Deadlock Freedom Through Object Ownership

Eric Kerfoot
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, UK
eric.kerfoot@comlab.ox.ac.uk

Steve McKeever
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, UK
steve.mckeever@comlab.ox.ac.uk

Faraz Torshizi*
Department of Computer Science, University of Toronto
10 King's College Road
Toronto, Ontario, Canada
faraz@cs.toronto.edu

ABSTRACT

Active objects are an attractive method of introducing concurrency into Java-like languages by decoupling method execution from invocation. In this paper, we show how ownership is used in the Java [14] subset language CoJava [17] to prevent deadlock associated with active object method calls. This approach builds on existing type-based approaches that eliminates data races and data-based deadlock in concurrent systems. The novel addition is the use of ownership to organize active objects, thus preventing deadlock from arising when objects are allowed to block awaiting responses from others.

Typechecking is used to prevent threads from sharing mutable data, thus CoJava is free of data races and data-based deadlock. Behavioural deadlock is prevented by the use of promise objects which prevent clients from blocking indefinitely while awaiting responses. Ownership imposes a hierarchy on active objects; this allows owners to safely block while waiting for responses from owned objects. The paper also discusses the implications of this approach to specification with JML, formal reasoning about programs, and the consequences to runtime assertion checking.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages;
D.3.3 [**Language Constructs and Features**]: Concurrent programming structures—*Classes and objects*;
F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification techniques

Keywords

CoJava, Java, Ownership, Active Objects, Concurrency, Deadlock, Data Races

*Part of this work was conducted under an NSERC scholarship

IWACO '09, July 6 2009, Genova, Italy

1. INTRODUCTION

The Active Object Design Pattern [18] describes a simple method of introducing concurrency in object-oriented languages. It decouples method execution from method invocation, thereby introducing concurrency though asynchronous method invocation. This contrasts with sequential, passive objects where executing immediately follows invocation. However it does not focus on issues of concurrency correctness directly, specifically those of data races and deadlock. Type-based approaches have been demonstrated to effectively tackle data race issues, but how this can be used to prevent deadlock is an open question.

```
class A { B b;
  void m1() { b.m3(this);}
  void m2() { ... }
}
class B { void m3(A a) { a.m2();} }
```

Figure 1: Deadlock Situation

Consider the situation when two active objects send requests to one another. Typically an active object cannot respond while it deals with a request. Thus if two objects are waiting for responses from one another, deadlock has occurred. Figure 1 sketches such a situation where classes `A` and `B` are active object types. While executing `m1()`, the call to `m3()` causes `m2()` to be called. The object cannot respond when `m2()` is called, thus both objects wait for one another indefinitely. Deadlock in this situation is a result of method re-entry.

This paper describes an active object-based approach used in CoJava [17]. This work builds on the type-based approach to data race-freedom, but uses ownership to organize active objects into hierarchies. The purpose of using ownership is to prevent circular relationships where two or more active objects wait indefinitely for one another. CoJava uses JML-like [19] annotations to declare active object types as these can be applied to existing Java types.

CoJava addresses the deadlock problem in two ways to prevent blocking in re-entrant calls:

- Active objects unrelated through ownership must communicate in a way that does not block. This is achieved through the use of promise objects [21] that act as receptacles for return values. A client may thus defer collecting results until a later time rather than blocking. The client may wait to get a response for a time

period, if no result comes within this period, a timeout event occurs.

- Ownership is used to organize objects into hierarchies. An owner may safely block waiting for responses from the objects it owns, so it need not use the promise object approach. Owned active objects also imply that fewer objects may communicate with it, affording greater understanding and control over their runtime behaviour. This will be shown to have implications for specification and formal reasoning.

The CoJava concurrency methodology focuses on correct active objects while introducing as few new concepts as possible. Each active object will have one thread associated with it and so guarantees safety at the expense of efficiency. The methodology uses tool-generated proxies to implement active objects. This tool performs necessary type checks to ensure that mutable data is never shared between active objects, hence no locks are used.

The paper is organized as follows: Section 2 describes the CoJava language in brief, including the ownership type system. Section 3 describes the active object approach used by CoJava and the type requirements necessary to prevent the sharing of mutable state. This includes the concept of admissibility and immutable types. Section 4 discusses the relationship between active objects and ownership, and how hierarchy and encapsulation organize threads into cooperative units. This section also includes discussion on the implications for sequential reasoning and runtime assertion checking.

