve deadlock-freedom by defining a partial order on all objects in the heap, and we enforce that objects are locked in accordance with this partial order. In order to define the partial order, we exploit the tree structure imposed by the ownership system. Firstly, all shared objects, which constitute the roots of the shared aggregates, are assigned a unique id at run time (specifically, at the time the object is shared). Secondly, within a single aggregate, each object defines a partial order on its owned objects using a *childOrdinals* table that maps objects to integers. These two ingredients together allow us to map each object in the shared heap to a string of integers. The lexicographical order on these strings defines the required locking order on the objects.

Note that the system allows the partial order to be modified at run time, by updating the *childOrdinals* tables. Of course, these updates are subject to the general rule that field updates may occur only when the object is writable. It follows that a *childOrdinals* table that is being updated by a given transaction is not relevant to the deadlock-freedom of that transaction, since the transaction is not allowed to attempt to lock children of a writable object.

The system is shown formally in Fig. 6.

The methodology enforces the locking order using the per-thread *pathNodes* and *pathOrdinals* stacks. *pathNodes* keeps track of the path from the root of the current aggregate to the current lock. The *current*

Global field: *nextRootOrdinal*  
 Per-object fields: *owner, shared, writer, readers, childOrdinals, ordinal*  
 Per-thread fields: *locks, pathNodes, pathOrdinals*

```

Ordinal(o)  $\stackrel{\text{def}}{=}$  (if o.owner = null then o.rootOrdinal else o.owner.childOrdinals[o])
CanLock(o)  $\stackrel{\text{def}}{=}$ 
  0 < tid.txDepth  $\wedge$ 
  (if tid.pathNodes.IsEmpty then o.shared else o.owner = tid.pathNodes.Top)  $\wedge$ 
  tid.pathOrdinals.Top < Ordinal(o)

x := new C;  $\equiv$ 
x := o.f;  $\equiv$ 
o.f := x;  $\equiv$ 
  as in Fig. 3
connect p to o;  $\equiv$ 
disconnect p from o;  $\equiv$ 
share o;  $\equiv$ 
share(Fig. 3) o;
o.ordinal  $\leftarrow$  nextRootOrdinal++;

lockW o;  $\equiv$ 
lockW(Fig. 5) o;
tid.pathOrdinals.Top  $\leftarrow$  Ordinal(o);

lockR o;  $\equiv$ 
lockR(Fig. 5) o;
tid.pathOrdinals.Top  $\leftarrow$  Ordinal(o);
tid.pathNodes.Push(o);
tid.pathOrdinals.Push(- $\infty$ );

pop;  $\equiv$ 
assert  $\neg$ tid.pathNodes.IsEmpty;
tid.pathNodes.Pop();
tid.pathOrdinals.Pop();

transaction { S }  $\equiv$ 
if (tid.txDepth = 0)
  tid.pathOrdinals.Push(- $\infty$ );
tid.txDepth++;
S
tid.txDepth--;
if (tid.txDepth = 0)
  commit;
where
commit;  $\equiv$ 
tid.pathNodes.Clear();
tid.pathOrdinals.Clear();
commit(Fig. 5);

```

Figure 6: The deadlock prevention system

*lock* serves as a cursor for moving through the ownership tree; it may only move forward. *pathOrdinals* records the ordinal of each object in *pathNodes*, plus the current child ordinal of the current lock (which is the top element of *pathNodes*).

## 6 Genericity

To maximize the potential for fine-grained locking, we wish to enable fine-grained locking across abstraction boundaries. For example, an implementation of an abstract data type (ADT) may or may not choose to support fine-grained locking within its implementation. Client code should not have to be dependent on this decision. Therefore, the root object of the ADT should not be locked by the client, since this would prevent the ADT implementation from choosing whether to acquire a write lock or a read lock. Deferring the choice of lock to the callee enables genericity of clients.

A second dimension of genericity that we wish to provide is genericity of a module with respect to the context in which it is invoked. Consider again the case of an ADT. When a client calls into the ADT, the ADT may find itself in any of two contexts: either 1) it is part of an aggregate to which the current thread already has exclusive access, such as a thread-local aggregate or a sub-aggregate of an aggregate that has been locked for writing, or 2) the current thread does not yet hold a lock on the ADT. We achieve genericity of modules such as this ADT with respect to their context, by providing language constructs

that can be used in the implementation of a module to ensure that the module has access to a given object, regardless of whether the thread already holds a lock on the object. Specifically, we provide two block constructs:

<pre> <b>write</b> (<i>o</i>) { <i>S</i> } ≡   <b>if</b> (<i>o</i> ∉ <b>tid</b>.<i>Writable</i>)     <b>lockW</b><sub>(Fig. 6)</sub> <i>o</i>;   <i>S</i> </pre>	<pre> <b>write_indirectly</b> (<i>o</i>) { <i>S</i> } ≡   <b>if</b> (<i>o</i> ∉ <b>tid</b>.<i>Readable</i>)     <b>lockR</b><sub>(Fig. 6)</sub> <i>o</i>; <i>S</i> <b>pop</b><sub>(Fig. 6)</sub>;   <b>else</b>     <i>S</i> </pre>
--	---

## 7 Modularity

The system presented in the previous section supports programming generic modules that perform fine-grained locking of aggregate objects and guarantees data-race-freedom, deadlock-freedom, and serializability of transactions. However, the compliance rules of the system are not suitable for modular static verification. Specifically, the state space defined by the various fields introduced as part of the system is not modular; the state that is relevant to a given module is not separate from the state from which that module should abstract. Therefore, to arrive at the final programming methodology, in this section we perform a refactoring of the state space so that the state of each module is in separate fields.

The methodology is formalized in Figs. 7 and 8. The methodology introduces the following special fields. (**int**<sup>+</sup> denotes **int** ∪ {−∞, +∞}.)

<p>Per-object fields:</p> <pre> <b>volatile bool</b> <i>shared</i>; <b>int</b> <i>ordinal</i>; <b>Map</b>(<b>object</b>, <b>int</b>) <i>childOrdinals</i>; </pre> <p>Per-thread fields:</p> <pre> <b>int</b> <i>txDepth</i>; <b>Set</b>(<b>object</b>) <i>locks</i>; <b>int</b><sup>+</sup> <i>currentRootOrdinal</i>; <b>object</b> <i>currentRoot</i>; </pre>	<p>Global field:</p> <pre> <b>int</b> <i>nextRootOrdinal</i>; </pre> <p>Per-thread-and-object fields:</p> <pre> <b>int</b><sup>+</sup> <i>writeTokenCount</i>; <b>bool</b> <i>reading</i>; <b>bool</b> <i>writing</i>; <b>bool</b> <i>own</i>; <b>bool</b> <i>needsLocking</i>; <b>int</b><sup>+</sup> <i>currentChildOrdinal</i>; <b>object</b> <i>currentChild</i>; </pre>
---	--

**Claim 4** *The methodology allows writing generic modules that can be verified both statically using sequential Hoare-style reasoning and through run-time checking. Compliance with the methodology implies deadlock-freedom, and serializability of transactions.*

## 8 Discussion and Related Work

We have built our system on top of standard techniques for achieving fine-grained locking, deadlock-freedom, and serializability. What we believe is new is the way we integrate these techniques into a programming methodology that enables static verification of generic modules.

An important category of related work proposes the use of optimistic concurrency rather than locking to increase parallelism. Different systems make different trade-offs between the amount of programmer-supplied code annotation, the run-time efficiency, the safety guarantees provided, and other factors. Some systems (e.g. [1]) provide strong safety guarantees but require that the program be written in a pure functional language. Others require very few annotations but incur a heavy performance penalty. With the current work, we are exploring a design point where the system requires a non-trivial amount of programmer annotation, but in return offers strong safety guarantees and good performance in an object-oriented language.

```

o := new C; ≡
  o ← new C;
  o.writeTokenCount ← +∞;
  o.reading ← true;
  o.writing ← true;
  o.own ← true;
  o.shared ← false;

x := o.f; ≡
  assert o.reading;
  x ← o.f;

o.f := v; ≡
  assert o.writing;
  o.f ← v;

start_writing o; ≡
  assert o.writeTokenCount > 0;
  assert ¬o.reading ∧ ¬o.writing;
  if (o.needsLocking) {
    lockW0 o;
    tid.locks.Push(o);
    o.needsLocking ← false;
  }
  o.reading ← true;
  o.writing ← true;
  foreach (p ∈ rep(o)) {
    p.own ← true;
    p.writeTokenCount ← +∞;
  }

end_writing o; ≡
  assert o.writing;
  foreach (p ∈ rep(o)) {
    assert p.own;
    assert ¬p.reading ∧ ¬p.writing;
  }
  o.reading ← false;
  o.writing ← false;
  foreach (p ∈ rep(o)) {
    p.own ← false;
    p.writeTokenCount ← 0;
  }

share o; ≡
  assert o.own;
  assert ¬o.reading ∧ ¬o.writing;
  o.own ← false;
  o.writeTokenCount ← 0;
  o.shared ← true;
  o.ordinal ← nextRootOrdinal++;

acquire o; ≡
  assert 0 < tid.txDepth;
  assert tid.currentRoot = null;
  assert o.shared;
  assert tid.currentRootOrdinal < o.ordinal;
  tid.currentRoot ← o;
  lockW0 o;
  tid.locks.Push(o);
  o.writeTokenCount ← +∞;
  tid.currentRootOrdinal ← o.ordinal;

release o; ≡
  assert tid.currentRoot = o;
  assert ¬o.reading ∧ ¬o.writing;
  o.writeTokenCount ← 0;
  tid.currentRoot ← null;

acquire_finegrained o; ≡
  assert 0 < tid.txDepth;
  assert tid.currentRoot == null;
  assert o.shared;
  assert tid.currentRootOrdinal < o.ordinal;
  tid.currentRoot ← o;
  o.needsLocking ← true;
  o.writeTokenCount ← 1;
  tid.currentRootOrdinal ← o.ordinal;

release_finegrained o; ≡
  assert tid.currentRoot = o;
  assert ¬o.reading ∧ ¬o.writing;
  o.writeTokenCount ← 0;
  tid.currentRoot ← null;

transaction S; ≡
  if (tid.txDepth = 0)
    tid.currentRootOrdinal ← -∞;
  tid.txDepth++;
  S
  tid.txDepth--;
  if (tid.txDepth = 0)
    commit;

where
  commit; ≡
    while (o ← tid.locks.Pop())
      unlock0 o;