2. COJAVA

This section briefly introduces CoJava [17], a subset of the Java language with formal type and operational rules. CoJava is a framework towards object ownership and simple concurrency in Java. CoJava is intended to be smaller than full Java so that its formal semantic description can be kept concise. This makes the semantics for ownership and concurrency much more manageable and with less need to consider complex features of Java. The CoJava Tool operates as a type checker to enforce non-Java type rules and generates output Java code from CoJava input.

A subset of JML is used to define specifications in CoJava, which are presented as annotations embedded in regular Java comments beginning with `//@` or `/*@`. The subset of annotations used to specify CoJava include those for contracts and type annotations. The full CoJava grammar is shown in Figure 2, and the tool with documentation including formal rules can be found at
http://devel.softeng.ox.ac.uk/cojava.

2.1 Ownership

CoJava's transitive ownership [2, 8, 24] type system uses a type modifier represented as the JML-like annotation `/*@ owned @*/`. This is applied to the types of attributes and variables to indicate that they reference objects *owned* by the current object. Not all objects have owners, unlike other schemes where some owner (such as a top-level owner) must be specified. This allows Java code written without ownership annotations to be easily integrated with CoJava code.

Type	::=	**int** | **char** | **boolean** | C
Defs	::=	(*Class* | *Inter*)*
Class	::=	**class** C [**extends** D] [**implements** $\overline{I}$] { *Attr** *Constr* *Meth** }
Inter	::=	**interface** C [**extends** $\overline{I}$] { *IMeth** }
Constr	::=	**public** C($\overline{Type\ x}$) { [**super**($\overline{E}$);] *Stmt** }
Attr	::=	*Mod Type* `x ;`
Meth	::=	*Mod Type* `m`($\overline{Type\ x}$) { *Stmt** **return** *Exp*;}
		| *Mod* **void** `m`($\overline{Type\ x}$) { *Stmt** }
IMeth	::=	**public** (*Type* | **void**) `m`($\overline{Type\ x}$);
Mod	::=	**public** | **private** | **protected**
Stmt	::=	*Type* `x` [`=` *Exp*]`;` | *Name* `=` *Exp*`;`
		| *Call*`;` | `;` | { *Stmt** } | **while**(*Exp*) *Stmt*
		| **if**(*Exp*) *Stmt* **else** *Stmt*
		| **for**(*Type* `x :` *Exp*) *Stmt*
		| **for**(*Type* `x =` *Exp* `;` *Exp* `; x =` *Exp*) *Stmt*
Name	::=	**this**`.y` | **this**`.x.y` | `x.y` | `x`
Call	::=	**this**`.m`($\overline{Exp}$) | **this**`.x.m`($\overline{Exp}$)
		| `x.m`($\overline{Exp}$) | **super**`.m`($\overline{Exp}$)
Exp	::=	**null** | *lit* | *Name* | *Call* | (*Type*) *Exp* | (*Exp*)
		| **new** `C`($\overline{Exp}$) | `-` *Exp* | `!` *Exp* | *Exp Op Exp*
Op	::=	`+` | `-` | `*` | `/` | `%` | `&&` | `||` | `==` | `!=` | `<` | `>` | `<=` | `>=`

Figure 2: The CoJava Grammar

If an object `a` instantiates an object `b` with the owned annotation included in the **new** expression, then `a` is the *immediate owner* of `b`. This is purely defined as a property of the type and does not affect what class defines `b`. If `b` owns some object `c`, then `a` also owns `c` transitively.

An owned *type* has the aforementioned annotation before the type name, thus the owned version of type `T` is `/*@ owned @*/ T`. A variable (eg. a local variable, argument, or attribute) with an owned type is called an owned variable. An object referenced by such a variable is called an owned object; by virtue of the type it is owned by the current object.

The following is a set of properties of the CoJava type system that the tool enforces:

- Owned values cannot be assigned to non-owned variables and attributes, or vice versa. This ensures that an owned object can only be aliased through an owned reference.
- Methods with owned arguments can be called, and owned attributes assigned to, only when the receiver is **this**. This prevents any client from changing the relationship between owned objects, especially from breaking the hierarchical structure.
- Methods returning owned references and owned attributes can only be accessed through an owned receiver. This prevents non-owners from accessing critical internal state.
- Within the method bodies of some class `T`, **this** has type `/*@ owned @*/ T`. Since non-owing clients cannot call methods returning owned values, **this** is not accessible to non-owners.

As a result, the following properties are guaranteed by the type system for well-typed CoJava programs:

- Objects cannot be aliased by owned and non-owned references at once (excluding **this**).
- Owned objects are organized into tree structures encapsulated by their owners.
- If an object is aliased through a non-owned reference, its owned objects are inaccessible to its clients since no method call or attribute access makes them available.
- Since methods accepting owned arguments and owned attributes can only be accessed by **this**, clients cannot pass owned references to an object.
- Owners can access the owned objects transitively, but they cannot modify the object structures created by those objects it owns.

For example, consider an object `list` which contains a linked list of objects. Each node in the list owns the next node and `list` owns the first object called `head`. The `list` object can access any of the nodes but it cannot insert a node into the list; it must call a method of the nodes to do this. If `list` is aliased through a non-owned reference, then no client can access these nodes.

Ownership enforces the encapsulation of `head` and every other node by `list`. In this case, it also enforces structural correctness since no node in the chain may alias any nodes above it, creating a loop. If every method of `list` returning an owned object and every owned attribute were private, then even owners of `list` cannot access the nodes, and so an even stronger encapsulation property (denoted by the **contained** class modifier) is enforced.

This section gave a brief definition of CoJava, in particular its ownership type system. The next section describes how active objects are implemented and what requirements are needed for their type system.

3. ACTIVE OBJECT CONCURRENCY

CoJava active objects are defined with the **threaded** JML-like annotation, which can be applied to class definitions or variable and attribute type declarations. An active (or threaded) object is in fact a proxy object generated by the CoJava Tool. The methods of this proxy have a similar signature to those found in the object they encapsulate. These proxies have their own type, such that the active version of type `T` is `/*@ threaded @*/ T`. Proxies can be generated for existing class types without custom code or annotations, therefore compatibility with existing Java code is provided within the bounds of the admissibility criteria discussed in Section 3.2. Immutable types that can be safely shared between threads are defined Section 3.3.

CoJava prevents data races by restricting what types can be shared between active objects, thus it is lock-free. Excluding locks also prevents deadlock when multiple threads wait to lock the same data. Less waiting occurs without locks, thus concurrency methodologies that exclude them offer efficiency advantages [12].

Ownership itself is not used to provide these benefits in CoJava. Owned types can determine which objects may be safely shared or transferred between threads, or which need to be locked, as demonstrated in other concurrency research efforts [5, 6, 7, 10]. For example, a locking regime may state that when a lock on an owner is acquired, every object it owns is also locked. Ownership may also be used to designate owning threads for every object, such that only objects owned by a thread may be modified during the course of its execution. Transferring ownership between threads would also facilitate sharing in this scheme.

These ownership approaches contrast with verification approaches [20, 16]. The goal is to prove that data race and deadlock freedom result from correct synchronization schemes, rather than deriving these as properties of the type system.

Guava [3] however uses special types to implement a form of active objects, value semantics, and unique references [1] as well as ownership. Kilim [27] uses special message types that are unencapsulated values and have at most one owner at any given time. The purpose of these techniques is to prevent mutable data sharing between active objects. CoJava accomplishes this through the admissibility criteria discussed below.

The threaded proxy classes operate by placing a message on an internal queue whenever one of its methods is called. The methods of the proxy return promise objects of type `Result`. A promise object functions as a receptacle that eventually will store the return value for the call once it is calculated. `Result` also includes methods to test whether timeout or error events have occurred. This operates in a similar manner to the active object implementation in the JAC [22] language.

The sent message contains the arguments passed with the call, and will be eventually processed by the internal thread. The thread will call the actual method on an internal instance of the base type, and place any possible results in the promise object. Only one message is processed at once, thus no more than one method is ever called at any one time on the local object.