```

Figure 7: The programming methodology. (See also Fig. 8.) Vertical bars indicate code that may be ignored for static verification.

```

start_writing_indirectly  $o$ ;  $\equiv$ 
  assert  $o.writeTokenCount > 0$ ;
  assert  $\neg o.reading \wedge \neg o.writing$ ;
  | if ( $o.needsLocking \neq null$ ) {
    |   lockR0  $o$ ;
    |   tid.locks.Push( $o$ );
    | }
   $o.writeTokenCount --$ ;
   $o.reading \leftarrow true$ ;
   $o.currentChildOrdinal \leftarrow -\infty$ ;
   $o.currentChild \leftarrow null$ ;

end_writing_indirectly  $o$ ;  $\equiv$ 
  assert  $o.reading \wedge \neg o.writing$ ;
  assert  $o.currentChild == null$ ;
  foreach ( $p \in rep(o)$ ) {
    | assert  $\neg p.reading \wedge \neg p.writing$ ;
    | }
   $o.reading \leftarrow false$ ;
  foreach ( $p \in rep(o)$ ) {
    |  $p.writeTokenCount \leftarrow 0$ ;
    | }

open_child( $o, p$ );  $\equiv$ 
  assert  $o.reading \wedge \neg o.writing$ ;
  assert  $o.currentChild == null$ ;
  assert  $p \in rep(o)$ ;
  assert  $o.currentChildOrdinal < o.childOrdinals[p]$ ;
   $o.currentChild \leftarrow p$ ;
   $o.currentChildOrdinal \leftarrow o.childOrdinals[p]$ ;
   $p.writeTokenCount \leftarrow 1$ ;
  |  $p.needsLocking \leftarrow o.needsLocking$ ;

close_child( $o, p$ );  $\equiv$ 
  assert  $o.reading \wedge \neg o.writing$ ;
  assert  $o.currentChild == p$ ;
  assert  $\neg p.reading \wedge \neg p.writing$ ;
   $p.writeTokenCount \leftarrow 0$ ;
   $o.currentChild \leftarrow null$ ;

lock_child( $o, p$ );  $\equiv$ 
  assert  $o.reading \wedge \neg o.writing$ ;
  assert  $o.currentChild == null$ ;
  assert  $p \in rep(o)$ ;
  assert  $o.currentChildOrdinal < o.childOrdinals[p]$ ;
   $o.currentChildOrdinal \leftarrow o.childOrdinals[p]$ ;
  | if ( $o.needsLocking$ ) {
    |   lockW0  $o$ ;
    |   tid.locks.Push( $p$ );
    | }
   $p.writeTokenCount \leftarrow +\infty$ ;

```

Figure 8: The programming methodology. (See also Fig. 7.) Vertical bars indicate code that may be ignored for static verification.

## 9 Conclusion

We presented a preliminary result in our investigation of programming methodologies for safe and efficient concurrency in object-oriented languages. The presented methodology enables modular static verification as well as run-time checking of the deadlock-freedom and serializability of object-oriented modules that perform fine-grained locking and that are generic with respect to the locking behavior of their callers and their callees. We are currently proving the claims, evaluating the system, and studying related work.

## References

- [1] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.
- [2] Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In Bernhard Aichernig and Bernhard Beckert, editors, *Proc. 3rd IEEE Int. Conf. Software Engineering and Formal Methods*. IEEE Computer Society Press, 2005. To appear.