The use of promise objects implies that clients do not block waiting for the methods of threaded objects to be executed, but may defer collecting results to a later time. The promise objects allow the client to wait for a specified finite time period before a timeout event is considered to have occurred. Such events indicate a situation that would otherwise result in deadlock, where two or more threaded objects are waiting for each other to respond. We illustrate this concurrency approach with a producer-consumer example that demonstrates communication between threaded objects.

3.1 Example

Figure 3 sketches a simple producer-consumer example using a threaded class `StringQueue`. `Producer` sends a `String` to the queue by calling the `add()` method of `q`, which is in fact a threaded proxy. The message for the call is placed on `q`'s message queue and will eventually be executed.

`Consumer` requests an item from the queue through a call to the method `get()`, which returns a `Result` object `r` instead of the actual result. When the queue's thread executes the actual call, the value will be placed in `r` which can be retrieved through the `objectResult()` method. The argument to this call is the timeout value in milliseconds, if the caller waits longer than the given time then the call returns **null** and `r`'s method `hasTimedOut()` will return **true**.

The type `/*@ threaded @*/ Producer` represents the

proxy type. An instance of this type contains a private queue of messages, a thread processing the messages, and an instance of `Producer` (the delegate object) whose methods the thread calls in the course of processing the messages. The delegate object's JML contracts are checked at runtime in the same way that a regular local object's contracts would be checked. If a violation occurs this is captured by the `Result` object.

```
class Producer {
  public void produce(StringQueue q) {
    for(int c=0;true;c=c+1)
      q.add(""+c);
  }
}
class Consumer {
  public void consume(StringQueue q) {
    while(true){
      Result r=q.get();
      String i=(String)r.objectResult(100);

      if(r.hasTimedOut()) // handle timeout
      else if(r.isError()) // handle error
      else  .. // consume i
    }
  }
}
...
StringQueue i=new StringQueue(10);
/*@ threaded @*/ Producer p=
    new /*@ threaded @*/ Producer();
/*@ threaded @*/ Consumer c=
    new /*@ threaded @*/ Consumer();
p.produce(i); c.consume(i);
```

Figure 3: The Producer Consumer Example

3.2 Admissibility and Data-race Freedom

CoJava threaded (active) objects prevent data races by not sharing mutable data. This is accomplished by restricting what types of objects may be passed through a threaded object's public interface, which in effect is the thread boundary separating one thread from all others.

Consequently, mutex locks or other means of access control are not needed in CoJava. Implementations are thus simplified since code to provide synchronization is not needed. It also implies that there will be no waiting by one object for a shared object to be released so that it can be locked, hence the performance of concurrent code will be enhanced. This comes at the cost of restricting what types of objects can be transmitted between threaded objects, and requires time and space to serialize objects implementing the `StringSerializable` interface.

Rather than using ownership or reference uniqueness, a simple concept of admissibility is employed:

- An *admissible type* is a primitive type, an immutable object type, a threaded object type, or a subtype of the interface `StringSerializable` which effects cloning by converting objects to and from `String` representations.
- An *admissible method* is one whose argument types and return type are all admissible types.
- An *admissible constructor* is one whose argument types are admissible types.
- An *admissible attribute* is one with admissible type.

An object whose type is admissible can be safely passed over a thread boundary since it is either safe to share with multiple threads or can be cloned. The CoJava Tool will generate methods in the threaded proxy class only for public admissible methods of the original type `T`. Accessor and mutator methods are generated only for public admissible attributes. There must be an admissible constructor for instances of this threaded type to exist.

Objects implementing `StringSerializable` are turned into `String` objects by the proxy, then converted back to objects before the actual call occurs. This mechanism is used for simplicity in CoJava, which cannot make use of Java's cloning mechanism since exceptions are not included in the language.

Admissibility allows CoJava threaded objects to share pre-existing object types. The Java types `String`, `Integer`, and others are known to be ostensibly immutable and so the CoJava tool is aware of this through model definitions.

Threaded versions of existing types can also be instantiated, assuming they have admissible members and that they can be imported into the CoJava environment through model type definitions. The CoJava concurrency approach is based on types and not introduced custom programming constructs. Therefore Java library types such as `String` can have threaded instances of type `/*@ threaded @*/ String`, which can only make available the methods of `String` that are admissible.

3.3 Immutable Types

Often an object type is defined whose instances do not change state over their lifetimes. These immutable objects, such as Java's `String`, can be safely shared by threaded objects since data races are only a product of mutable state. JAC [22], for example, employs immutability to allow such safe sharing between threads.

Immutability is enforced through compiler checks in addition to ownership types, an approach similar to other immutable object schemes [4, 15]. The tool enforces the following requirements on classes declared as immutable:

- All public constructors and methods must be pure.
- All methods with owned arguments or return types must be non-public.
- All attributes must be non-public.
- All object attribute must be owned or immutable.
- Immutable types can only extend `Object` or other immutable types.
- No mutable type can extend or implement an immutable type.

3.4 Wait Conditions

Classically, a precondition must be established before a method is called. It is often useful to state a precondition that will eventually be true when a method of an active object is called. A wait condition is a special precondition that is not required to be true when the message for the method is sent, but is expected to eventually be fulfilled. In the meantime, if the condition is not met, then the message can

be temporarily skipped. Wait conditions are translated into regular preconditions as well, such that the passive instances of types with the conditions will still correctly check method requirements.

```
/*@ threaded @*/ class StringQueue {
public /*@ owned @*/ ArrayList items;

//@ wait !isFull();
//@ ensures items.size()==
//@   \old(items.size())+1 &&
//@   items.get(\old(items.size()))==i;
public void add(String i){ items.add(i); }

//@ wait size()>0;
//@ ensures items.size()==
//@   \old(items.size())-1 &&
//@   \result== \old(items.get(0));
public String get() {
  String s=(String)items.get(0);
  items.remove(0); return s;
}
...
}
```

Figure 4: The StringQueue class

Figure 4 describes two methods from the `StringQueue` class that use wait conditions, specified with the **wait** keyword. The method `add()` requires that the queue not be full when it attempts to add an item. If the wait condition is not satisfied, the clients wait for other threads to remove elements from the queue. Once the wait condition is met, the call is performed when the delayed message is next processed by the internal thread.

Wait conditions provide an atomic means of checking for a condition. If the wait condition for `add()` were a regular precondition, a client of the queue must instead check whether `isFull()` was true or not before making the call. Between this check and the actual call, another client might call `add()` and cause the queue to become full again, thus precipitating a precondition violation. By using a wait condition, there is no opportunity for this to occur between the condition being fulfilled and the actual call being executed.

This section outlined the active object approach used in CoJava. Active objects require proxies generated by the CoJava tool, and type checks ensure mutable state is not shared between them. The next section discusses implications of using ownership with active objects, such as how this affects sequential reasoning and runtime assertion checking.

4. CONCURRENT OWNERSHIP

This section discusses the application of ownership to active objects, specifically how it prevents deadlock from occurring when objects block waiting for results from threaded method calls. Since ownership organizes objects into hierarchies, calls that block may only be performed by owners when calling methods of owned objects. This ensures that blocking calls propagate down the hierarchy and do not form loops of blocking calls.

The purpose of the promise object type `Result` is to allow clients to wait for return values for a finite amount of time, i.e., they do not block, preventing deadlock. This was considered in Figure 1 that roughly sketched the situation where method re-entrancy between threaded objects resulted in deadlock.

Timeouts allow a program to progress when a deadlock situation occurs and to report the error. However, looping indefinitely until a call succeeds without timeout translates deadlock into livelock. There exists as yet no means of statically detecting this in CoJava, or knowing when a call will result in a timeout event.

Deadlock is a product of mutual aliasing where two or more objects are involved in a reciprocal calling situation. The CoJava ownership type system enforces hierarchy on objects, thus this relationship does not occur between an owned object and its owner.

An owner may safely block waiting for a response from objects it owns, since the hierarchy guarantees that the owned object will not be allowed to block indefinitely waiting for a response from its owner. Assuming that the method called by the owner does not loop indefinitely when timeout occurs (a form of divergence), the method will eventually return.

```
/*@ threaded @*/ class A {
  public /*@ owned @*/ B b; ...
  public boolean m1(A a)
    { b.m4(a); return b.m3(); }
  public boolean m2() { ... }
}
/*@ threaded @*/ class B { ...
  public boolean m3() { ... }
  public void m4(A a)
    { Result r=a.m2(); ... }
}
... A a = new A(); a.m1(a);
```

Figure 5: Threaded Ownership Example

Figure 5 illustrates calling a method of a threaded owned object directly. When `m3()` is called, it returns a **boolean** value rather than a promise object. The caller object `a` will block until this call completes with the expectation that `b` cannot initiate a blocking call back to `a` since it is owned.

However, when `m4()` is called, a reference value which happens to point to the same object as the caller is passed as an argument. This allows `b` to call a method of `a`, but will have to use the promise object and will always encounter a timeout. This is an instance where an owned object may acquire access to its owner through a regular reference. Assuming that `m4()` does not loop forever until the call succeeds, it will return some default value after (hopefully) reporting the timeout error. The `Result` type includes methods for querying if a timeout or error has occurred when waiting for a response, thus timeout events are treated as programming errors that can be reported and which allow the program to continue execution.

Although a timeout will occur when `m4()` is called, this does not result in deadlock, although one object has been allowed to block waiting for another. *The relationship between owner and ownee is asymmetrical and therefore can be used to ensure that only one party in a concurrent call sequence may block indefinitely awaiting a response.*

4.1 Specification and Sequential Reasoning

The previously discussed active object implementations do not discuss specification in significant depth. This con-

trasts with the SCOOP [9, 13, 25] language that extends Eiffel [23] with object-centered concurrency. A prototype Java version of the SCOOP model, called JSCOOP, is also available [28]. The SCOOP model uses locking semantics and a formal computational model [26] to determine what locks are required and when. The design-by-contract approach fundamental to Eiffel can thus be applied to active objects. This is, however, at the expense of complex semantics and compiler implementations.

CoJava owned threaded objects can be used in specifications in a seamless manner. Regular threaded objects require the use of the promise object and hence are very cumbersome in contracts, and a timeout event has undefined logical meaning. Most importantly, because ownership restricts who may alias an owned object, there is greater control over what messages it receives and when. This leads to greater understanding of semantics and a guarantee that properties established by contracts will hold.

When reasoning about passive objects, it is expected that a property established about an object will hold until another operation is performed on it. This cannot be established locally for threaded objects since after such a property is established, the threaded object might receive a message that invalidates it.

Ownership restricts when this happens by controlling who may send messages to a threaded object. Methods may be defined assuming that their type's instances have sole access to threaded objects. However, if other owners exist, it can be assumed that they are careful about mutating transitively owned objects. The same encapsulation is beneficial for passive objects therefore it serves a very useful function for threaded objects as well.

```
/*@ owned @*/ StringQueue queue;...
queue.add("Hello");
// assert queue.size()>0
queue.get();
```

Figure 6: StringQueue Example

Figure 6 illustrates a use of the `StringQueue` class, where between the two method calls the property that the queue is non-empty is assumed to hold. Messages sent to threaded objects are processed in order. A property established by a method will therefore hold when subsequent messages are processed. Once the message for `get()` is processed, the queue will be storing the given item and the so precondition will certainly be met.

```
//@ ensures q.contains(s);
public void addString(StringQueue q,
        String s){
  q.add(s);
}
```

Figure 7: Add Example

This block of code assumes that either the current object is the sole owner of `queue`, or that any other owners are responsible for invoking the method containing this code in a safe manner that does not interfere with `queue`. Ensuring that an owned threaded object is not accessible to transitive owners can be done by annotating the owning class with the **contained** keyword. However, often a class may want its owners to access its owned state in a safe manner, which demands careful co-operation.

The queue exists in a separate thread context and performs operations asynchronously, but since messages are processed in order, sequential reasoning can still be applied. This also extends to contracts, as Figure 7 illustrates. The method `addString()` may exit before the call `add()` is actually processed by `q`, therefore the postcondition isn't true just yet. However, if a client of `q` attempted to call its methods to query whether `s` was indeed contained by it, the calls would not be processed until the completion of the original `add()` call. Although in this case the postcondition is yet to be established, it will be observably true to external clients after `addString()` completes.

Wait conditions may introduce the possibility of deadlock in CoJava, with or without ownership. If a wait condition for a method is always false, then any client calling that method will always encounter a timeout event, or deadlock if it is an owner. In Figure 6 an owned `StringQueue` is given an item by calling `add()`. If the queue was already full and there was only the one owner, then deadlock will result. Since no other objects can remove items from the queue, the wait condition for that method call can never be fulfilled.

As a result, owners must treat wait conditions as regular preconditions since there is no guarantee that other owning clients exist that may cause the condition to be fulfilled. To support this, the generated proxies are aware of what messages are sent as part of owned object calls and will not wait when these are processed. Since the wait conditions will still be treated like preconditions, if runtime assertion checking is being used then the condition will be checked.

4.2 Barbershop Example

A larger example is introduced that illustrates how ownership is used to co-ordinate active objects as co-operating threads. Figure 8 presents the CoJava implementation of a barber class for the Sleeping Barber concurrency example [11]. The barber contains a queue of customers with a capacity of two, who are represented by their names in `String` form. Although the queue is owned, it is public and thus accessible to owners of barbers, which are expected to add names to the queue and call `wakeUp()` to cause the barber to cut hair. By making the queue owned, this allows only owners to add items to it.

The owner of a `Barber` instance would be the shop it works in, illustrated in Figure 9. The `BarberShop` class is declared as **contained** and so even owners of its instances cannot access its `Barber` object. The `Barber` therefore provides a service exclusively to its owner. The relationship between the shop and the barber is similar to the producer-consumer one previously discussed, where the shop produces clients while the barber consumes. The shop and the barber can operate asynchronously since the queue of customers is itself a threaded type.

Multiple calls might cause a precondition violation to occur if the queue was empty when the call is eventually made. The `BarberShop` is thus responsible for calling the methods of `Barber` only at safe times, and must use objects it owns transitively in a safe manner that does not adversely affect the operation of other owners.

The method `getHaircut()` does not rely on the wait con-

```
class Barber {
  public /*@ owned @*/ StringQueue queue;

  public Barber() {
    queue=new /*@ owned @*/ StringQueue(2);
  }

  //@ ensures queue.size()==0;
  public void wakeUp() {
    while(queue.size()>0)
       cutHair();
  }

  //@ requires queue.size()>0;
  //@ ensures
  //@   queue.size()==\old(queue.size())-1;
  public void cutHair() {
    String c=queue.get();
    // cut c's hair
  }
}
```

Figure 8: The Barber Class

```
/*@ contained @*/ class BarberShop {
private /*@ owned @*/ StringQueue queue;
private /*@ owned threaded @*/
    Barber barber;
  //@ invariant queue==barber.queue;

  public BarberShop() {
    barber=
      new /*@ owned threaded @*/ Barber();
    queue=barber.queue;
  }

  public void getHaircut(String customer) {
    if(queue.isFull())
      // customer is discarded
    else {
      queue.add(customer);
      barber.wakeUp();
    }
  }
}
```

Figure 9: The BarberShop Class

dition for `add()` but instead explicitly checks to see if the queue is full. Since the queue is owned, the wait condition is treated as a regular precondition, therefore this check is necessary. Because the `BarberShop` is contained it is known that no object it owns is accessible to external clients, owned or otherwise. Combined with the knowledge that only the `Barber` object can alias the queue and it never adds items, no object will add an item to the queue in the time between the `isFull()` check and the call to `add()`. If this structural assumption were not true, a client might cause the queue to be full and thus cause a precondition violation to occur.

This section has discussed ownership and its implication for the organization of active objects. By co-ordinating how active objects interact, this allows CoJava to formally co-ordinate threads of control in a data race-free, deadlock-free manner. Custom runtime assertion checking code prevents the evaluation of JML contracts from introducing deadlock when the actual body of the relevant methods do not. The Barbershop example illustrates how one owned threaded object is used as a service provider exclusively by one owner, thus eliminating any chance of unintended interaction with other threads of control.

5. CONCLUSION

CoJava has thus been presented as a very small language implementing a complex concurrency design pattern. Simple type-based techniques statically ensure that well-typed CoJava programs are data race-free, and deadlock is prevented by the use of promise objects or ownership.

A number of issues remain for subjects of further research. One that has not been discussed here is the use of wait conditions in place of preconditions for methods. A wait condition causes the message for a method call to be temporarily skipped in the queue if the associated condition for it was not met. This is designed to allow subsequent messages to be processed in the hopes that the condition will be met and the call can continue. A wait condition that cannot be met implies that the message for that method is never processed. If the receiver is not owned, then this may not be catastrophic although the original caller will always get a timeout from the associated promise object. A greater understanding of what is feasible for wait conditions is needed on the part of the programmer, or greater understanding on the tool side to disallow conditions that produce endless waiting.

Every threaded object in CoJava has one thread for processing messages, and might have many more when performing runtime assertion checking. This is very inefficient although correct. A more ideal solution would involve a pool of threads which are not attached to any particular object, but process messages for threaded objects as needed. This implies that for efficiency reasons there would likely be more threaded objects than threads, since threaded objects might potentially spend a lot of time being idle and thus not require dedicated threads.

The CoJava Tool was implemented as an experimental type checker and code generator. It is not suitable for use as a real-world development tool, neither is CoJava a practical language for such work. Many features of Java, not least of all generics and exceptions, are absent in CoJava and cannot be correctly handled by the tool. Future work on the tool would focus on extending it to accept the entirety of the Java language, and address the existing rough areas of the current implementation.

CoJava itself suffers from a profusion of keywords and annotations. Though many are necessary, a simpler language is an attractive goal as it places lesser burden on the programmer. The ownership type system was designed for simplicity in contrast to other schemes, but has a number of drawbacks. An owner of a linked list, for example, cannot insert a node in the middle, instead complicated shuffling of data is necessary. The shortcomings of simplicity are being addressed as CoJava and its tool are being developed.

Active objects have been a topic of much work and progress has been made towards feasible, flexible, and highly useful means of introducing concurrency into Java-like languages. CoJava's focus has been on static correctness through the use of type systems. This paper has demonstrated how ownership can be leveraged to prevent deadlock in cases when one object blocks waiting for another, and how it can be

used to organize threads in more coherent and robust ways.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding, 2002.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32, 1997.

[3] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 382–400. ACM Press, 2000.

[4] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *In OOPSLA*, pages 35–49. ACM Press, 2004.

[5] C. Boyapati, R. Lee, and M. 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*, volume 37, pages 211–230, New York, NY, USA, November 2002. ACM Press.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 56–69, New York, NY, USA, 2001. ACM.

[7] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, Oct. 18–22 1998. ACM Press.

[9] M. Compton. SCOOP: an Investigation of Concurrency in Eiffel. Master's thesis, Department of Computer Science, The Australian National University, 2000.

[10] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, August 2007.

[11] E. W. Dijkstra. Cooperating sequential processes, technical report EWD-123. 1965.

[12] W. B. Easton. Process synchronization without long-term interlock. In *SOSP '71: Proceedings of the third ACM symposium on Operating systems principles*, pages 95–100, New York, NY, USA, 1971. ACM.

[13] O. Fuks, J. S. Ostroff, and R. F. Paige. SECG: The SCOOP-to-Eiffel Code Generator. *JOT Journal of Object Technology*, 11(3), 2004.

[14] J. Gosling et al. *The Java Language Specification*. GO-TOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan, 1996.

[15] C. Haack, E. Poll, J. Schï£¡fer, and A. Schubert. Immutable objects for a java-like language. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2007.

[16] B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A programming model for concurrent object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 31(1):1–48, 2008.

[17] E. Kerfoot and S. McKeever. Maintaining invariants through object coupling mechanisms. In T. Wrigstad, editor, *3rd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), in conjunction with ECOOP 2007*, Berlin, Germany, July 2007.

[18] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[19] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[20] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2009.

[21] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM.

[22] K.-P. Löhr and M. Haustein. The JAC system: Minimizing the differences between concurrent and sequential java code. *Journal of Object Technology*, 5(7), 2006.

[23] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[24] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[25] P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming 17061*. PhD thesis, ETH Zurich, 2007.

[26] J. S. Ostroff, F. A. Torshizi, H. F. Huang, and B. Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing*, 10.1007/s00165-008-0073-8, 2008.

[27] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *European Conference on Object Oriented Programming ECOOP 2008*, 2008.

[28] F. Torshizi, J. S. Ostroff, R. F. Paige, K. J. Doyle, and J. Lau. Jscoop: A high-level concurrency framework for java. Technical Report CSE-2008-09, York University, 2008.